Programming in C UNIX System Calls and Subroutines using C,

Substantially Updated March 1999

NetGuide Gold Site

Next Up Previous
Next: Copyright

Search for Keywords in C Notes

Keyword Searcher

Download Postscript Version of Notes

Click Here to Download Course Notes. **Local Students Only.**

Algorithm Animations

Direct link to Java Algorithm Animations (C related)

C COURSEWARE

Lecture notes + integrated exercises, solutions and marking

Online Marking of C Programs --- CEILIDH

Dave Marshall
29/3/1999

Copyright

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: About This Course **Up:** Programming in C **Previous:** Copyright

Books

- Brian W Kernighan and Dennis M Ritchie, The C Programming Language 2nd Ed, Prentice-Hall, 1988.
- Kenneth E. Martin, C Through UNIX, WCB Group, 1992.
- Keith Tizzard, C for Professional Programmers, Ellis Horwood, 1986.
- Chris Carter, Structured Programming into ANSI C, Pittman, 1991.
- C. Charlton, P. Leng and Janet Little, A Course on C, McGraw Hill, 1992.
- G. Bronson and S. Menconi, A First Book on C: Fundamentals of C Programming (2nd ed.), West Publishing, 1991.
- Any book on ANSI C will probably do, some UNIX may help.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Course Material and On-line facilities **Up:** Programming in C **Previous:** Books

About This Course

This course aims to teach a sound basis of C PROGRAMMING.

We will start with basic ideas and hopefully extend these to include some advanced features of C. We will particularly look at how C uses pointers, references low level memory and bytes and how it interfaces with the operating system.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Exercises - Using X WindowsEditing and **Up:** About This Course **Previous:** About This Course

Course Material and On-line facilities

Obviously you have been provided with the course notes that you are reading.

In addition several on line facilities will be employed in this course.

- **Ceilidh** - an online tutoring and program marking facility (see Appendix for details. All exercises given can be answered in Ceilidh. Some alternative C course notes are also available. Ceilidh will mark any exercise submitted very quickly.
- All program listings are available in the `/well/dave/C/EXAMPLES` directory. Feel free to copy these to help speed up your writing of programs. Mind Ceilidh helps with this also by providing skeleton programs.
- The course notes are also on-line. Run the `mosaic` program and select `comma lecture notes`.

We will be using the departments Dec Workstations which are unix based.

If you have not use unix or a workstation before do not worry the first tutorial session is to be used for this purpose.

Details on how to use the system are in Appendix . Also try the exercises that follow.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: The C Program **Up:** About This Course **Previous:** Course Material and On-line facilities

Exercises - Using X Windows, Editing and UNIX Basics

1. Practice opening, closing and moving windows around the screen and to/from the background/foreground. Get used to using the mouse and its buttons for such tasks.
2. Figure out the function of each of the three mouse buttons. Pay particular attention to the different functions the buttons in different windows (applications) and also when the mouse is pointing to the background.
3. Find out how to resize windows etc. and practice this.
4. Fire up `textedit` application and practice editing text files. Create any files you wish for now. Figure out basic options like cut and paste of text around the file, saving and loading files, searching for strings in the text and replacing strings.

   Particularly pay attention in getting used to using the Key Strokes and / or mouse to perform the above tasks.
5. Use Unix Commands (see Appendix) to
   1. Copy a file (created by text editor or other means) to another file called spare.
   2. Rename your original file to b called new.
   3. Delete the file spare.
   4. Display your original file on the terminal.
   5. Print your file out.
6. Familiarise yourself with other UNIX functions by creating various files of text etc. and trying out the various functions listed in handouts.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Why Use CEILIDH ? **Up:** Programming in C **Previous:** Exercises

Ceilidh - On Line C Tutoring System

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: About this document ... **Up:** Programming in C **Previous:** timer.c

Using Dec Workstations and Unix

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Contents

Dave Marshall
1/5/1999

Subsections

The Common Desktop Environment

In order to use Solaris and most other Unix Systems you will need to be familiar with the Common Desktop Environment (CDE). Before embarking on learning C with briefly introduce the main features of the CDE.

Most major Unix vendors now provide the CDE as standard. Consequently, most users of the X Window system will now be exposed to the CDE. Indeed, continuing trends in the development of Motif and CDE will probably lead to a convergence of these technologies in the near future. This section highlights the key features of the CDE from a Users perspective.

Upon login, the user is presented with the CDE Desktop (Fig. 1.1). The desktop includes a front panel (Fig. 1.2) , multiple virtual workspaces, and window management. CDE supports the running of applications from a file manager, from an application manager and from the front panel. Each of the subcomponents of the desktop are described below.

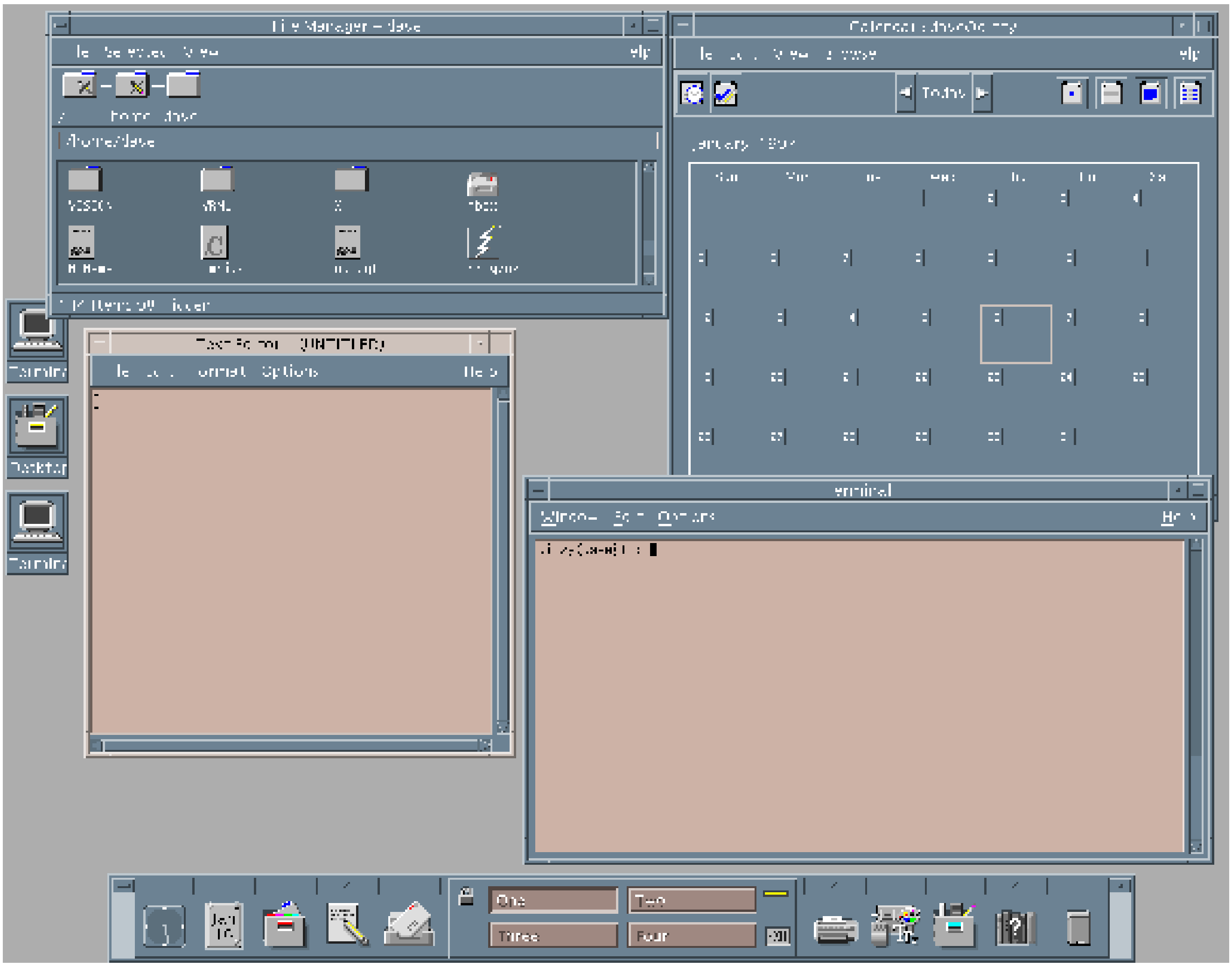


Fig. 1.1 Sample CDE Desktop

The front panel

The front panel (Fig. 1.2) contains a set of icons and popup menus (more like roll-up menus) that appear at the bottom of the screen, by default (Fig. 1.1). The front panel contains the most regularly used applications and tools for managing the workspace. Users can drag-and-drop application icons from the file manager or application manager to the popups for addition of the application(s) to the associated menu. The user can also manipulate the default actions and icons for the popups. The front panel can be locked so that users can't change it. A user can configure several virtual workspaces -- each with different backgrounds and colors if desired. Each workspace can have any number of applications running in it. An application can be set to appear in one, more than one, or all workspaces simultaneously.

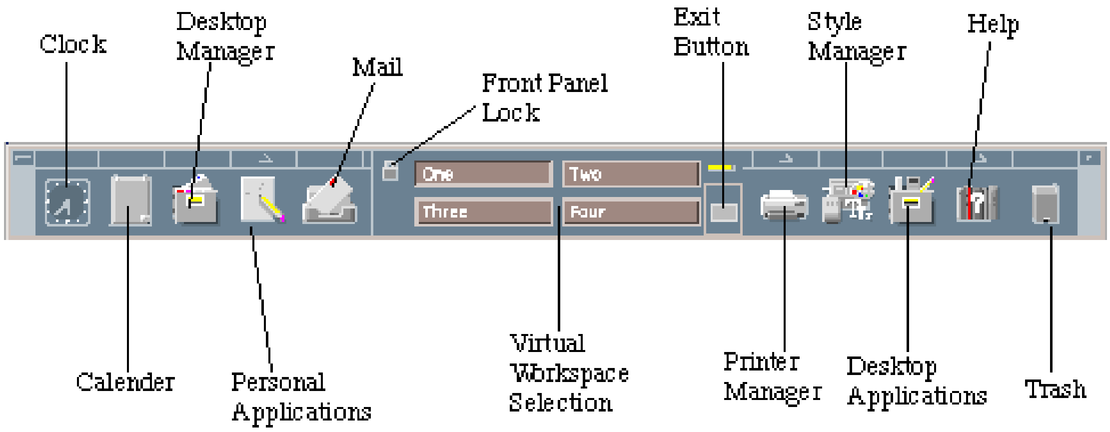


Fig. 1.2 Clients, Servers and Xlib

The file manager

CDE includes a standard file manager. The functionality is similar to that of the Microsoft Windows, Macintosh, or Sun Open Look file manager. Users can directly manipulate icons associated with UNIX files, drag-and-drop them, and launch associated applications.

The application manager

The user interaction with the application manager is similar to the file manager except that is is intended to be a list of executable modules available to a particular user. The user launches the application manager from an icon in the front panel. Users are notified when a new application is available on a server by additions (or deletions) to the list of icons in the application manager window. Programs and icons can be installed and pushed out to other workstations as an integral part of the installation process. The list of workstations that new software is installed on is configurable. The application manager comes preconfigured to include several utilities and programs.

The session manager

The session manager is responsible for the start up and shut down of a user session. In the CDE, applications that are made *CDE aware* are warned via an X Event when the X session is closing down. The application responds by returning a string that can be used by the session manager at the user's next login to restart the application. CDE can remember two sessions per user. One is the *current* session, where a snapshot of the currently running applications is saved. These applications can be automatically restarted at the user's next login. The other is the default login, which is analogous to starting an X session in the Motif window manager. The user can choose which of the two sessions to use at the next login.

Other CDE desktop tools

CDE 1.0 includes a set of applications that enable users to become productive immediately. Many of these are available directly from the front panel, others from the desktop or personal application managers. Common and productive desktop tools include:

Mail Tool
: -- Used to compose, view, and manage electronic mail through a GUI. Allows the inclusion of attachments and communications with other applications through the messaging system.

Calendar Manager
: -- Used to manage, schedule, and view appointments, create calendars, and interact with the Mail Tool.

Editor
: -- A text editor with common functionality including data transfer with other applications via the clipboard, drag and drop, and primary and quick transfer.

Terminal Emulator
: -- An *xterm* terminal emulator.

Calculator
: -- A standard calculator with scientific, financial, and logical modes.

Print Manager
: -- A graphical print job manager for the scheduling and management of print jobs on any available printer.

Help System
: -- A context-sensitive graphical help system based on Standard Generalized Markup Language (SGML).

Style Manager
: -- A graphical interface that allows a user to interactively set their preferences, such as colors, backdrops, and fonts, for a session.

Icon Editor
: -- This application is a fairly full featured graphical icon (pixmap) editor.

Application development tools

CDE includes two components for application development. The first is a shell command language interpreter that has built-in commands for most X Window system and CDE functions. The interpreter is based on ksh93 (The Korn Shell), and should provide anyone familiar with shell scripts the ability to develop X, Motif, and CDE applications.

To support interactive user interface development, developers can use the Motif Application Builder. This is a GUI front end for building Motif applications that generates C source code. The source code is then compiled and linked with the X and Motif libraries to produce the executable binary.

Application integration

CDE provides a number of tools to ease integration. The overall model of the CDE session is intended to allow a straightforward integration for virtually all types of applications. Motif and other X toolkit applications usually require little integration.

The task of integrating in-house and third party applications into a desktop, often the most difficult aspect of a desktop installation, is simplified by CDE. The power and advantage of CDE functionality can be realized in most cases without recompiling applications.

For example, Open Look applications can be integrated through the use of scripts that perform front-end execution of the application and scripts that perform pre- and post-session processing.

After the initial task of integrating applications so that they fit within session management, further integration can be done to increase their overall common *look-and-feel* with the rest of the desktop and to take advantage of the full range of CDE functionality. Tools that ease this aspect of integration include an *Icon Editor* used to create colour and monochrome icons. Images can be copied from the desktop into an icon, or they can be drawn freehand.

The *Action Creation Utility* is used to create action entries in the action database. Actions allow applications to be launched using desktop icons, and they ease administration by removing an application's specific details from the user interface.

The *Application Gather* and *Application Integrate* routines are used to control and format the application manager. They simplify installations so that applications can be accessible from virtually anywhere on the network.

Windows and the Window Manager

From a user's perspective, one of the first distinguishing features of Motif's *look and feel* is the *window frame* (Fig. 1.3). Every application window is contained inside such a frame. The following items appear in the window frame:

Fig. 1.3 The Motif Window Frame

Title Bar

-- This identifies the window by a text string. The string is usually the name of the application program. However, an application's resource controls the label (Chapter).

Window Menu

-- Every window under the control of *mwm* has a window menu. The application has a certain amount of control over items that can be placed in the menu. The *Motif Style Guide* insists that certain commands are always available in this menu and that they can be accessed from either mouse or keyboard selection. Keyboard selections are called *mnemonics* and allow routine actions (that may involve several mouse actions) to be called from the keyboard. The action from the keyboard usually involves pressing two keys at the same time: the `Meta` key and another key. The default window menu items and *mnemonics* are listed below and illustrated in Fig. 1.4:

Fig. 1.4 The Window Menu

- **Restore (`Meta+F5`)** -- Restore window to previous size after iconification (*see* below).
- **Move (`Meta+F7`)** -- Allows the window to be repositioned with a drag of the mouse.
- **Size (`Meta+F8`)** -- Allows the size of the window to be changed by dragging on the corners of the window.
- **Minimize (`Meta+F9`)** -- Iconify the window.
- **Maximize (`Meta+F10`)** -- Make the window the size of the root window, usually the whole of the display size.
- **Lower (`Meta+F3`)** -- Move the window to the bottom of the window stack. Windows may be *tiled* on top of each other (*see* below). The front window being the top of the stack.
- **Close (`Meta+F4`)** -- Quit the program. Some simple applications (Chapter) provide no *internal* means of termination. The `Close` option being the only means to achieve this.

Minimize Button

-- another way to iconify a window .

Maximize Button

-- another way to make a window the size of the root window .

The window manager must also be able to manage multiple windows from multiple client applications. There are a few important issues that need to be resolved. When running several applications together, several windows may be displayed on the screen. As a result, the display may appear cluttered and hard to navigate. The window manager provides two mechanisms to help deal with such problems:

Active Window

-- Only one window can receive input at any time. If you are selecting a graphical object with a mouse, then it is relatively easy for the window manager to detect this and schedule appropriate actions related to the chosen object. It is not so easy when you enter data or make selections directly from the keyboard. To resolve this only one window at a time is allowed *keyboard focus*. This window is called the *active window*. The selection of the active window will depend on the system configuration which the user typically has control over. There are two common methods for selecting the active window:

Focus follows pointer

-- The active window is the window is the window underneath mouse pointer.

Click-to-type

-- The active window is selected, by clicking on an area of the window, and remains active until another window is selected no matter where the mouse points.

When a window is made active its appearance will change slightly:

- Its outline frame will become shaded.

- The cursor will change appearance when placed in the window.
- The window may jump, or be *raised* to the top of the window stack.

The exact appearance of the above may vary from system to system and may be controlled by the user by setting environment settings in the window manager.

Window tiling

-- Windows may be stacked on top of each other. The window manager tries to maintain a three-dimensional *look and feel* . Apart from the fact that buttons, dialog boxes appear to be elevated from the screen, windows are shaded and framed in a three-dimensional fashion. The top window (or currently active window) will have slightly different appearance for instance.

The window menu has a few options for controlling the tiling of a window. Also a window can be brought to the top of the stack, or *raised* by clicking a part of its frame.

Iconification

-- If a window is currently active and not required for input or displaying output then it may be *iconified* or *minimised* thus reducing the screen clutter. An icon (Fig. 1.5) is a small graphical symbol that represents the window (or application). It occupies a significantly less amount of screen area. Icons are usually arranged around the perimeter (typically bottom or left side) of the screen. The application will still be running and occupying computer memory. The window related to the icon may be reverted to by either double clicking on the icon, or selecting *Restore* or *Maximise* from the icon's window menu.

Figure 1.5: Sample Icon from Xterm Application

The Root Menu

The *Root Menu* is the main menu of the window manager. The root menu typically is used to control the whole display, for example starting up new windows and quitting the desktop. To display the Root menu:

- Move the mouse pointer to the Root Window.
- Hold down the left mouse button.

The default Root Menu has the following The root menu can be customised to start up common applications for example. The root menu for the *mwm* (Fig. 1.6) and *dtwm* (Fig. 1.7) have slightly different appearance but have broadly similar actions, which are summarised below:

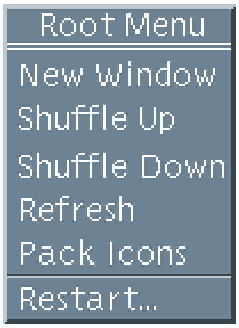


Fig. 1.6 The *mwm* Root Menu

Fig. 1.7 The CDE *dtwm* Root Menu

Program
: (*dtwm*) -- A sub-menu is displayed that allows a variety of programs to be called from the desktop, for example to create a new window. The list of available programs can be customised from the desktop.

New Window
: (*mwm*) -- Create a new window which is usually an *Xterm* window.

Shuffle Up
: -- Move the bottom of the window stack to the top.

Shuffle Down
: -- Move the top of the window stack to the bottom.

Refresh
: -- Refresh the current screen display.

Restart
: -- Restart the Workspace.

Logout
: (*dtwm*) -- Quit the Window Manager.

Exercises

Exercise 12158

Exercise~\ref{ex.cde1}

Add an application to the application manager

Exercise 12159

Practice opening, closing and moving windows around the screen and to/from the background/foreground. Get used to using the mouse and its buttons for such tasks.

Exercise 12160

Figure out the function of each of the three mouse buttons. Pay particular attention to the different functions the buttons in different windows (applications) and also when the mouse is pointing to the background.

Exercise 12161

Find out how to resize windows etc. and practice this.

Exercise 12162

Fire up the texteditor of your choice (You may use `dtpad` (basic but functional), `textedit` application (SOLARIS basic editor), emacs/Xemacs, or vi) and practice editing text files. Create any files you wish for now. Figure out basic options like cut and paste of text around the file, saving and loading files, searching for strings in the text and replacing strings.

Particularly pay attention in getting used to using the Key Strokes and / or mouse to perform the above tasks.

Exercise 12163

Use Unix Commands to

1. Copy a file (created by text editor or other means) to another file called spare.
2. Rename your original file to b called new.
3. Delete the file spare.
4. Display your original file on the terminal.
5. Print your file out.

Exercise 12164

Familiarise yourself with other UNIX functions by creating various files of text etc. and trying out the various functions listed in handouts.

Dave Marshall
1/5/1999

Subsections

C/C++ Program Compilation

In this chapter we begin by outlining the basic processes you need to go through in order to compile your C (or C++) programs. We then proceed to formally describe the C compilation model and also how C supports additional libraries.

Creating, Compiling and Running Your Program

The stages of developing your C program are as follows. (See Appendix and exercises for more info.)

Creating the program

Create a file containing the complete program, such as the above example. You can use any ordinary editor with which you are familiar to create the file. One such editor is *textedit* available on most UNIX systems.

The filename must by convention end ``.c'' (full stop, lower case c), *e.g. myprog.c* or *progtest.c*. The contents must obey C syntax. For example, they might be as in the above example, starting with the line `/* Sample ....` (or a blank line preceding it) and ending with the line `} /* end of program */` (or a blank line following it).

Compilation

There are many C compilers around. The `cc` being the default Sun compiler. The GNU C compiler `gcc` is popular and available for many platforms. PC users may also be familiar with the Borland `bcc` compiler.

There are also equivalent C++ compilers which are usually denoted by `CC` (*note* upper case CC. For example Sun provides CC and GNU `GCC`. The GNU compiler is also denoted by `g++`

Other (less common) C/C++ compilers exist. All the above compilers operate in essentially the same manner and share many common command line options. Below and in Appendix we list and give example uses many of the common compiler options. However, the **best** source of each compiler is through the online manual pages of your system: *e.g.* `man cc`.

For the sake of compactness in the basic discussions of compiler operation we will simply refer to the `cc` compiler -- other compilers can simply be substituted in place of `cc` unless otherwise stated.

To Compile your program simply invoke the command `cc`. The command must be followed by the name of the (C) program you wish to compile. A number of compiler options can be specified also. We will not concern ourselves with many of these options yet, some useful and often essential options are introduced below -- See Appendix or online manual help for further details.

Thus, the basic compilation command is:

```
cc program.c
```

where *program.c* is the name of the file.

If there are obvious errors in your program (such as mistypings, misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them.

There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations.

When the compiler has successfully digested your program, the compiled version, or executable, is left in a file called *a.out* or if the compiler option -*o* is used : the file listed after the -*o*.

It is more convenient to use a -*o* and filename in the compilation as in

```
cc -o program program.c
```

which puts the compiled program into the file program (or any file you name following the "-o" argument) **instead** of putting it in the file a.out .

Running the program

The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case *program* (or *a.out*)

This executes your program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output.

If so, you must return to edit your program source, and recompile it, and run it again.

The C Compilation Model

We will briefly highlight key features of the C Compilation model (Fig. 2.1) here.

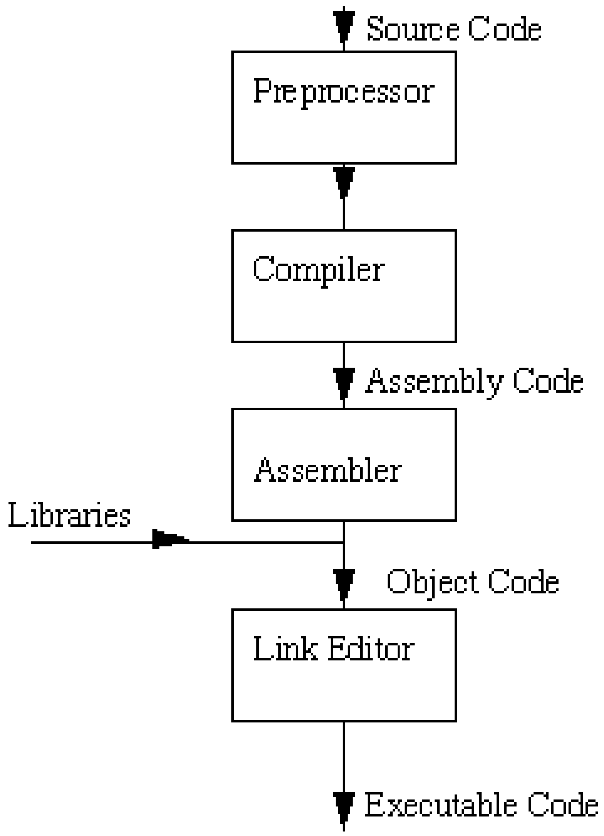


Fig. 2.1 The C Compilation Model

The Preprocessor

We will study this part of the compilation process in greater detail later (Chapter 13. However we need some basic information for some C programs.

The Preprocessor accepts source code as input and is responsible for

- removing comments
- interpreting special *preprocessor directives* denoted by #.

For example

- `#include` -- includes contents of a named file. Files usually called *header* files. *e.g*
  - `#include <math.h>` -- standard library maths file.
  - `#include <stdio.h>` -- standard library I/O file
- `#define` -- defines a symbolic name or constant. Macro substitution.
  - `#define MAX_ARRAY_SIZE 100`

C Compiler

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler

The assembler creates object code. On a UNIX system you may see files with a `.o` suffix (`.OBJ` on MSDOS) to indicate object code files.

Link Editor

If a source file references library functions or functions defined in other source files the *link editor* combines these functions (with `main()`) to create an executable file. External Variable references resolved here also. *More on this later* (Chapter 34).

Some Useful Compiler Options

Now that we have a basic understanding of the compilation model we can now introduce some useful and sometimes essential common compiler options. Again see the online `man` pages and Appendix for further information and additional options.

`-c`

Suppress the linking process and produce a `.o` file for each source file listed. Several can be subsequently linked by the `cc` command, for example:

```
cc file1.o file2.o ...... -o executable
```

`-llibrary`

Link with object libraries. This option must follow the source file arguments. The object libraries are archived and can be standard, third party or user created libraries (We discuss this topic briefly below and also in detail later (Chapter 34). Probably the most commonly used library is the math library (`math.h`). You must link in this library explicitly if you wish to use the maths functions (**note** do note forget to

#include <math.h> header file), for example:

```
cc calc.c -o calc -lm
```

Many other libraries are linked in this fashion (see below)

-Ldirectory

Add directory to the list of directories containing object-library routines. The linker always looks for standard and other system libraries in /lib and /usr/lib. If you want to link in libraries that you have created or installed yourself (unless you have certain privileges and get the libraries installed in /usr/lib) you **will** have to specify where you files are stored, for example:

```
cc prog.c -L/home/myname/mylibs mylib.a
```

-Ipathname

Add pathname to the list of directories in which to search for #include files with relative filenames (not beginning with slash /).

BY default, The preprocessor first searches for #include files in the directory containing source file, then in directories named with -I options (if any), and finally, in /usr/include. So to include header files stored in /home/myname/myheaders you would do:

```
cc prog.c -I/home/myname/myheaders
```

Note: System library header files are stored in a special place (/usr/include) and are not affected by the -I option. System header files and user header files are included in a slightly different manner (see Chapters 13 and 34)

-g

invoke debugging option. This instructs the compiler to produce additional symbol table information that is used by a variety of debugging utilities.

-D

define symbols either as identifiers (-D*identifer*) or as values (-D*symbol*=*value*) in a similar fashion as the #define preprocessor command. For more details on the use of this argument see Chapter 13.

For further information on general compiler options and the GNU compiler refer to Appendix .

Using Libraries

C is an extremely small language. Many of the functions of other languages are not included in C. *e.g.* No built in I/O, string handling or maths functions.

What use is C then?

C provides functionality through a rich set function libraries.

As a result most C implementations include *standard* libraries of functions for many facilities (I/O *etc.*). For many practical purposes these may be regarded as being part of C. But they may vary from machine to machine. (*cf* Borland C for a PC to UNIX C).

A programmer can also develop his or her own function libraries and also include special purpose third party libraries (*e.g.* NAG, PHIGS).

All libraries (except standard I/O) need to be explicitly linked in with the `-l` and, possibly, `-L` compiler options described above.

UNIX Library Functions

The UNIX system provides a large number of C functions as libraries. Some of these implement frequently used operations, while others are very specialised in their application.

Do Not Reinvent Wheels: It is wise for programmers to check whether a library function is available to perform a task before writing their own version. This will reduce program development time. The library functions have been tested, so they are more likely to be correct than any function which the programmer might write. This will save time when debugging the program.

Later chapters deal with all important standard library issues and other common system libraries.

Finding Information about Library Functions

The UNIX manual has an entry for all available functions. Function documentation is stored in *section 3* of the manual, and there are many other useful system calls in *section 2*. If you already know the name of the function you want, you can read the page by typing (to find about `sqrt`):

```
man 3 sqrt
```

If you don't know the name of the function, a full list is included in the introductory page for section 3 of the manual. To read this, type

```
man 3 intro
```

There are approximately 700 functions described here. This number tends to increase with each upgrade of the system.

On any manual page, the SYNOPSIS section will include information on the use of the function. For example:

```
#include <time.h>

char *ctime(time_t *clock)
```

This means that you must have

```
#include <time.h>
```

in your file before you call `ctime`. And that function ctime takes a pointer to type `time_t` as an argument, and returns a `string (char *)`. `time_t` will probably be defined in the same manual page.

The DESCRIPTION section will then give a short description of what the function does. For example:

```
ctime() converts a long integer, pointed to by clock,  to  a
26-character  string  of the form produced by asctime().
```

Lint -- A C program verifier

You will soon discover (if you have not already) that the C compiler is pretty vague in many aspects of checking program correctness, particularly in type checking. Careful use of prototyping of functions can assist modern C compilers in this task. However, There is still no guarantee that once you have successfully compiled your program that it will run correctly.

The UNIX utility `lint` can assist in checking for a multitude of programming errors. Check out the online manual pages (`man lint`) for complete details of lint. It is well worth the effort as it can help save many hours debugging your C code.

To run lint simply enter the command:

`lint myprog.c`.

Lint is particularly good at checking type checking of variable and function assignments, efficiency, unused variables and function identifiers, unreachable code and possibly memory leaks. There are many useful options to help control lint (see `man lint`).

Exercises

Exercise 12171

Enter, compile and run the following program:

```
main()

{ int i;

   printf("\t Number \t\t Square of Number\n\n");

   for (i=0; i<=25;++i)
    printf("\t %d \t\t\t %d \n",i,i*i);
```

```
        }
```

Exercise 12172

The following program uses the math library. Enter compile and run it correctly.

```
#include <math.h>

        main()

        { int i;

           printf("\t Number \t\t Square Root of Number\n\n");

           for (i=0; i<=360; ++i)
            printf("\t %d \t\t\t %d \n",i, sqrt((double) i));

        }
```

Exercise 12173

Look in `/lib` and `/usr/lib` and see what libraries are available.

- Use the `man` command to get details of library functions
- Explore the libraries to see what each contains by running the command `ar t libfile`.

Exercise 12174

Look in `/usr/include` and see what header files are available.

- Use the `more` or `cat` commands to view these text files
- Explore the header files to see what each contains, note the include, define, type definitions and function prototypes declared in them

Exercise 12175

Suppose you have a C program whose main function is in `main.c` and has other functions in the files `input.c` and `output.c`:

- What command(s) would you use on your system to compile and link this program?
- How would you modify the above commands to link a library called `process1` stored in the standard system library directory?
- How would you modify the above commands to link a library called `process2` stored in your home directory?
- Some header files need to be read and have been found to located in a `header` subdirectory of your home directory and also in the current working directory. How would you modify the compiler commands to account for this?

Exercise 12176

Suppose you have a C program composed of several separate files, and they include one another as shown below:

Figure 1.5: Sample Icon from Xterm Application

File	Include Files
`main.c`	`stdio.h, process1.h`
`input.c`	`stdio.h, list.h`
`output.c`	`stdio.h`
`process1.c`	`stdio.h, process1.h`
`process2.c`	`stdio.h, list.h`

- Which files have to recompiled after you make changes to `process1.c`?
- Which files have to recompiled after you make changes to `process1.h`?
- Which files have to recompiled after you make changes to `list.h`?

Dave Marshall
1/5/1999

Subsections

C Basics

Before we embark on a brief tour of C's basic syntax and structure we offer a brief history of C and consider the characteristics of the C language.

In the remainder of the Chapter we will look at the basic aspects of C programs such as C program structure, the declaration of variables, data types and operators. We will assume knowledge of a high level language, such as PASCAL.

It is our intention to provide a quick guide through similar C principles to most high level languages. Here the syntax may be slightly different but the concepts exactly the same.

C does have a few surprises:

- Many High level languages, like PASCAL, are highly disciplined and structured.
- **However beware** -- C is much more flexible and free-wheeling. This freedom gives C much more **power** that experienced users can employ. The above example below (`mystery.c`) illustrates how bad things could really get.

History of C

The *milestones* in C's development as a language are listed below:

- UNIX developed c. 1969 -- DEC PDP-7 Assembly Language
- BCPL -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language ``B" a second attempt. c. 1970.
- A totally new language ``C" a successor to ``B". c. 1971
- By 1973 UNIX OS almost totally written in ``C".

Characteristics of C

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

As an extreme example the following C code (`mystery.c`) is actually *legal* C code.

```
#include <stdio.h>

main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

It will compile and run and produce meaningful output. Try this program out. Try to compile and run it yourself. Alternatively you may run it from here and see the output.

Clearly nobody ever writes code like or at least should never. This piece of code actually one an international Obfuscated C Code Contest http://reality.sgi.com/csp/iocc The standard for C programs was originally the features set by Brian Kernighan. In order to make the language more internationally acceptable, an international standard was developed, ANSI C (American National Standards Institute).

C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions

We must have a `main()` function.

A function has the form:

```
type function_name (parameters)
                {
                                local variables

                                C Statements

                 }
```

If the type definition is omitted C assumes that function returns an **integer** type. **NOTE:** This can be a source of problems in a program.

So returning to our first C program:

```
  /* Sample program */

                main()
                                {

                                printf( ``I Like C \n'' );

                                exit ( 0 );

                                }
```

NOTE:

- C requires a semicolon at the end of **every** statement.
- printf is a *standard* C function -- called from main.
- \n signifies newline. **Formatted output** -- more later.
- exit() is also a standard function that causes the program to terminate. Strictly speaking it is not needed here as it is the last line of main() and the program will terminate anyway.

Let us look at another printing statement:

```
printf(``.\n.1\n..2\n...3\n'');
```

The output of this would be:

```
    .
                .1
                ..2
                ...3
```

Variables

C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	-32768	$+32767$
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

The Pascal Equivalents are:

C type	Pascal equivalent
char	char
unsigned char	—
short int	integer
unsigned short int	—
long int	longint
float	real
double	extended

On UNIX systems all `int`s are `long int`s unless specified as `short int` explicitly.

NOTE: There is **NO** Boolean type in C -- you should use `char, int` or (better) `unsigned char`.

`Unsigned` can be used with all `char` and `int` types.

To declare a variable in C, do:

`var_type` *list variables*;

```
e.g. int i,j,k;
                 float x,y,z;
                 char ch;
```

Defining Global Variables

Global variables are defined above `main()` in the following way:-

```
short number,sum;
        int bignumber,bigsum;
        char letter;

        main()
                        {

                        }
```

It is also possible to pre-initialise global variables using the = operator for assignment.

NOTE: The = operator is the same as := is Pascal.

For example:-

```
float sum=0.0;
        int bigsum=0;
        char letter=`A';

        main()
                        {

                        }
```

This is the same as:-

```
float sum;
        int bigsum;
        char letter;

        main()
                        {

                        sum=0.0;
                        bigsum=0;
                        letter=`A';

                        }
```

...but is more efficient.

C also allows multiple assignment statements using =, for example:

```
a=b=c=d=3;
```

...which is the same as, but more efficient than:

```
a=3;
        b=3;
```

```
                c=3;
                d=3;
```

This kind of assignment is only possible if all the variable types in the statement are the same.

You can define your own types use typedef. This will have greater relevance later in the course when we learn how to create more complex data structures.

As an example of a simple use let us consider how we may define two new types real and letter. These new types can then be used in the same way as the pre-defined C types:

```
          typedef real float;
                 typedef letter char;
```

Variables declared:

```
                 real sum=0.0;
                 letter nextletter;
```

Printing Out and Inputting Variables

C uses formatted output. The `printf` function has a special formatting character (%) -- a character following this defines a certain format for a variable:

```
    %c -- characters
                 %d -- integers
                 %f -- floats
```

e.g. `printf(``%c %d %f'',ch,i,x);`

NOTE: Format statement enclosed in ``...'', variables follow after. Make sure order of format and variable data types match up.

`scanf()` is the function for inputting values to a data structure: Its format is similar to `printf`:

i.e. `scanf(``%c %d %f'',&ch,&i,&x);`

NOTE: & before variables. Please accept this for now and **remember** to include it. It is to do with pointers which we will meet later (Section 17.4.1).

Constants

ANSI C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The `const` keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

Note:

- You can declare the `const` before or after the type. Choose one an stick to it.
- It is usual to initialise a `const` with a value as it cannot get a value *any other way*.

The preprocessor #`define` is another more flexible (see Preprocessor Chapters) method to define *constants* in a program.

You frequently see const declaration in function parameters. This says simply that the function is **not** going to change the value of the parameter.

The following function definition used concepts we have not met (see chapters on functions, strings, pointers, and standard libraries) but for completenes of this section it is is included here:

```
void strcpy(char *buffer, char const *string)
```

The second argiment `string` is a C string that will not be altered by the string copying standard library function.

Arithmetic Operations

As well as the standard arithmetic operators (`+ - * /`) found in most languages, C provides some more operators. There are some notable differences with other languages, such as Pascal.

Assignment is = *i.e. i* = 4; `ch` = `y';

Increment ++, Decrement - which are more efficient than their long hand equivalents, for example:- `x++` is faster than `x=x+1`.

The `++` and `-` operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.

In the example below, `++z` is pre-fixed and the `w-` is post-fixed:

```
int x,y,w;

        main()
                          {

                          x=((++z)-(w-)) % 100;

                          }
```

This would be equivalent to:

```
int x,y,w;

        main()
                          {

                         z++;
                         x=(z-w) % 100;
                         w-;

                          }
```

`The % (modulus) operator only works with integers.`

`Division / is for both integer and float division. So be careful.`

The answer to: *x* = 3 / 2 is 1 even if *x* is declared a float!!

RULE: If both arguments of / are integer then do integer division.

So make sure you do this. The correct (for division) answer to the above is *x* = 3.0 / 2 or *x*= 3 / 2.0 or (better) *x* = 3.0 / 2.0.

There is also a convenient **shorthand** way to express computations in C.

It is very common to have expressions like: *i* = *i* + 3 or *x* = *x**(*y* + 2)

This can written in C (generally) in a *shorthand* form like this:

$$expr_1 \; op = expr_2$$

which is equivalent to (but more efficient than):

$$expr_1 = expr_1 \; op \; expr_2$$

So we can rewrite *i* = *i* + 3 as *i* += 3

and *x* = *x**(*y* + 2) as *x* *= *y* + 2.

NOTE: that *x* *= *y* + 2 means *x* = *x**(*y* + 2) and **NOT** *x* = *x***y* + 2.

Comparison Operators

To test for equality is ==

A warning: Beware of using ``=" instead of ``==", such as writing accidentally

```
if ( i = j ) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if `j` is non-zero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Logical Operators

Logical operators are usually used with conditional statements which we shall meet in the next Chapter.

The two basic logical operators are:

&& for logical AND, || for logical OR.

Beware & and | have a different meaning for bitwise AND and OR (*more on this later* in Chapter 12).

Order of Precedence

It is necessary to be careful of the meaning of such expressions as `a + b * c`

We may want the effect as either

```
(a + b) * c
```

or

```
a + (b * c)
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that

```
a - b - c
```

is evaluated as

```
( a - b ) - c
```

as you would expect.

From high priority to low priority the order for all C operators (we have not met all of them yet) is:

```
    ( )  [  ]  -> .
                ! ~ - * & sizeof cast ++ -
                      (these are right->left)
                * / %
                + -
                < <= >= >
                == !=
                &
                ^                 |
                &&
                ||
                ?:        (right->left)
                = += -= (right->left)
                ,   (comma)
```

Thus

```
a < 10 && 2 * b < c
```

is interpreted as

```
( a < 10 ) && ( ( 2 * b ) < c )
```

and

```
  a =

                b =
                                spokes / spokes_per_wheel
                                + spares;
```

as

```
a =
                ( b =
                                ( spokes / spokes_per_wheel )
                                + spares
                );
```

Exercises

Write C programs to perform the following tasks.

Exercise 12270

Input two numbers and work out their sum, average and sum of the squares of the numbers.

Exercise 12271

Input and output your name, address and age to an appropriate structure.

Exercise 12272

Write a program that works out the largest and smallest values from a set of 10 inputted numbers.

Exercise 12273

Write a program to read a "float" representing a number of degrees Celsius, and print as a "float" the equivalent temperature in degrees Fahrenheit. Print your results in a form such as

100.0 degrees Celsius converts to 212.0 degrees Fahrenheit.

Exercise 12274

Write a program to print several lines (such as your name and address). You may use either several printf instructions, each with a newline character in it, or one printf with several newlines in the string.

Exercise 12275

Write a program to read a positive integer at least equal to 3, and print out all possible permutations of three positive integers less or equal to than this value.

Exercise 12276

Write a program to read a number of units of length (a float) and print out the area of a circle of that radius. Assume that the value of pi is 3.14159 (an appropriate declaration will be given you by ceilidh - select setup).

Your output should take the form: The area of a circle of radius ... units is .... units.

If you want to be clever, and have looked ahead in the notes, print the message Error: Negative values not permitted. if the input value is negative.

Exercise 12277

Given as input a floating (real) number of centimeters, print out the equivalent number of feet (integer) and inches (floating, 1 decimal), with the inches given to an accuracy of one decimal place.

Assume 2.54 centimeters per inch, and 12 inches per foot.

If the input value is 333.3, the output format should be:

333.3 centimeters is 10 feet 11.2 inches.

Exercise 12278

Given as input an integer number of seconds, print as output the equivalent time in hours, minutes and seconds. Recommended output format is something like

7322 seconds is equivalent to 2 hours 2 minutes 2 seconds.

Exercise 12279

Write a program to read two integers with the following significance.

The first integer value represents a time of day on a 24 hour clock, so that 1245 represents quarter to one mid-day, for example.

The second integer represents a time duration in a similar way, so that 345 represents three hours and 45 minutes.

This duration is to be added to the first time, and the result printed out in the same notation, in this case 1630 which is the time 3 hours and 45 minutes after 12.45.

Typical output might be Start time is 1415. Duration is 50. End time is 1505.

There are a few extra marks for spotting.

Start time is 2300. Duration is 200. End time is 100.

Dave Marshall
1/5/1999

Subsections

Conditionals

This Chapter deals with the various methods that C can control the *flow* of logic in a program. Apart from slight syntactic variation they are similar to other languages.

As we have seen following logical operations exist in C:

==, !=, $||$, &&.

One other operator is the unitary - it takes only one argument - *not* **!**.

These operators are used in conjunction with the following statements.

The `if` statement

The `if` statement has the same function as other languages. It has three basic forms:

```
if  (expression)
                                statement
```

...or:

```
if  (expression)
                                statement1
               else
                                statement2
```

...or:

```
if  (expression)
                                statement1
               else if (expression)
                                statement2
               else
                                statement3
```

For example:-

```
int x,y,w;

               main()
                                {

                                if (x>0)
                                               {
                                               z=w;
                                               ........
                                               }
                               else
                                               {
                                               z=y;
                                               ........
```

```
        }
    }
```

The ? operator

The `?` (*ternary condition*) operator is a more efficient form for expressing simple `if` statements. It has the following form:

$$expression_1 \; ? \; expression_2 : \; expression_3$$

It simply states:

`if` $expression_1$ `then` $expression_2$ `else` $expression_3$

For example to assign the maximum of `a` and `b` to `z`:

```
z = (a>b) ? a : b;
```

which is the same as:

```
if (a>b)
        z = a;
    else
        z=b;
```

The `switch` statement

The C `switch` is similar to Pascal's `case` statement and it allows multiple choice of a selection of items at one level of a conditional where it is a far neater way of writing multiple `if` statements:

```
switch (expression) {
        case item1:
                statement1;
                break;
        case item2:
                statement2;
                break;
                ⋮
⋮
        case itemn:
                statementn;
                break;
        default:
                statement;
                break;
    }
```

In each case the value of $item_i$ must be a constant, variables are not allowed.

The `break` is needed if you want to terminate the `switch` after execution of one choice. Otherwise the next case would get evaluated. **Note:** This is unlike most other languages.

We can also have **null** statements by just including a ; or let the switch statement *fall through* by omitting any statements (see *e.g.* below).

The `default` case is optional and catches any other cases.

For example:-

```
switch (letter)
        {
         case `A':
         case `E':
         case `I':
         case `O':
         case `U':
                    numberofvowels++;
                    break;

         case ` ':
                    numberofspaces++;
                    break;

         default:
                    numberofconstants++;
                    break;
        }
```

In the above example if the value of `letter is `A', `E', `I', `O' or `U' then numberofvowels is incremented.`

`If the value of letter is ` ' then numberofspaces is incremented.`

`If none of these is true then the default condition is executed, that is numberofconstants is incremented.`

Exercises

Exercise 12304

Write a program to read two characters, and print their value when interpreted as a 2-digit hexadecimal number. Accept upper case letters for values from 10 to 15.

Exercise 12305

Read an integer value. Assume it is the number of a month of the year; print out the name of that month.

Exercise 12306

Given as input three integers representing a date as day, month, year, print out the number day, month and year for the following day's date.

Typical input: 28 2 1992 Typical output: Date following 28:02:1992 is 29:02:1992

Exercise 12307

Write a program which reads two integer values. If the first is less than the second, print the message up. If the second is less than the first, print the message down If the numbers are equal, print the message equal If there is an error reading the data, print a message containing the word Error and perform exit(0);

Dave Marshall
1/5/1999

Subsections

Looping and Iteration

This chapter will look at C's mechanisms for controlling looping and iteration. Even though some of these mechanisms may look familiar and indeed will operate in standard fashion most of the time. **NOTE:** some non-standard features are available.

The `for` statement

The C `for` statement has the following form:

```
for  (expression1; 2; expression3)
                        statement;
                        or {block of statements}
```

expression$_1$ initialises; *expression*$_2$ is the terminate test; *expression*$_3$ is the modifier (which may be more than just simple increment);

NOTE: C basically treats `for` statements as `while` type loops

For example:

```
int x;

           main()
                        {
                       for (x=3;x>0;x-)
                                    {
                                    printf("x=%d\n",x);

                                    }
                        }
```

...outputs:

```
x=3
          x=2
          x=1
```

`...to the screen`

`All the following are legal for statements in C. The practical application of such statements is not important here, we are just trying to illustrate peculiar features of C for that may be useful:-`

```
for (x=0;((x>3) && (x<9)); x++)

           for (x=0,y=4;((x>3) && (y<9)); x++,y+=2)
```

```
                    for (x=0,y=4,z=4000;z; z/=10)
```

The second example shows that multiple expressions can be separated a ,.

In the third example the loop will continue to iterate until z becomes 0;

The `while` statement

The `while` statement is similar to those used in other languages although more can be done with the `expression` statement -- a standard feature of C.

The `while` has the form:

```
  while (expression)
                            statement
```

For example:

```
  int x=3;

                    main()
                            { while (x>0)
                                    { printf("x=%d\n",x);
                                            x-;
                                    }
                            }
```

...outputs:

```
  x=3
                  x=2
                  x=1
```

...to the screen.

Because the while loop can accept expressions, not just conditions, the following are all legal:-

```
  while (x-);
                  while (x=x+1);
                  while (x+=5);
```

Using this type of expression, only when the result of x-, x=x+1, or x+=5, evaluates to 0 will the while condition fail and the loop be exited.

We can go further still and perform complete operations within the while *expression*:

```
  while (i++ < 10);

                  while ( (ch = getchar()) != `q')
                    putchar(ch);
```

The first example counts i up to 10.

The second example uses C standard library functions (See Chapter 18) getchar() - reads a character from the keyboard - and putchar() - writes a given char to screen. The while loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read. **NOTE:** This type of operation is used a lot in C and not just with character reading!! (See Exercises).

The `do-while` statement

C's `do-while` statement has the form:

```
do
        statement;
        while (expression);
```

It is similar to PASCAL's `repeat` ... `until` except `do while` *expression* is true.

For example:

```
int x=3;

    main()
        { do {
            printf("x=%d\n",x-);

            }
        while (x>0);
        }
```

..outputs:-

```
x=3
    x=2
    x=1
```

NOTE: The postfix x- operator which uses the current value of x while printing and *then* decrements x.

`break` and `continue`

C provides two commands to control how we loop:

- `break` -- exit form loop or switch.
- `continue` -- skip 1 iteration of loop.

Consider the following example where we read in integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is great than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

```
while (scanf( ``%d'', &value ) == 1 && value != 0) {

        if (value < 0) {
            printf(``Illegal value\n'');

            break;
            /* Abandon the loop */
        }

        if (value > 100) {
```

```
                                printf(``Invalid value\n'');

                                continue;
                                /* Skip to start loop again */
                        }

                        /* Process the value read */
                          /* guaranteed between 1 and 100 */
                             ....;

                        ....;
        } /* end while value != 0 */
```

Exercises

Exercise 12327

Write a program to read in 10 numbers and compute the average, maximum and minimum values.

Exercise 12328

Write a program to read in numbers until the number -999 is encountered. The sum of all number read until this point should be printed out.

Exercise 12329

Write a program which will read an integer value for a base, then read a positive integer written to that base and print its value.

Read the second integer a character at a time; skip over any leading non-valid (i.e. not a digit between zero and ``base-1") characters, then read valid characters until an invalid one is encountered.

```
           Input        Output
        ==========      ======
        10    1234       1234
         8      77         63   (the value of 77 in base 8, octal)
         2    1111         15   (the value of 1111 in base 2, binary)
```

The base will be less than or equal to 10.

Exercise 12330

Read in three values representing respectively

a capital sum (integer number of pence),

a rate of interest in percent (float),

and a number of years (integer).

Compute the values of the capital sum with compound interest added over the given period of years. Each year's interest is calculated as

interest = capital * interest_rate / 100;

and is added to the capital sum by

capital += interest;

Print out money values as pounds (pence / 100.0) accurate to two decimal places.

Print out a floating value for the value with compound interest for each year up to the end of the period.

Print output year by year in a form such as:

```
Original sum 30000.00 at  12.5 percent for 20 years

Year Interest  Sum
----+-------+--------
  1  3750.00 33750.00
  2  4218.75 37968.75
  3  4746.09 42714.84
  4  5339.35 48054.19
  5  6006.77 54060.96
  6  6757.62 60818.58
  7  7602.32 68420.90
  8  8552.61 76973.51
  9  9621.68 86595.19
 10 10824.39 97419.58
```

Exercise 12331

Read a positive integer value, and compute the following sequence: If the number is even, halve it; if it's odd, multiply by 3 and add 1. Repeat this process until the value is 1, printing out each value. Finally print out how many of these operations you performed.

Typical output might be:

```
 Inital value is 9
 Next value is  28
 Next value is  14
 Next value is   7
 Next value is  22
 Next value is  11
 Next value is  34
 Next value is  17
 Next value is  52
 Next value is  26
 Next value is  13
 Next value is  40
 Next value is  20
 Next value is  10
 Next value is   5
 Next value is  16
 Next value is   8
 Next value is   4
 Next value is   2
 Final value 1, number of steps 19
```

If the input value is less than 1, print a message containing the word

```
       Error
```

and perform an

```
        exit( 0 );
```

Exercise 12332

Write a program to count the vowels and letters in free text given as standard input. Read text a character at a time until you encounter end-of-data.

Then print out the number of occurrences of each of the vowels a, e, i, o and u in the text, the total number of letters, and each of the vowels as an integer percentage of the letter total.

Suggested output format is:

```
Numbers of characters:
a   3 ; e   2 ; i   0 ; o   1 ; u   0 ; rest  17
Percentages of total:
a  13%; e   8%; i   0%; o   4%; u   0%; rest  73%
```

Read characters to end of data using a construct such as

```
char ch;
while(
    ( ch = getchar() ) >= 0
) {
    /* ch is the next character */    ....
}
```

to read characters one at a time using `getchar()` until a negative value is returned.

Exercise 12333

Read a file of English text, and print it out one word per line, all punctuation and non-alpha characters being omitted.

For end-of-data, the program loop should read until "getchar" delivers a value <= 0. When typing input, end the data by typing the end-of-file character, usually control-D. When reading from a file, "getchar" will deliver a negative value when it encounters the end of the file.

Typical output might be

```
Read
a
file
of
English
text
and
print
it
out
one
```

etc.

Dave Marshall
1/5/1999

Subsections

Arrays and Strings

In principle arrays in C are similar to those found in other languages. As we shall shortly see arrays are defined slightly differently and there are many subtle differences due the close link between array and pointers. We will look more closely at the link between pointer and arrays later in Chapter 9.

Single and Multi-dimensional Arrays

Let us first look at how we define arrays in C:

```
int listofnumbers[50];
```

BEWARE: In C Array subscripts start at **0** and end one less than the array size. For example, in the above case valid subscripts range from 0 to 49. This is a **BIG** difference between C and other languages and does require a bit of practice to get in *the right frame of mind*.

Elements can be accessed in the following ways:-

```
thirdnumber=listofnumbers[2];
               listofnumbers[5]=100;
```

```
Multi-dimensional arrays can be defined as follows:
```

```
int tableofnumbers[50][50];
```

```
for two dimensions.
```

```
For further dimensions simply add more [ ]:
```

```
int bigD[50][50][40][30]......[50];
```

```
Elements can be accessed in the following ways:
```

```
anumber=tableofnumbers[2][3];
               tableofnumbers[25][16]=100;
```

Strings

In C Strings are defined as arrays of characters. For example, the following defines a string of 50 characters:

```
char name[50];
```

C has no string handling facilities built in and so the following are all illegal:

```
char firstname[50],lastname[50],fullname[100];

               firstname= "Arnold"; /* Illegal */
```

```
lastname= "Schwarznegger"; /* Illegal */
fullname= "Mr"+firstname
               +lastname; /* Illegal */
```

However, there is a special library of string handling routines which we will come across later.

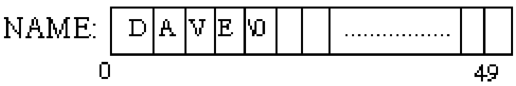
To print a string we use printf with a special **%s** control character:

```
printf(``%s'',name);
```

NOTE: We just need to give the name of the string.

In order to allow variable length strings the \0 character is used to indicate the end of a string.

So we if we have a string, char NAME[50]; and we store the ``DAVE'' in it its contents will look like:



Exercises

Exercise 12335

Write a C program to read through an array of any type. Write a C program to scan through this array to find a particular value.

Exercise 12336

Read ordinary text a character at a time from the program's standard input, and print it with each line reversed from left to right. Read until you encounter end-of-data (see below).

You may wish to test the program by typing

```
prog5rev | prog5rev
```

to see if an exact copy of the original input is recreated.

To read characters to end of data, use a loop such as either

```
char ch;
while( ch = getchar(), ch >= 0 ) /* ch < 0 indicates end-of-data */
```

or

```
char ch;
while( scanf( "%c", &ch ) == 1 ) /* one character read */
```

Exercise 12337

Write a program to read English text to end-of-data (type control-D to indicate end of data at a terminal, see below for detecting it), and print a count of word lengths, i.e. the total number of words of length 1 which occurred, the number of length 2, and so on.

Define a word to be a sequence of alphabetic characters. You should allow for word lengths up to 25 letters.

Typical output should be like this:

```
              length 1 : 10 occurrences
                      length 2 : 19 occurrences
        length 3 : 127 occurrences
               length 4 : 0 occurrences
              length 5 : 18 occurrences
                      ....
```

To read characters to end of data see above question.

Dave Marshall
1/5/1999

Subsections

Functions

C provides functions which are again similar most languages. One difference is that C regards `main()` as function. Also unlike some languages, such as Pascal, C does not have *procedures* -- it uses functions to service both requirements.

Let us remind ourselves of the form of a function:

```
returntype fn_name(1, parameterdef2,...)

{
 localvariables
 functioncode
}
```

Let us look at an example to find the average of two integers:

```
float findaverage(float a, float b)
{ float average;

  average=(a+b)/2;
  return(average);
}
```

We would *call* the function as follows:

```
main()
{   float a=5,b=15,result;

    result=findaverage(a,b);
    printf("average=%f\n",result);
}
```

Note: The return statement passes the result back to the main program.

void functions

The `void` function provide a way of emulating PASCAL type procedures.

If you do not want to return a value you must use the return type `void` and miss out the `return` statement:

```
void squares()
                              { int loop;

                                            for (loop=1;loop<10;loop++);
                                                         printf("%d
\n",loop*loop);

                              }

              main()

                              {             squares();
                              }
```

NOTE: We must have () even for no parameters unlike some languages.

Functions and Arrays

Single dimensional arrays can be passed to functions as follows:-

```
float findaverage(int size,float list[])

                              { int i;
                                            float sum=0.0;

                                            for (i=0;i<size;i++)
                                                         sum+=list[i];
                                            return(sum/size);
                              }
```

Here the declaration `float list[]` tells C that list is an array of float. **Note** we do not specify the dimension of the array when it is a *parameter* of a function.

Multi-dimensional arrays can be passed to functions as follows:

```
void printtable(int xsize,int ysize,
                                float table[][5])

                              { int x,y;

                                            for (x=0;x<xsize;x++)
                                                         { for
(y=0;y<ysize;y++)

printf("\t%f",table[x][y]);

                                                         printf("\n");

                                           }
                              }
```

Here `float table[][5]` tells C that `table` is an array of dimension $N \times 5$ of `float`. **Note** we must specify the second (and subsequent) dimension of the array BUT not the first dimension.

Function Prototyping

Before you use a function C must have *knowledge* about the type it returns and the parameter types the function expects.

The ANSI standard of C introduced a new (better) way of doing this than previous versions of C. (Note: All new versions of C now adhere to the ANSI standard.)

The importance of prototyping is twofold.

- It makes for more structured and therefore easier to read code.
- It allows the C compiler to check the *syntax* of function calls.

How this is done depends on the scope of the function (See Chapter 34). Basically if a functions has been defined before it is used (called) then you are ok to merely use the function.

If NOT then you must *declare* the function. The declaration simply states the type the function returns and the type of parameters used by the function.

It is usual (and therefore **good**) practice to prototype all functions at the start of the program, although this is not strictly necessary.

To *declare* a function prototype simply state the type the function returns, the function name and in brackets list the type of parameters in the order they appear in the function definition.

e.g.

```
int strlen(char []);
```

This states that a function called `strlen` returns an integer value and accepts a single string as a parameter.

NOTE: Functions can be prototyped and variables defined on the same line of code. This used to be more popular in pre-ANSI C days since functions are usually prototyped separately at the start of the program. This is still perfectly legal though: order they appear in the function definition.

e.g.

```
int length, strlen(char []);
```

Here `length` is a variable, `strlen` the function as before.

Exercises

Exercise 12346

Write a function ``replace'' which takes a pointer to a string as a parameter, which replaces all spaces in that string by minus signs, and delivers the number of spaces it replaced.

Thus

```
        char *cat = "The cat sat";
        n = replace( cat );
```

should set

```
        cat to "The-cat-sat"
```

and

```
        n to 2.
```

Exercise 12347

Write a program which will read in the source of a C program from its standard input, and print out all the starred items in the following statistics for the program (all as integers). (Note the comment on tab characters at the end of this specification.)

Print out the following values:

```
  Lines:
  *  The total number of lines
  *  The total number of blank lines
        (Any lines consisting entirely of white space should be
        considered as blank lines.)
     The percentage of blank lines (100 * blank_lines / lines)

  Characters:
  *  The total number of characters after tab expansion
  *  The total number of spaces after tab expansion
  *  The total number of leading spaces after tab expansion
      (These are the spaces at the start of a line, before any visible
        character; ignore them if there are no visible characters.)
    The average number of
      characters per line
      characters per line ignoring leading spaces
      leading spaces per line
      spaces per line ignoring leading spaces

  Comments:
  *  The total number of comments in the program
  *  The total number of characters in the comments in the program
       excluding the "/*" and "*/" thenselves
    The percentage of number of comments to total lines
    The percentage of characters in comments to characters

  Identifiers:
    We are concerned with all the occurrences of "identifiers" in the
      program where each part of the text starting with a letter,
      and continuing with letter, digits and underscores is considered
      to be an identifier, provided that it is not
          in a comment,
          or in a string,
          or within primes.
        Note that
            "abc\"def"
        the internal escaped quote does not close the string.
        Also, the representation of the escape character is
            '\\'
 and of prime is
            '\''
      Do not attempt to exclude the fixed words of the language,
      treat them as identifiers. Print
  *  The total number of identifier occurrences.
  *  The total number of characters in them.
    The average identifier length.
```

```
  Indenting:
  *  The total number of times either of the following occurs:
      a line containing a "}" is more indented than the preceding line
      a line is preceded by a line containing a "{" and is less
        indented than it.
      The "{" and "}" must be ignored if in a comment or string or
        primes, or if the other line involved is entirely comment.
    A single count of the sum of both types of error is required.
```

NOTE: All tab characters (") on input should be interpreted as multiple spaces using the rule:

```
  "move to the next modulo 8 column"
  where the first column is numbered column 0.
 col before tab | col after tab
        ---------------+--------------
                0      |      8
                1      |      8
                7      |      8
                8      |     16
                9      |     16
               15      |     16
               16      |     24
```

To read input a character at a time the skeleton has code incorporated to read a line at a time for you using

```
        char ch;
        ch = getchar();
```

Which will deliver each character exactly as read. The "getline" function then puts the line just read in the global array of characters "linec", null terminated, and delivers the length of the line, or a negative value if end of data has been encountered.

You can then look at the characters just read with (for example)

```
        switch( linec[0] ) {
        case ' ': /* space ..... */
                break;
        case '\t': /* tab character .... */
                break;
        case '\n': /* newline ... */
                break;
        ....
        } /* end switch */
```

End of data is indicated by scanf NOT delivering the value 1.

Your output should be in the following style:

```
        Total lines                         126
        Total blank lines                   3
        Total characters                    3897
        Total spaces                        1844
        Total leading spaces                1180
        Total comments                      7
        Total chars in comments             234
        Total number of identifiers         132
        Total length of identifiers         606
        Total indenting errors              2
```

You may gather that the above program (together with the unstarred items) forms the basis of part of your marking system! Do the easy bits first, and leave it at that if some aspects worry you. Come back to me if you think my solution (or the specification) is wrong! That is quite possible!

Exercise 12348

It's rates of pay again!

Loop performing the following operation in your program:

Read two integers, representing a rate of pay (pence per hour) and a number of hours. Print out the total pay, with hours up to 40 being paid at basic rate, from 40 to 60 at rate-and-a-half, above 60 at double-rate. Print the pay as pounds to two decimal places.

Terminate the loop when a zero rate is encountered. At the end of the loop, print out the total pay.

The code for computing the pay from the rate and hours is to be written as a function.

The recommended output format is something like:

```
Pay at 200 pence/hr for 38 hours is 76.00 pounds
Pay at 220 pence/hr for 48 hours is 114.40 pounds
Pay at 240 pence/hr for 68 hours is 206.40 pounds
Pay at 260 pence/hr for 48 hours is 135.20 pounds
Pay at 280 pence/hr for 68 hours is 240.80 pounds
Pay at 300 pence/hr for 48 hours is 156.00 pounds
Total pay is 928.80 pounds
```

The ``program features'' checks that explicit values such as 40 and 60 appear only once, as a `#define` or initialised variable value. This represents good programming practice.

Dave Marshall
1/5/1999

Subsections

Further Data Types

This Chapter discusses how more advanced data types and structures can be created and used in a C program.

Structures

Structures in C are similar to records in Pascal. For example:

```
  struct gun
                              {
                              char name[50];
                              int magazinesize;
                              float calibre;
                              };

                struct gun arnies;
```

defines a new structure `gun` and makes `arnies` an instance of it.

NOTE: that `gun` is a *tag* for the structure that serves as shorthand for future declarations. We now only need to say `struct gun` and the body of the structure is implied as we do to make the `arnies` variable. The tag is *optional*.

Variables can also be declared between the `}` and `;` of a struct declaration, *i.e.*:

```
  struct gun
                              {
                              char name[50];
                              int magazinesize;
                              float calibre;
                              } arnies;
```

`struct`'s can be pre-initialised at declaration:

```
  struct gun arnies={"Uzi",30,7};
```

which gives arnie a 7mm. Uzi with 30 rounds of ammunition.

To access a member (or field) of a struct, C provides the . operator. For example, to give arnie more rounds of ammunition:

```
arnies.magazineSize=100;
```

Defining New Data Types

typedef can also be used with structures. The following creates a new type agun which is of type struct gun and can be initialised as usual:

```
typedef struct gun
                              {
                              char name[50];
                              int magazinesize;
                              float calibre;
                              } agun;

               agun arnies={"Uzi",30,7};
```

Here gun still acts as a *tag* to the struct and is optional. Indeed since we have defined a new data type it is not really of much use,

agun is the new data type. arnies is a variable of type agun which is a structure.

C also allows arrays of structures:

```
typedef struct gun
                              {
                              char name[50];
                              int magazinesize;
                              float calibre;
                              } agun;

              agun arniesguns[1000];
```

This gives arniesguns a 1000 guns. This may be used in the following way:

```
arniesguns[50].calibre=100;
```

gives Arnie's gun number 50 a calibre of 100mm, and:

```
itscalibre=arniesguns[0].calibre;
```

assigns the calibre of Arnie's first gun to itscalibre.

Unions

A union is a variable which may hold (at different times) objects of different sizes and types. C uses the `union` statement to create unions, for example:

```
union number
                        {
                        short shortnumber;
                        long longnumber;
                        double floatnumber;
                        } anumber
```

defines a union called `number` and an instance of it called `anumber`. `number` is a union *tag* and acts in the same way as a tag for a structure.

Members can be accessed in the following way:

```
printf("%ld\n",anumber.longnumber);
```

This clearly displays the value of longnumber.

When the C compiler is allocating memory for unions it will always reserve enough room for the largest member (in the above example this is 8 bytes for the double).

In order that the program can keep track of the type of union variable being used at a given time it is common to have a structure (with union embedded in it) and a variable which flags the union type:

An example is:

```
  typedef struct
                        { int maxpassengers;
                        } jet;

              typedef struct
                        { int liftcapacity;
                        } helicopter;

              typedef struct
                        { int maxpayload;
                        } cargoplane;

               typedef        union
                        { jet jetu;
                                helicopter helicopteru;
                                cargoplane cargoplaneu;
                        } aircraft;

              typedef         struct
                        { aircrafttype kind;
                                int speed;
                                aircraft description;
```

```
} an_aircraft;
```

This example defines a base union aircraft which may either be jet, helicopter, or cargoplane.

In the an_aircraft structure there is a kind member which indicates which structure is being held at the time.

Coercion or Type-Casting

C is one of the few languages to allow *coercion*, that is forcing one variable of one type to be another type. C allows this using the cast operator `()`. So:

```
int integernumber;
        float floatnumber=9.87;

                integernumber=(int)floatnumber;
```

assigns 9 (the fractional part is thrown away) to `integernumber`.

And:

```
int integernumber=10;
        float floatnumber;

                floatnumber=(float)integernumber;
```

assigns 10.0 to floatnumber.

Coercion can be used with any of the simple data types including char, so:

```
    int integernumber;
        char letter='A';

                integernumber=(int)letter;
```

assigns 65 (the ASCII code for `A') to integernumber.

Some typecasting is done automatically -- this is mainly with integer compatibility.

A good rule to follow is: **If in doubt cast**.

Another use is the make sure division behaves as requested: If we have two integers internumber and anotherint and we want the answer to be a float then :

```
e.g.
 floatnumber =
        (float) internumber / (float) anotherint;
```

```
ensures floating point division.
```

Enumerated Types

Enumerated types contain a list of constants that can be addressed in integer values.

We can declare types and variables as follows.

```
enum days {mon, tues, ..., sun} week;

enum days week1, week2;
```

NOTE: As with arrays first enumerated name has index value 0. So `mon` has value 0, `tues` 1, and so on.

`week1` and `week2` are variables.

We can define other values:

```
enum escapes { bell = `\a',

               backspace = `\b',  tab = `\t',

               newline = `\n', vtab = `\v',

               return = `\r'};
```

We can also override the 0 start value:

```
enum months {jan = 1, feb, mar, ......, dec};
```

Here it is implied that feb = 2 *etc.*

Static Variables

A **`static`** variable is local to particular function. However, it is only initialised once (on the first call to function).

Also the value of the variable on leaving the function remains **intact**. On the next call to the function the the `static` variable has the same value as on leaving.

To define a `static` variable simply prefix the variable declaration with the `static` keyword. For example:

```
void stat(); /* prototype fn */

               main()
                { int i;

                                for (i=0;i<5;++i)
```

```
                                        stat();
                }


stat()
                {            int auto_var = 0;
                              static int static_var = 0;

                              printf( ``auto = %d, static = %d \n'',

                                              auto_var, static_var);
                              ++auto_var;
                              ++static_var;
                }
```

Output is:

```
  auto_var = 0, static_var= 0
                auto_var = 0, static_var = 1
                auto_var = 0, static_var = 2
                auto_var = 0, static_var = 3
                auto_var = 0, static_var = 4
```

```
Clearly the auto_var variable is created each time. The static_var is
created once and remembers its value.
```

Exercises

Exercise 12386

Write program using enumerated types which when given today's date will print out tomorrow's date in the for 31st January, for example.

Exercise 12387

Write a simple database program that will store a persons details such as age, date of birth, address *etc.*

Dave Marshall
1/5/1999

Subsections

Pointers

Pointer are a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers.

C uses *pointers* a lot. **Why?**:

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

C uses pointers explicitly with:

- Arrays,
- Structures,
- Functions.

NOTE: Pointers are perhaps the most difficult part of C to understand. C's implementation is slightly different DIFFERENT from other languages.

What is a Pointer?

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The *unary* or *monadic* operator **&** gives the ``address of a variable".

The *indirection* or dereference operator ***** gives the ``contents of an object *pointed to* by a pointer".

To declare a pointer to a variable do:

```
int *pointer;
```

NOTE: We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**, for instance.

Consider the effect of the following code:

```
int x = 1, y = 2;
              int *ip;

              ip = &x;
```

```
y = *ip;
```

```
x = ip;
```

```
                  *ip = 3;
```

It is worth considering what is going on at the *machine level* in memory to fully understand how pointer work. Consider Fig. 9.1. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000. **Note** A pointer is a variable and thus its values need to be stored somewhere. It is the nature of the pointers value that is *new*.

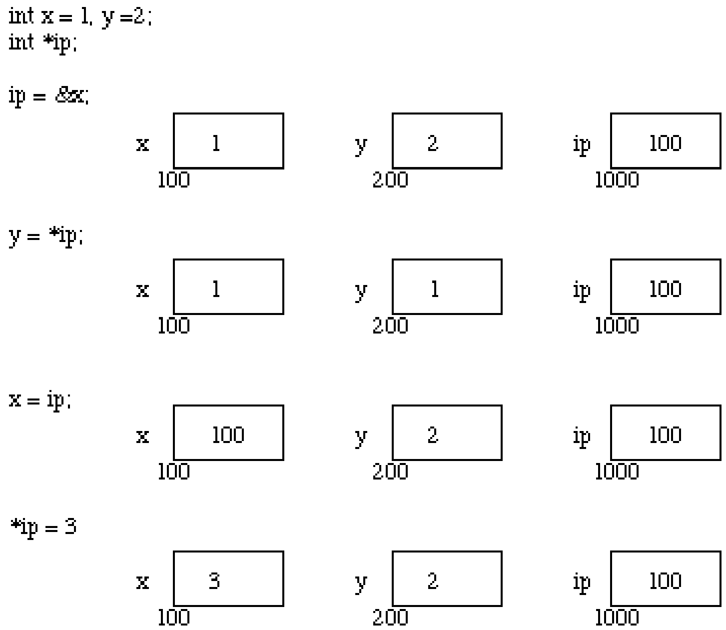


Fig. 9.1 Pointer, Variables and Memory Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be a *pointer to an integer* and is assigned to the address of x (&x). So ip gets loaded with the value 100.

Next y gets assigned to the *contents of* ip. In this example ip currently *points* to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1.

We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly **legal** (although not all that common) to assign the current value of ip to x. The value of ip at this instant is 100.

Finally we can assign a value to the contents of a pointer (*ip).

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.

So ...

```
   int *ip;

                  *ip = 100;
```

will generate an error (program crash!!).

The correct use is:

```
int *ip;
               int x;

               ip = &x;
               *ip = 100;
```

We can do integer arithmetic on a pointer:

```
float *flp, *flq;

               *flp = *flp + 10;

               ++*flp;

               (*flp)++;

               flq = flp;
```

NOTE: A pointer to any variable type is an address in memory -- which is an integer address. A pointer is definitely NOT an integer.

The reason we associate a pointer to a data type is so that it knows how many bytes the data is stored in. When we increment a pointer we increase the pointer by one ``block'' memory.

So for a character pointer ++ch_ptr adds 1 byte to the address.

For an integer or float ++ip or ++flp adds 4 bytes to the address.

Consider a float variable (fl) and a pointer to a float (flp) as shown in Fig. 9.2.

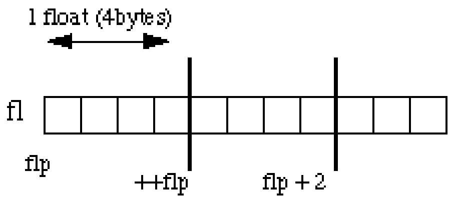


Fig. 9.2 Pointer Arithmetic Assume that flp points to fl then if we increment the pointer (++flp) it moves to the position shown 4 bytes on. If on the other hand we added 2 to the pointer then it moves 2 **float positions** *i.e* 8 bytes as shown in the Figure.

Pointer and Functions

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. Other languages do this (*e.g.* `var` parameters in PASCAL). C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.

The best way to study this is to look at an example where we must be able to receive changed parameters.

Let us try and write a function to swap variables around?

The usual function *call*:

`swap(a, b)` WON'T WORK.

Pointers provide the solution: *Pass the address of the variables to the functions and access address of function.*

Thus our function call in our program would look like this:

`swap(&a, &b)`

The Code to swap is fairly straightforward:

```
    void swap(int *px, int *py)

                    { int temp;

                                  temp = *px;
                                  /* contents of pointer */

                                  *px = *py;
                                  *py = temp;
                 }
```

We can return pointer from functions. A common example is when passing back structures. *e.g.*:

```
typedef struct {float x,y,z;} COORD;

                 main()

                  {  COORD p1, *coord_fn();
                                              /* declare fn to return ptr of
                                               COORD type */

                                               ....
                                               p1 = *coord_fn(...);
                                  /* assign contents of address returned */
                                               ....
                                  }

   COORD *coord_fn(...)

                  {  COORD p;

                                               .....
                                               p = ....;
                                               /* assign structure values */

                                               return &p;
                                               /* return address of p */
                                  }
```

Here we return a pointer whose contents are immediately *unwrapped* into a variable. We must do this straight away as the variable we pointed to was local to a function that has now finished. This means that the address space is free and can be overwritten. It will not have been overwritten straight after the function ha squit though so this is perfectly safe.

Pointers and Arrays

Pointers and arrays are very closely linked in C.

Hint: think of array elements arranged in consecutive memory locations.

Consider the following:

```
int a[10], x;
                int *pa;

                pa = &a[0];  /* pa pointer to address of a[0] */

                x = *pa;
                /* x = contents of pa (a[0] in this case) */
```

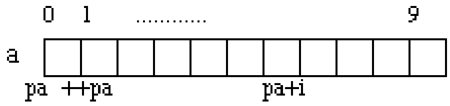


Fig. 9.3 Arrays and Pointers

To get somewhere in the array (Fig. 9.3) using a pointer we could do:

`pa + i` $\equiv$ `a[i]`

WARNING: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more subtle in its link between arrays and pointers.

For example we can just type

```
pa = a;
```

instead of

```
pa = &a[0]
```

and

`a[i]` can be written as `*(a + i)`.
i.e. `&a[i]` $\equiv$ `a + i`.

We also express pointer addressing like this:

`pa[i]` $\equiv$ `*(pa + i)`.

However pointers and arrays are different:

- A pointer is a variable. We can do
  pa = a and pa++.
- An Array <u>is not</u> a variable. a = pa and a++ ARE ILLEGAL.

This stuff is very important. Make sure you understand it. We will see a lot more of this.

We can now understand how arrays are passed to functions.

When an array is passed to a function what is actually passed is its initial elements location in memory.

So:

`strlen(s)` $\equiv$ `strlen(&s[0])`

This is why we declare the function:

```
int strlen(char s[]);
```

An equivalent declaration is : int strlen(char *s);
since char s[] $\equiv$ char *s.

strlen() is a *standard library* function (Chapter 18) that returns the length of a string. Let's look at how we may write a function:

```
int strlen(char *s)
                { char *p = s;

                              while (*p != `\0');

                                p++;
                              return p-s;
                }
```

Now lets write a function to copy a string to another string. strcpy() is a standard library function that does this.

```
void strcpy(char *s, char *t)
                {  while ( (*s++ = *t++) != `\0');}
```

This uses pointers and assignment by value.

Very Neat!!

NOTE: Uses of Null statements with while.

Arrays of Pointers

We can have arrays of pointers since pointers are variables.

Example use:

Sort lines of text of different length.

NOTE: Text can't be moved or compared in a single operation.

Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.

How can we do this?:

- Store lines end-to-end in one big `char` array (Fig. 9.4). \n will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using `strcmp()` standard library function.
- If 2 lines are out of order -- swap pointer in pointer array (not text).

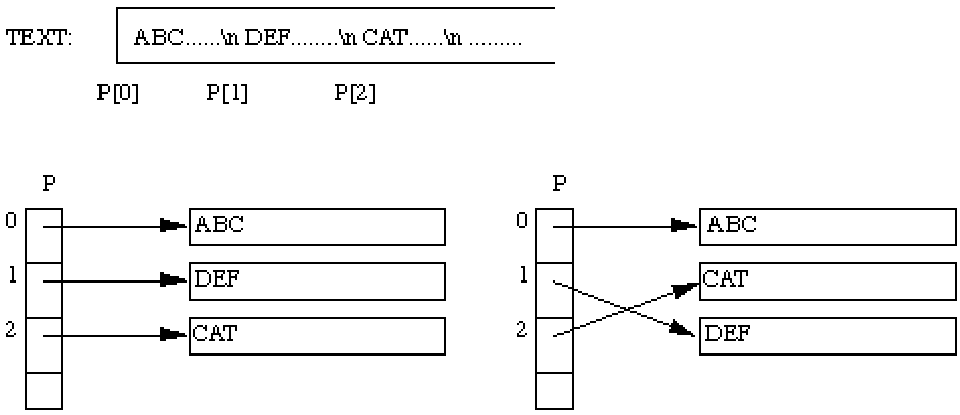


Fig. 9.4 Arrays of Pointers (String Sorting Example)

This eliminates:

- complicated storage management.
- high overheads of moving lines.

Multidimensional arrays and pointers

We should think of multidimensional arrays in a different way in C:

A 2D array is really a 1D array, each of whose elements is itself an array

Hence

`a[n][m]` notation.

Array elements are stored row by row.

When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant.

The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.

Consider`int a[5][35]` to be passed in a function:

We can do:

```
f(int a[][35]) {.....}
```

or even:

```
f(int (*a)[35]) {.....}
```

We need parenthesis (*a) since [] have a higher precedence than *

So:

`int (*a)[35];` declares a pointer to an array of 35 `int`s.

`int *a[35];` declares an array of 35 pointers to `int`s.

Now lets look at the (subtle) difference between pointers and arrays. Strings are a common application of this.

Consider:

```
char *name[10];

char Aname[10][20];
```

We can legally do `name[3][4]` and `Aname[3][4]` in C.

However

- `Aname` is a true 200 element 2D char array.
- access elements via
  *20*row + col + base_address*
  in memory.
- `name` has 10 pointer elements.

NOTE: If each pointer in `name` is set to point to a 20 element array then and only then will 200 chars be set aside (+ 10 elements).

The advantage of the latter is that each pointer can point to arrays be of different length.

Consider:

```
   char *name[] = { ``no month'', ``jan'',
                                 ``feb'', ... };
              char Aname[][15] = { ``no month'', ``jan'',
                                 ``feb'', ... };
```

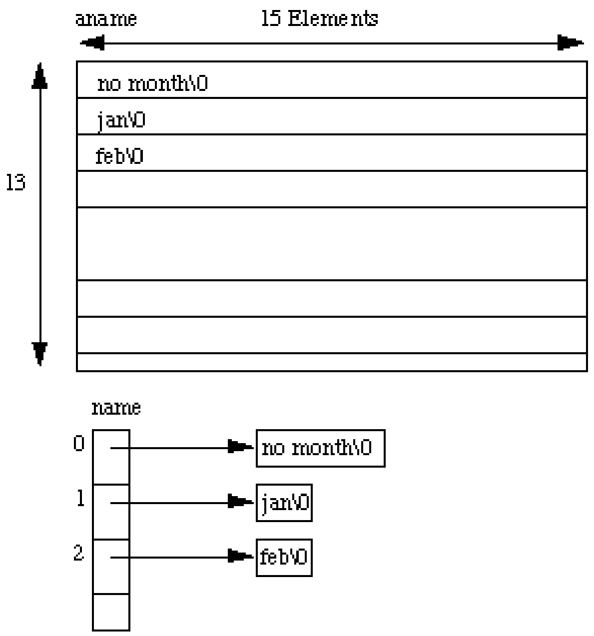


Fig. 2D Arrays and Arrays of Pointers

Static Initialisation of Pointer Arrays

Initialisation of arrays of pointers is an ideal application for an internal static array.

```
some_fn()
                  { static char *months = { ``no month'',
                                              ``jan'', ``feb'',
                                              ...};

                                 }
```

`static reserves a private permanent bit of memory.`

Pointers and Structures

These are fairly straight forward and are easily defined. Consider the following:

```
   struct COORD {float x,y,z;} pt;
                 struct COORD *pt_ptr;
```

```
pt_ptr = &pt; /* assigns pointer to pt */
```

the -> operator lets us access a member of the structure pointed to by a pointer.*i.e.*:

```
 pt_ptr->x = 1.0;

 pt_ptr->y = pt_ptr->y - 3.0;
```

Example: Linked Lists

```
  typedef struct {  int value;
                                         ELEMENT *next;
                        } ELEMENT;
```

```
ELEMENT n1, n2;
```

```
n1.next = &n2;
```

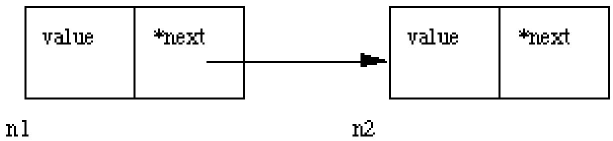


Fig. **Linking Two Nodes NOTE:** We can only declare `next` as a pointer to `ELEMENT`. We cannot have a element of the variable type as this would set up a *recursive* definition which is **NOT ALLOWED**. We are allowed to set a pointer reference since 4 bytes are set aside for any pointer.

The above code links a node n1 to n2 (Fig. 9.6) we will look at this matter further in the next Chapter.

Common Pointer Pitfalls

Here we will highlight two common mistakes made with pointers.

Not assigning a pointer to memory address before using it

```
  int *x;
```

```
                  *x = 100;
```

we need a physical location say: `int y;`

```
                  x = &y;
                  *x = 100;
```

This may be hard to spot. **NO COMPILER ERROR**. Also x could some random address at initialisation.

Illegal indirection

Suppose we have a function `malloc()` which tries to allocate memory dynamically (at run time) and returns a pointer to block of memory requested if successful or a `NULL` pointer
otherwise.

`char *malloc()` -- a standard library function (see later).

Let us have a pointer: `char *p`;

Consider:

`*p = (char *) malloc(100);` /* request 100 bytes of memory */

`*p = `y';`

There is mistake above. What is it?

No * in

`*p = (char *) malloc(100);`

Malloc returns a pointer. Also `p` does not point to any address.

The correct code should be:

`p = (char *) malloc(100);`

If code rectified one problem is if no memory is available and `p` is `NULL`. Therefore we can't do:
`*p = `y';`.

A good C program would check for this:

```
 p = (char *) malloc(100);
                if ( p == NULL)
                  { printf(``Error: Out of Memory \n'');
                                                   exit(1);
                                }
                *p = `y';
```

Exercise

Exercise 12453

Write a C program to read through an array of any type using pointers. Write a C program to scan through this array to find a particular value.

Exercise 12454

Write a program to find the number of times that a given word(i.e. a short string) occurs in a sentence (i.e. a long string!).

Read data from standard input. The first line is a single word, which is followed by general text on the second line. Read both up to a newline character, and insert a terminating null before processing.

Typical output should be:

```
         The word is "the".
         The sentence is "the cat sat on the mat".
        The word occurs 2 times.
```

Exercise 12455

Write a program that takes three variable (a, b, b) in as separate parameters and rotates the values stored so that value a goes to be, b, to c and c to a.

Dave Marshall
1/5/1999

Subsections

Dynamic Memory Allocation and Dynamic Structures

Dynamic allocation is a pretty unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

We will look at two common applications of this:

- dynamic arrays
- dynamic data structure *e.g.* linked lists

Malloc, Sizeof, and Free

The Function `malloc` is most commonly used to attempt to ``grab'' a continuous portion of memory. It is defined by:

```
void *malloc(size_t number_of_bytes)
```

That is to say it returns a pointer of type `void *` that is the start in memory of the reserved portion of size `number_of_bytes`. If memory cannot be allocated a `NULL` pointer is returned.

Since a `void *` is returned the C standard states that this pointer can be converted to any type. The `size_t` argument type is defined in `stdlib.h` and is an *unsigned type*.

So:

```
    char *cp;
                 cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to `cp`.

Also it is usual to use the sizeof() function to specify the number of bytes:

```
    int *ip;
                 ip = (int *) malloc(100*sizeof(int));
```

Some C compilers may require to cast the type of conversion. The (int *) means coercion to an integer pointer. Coercion to the correct pointer type is very important to ensure pointer arithmetic is performed correctly. I personally use it as a means of ensuring that I am totally correct in my coding and use cast all the time.

It is good practice to use sizeof() even if you know the actual size you want -- it makes for device independent (portable) code.

sizeof can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function.

SO:

```
   int i;
                 struct COORD {float x,y,z};
                 typedef struct COORD PT;
```

```
sizeof(int), sizeof(i),
sizeof(struct COORD) and
sizeof(PT) are all ACCEPTABLE
```

In the above we can use the link between pointers and arrays to treat the reserved memory like an array. *i.e* we can do things like:

```
ip[0] = 100;
```

or

```
for(i=0;i<100;++i) scanf("%d",ip++);
```

When you have finished using a portion of memory you should always free() it. This allows the memory *freed* to be aavailable again, possibly for further malloc() calls

The function free() takes a pointer as an argument and frees the memory to which the pointer refers.

Calloc and Realloc

There are two additional memory allocation functions, `Calloc()` and `Realloc()`. Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size};
```

```
void *realloc( void *ptr, size_t new_size);
```

`Malloc` does not initialise memory (to *zero*) in any way. If you wish to initialise memory then use `calloc`. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between `calloc` and `malloc` in that `calloc` takes the number of desired elements, `num_elements`, and element_size, `element_size`, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
int *ip;
          ip = (int *) calloc(100, sizeof(int));
```

Realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged.

If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You **must** use this new value. If new memory cannot be reallocated then realloc returns NULL.

Thus to change the size of memory allocated to the *ip pointer above to an array block of 50 integers instead of 100, simply do:

```
ip = (int *) calloc( ip, 50);
```

Linked Lists

Let us now return to our linked list example:

```
typedef struct {  int value;
                                    ELEMENT *next;
                         } ELEMENT;
```

We can now try to grow the list dynamically:

```
 link = (ELEMENT *) malloc(sizeof(ELEMENT));
```

`This will allocate memory for a new link.`

`If we want to deassign memory from a pointer use the free() function:`

```
  free(link)
```

`See Example programs (queue.c) below and try exercises for further practice.`

Full Program: `queue.c`

A queue is basically a special case of a linked list where one data element joins the list at the left end and leaves in a ordered fashion at the other end.

The full listing for `queue.c` is as follows:

```
/*                                                                  */
/* queue.c                                                          */
/* Demo of dynamic data structures in C                             */

#include <stdio.h>

#define FALSE 0
#define NULL 0

typedef struct {
    int     dataitem;
    struct listelement *link;
}               listelement;

void Menu (int *choice);
listelement * AddItem (listelement * listpointer, int data);
listelement * RemoveItem (listelement * listpointer);
void PrintQueue (listelement * listpointer);
void ClearQueue (listelement * listpointer);

main () {
    listelement listmember, *listpointer;
    int     data,
            choice;

    listpointer = NULL;
    do {
        Menu (&choice);
        switch (choice) {
            case 1:
                printf ("Enter data item value to add  ");
                scanf ("%d", &data);
                listpointer = AddItem (listpointer, data);
                break;
            case 2:
                if (listpointer == NULL)
                    printf ("Queue empty!\n");
                else
                    listpointer = RemoveItem (listpointer);
                break;
            case 3:
                PrintQueue (listpointer);
                break;
```

```
            case 4:
                break;

            default:
                printf ("Invalid menu choice - try again\n");
                break;
        }
    } while (choice != 4);
    ClearQueue (listpointer);
}                               /* main */

void Menu (int *choice) {

    char    local;

    printf ("\nEnter\t1 to add item,\n\t2 to remove item\n\
\t3 to print queue\n\t4 to quit\n");
    do {
        local = getchar ();
        if ((isdigit (local) == FALSE) && (local != '\n')) {
            printf ("\nyou must enter an integer.\n");
            printf ("Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n");
        }
    } while (isdigit ((unsigned char) local) == FALSE);
    *choice = (int) local - '0';
}

listelement * AddItem (listelement * listpointer, int data) {

    listelement * lp = listpointer;

    if (listpointer != NULL) {
        while (listpointer -> link != NULL)
            listpointer = listpointer -> link;
        listpointer -> link = (struct listelement  *) malloc (sizeof (listelement));
        listpointer = listpointer -> link;
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return lp;
    }
    else {
        listpointer = (struct listelement  *) malloc (sizeof (listelement));
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return listpointer;
    }
}

listelement * RemoveItem (listelement * listpointer) {

    listelement * tempp;
    printf ("Element removed is %d\n", listpointer -> dataitem);
    tempp = listpointer -> link;
    free (listpointer);
    return tempp;
}

void PrintQueue (listelement * listpointer) {

    if (listpointer == NULL)
        printf ("queue is empty!\n");
    else
        while (listpointer != NULL) {
```

```
            printf ("%d\t", listpointer -> dataitem);
            listpointer = listpointer -> link;
        }
    printf ("\n");
}

void ClearQueue (listelement * listpointer) {

    while (listpointer != NULL) {
        listpointer = RemoveItem (listpointer);
    }
}
```

Exercises

Exercise 12456

Write a program that reads a number that says how many <u>integer</u> numbers are to be stored in an array, creates an array to fit the <u>exact</u> size of the data and then reads in that many numbers into the array.

Exercise 12457

Write a program to implement the linked list as described in the notes above.

Exercise 12458

Write a program to sort a sequence of numbers using a binary tree (Using Pointers). A binary tree is a tree structure with only two (possible) branches from each node (Fig. 10.1). Each branch then represents a false or true decision. To sort numbers simply assign the left branch to take numbers less than the node number and the right branch any other number (greater than or equal to). To obtain a sorted list simply search the tree in a depth first fashion.

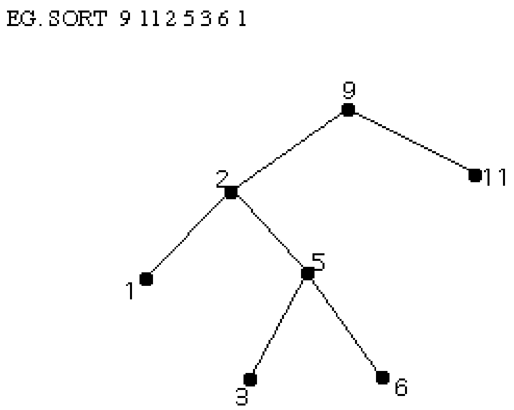


Fig. 10.1 Example of a binary tree sort Your program should: Create a binary tree structure. Create routines for loading the tree appropriately. Read in integer numbers terminated by a zero. Sort numbers into numeric ascending order. Print out the resulting ordered values, printing ten numbers per line as far as possible.

Typical output should be

```
    The sorted values are:
     2  4  6  6  7  9 10 11 11 11
    15 16 17 18 20 20 21 21 23 24
    27 28 29 30
```

Dave Marshall
1/5/1999

Subsections

Advanced Pointer Topics

We have introduced many applications and techniques that use pointers. We have introduced some advanced pointer issues already. This chapter brings together some topics we have briefly mentioned and others to complete our study C pointers.

In this chapter we will:

- Examine pointers to pointers in more detail.
- See how pointers are used in command line input in C.
- Study pointers to functions

Pointers to Pointers

We introduced the concept of a pointer to a pointer previously. You can have a pointer to a pointer of any type.

Consider the following:

```
char ch;  /* a character */
char *pch; /* a pointer to a character */
char **ppch; /* a pointer to a pointer to a character */
```

We can visualise this in Figure 11.1. Here we can see that `**ppch` refers to memory address of `*pch` which refers to the memory address of the variable `ch`. But what does this mean in practice?

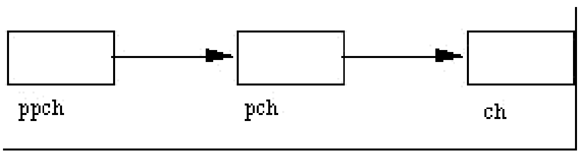


Fig. 11.1 Pointers to pointers Recall that `char *` refers to a (`NULL` terminated string. So one common and convenient notion is to declare a pointer to a pointer to a string (Figure 11.2)

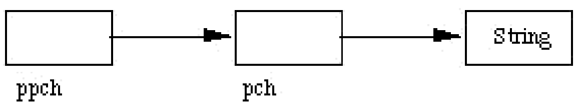


Fig. 11.2 Pointer to String Taking this one stage further we can have several strings being pointed to by the pointer (Figure 11.3)

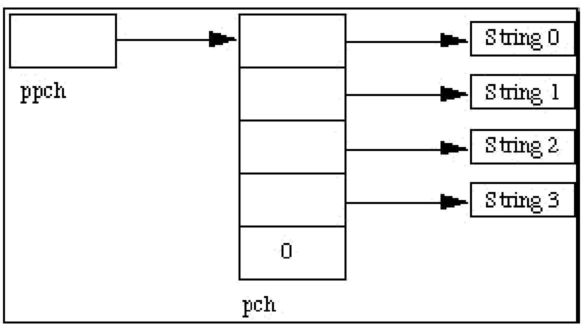


Fig. 11.3 Pointer to Several Strings We can refer to individual strings by `ppch[0], ppch[1], ....`. Thus this is identical to declaring `char *ppch[]`.

One common occurrence of this type is in C command line argument input which we now consider.

Command line input

C lets read arguments from the command line which can then be used in our programs.

We can type arguments after the program name when we run the program.

We have seen this with the compiler for example

```
c89 -o prog prog.c
```

`c89` is the program, `-o prog prog.c` the arguments.

In order to be able to use such arguments in our code we must define them as follows:

```
main(int argc, char **argv)
```

So our `main` function now has its own arguments. These are the only arguments main accepts.

- **argc** is the number of arguments typed -- including the program name.
- **argv** is an array of strings holding each command line argument -- including the program name in the first array element.

A simple program example:

```
#include<stdio.h>

main (int argc, char **argv)
    { /* program to print arguments
                                 from command line */

                                 int i;

                                 printf(``argc = %d\n\n'',argc);

                                 for (i=0;i<argc;++i)
                                                 printf(``argv[%d]: %s\n'',

                                                                 i, argv[i]);
                 }
```

Assume it is compiled to run it as args.

So if we type:

```
args f1 ``f2'' f3 4 stop!
```

```
The output would be:

   argc = 6

                 argv[0] = args
                 argv[1] = f1
                 argv[2] = f2
                 argv[3] = f3
                 argv[4] = 4
                 argv[5] = stop!
```

```
NOTE: • argv[0] is program name.
                • argc counts program name
                • Embedded `` '' are ignored.
                Blank spaces delimit end of arguments.
                Put blanks in `` '' if needed.
```

Pointers to a Function

Pointer to a function are perhaps on of the more confusing uses of pointers in C. Pointers to functions are not as common as other pointer uses. However, one common use is in a passing pointers to a function as a parameter in a function call. (Yes this is getting confusing, hold on to your hats for a moment).

This is especially useful when alternative functions maybe used to perform similar tasks on data. You can pass the data and the function to be used to some *control* function for instance. As we will see shortly the C standard library provided some basic sorting (`qsort`) and searching (`bsearch`) functions for free. You can easily embed your own functions.

To declare a pointer to a function do:

```
int (*pf) ();
```

This simply declares a pointer `*pf` to function that returns and `int`. No actual function is *pointed* to yet.

If we have a function `int f()` then we may simply (!!) write:

```
pf = &f;
```

For compiler prototyping to fully work it is better to have full function prototypes for the function and the pointer to a function:

```
int f(int);
int (*pf) (int) = &f;
```

Now `f()` returns an `int` and takes one `int` as a parameter.

You can do things like:

```
ans = f(5);
ans = pf(5);
```

which are equivalent.

The `qsort` standard library function is very useful function that is designed to sort an array by a *key* value of *any type* into ascending order, as long as the elements of the array are of fixed type.

qsort is prototyped in (`stdlib.h`):

```
void qsort(void *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

The argument `base` points to the array to be sorted, `num_elements` indicates how long the array is, `element_size` is the size in bytes of each array element and the final argument `compare` is a pointer to a function.

`qsort` calls the `compare` function which is user defined to compare the data when sorting. Note that `qsort` maintains it's data type independence by giving the comparison responsibility to the user. The compare function must return certain (`integer`) values according to the comparison result:

less than zero
: if first value is less than the second value

zero
: if first value is equal to the second value

greater than zero

: if first value is greater than the second value

Some quite complicated data structures can be sorted in this manner. For example, to sort the following structure by `integer` key:

```
typedef struct {
        int   key;
                                                                        struct other_data;
} Record;
```

We can write a compare function, `record_compare`:

```
int record\_compare(void const *a, void  const *a)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Assuming that we have an `array` of `array_length Record`s suitably filled with date we can call `qsort` like this:

```
qsort( array, arraylength, sizeof(Record), record_compare);
```

Further examples of standard library and system calls that use pointers to functions may be found in Chapters 15.4 and 19.1.

Exercises

Exercise 12476

Write a program last that prints the last n lines of its text input. By default n should be 5, but your program should allow an optional argument so that

```
last -n
```

prints out the last n lines, where n is any integer. Your program should make the best use of available storage. (Input of text could be by reading a file specified from the command or reading a file from standard input)

Exercise 12477

Write a program that sorts a list of integers in ascending order. However if a -r flag is present on the command line your program should sort the list in descending order. (You may use any sorting routine you wish)

Exercise 12478

Write a program that reads the following structure and sorts the data by keyword using `qsort`

```
typedef struct {
        char   keyword[10];
                                                                        int    other_data;
} Record;
```

Exercise 12479

An *insertion sort* is performed by adding values to an array one by one. The first value is simply stored at the beginning of the array. Each subsequent value is added by finding its ordered position in the array, moving data as needed to accommodate the value and inserting the value in this position.

Write a function called `insort` that performs this task and behaves in the same manner as `qsort`, *i.e* it can sort an array by a *key* value of *any type* and it has similar prototyping.

Dave Marshall
1/5/1999

Subsections

Low Level Operators and Bit Fields

We have seen how pointers give us control over low level memory operations.

Many programs (*e.g.* systems type applications) must actually operate at a low level where individual bytes must be operated on.

NOTE: The combination of pointers and bit-level operators makes C useful for many low level applications and can almost replace assembly code. (Only about 10 % of UNIX is assembly code the rest is C!!.)

Bitwise Operators

The *bitwise* operators of C a summarised in the following table:

Table: Bitwise operators

&	AND
$\mid$	OR
$\wedge$	XOR
$\sim$	One's Compliment
	$0 \rightarrow 1$
	$1 \rightarrow 0$
<<	Left shift
>>	Right Shift

DO NOT confuse & with &&: & is bitwise AND, && logical AND. Similarly for $\mid$ and $\mid\mid$.

$\sim$ is a unary operator -- it only operates on one argument to right of the operator.

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (*i.e.* There is **NO** wrap around).

For example: x << 2 shifts the bits in x by 2 places to the left.

So:

if x = 00000010 (binary) or 2 (decimal)

then:

$x >>= 2 \Rightarrow x = 00000000$ or 0 (decimal)

Also: if x = 00000010 (binary) or 2 (decimal)

$x <<= 2 \Rightarrow x = 00001000$ or 8 (decimal)

Therefore a shift left is equivalent to a multiplication by 2.

Similarly a shift right is equal to division by 2

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 *use shifts*.

To illustrate many points of bitwise operators let us write a function, `Bitcount`, that counts bits set to 1 in an 8 bit number (`unsigned char`) passed as an argument to the function.

```
int bitcount(unsigned char x)

                 { int count;

                                 for (count=0; x != 0; x>>=1);
                                                 if ( x & 01)
                                                                 count++;
                                 return count;
                 }
```

This function illustrates many C program points:

- `for loop` <u>`not`</u> `used for simple counting operation`
- `x`$>>= 1 \Rightarrow$ `x = x >> 1`
- `for loop will repeatedly shift right x until x becomes 0`
- `use expression evaluation of x & 01 to control if`
- `x & 01` *`masks`* `of 1st bit of x if this is 1 then count++`

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. *e.g.* 1 bit flags can be compacted -- Symbol tables in compilers.
- Reading external file formats -- non-standard file formats could be read in. *E.g.* 9 bit integers.

C lets us do this in a structure definition by putting :*bit length* after the variable. *i.e.*

```
struct packed_struct {
                 unsigned int f1:1;
                 unsigned int f2:1;
                 unsigned int f3:1;
                 unsigned int f4:1;
                 unsigned int type:4;
                 unsigned int funny_int:9;
} pack;
```

Here the `packed_struct contains 6 members: Four 1 bit` *`flags`* `f1..f3, a 4 bit type and a 9 bit funny_int.`

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word (see comments on bit fiels portability below).

Access members as usual via:

```
pack.type = 7;
```

NOTE:

- Only *n* lower bits will be assigned to an *n* bit number. So type cannot take values larger than 15 (4 bits long).
- Bit fields are always converted to integer type for computation.
- You are allowed to mix ``normal'' types with bit fields.
- The unsigned definition is important - ensures that no bits are used as a $\pm$ flag.

Bit Fields: Practical Example

Frequently device controllers (*e.g.* disk drives) and the operating system need to communicate at a low level. Device controllers contain several *registers* which may be packed together in one integer (Figure 12.1).

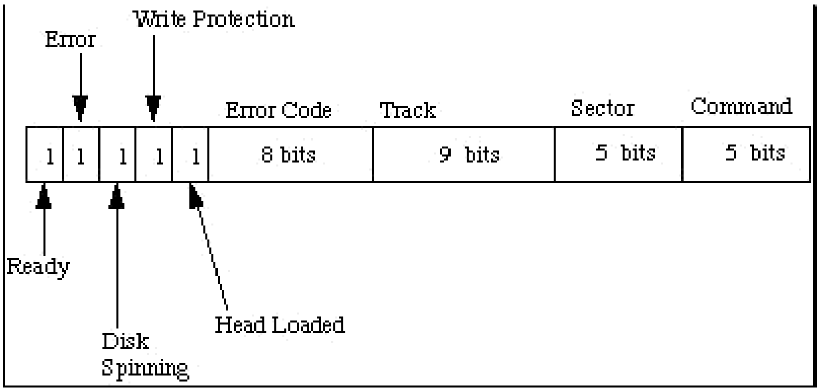


Fig. 12.1 Example Disk Controller Register We could define this register easily with bit fields:

```
struct DISK_REGISTER  {
     unsigned ready:1;
     unsigned error_occured:1;
     unsigned disk_spinning:1;
     unsigned write_protect:1;
     unsigned head_loaded:1;
     unsigned error_code:8;
     unsigned track:9;
     unsigned sector:5;
     unsigned command:5;
};
```

To access values stored at a particular memory address, `DISK_REGISTER_MEMORY` we can assign a pointer of the above structure to access the memory via:

```
struct DISK_REGISTER *disk_reg = (struct DISK_REGISTER *) DISK_REGISTER_MEMORY;
```

The disk driver code to access this is now relatively straightforward:

```
/* Define sector and track to start read */
```

```
disk_reg->sector = new_sector;
disk_reg->track = new_track;
disk_reg->command = READ;

/* wait until operation done, ready will be true */

while ( ! disk_reg->ready ) ;

/* check for errors */

if (disk_reg->error_occured)
  { /* interrogate disk_reg->error_code for error type */
    switch (disk_reg->error_code)
    ......
  }
```

A note of caution: Portability

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

- integers may be signed or unsigned
- Many compilers limit the maximum number of bits in the bit field to the size of an `integer` which may be either 16-bit or 32-bit varieties.
- Some bit field members are stored left to right others are stored right to left in memory.
- If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

If portability of code is a premium you can use bit shifting and masking to achieve the same results but not as easy to express or read. For example:

```
unsigned int  *disk_reg = (unsigned int *) DISK_REGISTER_MEMORY;

/* see if disk error occured */

disk_error_occured = (disk_reg & 0x40000000) >> 31;
```

Exercises

Exercise 12507

Write a function that prints out an 8-bit (unsigned char) number in binary format.

Exercise 12514

Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of an unsigned char variable y (leaving other bits unchanged).

E.g. if *x* = 10101010 (170 decimal) and *y* = 10100111 (167 decimal) and *n* = 3 and *p* = 6 say then you need to strip off 3 bits of y (111) and put them in x at position 10*xxx*010 to get answer 10111010.

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10101010 (binary)
y = 10100111 (binary)
setbits n = 3, p = 6 gives x = 10111010 (binary)
```

Exercise 12515

Write a function that inverts the bits of an unsigned char x and stores answer in y.

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10101010 (binary)
x inverted = 01010101 (binary)
```

Exercise 12516

Write a function that rotates (**NOT shifts**) to the right by n bit positions the bits of an unsigned char x.ie no bits are lost in this process.

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10100111 (binary)
x rotated by 3 = 11110100 (binary)
```

Note: All the functions developed should be as concise as possible

---

Dave Marshall
1/5/1999

Subsections

The C Preprocessor

Recall that preprocessing is the first step in the C program compilation stage -- this feature is unique to C compilers.

The preprocessor more or less provides its own language which can be a very powerful tool to the programmer. Recall that all preprocessor directives or commands begin with a #.

Use of the preprocessor is advantageous since it makes:

- programs easier to develop,
- easier to read,
- easier to modify
- C code more transportable between different machine architectures.

The preprocessor also lets us customise the language. For example to replace { ... } block statements delimiters by PASCAL like `begin ... end` we can do:

```
 #define begin {
              #define end  }
```

During compilation all occurrences of `begin and end get replaced by corresponding { or } and so the subsequent C compilation stage does not know any difference!!!.`

`Lets look at #define in more detail`

#define

Use this to define constants or any macro substitution. Use as follows:

```
#define  <macro> <replacement name>
```

For Example

```
#define FALSE  0
#define TRUE  !FALSE
```

We can also define small ``functions" using #define. For example max. of two variables:

```
#define max(A,B) ( (A) > (B) ? (A):(B))
```

? is the ternary operator in C.

Note: that this does not define a proper function max.

All it means that wherever we place max(C†,D†) the text

gets replaced by the appropriate definition. [† = any

variable names - not necessarily C and D]

So if in our C code we typed something like:

```
x = max(q+r,s+t);
```

after preprocessing, if we were able to look at the code it would appear like this:

```
x = ( (q+r) > (r+s) ? (q+r) : (s+t));
```

Other examples of #define could be:

```
#define Deg_to_Rad(X) (X*M_PI/180.0)
/* converts degrees to radians, M_PI is the value
of pi and is defined in math.h library */

#define LEFT_SHIFT_8 <<8
```

NOTE: The last macro LEFT_SHIFT_8 is only
valid so long as replacement context is valid *i.e.*
x = y LEFT_SHIFT_8.

#undef

This commands <u>undefined</u> a macro. A macro **must** be undefined before being redefined to a different value.

#include

This directive includes a file into code.

It has two possible forms:

```
#include <file>
```

or

```
#include ``file''
```

`<file>` tells the compiler to look where system include files are held. Usually UNIX systems store files in $\backslash usr \backslash include \backslash$ directory.

` ``file'' ` looks for a file in the current directory (where program was run from)

Included files usually contain C prototypes and declarations from header files and <u>not</u> (algorithmic) C code (SEE next Chapter for reasons)

#if -- Conditional inclusion

`#if` evaluates a constant integer expression. You <u>always</u> need a `#endif` to delimit end of statement.

We can have *else etc.* as well by using `#else` and `#elif` -- else if.

Another common use of `#if` is with:

`#ifdef`
: -- if defined and

`#ifndef`
: -- if not defined

These are useful for checking if macros are set -- perhaps from different program modules and header files.

For example, to set integer size for a portable C program between TurboC (on MSDOS) and Unix (or other) Operating systems. Recall that TurboC uses 16 bits/integer and UNIX 32 bits/integer.

Assume that if TurboC is running a macro `TURBOC` will be defined. So we just need to check for this:

```
#ifdef TURBOC
                 #define INT_SIZE 16
               #else
                               #define INT_SIZE  32
               #endif
```

As another example if running program on MSDOS machine we want to include file msdos.h otherwise a default.h file. A macro `SYSTEM is set (by OS) to type of system so check for this:`

```
#if SYSTEM == MSDOS
                 #include <msdos.h>
               #else
                               #include ``default.h''
              #endif
```

Preprocessor Compiler Control

You can use the `cc` compiler to control what values are set or defined from the command line. This gives some flexibility in setting customised values and has some other useful functions. The `-D` compiler option is used. For example:

```
cc -DLINELENGTH=80 prog.c -o prog
```

has the same effect as:

```
#define LINELENGTH 80
```

Note that any `#define` or `#undef` **within** the program (`prog.c` above) **override** command line settings.

You can also set a symbol without a value, for example:

```
cc -DDEBUG prog.c -o prog
```

Here the value is assumed to be 1.

The setting of such flags is useful, especially for debugging. You can put commands like:

```
#ifdef DEBUG
     print("Debugging: Program Version 1\");
#else
     print("Program Version 1 (Production)\");
#endif
```

Also since preprocessor command can be written anywhere in a C program you can filter out variables etc for printing *etc.* when debugging:

```
x = y *3;

#ifdef DEBUG
     print("Debugging: Variables (x,y) = \",x,y);
#endif
```

The `-E` command line is worth mentioning just for academic reasons. It is not that practical a command. The `-E command` will force the compiler to stop after the preprocessing stage and output the current state of your program. Apart from being debugging aid for preprocessor commands and also as a useful initial learning tool (try this option out with some of the examples above) it is not that commonly used.

Other Preprocessor Commands

There are few other preprocessor directives available:

`#error`

`text of error message` -- generates an appropriate compiler error message. *e.g*

```
#ifdef OS_MSDOS
                 #include <msdos.h>
               #elifdef OS_UNIX
                                #include ``default.h''
              #else
                               #error Wrong OS!!
              #endif
```

 # line

`number "string" -- informs the preprocessor that the number is the next number of line of input. "string" is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of`

```
the original source files instead of the intermediate C
(translated) source files.
```

Exercises

Exercise 12529

Define a preprocessor macro `swap(t, x, y)` that will swap two arguments `x` and `y` of a given type `t`.

Exercise 12531

Define a preprocessor macro to select:

- the least significant bit from an `unsigned char`
- the *n*th (assuming least significant is 0) bit from an `unsigned char`.

Dave Marshall
1/5/1999

Subsections

C, UNIX and Standard Libraries

There is a very close link between C and most operating systems that run our C programs. Almost the whole of the UNIX operating system is written in C. This Chapter will look at how C and UNIX interface together.

We have to use UNIX to maintain our file space, edit, compile and run programs *etc.*.

However UNIX is much more useful than this:

Advantages of using UNIX with C

- **Portability** -- UNIX, or a variety of UNIX, is available on many machines. Programs written in *standard* UNIX and C should run on any of them with little difficulty.
- **Multiuser / Multitasking** -- many programs can share a machines processing power.
- **File handling** -- hierarchical file system with many file handling routines.
- **Shell Programming** -- UNIX provides a powerful command interpreter that understands over 200 commands and can also run UNIX and user-defined programs.
- **Pipe** -- where the output of one program can be made the input of another. This can done from command line or within a C program.
- **UNIX utilities** -- there over 200 utilities that let you accomplish many routines without writing new programs. *e.g.* make, grep, diff, awk, more ....
- **System calls** -- UNIX has about 60 system calls that are at the *heart* of the operating system or the *kernel* of UNIX. The calls are actually written in C. All of them can be accessed from C programs. Basic I/0, system clock access are examples. The function `open()` is an example of a system call.
- **Library functions** -- additions to the operating system.

Using UNIX System Calls and Library Functions

To use system calls and library functions in a C program we simply call the appropriate C function.

Examples of standard library functions we have met include the higher level I/O functions -- `fprintf()`, `malloc()` ...

Aritmetic operators, random number generators -- `random(), srandom(), lrand48(), drand48()` *etc.* and basic C types to string conversion are memebers of the `stdlib.h` standard library.

All math functions such as `sin(), cos(), sqrt()` are standard math library (`math.h`) functions and others follow in a similar fashion.

For most system calls and library functions we have to include an appropriate header file. *e.g.* `stdio.h, math.h`

To use a function, ensure that you have made the required `#includes` in your C file. Then the function can be called as though you had defined it yourself.

It is important to ensure that your arguments have the expected types, otherwise the function will probably produce strange results. `lint` is quite good at checking such things.

Some libraries require extra options before the compiler can support their use. For example, to compile a program including functions from the `math.h` library the command might be

```
cc mathprog.c -o mathprog -lm
```

The final `-lm` is an instruction to link the maths library with the program. The manual page for each function will usually inform you if any special compiler flags are required.

Information on nearly all system calls and library functions is available in manual pages. These are available on line: Simply type `man` function name.

e.g. `man drand48`

would give information about this random number generator.

Over the coming chapters we will be investigating in detail many aspects of the C

Standard Library and also other UNIX libraries.

Dave Marshall
1/5/1999

Subsections

Integer Functions, Random Number, String Conversion, Searching and Sorting: `<stdlib.h>`

To use all functions in this library you must:

```
#include <stdlib.h>
```

There are three basic categories of functions:

- Arithmetic
- Random Numbers
- String Conversion

The use of all the functions is relatively straightforward. We only consider them briefly in turn in this Chapter.

Arithmetic Functions

There are 4 basic integer functions:

```
int abs(int number);
long int labs(long int number);

div_t div(int numerator,int denominator);
ldiv_t ldiv(long int numerator, long int denominator);
```

Essentially there are two functions with integer and long integer compatibility.

`abs`
: functions return the absolute value of its `number` arguments. For example, abs(2) returns 2 as does abs(-2).

`div`
: takes two arguments, `numerator` and `denominator` and produces a quotient and a remainder of the integer division. The `div_t` structure is defined (in `stdlib.h`) as follows:

```
typedef struct {
        int  quot; /* quotient */
        int  rem;  /* remainder */
} div_t;
```

(`ldiv_t` is similarly defined).

Thus:

```
#include <stdlib.h>
....

int num = 8, den = 3;
div_t ans;
```

```
ans = div(num,den);

printf("Answer:\n\t Quotient = %d\n\t Remainder = %d\n", \
ans.quot,ans.rem);
```

Produces the following output:

```
Answer:
                Quotient = 2
  Remainder = 2
```

Random Numbers

Random numbers are useful in programs that need to simulate random events, such as games, simulations and experimentations. In practice no functions produce truly random data -- they produce *pseudo-random* numbers. These are computed form a given formula (different generators use different formulae) and the number sequences they produce are repeatable. A *seed* is usually set from which the sequence is generated. Therefore is you set the same seed all the time the same set will be be computed.

One common technique to introduce further randomness into a random number generator is to use the time of the day to set the seed, as this will always be changing. (We will study the standard library time functions later in Chapter 20).

There are many (pseudo) random number functions in the standard library. They all operate on the same basic idea but generate different number sequences (based on different generator functions) over different number ranges.

The simplest set of functions is:

```
int rand(void);
void srand(unsigned int seed);
```

`rand()` returns successive pseudo-random numbers in the range from 0 to (2^15)-1.
`srand()` is used to set the seed. A simple example of using the time of the day to initiate a seed is via the call:

```
srand( (unsigned int) time( NULL ));
```

The following program `card.c` illustrates the use of these functions to simulate a pack of cards being shuffled:

```
/*
** Use random numbers to shuffle the "cards" in the deck.  The second
** argument indicates the number of cards.  The first time this
** function is called, srand is called to initialize the random
** number generator.
*/
#include <stdlib.h>
#include <time.h>
#define TRUE    1
#define FALSE   0

void shuffle( int *deck, int n_cards )
{
        int     i;
        static  int     first_time = TRUE;

        /*
        ** Seed the random number generator with the current time
        ** of day if we haven't done so yet.
        */
        if( first_time ){
                first_time = FALSE;
                srand( (unsigned int)time( NULL ) );
```

```
        }

        /*
        ** "Shuffle" by interchanging random pairs of cards.
        */
        for( i = n_cards - 1; i > 0; i -= 1 ){
                int     where;
                int     temp;

                where = rand() % i;
                temp = deck[ where ];
                deck[ where ] = deck[ i ];
                deck[ i ] = temp;
        }
}
```

There are several other random number generators available in the standard library:

```
double drand48(void);
double erand48(unsigned short xsubi[3]);
long lrand48(void);
long nrand48(unsigned short xsubi[3]);
long mrand48(void);
long jrand48(unsigned short xsubi[3]);
void srand48(long seed);
unsigned short *seed48(unsigned short seed[3]);
void lcong48(unsigned short param[7]);
```

This family of functions generates uniformly distributed pseudo-random numbers.

Functions drand48() and erand48() return non-negative double-precision floating-point values uniformly distributed over the interval `[0.0, 1.0)`.

Functions lrand48() and nrand48() return non-negative long integers uniformly distributed over the interval `[0, 2**31)`.

Functions mrand48() and jrand48() return signed long integers uniformly distributed over the interval `[-2**31, 2**31)`.

Functions srand48(), seed48(), and lcong48() set the seeds for drand48(), lrand48(), or mrand48() and one of these should be called first.

Further examples of using these functions is given is Chapter 20.

String Conversion

There are a few functions that exist to convert strings to integer, long integer and float values. They are:

`double atof(char *string)` -- Convert string to floating point value.
`int atoi(char *string)` -- Convert string to an integer value
`int atol(char *string)` -- Convert string to a long integer value.
`double strtod(char *string, char *endptr)` -- Convert string to a floating point value.
`long strtol(char *string, char *endptr, int radix)` -- Convert string to a long integer using a given radix.
`unsigned long strtoul(char *string, char *endptr, int radix)` -- Convert string to unsigned long.

Most of these are fairly straightforward to use. For example:

```
char *str1 = "100";
char *str2 = "55.444";
char *str3 = "      1234";
char *str4 = "123four";
```

```
char *str5 = "invalid123";

int i;
float f;

i = atoi(str1);  /* i = 100 */
f = atof(str2);  /* f = 55.44 */
i = atoi(str3);  /* i = 1234 */
i = atoi(str4);  /* i = 123 */
i = atoi(str5);  /* i = 0 */
```

Note:

- Leading blank characters are skipped.
- Trailing illegal characters are ignored.
- If conversion cannot be made zero is returned and `errno` (See Chapter 17) is set with the value `ERANGE`.

Searching and Sorting

The `stdlib.h` provides 2 useful functions to perform general searching and sorting of data on any type. In fact we have already introduced the `qsort()` function in Chapter 11.3. For completeness we list the prototype again here but refer the reader to the previous Chapter for an example.

The `qsort` standard library function is very useful function that is designed to sort an array by a *key* value of *any type* into ascending order, as long as the elements of the array are of fixed type.

qsort is prototyped (in `stdlib.h`):

```
void qsort(void *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

Similarly, there is a binary search function, `bsearch()` which is prototyped (in `stdlib.h`) as:

```
void *bsearch(const void *key, const void *base, size_t nel,
     size_t size, int (*compare)(const  void  *,  const  void *));
```

Using the same `Record` structure and `record_compare` function as the `qsort()` example (in Chapter 11.3):

```
typedef struct {
        int   key;
                                                                struct other_data;
} Record;

int record\_compare(void const *a, void  const *a)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Also, Assuming that we have an `array` of `array_length Record`s suitably filled with date we can call `bsearch()` like this:

```
Record key;
Record *ans;

key.key =  3; /* index value to be searched for */
ans = bsearch(&key, array, arraylength, sizeof(Record), record_compare);
```

The function `bsearch()` return a pointer to the field whose key filed is filled with the matched value of `NULL` if no match found.

Note that the type of the `key` argument **must** be the same as the array elements (`Record` above), even though only the `key.key` element is required to be set.

Exercises

Exercise 12534

Write a program that simulates throwing a six sided die

Exercise 12535

Write a program that simulates the UK National lottery by selecting six different whole numbers in the range 1 - 49.

Exercise 12536

Write a program that read a number from command line input and generates a random floating point number in the range 0 - the input number.

Dave Marshall
1/5/1999

Subsections

Mathematics: `<math.h>`

Mathematics is relatively straightforward library to use again. You **must** `#include <math.h>` and must **remember** to link in the math library at compilation:

```
 cc mathprog.c -o mathprog -lm
```

A common source of error is in forgetting to include the `<math.h>` file (and yes experienced programmers make this error also). Unfortunately the C compiler does not help much. Consider:

```
double x;
x = sqrt(63.9);
```

Having not seen the prototype for `sqrt` the compiler (by default) assumes that the function returns an `int` and converts the value to a `double` with meaningless results.

Math Functions

Below we list some common math functions. Apart from the note above they should be easy to use and we have already used some in previous examples. We give no further examples here:

`double acos(double x)` -- Compute arc cosine of x.
`double asin(double x)` -- Compute arc sine of x.
`double atan(double x)` -- Compute arc tangent of x.
`double atan2(double y, double x)` -- Compute arc tangent of y/x.
`double ceil(double x)` -- Get smallest integral value that exceeds x.
`double cos(double x)` -- Compute cosine of angle in radians.
`double cosh(double x)` -- Compute the hyperbolic cosine of x.
`div_t div(int number, int denom)` -- Divide one integer by another.
`double exp(double x` -- Compute exponential of x
`double fabs (double x )` -- Compute absolute value of x.
`double floor(double x)` -- Get largest integral value less than x.

`double fmod(double x, double y)` -- Divide x by y with integral quotient and return remainder.
`double frexp(double x, int *expptr)` -- Breaks down x into mantissa and exponent of no.
`labs(long n)` -- Find absolute value of long integer n.
`double ldexp(double x, int exp)` -- Reconstructs x out of mantissa and exponent of two.
`ldiv_t ldiv(long number, long denom)` -- Divide one long integer by another.
`double log(double x)` -- Compute log(x).
`double log10 (double x )` -- Compute log to the base 10 of x.
`double modf(double x, double *intptr)` -- Breaks x into fractional and integer parts.
`double pow (double x, double y)` -- Compute x raised to the power y.
`double sin(double x)` -- Compute sine of angle in radians.
`double sinh(double x)` - Compute the hyperbolic sine of x.
`double sqrt(double x)` -- Compute the square root of x.
`void srand(unsigned seed)` -- Set a new seed for the random number generator (rand).
`double tan(double x)` -- Compute tangent of angle in radians.
`double tanh(double x)` -- Compute the hyperbolic tangent of x.

Math Constants

The `math.h` library defines many (often neglected) constants. It is always advisable to use these definitions:

`HUGE` -- The maximum value of a single-precision floating-point number.

`M_E` -- The base of natural logarithms (e).

`M_LOG2E` -- The base-2 logarithm of e.

`M_LOG10E` - The base-10 logarithm of e.

`M_LN2` -- The natural logarithm of 2.

`M_LN10` -- The natural logarithm of 10.

`M_PI` -- π.

`M_PI_2` -- $\pi/2$.

`M_PI_4` -- $\pi/4$.

`M_1_PI` -- $1/\pi$.

`M_2_PI` -- $2/\pi$.

`M_2_SQRTPI` -- $2/\sqrt{\pi}$.

`M_SQRT2` -- The positive square root of 2.

`M_SQRT1_2` -- The positive square root of 1/2.

`MAXFLOAT` -- The maximum value of a non-infinite single- precision floating point number.

`HUGE_VAL` -- positive infinity.

There are also a number a machine dependent values defined in `#include <value.h>` -- see `man value` or list `value.h` for further details.

Dave Marshall
1/5/1999

Subsections

Input and Output (I/O):`stdio.h`

This chapter will look at many forms of I/O. We have briefly mentioned some forms before will look at these in much more detail here.

Your programs will need to include the standard I/O *header* file so do:

```
#include <stdio.h>
```

Reporting Errors

Many times it is useful to report errors in a C program. The standard library `perror()` is an easy to use and convenient function. It is used in conjunction with `errno` and frequently on encountering an error you may wish to terminate your program early. Whilst not strictly part of the `stdio.h` library we introduce the concept of `errno` and the function `exit()` here. We will meet these concepts in other parts of the Standard Library also.

`perror()`

The function `perror()` is prototyped by:

```
void perror(const char *message);
```

perror() produces a message (on standard error output -- see Section 17.2.1), describing the last error encountered, returned to `errno` (see below) during a call to a system or library function. The argument string `message` is printed first, then a colon and a blank, then the message and a newline. If `message` is a NULL pointer or points to a null string, the colon is not printed.

errno

`errno` is a special system variable that is set if a system call cannot perform its set task. It is defined in `#include <errno.h>`.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program (although this is uncommon practice) otherwise it simply retains its last value returned by a system call or library function.

exit()

The function `exit()` is prototyped in `#include <stdlib>` by:

`void exit(int status)`

Exit simply terminates the execution of a program and returns the exit `status` value to the operating system. The `status` value is used to indicate if the program has terminated properly:

- it exist with a `EXIT_SUCCESS` value on successful termination
- it exist with a `EXIT_FAILURE` value on unsuccessful termination.

On encountering an error you may frequently call an `exit(EXIT_FAILURE)` to terminate an errant program.

Streams

Streams are a portable way of reading and writing data. They provide a flexible and efficient means of I/O.

A Stream is a file or a physical device (*e.g.* printer or monitor) which is manipulated with a **pointer** to the stream.

There exists an internal C data structure, `FILE`, which represents all streams and is defined in `stdio.h`. We simply need to refer to the `FILE` structure in C programs when performing I/O with streams.

We just need to declare a variable or pointer of this type in our programs.

We do not need to know any more specifics about this definition.

We must open a stream before doing any I/O,

then access it

and then close it.

Stream I/O is **BUFFERED**: That is to say a fixed ``chunk'' is read from or written to a file via some temporary storage area (the buffer). This is illustrated in Fig. 17.1. NOTE the file pointer actually points to this buffer.

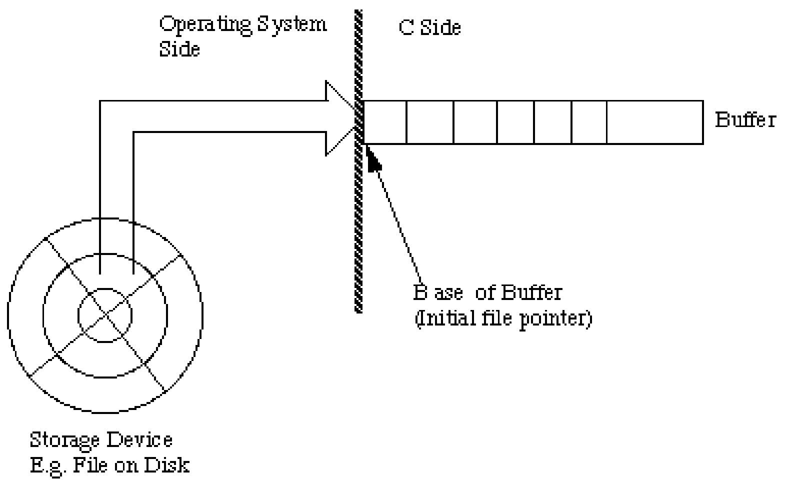


Fig. **Stream I/O Model** This leads to efficient I/O but **beware**: data written to a buffer does not appear in a file (or device) until the buffer is flushed or written out. (\n does this). Any abnormal exit of code can cause problems.

Predefined Streams

UNIX defines 3 predefined streams (in `stdio.h`):

`stdin,  stdout,  stderr`

They all use text a the method of I/O.

`stdin` and `stdout` can be used with files, programs, I/O devices such as keyboard, console, *etc.*. `stderr` always goes to the console or screen.

The console is the default for `stdout` and `stderr`. The keyboard is the default for `stdin`.

Predefined stream are automatically open.

Redirection

This how we override the UNIX default predefined I/O defaults.

This is not part of C but operating system dependent. We will do redirection from the command line.

> -- redirect `stdout` to a file.

So if we have a program, `out`, that usually prints to the screen then

```
out > file1
```

will send the output to a file, `file1`.

< -- redirect `stdin` from a file to a program.

So if we are expecting input from the keyboard for a program, `in` we can read similar input from a file

`in < file2`.

| -- *pipe*: puts `stdout` from one program to `stdin` of another

```
prog1 | prog2
```

e.g. Sent output (usually to console) of a program direct to printer:

```
out | lpr
```

Basic I/O

There are a couple of function that provide basic I/O facilities.

probably the most common are: `getchar()` and `putchar()`. They are defined and used as follows:

- `int getchar(void)` -- reads a char from `stdin`
- `int putchar(char ch)` -- writes a char to `stdout`, returns character written.

```
int ch;

                ch = getchar();
                (void) putchar((char) ch);
```

Related Functions:

```
 int getc(FILE *stream),
int putc(char ch,FILE *stream)
```

Formatted I/O

We have seen examples of how C uses formatted I/O already. Let's look at this in more detail.

Printf

The function is defined as follows:

`int printf(char *format, arg list ...)` --
prints to `stdout` the list of arguments according specified format string. Returns number of characters printed.

The **format string** has 2 types of object:

- *ordinary characters* -- these are copied to output.
- *conversion specifications* -- denoted by % and listed in Table 17.1.

Table: Printf/scanf format characters

Format Spec (%)	Type	Result
c	char	single character
i,d	int	decimal number
o	int	octal number
x,X	int	hexadecimal number
		lower/uppercase notation
u	int	unsigned int
s	char *	print string
		terminated by \0
f	double/float	format -m.ddd...
e,E	"	Scientific Format
		-1.23e002
g,G	"	e or f whichever
		is most compact
%	-	print % character

Between % and format char we can put:

- (minus sign)
: -- left justify.

integer number
: -- field width.

m.d
: -- m = field width, d = precision of number of digits after decimal point or number of chars from a string.

So:

```
printf("%-2.3f\n",17.23478);
```

The output on the screen is:

```
17.235
```

`and:`

```
printf("VAT=17.5%%\n");
```

`...outputs:`

```
VAT=17.5%
```

scanf

This function is defined as follows:

`int scanf(char *format, args....)` -- reads from stdin and puts input in address of variables specified in `args` list. Returns number of chars read.

Format control string similar to `printf`

Note: The ADDRESS of variable or a pointer to one is required by `scanf`.

```
scanf(``%d'',&i);
```

We can just give the name of an array or string to scanf since this corresponds to the start address of the array/string.

```
char string[80];
        scanf(``%s'',string);
```

Files

Files are the most common form of a stream.

The first thing we must do is *open* a file. The function `fopen()` does this:

```
FILE *fopen(char *name, char *mode)
```

`fopen` returns a pointer to a FILE. The `name` string is the name of the file on disc that we wish to access. The `mode` string controls our type of access. If a file cannot be accessed for any reason a `NULL` pointer is returned.

```
Modes include: ``r'' -- read,
                ``w'' -- write and
                ``a'' -- append.
```

To open a file we must have a stream (file pointer) that *points* to a `FILE` structure.

So to open a file, called *myfile.dat* for reading we would do:

```
FILE *stream, *fopen();
        /* declare a stream and prototype fopen */

        stream = fopen(``myfile.dat'',``r'');
```

```
it is good practice to to check file is opened
correctly:

                  if ( (stream = fopen( ``myfile.dat'',
                                              ``r'')) == NULL)
                                  {  printf(``Can't open %s\n'',

                                                ``myfile.dat'');
                                   exit(1);
                                  }
                  ......
```

Reading and writing files

The functions `fprintf` and `fscanf` a commonly used to access files.

```
int fprintf(FILE *stream, char *format, args..)
                  int fscanf(FILE *stream, char *format, args..)
```

These are similar to `printf and scanf except that data is read from the` *stream* `that must have been opened with fopen().`

The stream pointer is automatically incremented with ALL file read/write functions. We **do not** have to worry about doing this.

```
  char *string[80]
                  FILE *stream, *fopen();

                  if ( (stream = fopen(...)) != NULL)
                                  fscanf(stream,``%s'', string);
```

Other functions for files:

`int getc(FILE *stream), int fgetc(FILE *stream)`

`int putc(char ch, FILE *s), int fputc(char ch, FILE *s)`

These are like getchar, putchar.

getc is defined as preprocessor MACRO in stdio.h. fgetc is a C library function. Both achieve the same result!!

`fflush(FILE *stream) -- flushes a stream.`

`fclose(FILE *stream) -- closes a stream.`

We can access predefined streams with fprintf *etc.*

```
fprintf(stderr,``Cannot Compute!!\n'');

 fscanf(stdin,``%s'',string);
```

sprintf and sscanf

These are like `fprintf` and `fscanf` except they read/write to a string.

`int sprintf(char *string, char *format, args..)`

`int sscanf(char *string, char *format, args..)`

For Example:

```
  float full_tank = 47.0; /* litres */
                float miles = 300;
                char miles_per_litre[80];

                sprintf( miles_per_litre,``Miles per litre
                                = %2.3f'', miles/full_tank);
```

Stream Status Enquiries

There are a few useful stream enquiry functions, prototyped as follows:

```
 int feof(FILE *stream);
 int ferror(FILE *stream);
 void clearerr(FILE *stream);
 int fileno(FILE *stream);
```

Their use is relatively simple:

`feof()`

-- returns true if the stream is currently at the end of the file. So to read a stream,`fp`, line by line you could do:

```
while ( !feof(fp) )
  fscanf(fp,"%s",line);
```

`ferror()`

-- reports on the error state of the stream and returns true if an error has occurred.

`clearerr()`

-- resets the error indication for a given stream.

`fileno()`

-- returns the integer file descriptor associated with the named stream.

Low Level I/O

This form of I/O is UNBUFFERED -- each read/write request results in accessing disk (or device) directly to fetch/put a specific number of **bytes**.

There are no formatting facilities -- we are dealing with bytes of information.

This means we are now using binary (and not text) files.

Instead of file pointers we use *low level* `file handle or file descriptors` which give a unique integer number to identify each file.

To Open a file use:

`int open(char *filename, int flag, int perms)` -- this returns a file descriptor or -1 for a **fail**.

The `flag` controls file access and has the following predefined in `fcntl.h`:

`O_APPEND, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_WRONLY` + others see online `man` pages or reference manuals.

`perms` -- best set to 0 for most of our applications.

The function:

```
 creat(char *filename, int perms)
```

can also be used to create a file.

`int close(int handle)` -- close a file

```
int read(int handle, char *buffer,
unsigned length)
```

```
int write(int handle, char *buffer, unsigned length)
```

are used to read/write a specific number of bytes from/to a file (handle) stored or to be put in the memory location specified by `buffer`.

The `sizeof()` function is commonly used to specify the `length`.

read and write return the number of bytes read/written or -1 if they fail.

```
/* program to  read a list of floats from a binary file */
/* first byte of file is an integer saying how many  */
/* floats in file. Floats follow after it, File name got from */
/* command line */

#include<stdio.h>
#include<fcntl.h>

float bigbuff[1000];

main(int argc, char **argv)

{  int fd;
                 int bytes_read;
                 int file_length;

                 if ( (fd = open(argv[1],O_RDONLY)) = -1)
                                 { /* error file not open */....
                                                 perror("Datafile");
                                                 exit(1);
                                 }
                 if ( (bytes_read = read(fd,&file_length,
                                                 sizeof(int))) == -1)
                                 { /* error reading file */...
                                                 exit(1);
                                 }
                 if ( file_length > 999 )  {/* file too big */ ....}
                 if ( (bytes_read = read(fd,bigbuff,
                                                file_length*sizeof(float))) == -1)
                                 { /* error reading open */...
                                                 exit(1);
                                 }
}
```

Exercises

Exercise 12573

Write a program to copy one named file into another named file. The two file names are given as the first two arguments to the program.

Copy the file a block (512 bytes) at a time.

```
Check:  that the program has two arguments
                  or print "Program need two arguments"
                that the first name file is readable
                  or print "Cannot open file .... for reading"
                 that the second file is writable
                   or print "Cannot open file .... for writing"
```

Exercise 12577

Write a program last that prints the last *n* lines of a text file, by *n* and the file name should be specified form command line input. By default *n* should be 5, but your program should allow an optional argument so that

```
last -n file.txt
```

prints out the last n lines, where n is any integer. Your program should make the best use of available storage.

Exercise 12578

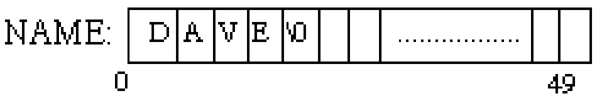
Write a program to compare two files and print out the lines where they differ. Hint: look up appropriate string and file handling library routines. This should not be a very long program.

Dave Marshall
1/5/1999

Subsections

String Handling: `<string.h>`

Recall from our discussion of arrays (Chapter 6) that strings are defined as an array of characters or a pointer to a portion of memory containing ASCII characters. A `string` in C is a sequence of zero or more characters followed by a `NULL` (`\0`)character:



It is important to preserve the `NULL` terminating character as it is how C defines and manages variable length strings. **All** the C standard library functions require this for successful operation.

In general, apart from some *length-restricted functions* (`strncat(), strncmp,()` and `strncpy()`), unless you create strings by hand you should not encounter any such problems, . You should use the many useful string handling functions and not really need to *get your hands dirty* dismantling and assembling strings.

Basic String Handling Functions

All the string handling functions are prototyped in:

```
#include <string.h>
```

The common functions are described below:

`char *stpcpy (const char *dest,const char *src)` -- Copy one string into another.
`int strcmp(const char *string1,const char *string2)` - Compare string1 and string2 to determine alphabetic order.
`char *strcpy(const char *string1,const char *string2)` -- Copy string2 to stringl.
`char *strerror(int errnum)` -- Get error message corresponding to specified error number.
`int strlen(const char *string)` -- Determine the length of a string.
`char *strncat(const char *string1, char *string2, size_t n)` --

Append n characters from string2 to stringl.
`int strncmp(const char *string1, char *string2, size_t n)` -- Compare first n characters of two strings.
`char *strncpy(const char *string1,const char *string2, size_t n)` -- Copy first n characters of string2 to stringl .
`int strcasecmp(const char *s1, const char *s2)` -- case insensitive version of `strcmp()`.
`int strncasecmp(const char *s1, const char *s2, int n)` -- case insensitive version of `strncmp()`.

The use of most of the functions is straightforward, for example:

```
char *str1 = "HELLO";
char *str2;
int length;

length = strlen("HELLO"); /* length = 5 */
(void) strcpy(str2,str1);
```

Note that both `strcat()` and `strcopy()` both return a copy of their first argument which is the destination array. Note the order of the arguments is *destination array* followed by *source array* which is sometimes easy to get the wrong around when programming.

The `strcmp()` function *lexically* compares the two input strings and returns:

Less than zero
: -- if `string1` is lexically less than `string2`

Zero
: -- if `string1` and `string2` are lexically equal

Greater than zero
: -- if `string1` is lexically greater than `string2`

This can also confuse beginners and experience programmers forget this too.

The `strncat(), strncmp,()` and `strncpy()` copy functions are string restricted version of their more general counterparts. They perform a similar task but only up to the first `n` characters. Note the the `NULL` terminated requirement may get violated when using these functions, for example:

```
char *str1 = "HELLO";
char *str2;
int length = 2;


(void) strcpy(str2,str1, length); /* str2 = "HE" */
```

`str2` is NOT NULL TERMINATED!! -- BEWARE

String Searching

The library also provides several string searching functions:

`char *strchr(const char *string, int c)` -- Find first occurrence of character `c` in string.
`char *strrchr(const char *string, int c)` -- Find last occurrence of character `c` in string.
`char *strstr(const char *s1, const char *s2)` -- locates the first occurrence of the string `s2` in string `s1`.
`char *strpbrk(const char *s1, const char *s2)` -- returns a pointer to the first occurrence in string s1 of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`
`size_t strspn(const char *s1, const char *s2)` -- returns the number of characters at the begining of `s1` that match `s2`.
`size_t strcspn(const char *s1, const char *s2)` -- returns the number of characters at the begining of `s1` that *do not* match `s2`.
`char *strtok(char *s1, const char *s2)` -- break the string pointed to by `s1` into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by `s2`.
`char *strtok_r(char *s1, const char *s2, char **lasts)` -- has the same functionality as strtok() except that a pointer to a string placeholder lasts must be supplied by the caller.

`strchr()` and `strrchr()` are the simplest to use, for example:

```
char *str1 = "Hello";
char *ans;

ans = strchr(str1,'l');
```

After this execution, `ans` points to the location `str1 + 2`

`strpbrk()` is a more general function that searches for the first occurrence of any of a group of characters, for example:

```
char *str1 = "Hello";
char *ans;

ans = strpbrk(str1,'aeiou');
```

Here, `ans` points to the location `str1 + 1`, the location of the first `e`.

strstr() returns a pointer to the specified search string or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1. For example,

```
char *str1 = "Hello";
char *ans;
```

```
ans = strstr(str1,'lo');
```

will yield `ans` = `str + 3`.

`strtok()` is a little more complicated in operation. If the first argument is not NULL then the function finds the position of any of the second argument characters. However, the position is remembered and any subsequent calls to `strtok()` will start from this position if on these subsequent calls the first argument is `NULL`. For example, If we wish to break up the string `str1` at each space and print each token on a new line we could do:

```
char *str1 = "Hello Big Boy";
char *t1;


for ( t1 = strtok(str1," ");
      t1 != NULL;
      t1 = strtok(NULL, " ") )

printf("%s\n",t1);
```

Here we use the for loop in a non-standard counting fashion:

- The initialisation calls `strtok()` loads the function with the string `str1`
- We terminate when t1 is `NULL`
- We keep assigning tokens of `str1` to `t1` until termination by calling `strtok()` with a `NULL` first argument.

Character conversions and testing: `ctype.h`

We conclude this chapter with a related library `#include <ctype.h>` which contains many useful functions to convert and test *single* characters. The common functions are prototypes as follows:

Character testing:

`int isalnum(int c)` -- True if c is alphanumeric.
`int isalpha(int c)` -- True if c is a letter.
`int isascii(int c)` -- True if c is ASCII .
`int iscntrl(int c)` -- True if c is a control character.
`int isdigit(int c)` -- True if c is a decimal digit
`int isgraph(int c)` -- True if c is a graphical character.
`int islower(int c)` -- True if c is a lowercase letter
`int isprint(int c)` -- True if c is a printable character
`int ispunct (int c)` -- True if c is a punctuation character.
`int isspace(int c)` -- True if c is a space character.

`int isupper(int c)` -- True if c is an uppercase letter.
`int isxdigit(int c)` -- True if c is a hexadecimal digit

Character Conversion:

`int toascii(int c)` -- Convert `c` to ASCII .
`tolower(int c)` -- Convert c to lowercase.
`int toupper(int c)` -- Convert c to uppercase.

The use of these functions is straightforward and we do not give examples here.

Memory Operations: `<memory.h>`

Finally we briefly overview some basic memory operations. Although not strictly string functions the functions are prototyped in `#include <string.h>`:

`void *memchr (void *s, int c, size_t n)` -- Search for a character in a buffer .
`int memcmp (void *s1, void *s2, size_t n)` -- Compare two buffers.
`void *memcpy (void *dest, void *src, size_t n)` -- Copy one buffer into another .
`void *memmove (void *dest, void *src, size_t n)` -- Move a number of bytes from one buffer lo another.
`void *memset (void *s, int c, size_t n)` -- Set all bytes of a buffer to a given character.

Their use is fairly straightforward and not dissimilar to comparable string operations (except the exact length (`n`) of the operations must be specified as there is no natural termination here).

Note that in all case to **bytes** of memory are copied. The `sizeof()` function comes in handy again here, for example:

```
char src[SIZE],dest[SIZE];
int  isrc[SIZE],idest[SIZE];

memcpy(dest,src, SIZE); /* Copy chars (bytes) ok */

memcpy(idest,isrc, SIZE*sizeof(int)); /* Copy arrays of ints */
```

`memmove()` behaves in exactly the same way as `memcpy()` except that the source and destination locations may overlap.

`memcmp()` is similar to `strcmp()` except here *unsigned bytes* are compared and returns less than zero if `s1` is less than `s2` *etc*.

Exercises

Exercise 12584

Write a function similar to strlen that can handle unterminated strings. Hint: you will need to know and pass in the length of the string.

Exercise 12585

Write a function that returns true if an input string is a palindrome of each other. A palindrome is a word that reads the same backwards as it does forwards *e.g* ABBA.

Exercise 12586

Suggest a possible implementation of the `strtok()` function:

1. using other string handling functions.
2. from first pointer principles

How is the storage of the tokenised string achieved?

Exercise 12587

Write a function that converts all characters of an input string to upper case characters.

Exercise 12591

Write a program that will reverse the contents stored in memory in bytes. That is to say if we have *n* bytes in memory byte *n* becomes byte 0, byte *n*-1 becomes byte 1 *etc.*

Dave Marshall
1/5/1999

Subsections

File Access and Directory System Calls

There are many UNIX utilities that allow us to manipulate directories and files. `cd, ls, rm, cp, mkdir` *etc.* are examples we have (hopefully) already met.

We will now see how to achieve similar tasks from within a C program.

Directory handling functions: <unistd.h>

This basically involves calling appropriate functions to traverse a directory hierarchy or inquire about a directories contents.

`int chdir(char *path)` -- changes directory to specified path string.

Example: C emulation of UNIX's `cd` command:

```
#include<stdio.h>
#include<unistd.h>

main(int argc,char **argv)
   {
                              if (argc < 2)
                                             {  printf(``Usage: %s
                                                             <pathname
> \n'',argv[0]);

                                                            exit(1);
                                              }
                              if (chdir(argv[1]) != 0)
                                             { printf(``Error in chdir\n'');
                                                            exit(1);
                                             }
               }
```

`char *getwd(char *path) -- get the full pathname of the current working directory. path is a pointer to a string where the pathname will be returned. getwd returns a pointer to the string or NULL if an error occurs.`

Scanning and Sorting Directories: <sys/types.h>,<sys/dir.h>

Two useful functions (On BSD platforms and **NOT** in multi-threaded application) are available

`scandir(char *dirname, struct direct **namelist, int (*select)(),` `int (*compar)())` -- reads the directory dirname and builds an array of pointers to directory entries or -1 for an error. `namelist` is a pointer to an array of structure pointers.

`(*select))()` is a pointer to a function which is called with a pointer to a directory entry (defined in <sys/types> and should return a non zero value <u>if</u> the directory entry should be <u>included</u> in the array. If this pointer is NULL, then all the directory entries will be included.

The last argument is a pointer to a routine which is passed to `qsort` (see `man qsort`) -- a built in function which sorts the completed array. <u>If</u> this pointer is NULL, the array is <u>not</u> sorted.

`alphasort(struct direct **d1, **d2)` -- alphasort() is a built in routine which will sort the array alphabetically.

Example - a simple C version of UNIX `ls` utility

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE

extern  int alphasort();

char pathname[MAXPATHLEN];

main()   { int count,i;
                                struct direct **files;
                                int file_select();

                                if (getwd(pathname) == NULL )
                                                { printf("Error getting path\n");

                                                                exit(0);
                                                }
                                printf("Current Working Directory = %s\n",pathname);

                                count =
                                  scandir(pathname, &files, file_select, alphasort);

                                /* If no files found, make a non-selectable menu item
*/
                                if              (count <= 0)
                                                {               printf(``No files in
this directory\n'');

                                                                exit(0);
                                                }
                                printf(``Number of files = %d\n'',count);

                                for (i=1;i<count+1;++i)
                                                 printf(``%s  '',files[i-1]->d_name);
                                printf(``\n''); /* flush buffer */

                }


int file_select(struct direct   *entry)

                {if ((strcmp(entry->d_name, ``.'') == 0) ||
                                                (strcmp(entry->d_name, ``..'') == 0))
                                                 return (FALSE);
                                else
                                                                return (TRUE);
```

```
                }
```

scandir returns the current directory (.) and the directory above this (..) as well as all files so we need to check for these and return FALSE so that they are not included in our list.

Note: scandir and alphasort have definitions in sys/types.h and sys/dir.h. MAXPATHLEN and getwd definitions in sys/param.h

We can go further than this and search for specific files: Let's write a modified file_select() that only scans for files with a .c, .o or .h suffix:

```
int file_select(struct direct   *entry)

                {char *ptr;
                                 char *rindex(char *s, char c);

                                if ((strcmp(entry->d_name, ``.'')== 0) ||
                                                (strcmp(entry->d_name, ``..'') == 0))
                                                 return (FALSE);

                                 /* Check for filename extensions */
                                 ptr = rindex(entry->d_name, '.')
                                 if ((ptr != NULL) &&
                                                 ((strcmp(ptr, ``.c'') == 0)
                                                 || (strcmp(ptr, ``.h'') == 0)
                                                 || (strcmp(ptr, ``.o'') == 0) ))
                                                                 return (TRUE);
                                 else
                                                 return(FALSE);
                }
```

NOTE: rindex() is a string handling function that returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c does not occur in the string. (index() is similar function but assigns a pointer to 1st occurrence.)

The function struct direct *readdir(char *dir) also exists in <sys/dir.h>> to return a given directory dir listing.

File Manipulation Routines: unistd.h, sys/types.h, sys/stat.h

There are many system calls that can applied directly to files stored in a directory.

File Access

`int access(char *path, int mode)` -- determine accessibility of file.

`path` points to a path name naming a file. `access()` checks the named file for accessibility according to `mode`, defined in `#include <unistd.h>`:

R_OK
: - test for read permission

W_OK
: - test for write permission

X_OK
: - test for execute or search permission

F_OK

- test whether the directories leading to the file can be searched and the file exists.

`access()` returns: 0 on success, -1 on failure and sets `errno` to indicate the error. See `man` pages for list of errors.

errno

`errno` is a special system variable that is set if a system call cannot perform its set task.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program other wise it simply retains its last value.

`int chmod(char *path, int mode)` change the mode of access of a file. specified by `path` to the given `mode`.

`chmod()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are defined in `#include <sys/stat.h>`

The access mode of a file can be set using predefined macros in `sys/stat.h` -- see `man` pages -- or by setting the mode in a a 3 digit octal number.

The rightmost digit specifies owner privileges, middle group privileges and the leftmost other users privileges.

For each octal digit think of it a 3 bit binary number. Leftmost bit = read access (on/off) middle is write, right is executable.

So 4 (octal 100) = read only, 2 (010) = write, 6 (110) = read and write, 1 (001) = execute.

so for access mode 600 gives user read and write access others no access. 666 gives everybody read/write access.

NOTE: a UNIX command `chmod` also exists

File Status

Two useful functions exist to inquire about the files current status. You can find out how large the file is (`st_size`) when it was created (`st_ctime`) *etc.* (see `stat` structure definition below. The two functions are prototyped in `<sys/stat.h>`

```
int stat(char *path, struct stat *buf),
int fstat(int fd, struct
stat *buf)
```

`stat()` obtains information about the file named by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

`fstat()` obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an `open` call (Low level I/O).

`stat(), and fstat()` return 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are again defined in `#include <sys/stat.h>`

`buf` is a pointer to a stat structure into which information is placed concerning the file. A stat structure is define in `#include <sys/types.h>`, as follows

```
struct stat {
          mode_t   st_mode;     /* File mode (type, perms) */
          ino_t    st_ino;      /* Inode number */
          dev_t    st_dev;      /* ID of device containing */
                                /* a directory entry for this file */
          dev_t    st_rdev;     /* ID of device */
                                /* This entry is defined only for */
                                /* char special or block special files */
          nlink_t  st_nlink;    /* Number of links */
          uid_t    st_uid;      /* User ID of the file's owner */
          gid_t    st_gid;      /* Group ID of the file's group */
          off_t    st_size;     /* File size in bytes */
          time_t   st_atime;    /* Time of last access */
```

```
          time_t   st_mtime;    /* Time of last data modification */
          time_t   st_ctime;    /* Time of last file status change */
                                /* Times measured in seconds since */
                                /* 00:00:00 UTC, Jan. 1, 1970 */
          long     st_blksize;  /* Preferred I/O block size */
          blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
}
```

File Manipulation:stdio.h, unistd.h

There are few functions that exist to delete and rename files. Probably the most common way is to use the `stdio.h` functions:

```
int remove(const char *path);
int rename(const char *old, const char *new);
```

Two system calls (defined in `unistd.h`) which are actually used by `remove()` and `rename()` also exist but are probably harder to remember unless you are familiar with UNIX.

`int unlink(cons char *path)` -- removes the directory entry named by `path`

`unlink()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors listed in `#include <sys/stat.h>`

A similar function `link(const char *path1, const char *path2)` creates a linking from an existing directory entry `path1` to a new entry `path2`

Creating Temporary FIles:<stdio.h>

Programs often need to create files just for the life of the program. Two convenient functions (plus some variants) exist to assist in this task. Management (deletion of files etc) is taken care of by the Operating System.

The function `FILE *tmpfile(void)` creates a temporary file and opens a corresponding stream. The file will automatically be deleted when all references to the file are closed.

The function `char *tmpnam(char *s)` generate file names that can safely be used for a temporary file. Variant functions `char *tmpnam_r(char *s)` and `char *tempnam(const char *dir, const char *pfx)` also exist

NOTE: There are a few more file manipulation routines not listed here see `man` pages.

Exercises

Exercise 12675

Write a C program to emulate the `ls -l` UNIX command that prints all files in a current directory and lists access privileges etc. DO NOT simply `exec ls -l` from the program.

Exercise 12676

Write a program to print the lines of a file which contain a word given as the program argument (a simple version of `grep` UNIX utility).

Exercise 12677

Write a program to list the files given as arguments, stopping every 20 lines until a key is hit.(a simple version of `more` UNIX utility)

Exercise 12678

Write a program that will list all files in a current directory and all files in subsequent sub directories.

Exercise 12679

Write a program that will only list subdirectories in alphabetical order.

Exercise 12680

Write a program that shows the user all his/her C source programs and then prompts interactively as to whether others should be granted read permission; if affirmative such permission should be granted.

Exercise 12681

Write a program that gives the user the opportunity to remove any or all of the files in a current working directory. The name of the file should appear followed by a prompt as to whether it should be removed.

Dave Marshall
1/5/1999

Subsections

Time Functions

In this chapter we will look at how we can access the clock time with UNIX system calls.

There are many more time functions than we consider here - see `man` pages and standard library function listings for full details. In this chapter we concentrate on applications of timing functions in C

Uses of time functions include:

- telling the time.
- timing programs and functions.
- setting number seeds.

Basic time functions

Some of thge basic time functions are prototypes as follows:

`time_t time(time_t *tloc)` -- returns the time since `00:00:00 GMT, Jan. 1, 1970`, measured in seconds.

If `tloc` is not NULL, the return value is also stored in the location to which tloc points.

`time()` returns the value of time on success.

On failure, it returns `(time_t) -1`. `time_t` is typedefed to a long (int) in <sys/types.h> and <sys/time.h> header files.

`int ftime(struct timeb *tp)` -- fills in a structure pointed to by `tp`, as defined in <sys/timeb.h>:

```
   struct timeb
                  { time_t time;
                                            unsigned short millitm;
                                            short timezone;
                                            short dstflag;
                              };
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Day light Saving time applies locally during the appropriate part of the year.

On success, `ftime() returns no useful value. On failure, it returns -1.`

`Two other functions defined` *etc.* `in #include <time.h>`

```
char *ctime(time_t *clock),
char *asctime(struct tm *tm)
```

`ctime() converts a long integer, pointed to by clock, to a 26-character` string of the form produced by asctime(). It first breaks down clock to a tm structure by calling localtime(), and then calls asctime() to convert that tm structure to a string.

asctime() converts a time value contained in a tm structure to a 26-character string of the form:

```
   Sun Sep 16 01:03:52 1973
```

asctime() returns a pointer to the string.

Example time applications

we mentioned above three possible uses of time functions (there are many more) but these are very common.

Example 1: Time (in seconds) to perform some computation

This is a simple program that illustrates that calling the time function at distinct moments and noting the different times is a simple method of timing fragments of code:

```
/* timer.c */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
  {  int i;
                                time_t t1,t2;

                                (void) time(&t1);
                                for (i=1;i<=300;++i)
                                  printf(``%d %d %d\n'',i, i*i, i*i*i);
                                 (void) time(&t2);
                                 printf(``\n Time to do 300 squares and
                                cubes= %d seconds\n'', (int) t2-t1);
                 }
```

Example 2: Set a random number seed

We have seen a similar example previously, this time we use the `lrand48()` function to generate of number sequence:

```
/* random.c */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>
```

```
main()
  { int i;
                                    time_t t1;

                                    (void) time(&t1);
                                    srand48((long) t1);
                                    /* use time in seconds to set seed */
                                    printf(``5 random numbers
                                      (Seed = %d):\n'',(int) t1);

                                    for (i=0;i<5;++i)
                                      printf(``%d '', lrand48());
                                    printf(``\n\n''); /* flush print buffer */

                }
```

lrand48() returns non-negative long integers uniformly distributed over the interval (0, 2**31).

A similar function drand48() returns double precision numbers in the range [0.0,1.0).

srand48() sets the seed for these random number generators. It is important to have different seeds when we call the functions otherwise the same set of pseudo-random numbers will generated. time() always provides a unique seed.

Exercises

Exercise 12708

Write a C program that times a fragment of code in milliseconds.

Exercise 12709

Write a C program to produce a series of floating point random numbers in the ranges (a) 0.0 - 1.0 (b) 0.0 - n where n is any floating point value. The seed should be set so that a unique sequence is guaranteed.

Dave Marshall
1/5/1999

Subsections

Process Control: `<stdlib.h>,<unistd.h>`

A *process* is basically a single running program. It may be a ``system'' program (*e.g* login, update, csh) or program initiated by the user (textedit, dbxtool or a user written one).

When UNIX runs a process it gives each process a unique number - a process ID, `pid`.

The UNIX command `ps` will list all current processes running on your machine and will list the pid.

The C function `int getpid()` will return the pid of process that called this function.

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Running UNIX Commands from C

We can run commands from a C program just as if they were from the UNIX command line by using the `system()` function. **NOTE:** this can save us a lot of time and hassle as we can run other (proven) programs, scripts *etc.* to do set tasks.

`int system(char *string)` -- where string can be the name of a unix utility, an executable shell script or a user program. System returns the exit status of the shell. System is prototyped in `<stdlib.h>`

Example: Call `ls` from a program

```
main()
{ printf(``Files in Directory are:\n'');

                 system(``ls -l'');
}
```

```
system is a call that is made up of 3 other system calls: execl(), wait() and
fork() (which are prototyed in <unistd>)
```

execl()

`execl` has 5 other related functions -- see `man` pages.

`execl` stands for *execute* and *leave* which means that a process will get executed and then terminated by `execl`.

It is defined by:

`execl(char *path, char *arg0,...,char *argn, 0);`

The last parameter must always be 0. It is a *NULL terminator*. Since the argument list is variable we must have some way of telling C when it is to end. The NULL terminator does this job.

where `path` points to the name of a file holding a command that is to be executed, `argo` points to a string that is the same as path (or at least its last component.

`arg1 ... argn` are pointers to arguments for the command and 0 simply marks the end of the (variable) list of arguments.

So our above example could look like this also:

```
main()
{ printf(``Files in Directory are:\n'');

                  execl(`/bin/ls'',``ls'', ``-l'',0);
}
```

fork()

`int fork()` turns a single process into 2 identical processes, known as the *parent* and the *child*. On success, fork() returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, fork() returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

NOTE: The child process will have its own unique PID.

The following program illustrates a simple use of fork, where two copies are made and run together (multitasking)

```
main()
{ int return_value;

                 printf(``Forking process\n'');

                 fork();
                 printf(``The process id is %d
                   and return value is %d\n",

                   getpid(), return_value);
                 execl(``/bin/ls/'',``ls'',``-l'',0);
                 printf(``This line is not printed\n'');

}
```

The Output of this would be:

```
Forking process
The process id is 6753 and return value is 0
The process id is 6754 and return value is 0
two lists of files in current directory
```

NOTE: The processes have unique ID's which will be different at each run.

It also impossible to tell in advance which process will get to CPU's time -- so one run may differ from the next.

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child. We can trap any errors if fork returns a -1. *i.e.*:

```
int pid; /* process identifier */

pid = fork();
if ( pid < 0 )
                 { printf(``Cannot fork!!\n'');

                                 exit(1);
                 }
if ( pid == 0 )
                 { /* Child process */ ...... }
else
                 { /* Parent process pid is child's pid */
                 .... }
```

wait()

`int wait (int *status_location)` -- will force a parent process to wait for a child process to stop or terminate. `wait()` return the pid of the child or -1 for an error. The exit status of the child is returned to `status_location`.

exit()

`void exit(int status)` -- terminates the process which calls this function and returns the exit `status` value. Both UNIX and C (forked) programs can read the status value.

By convention, a status of 0 means *normal termination* any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the `sys/stat.h` header file. We can easily derive our own conventions.

A complete example of forking program is originally titled `fork.c`:

```
/* fork.c - example of a fork in a program */
/* The program asks for UNIX commands to be typed and inputted to a string*/
/* The string is then "parsed" by locating blanks etc. */
/* Each command and sorresponding arguments are put in a args array */
/* execvp is called to execute these commands in child process */
/* spawned by fork() */

/* cc -o fork fork.c */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
    char buf[1024];
    char *args[64];

    for (;;) {
        /*
         * Prompt for and read a command.
         */
        printf("Command: ");
```

```
        if (gets(buf) == NULL) {
            printf("\n");
            exit(0);
        }

        /*
         * Split the string into arguments.
         */
        parse(buf, args);

        /*
         * Execute the command.
         */
        execute(args);
    }
}

/*
 * parse--split the command in buf into
 *         individual arguments.
 */
parse(buf, args)
char *buf;
char **args;
{
    while (*buf != NULL) {
        /*
         * Strip whitespace.  Use nulls, so
         * that the previous argument is terminated
         * automatically.
         */
        while ((*buf == ' ') || (*buf == '\t'))
            *buf++ = NULL;

        /*
         * Save the argument.
         */
        *args++ = buf;

        /*
         * Skip over the argument.
         */
        while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
            buf++;
    }

    *args = NULL;
}

/*
 * execute--spawn a child process and execute
 *           the program.
 */
execute(args)
char **args;
{
    int pid, status;

    /*
```

```
     * Get a child process.
     */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);

        /* NOTE: perror() produces a short  error  message  on  the  standard
           error describing the last error encountered during a call to
           a system or library function.
       */
    }

    /*
     * The child executes the code inside the if.
     */
    if (pid == 0) {
        execvp(*args, args);
        perror(*args);
        exit(1);

       /* NOTE: The execv() vnd execvp versions of execl() are useful when the
          number  of  arguments is unknown in advance;
          The arguments to execv() and execvp()  are the name
          of the file to be executed and a vector of strings  contain-
          ing  the  arguments.   The last argument string must be fol-
          lowed by a 0 pointer.

          execlp() and execvp() are called with the same arguments  as
          execl()  and  execv(),  but duplicate the shell's actions in
          searching for an executable file in a list  of  directories.
          The directory list is obtained from the environment.
        */
    }

    /*
     * The parent executes the wait.
     */
    while (wait(&status) != pid)
        /* empty */ ;
}
```

Exerises

Exercise 12727

Use `popen()` to pipe the `rwho` (UNIX command) output into `more` (UNIX command) in a C program.

Dave Marshall
1/5/1999

Subsections

Interprocess Communication (IPC), Pipes

We have now began to see how multiple processes may be running on a machine and maybe be controlled (spawned by `fork()` by one of our programs.

In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task. We will consider:

- Pipes
- Signals
- Message Queues
- Semaphores
- Shared Memory
- Sockets

In this chapter, we will study the piping of two processes. We will study the others in turn in subsequent chapters.

Piping in a C program: `<stdio.h>`

Piping is a process where the input of one process is made the input of another. We have seen examples of this from the UNIX command line using |.

We will now see how we do this from C programs.

We will have two (or more) `forked` processes and will communicate between them.

We must first open a *pipe*

UNIX allows two ways of opening a pipe.

`popen()` -- Formatted Piping

`FILE *popen(char *command, char *type)` -- opens a pipe for I/O where the command is the process that will be connected to the calling process thus creating the *pipe*. The type is either ``r'' - for reading, or ``w'' for writing.

`popen()` returns is a stream pointer or NULL for any errors.

A pipe opened by `popen()` should always be closed by `pclose(FILE *stream)`.

We use `fprintf()` and `fscanf()` to communicate with the pipe's `stream`.

`pipe()` -- Low level Piping

`int pipe(int fd[2])` -- creates a pipe and returns two file descriptors, `fd[0], fd[1]`. `fd[0]` is opened for reading, `fd[1]` for writing.

`pipe()` returns 0 on success, -1 on failure and sets `errno` accordingly.

The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using `read()` and `write()`.

Pipes opened with `pipe()` should be closed with `close(int fd)`.

Example: Parent writes to a child

```
int pdes[2];

pipe(pdes);
if ( fork() == 0 )
  { /* child */
                                 close(pdes[1]); /* not required */
                                 read( pdes[0]); /* read from parent */
                                 .....
                 }
else
                 { close(pdes[0]); /* not required */
                                 write( pdes[1]); /* write to child */
                                 .....
                 }
```

An futher example of piping in a C program is `plot.c and subroutines and it performs as follows:`

- `The program has two modules plot.c (main) and plotter.c.`
- `The program relies on you having installed the freely` *gnuplot* `graph drawing program in the directory /usr/local/bin/ (in the listing below at least) -- this path could easily be changed.`
- `The program plot.c calls` *gnuplot*
- `Two Data Stream is generated from Plot`
  - $y = sin(x)$
  - $y = sin(1/x)$
- `2 Pipes created -- 1 per Data Stream.`
- °*Gnuplot* `produces ``live'' drawing of output.`

`The code listing for plot.c is:`

```
/* plot.c - example of unix pipe. Calls gnuplot graph drawing package to draw
   graphs from within a C program. Info is piped to gnuplot */
/* Creates 2 pipes one will draw graphs of y=0.5 and y = random 0-1.0 */
/* the other graphs of y = sin (1/x) and y = sin x */

/* Also user a plotter.c module */
/* compile: cc -o plot plot.c plotter.c */

#include "externals.h"
#include <signal.h>
```

```
#define DEG_TO_RAD(x) (x*180/M_PI)

double drand48();
void quit();

FILE *fp1, *fp2, *fp3, *fp4, *fopen();

main()
{   float i;
    float y1,y2,y3,y4;

    /* open files which will store plot data */
    if ( ((fp1 = fopen("plot11.dat","w")) == NULL) ||
           ((fp2 = fopen("plot12.dat","w")) == NULL) ||
            ((fp3 = fopen("plot21.dat","w")) == NULL) ||
             ((fp4 = fopen("plot22.dat","w")) == NULL) )
              { printf("Error can't open one or more data files\n");
                exit(1);
              }

    signal(SIGINT,quit); /* trap ctrl-c call quit fn */
    StartPlot();
    y1 = 0.5;
    srand48(1); /* set seed */
    for (i=0;;i+=0.01) /* increment i forever use ctrl-c to quit prog */
      { y2 =  (float) drand48();
        if (i == 0.0)
           y3 = 0.0;
       else
           y3 = sin(DEG_TO_RAD(1.0/i));
        y4 = sin(DEG_TO_RAD(i));

        /* load files */
        fprintf(fp1,"%f %f\n",i,y1);
        fprintf(fp2,"%f %f\n",i,y2);
        fprintf(fp3,"%f %f\n",i,y3);
        fprintf(fp4,"%f %f\n",i,y4);

        /* make sure buffers flushed so that gnuplot */
        /*  reads up to data file */
        fflush(fp1);
        fflush(fp2);
        fflush(fp3);
        fflush(fp4);

        /* plot graph */
        PlotOne();
        usleep(250); /* sleep for short time */
      }
}

void quit()
{  printf("\nctrl-c caught:\n Shutting down pipes\n");
   StopPlot();

   printf("closing data files\n");
   fclose(fp1);
   fclose(fp2);
```

```
   fclose(fp3);
   fclose(fp4);

   printf("deleting data files\n");
   RemoveDat();
}
```

The plotter.c module is as follows:

```
/* plotter.c module */
/* contains routines to plot a data file produced by another program  */
/* 2d data plotted in this version                                     */
/**********************************************************************/

#include "externals.h"

static FILE *plot1,
       *plot2,
       *ashell;

static char *startplot1 = "plot [] [0:1.1]'plot11.dat' with lines,
            'plot12.dat' with lines\n";

static char *startplot2 = "plot 'plot21.dat' with lines,
            'plot22.dat' with lines\n";

static char *replot = "replot\n";
static char *command1= "/usr/local/bin/gnuplot> dump1";
static char *command2= "/usr/local/bin/gnuplot> dump2";
static char *deletefiles = "rm plot11.dat plot12.dat plot21.dat plot22.dat";
static char *set_term = "set terminal x11\n";

void
StartPlot(void)
 { plot1 = popen(command1, "w");
   fprintf(plot1, "%s", set_term);
   fflush(plot1);
   if (plot1 == NULL)
      exit(2);
   plot2 = popen(command2, "w");
   fprintf(plot2, "%s", set_term);
   fflush(plot2);
   if (plot2 == NULL)
      exit(2);
 }

void
RemoveDat(void)
 { ashell = popen(deletefiles, "w");
   exit(0);
 }

void
StopPlot(void)
 { pclose(plot1);
   pclose(plot2);
 }
```

```
void
PlotOne(void)
 { fprintf(plot1, "%s", startplot1);
   fflush(plot1);

   fprintf(plot2, "%s", startplot2);
   fflush(plot2);
 }

void
RePlot(void)
 { fprintf(plot1, "%s", replot);
   fflush(plot1);
 }
```

The header file externals.h contains the following:

```
/* externals.h */
#ifndef EXTERNALS
#define EXTERNALS

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* prototypes */

void StartPlot(void);
void RemoveDat(void);
void StopPlot(void);
void PlotOne(void);
void RePlot(void);
#endif
```

Exercises

Exercise 12733

Setup a two-way pipe between parent and child processes in a C program. i.e. both can send and receive signals.

Dave Marshall
1/5/1999

Subsections

IPC:Interrupts and Signals: `<signal.h>`

In this section will look at ways in which two processes can communicate. When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens. Signals can be synchronously generated by an error in an application, such as `SIGFPE` and `SIGSEGV`, but most signals are asynchronous. Signals can be posted to a process when the system detects a software event, such as a user entering an interrupt or stop or a kill request from another process. Signals can also be come directly from the OS kernel when a hardware event such as a bus error or an illegal instruction is encountered. The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future. Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored. Each signal has a default action which is one of the following:

- The signal is discarded after being received
- The process is terminated after the signal is received
- A core file is written, then the process is terminated
- Stop the process after the signal is received

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Macros are defined in `<signal.h>` header file for common signals.

These include:

SIGHUP 1 /* hangup */	SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */	SIGILL 4 /* illegal instruction */
SIGABRT 6 /* used by abort */	SIGKILL 9 /* hard kill */
SIGALRM 14 /* alarm clock */	
SIGCONT 19 /* continue a stopped process */	
SIGCHLD 20 /* to parent on child stop or exit */	

Signals can be numbered from 0 to 31.

Sending Signals -- `kill(), raise()`

There are two common functions used to send signals

`int kill(int pid, int signal)` - a system call that send a `signal` to a process, `pid`. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

`kill()` returns 0 for a successful call, -1 otherwise and sets `errno` accordingly.

`int raise(int sig)` sends the signal sig to the executing program. `raise()` actually uses `kill()` to send the signal to the executing program:

```
        kill(getpid(), sig);
```

There is also a UNIX command called kill that can be used to send signals from the command line - see `man` pages.

NOTE: that unless caught or ignored, the `kill` signal terminates the process. Therefore protection is built into the system.

Only processes with certain access privileges can be killed off.

Basic rule: *only processes that have the same user can send/receive messages.*

The `SIGKILL` signal cannot be caught or ignored and will always terminate a process.

For example`kill(getpid(),SIGINT);` would send the interrupt signal to the id of the calling process.

This would have a similar effect to `exit()` command. Also `ctrl-c` typed from the command sends a `SIGINT` to the process currently being.

`unsigned int alarm(unsigned int seconds)` -- sends the signal `SIGALRM` to the invoking process after seconds seconds.

Signal Handling -- `signal()`

An application program can specify a function called a signal handler to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to catch the signal. A process can deal with a signal in one of the following ways:

- The process can let the default action happen
- The process can block the signal (some signals cannot be ignored)
- the process can catch the signal with a handler.

Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself

Receiving signals is straighforward with the function:

`int (*signal(int sig, void (*func)()))()` -- that is to say the function `signal()` will call the `func` functions if the process receives a signal `sig`. Signal returns a pointer to function `func` if successful or it returns an error to `errno` and -1 otherwise.

`func()` can have three values:

`SIG_DFL`

-- a pointer to a system default function `SID_DFL()`, which will terminate the process upon receipt of `sig`.

SIG_IGN

-- a pointer to system ignore function `SIG_IGN()` which will disregard the `sig` action (UNLESS it is `SIGKILL`).

A function address

-- a user specified function.

`SIG_DFL and SIG_IGN` are defined in `signal.h` (standard library) header file.

Thus to ignore a `ctrl-c` command from the command line. we could do:

```
 signal(SIGINT, SIG_IGN);
```

TO reset system so that `SIGINT` causes a termination at any place in our program, we would do:

```
 signal(SIGINT, SIG_DFL);
```

So lets write a program to trap a `ctrl-c` but not quit on this signal. We have a function `sigproc()` that is executed when we trap a `ctrl-c`. We will also set another function to quit the program if it traps the `SIGQUIT` signal so we can terminate our program:

```
#include <stdio.h>

void sigproc(void);

void quitproc(void);

main()
{ signal(SIGINT, sigproc);
                 signal(SIGQUIT, quitproc);
                 printf(``ctrl-c disabled use ctrl-\\ to quit\n'');
                 for(;;); /* infinite loop */}

void sigproc()
{                signal(SIGINT, sigproc); /*  */
                 /* NOTE some versions of UNIX will reset signal to default
                 after each call. So for portability reset signal each time */

                 printf(``you have pressed ctrl-c \n'');

}

void quitproc()
{                printf(``ctrl-\\ pressed to quit\n'');
                 exit(0); /* normal exit status */
}
```

`sig_talk.c` -- complete example program

Let us now write a program that communicates between child and parent processes using `kill()` and `signal()`.

`fork()` creates the child process from the parent. The `pid` can be checked to decide whether it is the child (== 0) or the parent (pid = child process id).

The parent can then send messages to child using the pid and `kill()`.

The child picks up these signals with `signal()` and calls appropriate functions.

An example of communicating process using signals is `sig_talk.c`:

```
/* sig_talk.c --- Example of how 2 processes can talk */
/* to each other using kill() and signal() */
/* We will fork() 2 process and let the parent send a few */
/* signals to it`s child  */

/* cc sig_talk.c -o sig_talk  */

#include <stdio.h>
#include <signal.h>

void sighup(); /* routines child will call upon sigtrap */
void sigint();
void sigquit();

main()
{ int pid;

  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

   if (pid == 0)
     { /* child */
       signal(SIGHUP,sighup); /* set function calls */
       signal(SIGINT,sigint);
       signal(SIGQUIT, sigquit);
       for(;;); /* loop for ever */
     }
  else /* parent */
     {  /* pid hold id of child */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGINT\n\n");
       kill(pid,SIGINT);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGQUIT\n\n");
       kill(pid,SIGQUIT);
       sleep(3);
     }
}
```

```
void sighup()

{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint()

{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit()

{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

Other signal functions

There are a few other functions defined in `signal.h`:

`int sighold(int sig)` -- adds `sig` to the calling process's signal mask

`int sigrelse(int sig)` -- removes `sig` from the calling process's signal mask

`int sigignore(int sig)` -- sets the disposition of `sig` to `SIG_IGN`

`int sigpause(int sig)` -- removes `sig` from the calling process's signal mask and suspends the calling process until a signal is received

---

Dave Marshall
1/5/1999

Subsections

IPC:Message Queues:<`sys/msg.h`>

The basic idea of a *message queue* is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The *sending* process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or `type` so that processes can select the appropriate message. Process must share a common `key` in order to gain access to the queue in the first place (subject to other permissions -- see below).

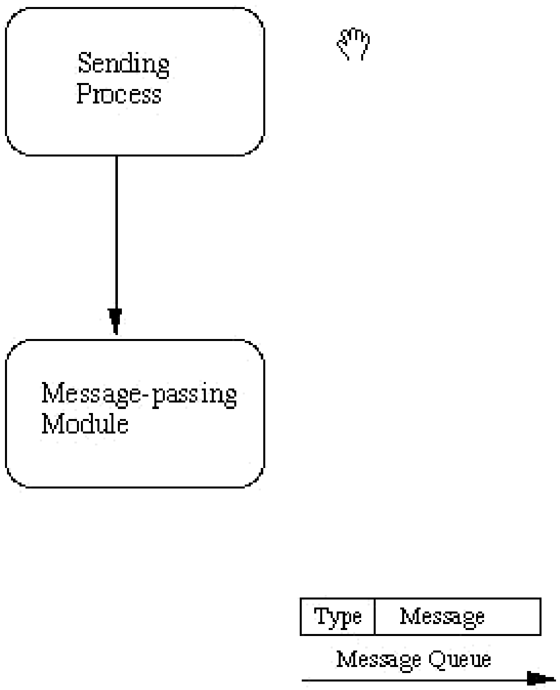


Fig. 24.1 Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the `msgget` function see below) Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initialising the Message Queue

The `msgget()` function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the `msgget()` function.

```
#include <sys/ipc.h>;
#include <sys/msg.h>;

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
...
```

IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a `key_t key` argument. (`key_t` is essentially an `int` type defined in <sys/types.h>

The `key` is an arbitrary value or one that can be derived from a common seed at run time. One way is with `ftok()`

, which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process. When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the `IPC_CREAT` and `IPC_EXCL` flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a `key` based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

Controlling message queues

The `msgctl()` function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using `msgctl()` Also, any process with permission to do so can use `msgctl()` for control operations.

The `msgctl()` function is prototypes as follows:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf )
```

The `msqid` argument must be the ID of an existing message queue. The `cmd argument` is one of:

`IPC_STAT`
: -- Place information about the status of the queue in the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

`IPC_SET`
: -- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

`IPC_RMID`
: -- Remove the message queue specified by the `msqid` argument.

The following code illustrates the `msgctl()` function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions send and receive messages, respectively:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
          int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
          int msgflg);
```

The `msqid` argument **must** be the ID of an existing message queue. The `msgp` argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
 struct mymsg {
      long      mtype;    /* message type */
      char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The `msgsz` argument specifies the length of the message in bytes.

The structure member `msgtype` is the received message's type as specified by the sending process.

The argument `msgflg` specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (`msgflg & IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (`msgflg & IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier `msqid` is removed from the system; when this occurs, `errno` is set equal to `EIDRM` and -1 is returned.
  - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

  Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`:
  - `msg_qnum` is incremented by 1.
  - `msg_lspid` is set equal to the process ID of the calling process.
  - `msg_stime` is set equal to the current time.

The following code illustrates `msgsnd()` and `msgrcv()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */
```

```
...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
exit(1);

...

msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");
...
```

POSIX Messages: `<mqueue.h>`

The POSIX message queue functions are:

`mq_open()` -- Connects to, and optionally creates, a named message queue.

`mq_close()` -- Ends the connection to an open message queue.

`mq_unlink()` -- Ends the connection to an open message queue and causes the queue to be removed when the last process closes it.

`mq_send()` -- Places a message in the queue.

`mq_receive()` -- Receives (removes) the oldest, highest priority message from the queue.

`mq_notify()` -- Notifies a process or thread that a message is available in the queue.

`mq_setattr()` -- Set or get message queue attributes.

The basic operation of these functions is as described above. For full function prototypes and further information see the UNIX `man pages`

Example: Sending messages between two processes

The following two programs should be compiled and run *at the same time* to illustrate basic principle of message passing:

`message_send.c`

-- Creates a message queue and sends one message to the queue.

`message_rec.c`

-- Reads the message from the queue.

`message_send.c` -- creating and sending to a simple message queue

The full code listing for `message_send.c` is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
         long    mtype;
         char    mtext[MSGSZ];
         } message_buf;

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0) {
        perror("msgget");
        exit(1);
    }
    else
     (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);


    /*
     * We'll send message type 1
     */

    sbuf.mtype = 1;
```

```
    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    (void) strcpy(sbuf.mtext, "Did you get this?");

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    buf_length = strlen(sbuf.mtext) ;



    /*
     * Send a message.
     */
    if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
        exit(1);
    }

   else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);

    exit(0);
}
```

The essential points to note here are:

- The Message queue is created with a basic `key` and message flag `msgflg = IPC_CREAT | 0666` -- create queue and make it read and appendable by all.
- A message of type (`sbuf.mtype`) 1 is sent to the queue with the message ``Did you get this?''

`message_rec.c` -- receiving the above message

The full code listing for `message_send.c`'s companion process, `message_rec.c` is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;


main()
{
    int msqid;
    key_t key;
    message_buf  rbuf;
```

```
    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0) {
        perror("msgget");
        exit(1);
    }


    /*
     * Receive an answer of message type 1.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }

    /*
     * Print the answer.
     */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is opened with `msgget` (message flag `0666`) and the *same* `key` as `message_send.c`.
- A message of the *same* type 1 is received from the queue with the message ``Did you get this?'' stored in `rbuf.mtext`.

Some further example message queue programs

The following suite of programs can be used to investigate interactively a variety of massage passing ideas (see exercises below).

The message queue **must** be initialised with the `msgget.c` program. The effects of controlling the queue and sending and receiving messages can be investigated with `msgctl.c` and `msgop.c` respectively.

`msgget.c`: Simple Program to illustrate `msget()`

```
/*
 * msgget.c: Illustrate the msgget() function.
 * This is a simple exerciser of the msgget() function. It prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
extern void  exit();
extern void  perror();

main()
{
 key_t  key;  /* key to be passed to msgget() */
 int  msgflg,  /* msgflg to be passed to msgget() */
   msqid;  /* return value from msgget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);
 (void) fprintf(stderr, "\nExpected flags for msgflg argument
are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter msgflg value: ");
 (void) scanf("%i", &msgflg);

 (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,
%#o)\n",
 key, msgflg);
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
   "msgget: msgget succeeded: msqid = %d\n", msqid);
  exit(0);
 }
}
```

`msgctl.c`Sample Program to Illustrate `msgctl()`

```
/*
 * msgctl.c:  Illustrate the msgctl() function.
 *
 * This is a simple exerciser of the msgctl() function.  It allows
 * you to perform one control operation on one message queue.  It
 * gives up immediately if any control operation fails, so be
careful
 * not to set permissions to preclude read permission; you won't
be
```

```
 * able to reset the permissions with this code if you do.
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission
for \
     yourself, this program will fail frequently!";

main()
{
 struct msqid_ds buf;     /* queue descriptor buffer for IPC_STAT
          and IP_SET commands */
 int    cmd,  /* command to be given to msgctl() */
     msqid;  /* queue ID to be given to msgctl() */

 (void fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the msqid and cmd arguments for the msgctl() call. */
 (void) fprintf(stderr,
  "Please enter arguments for msgctls() as requested.");
 (void) fprintf(stderr, "\nEnter the msqid: ");
 (void) scanf("%i", &msqid);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\nEnter the value for the command: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_SET:
   /* Modify settings in the message queue control structure.
*/
   (void) fprintf(stderr, "Before IPC_SET, get current
values:");
   /* fall through to IPC_STAT processing */
  case IPC_STAT:
   /* Get a copy of the current message queue control
    * structure and show it to the user. */
   do_msgctl(msqid, IPC_STAT, &buf);
   (void) fprintf(stderr, ]
   "msg_perm.uid = %d\n", buf.msg_perm.uid);
   (void) fprintf(stderr,
   "msg_perm.gid = %d\n", buf.msg_perm.gid);
   (void) fprintf(stderr,
   "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
   (void) fprintf(stderr,
   "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
```

```
        (void) fprintf(stderr, "msg_perm.mode = %#o, ",
        buf.msg_perm.mode);
        (void) fprintf(stderr, "access permissions = %#o\n",
        buf.msg_perm.mode & 0777);
        (void) fprintf(stderr, "msg_cbytes = %d\n",
            buf.msg_cbytes);
        (void) fprintf(stderr, "msg_qbytes = %d\n",
            buf.msg_qbytes);
        (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
        (void) fprintf(stderr, "msg_lspid = %d\n",
            buf.msg_lspid);
        (void) fprintf(stderr, "msg_lrpid = %d\n",
            buf.msg_lrpid);
        (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
        ctime(&buf.msg_stime) : "Not Set\n");
        (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
        ctime(&buf.msg_rtime) : "Not Set\n");
        (void) fprintf(stderr, "msg_ctime = %s",
            ctime(&buf.msg_ctime));
        if (cmd == IPC_STAT)
         break;
        /*  Now continue with IPC_SET. */
        (void) fprintf(stderr, "Enter msg_perm.uid: ");
        (void) scanf ("%hi", &buf.msg_perm.uid);
        (void) fprintf(stderr, "Enter msg_perm.gid: ");
        (void) scanf("%hi", &buf.msg_perm.gid);
        (void) fprintf(stderr, "%s\n", warning_message);
        (void) fprintf(stderr, "Enter msg_perm.mode: ");
        (void) scanf("%hi", &buf.msg_perm.mode);
        (void) fprintf(stderr, "Enter msg_qbytes: ");
        (void) scanf("%hi", &buf.msg_qbytes);
        do_msgctl(msqid, IPC_SET, &buf);
        break;
       case IPC_RMID:
       default:
        /* Remove the message queue or try an unknown command. */
        do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
        break;
     }
     exit(0);
    }

    /*
     * Print indication of arguments being passed to msgctl(), call
     * msgctl(), and report the results. If msgctl() fails, do not
     * return; this example doesn't deal with errors, it just reports
     * them.
     */
    static void
    do_msgctl(msqid, cmd, buf)
    struct msqid_ds    *buf;  /* pointer to queue descriptor buffer */
    int    cmd,  /* command code */
        msqid;  /* queue ID */
    {
     register int rtrn;  /* hold area for return value from msgctl()
    */

     (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d,
```

```
%s)\n",
   msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
 rtrn = msgctl(msqid, cmd, buf);
 if (rtrn == -1) {
  perror("msgctl: msgctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "msgctl: msgctl returned %d\n",
      rtrn);
 }
}
```

`msgop.c`: Sample Program to Illustrate `msgsnd()` and `msgrcv()`

```
/*
 * msgop.c: Illustrate the msgsnd() and msgrcv() functions.
 *
 * This is a simple exerciser of the message send and receive
 * routines. It allows the user to attempt to send and receive as
many
 * messages as wanted to or from one message queue.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

static int ask();
extern void exit();
extern char *malloc();
extern void perror();

char first_on_queue[] = "-> first message on queue",
 full_buf[] = "Message buffer overflow. Extra message text\
      discarded.";

main()
{
 register int    c;  /* message text input */
 int    choice; /* user's selected operation code */
 register int    i;  /* loop control for mtext */
 int    msgflg; /* message flags for the operation */
 struct msgbuf     *msgp;  /* pointer to the message buffer */
 int    msgsz;  /* message size */
 long    msgtyp;  /* desired message type */
 int    msqid,  /* message queue ID to be used */
     maxmsgsz, /* size of allocated message buffer */
     rtrn;  /* return value from msgrcv or msgsnd */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
```

```
 /* Get the message queue ID and set up the message buffer. */
 (void) fprintf(stderr, "Enter msqid: ");
 (void) scanf("%i", &msqid);
 /*
  * Note that <sys/msg.h> includes a definition of struct
msgbuf
  * with the mtext field defined as:
  *   char mtext[1];
  * therefore, this definition is only a template, not a
structure
  * definition that you can use directly, unless you want only
to
  * send and receive messages of 0 or 1 byte. To handle this,
  * malloc an area big enough to contain the template - the size
  * of the mtext template field + the size of the mtext field
  * wanted. Then you can use the pointer returned by malloc as a
  * struct msgbuf with an mtext field of the size you want. Note
  * also that sizeof msgp->mtext is valid even though msgp
isn't
  * pointing to anything yet. Sizeof doesn't dereference msgp,
but
  * uses its type to figure out what you are asking about.
  */
 (void) fprintf(stderr,
  "Enter the message buffer size you want:");
 (void) scanf("%i", &maxmsgsz);
 if (maxmsgsz < 0) {
  (void) fprintf(stderr, "msgop: %s\n",
    "The message buffer size must be >= 0.");
  exit(1);
 }
 msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct
msgbuf)
    - sizeof msgp->mtext + maxmsgsz));
 if (msgp == NULL) {
  (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
    "could not allocate message buffer for", maxmsgsz);
  exit(1);
 }
 /* Loop through message operations until the user is ready to
  quit. */
 while (choice = ask()) {
  switch (choice) {
  case 1: /* msgsnd() requested: Get the arguments, make the
    call, and report the results. */
   (void) fprintf(stderr, "Valid msgsnd message %s\n",
    "types are positive integers.");
   (void) fprintf(stderr, "Enter msgp->mtype: ");
   (void) scanf("%li", &msgp->mtype);
   if (maxmsgsz) {
    /* Since you've been using scanf, you need the loop
       below to throw away the rest of the input on the
       line after the entered mtype before you start
       reading the mtext. */
    while ((c = getchar()) != '\n' && c != EOF);
    (void) fprintf(stderr, "Enter a %s:\n",
        "one line message");
    for (i = 0; ((c = getchar()) != '\n'); i++) {
     if (i >= maxmsgsz) {
```

```
            (void) fprintf(stderr, "\n%s\n", full_buf);
            while ((c = getchar()) != '\n');
            break;
           }
           msgp->mtext[i] = c;
          }
          msgsz = i;
         } else
          msgsz = 0;
         (void) fprintf(stderr,"\nMeaningful msgsnd flag is:\n");
         (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
          IPC_NOWAIT);
         (void) fprintf(stderr, "Enter msgflg: ");
         (void) scanf("%i", &msgflg);
         (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
          "msgop: Calling msgsnd", msqid, msgsz, msgflg);
         (void) fprintf(stderr, "msgp->mtype = %ld\n",
             msgp->mtype);
         (void) fprintf(stderr, "msgp->mtext = \"");
         for (i = 0; i < msgsz; i++)
          (void) fputc(msgp->mtext[i], stderr);
          (void) fprintf(stderr, "\"\n");
          rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
          if (rtrn == -1)
           perror("msgop: msgsnd failed");
          else
           (void) fprintf(stderr,
              "msgop: msgsnd returned %d\n", rtrn);
          break;
        case 2: /* msgrcv() requested: Get the arguments, make the
             call, and report the results. */
         for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
            (void) scanf("%i", &msgsz))
          (void) fprintf(stderr, "%s (0 <= msgsz <= %d): ",
              "Enter msgsz", maxmsgsz);
         (void) fprintf(stderr, "msgtyp meanings:\n");
         (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
         (void) fprintf(stderr, "\t>0 %s of given type\n",
          first_on_queue);
         (void) fprintf(stderr, "\t<0 %s with type <= |msgtyp|\n",
             first_on_queue);
         (void) fprintf(stderr, "Enter msgtyp: ");
         (void) scanf("%li", &msgtyp);
         (void) fprintf(stderr,
             "Meaningful msgrcv flags are:\n");
         (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
             MSG_NOERROR);
         (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
             IPC_NOWAIT);
         (void) fprintf(stderr, "Enter msgflg: ");
         (void) scanf("%i", &msgflg);
         (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
             "msgop: Calling msgrcv", msqid, msgsz,
             msgtyp, msgflg);
         rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
         if (rtrn == -1)
          perror("msgop: msgrcv failed");
         else {
```

```
    (void) fprintf(stderr, "msgop: %s %d\n",
        "msgrcv returned", rtrn);
    (void) fprintf(stderr, "msgp->mtype = %ld\n",
        msgp->mtype);
    (void) fprintf(stderr, "msgp->mtext is: \"");
    for (i = 0; i < rtrn; i++)
     (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
   }
   break;
  default:
   (void) fprintf(stderr, "msgop: operation unknown\n");
   break;
  }
 }
 exit(0);
}

/*
 * Ask the user what to do next. Return the user's choice code.
 * Don't return until the user selects a valid choice.
 */
static
ask()
{
 int response; /* User's response. */

 do {
  (void) fprintf(stderr, "Your options are:\n");
  (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
  (void) fprintf(stderr, "\tmsgsnd =\t1\n");
  (void) fprintf(stderr, "\tmsgrcv =\t2\n");
  (void) fprintf(stderr, "Enter your choice: ");

  /* Preset response so "^D" will be interpreted as exit. */
  response = 0;
  (void) scanf("%i", &response);
 } while (response < 0 || response > 2);

 return(response);
}
```

Exercises

Exercise 12755

Write a 2 programs that will both send and messages and construct the following dialog between them

- (Process 1) Sends the message "Are you hearing me?"
- (Process 2) Receives the message and replies "Loud and Clear".
- (Process 1) Receives the reply and then says "I can hear you too".

Exercise 12756

Compile the programs `msgget.c`, `msgctl.c` and `msgop.c` and then

- investigate and understand fully the operations of the flags (access, creation *etc.* permissions) you can set interactively in the programs.
- Use the programs to:

- Send and receive messages of two different message `type`s.
- Place several messages on the queue and inquire about the state of the queue with `msgctl.c`. Add/delete a few messages (using `msgop.c` and perform the inquiry once more.
- Use `msgctl.c` to alter a message on the queue.
- Use `msgctl.c` to delete a message from the queue.

Exercise 12757

Write a *server* program and two *client* programs so that the *server* can communicate privately to *each client* individually via a *single* message queue.

Exercise 12758

Implement a *blocked* or *synchronous* method of message passing using signal interrupts.

Dave Marshall
1/5/1999

Subsections

IPC:Semaphores

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter. In the computer version, a semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it.

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

Semaphores can be operated on as individual units or as elements in a set. Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter weight semaphores are available in the threads library (see `man semaphore` and also Chapter 30.3) and POSIX semaphores (see below briefly). Threads library semaphores must be used with mapped memory . A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements.

In a similar fashion to message queues, the semaphore set must be initialized using `semget()`; the semaphore creator can change its ownership or permissions using `semctl()`; and semaphore operations are performed via the `semop()` function. These are now discussed below:

Initializing a Semaphore Set

The function `semget()` initializes or gains access to a semaphore. It is prototyped by:

```
int semget(key_t key, int nsems, int semflg);
```

When the call succeeds, it returns the semaphore ID (`semid`).

The `key` argument is a access value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed.

The `semflg` argument specifies the initial access permissions and creation control flags.

The following code illustrates the semget() function.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
...
key_t key; /* key to pass to semget() */
int semflg; /* semflg to pass tosemget() */
int nsems; /* nsems to pass to semget() */
int semid; /* return value from semget() */

...

key = ...
nsems = ...
semflg = ... ...
if ((semid = semget(key, nsems, semflg)) == -1) {
                perror("semget: semget failed");
                exit(1); }
else
   ...
```

Controlling Semaphores

`semctl()` changes permissions and other characteristics of a semaphore set. It is prototyped as follows:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

It must be called with a valid semaphore ID, `semid`. The `semnum` value selects a semaphore within an array by its index. The `cmd` argument is one of the following control flags:

GETVAL
: -- Return the value of a single semaphore.

SETVAL
: -- Set the value of a single semaphore. In this case, arg is taken as arg.val, an int.

GETPID
: -- Return the `PID` of the process that performed the last operation on the semaphore or array.

GETNCNT
: -- Return the number of processes waiting for the value of a semaphore to increase.

GETZCNT
: -- Return the number of processes waiting for the value of a particular semaphore to reach zero.

GETALL
: -- Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts (see below).

SETALL
: -- Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts.

IPC_STAT
: -- Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.

IPC_SET
: -- Set the effective user and group identification and permissions. In this case, `arg` is taken as `arg.buf`.

IPC_RMID
: -- Remove the specified semaphore set.

A process must have an effective user identification of owner, creator, or superuser to perform an `IPC_SET` or `IPC_RMID` command. Read and write permission is required as for the other control commands. The following code illustrates `semctl()`.

The fourth argument `union semun arg` is optional, depending upon the operation requested. If required it is of type `union semun`, which must be *explicitly* declared by the application program as:

```
          union semun {
```

```
                int val;
                struct semid_ds *buf;
                ushort *array;
           } arg;

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
                int val;
                struct semid_ds *buf;
                ushort *array;
           } arg;

int i;
int semnum = ....;
int cmd = GETALL; /* get value */



...
i = semctl(semid, semnum, cmd, arg);
if (i == -1) {
                 perror("semctl: semctl failed");
  exit(1);
 }
else
...
```

Semaphore Operations

`semop()` performs operations on a semaphore set. It is prototyped by:

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

The `semid` argument is the semaphore ID returned by a previous `semget()` call. The `sops` argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- The semaphore number
- The operation to be performed
- Control flags, if any.

The `sembuf` structure specifies a semaphore operation, as defined in <`sys/sem.h`>.

```
struct sembuf {
        ushort_t        sem_num;        /* semaphore number */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

The `nsops` argument specifies the length of the array, the maximum size of which is determined by the `SEMOPM` configuration option; this is the maximum number of operations allowed by a single semop() call, and is set to 10 by default. The operation to be performed is determined as follows:

- A positive integer increments the semaphore value by that amount.
- A negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether `IPC_NOWAIT` is in effect.
- A value of zero means to wait for the semaphore value to reach zero.

There are two control flags that can be used with `semop()`:

IPC_NOWAIT

-- Can be set for any operations in the array. Makes the function return without changing any semaphore value if any operation for which `IPC_NOWAIT` is set cannot be performed. The function fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

SEM_UNDO

-- Allows individual operations in the array to be undone when the process exits.

This function takes a pointer, `sops`, to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. To increment or decrement a semaphore requires write permission. When an operation fails, none of the semaphores is altered.

The process blocks (unless the `IPC_NOWAIT` flag is set), and remains blocked until:

- the semaphore operations can all finish, so the call succeeds,
- the process receives a signal, or
- the semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop()` call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the `SEM_UNDO` control flag makes `semop()` allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state. If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state. When performing a semaphore operation with `SEM_UNDO` in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancel to zero. When the undo structure reaches zero, it is removed.

NOTE: Using `SEM_UNDO` inconsistently can lead to excessive resource consumption because allocated undo structures might not be freed until the system is rebooted.

The following code illustrates the `semop()` function:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

...
int i;
int nsops; /* number of operations to do */
int semid; /* semid of semaphore set */
struct sembuf *sops; /* ptr to operations to perform */

...

if ((semid = semop(semid, sops, nsops)) == -1)
{
        perror("semop: semop failed");
 exit(1);
}
else
(void) fprintf(stderr, "semop: returned %d\n", i);
...
```

POSIX Semaphores: <semaphore.h>

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to twenty five semaphores. The POSIX semaphore functions are:

`sem_open()` -- Connects to, and optionally creates, a named semaphore

`sem_init()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_close()` -- Ends the connection to an open semaphore.

`sem_unlink()` -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

`sem_destroy()` -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_getvalue()` -- Copies the value of the semaphore into the specified integer.

`sem_wait(), sem_trywait()` -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

`sem_post()` -- Increments the count of the semaphore.

The basic operation of these functions is essence the same as described above, except note there are more specialised functions, here. These are not discussed further here and the reader is referred to the online `man` pages for further details.

`semaphore.c`: Illustration of simple semaphore passing

```
/* semaphore.c --- simple illustration of dijkstra's semaphore analogy
 *
 *   We fork() a  child process so that we have two processes running:
 *   Each process communicates via a semaphore.
 *   The respective process can only do its work (not much here)
 *   When it notices that the semaphore track is free when it returns to 0
 *   Each process must modify the semaphore accordingly
 */

 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/sem.h>

 union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          };

main()
{ int i,j;
  int pid;
  int semid; /* semid of semaphore set */
  key_t key = 1234; /* key to pass to semget() */
  int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
  int nsems = 1; /* nsems to pass to semget() */
  int nsops; /* number of operations to do */
  struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
  /* ptr to operations to perform */

  /* set up semaphore */
```

```
  (void) fprintf(stderr, "\nsemget: Setting up seamaphore: semget(%#lx, %\
%#o)\n",key, nsems, semflg);
   if ((semid = semget(key, nsems, semflg)) == -1) {
        perror("semget: semget failed");
        exit(1);
      } else
        (void) fprintf(stderr, "semget: semget succeeded: semid =\
%d\n", semid);


  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

if (pid == 0)
     { /* child */
       i = 0;


       while (i  < 3) {/* allow for 3 semaphore sets */

       nsops = 2;

       /* wait for semaphore to reach zero */

       sops[0].sem_num = 0; /* We only use one track */
       sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
       sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


       sops[1].sem_num = 0;
       sops[1].sem_op = 1; /* increment semaphore -- take control of track */
       sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

       /* Recap the call to be made. */

       (void) fprintf(stderr,"\nsemop:Child  Calling semop(%d, &sops, %d) with:",
semid, nsops);
       for (j = 0; j < nsops; j++)
        {
          (void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
          (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
          (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
        }

       /* Make the semop() call and report the results. */
        if ((j = semop(semid, sops, nsops)) == -1) {
                perror("semop: semop failed");
                }
            else
              {
                (void) fprintf(stderr, "\tsemop: semop returned %d\n", j);

                (void) fprintf(stderr, "\n\nChild Process Taking Control of Track:
%d/3 times\n", i+1);
                sleep(5); /* DO Nothing for 5 seconds */

                 nsops = 1;

                /* wait for semaphore to reach zero */
```

```
                sops[0].sem_num = 0;
                sops[0].sem_op = -1; /* Give UP COntrol of track */
                sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore,
asynchronous  */


                if ((j = semop(semid, sops, nsops)) == -1) {
                        perror("semop: semop failed");
                        }
                   else
                      (void) fprintf(stderr, "Child Process Giving up Control of
Track: %d/3 times\n", i+1);
                sleep(5); /* halt process to allow parent to catch semaphor change
first */
              }
        ++i;
      }

     }
  else /* parent */
     {  /* pid hold id of child */

        i = 0;


       while (i  < 3) { /* allow for 3 semaphore sets */

       nsops = 2;

       /* wait for semaphore to reach zero */
       sops[0].sem_num = 0;
       sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
       sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


       sops[1].sem_num = 0;
       sops[1].sem_op = 1; /* increment semaphore -- take control of track */
       sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

       /* Recap the call to be made. */

       (void) fprintf(stderr,"\nsemop:Parent Calling semop(%d, &sops, %d) with:",
semid, nsops);
       for (j = 0; j < nsops; j++)
        {
          (void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
          (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
          (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
        }

       /* Make the semop() call and report the results. */
        if ((j = semop(semid, sops, nsops)) == -1) {
                perror("semop: semop failed");
                }
           else
             {
                (void) fprintf(stderr, "semop: semop returned %d\n", j);

                (void) fprintf(stderr, "Parent Process Taking Control of Track: %d/3
times\n", i+1);
                sleep(5); /* Do nothing for 5 seconds */

                 nsops = 1;
```

```
                /* wait for semaphore to reach zero */
                sops[0].sem_num = 0;
                sops[0].sem_op = -1; /* Give UP COntrol of track */
                sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore,
asynchronous  */

                if ((j = semop(semid, sops, nsops)) == -1) {
                        perror("semop: semop failed");
                        }
                   else
                      (void) fprintf(stderr, "Parent Process Giving up Control of
Track: %d/3 times\n", i+1);
                sleep(5); /* halt process to allow child to catch semaphor change
first */
              }
        ++i;

      }

     }
}
```

The key elements of this program are as follows:

- After a semaphore is created with as simple key `1234`, two prcesses are forked.
- Each process (parent and child) essentially performs the same operations:
    - Each process accesses the same semaphore *track* (`sops[].sem_num = 0`).
    - Each process waits for the *track* to become free and then attempts to take control of *track*

      This is achieved by setting appropriate `sops[].sem_op` values in the array.
    - Once the process has control it sleeps for 5 seconds (in reality some processing would take place in place of this simple illustration)
    - The process then gives up control of the *track* `sops[1].sem_op = -1`
    - an additional sleep operation is then performed to ensure that the other process has time to access the semaphore before a subsequent (same process) semaphore read.

      Note: There is no synchronisation here in this simple example an we have no control over how the OS will schedule the processes.

Some further example semaphore programs

The following suite of programs can be used to investigate interactively a variety of semaphore ideas (see exercises below).

The semaphore **must** be initialised with the `semget.c` program. The effects of controlling the semaphore queue and sending and receiving semaphore can be investigated with `semctl.c` and `semop.c` respectively.

`semget.c`: Illustrate the `semget()` function

```
/*
 * semget.c: Illustrate the semget() function.
 *
 * This is a simple exerciser of the semget() function. It prompts
 * for the arguments, makes the call, and reports the results.
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>
```

```
extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to pass to semget() */
 int  semflg;   /* semflg to pass to semget() */
 int  nsems;   /* nsems to pass to semget() */
 int  semid;   /* return value from semget() */

 (void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 (void) fprintf(stderr, "Enter nsems value: ");
 (void) scanf("%i", &nsems);
 (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter semflg value: ");
 (void) scanf("%i", &semflg);
 (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %
     %#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
  perror("semget: semget failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "semget: semget succeeded: semid =
%d\n",
   semid);
  exit(0);
 }
}
```

`semctl.c`: Illustrate the `semctl()` function

```
/*
 * semctl.c:   Illustrate the semctl() function.
 *
 * This is a simple exerciser of the semctl() function. It lets you
 * perform one control operation on one semaphore set. It gives up
 * immediately if any control operation fails, so be careful not
to
 * set permissions to preclude read permission; you won't be able
to
 * reset the permissions with this code if you do.
 */
```

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <time.h>

struct semid_ds semid_ds;

static void   do_semctl();
static void   do_stat();
extern char   *malloc();
extern void   exit();
extern void   perror();

char    warning_message[] = "If you remove read permission\
    for yourself, this program will fail frequently!";

main()
{
 union semun    arg;    /* union to pass to semctl() */
 int    cmd,    /* command to give to semctl() */
     i,    /* work area */
     semid,    /* semid to pass to semctl() */
     semnum;    /* semnum to pass to semctl() */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "Enter semid value: ");
 (void) scanf("%i", &semid);

 (void) fprintf(stderr, "Valid semctl cmd values are:\n");
 (void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
 (void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
 (void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
 (void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
 (void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
 (void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
 (void) fprintf(stderr, "\nEnter cmd: ");
 (void) scanf("%i", &cmd);

 /* Do some setup operations needed by multiple commands. */
 switch (cmd) {
  case GETVAL:
  case SETVAL:
  case GETNCNT:
  case GETZCNT:
   /* Get the semaphore number for these commands. */
   (void) fprintf(stderr, "\nEnter semnum value: ");
   (void) scanf("%i", &semnum);
   break;
  case GETALL:
  case SETALL:
   /* Allocate a buffer for the semaphore values. */
   (void) fprintf(stderr,
    "Get number of semaphores in the set.\n");
```

```
      arg.buf = &semid_ds;
      do_semctl(semid, 0, IPC_STAT, arg);
      if (arg.array =
       (ushort *)malloc((unsigned)
        (semid_ds.sem_nsems * sizeof(ushort)))) {
       /* Break out if you got what you needed. */
       break;
      }
      (void) fprintf(stderr,
       "semctl: unable to allocate space for %d values\n",
       semid_ds.sem_nsems);
      exit(2);
    }

    /* Get the rest of the arguments needed for the specified
       command. */
    switch (cmd) {
     case SETVAL:
      /* Set value of one semaphore. */
      (void) fprintf(stderr, "\nEnter semaphore value: ");
      (void) scanf("%i", &arg.val);
      do_semctl(semid, semnum, SETVAL, arg);
      /* Fall through to verify the result. */
      (void) fprintf(stderr,
       "Do semctl GETVAL command to verify results.\n");
     case GETVAL:
      /* Get value of one semaphore. */
      arg.val = 0;
      do_semctl(semid, semnum, GETVAL, arg);
      break;
     case GETPID:
      /* Get PID of last process to successfully complete a
         semctl(SETVAL), semctl(SETALL), or semop() on the
         semaphore. */
      arg.val = 0;
      do_semctl(semid, 0, GETPID, arg);
      break;
     case GETNCNT:
      /* Get number of processes waiting for semaphore value to
         increase. */
      arg.val = 0;
      do_semctl(semid, semnum, GETNCNT, arg);
      break;
     case GETZCNT:
      /* Get number of processes waiting for semaphore value to
         become zero. */
      arg.val = 0;
      do_semctl(semid, semnum, GETZCNT, arg);
      break;
     case SETALL:
      /* Set the values of all semaphores in the set. */
      (void) fprintf(stderr,
          "There are %d semaphores in the set.\n",
          semid_ds.sem_nsems);
      (void) fprintf(stderr, "Enter semaphore values:\n");
      for (i = 0; i < semid_ds.sem_nsems; i++) {
       (void) fprintf(stderr, "Semaphore %d: ", i);
       (void) scanf("%hi", &arg.array[i]);
      }
      do_semctl(semid, 0, SETALL, arg);
      /* Fall through to verify the results. */
      (void) fprintf(stderr,
       "Do semctl GETALL command to verify results.\n");
```

```
    case GETALL:
     /* Get and print the values of all semaphores in the
        set.*/
     do_semctl(semid, 0, GETALL, arg);
     (void) fprintf(stderr,
         "The values of the %d semaphores are:\n",
         semid_ds.sem_nsems);
     for (i = 0; i < semid_ds.sem_nsems; i++)
      (void) fprintf(stderr, "%d ", arg.array[i]);
     (void) fprintf(stderr, "\n");
     break;
    case IPC_SET:
     /* Modify mode and/or ownership. */
     arg.buf = &semid_ds;
     do_semctl(semid, 0, IPC_STAT, arg);
     (void) fprintf(stderr, "Status before IPC_SET:\n");
     do_stat();
     (void) fprintf(stderr, "Enter sem_perm.uid value: ");
     (void) scanf("%hi", &semid_ds.sem_perm.uid);
     (void) fprintf(stderr, "Enter sem_perm.gid value: ");
     (void) scanf("%hi", &semid_ds.sem_perm.gid);
     (void) fprintf(stderr, "%s\n", warning_message);
     (void) fprintf(stderr, "Enter sem_perm.mode value: ");
     (void) scanf("%hi", &semid_ds.sem_perm.mode);
     do_semctl(semid, 0, IPC_SET, arg);
     /* Fall through to verify changes. */
     (void) fprintf(stderr, "Status after IPC_SET:\n");
    case IPC_STAT:
     /* Get and print current status. */
     arg.buf = &semid_ds;
     do_semctl(semid, 0, IPC_STAT, arg);
     do_stat();
     break;
    case IPC_RMID:
     /* Remove the semaphore set. */
     arg.val = 0;
     do_semctl(semid, 0, IPC_RMID, arg);
     break;
    default:
     /* Pass unknown command to semctl. */
     arg.val = 0;
     do_semctl(semid, 0, cmd, arg);
     break;
   }
   exit(0);
  }

  /*
   * Print indication of arguments being passed to semctl(), call
   * semctl(), and report the results. If semctl() fails, do not
   * return; this example doesn't deal with errors, it just reports
   * them.
   */
  static void
  do_semctl(semid, semnum, cmd, arg)
  union semun  arg;
  int  cmd,
    semid,
    semnum;
  {
   register int      i;   /* work area */

   void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d,
```

```
",
    semid, semnum, cmd);
 switch (cmd) {
  case GETALL:
   (void) fprintf(stderr, "arg.array = %#x)\n",
       arg.array);
   break;
  case IPC_STAT:
  case IPC_SET:
   (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
   break;
  case SETALL:
   (void) fprintf(stderr, "arg.array = [", arg.buf);
   for (i = 0;i < semid_ds.sem_nsems;) {
    (void) fprintf(stderr, "%d", arg.array[i++]);
    if (i < semid_ds.sem_nsems)
      (void) fprintf(stderr, ", ");
   }
   (void) fprintf(stderr, "])\n");
   break;
  case SETVAL:
  default:
   (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
   break;
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
 }
 (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}

/*
 * Display contents of commonly used pieces of the status
structure.
 */
static void
do_stat()
{
 (void) fprintf(stderr, "sem_perm.uid = %d\n",
       semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "sem_perm.gid = %d\n",
       semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "sem_perm.cuid = %d\n",
       semid_ds.sem_perm.cuid);
 (void) fprintf(stderr, "sem_perm.cgid = %d\n",
       semid_ds.sem_perm.cgid);
 (void) fprintf(stderr, "sem_perm.mode = %#o, ",
       semid_ds.sem_perm.mode);
 (void) fprintf(stderr, "access permissions = %#o\n",
       semid_ds.sem_perm.mode & 0777);
 (void) fprintf(stderr, "sem_nsems = %d\n",
semid_ds.sem_nsems);
 (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
      ctime(&semid_ds.sem_otime) : "Not Set\n");
 (void) fprintf(stderr, "sem_ctime = %s",
       ctime(&semid_ds.sem_ctime));
}
```

semop() Sample Program to Illustrate semop()

```
/*
 * semop.c: Illustrate the semop() function.
 *
 * This is a simple exerciser of the semop() function. It lets you
 * to set up arguments for semop() and make the call. It then
reports
 * the results repeatedly on one semaphore set. You must have read
 * permission on the semaphore set or this exerciser will fail.
(It
 * needs read permission to get the number of semaphores in the set
 * and to report the values before and after calls to semop().)
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

static int      ask();
extern void     exit();
extern void     free();
extern char     *malloc();
extern void     perror();

static struct semid_ds  semid_ds;            /* status of semaphore set */

static char    error_mesg1[] = "semop: Can't allocate space for %d\
        semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can't allocate space for %d\
        sembuf structures. Giving up.\n";

main()
{
 register int    i;   /* work area */
 int    nsops;   /* number of operations to do */
 int    semid;   /* semid of semaphore set */
 struct sembuf    *sops;   /* ptr to operations to perform */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Loop until the invoker doesn't want to do anymore. */
 while (nsops = ask(&semid, &sops)) {
  /* Initialize the array of operations to be performed.*/
  for (i = 0; i < nsops; i++) {
   (void) fprintf(stderr,
     "\nEnter values for operation %d of %d.\n",
       i + 1, nsops);
   (void) fprintf(stderr,
     "sem_num(valid values are 0 <= sem_num < %d): ",
     semid_ds.sem_nsems);
   (void) scanf("%hi", &sops[i].sem_num);
   (void) fprintf(stderr, "sem_op: ");
   (void) scanf("%hi", &sops[i].sem_op);
   (void) fprintf(stderr,
     "Expected flags in sem_flg are:\n");
```

```
     (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
       IPC_NOWAIT);
     (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
       SEM_UNDO);
     (void) fprintf(stderr, "sem_flg: ");
     (void) scanf("%hi", &sops[i].sem_flg);
    }

    /* Recap the call to be made. */
    (void) fprintf(stderr,
       "\nsemop: Calling semop(%d, &sops, %d) with:",
       semid, nsops);
    for (i = 0; i < nsops; i++)
    {
     (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
        sops[i].sem_num);
     (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
     (void) fprintf(stderr, "sem_flg = %#o\n",
        sops[i].sem_flg);
    }

    /* Make the semop() call and report the results. */
    if ((i = semop(semid, sops, nsops)) == -1) {
     perror("semop: semop failed");
    } else {
     (void) fprintf(stderr, "semop: semop returned %d\n", i);
    }
   }
  }

  /*
   * Ask if user wants to continue.
   *
   * On the first call:
   * Get the semid to be processed and supply it to the caller.
   * On each call:
   *  1. Print current semaphore values.
   *  2. Ask user how many operations are to be performed on the next
   *     call to semop. Allocate an array of sembuf structures
   *     sufficient for the job and set caller-supplied pointer to
  that
   *     array. (The array is reused on subsequent calls if it is big
   *     enough. If it isn't, it is freed and a larger array is
   *     allocated.)
   */
  static
  ask(semidp, sopsp)
  int   *semidp;  /* pointer to semid (used only the first time) */
  struct sembuf   **sopsp;
  {
   static union semun     arg;  /* argument to semctl */
   int     i;  /* work area */
   static int     nsops = 0;  /* size of currently allocated
             sembuf array */
   static int     semid = -1;   /* semid supplied by user */
   static struct sembuf    *sops;     /* pointer to allocated array */

   if (semid < 0) {
    /* First call; get semid from user and the current state of
       the semaphore set. */
    (void) fprintf(stderr,
      "Enter semid of the semaphore set you want to use: ");
    (void) scanf("%i", &semid);
```

```
  *semidp = semid;
  arg.buf = &semid_ds;
  if (semctl(semid, 0, IPC_STAT, arg) == -1) {
   perror("semop: semctl(IPC_STAT) failed");
   /* Note that if semctl fails, semid_ds remains filled
      with zeros, so later test for number of semaphores will
      be zero. */
   (void) fprintf(stderr,
     "Before and after values are not printed.\n");
  } else {
   if ((arg.array = (ushort *)malloc(
    (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
       == NULL) {
    (void) fprintf(stderr, error_mesg1,
       semid_ds.sem_nsems);
    exit(1);
   }
  }
 }
 /* Print current semaphore values. */
 if (semid_ds.sem_nsems) {
  (void) fprintf(stderr,
      "There are %d semaphores in the set.\n",
      semid_ds.sem_nsems);
  if (semctl(semid, 0, GETALL, arg) == -1) {
   perror("semop: semctl(GETALL) failed");
  } else {
   (void) fprintf(stderr, "Current semaphore values are:");
   for (i = 0; i < semid_ds.sem_nsems;
    (void) fprintf(stderr, " %d", arg.array[i++]));
   (void) fprintf(stderr, "\n");
  }
 }
 /* Find out how many operations are going to be done in the
next
    call and allocate enough space to do it. */
 (void) fprintf(stderr,
     "How many semaphore operations do you want %s\n",
     "on the next call to semop()?");
 (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
 i = 0;
 if (scanf("%i", &i) == EOF || i == 0)
  exit(0);
 if (i > nsops) {
  if (nsops)
   free((char *)sops);
  nsops = i;
  if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
   sizeof(struct sembuf)))) == NULL) {
   (void) fprintf(stderr, error_mesg2, nsops);
   exit(2);
  }
 }
 *sopsp = sops;
 return (i);
}
```

Exercises

Exercise 12763

Write 2 programs that will communicate **both ways** (*i.e* each process can read and write) when run concurrently via semaphores.

Exercise 12764

Modify the `semaphore.c` program to handle synchronous semaphore communication semaphores.

Exercise 12765

Write 3 programs that communicate together via semaphores according to the following specifications: `sem_server.c` -- a program that can communicate independently (on different semaphore tracks) with two clients programs. `sem_client1.c` -- a program that talks to `sem_server.c` on one track. `sem_client2.c` -- a program that talks to `sem_server.c` on another track to `sem_client1.c`.

Exercise 12766

Compile the programs `semget.c`, `semctl.c` and `semop.c` and then

- investigate and understand fully the operations of the flags (access, creation *etc.* permissions) you can set interactively in the programs.
- Use the prgrams to:
  - Send and receive semaphores of 3 different semaphore `track`s.
  - Inquire about the state of the semaphore queue with `semctl.c`. Add/delete a few semaphores (using `semop.c` and perform the inquiry once more.
  - Use `semctl.c` to alter a semaphore on the queue.
  - Use `semctl.c` to delete a semaphore from the queue.

Dave Marshall
1/5/1999

Subsections

IPC:Shared Memory

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

In the Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on the `mmap()` function and on the system's native virtual memory facility. Solaris 2.x also supports System V shared memory, which is another way to let multiple processes attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using `shmget()|`. The original owner of a shared memory segment can assign ownership to another user with `shmctl()`. It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`. Once created, a shared segment can be attached to a process address space using `shmat()`. It can be detached using `shmdt()` (see `shmop()`). The attaching process must have the appropriate permissions for `shmat()`. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in `<sys/shm.h>`.

Accessing a Shared Memory Segment

`shmget()` is used to obtain access to a shared memory segment. It is prottyped by:

```
int shmget(key_t key, size_t size, int shmflg);
```

The `key` argument is a access value associated with the semaphore ID. The `size` argument is the size in bytes of the requested shared memory. The `shmflg` argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

The following code illustrates `shmget()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

...

key_t key; /* key to be passed to shmget() */
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */

...

key = ...
size = ...
shmflg) = ...

if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed"); exit(1); } else {
   (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
...
```

Controlling a Shared Memory Segment

`shmctl()` is used to alter the permissions and other characteristics of a shared memory segment. It is prototyped as follows:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The process must have an effective `shmid` of owner, creator or superuser to perform this command. The `cmd` argument is one of following control commands:

SHM_LOCK

-- Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.

SHM_UNLOCK

-- Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.

IPC_STAT

-- Return the status information contained in the control structure and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.

IPC_SET

-- Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.

IPC_RMID

-- Remove the shared memory segment.

The `buf` is a sructure of type `struct shmid_ds` which is defined in <`sys/shm.h`>

The following code illustrates `shmctl()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

...

int cmd; /* command code for shmctl() */
int shmid; /* segment ID */
struct shmid_ds shmid_ds; /* shared memory data structure to
                             hold results */
...

shmid = ...
cmd = ...
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
    perror("shmctl: shmctl failed");
    exit(1);
   }
...
```

Attaching and Detaching a Shared Memory Segment

`shmat()` and `shmdt()` are used to attach and detach shared memory segments. They are prototypes as follows:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

`shmat()` returns a pointer, `shmaddr`, to the head of the shared segment associated with a valid `shmid`. `shmdt()` detaches the shared memory segment located at the address indicated by `shmaddr`

. The following code illustrates calls to `shmat()` and `shmdt()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

static struct state { /* Internal record of attached segments. */
          int shmid; /* shmid of attached segment */
          char *shmaddr; /* attach point */
          int shmflg; /* flags used on attach */
         } ap[MAXnap]; /* State of current attached segments. */
int nap; /* Number of currently attached segments. */

...

char *addr; /* address work variable */
register int i; /* work area */
register struct state *p; /* ptr to current state entry */
...

p = &ap[nap++];
p->shmid = ...
p->shmaddr = ...
p->shmflg = ...

p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1) {
     perror("shmop: shmat failed");
     nap--;
    } else
    (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
p->shmaddr);

...
i = shmdt(addr);
if(i == -1) {
    perror("shmop: shmdt failed");
    } else {
  (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);

for (p = ap, i = nap; i--; p++)
  if (p->shmaddr == addr) *p = ap[--nap];

}
...
```

Example two processes comunicating via shared memory: `shm_server.c`, `shm_client.c`

We develop two programs here that illustrate the passing of a simple piece of memery (a string) between the processes if running simulatenously:

shm_server.c

-- simply creates the string and shared memory portion.

shm_client.c

-- attaches itself to the created shared memory portion and uses the string (`printf`.

The code listings of the 2 programs no follow:

shm_server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now put some things into the memory for the
     * other process to read.
     */
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
```

```
        *s++ = c;
    *s = NULL;

    /*
     * Finally, we wait until the other process
     * changes the first character of our memory
     * to '*', indicating that it has read what
     * we put there.
     */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

shm_client.c

```
/*
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
```

```
        exit(1);
    }

    /*
     * Now read what the server put in the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory. The major differences are to use `shm_open()` to open the shared memory object (instead of calling `open()`) and use `shm_unlink()` to close and delete the object (instead of calling `close()` which does not remove the object). The options in `shm_open()` are substantially fewer than the number of options provided in `open()`.

Mapped memory

In a system with fixed memory (non-virtual), the address space of a process occupies and is limited to a portion of the system's main memory. In Solaris 2.x virtual memory the actual address space of a process occupies a file in the swap partition of disk storage (the file is called the backing store). Pages of main memory buffer the active (or recently active) portions of the process address space to provide code for the CPU(s) to execute and data for the program to process.

A page of address space is loaded when an address that is not currently in memory is accessed by a CPU, causing a page fault. Since execution cannot continue until the page fault is resolved by reading the referenced address segment into memory, the process sleeps until the page has been read. The most obvious difference between the two memory systems for the application developer is that virtual memory lets applications occupy much larger address spaces. Less obvious advantages of virtual memory are much simpler and more efficient file I/O and very efficient sharing of memory between processes.

Address Spaces and Mapping

Since backing store files (the process address space) exist only in swap storage, they are not included in the UNIX named file space. (This makes backing store files inaccessible to other processes.) However, it is a simple extension to allow the logical insertion of all, or part, of one, or more, named files in the backing store and to treat the result as a single address space. This is called mapping. With mapping, any part of any readable or writable file can be logically included in a process's address space. Like any other portion of the process's address space, no page of the file is not actually loaded into memory until a page fault forces this action. Pages of memory are written to the file only if their contents have been modified.

So, reading from and writing to files is completely automatic and very efficient. More than one process can map a single named file. This provides very efficient memory sharing between processes. All or part of other files can also be shared between processes.

Not all named file system objects can be mapped. Devices that cannot be treated as storage, such as terminal and network device files, are examples of objects that cannot be mapped. A process address space is defined by all of the files (or portions of files) mapped into the address space. Each mapping is sized and aligned to the page boundaries of the system on which the process is executing. There is no memory associated with processes themselves.

A process page maps to only one object at a time, although an object address may be the subject of many process mappings. The notion of a "page" is not a property of the mapped object. Mapping an object only provides the potential for a process to read or write the object's contents. Mapping makes the object's contents directly addressable by a process. Applications can access the storage resources they use directly rather than indirectly through read and write. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the read, modify buffer, write cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Because the file system name space includes any directory trees that are connected from other systems via NFS, any networked file can also be mapped into a process's address space.

Coherence

Whether to share memory or to share data contained in the file, when multiple process map a file simultaneously there may be problems with simultaneous access to data elements. Such processes can cooperate through any of the synchronization mechanisms provided in Solaris 2.x. Because they are very light weight, the most efficient synchronization mechanisms in Solaris 2.x are the threads library ones.

Creating and Using Mappings

`mmap()` establishes a mapping of a named file system object (or part of one) into a process address space. It is the basic memory management function and it is very simple.

- First `open()` the file, then
- `mmap()` it with appropriate access and sharing options
- Away you go.

`mmap` is prototypes as follows:

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
          int fildes, off_t off);
```

The mapping established by `mmap()` replaces any previous mappings for specified address range. The `flags` `MAP_SHARED` and `MAP_PRIVATE` specify the mapping type, and one of them must be specified. `MAP_SHARED` specifies that writes modify the mapped object. No further operations on the object are needed to make the change. `MAP_PRIVATE` specifies that an initial write to the mapped area creates a copy of the page and all writes reference the copy. Only modified pages are copied.

A mapping type is retained across a `fork()`. The file descriptor used in a mmap call need not be kept open after the mapping is established. If it is closed, the mapping remains until the mapping is undone by munmap() or be replacing in with a new mapping. If a mapped file is shortened by a call to truncate, an access to the area of the file that no longer exists causes a SIGBUS signal.

The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address that the system chooses.:

```
int fd;
caddr_t result;
if ((fd = open("/dev/zero", O_RDWR)) == -1)
   return ((caddr_t)-1);

result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
(void) close(fd);
```

Other Memory Control Functions

int mlock(caddr_t addr, size_t len) causes the pages in the specified address range to be locked in physical memory. References to locked pages (in this or other processes) do not result in page faults that require an I/O operation. This operation ties up physical resources and can disrupt normal system operation, so, use of mlock() is limited to the superuser. The system lets only a configuration dependent limit of pages be locked in memory. The call to mlock fails if this limit is exceeded.

int munlock(caddr_t addr, size_t len) releases the locks on physical pages. If multiple mlock() calls are made on an address range of a single mapping, a single munlock call is release the locks. However, if different mappings to the same pages are mlocked, the pages are not unlocked until the locks on all the mappings are released. Locks are also released when a mapping is removed, either through being replaced with an mmap operation or removed with munmap. A lock is transferred between pages on the ``copy-on-write' event associated with a MAP_PRIVATE mapping, thus locks on an address range that includes MAP_PRIVATE mappings will be retained transparently along with the copy-on-write redirection (see mmap above for a discussion of this redirection)

int mlockall(int flags) and int munlockall(void) are similar to mlock() and munlock(), but they operate on entire address spaces. mlockall() sets locks on all pages in the address space and munlockall() removes all locks on all pages in the address space, whether established by mlock or mlockall.

int msync(caddr_t addr, size_t len, int flags) causes all modified pages in the specified address range to be flushed to the objects mapped by those addresses. It is similar to fsync() for files.

long sysconf(int name) returns the system dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set.

int mprotect(caddr_t addr, size_t len, int prot) assigns the specified protection to all pages in the specified address range. The protection cannot exceed the permissions allowed on the underlying object.

int brk(void *endds) and void *sbrk(int incr) are called to add storage to the data segment of a process. A process can manipulate this area by calling brk() and sbrk(). brk() sets the system idea of the lowest data segment location not used by the caller to addr (rounded up to the next multiple of the system page size). sbrk() adds incr bytes to the caller data space and returns a pointer to the start of the new data area.

Some further example shared memory programs

The following suite of programs can be used to investigate interactively a variety of shared ideas (see exercises below).

The semaphore **must** be initialised with the `shmget.c` program. The effects of controlling shared memory and accessing can be investigated with `shmctl.c` and `shmop.c` respectively.

`shmget.c`:Sample Program to Illustrate shmget()

```
/*
 * shmget.c: Illustrate the shmget() function.
 *
 * This is a simple exerciser of the shmget() function. It
prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to be passed to shmget() */
 int  shmflg;   /* shmflg to be passed to shmget() */
 int  shmid;   /* return value from shmget() */
 int  size;   /* size to be passed to shmget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the key. */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* Get the size of the segment. */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);
```

```
 /* Get the shmflg value. */
 (void) fprintf(stderr,
    "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* Make the call and report the results. */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
  perror("shmget: shmget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
     "shmget: shmget returned %d\n", shmid);
  exit(0);
 }
}
```

`shmctl.c`: Sample Program to Illustrate `shmctl()`

```
/*
 * shmctl.c: Illustrate the shmctl() function.
 *
 * This is a simple exerciser of the shmctl() function. It lets you
 * to perform one control operation on one shared memory segment.
 * (Some operations are done for the user whether requested or
not.
 * It gives up immediately if any control operation fails. Be
careful
 * not to set permissions to preclude read permission; you won't
be
 *able to reset the permissions with this code if you do.)
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   <time.h>
static void   do_shmctl();
extern void   exit();
```

```
extern void   perror();

main()
{
 int  cmd;  /* command code for shmctl() */
 int  shmid;  /* segment ID */
 struct shmid_ds  shmid_ds;     /* shared memory data structure to
         hold results */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
     "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get shmid and cmd. */
 (void) fprintf(stderr,
     "Enter the shmid for the desired segment: ");
 (void) scanf("%i", &shmid);
 (void) fprintf(stderr, "Valid shmctl cmd values are:\n");
 (void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
 (void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
 (void) fprintf(stderr, "Enter the desired cmd value: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_STAT:
   /* Get shared memory segment status. */
   break;
  case IPC_SET:
   /* Set owner UID and GID and permissions. */
   /* Get and print current values. */
   do_shmctl(shmid, IPC_STAT, &shmid_ds);
   /* Set UID, GID, and permissions to be loaded. */
   (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.uid);
   (void) fprintf(stderr, "Enter shm_perm.gid: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.gid);
   (void) fprintf(stderr,
    "Note: Keep read permission for yourself.\n");
   (void) fprintf(stderr, "Enter shm_perm.mode: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.mode);
   break;
  case IPC_RMID:
   /* Remove the segment when the last attach point is
      detached. */
   break;
  case SHM_LOCK:
   /* Lock the shared memory segment. */
   break;
```

```
  case SHM_UNLOCK:
   /* Unlock the shared memory segment. */
   break;
  default:
   /* Unknown command will be passed to shmctl. */
   break;
 }
 do_shmctl(shmid, cmd, &shmid_ds);
 exit(0);
}

/*
 * Display the arguments being passed to shmctl(), call shmctl(),
 * and report the results. If shmctl() fails, do not return; this
 * example doesn't deal with errors, it just reports them.
 */
static void
do_shmctl(shmid, cmd, buf)
int   shmid,   /* attach point */
   cmd;   /* command code */
struct shmid_ds   *buf;   /* pointer to shared memory data structure */
{
 register int    rtrn;  /* hold area */

 (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d,
buf)\n",
  shmid, cmd);
 if (cmd == IPC_SET) {
  (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
     buf->shm_perm.uid);
  (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
     buf->shm_perm.gid);
  (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
     buf->shm_perm.mode);
 }
 if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
  perror("shmctl: shmctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
      "shmctl: shmctl returned %d\n", rtrn);
 }
 if (cmd != IPC_STAT && cmd != IPC_SET)
  return;

 /* Print the current status. */
 (void) fprintf(stderr, "\nCurrent status:\n");
 (void) fprintf(stderr, "\tshm_perm.uid = %d\n",
      buf->shm_perm.uid);
 (void) fprintf(stderr, "\tshm_perm.gid = %d\n",
      buf->shm_perm.gid);
 (void) fprintf(stderr, "\tshm_perm.cuid = %d\n",
      buf->shm_perm.cuid);
```

```
 (void) fprintf(stderr, "\tshm_perm.cgid = %d\n",
      buf->shm_perm.cgid);
 (void) fprintf(stderr, "\tshm_perm.mode = %#o\n",
      buf->shm_perm.mode);
 (void) fprintf(stderr, "\tshm_perm.key = %#x\n",
      buf->shm_perm.key);
 (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
 (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
 (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
 (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
 (void) fprintf(stderr, "\tshm_atime = %s",
   buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_dtime = %s",
   buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_ctime = %s",
      ctime(&buf->shm_ctime));
}
```

shmop.c: Sample Program to Illustrate shmat() and shmdt()

```
/*
 * shmop.c: Illustrate the shmat() and shmdt() functions.
 *
 * This is a simple exerciser for the shmat() and shmdt() system
 * calls. It allows you to attach and detach segments and to
 * write strings into and read strings from attached segments.
 */

#include   <stdio.h>
#include   <setjmp.h>
#include   <signal.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

#define   MAXnap  4 /* Maximum number of concurrent attaches. */

static   ask();
static void  catcher();
extern void  exit();
static   good_addr();
extern void  perror();
extern char  *shmat();

static struct state     { /* Internal record of currently attached
segments. */
 int  shmid;   /* shmid of attached segment */
 char  *shmaddr;   /* attach point */
 int  shmflg;   /* flags used on attach */
} ap[MAXnap];     /* State of current attached segments. */
```

```
static int    nap;  /* Number of currently attached segments. */
static jmp_buf   segvbuf;   /* Process state save area for SIGSEGV
         catching. */

main()
{
 register int    action;   /* action to be performed */
 char    *addr;   /* address work area */
 register int    i;   /* work area */
 register struct state   *p;    /* ptr to current state entry */
 void   (*savefunc)();  /* SIGSEGV state hold area */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 while (action = ask()) {
  if (nap) {
   (void) fprintf(stderr,
      "\nCurrently attached segment(s):\n");
   (void) fprintf(stderr, " shmid address\n");
   (void) fprintf(stderr, "------ ----------\n");
   p = &ap[nap];
   while (p-- != ap) {
    (void) fprintf(stderr, "%6d", p->shmid);
    (void) fprintf(stderr, "%#11x", p->shmaddr);
    (void) fprintf(stderr, " Read%s\n",
     (p->shmflg & SHM_RDONLY) ?
     "-Only" : "/Write");
   }
  } else
   (void) fprintf(stderr,
    "\nNo segments are currently attached.\n");
  switch (action) {
  case 1:   /* Shmat requested. */
   /* Verify that there is space for another attach. */
   if (nap == MAXnap) {
    (void) fprintf(stderr, "%s %d %s\n",
       "This simple example will only allow",
       MAXnap, "attached segments.");
    break;
   }
   p = &ap[nap++];
   /* Get the arguments, make the call, report the
    results, and update the current state array. */
   (void) fprintf(stderr,
    "Enter shmid of segment to attach: ");
   (void) scanf("%i", &p->shmid);

   (void) fprintf(stderr, "Enter shmaddr: ");
   (void) scanf("%i", &p->shmaddr);
   (void) fprintf(stderr,
```

```
     "Meaningful shmflg values are:\n");
    (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
     SHM_RDONLY);
    (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
     SHM_RND);
    (void) fprintf(stderr, "Enter shmflg value: ");
    (void) scanf("%i", &p->shmflg);

    (void) fprintf(stderr,
     "shmop: Calling shmat(%d, %#x, %#o)\n",
     p->shmid, p->shmaddr, p->shmflg);
    p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
    if(p->shmaddr == (char *)-1) {
     perror("shmop: shmat failed");
     nap--;
    } else {
     (void) fprintf(stderr,
      "shmop: shmat returned %#8.8x\n",
      p->shmaddr);
    }
    break;

   case 2:   /* Shmdt requested. */
    /* Get the address, make the call, report the results,
     and make the internal state match. */
    (void) fprintf(stderr,
     "Enter detach shmaddr: ");
    (void) scanf("%i", &addr);

    i = shmdt(addr);
    if(i == -1) {
     perror("shmop: shmdt failed");
    } else {
     (void) fprintf(stderr,
      "shmop: shmdt returned %d\n", i);
     for (p = ap, i = nap; i--; p++) {
      if (p->shmaddr == addr)
       *p = ap[--nap];
     }
    }
    break;
   case 3: /* Read from segment requested. */
    if (nap == 0)
     break;

    (void) fprintf(stderr, "Enter address of an %s",
     "attached segment: ");
    (void) scanf("%i", &addr);

    if (good_addr(addr))
     (void) fprintf(stderr, "String @ %#x is `%s'\n",
      addr, addr);
    break;
```

```
  case 4: /* Write to segment requested. */
   if (nap == 0)
    break;

   (void) fprintf(stderr, "Enter address of an %s",
    "attached segment: ");
   (void) scanf("%i", &addr);

   /* Set up SIGSEGV catch routine to trap attempts to
    write into a read-only attached segment. */
   savefunc = signal(SIGSEGV, catcher);

   if (setjmp(segvbuf)) {
    (void) fprintf(stderr, "shmop: %s: %s\n",
     "SIGSEGV signal caught",
     "Write aborted.");
   } else {
    if (good_addr(addr)) {
     (void) fflush(stdin);
     (void) fprintf(stderr, "%s %s %#x:\n",
      "Enter one line to be copied",
      "to shared segment attached @",
      addr);
     (void) gets(addr);
    }
   }
   (void) fflush(stdin);

   /* Restore SIGSEGV to previous condition. */
   (void) signal(SIGSEGV, savefunc);
   break;
  }
 }
 exit(0);
 /*NOTREACHED*/
}
/*
** Ask for next action.
*/
static
ask()
{
 int  response;   /* user response */
 do {
   (void) fprintf(stderr, "Your options are:\n");
   (void) fprintf(stderr, "\t^D = exit\n");
   (void) fprintf(stderr, "\t 0 = exit\n");
   (void) fprintf(stderr, "\t 1 = shmat\n");
   (void) fprintf(stderr, "\t 2 = shmdt\n");
   (void) fprintf(stderr, "\t 3 = read from segment\n");
   (void) fprintf(stderr, "\t 4 = write to segment\n");
   (void) fprintf(stderr,
    "Enter the number corresponding to your choice: ");
```

```
   /* Preset response so "^D" will be interpreted as exit. */
   response = 0;
   (void) scanf("%i", &response);
 } while (response < 0 || response > 4);
 return (response);
}
/*
** Catch signal caused by attempt to write into shared memory
segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
{
 longjmp(segvbuf, 1);
 /*NOTREACHED*/
}
/*
** Verify that given address is the address of an attached
segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char *address;
{
 register struct state          *p;   /* ptr to state of attached
segment */

 for (p = ap; p != &ap[nap]; p++)
   if (p->shmaddr == address)
    return(1);
 return(0);
}
```

Exercises

Exercise 12771

Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronise and notify each process when operations such as memory loaded and memory read have been performed.

Exercise 12772

Compile the programs `shmget.c`, `shmctl.c` and `shmop.c` and then

- investigate and understand fully the operations of the flags (access, creation *etc.* permissions) you can set interactively in the programs.
- Use the prgrams to:
  - Exchange data between two processe running as `shmop.c`.

- Inquire about the state of shared memory with `shmctl.c`.
- Use `semctl.c` to lock a shared memory segment.
- Use `semctl.c` to delete a shared memory segment.

Exercise 12773

Write 2 programs that will communicate via mapped memory.

Dave Marshall
1/5/1999

Subsections

IPC:Sockets

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see `<sys/socket.h>`), of which only the UNIX and Internet domains are normally used Solaris 2.x Sockets can be used to communicate between processes on a single system, like other forms of IPC.

The UNIX domain provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain.

Internet domain communication uses the TCP/IP internet protocol suite.

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type. There are five types of socket.

A stream socket

: -- provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries. A stream operates much like a telephone conversation. The socket type is `SOCK_STREAM`, which, in the Internet domain, uses Transmission Control Protocol (TCP).

A datagram socket

: -- supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent. Record boundaries in the data are preserved. Datagram sockets operate much like passing letters back and forth in the mail. The socket type is `SOCK_DGRAM`, which, in the Internet domain, uses User Datagram Protocol (UDP).

A sequential packet socket

: -- provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length. The socket type is `SOCK_SEQPACKET`. No protocol for this type has been implemented for any protocol family.

A raw socket

: provides access to the underlying communication protocols.

These sockets are usually datagram oriented, but their exact characteristics depend on the interface provided by the protocol.

Socket Creation and Naming

`int socket(int domain, int type, int protocol)` is called to create a socket in the specified domain and of the specified type. If a `protocol` is not specified, the system defaults to a protocol that supports the specified socket type. The socket handle (a descriptor) is returned. A remote process has no way to identify a socket until an address is bound to it. Communicating processes connect through addresses. In the UNIX domain, a connection is usually composed of one or two path names. In the Internet domain, a connection is composed of local and remote addresses and local and remote ports. In most domains, connections must be unique.

`int bind(int s, const struct sockaddr *name, int namelen)` is called to bind a path or internet address to a socket. There are three different ways to call `bind()`, depending on the domain of the socket.

- For UNIX domain sockets with paths containing 14, or fewer characters, you can:

  ```
  #include <sys/socket.h>
   ...
  bind (sd, (struct sockaddr *) &addr, length);
  ```
- If the path of a UNIX domain socket requires more characters, use:

  ```
  #include <sys/un.h>
  ...
  bind (sd, (struct sockaddr_un *) &addr, length);
  ```
- For Internet domain sockets, use

  ```
  #include <netinet/in.h>
  ...
  bind (sd, (struct sockaddr_in *) &addr, length);
  ```

In the UNIX domain, binding a name creates a named socket in the file system. Use `unlink()` or `rm ()` to remove the socket.

Connecting Stream Sockets

Connecting sockets is usually not symmetric. One process usually acts as a server and the other process is the client. The server binds its socket to a previously agreed path or address. It then blocks on the socket. For a `SOCK_STREAM` socket, the server calls `int listen(int s, int backlog)` , which specifies how many connection requests can be queued. A client initiates a connection to the server's socket by a call to `int connect(int s, struct sockaddr *name, int namelen)` . A UNIX domain call is like this:

```
struct sockaddr_un server;
...
connect (sd, (struct sockaddr_un *)&server, length);
```

while an Internet domain call would be:

```
struct sockaddr_in;
...
connect (sd, (struct sockaddr_in *)&server, length);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. For a `SOCK_STREAM` socket, the server calls accept(3N) to complete the connection.

`int accept(int s, struct sockaddr *addr, int *addrlen)` returns a new socket descriptor which is valid only for the particular connection. A server can have multiple `SOCK_STREAM` connections active at one time.

Stream Data Transfer and Closing

Several functions to send and receive data from a `SOCK_STREAM` socket. These are `write(), read(), int send(int s, const char *msg, int len, int flags),` and `int recv(int s, char *buf, int len, int flags)`. `send()` and `recv()` are very similar to `read()` and `write()`, but have some additional operational `flags`.

The flags parameter is formed from the bitwise OR of zero or more of the following:

`MSG_OOB`
: -- Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support "out-of-band" data. Only `SOCK_STREAM` sockets created in the `AF_INET` address family support out-of-band data.

`MSG_DONTROUTE`
: -- The `SO_DONTROUTE` option is turned on for the duration of the operation. It is used only by diagnostic or routing pro- grams.

`MSG_PEEK`
: -- "Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

A `SOCK_STREAM` socket is discarded by calling `close()`.

Datagram sockets

A datagram socket does not require that a connection be established. Each message carries the destination address. If a particular local address is needed, a call to `bind()` must precede any data transfer. Data is sent through calls to `sendto()` or `sendmsg()`. The `sendto()` call is like a `send()` call with the destination address also specified. To receive datagram socket messages, call `recvfrom()` or `recvmsg()`. While `recv()` requires one buffer for the arriving data, `recvfrom()` requires two buffers, one for the incoming message and another to receive the source address.

Datagram sockets can also use `connect()` to connect the socket to a specified destination socket. When this is done, `send()` and `recv()` are used to send and receive data.

`accept()` and `listen()` are not used with datagram sockets.

Socket Options

Sockets have a number of options that can be fetched with `getsockopt()` and set with `setsockopt()`. These functions can be used at the native socket level (`level = SOL_SOCKET`), in which case the socket option name must be specified. To manipulate options at any other level the protocol number of the desired protocol controlling the option of interest must be specified (see `getprotoent()` in `getprotobyname()`).

Example Socket Programs:`socket_server.c`,`socket_client`

These two programs show how you can establish a socket connection using the above functions.

`socket_server.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```
#include <stdio.h>

#define NSTRS       3           /* no. of strings  */
#define ADDRESS     "mysocket"  /* addr to connect */

/*
 * Strings we send to the client.
 */
char *strs[NSTRS] = {
    "This is the first string from the server.\n",
    "This is the second string from the server.\n",
    "This is the third string from the server.\n"
};

main()
{
    char c;
    FILE *fp;
    int fromlen;
    register int i, s, ns, len;
    struct sockaddr_un saun, fsaun;

    /*
     * Get a socket to work with.  This socket will
     * be in the UNIX domain, and will be a
     * stream socket.
     */
    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("server: socket");
        exit(1);
    }

    /*
     * Create the address we will be binding to.
     */
    saun.sun_family = AF_UNIX;
    strcpy(saun.sun_path, ADDRESS);

    /*
     * Try to bind the address to the socket.  We
     * unlink the name first so that the bind won't
     * fail.
     *
     * The third argument indicates the "length" of
     * the structure, not just the length of the
     * socket name.
     */
    unlink(ADDRESS);
    len = sizeof(saun.sun_family) + strlen(saun.sun_path);

    if (bind(s, &saun, len) < 0) {
        perror("server: bind");
        exit(1);
    }

    /*
     * Listen on the socket.
     */
    if (listen(s, 5) < 0) {
        perror("server: listen");
```

```
        exit(1);
    }

    /*
     * Accept connections.  When we accept one, ns
     * will be connected to the client.  fsaun will
     * contain the address of the client.
     */
    if ((ns = accept(s, &fsaun, &fromlen)) < 0) {
        perror("server: accept");
        exit(1);
    }

    /*
     * We'll use stdio for reading the socket.
     */
    fp = fdopen(ns, "r");

    /*
     * First we send some strings to the client.
     */
    for (i = 0; i < NSTRS; i++)
        send(ns, strs[i], strlen(strs[i]), 0);

    /*
     * Then we read some strings from the client and
     * print them out.
     */
    for (i = 0; i < NSTRS; i++) {
        while ((c = fgetc(fp)) != EOF) {
            putchar(c);

            if (c == '\n')
                break;
        }
    }

    /*
     * We can simply use close() to terminate the
     * connection, since we're done with both sides.
     */
    close(s);

    exit(0);
}
```

socket_client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NSTRS       3           /* no. of strings  */
#define ADDRESS     "mysocket"  /* addr to connect */

/*
 * Strings we send to the server.
```

```
 */
char *strs[NSTRS] = {
    "This is the first string from the client.\n",
    "This is the second string from the client.\n",
    "This is the third string from the client.\n"
};

main()
{
    char c;
    FILE *fp;
    register int i, s, len;
    struct sockaddr_un saun;

    /*
     * Get a socket to work with.  This socket will
     * be in the UNIX domain, and will be a
     * stream socket.
     */
    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("client: socket");
        exit(1);
    }

    /*
     * Create the address we will be connecting to.
     */
    saun.sun_family = AF_UNIX;
    strcpy(saun.sun_path, ADDRESS);

    /*
     * Try to connect to the address.  For this to
     * succeed, the server must already have bound
     * this address, and must have issued a listen()
     * request.
     *
     * The third argument indicates the "length" of
     * the structure, not just the length of the
     * socket name.
     */
    len = sizeof(saun.sun_family) + strlen(saun.sun_path);

    if (connect(s, &saun, len) < 0) {
        perror("client: connect");
        exit(1);
    }

    /*
     * We'll use stdio for reading
     * the socket.
     */
    fp = fdopen(s, "r");

    /*
     * First we read some strings from the server
     * and print them out.
     */
    for (i = 0; i < NSTRS; i++) {
        while ((c = fgetc(fp)) != EOF) {
            putchar(c);
```

```
            if (c == '\n')
                break;
        }
    }

    /*
     * Now we send some strings to the server.
     */
    for (i = 0; i < NSTRS; i++)
        send(s, strs[i], strlen(strs[i]), 0);

    /*
     * We can simply use close() to terminate the
     * connection, since we're done with both sides.
     */
    close(s);

    exit(0);
}
```

Exercises

Exercise 12776

Configure the above `socket_server.c` and `socket_client.c` programs for you system and compile and run them. You will need to set up socket `ADDRESS` definition.

Dave Marshall
1/5/1999

Subsections

Threads: Basic Theory and Libraries

This chapter examines aspects of threads and multiprocessing (and multithreading). We will firts study a little theory of threads and also look at how threading can be effectively used to make programs more efficient. The C thread libraries will then be introduced. The following chapters will look at further thead issues sucj a synchronisation and practical examples.

Processes and Threads

We can think of a **thread** as basically a *lightweight* process. In order to understand this let us consider the two main characteristics of a process:

Unit of resource ownership

-- A process is allocated:

- a virtual address space to hold the process image
- control of some resources (files, I/O devices...)

Unit of dispatching

- A process is an execution path through one or more programs:

- execution may be interleaved with other processes
- the process has an execution state and a dispatching priority

If we treat these two characteristics as being independent (as does modern OS theory):

- The unit of resource ownership is usually referred to as a **process** or task. This Processes have:
  - a virtual address space which holds the process image.
  - protected access to processors, other processes, files, and I/O resources.
- The unit of dispatching is usually referred to a **thread** or a lightweight process. Thus a thread:
  - Has an execution state (running, ready, etc.)
  - Saves thread context when not running
  - Has an execution stack and some per-thread static storage for local variables
  - Has access to the memory address space and resources of its process
- all threads of a process share this when one thread alters a (non-private) memory item, all other threads (of the process) sees that a file open with one thread, is available to others

Benefits of Threads vs Processes

If implemented correctly then threads have some advantages of (multi) processes, They take:

- Less time to create a new thread than a process, because the newly created thread uses the current process address space.
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
- Less communication overheads -- communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

Multithreading vs. Single threading

Just a we can multiple processes running on some systems we can have multiple threads running:

Single threading

-- when the OS does not recognize the concept of thread

Multithreading

-- when the OS supports multiple threads of execution within a single process

Figure 28.1 shows a variety of models for threads and processes.

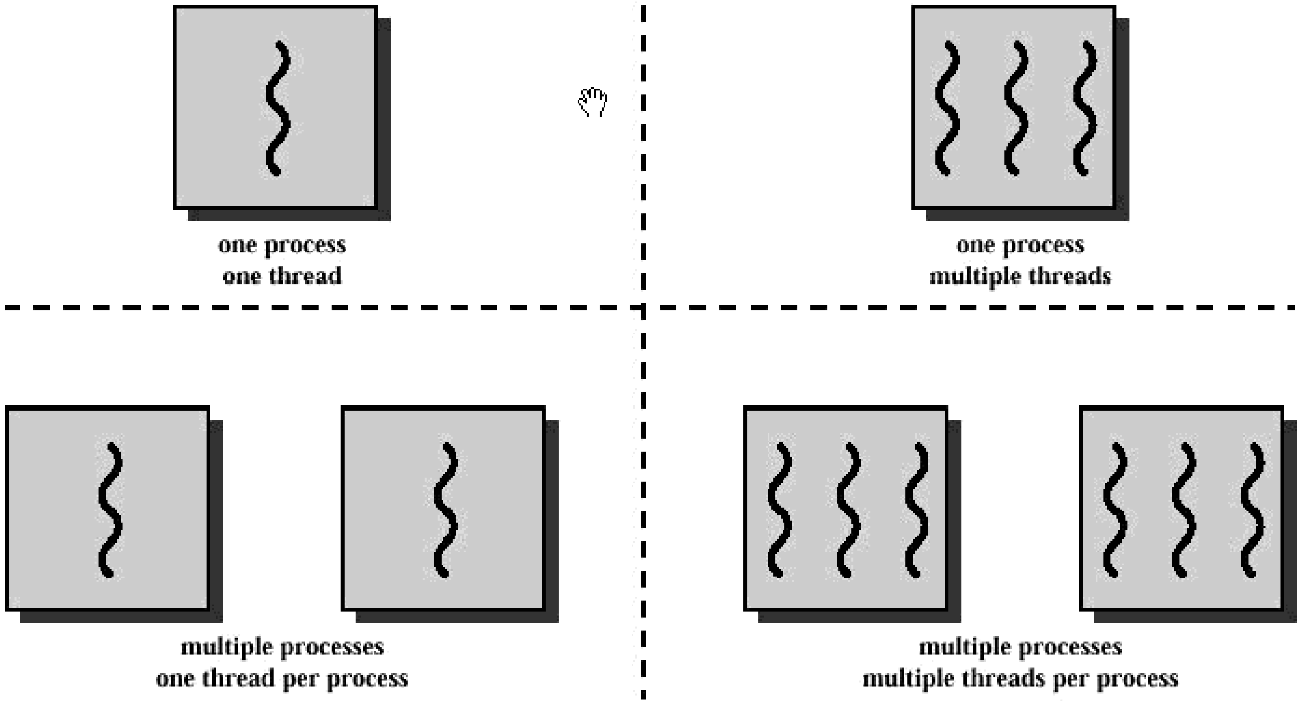


Fig. 28.1 Threads and Processes Some example popular OSs and their thread support is:

MS-DOS

-- support a single user process and a single thread

UNIX

-- supports multiple user processes but only supports one thread per process

Solaris

-- supports multiple threads

Multithreading your code can have many benefits:

- Improve application responsiveness -- Any program in which many activities are not dependent upon each other can be redesigned so that each activity is defined as a thread. For example, the user of a multithreaded GUI does not have to wait for one activity to complete before starting another.
- Use multiprocessors more efficiently -- Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors. Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.
- Improve program structure -- Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than single threaded programs.
- Use fewer system resources -- Programs that use two or more processes that access common data through shared memory are applying more than one thread of control. However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state information makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes, or to synchronize their actions.

Figure 28.2 illustrates different process models and thread control in a single thread and multithreaded application.

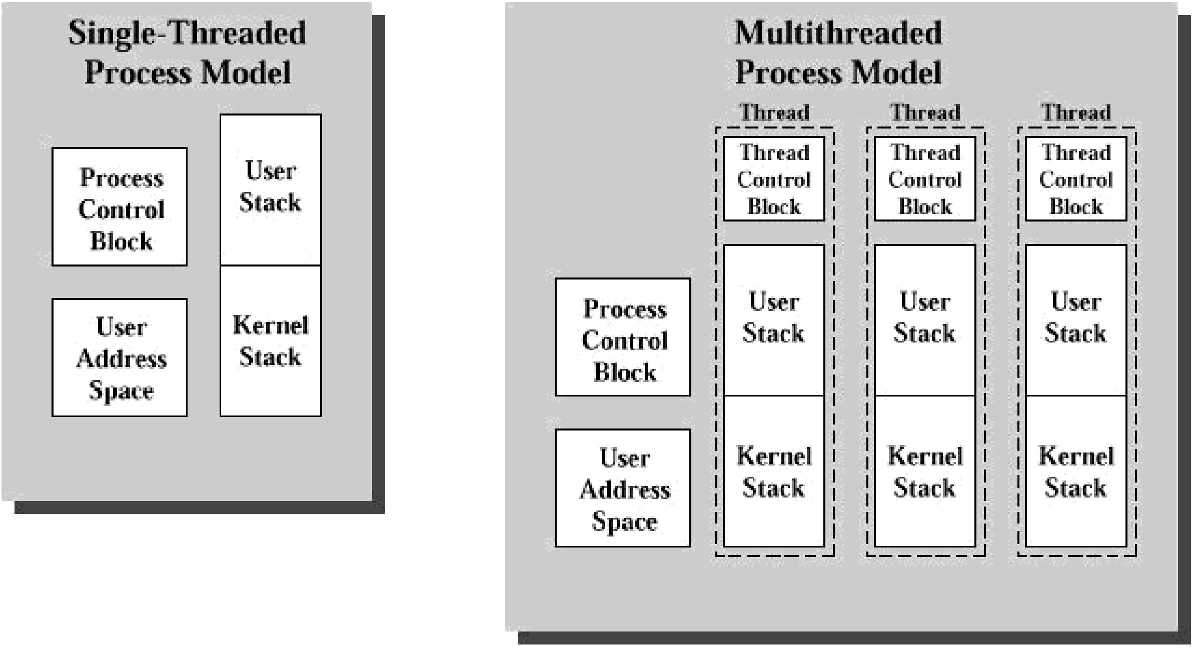


Fig. 28.2 Single and Multi- Thread Applicatiions

Some Example applications of threads

:

Example : A file server on a LAN

- It needs to handle several file requests over a short period
- Hence more efficient to create (and destroy) a single thread for each request
- Multiple threads can possibly be executing simultaneously on different processors

Example 2: Matrix Multiplication

Matrix Multilication essentially involves taking the rows of one matrix and multiplying and adding corresponding columns in a second matrix *i.e*:

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Fig. 28.3 Matrix Multiplication (3x3 example) Note that each *element* of the resultant matrix can be computed independently, that is to say by a different thread.

We will develop a C++ example program for matrix multiplication later (see Chapter).

Thread Levels

There are two broad categories of thread implementation:

- User-Level Threads -- Thread Libraries.
- Kernel-level Threads -- System Calls.

There are merits to both, in fact some OSs allow access to both levels (*e.g.* Solaris).

User-Level Threads (ULT)

In this level, the kernel is not aware of the existence of threads -- All thread management is done by the application by using a thread library. Thread switching does not require kernel mode privileges (no mode switch) and scheduling is application specific

Kernel activity for ULTs:

- The kernel is not aware of thread activity but it is still managing process activity
- When a thread makes a system call, the whole process will be blocked but for the thread library that thread is still in the running state
- So thread states are independent of process states

Advantages and inconveniences of ULT

Advantages:

- Thread switching does not involve the kernel -- no mode switching
- Scheduling can be application specific -- choose the best algorithm.
- ULTs can run on any OS -- Only needs a thread library

Disadvantages:

- Most system calls are blocking and the kernel blocks processes -- So all threads within the process will be blocked
- The kernel can only assign processes to processors -- Two threads within the same process cannot run simultaneously on two processors

Kernel-Level Threads (KLT)

In this level, All thread management is done by kernel No thread library but an API (system calls) to the kernel thread facility exists. The kernel maintains context information for the process and the threads, switching between threads requires the kernel Scheduling is performed on a thread basis.

Advantages and inconveniences of KLT

Advantages

- the kernel can simultaneously schedule many threads of the same process on many processors blocking is done on a thread level
- kernel routines can be multithreaded

Disadvantages:

- thread switching within the same process involves the kernel, *e.g* if we have 2 mode switches per thread switch this results in a significant slow down.

Combined ULT/KLT Approaches

Idea is to combine the best of both approaches

Solaris is an example of an OS that combines both ULT and KLT (Figure 28.4:

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- Process includes the user's address space, stack, and process control block
- User-level threads (threads library) invisible to the OS are the interface for application parallelism
- Kernel threads the unit that can be dispatched on a processor
- Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT

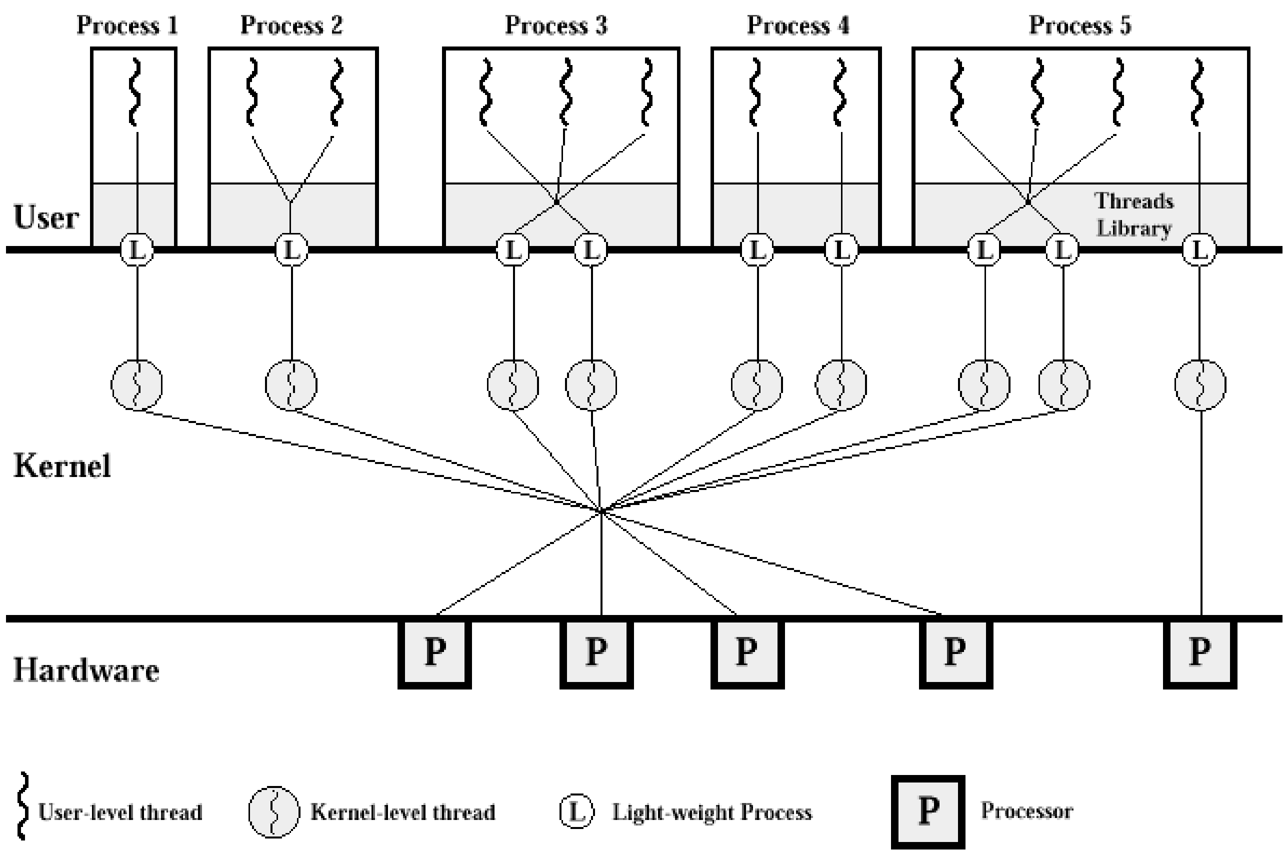


Fig. 28.4 Solaris Thread Implementation

Threads libraries

The interface to multithreading support is through a subroutine library, libpthread for POSIX threads, and libthread for Solaris threads. They both contain code for:

- creating and destroying threads
- passing messages and data between threads
- scheduling thread execution
- saving and restoring thread contexts

The POSIX Threads Library: `libpthread`, <pthread.h>

Creating a (Default) Thread

Use the function `pthread_create()` to add a new thread of control to the current process. It is prototyped by:

```
int pthread_create(pthread\_t *tid, const pthread\_attr\_t *tattr,
void*(*start_routine)(void *), void *arg);
```

When an attribute object is not specified, it is NULL, and the *default* thread is created with the following attributes:

- It is unbounded
- It is nondetached
- It has a a default stack and stack size
- It inhetits the parent's priority

You can also create a default attribute object with `pthread_attr_init()` function, and then use this attribute object to create a default thread. See the Section 29.2.

An example call of default thread creation is:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;
/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The `pthread_create()` function is called with `attr` having the necessary state behavior. `start_routine` is the function with which the new thread begins execution. When `start_routine` returns, the thread exits with the exit status set to the value returned by `start_routine`.

When `pthread_create` is successful, the ID of the thread created is stored in the location referred to as `tid`.

Creating a thread using a NULL attribute argument has the same effect as using a default attribute; both create a default thread. When tattr is initialized, it acquires the default behavior.

`pthread_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred.

Wait for Thread Termination

Use the `pthread_join` function to wait for a thread to terminate. It is prototyped by:

```
int pthread_join(thread_t tid, void **status);
```

An example use of this function is:

```
#include <pthread.h>
pthread_t tid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);
/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

The `pthread_join()` function blocks the calling thread until the specified thread terminates. The specified thread must be in the current process and must not be detached. When `status` is not `NULL`, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully. Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of `ESRCH`. After `pthread_join()` returns, any stack storage associated with the thread can be reclaimed by the application.

The `pthread_join()` routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument. If you are interested in the exit code of the defunct thread, supply the address of an area to receive it. Remember that `pthread_join()` works only for target threads that are nondetached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached. Think of a detached thread as being the thread you use in most instances and reserve nondetached threads for only those situations that require them.

A Simple Threads Example

In this Simple Threads fragment below, one thread executes the procedure at the top, creating a helper thread that executes the procedure fetch, which involves a complicated database lookup and might take some time.

The main thread wants the results of the lookup but has other work to do in the meantime. So it does those other things and then waits for its helper to complete its job by executing `pthread_join()`. An argument, `pbe`, to the new thread is passed as a stack parameter. This can be done here because the main thread waits for the spun-off thread to terminate. In general, though, it is better to `malloc()` storage from the heap instead of passing an address to thread stack storage, which can disappear or be reassigned if the thread terminated.

The source for `thread.c` is as follows:

```
void mainline (...)
{
struct phonebookentry *pbe;
pthread_attr_t tattr;
pthread_t helper;
int status;
pthread_create(&helper, NULL, fetch, &pbe);
/* do something else for a while */
pthread_join(helper, &status);
/* it's now safe to use result */
}
void fetch(struct phonebookentry *arg)
{
struct phonebookentry *npbe;
/* fetch value from a database */
npbe = search (prog_name)
if (npbe != NULL)
*arg = *npbe;
pthread_exit(0);
}
struct phonebookentry {
char name[64];
char phonenumber[32];
char flags[16];
}
```

Detaching a Thread

The function `pthread_detach()` is an alternative to `pthread_join()` to reclaim storage for a thread that is created with a detachstate attribute set to `PTHREAD_CREATE_JOINABLE`. It is prototyped by:

```
int pthread\_detach(thread\_t tid);
```

A simple example of calling this fucntion to detatch a thread is given by:

```
#include <pthread.h>
pthread_t tid;
int ret;
/* detach thread tid */
ret = pthread_detach(tid);
```

The `pthread_detach()` function is used to indicate to the implementation that storage for the thread `tid` can be reclaimed when the thread terminates. If tid has not terminated, `pthread_detach()` does not cause it to terminate. The effect of multiple `pthread_detach()` calls on the same target thread is unspecified.

`pthread_detach()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_detach()` fails and returns the an error value.

Create a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data: local data and global data. For multithreaded C programs a third class is added:*thread-specific data (TSD)*. This is very much like global data, except that it is private to a thread.

Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a key that is global to all threads in the process. Using the key, a thread can access a pointer `(void *)` that is maintained per-thread.

The function `pthread_keycreate()` is used to allocate a key that is used to identify thread-specific data in a process. The key is global to all threads in the process, and all threads initially have the value `NULL` associated with the key when it is created.

`pthread_keycreate()` is called once for each key before the key is used. There is no implicit synchronization. Once a key has been created, each thread can bind a value to the key. The values are specific to the thread and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a destructor function. `pthread_keycreate()` is prototyped by:

```
int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
int ret;
/* key create without destructor */
ret = pthread_key_create(&key, NULL);
/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

When `pthread_keycreate()` returns successfully, the allocated `key` is stored in the location pointed to by `key`. The caller must ensure that the storage and access to this key are properly synchronized. An optional destructor function, destructor, can be used to free stale storage. When a key has a non-NULL destructor function and the thread has a non-`NULL` value associated with that key, the destructor function is called with the current associated value when the thread exits. The order in which the destructor functions are called is unspecified.

`pthread_keycreate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_keycreate()` fails and returns an error value.

Delete the Thread-Specific Data Key

The function `pthread_keydelete()` is used to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated and will return an error if ever referenced. (There is no comparable function in Solaris threads.)

`pthread_keydelete()` is prototyped by:

```
int pthread_key_delete(pthread_key_t key);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
int ret;
/* key previously created */
ret = pthread_key_delete(key);
```

Once a `key` has been deleted, any reference to it with the `pthread_setspecific()` or `pthread_getspecific()` call results in the `EINVAL` error.

It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

`pthread_keydelete()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_keycreate()` fails and returns the corresponding value.

Set the Thread-Specific Data Key

The function `pthread_setspecific()` is used to set the thread-specific binding to the specified thread-specific data key. It is prototyped by :

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
void *value;
int ret;
```

```
/* key previously created */
ret = pthread_setspecific(key, value);
```

`pthread_setspecific()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_setspecific()` fails and returns an error value.

Note: `pthread_setspecific()` does *not* free its storage. If a new binding is set, the existing binding must be freed; otherwise, a *memory leak can occur*.

Get the Thread-Specific Data Key

Use `pthread_getspecific()` to get the calling thread's binding for key, and store it in the location pointed to by value. This function is prototyped by:

```
int pthread_getspecific(pthread_key_t key);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
void *value;
/* key previously created */
value = pthread_getspecific(key);
```

Global and Private Thread-Specific Data Example

Thread-Specific Data Global but Private

Consider the following code:

```
body() {
...
while (write(fd, buffer, size) == -1) {
if (errno != EINTR) {
fprintf(mywindow, "%s\n", strerror(errno));
exit(1);
}
}
...
}
```

This code may be executed by any number of threads, but it has references to two global variables, errno and mywindow, that really should be references to items private to each thread.

References to errno should get the system error code from the routine called by this thread, not by some other thread. So, references to errno by one thread refer to a different storage location than references to errno by other threads. The mywindow variable is intended to refer to a stdio stream connected to a window that is private to the referring thread. So, as with `errno`, references to mywindow by one thread should refer to a different storage location (and, ultimately, a different window) than references to mywindow by other threads. The only difference here is that the threads library takes care of errno, but the programmer must somehow make this work for mywindow. The next example shows how the references to mywindow work. The preprocessor converts references to mywindow into invocations of the `mywindow` procedure. This routine in turn invokes `pthread_getspecific()`, passing it the `mywindow_key` global variable (it really is a global variable) and an output parameter, `win`, that receives the identity of this thread's window.

Turning Global References Into Private References Now consider this code fragment:

```
thread_key_t mywin_key;
FILE *_mywindow(void) {
FILE *win;
pthread_getspecific(mywin_key, &win);
return(win);
}
#define mywindow _mywindow()
```

```
void routine_uses_win( FILE *win) {
...
}
void thread_start(...) {
...
make_mywin();
...
routine_uses_win( mywindow )
...
}
```

The `mywin_key` variable identifies a class of variables for which each thread has its own private copy; that is, these variables are thread-specific data. Each thread calls `make_mywin` to initialize its window and to arrange for its instance of mywindow to refer to it. Once this routine is called, the thread can safely refer to `mywindow` and, after `mywindow`, the thread gets the reference to its private window. So, references to mywindow behave as if they were direct references to data private to the thread.

We can now set up our initial Thread-Specific Data:

```
void make_mywindow(void) {
FILE **win;
static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;
pthread_once(&mykeycreated, mykeycreate);
win = malloc(sizeof(*win));
create_window(win, ...);
pthread_setspecific(mywindow_key, win);
}
void mykeycreate(void) {
pthread_keycreate(&mywindow_key, free_key);
}
void free_key(void *win) {
free(win);
}
```

First, get a unique value for the key, `mywin_key`. This key is used to identify the thread-specific class of data. So, the first thread to call `make_mywin` eventually calls `pthread_keycreate()`, which assigns to its first argument a unique key. The second argument is a destructor function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, a call is made to the `create_window` routine, which sets up a window for the thread and sets the storage pointed to by win to refer to it. Finally, a call is made to `pthread_setspecific()`, which associates the value contained in win (that is, the location of the storage containing the reference to the window) with the key. After this, whenever this thread calls `pthread_getspecific()`, passing the global key, it gets the value that was associated with this key by this thread when it called `pthread_setspecific()`. When a thread terminates, calls are made to the destructor functions that were set up in `pthread_key_create()`. Each destructor function is called only if the terminating thread established a value for the key by calling `pthread_setspecific()`.

Getting the Thread Identifiers

The function `pthread_self()` can be called to return the ID of the calling thread. It is prototyped by:

```
pthread_t pthread_self(void);
```

It is use is very straightforward:

```
#include <pthread.h>
pthread_t tid;
tid = pthread_self();
```

Comparing Thread IDs

The function `pthread_equal()` can be called to compare the thread identification numbers of two threads. It is prototyped by:

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

It is use is straightforward to use, also:

```
#include <pthread.h>
pthread_t tid1, tid2;
int ret;
ret = pthread_equal(tid1, tid2);
```

As with other comparison functions, `pthread_equal()` returns a non-zero value when `tid1` and `tid2` are equal; otherwise, zero is returned. When either `tid1` or `tid2` is an invalid thread identification number, the result is unpredictable.

Initializing Threads

Use `pthread_once()` to call an initialization routine the first time `pthread_once()` is called -- Subsequent calls to have no effect. The prototype of this function is:

```
int pthread_once(pthread_once_t *once_control,
void (*init_routine)(void));
```

Yield Thread Execution

The function `sched_yield()` to cause the current thread to yield its execution in favor of another thread with the same or greater priority. It is prototyped by:

```
int sched_yield(void);
```

It is clearly a simple function to call:

```
#include <sched.h>
int ret;
ret = sched_yield();
```

`sched_yield()` returns zero after completing successfully. Otherwise -1 is returned and errno is set to indicate the error condition.

Set the Thread Priority

Use `pthread_setschedparam()` to modify the priority of an existing thread. This function has no effect on scheduling policy. It is prototyped as follows:

```
int pthread_setschedparam(pthread_t tid, int policy,
const struct sched_param *param);
```

and used as follows:

```
#include <pthread.h>
pthread_t tid;
int ret;
struct sched_param param;
int priority;
/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;
/* only supported policy, others will result in ENOTSUP */

policy = SCHED_OTHER;
/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid, policy, &param);
```

`pthread_setschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the `pthread_setschedparam()` function fails and returns an error value.

Get the Thread Priority

`int pthread_getschedparam(pthread_t tid, int policy, struct schedparam *param)` gets the priority of the existing thread.

An example call of this function is:

```
#include <pthread.h>
pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;
/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);
/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

`pthread_getschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the error value set.

Send a Signal to a Thread

Signal may be sent to threads is a similar fashion to those for process as follows:

```
#include <pthread.h>
#include <signal.h>
int sig;
pthread_t tid;
int ret;
ret = pthread_kill(tid, sig);
```

`pthread_kill()` sends the signal `sig` to the thread specified by `tid`. `tid` must be a thread within the same process as the calling thread. The `sig` argument must be a valid signal of the same type defined for `signal()` in < `signal.h`> (See Chapter 23)

When `sig` is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of tid.

This function returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, `pthread_kill()` fails and returns an error value.

Access the Signal Mask of the Calling Thread

The function `pthread_sigmask()` may be used to change or examine the signal mask of the calling thread. It is prototyped as follows:

```
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
```

Example uses of this function include:

```
#include <pthread.h>
#include <signal.h>
int ret;
sigset_t old, new;
ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how determines how the signal set is changed. It can have one of the following values:

SIG_SETMASK
: -- Replace the current signal mask with new, where new indicates the new signal mask.

SIG_BLOCK
: -- Add new to the current signal mask, where new indicates the set of signals to block.

SIG_UNBLOCK

-- Delete new from the current signal mask, where new indicates the set of signals to unblock.

When the value of `new` is `NULL`, the value of how is not significant and the signal mask of the thread is unchanged. So, to inquire about currently blocked signals, assign a `NULL` value to the new argument. The old variable points to the space where the previous signal mask is stored, unless it is `NULL`.

`pthread_sigmask()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_sigmask()` fails and returns an errro value.

Terminate a Thread

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine; see `pthread_create()`
- By calling `pthread_exit()`, supplying an exit status
- By termination with POSIX cancel functions; see `pthread_cancel()`

The `void pthread_exit(void *status)` is used terminate a thread in a similar fashion the `exit()` for a process:

```
#include <pthread.h>
int status;
pthread_exit(&status); /* exit with status */
```

The `pthread_exit()` function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by status are retained until the thread is waited for (blocked). Otherwise, status is ignored and the thread's ID can be reclaimed immediately.

The `pthread_cancel()` function to cancel a thread is prototyped:

```
int pthread_cancel(pthread_t thread);
```

and called:

```
#include <pthread.h>
pthread_t thread;
int ret;
ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions,

`pthread_setcancelstate()` and `pthread_setcanceltype()` (see `man` pages for further information on these functions), determine that state.

`pthread_cancel()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns an error value.

Solaris Threads: `<thread.h>`

Solaris have many similarities to POSIX threads,In this sectionfocus on the Solaris features that are not found in POSIX threads. Where functionality is virtually the same for both Solaris threads and for pthreads, (even though the function names or arguments might differ), only a brief example consisting of the correct include file and the function prototype is presented. Where return values are not given for the Solaris threads functions, see the appropriate `man` pages.

The Solaris threads API and the pthreads API are two solutions to the same problem: building parallelism into application software. Although each API is complete in itself, you can safely mix Solaris threads functions and pthread functions in the same program.

The two APIs do not match exactly, however. Solaris threads supports functions that are not found in pthreads, and pthreads includes functions that are not supported in the Solaris interface. For those functions that do match, the associated arguments might not, although the information content is effectively the same.

By combining the two APIs, you can use features not found in one to enhance the other. Similarly, you can run applications using Solaris threads, exclusively, with applications using pthreads, exclusively, on the same system.

To use the Solaris threads functions described in this chapter, you must link with the Solaris threads library `-lthread` and include the <`thread.h`> in all programs.

Unique Solaris Threads Functions

Let us begin by looking at some functions that are unique to Solaris threads:

- Suspend Thread Execution
- Continue a Suspended Thread
- Set Thread Concurrency Level
- Get Thread Concurrency

Suspend Thread Execution

The function `thr_suspend()` immediately suspends the execution of the thread specified by a target thread, (`tid` below). It is prototyped by:

```
int thr_suspend(thread_t tid);
```

On successful return from `thr_suspend()`, the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to `thr_suspend()` have no effect. Signals cannot awaken the suspended thread; they remain pending until the thread resumes execution.

A simple example call is as follows:

```
#include <thread.h>

thread_t tid; /* tid from thr_create() */
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;
int ret;
ret = thr_suspend(tid);
/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

Note: `pthread_t tid` as defined in pthreads is the same as `thread_t` tid in Solaris threads. `tid` values can be used interchangeably either by assignment or through the use of casts.

Continue a Suspended Thread

The function `thr_continue()` resumes the execution of a suspended thread. It is prototypes as follows:

```
int thr_continue(thread_t tid);
```

Once a suspended thread is continued, subsequent calls to `thr_continue()` have no effect.

A suspended thread will *not* be awakened by a signal. The signal stays pending until the execution of the thread is resumed by `thr_continue()`.

`thr_continue()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_continue()` The following code fragment illustrates the use of the function:

```
thread_t tid; /* tid from thr_create()*/
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;
int ret;
ret = thr_continue(tid);
/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

Set Thread Concurrency Level

By default, Solaris threads attempt to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress. When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through `thr_setconcurrency(int new_level)`. To get the number of threads being used, use the function `thr_getconcurrencyint(void)`:

`thr_setconcurrency()` provides a hint to the system about the required level of concurrency in the application. The system ensures that a sufficient number of threads are active so that the process continues to make progress, for example:

```
#include <thread.h>
int new_level;
int ret;

ret = thr_setconcurrency(new_level);
```

Unbound threads in a process might or might not be required to be simultaneously active. To conserve system resources, the threads system ensures by default that enough threads are active for the process to make progress, and that the process will not deadlock through a lack of concurrency. Because this might not produce the most effective level of concurrency, `thr_setconcurrency()` permits the application to give the threads system a hint, specified by `new_level`, for the desired level of concurrency. The actual number of simultaneously active threads can be larger or smaller than `new_level`. Note that an application with multiple compute-bound threads can fail to schedule all the runnable threads if `thr_setconcurrency()` has not been called to adjust the level of execution resources. You can also affect the value for the desired concurrency level by setting the `THR_NEW_LW` flag in `thr_create()`. This effectively increments the current level by one.

`thr_setconcurrency()` a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_setconcurrency()` fails and returns the corresponding value to errno.

Readers/Writer Locks

Readers/Writer locks are another unique feature of Solaris threads. They allow simultaneous read access by many threads while restricting write access to only one thread at a time.

When any thread holds the lock for reading, other threads can also acquire the lock for reading but must wait to acquire the lock for writing. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing. Readers/writer locks are slower than mutexes, but can improve performance when they protect data that are not frequently written but that are read by many concurrent threads. Use readers/writer locks to synchronize threads in this process and other processes by allocating them in memory that is writable and shared among the cooperating processes (see mmap(2)) and by initializing them for this behavior. By default, the acquisition order is not defined when multiple threads are waiting for a readers/writer lock. However, to avoid writer starvation, the Solaris threads package tends to favor writers over readers. Readers/writer locks must be initialized before use.

Initialize a Readers/Writer Lock

The function `rwlock_init()` initialises the readers/writer lock. it is prototypes in <`synch.h`> or <`thread.h`> as follows:

```
int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

The readers/writer lock pointed to by `rwlp` and to set the lock state to unlocked. `type` can be one of the following

`USYNC_PROCESS`

-- The readers/writer lock can be used to synchronize threads in this process and other processes.

`USYNC_THREAD`

-- The readers/writer lock can be used to synchronize threads in this process, only.

Note: that `arg` is currently ignored.

`rwlock_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value to `errno`.

Multiple threads must not initialize the same readers/writer lock simultaneously. Readers/writer locks can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. A readers/writer lock must not be reinitialized while other threads might be using it.

An example code fragment that initialises Readers/Writer Locks with Intraprocess Scope is as follows:

```
#include <thread.h>

rwlock_t rwlp;
int ret;
/* to be used within this process only */
ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
```

```
Initializing Readers/Writer Locks with Interprocess Scope
#include <thread.h>
rwlock_t rwlp;
int ret;
/* to be used among all processes */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

Acquire a Read Lock

To acquire a read lock on the readers/writer lock use the `rw_rdlock()` function:

```
int rw_rdlock(rwlock_t *rwlp);
```

The readers/writer lock pointed to by `rwlp`. When the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. Otherwise, the read lock is acquired.

`rw_rdlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value to `errno`.

A function `rw_tryrdlock(rwlock_t *rwlp)` may also be used to attempt to acquire a read lock on the readers/writer lock pointed to by rwlp. When the readers/writer lock is already locked for writing, it returns an error. Otherwise, the read lock is acquired. This function returns zero after completing successfully. Any other returned value indicates that an error occurred.

Acquire a Write Lock

The function `rw_wrlock(rwlock_t *rwlp)` acquires a write lock on the readers/writer lock pointed to by `rwlp`. When the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread at a time can hold a write lock on a readers/writer lock.

`rw_wrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Use `rw_trywrlockrwlock_t *rwlp)` to attempt to acquire a write lock on the readers/writer lock pointed to by rwlp. When the readers/writer lock is already locked for reading or writing, it returns an error.

`rw_trywrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Unlock a Readers/Writer Lock

The function `rw_unlock(rwlock_t *rwlp)` unlocks a readers/writer lock pointed to by `rwlp`. The readers/writer lock must be locked and the calling thread must hold the lock either for reading or writing. When any other threads are waiting for the readers/writer lock to become available, one of them is unblocked.

`rw_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Destroy Readers/Writer Lock State

The function `rwlock_destroy(rwlock_t *rwlp)` destroys any state associated with the readers/writer lock pointed to by `rlwp`. The space for storing the readers/writer lock is not freed.

`rwlock_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Readers/Writer Lock Example

The following example uses a bank account analogy to demonstrate readers/writer locks. While the program could allow multiple threads to have concurrent read-only access to the account balance, only a single writer is allowed. Note that the `get_balance()` function needs the lock to ensure that the addition of the checking and saving balances occurs atomically.

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...
float
get_balance() {
float bal;
rw_rdlock(&account_lock);
bal = checking_balance + saving_balance;
rw_unlock(&account_lock);
return(bal);
```

```
}
void
transfer_checking_to_savings(float amount) {
rw_wrlock(&account_lock);
checking_balance = checking_balance - amount;
saving_balance = saving_balance + amount;
rw_unlock(&account_lock);
}
```

Similar Solaris Threads Functions

Here we simply list the similar thread functions and their prototype definitions, except where the complexity of the function merits further exposition. .

Create a Thread

The `thr_create()` routine is one of the most elaborate of all the Solaris threads library routines.

It is prototyped as follows:

```
int thr_create(void *stack_base, size_t stack_size,
void *(*start_routine) (void *), void *arg, long flags,
thread_t *new_thread);
```

Thjis function adds a new thread of control to the current process. Note that the new thread does not inherit pending signals, but it does inherit priority and signal masks.

`stack_base` contains the address for the stack that the new thread uses. If `stack_base` is NULL then `thr_create()` allocates a stack for the new thread with at least `stac_size` bytes. `stack_size` Contains the size, in number of bytes, for the stack that the new thread uses. If `stack_size` is zero, a default size is used. In most cases, a zero value works best. If `stack_size` is not zero, it must be greater than the value returned by `thr_min_stack(void)` inquiry function.

There is no general need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved.

`start_routine` contains the function with which the new thread begins execution. When `start_routine` returns, the thread exits with the exit status set to the value returned by `start_routine`

`arg` can be anything that is described by void, which is typically any 4-byte value. Anything larger must be passed indirectly by having the argument point to it.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode them as one (such as by putting them in a structure).

`flags` specifies attributes for the created thread. In most cases a zero value works best. The value in flags is constructed from the bitwise inclusive OR of the following:

THR_SUSPENDED

-- Suspends the new thread and does not execute `start_routine` until the thread is started by `thr_continue()`. Use this to operate on the thread (such as changing its priority) before you run it. The termination of a detached thread is ignored.

THR_DETACHED

-- Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate. Note - When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create()`.

THR_BOUND

-- Permanently binds the new thread to an LWP (the new thread is a bound thread).

THR_NEW_LWP

-- Increases the concurrency level for unbound threads by one. The effect is similar to incrementing concurrency by one with `thr_setconcurrency()`, although `THR_NEW_LWP` does not affect the level set through the `thr_setconcurrency()` function. Typically, `THR_NEW_LWP` adds a new LWP to the pool of LWPs running unbound threads.

When you specify both `THR_BOUND` and `THR_NEW_LWP`, two LWPs are typically created -- one for the bound thread and another for the pool of LWPs running unbound threads.

THR_DAEMON

-- Marks the new thread as a daemon. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

A process can exit either by calling `exit()` or by having every thread in the process that was not created with the `THR_DAEMON` flag call `thr_exit()`. An application, or a library it calls, can create one or more threads that should be ignored (not counted) in the decision of whether to exit. The `THR_DAEMON`l flag identifies threads that are not counted in the process exit criterion.

`new_thread` points to a location (when `new_thread` is not `NULL`) where the ID of the new thread is stored when `thr_create()` is successful. The caller is responsible for supplying the storage this argument points to. The ID is valid only within the calling process. If you are not interested in this identifier, supply a zero value to `new_thread`.

`thr_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_create()` fails and returns the corresponding value to `errno`.

Get the Thread Identifier

The `int thr_self(void)` to get the ID of the calling thread.

Yield Thread Execution

`void thr_yield(void)` causes the current thread to yield its execution in favor of another thread with the same or greater priority; otherwise it has no effect. There is no guarantee that a thread calling `thr_yield()` will do so.

Signals and Solaris Threads

The following functions exist and operate as do pthreads.

`int thr_kill(thread_t target_thread, int sig)` sends a signal to a thread.

`int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset)` to change or examine the signal mask of the calling thread.

Terminating a Thread

The `void th_exit(void *status)` to terminates a thread.

The `int thr_join(thread_t tid, thread_t *departedid, void **status)` function to wait for a thread to terminate.

Therefore to join specific threads one would do:

```
#include <thread.h>
thread_t tid;
thread_t departedid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, (void**)&status);
/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);
/* waiting to join thread "tid" without return id and status */
ret = thr_join(tid, NULL, NULL);
```

When the `tid` is `(thread_t) 0`, then `thread_join()` waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes `thread_join()` to return.

To join any threads:

```
#include <thread.h>
thread_t tid;
thread_t departedid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = thr_join(NULL, &departedid, (void **)&status);
```

By indicating NULL as `thread id` in the `thr_join()`, a join will take place when any non detached thread in the process exits. The departedid will indicate the thread ID of exiting thread.

Creating a Thread-Specific Data Key

Except for the function names and arguments, thread specific data is the same for Solaris as it is for POSIX.

`int thr_keycreate(thread_key_t *keyp, void (*destructor) (void *value))` allocates a key that is used to identify thread-specific data in a process.

`int thr_setspecific(thread_key_t key, void *value)` binds value to the thread-specific data key, key, for the calling thread.

`int thr_getspecific(thread_key_t key, void **valuep)` stores the current value bound to key for the calling thread into the location pointed to by valuep.

In Solaris threads, if a thread is to be created with a priority other than that of its parent's, it is created in `SUSPEND` mode. While suspended, the threads priority is modified using the `int thr_setprio(thread_t tid, int newprio)` function call; then it is continued.

An unbound thread is usually scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. Its system priority is usually uniform and is inherited from the creating process.

The function `thr_setprio()` changes the priority of the thread, specified by `tid`, within the current process to the priority specified by newprio.

By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The `tid` will preempt lower priority threads, and will yield to higher priority threads. For example:

```
#include <thread.h>
thread_t tid;
int ret;
int newprio = 20;
/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPEND, &tid);
/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);
/* suspended child thread starts executing with new priority */

ret = thr_continue(tid);
```

Use int thr_getprio(thread_t tid, int *newprio) to get the current priority for the thread. Each thread inherits a priority from its creator. `thr_getprio()` stores the current priority, `tid`, in the location pointed to by `newprio`.

Example Use of Thread Specific Data:Rethinking Global Variables

Historically, most code has been designed for single-threaded programs. This is especially true for most of the library routines called from C programs. The following implicit assumptions were made for single-threaded code:

- When you write into a global variable and then, a moment later, read from it, what you read is exactly what you just wrote.
- This is also true for nonglobal, static storage.
- You do not need synchronization because there is nothing to synchronize with.

The next few examples discuss some of the problems that arise in multithreaded programs because of these assumptions, and how you can deal with them.

Traditional, single-threaded C and UNIX have a convention for handling errors detected in system calls. System calls can return anything as a functional value (for example, write returns the number of bytes that were transferred). However, the value -1 is reserved to indicate that something went wrong. So, when a system call returns -1, you know that it failed.

Consider the following piece of code:

```
extern int errno;

...

if (write(file_desc, buffer, size) == -1)
  { /* the system call failed */
```

```
    fprintf(stderr, "something went wrong, error code = %d\n", errno);
    exit(1);
  }
```

Rather than return the actual error code (which could be confused with normal return values), the error code is placed into the global variable `errno`. When the system call fails, you can look in `errno` to find out what went wrong.

Now consider what happens in a multithreaded environment when two threads fail at about the same time, but with different errors.

- Both expect to find their error codes in `errno`,
- **but** one copy of errno cannot hold both values.a

This global variable approach simply does not work for multithreaded programs. Threads solves this problem through a conceptually new storage class: *thread-specific data*.

This storage is similar to global storage in that it can be accessed from any procedure in which a thread might be running. However, it is private to the thread: when two threads refer to the thread-specific data location of the same name, they are referring to two different areas of storage.

So, when using threads, each reference to `errno` is thread-specific because each thread has a private copy of `errno`. This is achieved in this implementation by making `errno` a macro that expands to a function call.

Compiling a Multithreaded Application

There are many options to consider for header files, define flags, and linking.

Preparing for Compilation

The following items are required to compile and link a multithreaded program.

- A standard C compiler (`cc, gcc` *etc*)
- Include files:
  - <thread.h> and <pthread.h>
  - <errno.h>, <limits.h>, <signal.h>, <unistd.h>
- The Solaris threads library (`libthread`), the POSIX threads library (`libpthread`), and possibly the POSIX realtime library (`libposix4`) for semaphores
- MT-safe libraries (`libc, libm, libw, libintl, libnsl, libsocket, libmalloc, libmapmalloc`, and so on)

The include file <thread.h>, used with the `-lthread` library, compiles code that is upward compatible with earlier releases of the Solaris system. This library contains both interfaces: those with Solaris semantics and those with POSIX semantics. To call `thr_setconcurrency()` with POSIX threads, your program needs to include <thread.h>.

The include file <pthread.h>, used with the `-lpthread` library, compiles code that is conformant with the multithreading interfaces defined by the POSIX 1003.1c standard. For complete POSIX compliance, the define flag `_POSIX_C_SOURCE` should be set to a (long) value ≥ 199506, as follows:

```
cc [flags] file... -D_POSIX_C_SOURCE=N (where N 199506L)
```

You can mix Solaris threads and POSIX threads in the same application, by including both <thread.h> and <pthread.h>, and linking with either the `-lthread` or `-lpthread` library. In mixed use, Solaris semantics prevail when compiling with `-D_REENTRANT` flag set $\geq 199506L$ and linking with `-lthread`, whereas POSIX semantics prevail when compiling with

`D_POSIX_C_SOURCE` flag set $\geq 199506L$ and linking with `-lpthread`. Defining `_REENTRANT` or

`_POSIX_C_SOURCE`

Linking With libthread or libpthread

For POSIX threads behavior, load the `libpthread` library. For Solaris threads behavior, load the `libthread` library. Some POSIX programmers might want to link with `-lthread`to preserve the Solaris distinction between `fork()` and `fork1()`. All that `-lpthread` really does is to make `fork()` behave the same way as the Solaris `fork1()` call, and change the behavior of `alarm()`.

To use libthread, specify `-lthread` last on the `cc` command line.

To use libpthread, specify `-lpthread` last on the `cc` command line.

Do not link a *nonthreaded* program with `-lthread` or `-lpthread`. Doing so establishes multithreading mechanisms at link time that are initiated at run time. These *slow down* a single-threaded application, waste system resources, and produce misleading results when you debug your code.

Note: For C++ programs that use threads, use the `-mt` option, rather than `-lthread`, to compile and link your application. The `-mt` option links with `libthread` and ensures proper library linking order. (Using `-lthread` might cause your program to crash (core dump).

Linking with -lposix4 for POSIX Semaphores

The Solaris semaphore routines (see Chapter 30.3) are contained in the `libthread` library. By contrast, you link with the `-lposix4` library to get the standard POSIX semaphore routines (See Chapter 25)

Debugging a Multithreaded Program

The following list points out some of the more frequent oversights and errors that can cause bugs in multithreaded programs.

- Passing a pointer to the caller's stack as an argument to a new thread.
- Accessing global memory (shared changeable state) without the protection of a synchronization mechanism.
- Creating deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order (so that one thread controls the first resource and the other controls the second resource and neither can proceed until the other gives up).
- Trying to reacquire a lock already held (recursive deadlock).
- Creating a hidden gap in synchronization protection. This is caused when a code segment protected by a synchronization mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to the caller. The result is that it appears to the caller that the global data has been protected when it actually has not.
- Mixing UNIX signals with threads -- it is better to use the `sigwait()` model for handling asynchronous signals.
- Forgetting that default threads are created `PTHREAD_CREATE_JOINABLE` and must be reclaimed with `pthread_join()`. **Note**, `pthread_exit()` does not free up its storage space.
- Making deeply nested, recursive calls and using large automatic arrays can cause problems because multithreaded programs have a more limited stack size than single-threaded programs.
- Specifying an inadequate stack size, or using non-default stacks. And, note that multithreaded programs (especially those containing bugs) often behave differently in two successive runs, given identical inputs, because of differences in the thread scheduling order.

In general, multithreading bugs are statistical instead of deterministic. Tracing is usually a more effective method of finding order of execution problems than is breakpoint-based debugging.

Dave Marshall
1/5/1999

Subsections

Further Threads Programming:Thread Attributes (POSIX)

The previous chapter covered the basics of threads creation using default attributes. This chapter discusses setting attributes at thread creation time.

Note that only pthreads uses attributes and cancellation, so the API covered in this chapter is for POSIX threads only. Otherwise, the functionality for Solaris threads and pthreads is largely the same.

Attributes

Attributes are a way to specify behavior that is different from the default. When a thread is created with `pthread_create()` or when a synchronization variable is initialized, an attribute object can be specified. **Note:** however that the default atributes are usually sufficient for most applications.

Impottant Note: Attributes are specified *only at thread creation time*; they **cannot** be altered while the thread is **being used**.

Thus three functions are usually called in tandem

- Thread attibute intialisation -- `pthread_attr_init()` create a default `pthread_attr_t tattr`
- Thread attribute value change (unless defaults appropriate) -- a variety of `pthread_attr_*()` functions are available to set individual attribute values for the `pthread_attr_t tattr` structure. (see below).
- Thread creation -- a call to `pthread_create()` with approriate attribute values set in a `pthread_attr_t tattr` structure.

The following code fragment should make this point clearer:

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* call an appropriate functions to alter a default value */
```

```
ret = pthread_attr_*(&tattr,SOME_ATRIBUTE_VALUE_PARAMETER);

/* create the thread */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

In order to save space, code examples mainly focus on the attribute setting functions and the intializing and creation functions are ommitted. These **must** of course be present in all actual code fragtments.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type. Once an attribute is initialized and configured, it has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed. Using attribute objects has two primary advantages:

- First, it adds to code portability. Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities because the attribute object is hidden from the interface. If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well-defined location.
- Second, state specification in an application is simplified. As an example, consider that several sets of threads might exist within a process, each providing a separate service, and each with its own state requirements. At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized, and any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. The pthreads standard provides function calls to destroy attribute objects.

Initializing Thread Attributes

The function `pthread_attr_init()` is used to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

The function is prototyped by:

```
int pthread_attr_init(pthread_attr_t *tattr);
```

An example call to this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

The default values for attributes (`tattr`) are:

Attribute	Value	Result
scope	`PTHREAD_SCOPE_PROCESS`	New thread is
		unbound -
		not
		permanently
		attached to
		LWP.
detachstate	`PTHREAD_CREATE_JOINABLE`	Exit status

		and thread are
		preserved
		after the
		thread
		terminates.
stackaddr	`NULL`	New thread
		has
		system-allocated stack
		address.
stacksize	`1` megabyte	New thread
		has
		system-defined
		stack size.
		priority New thread
		inherits
		parent thread
		priority.
inheritsched	`PTHREAD_INHERIT_SCHED`	New thread
		inherits
		parent thread
		scheduling
		priority.
schedpolicy	`SCHED_OTHER`	New thread
		uses
		Solaris-defined
		fixed priority
		scheduling;
		threads run
		until
		preempted by a
		higher-priority
		thread or
		until they
		block or
		yield.

This function zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns an error value (to `errno`).

Destroying Thread Attributes

The function `pthread_attr_destroy()` is used to remove the storage allocated during initialization. The attribute object becomes invalid. It is prototyped by:

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

A sample call to this functions is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

Attribites are declared as for `pthread_attr_init()` above.

`pthread_attr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Thread's Detach State

When a thread is created detached (`PTHREAD_CREATE_DETACHED`), its thread ID and other resources can be reused as soon as the thread terminates.

If you do not want the calling thread to wait for the thread to terminate then call the function `pthread_attr_setdetachstate()`.

When a thread is created nondetached (`PTHREAD_CREATE_JOINABLE`), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a `pthread_join()` on the thread. Whether a thread is created detached or nondetached, the process does not exit until all threads have exited.

`pthread_attr_setdetachstate()` is prototyped by:

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);
```

`pthread_attr_setdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

An example call to detatch a thread with this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Note - When there is no explicit synchronization to prevent it, a newly created, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `pthread_create()`. For nondetached (`PTHREAD_CREATE_JOINABLE`) threads, it is very important that some thread join with it after it terminates -- otherwise the resources of that thread are not released for use by new threads. This commonly results in a memory leak. So when you do not want a thread to be joined, create it as a detached thread.

It is quite common that you will wish to create a thread which is detatched from creation. The following code illustrates how this may be achieved with the standard calls to initialise and set and then create a thread:

```
#include <pthread.h>
pthread_attr_t tattr;
```

```
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The function `pthread_attr_getdetachstate()` may be used to retrieve the thread create state, which can be either detached or joined. It is prototyped by:

```
int pthread_attr_getdetachstate(const pthread_attr_t *tattr, int *detachstate);
```

`pthread_attr_getdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to this fuction is:

```
#include <pthread.h>
pthread_attr_t tattr;
int detachstate;
int ret;

/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

Thread's Set Scope

A thread may be bound (`PTHREAD_SCOPE_SYSTEM`) or an unbound (`PTHREAD_SCOPE_PROCESS`). Both these types of types are accessible **only** within a given process.

The function `pthread_attr_setscope()` to create a bound or unbound thread. It is prototyped by:

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
```

Scope takes on the value of either `PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`.

`pthread_attr_setscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred and an appropriate value is returned.

So to set a bound thread at thread creation on would do the following function calls:

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
void start_routine;
void arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
/* BOUND behavior */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

If the following conditions occur, the function fails and returns the corresponding value.

The function `pthread_attr_getscope()` is used to retrieve the thread scope, which indicates whether the thread is bound or unbound. It is prototyped by:

```
int pthread_attr_getscope(pthread_attr_t *tattr, int *scope);
```

An example use of this function is:

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

If successful the approriate (`PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`) wil be stored in `scope`.

`pthread_att_getscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Thread Scheduling Policy

The POSIX draft standard specifies scheduling policy attributes of `SCHED_FIFO` (first-in-first-out), `SCHED_RR` (round-robin), or `SCHED_OTHER` (an implementation-defined method). `SCHED_FIFO` and `SCHED_RR` are optional in POSIX, and **only** are supported for *real time bound threads*.

Howver Note, currently, only the Solaris `SCHED_OTHER` default value is supported in pthreads. Attempting to set policy as `SCHED_FIFO` or `SCHED_RR` will result in the error `ENOSUP`.

The function is used to set the scheduling policy.It is prototyped by:

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

`pthread_attr_setschedpolicy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

To set the scheduling policy to `SCHED_OTHER` simply do:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

There is a function `pthread_attr_getschedpolicy()` that retrieves the scheduling policy. But, currently, it is not of great use as it can only return the (Solaris-based) `SCHED_OTHER` default value

Thread Inherited Scheduling Policy

The function `pthread_attr_setinheritsched()` can be used to the inherited scheduling policy of a thread. It is prototyped by:

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
```

An `inherit` value of `PTHREAD_INHERIT_SCHED` (the default) means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the `pthread_create()` call are to be ignored. If `PTHREAD_EXPLICIT_SCHED` is used, the attributes from the `pthread_create()` call are to be used.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call of this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

The function `pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit)` may be used to inquire a current threads scheduling policy.

Set Scheduling Parameters

Scheduling parameters are defined in the `sched_param` structure; **only** priority `sched_param.sched_priority` is supported. This priority is an integer value the higher the value the higher a thread's proiority for scehduling. Newly created threads run with this priority. The `pthread_attr_setschedparam()` is used to set this stucture appropiately. It is prototyped by:

```
int pthread_attr_setschedparam(pthread_attr_t *tattr,
const struct sched_param *param);
```

and returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to `pthread_attr_setschedparam()` is:

```
#include <pthread.h>
pthread_attr_t tattr;
int newprio;
sched_param param;


/* set the priority; others are unchanged */
newprio = 30;
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

The function `pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param)` may be used to inquire a current thread's priority of scheduling.

Thread Stack Size

Typically, thread stacks begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used as is.

When a stack is specified, the thread should also be created `PTHREAD_CREATE_JOINABLE`. That stack cannot be freed until the `pthread_join()` call for that thread has returned, because the thread's stack cannot be freed until the thread has terminated. The only reliable way to know if such a thread has terminated is through `pthread_join()`.

Generally, you do not need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of mmap to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Note: Because runtime stack requirements vary, you should be absolutely certain that the specified stack will satisfy the runtime requirements needed for library calls and dynamic linking.

There are very few occasions when it is appropriate to specify a stack, its size, or both. It is difficult even for an expert to know if the right size was specified. This is because even a program compliant with ABI standards cannot determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

Building Your Own Thread Stack

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and not have enough virtual memory to handle the gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables, and so on.

You can get the absolute minimum limit on stack size by calling the macro `PTHREAD_STACK_MIN` (defined in <pthread.h>), which returns the amount of stack space required for a thread that executes a `NULL` procedure. Useful threads need more than this, so be very careful when reducing the stack size.

The function `pthread_attr_setstacksize()` is used to set this a thread's stack size, it is prototyped by:

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, int stacksize);
```

The `stacksize` attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size.

`pthread_attr_setstacksize()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to set the stacksize is:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;


/* setting a new size */
stacksize = (PTHREAD_STACK_MIN + 0x4000);
ret = pthread_attr_setstacksize(&tattr, stacksize);
```

In the example above, size contains the size, in number of bytes, for the stack that the new thread uses. If size is zero, a default size is used. In most cases, a zero value works best. `PTHREAD_STACK_MIN` is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

The function `pthread_attr_getstacksize(pthread_attr_t *tattr, size_t *size)` may be used to inquire about a current threads stack size as follows:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;
/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &stacksize);
```

The current size of the stack is returned to the variable `stacksize`.

You may wish tp specify the base adress of thread's stack. The function `pthread_attr_setstackaddr()` does this task. It is prototyped by:

```
int pthread_attr_setstackaddr(pthread_attr_t *tattr,void *stackaddr);
```

The `stackaddr` parameter defines the base of the thread's stack. If this is set to non-null (NULL is the default) the system initializes the stack at that address.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

This example shows how to create a thread with both a custom stack address and a custom stack size.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;
int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);
/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);
/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);
/* address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

The function `pthread_attr_getstackaddr(pthread_attr_t *tattr,void * *stackaddr)` can be used to obtain the base address for a current thread's stack address.

Dave Marshall
1/5/1999

Subsections

Further Threads Programming:Synchronization

When we multiple threads running they will invariably need to communicate with each other in order *synchronise* their execution. This chapter describes the synchronization types available with threads and discusses when and how to use synchronization.

There are a few possible methods of synchronising threads:

- Mutual Exclusion (Mutex) Locks
- Condition Variables
- Semaphores

We wil frequently make use of *Synchronization objects*: these are variables in memory that you access just like data. Threads in different processes can communicate with each other through synchronization objects placed in threads-controlled shared memory, even though the threads in different processes are generally invisible to each other.

Synchronization objects can also be placed in files and can have lifetimes beyond that of the creating process.

Here are some example situations that require or can profit from the use of synchronization:

- When synchronization is the only way to ensure consistency of shared data.

- When threads in two or more processes can use a single synchronization object jointly. Note that the synchronization object should be initialized by only one of the cooperating processes, because reinitializing a synchronization object sets it to the unlocked state.
- When synchronization can ensure the safety of mutable data.
- When a process can map a file and have a thread in this process get a record's lock. Once the lock is acquired, any other thread in any process mapping the file that tries to acquire the lock is blocked until the lock is released.
- Even when accessing a single primitive variable, such as an integer. On machines where the integer is not aligned to the bus data width or is larger than the data width, a single memory load can use more than one memory cycle. While this cannot happen on the SPARC architectures, portable programs cannot rely on this.

Mutual Exclusion Locks

Mutual exclusion locks (mutexes) are a comon method of serializing thread execution. Mutual exclusion locks synchronize threads, usually by ensuring that only one thread at a time executes a critical section of code. Mutex locks can also preserve single-threaded code.

Mutex attributes may be associated with every thread. To change the default mutex attributes, you can declare and initialize an mutex attribute object and then alter specific values much like we have seen in the last chapter on more general POSIX attributes. Often, the mutex attributes are set in one place at the beginning of the application so they can be located quickly and modified easily.

After the attributes for a mutex are configured, you initialize the mutex itself. Functions are available to initialize or destroy, lock or unlock, or try to lock a mutex.

Initializing a Mutex Attribute Object

The function `pthread_mutexattr_init()` is used to initialize attributes associated with this object to their default values. It is prototyped by:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

Storage for each attribute object is allocated by the threads system during execution. `mattr` is an opaque type that contains a system-allocated attribute object. The possible values of mattr's scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.The default value of the pshared attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized mutex can be used within a process.

Before a mutex attribute object can be reinitialized, it must first be destroyed by `pthread_mutexattr_destroy()` (see below). The `pthread_mutexattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result. `pthread_mutexattr_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example of this function call is:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* initialize an attribute to default value */

ret = pthread_mutexattr_init(&mattr);
```

Destroying a Mutex Attribute Object

The function `pthread_mutexattr_destroy()` deallocates the storage space used to maintain the attribute object created by `pthread_mutexattr_init()`. It is prototyped by:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr);
```

which returns zero after completing successfully. Any other returned value indicates that an error occurred.

The function is called as follows:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

The Scope of a Mutex

The scope of a mutex variable can be either process private (intraprocess) or system wide (interprocess). The function `pthread_mutexattr_setpshared()` is used to set the scope of a mutex atrribute and it is prototype as follows:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr, int pshared);
```

If the mutex is created with the `pshared` (POSIX) attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in Solaris threads. If the mutex `pshared` attribute is set to `PTHREAD_PROCESS_PRIVATE`, only those threads created by the same process can operate on the mutex. This is equivalent to the `USYNC_THREAD` flag in `mutex_init()` in Solaris threads.

`pthread_mutexattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call is:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

ret = pthread_mutexattr_init(&mattr);

/* resetting to its default value: private */
ret = pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_PRIVATE);
```

The function `pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr, int *pshared)` may be used to obtain the scope of a current thread mutex as follows:

```
 #include <pthread.h>
pthread_mutexattr_t mattr;
int pshared, ret;

/* get pshared of mutex */ ret =
pthread_mutexattr_getpshared(&mattr, &pshared);
```

Initializing a Mutex

The function `pthread_mutex_init()` to initialize the mutex, it is prototyped by:

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
```

Here, `pthread_mutex_init()` initializes the mutex pointed at by `mp` to its default value if `mattr` is `NULL`, or to specify mutex attributes that have already been set with `pthread_mutexattr_init()`.

A mutex lock must not be reinitialized or destroyed while other threads might be using it. Program failure will result if either action is not done correctly. If a mutex is reinitialized or destroyed, the application must be sure the mutex is not currently in use. `pthread_mutex_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call is:

```
#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);
```

When the mutex is initialized, it is in an unlocked state. The effect of `mattr` being `NULL` is the same as passing the address of a default mutex attribute object, but without the memory overhead. Statically defined mutexes can be initialized directly to have default attributes with the macro `PTHREAD_MUTEX_INITIALIZER`.

To initialise a mutex with non-default values do something like:

```
/* initialize a mutex attribute */
ret = pthread_mutexattr_init(&mattr);

/* change mattr default values with some function */
 ret = pthread_mutexattr_*();

/* initialize a mutex to a non-default value */
ret = pthread_mutex_init(&mp, &mattr);
```

Locking a Mutex

The function `pthread_mute_lock()` is used to lock a mutex, it is prototyped by:

```
int pthread_mutex_lock(pthread_mutex_t *mp);
```

`pthread_mute_lock()` locks the mutex pointed to by `mp`. When the mutex is already locked, the calling thread blocks and the mutex waits on a prioritized queue. When `pthread_mute_lock()` returns, the mutex is locked and the calling thread is the owner. `pthread_mute_lock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Therefor to lock a mutex `mp` on would do the following:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;

ret = pthread_mutex_lock(&mp);
```

To unlock a mutex use the function `pthread_mutex_unlock()` whose prototype is:

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

Clearly, this function unlocks the mutex pointed to by `mp`.

The mutex must be locked and the calling thread **must** be the one that last locked the mutex (*i.e. the owner*). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked. `pthread_mutex_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call of `pthread_mutex_unlock()` is:

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

/* release the mutex */
ret = pthread_mutex_unlock(&mp);
```

Lock with a Nonblocking Mutex

The function `pthread_mutex_trylock()` to attempt to lock the mutex and is prototyped by:

```
int pthread_mutex_trylock(pthread_mutex_t *mp);
```

This function attempts to lock the mutex pointed to by `mp`. `pthread_mutex_trylock()` is a nonblocking version of `pthread_mutex_lock()`. When the mutex is already locked, this call returns with an error. Otherwise, the mutex is locked and the calling thread is the owner. `pthread_mutex_trylock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

The function is called as follows:

```
#include <pthread.h>
pthread_mutex_t mp;

/* try to lock the mutex */
int ret; ret = pthread_ mutex_trylock(&mp);
```

Destroying a Mutex

The function `pthread_mutex_destroy()` may be used to destroy any state associated with the mutex. It is prototyped by:

```
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

and destroys a mutex pointed to by mp.

Note: that the space for storing the mutex is not freed. `pthread_mutex_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

It is called by:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;

/* destroy mutex */
```

```
ret = pthread_mutex_destroy(&mp);
```

Mutex Lock Code Examples

Here are some code fragments showing mutex locking.

Mutex Lock Example

We develop two small functions that use the mutex lock for different purposes.

- The `increment_count function()` uses the mutex lock simply to ensure an atomic update of the shared variable, `count`.
- The `get_count()` function uses the mutex lock to guarantee that the (`long long`) 64-bit quantity count is read atomically. On a 32-bit architecture, a long long is really two 32-bit quantities.

The 2 functions are as follows:

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long long count;

void increment_count()
   { pthread\_mutex\_lock(&count_mutex);
     count = count + 1;
     pthread_mutex_unlock(&count_mutex);
   }


long long get_count()
   { long long c;
     pthread\_mutex\_lock(&count_mutex);
     c = count;
     pthread_mutex_unlock(&count_mutex);
     return (c);
    }
```

Recall that reading an integer value is an atomic operation because integer is the common word size on most machines.

Using Locking Hierarchies: Avoiding Deadlock

You may occasionally want to access two resources at once. For instance, you are using one of the resources, and then discover that the other resource is needed as well. However, there could be a problem if two threads attempt to claim both resources but lock the associated mutexes in different orders.

In this example, if the two threads lock mutexes 1 and 2 respectively, then a *deadlock* occurs when each attempts to lock the other mutex.

Thread 1	**Thread 2**
`/* use resource 1 */`	`/* use resource 2 */`
`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`
`/* NOW use resources 2 + 1 */`	`/* NOW use resources 1 + 2 */`
`pthread_mutex_lock(&m2);`	`pthread_mutex_lock(&m1);`

`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`

The best way to avoid this problem is to make sure that whenever threads lock multiple mutexes, they do so in the same order. This technique is known as lock hierarchies: order the mutexes by logically assigning numbers to them. Also, honor the restriction that you cannot take a mutex that is assigned n when you are holding any mutex assigned a number greater than n.

Note: The `lock_lint` tool can detect the sort of deadlock problem shown in this example.

The best way to avoid such deadlock problems is to use lock hierarchies. When locks are always taken in a prescribed order, deadlock should not occur. However, this technique cannot always be used :

- sometimes you must take the mutexes in an order other than prescribed.
- To prevent deadlock in such a situation, use `pthread_mutex_trylock()`. One thread must release its mutexes when it discovers that deadlock would otherwise be inevitable.

The idea of *Conditional Locking* use this approach:

Thread 1:

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);

/* no processing */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2:

```
for (; ;) {
  pthread_mutex_lock(&m2);
  if(pthread_mutex_trylock(&m1)==0)
    /* got it! */
    break;
 /* didn't get it */
 pthread_mutex_unlock(&m2);
 }
/* get locks; no processing */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

In the above example, thread 1 locks mutexes in the prescribed order, but thread 2 takes them out of order. To make certain that there is no deadlock, thread 2 has to take mutex 1 very carefully; if it were to block waiting for the mutex to be released, it is likely to have just entered into a deadlock with thread 1. To ensure this does not happen, thread 2 calls `pthread_mutex_trylock()`, which takes the mutex if it is available. If it is not, thread 2 returns immediately, reporting failure. At this point, thread 2 must release mutex 2, so that thread 1 can lock it, and then release both mutex 1 and mutex 2.

Nested Locking with a Singly Linked List

We have met basic linked structues in Section 10.3, when using threads which share a linked list structure the possibility of deadlock may arise.

By nesting mutex locks into the linked data structure and a simple ammendment of the link list code we can prevent deadlock by taking the locks in a prescribed order.

The modified linked is as follows:

```
typedef struct node1 {
     int value;
```

```
    struct node1 *link;
    pthread_mutex_t lock;
  } node1_t;
```

Note: we simply ammend a standard singly-linked list structure so that each node containing a mutex.

Assuming we have created a variable `node1_t ListHead`.

To remove a node from the list:

- first search the list starting at ListHead (which itself is never removed) until the desired node is found.
- To protect this search from the effects of concurrent deletions, lock each node before any of its contents are accessed.

  Because all searches start at ListHead, there is never a deadlock because the locks are always taken in list order.
- When the desired node is found, lock both the node and its predecessor since the change involves both nodes.

  Because the predecessor's lock is always taken first, you are again protected from deadlock.

The C code to remove an item from a singly linked list with nested locking is as follows:

```
node1_t *delete(int value)
  { node1_t *prev,
    *current; prev = &ListHead;

   pthread_mutex_lock(&prev->lock);
   while ((current = prev->link) != NULL)
     { pthread_mutex_lock(&current->lock);
       if (current->value == value)
         { prev->link = current->link;
           pthread_mutex_unlock(&current->lock);
           pthread_mutex_unlock(&prev->lock);
           current->link = NULL; return(current);
         }
       pthread_mutex_unlock(&prev->lock);
       prev = current;
      }
    pthread_mutex_unlock(&prev->lock);
    return(NULL);
   }
```

Solaris Mutex Locks

Similar mutual exclusion locks exist for in Solaris.

You should include the <synch.h> or <thread.h>libraries.

To initialize a mutex use `int mutex_init(mutex_t *mp, int type, void *arg))`. `mutex_init()` initializes the mutex pointed to by `mp`. The `type` can be one of the following (note that `arg` is currently ignored).

USYNC_PROCESS

-- The mutex can be used to synchronize threads in this and other processes.

USYNC_THREAD

-- The mutex can be used to synchronize threads in this process, only.

Mutexes can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using it.

The function `int mutex_destroy (mutex_t *mp)` destroys any state associated with the mutex pointed to by `mp`. **Note** that the space for storing the mutex is not freed.

To acquire a mutex lock use the function `mutex_lock(mutex_t *mp)` which locks the mutex pointed to by mp. When the mutex is already locked, the calling thread blocks until the mutex becomes available (blocked threads wait on a prioritized queue).

To release a mutex use `mutex_unlock(mutex_t *mp)` which unlocks the mutex pointed to by mp. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner).

To try to acquire a mutex use mutex_trylock(mutex_t *mp) to attempt to lock the mutex pointed to by `mp`. This function is a nonblocking version of `mutex_lock()`

Condition Variable Attributes

Condition variables can be usedto atomically block threads until a particular condition is true. Condition variables are *always* used in conjunction with mutex locks:

- With a condition variable, a thread can atomically block until a condition is satisfied.
- The condition is tested under the protection of a mutual exclusion lock (mutex).
    - When the condition is false, a thread usually blocks on a condition variable and atomically releases the mutex waiting for the condition to change.
    - When another thread changes the condition, it can signal the associated condition variable to cause one or more waiting threads to wake up, acquire the mutex again, and reevaluate the condition.

Condition variables can be used to synchronize threads among processes when they are allocated in memory that can be written to and is shared by the cooperating processes.

The scheduling policy determines how blocking threads are awakened. For the default `SCHED_OTHER`, threads are awakened in priority order. The attributes for condition variables must be set and initialized before the condition variables can be used.

As with mutex locks, The condiotion variable attributes must be initialised and set (or set to `NULL`) before an actual condition variable may be initialise (with appropriat attributes) and then used.

Initializing a Condition Variable Attribute

The function `pthread_condattr_init()` initializes attributes associated with this object to their default values. It is prototyped by:

```
int pthread_condattr_init(pthread_condattr_t *cattr);
```

Storage for each attribute object, `cattr`, is allocated by the threads system during execution. `cattr` is an opaque data type that contains a system-allocated attribute object. The possible values of cattr's scope are `PTHREAD_PROCESS_PRIVATE` and `PTHREAD_PROCESS_SHARED`. The default value of the pshared attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized condition variable can be used within a process.

Before a condition variable attribute can be reused, it must first be reinitialized by `pthread_condattr_destroy()`. The `pthread_condattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

`pthread_condattr_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

A simple example call of this function is :

```
#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_condattr_init(&cattr);
```

Destoying a Condition Variable Attribute

The function `pthread_condattr_destroy()` removes storage and renders the attribute object invalid, it is prototyped by:

```
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

`pthread_condattr_destroy()` returns zero after completing successfully and destroying the condition variable pointed to by `cattr`. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

The Scope of a Condition Variable

The scope of a condition variable can be either process private (intraprocess) or system wide (interprocess), as with mutex locks. If the condition variable is created with the pshared attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in the original Solaris threads. If the mutex pshared attribute is set to `PTHREAD_PROCESS_PRIVATE` (default value), only those threads created by the same process can operate on the mutex. Using `PTHREAD_PROCESS_PRIVATE` results in the same behavior as with the `USYNC_THREAD` flag in the original Solaris threads `cond_init()` call, which is that of a local condition variable. `PTHREAD_PROCESS_SHARED` is equivalent to a global condition variable.

The function `pthread_condattr_setpshared()` is used to set the scope of a condition variable, it is prototyped by:

```
int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared);
```

The condition variable attribute `cattr` must be initialised first and the value of `pshared` is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

`pthread_condattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A sample use of this function is as follows:

```
#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* Scope: all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* OR */
/* Scope: within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

The function `int pthread_condattr_getpshared(const pthread_condattr_t *cattr, int *pshared)` may be used to obtain the scope of a given condition variable.

Initializing a Condition Variable

The function `pthread_cond_init()` initializes the condition variable and is prototyped as follows:

```
int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);
```

The condition variable which is initialized is pointed at by `cv` and is set to its default value if `cattr` is `NULL`, or to specific `cattr` condition variable attributes that are already set with `pthread_condattr_init()`. The effect of `cattr` being NULL is the same as passing the address of a default condition variable attribute object, but without the memory overhead.

Statically-defined condition variables can be initialized directly to have default attributes with the macro `PTHREAD_COND_INITIALIZER`. This has the same effect as dynamically allocating `pthread_cond_init()` with null attributes. No error checking is done. Multiple threads must not simultaneously initialize or reinitialize the same condition variable. If a condition variable is reinitialized or destroyed, the application must be sure the condition variable is not in use.

`pthread_cond_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Sample calls of this function are:

```
#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */ ret =
pthread_cond_init(&cv, &cattr);
```

Block on a Condition Variable

The function `pthread_cond_wait()` is used to atomically release a mutex and to cause the calling thread to block on the condition variable. It is protoyped by:

```
int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
```

The mutex that is released is pointed to by `mutex` and the condition variable pointed to by `cv` is blocked.

`pthread_cond_wait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

A simple example call is:

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mutex;
int ret;

/* wait on condition variable */
ret = pthread_cond_wait(&cv, &mutex);
```

The blocked thread can be awakened by a `pthread_cond_signal()`, a `pthread_cond_broadcast()`, or when interrupted by delivery of a signal. Any change in the value of a condition associated with the condition variable cannot be inferred by the return of `pthread_cond_wait()`, and any such condition must be reevaluated. The `pthread_cond_wait()` routine always returns with the mutex locked and owned by the calling thread, even when returning an error. This function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically acquires it again before returning. In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to acquire the mutex lock again. Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired.

The recommended test method is to write the condition check as a while loop that calls `pthread_cond_wait()`, as follows:

```
pthread_mutex_lock();

while(condition_is_false)
  pthread_cond_wait();
pthread_mutex_unlock();
```

No specific order of acquisition is guaranteed when more than one thread blocks on the condition variable. Note also that `pthread_cond_wait()` is a cancellation point. If a cancel is pending and the calling thread has cancellation enabled, the thread terminates and begins executing its cleanup handlers while continuing to hold the lock.

To unblock a specific thread use `pthread_cond_signal()` which is prototyped by:

```
int pthread_cond_signal(pthread_cond_t *cv);
```

This unblocks one thread that is blocked on the condition variable pointed to by `cv`. `pthread_cond_signal()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

You should always call `pthread_cond_signal()` under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait. The scheduling policy determines the order in which blocked threads are awakened. For `SCHED_OTHER`, threads are awakened in priority order. When no threads are blocked on the condition variable, then calling `pthread_cond_signal()`l has no effect.

The folloowing code fragment illustrates how to avoid an infinite problem described above:

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
   { pthread_mutex_lock(&count_lock);

     while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
     count = count - 1;
     pthread_mutex_unlock(&count_lock);
   }

increment_count()
   { pthread_mutex_lock(&count_lock);
     if (count == 0)
       pthread_cond_signal(&count_nonzero);
     count = count + 1;
     pthread_mutex_unlock(&count_lock);
    }
```

You can also block until a specified event occurs. The function `pthread_cond_timedwait()` is used for this purpose. It is prototyped by:

```
int pthread_cond_timedwait(pthread_cond_t *cv,
      pthread_mutex_t *mp, const struct timespec *abstime);
```

`pthread_cond_timedwait()` is used in a similar manner to `pthread_cond_wait()`: `pthread_cond_timedwait()` blocks until the condition is signaled or until the time of day, specified by `abstime`, has passed. `pthread_cond_timedwait()` always returns with the mutex, `mp`, locked and owned by the calling thread, even when it is returning an error. `pthread_cond_timedwait()` is also a cancellation point.

`pthread_cond_timedwait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

An examle call of this function is:

```
#include <pthread.h>
#include <time.h>

pthread_timestruc_t to;
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* wait on condition variable */

ret = pthread_cond_timedwait(&cv, &mp, &abstime);


pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;

while (cond == FALSE)
  {  err = pthread_cond_timedwait(&c, &m, &to);
     if (err == ETIMEDOUT)
       { /* timeout, do something */
         break;
       }
   }
pthread_mutex_unlock(&m);
```

All threads may be unblocked in one function: `pthread_cond_broadcast()`. This function is prototyped as follows:

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

`pthread_cond_broadcast()` unblocks all threads that are blocked on the condition variable pointed to by `cv`, specified by `pthread_cond_wait()`. When no threads are blocked on the condition variable, `pthread_cond_broadcast()` has no effect.

`pthread_cond_broadcast()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

Since `pthread_cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, use carefully. For example, use `pthread_cond_broadcast()` to allow threads to contend for varying resource amounts when resources are freed:

```
#include <pthread.h>

pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
  { pthread_mutex_lock(&rsrc_lock);
    while (resources < amount)
       pthread_cond_wait(&rsrc_add, &rsrc_lock);

   resources -= amount;
   pthread_mutex_unlock(&rsrc_lock);
  }

add_resources(int amount)
  { pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
  }
```

Note: that in `add_resources` it does not matter whether resources is updated first or if `pthread_cond_broadcast()` is called first inside the mutex lock. Call `pthread_cond_broadcast()` under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

Destroying a Condition Variable State

The function `pthread_cond_destroy()` to destroy any state associated with the condition variable, it is prototyped by:

```
int pthread_cond_destroy(pthread_cond_t *cv);
```

The condition variable pointed to by `cv` will be destroyed by this call:

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

Note that the space for storing the condition variable is not freed.

`pthread_cond_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

Solaris Condition Variables

Similar condition variables exist in Solaris. The functions are prototyped in <thread.h>.

To initialize a condition variable use `int cond_init(cond_t *cv, int type, int arg)` which initializes the condition variable pointed to by `cv`. The `type` can be one of `USYNC_PROCESS` or `USYNC_THREAD`

(See Solaris mutex (Section 30.1.9 for more details). Note that `arg` is currently ignored.

Condition variables can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be reinitialized while other threads might be using it.

To destroy a condition variable use `int cond_destroy(cond_t *cv)` which destroys a state associated with the condition variable pointed to by `cv`. The space for storing the condition variable is not freed.

To wait for a condition use `int cond_wait(cond_t *cv, mutex_t *mp)` which atomically releases the mutex pointed to by `mp` and to cause the calling thread to block on the condition variable pointed to by `cv`.

The blocked thread can be awakened by `cond_signal(cond_t *cv)`, `cond_broadcast(cond_t *cv)`, or when interrupted by delivery of a signal or a fork. Use `cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by `cv`. Call this function under protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be signaled between its test and `cond_wait()`, causing an infinite wait. Use `cond_broadcast()` to unblock all threads that are blocked on the condition variable pointed to by `cv`. When no threads are blocked on the condition variable then `cond_broadcast()` has no effect.

Finally, to wait until the condition is signaled or for an absolute time use `int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime)` Use `cond_timedwait()` as you would use `cond_wait()`, except that `cond_timedwait()` does not block past the time of day specified by abstime. `cond_timedwait()` always returns with the mutex locked and owned by the calling thread even when returning an error.

Threads and Semaphores

POSIX Semaphores

Chapter 25 has dealt with semaphore programming for POSIX and System V IPC semaphores.

Semaphore operations are the same in both POSIX and Solaris. The function names are changed from `sema_` in Solaris to `sem_` in pthreads. Solaris semaphore are defined in <thread.h>.

In this section we give a brief description of Solaris thread semaphores.

Basic Solaris Semaphore Functions

To initialize the function `int sema_init(sema_t *sp, unsigned int count, int type, void *arg)` is used. `sema`. `type` can be one of the following):

`USYNC_PROCESS`
: -- The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore.

`USYNC_THREAD`
: -- The semaphore can be used to synchronize threads in this process.

`arg` is currently unused.

Multiple threads **must not** initialize the same semaphore simultaneously. A semaphore **must not** be reinitialized while other threads may be using it.

To increment a Semaphore use the function `int sema_post(sema_t *sp)`. `sema_post` atomically increments the semaphore pointed to by `sp`. When any threads are blocked on the semaphore, one is unblocked.

To block on a Semaphore use `int sema_wait(sema_t *sp)`. `sema_wait()` to block the calling thread until the count in the semaphore pointed to by `sp` becomes greater than zero, then atomically decrement it.

To decrement a Semaphore count use `int sema_trywait(sema_t *sp)`. `sema_trywait()` atomically decrements the count in the semaphore pointed to by `sp` when the count is greater than zero. This function is a nonblocking version of `sema_wait()`.

To destroy the Semaphore state call the function `sema_destroy(sema_t *sp)`. `sema_destroy()` to destroy any state associated with the semaphore pointed to by `sp`. The space for storing the semaphore is not freed.

Dave Marshall
1/5/1999

Subsections

Thread programming examples

This chapter gives some full code examples of thread programs. These examles are taken from a variety of sources:

- The sun workshop developers web page *http://www.sun.com/workshop/threads/share-code/* on threads is an excelleny source
- The web page *http://www.sun.com/workshop/threads/Berg-Lewis/examples.html* where example from the *Threads Primer* Book by D. Berg anD B. Lewis are also a major resource.

Using `thr_create()` and `thr_join()`

This example exercises the `thr_create()` and `thr_join()` calls. There is not a parent/child relationship between threads as there is for processes. This can easily be seen in this example, because threads are created and joined by many different threads in the process. The example also shows how threads behave when created with different attributes and options.

Threads can be created by any thread and joined by any other.

The main thread: In this example the main thread's sole purpose is to create new threads. Threads A, B, and C are created by the main thread. Notice that thread B is created suspended. After creating the new threads, the main thread exits. Also notice that the main thread exited by calling thr_exit(). If the main thread had used the exit() call, the whole process would have exited. The main thread's exit status and resources are held until it is joined by thread C.

Thread A: The first thing thread A does after it is created is to create thread D. Thread A then simulates some processing and then exits, using `thr_exit()`. Notice that thread A was created with the `THR_DETACHED` flag, so thread A's resources will be immediately reclaimed upon its exit. There is no way for thread A's exit status to be collected by a `thr_join()` call.

Thread B: Thread B was created in a suspended state, so it is not able to run until thread D continues it by making the `thr_continue()` call. After thread B is continued, it simulates some processing and then exits. Thread B's exit status and thread resources are held until joined by thread E.

Thread C: The first thing that thread C does is to create thread F. Thread C then joins the main thread. This action will collect the main thread's exit status and allow the main thread's resources to be reused by another thread. Thread C will block, waiting for the main thread to exit, if the main thread has not yet called `thr_exit()`. After joining the main thread, thread C will simulate some processing and then exit. Again, the exit status and thread resources are held until joined by thread E.

Thread D: Thread D immediately creates thread E. After creating thread E, thread D continues thread B by making the `thr_continue()` call. This call will allow thread B to start its execution. Thread D then tries to join thread E, blocking until thread E has exited. Thread D then simulates some processing and exits. If all went well, thread D should be the last nondaemon thread running. When thread D exits, it should do two things: stop the execution of any daemon threads and stop the execution of the process.

Thread E: Thread E starts by joining two threads, threads B and C. Thread E will block, waiting for each of these thread to exit. Thread E will then simulate some processing and will exit. Thread E's exit status and thread resources are held by the operating system until joined by thread D.

Thread F: Thread F was created as a bound, daemon thread by using the `THR_BOUND` and `THR_DAEMON` flags in the `thr_create()` call. This means that it will run on its own LWP until all the nondaemon threads have exited the process. This type of thread can be used when you want some type of "background" processing to always be running, except when all the "regular" threads have exited the process. If thread F was created as a non-daemon thread, then it would continue to run forever, because a process will continue while there is at least one thread still running. Thread F will exit when all the nondaemon threads have exited. In this case, thread D should be the last nondaemon thread running, so when thread D exits, it will also cause thread F to exit.

This example, however trivial, shows how threads behave differently, based on their creation options. It also shows what happens on the exit of a thread, again based on how it was created. If you understand this example and how it flows, you should have a good understanding of how to use `thr_create()` and `thr_join()` in your own programs. Hopefully you can also see how easy it is to create and join threads.

The source to `multi_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Function prototypes for thread routines */
void *sub_a(void *);
void *sub_b(void *);
void *sub_c(void *);
void *sub_d(void *);
void *sub_e(void *);
void *sub_f(void *);

thread_t thr_a, thr_b, thr_c;

void main()
{
thread_t main_thr;

main_thr = thr_self();
printf("Main thread = %d\n", main_thr);

if (thr_create(NULL, 0, sub_b, NULL, THR_SUSPENDED|THR_NEW_LWP, &thr_b))
        fprintf(stderr,"Can't create thr_b\n"), exit(1);

if (thr_create(NULL, 0, sub_a, (void *)thr_b, THR_NEW_LWP, &thr_a))
        fprintf(stderr,"Can't create thr_a\n"), exit(1);

if (thr_create(NULL, 0, sub_c, (void *)main_thr, THR_NEW_LWP, &thr_c))
        fprintf(stderr,"Can't create thr_c\n"), exit(1);

printf("Main Created threads A:%d B:%d C:%d\n", thr_a, thr_b, thr_c);
printf("Main Thread exiting...\n");
thr_exit((void *)main_thr);
}

void *sub_a(void *arg)
{
thread_t thr_b = (thread_t) arg;
thread_t thr_d;
int i;

printf("A: In thread A...\n");

if (thr_create(NULL, 0, sub_d, (void *)thr_b, THR_NEW_LWP, &thr_d))
        fprintf(stderr, "Can't create thr_d\n"), exit(1);

printf("A: Created thread D:%d\n", thr_d);
```

```
/* process
*/
for (i=0;i<1000000*(int)thr_self();i++);
printf("A: Thread exiting...\n");
thr_exit((void *)77);
}

void * sub_b(void *arg)
{
int i;

printf("B: In thread B...\n");

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("B: Thread exiting...\n");
thr_exit((void *)66);
}

void * sub_c(void *arg)
{
void *status;
int i;
thread_t main_thr, ret_thr;

main_thr = (thread_t)arg;

printf("C: In thread C...\n");

if (thr_create(NULL, 0, sub_f, (void *)0, THR_BOUND|THR_DAEMON, NULL))
        fprintf(stderr, "Can't create thr_f\n"), exit(1);

printf("C: Join main thread\n");

if (thr_join(main_thr,(thread_t *)&ret_thr, &status))
        fprintf(stderr, "thr_join Error\n"), exit(1);

printf("C: Main thread (%d) returned thread (%d) w/status %d\n", main_thr, ret_thr,
(int) status);

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("C: Thread exiting...\n");
thr_exit((void *)88);
}


void * sub_d(void *arg)
{
thread_t thr_b = (thread_t) arg;
int i;
thread_t thr_e, ret_thr;
void *status;

printf("D: In thread D...\n");

if (thr_create(NULL, 0, sub_e, NULL, THR_NEW_LWP, &thr_e))
        fprintf(stderr,"Can't create thr_e\n"), exit(1);
```

```
printf("D: Created thread E:%d\n", thr_e);
printf("D: Continue B thread = %d\n", thr_b);

thr_continue(thr_b);
printf("D: Join E thread\n");

if(thr_join(thr_e,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("D: E thread (%d) returned thread (%d) w/status %d\n", thr_e,
ret_thr, (int) status);

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("D: Thread exiting...\n");
thr_exit((void *)55);
}


void * sub_e(void *arg)
{
int i;
thread_t ret_thr;
void *status;

printf("E: In thread E...\n");
printf("E: Join A thread\n");

if(thr_join(thr_a,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: A thread (%d) returned thread (%d) w/status %d\n", ret_thr, ret_thr, (int)
status);
printf("E: Join B thread\n");

if(thr_join(thr_b,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: B thread (%d) returned thread (%d) w/status %d\n", thr_b, ret_thr, (int)
status);
printf("E: Join C thread\n");

if(thr_join(thr_c,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: C thread (%d) returned thread (%d) w/status %d\n", thr_c, ret_thr, (int)
status);

for (i=0;i<1000000*(int)thr_self();i++);

printf("E: Thread exiting...\n");
thr_exit((void *)44);
}


void *sub_f(void *arg)
{
int i;

printf("F: In thread F...\n");
```

```
while (1) {
        for (i=0;i<10000000;i++);
        printf("F: Thread F is still running...\n");
        }
}
```

Arrays

This example uses a data structure that contains multiple arrays of data. Multiple threads will concurrently vie for access to the arrays. To control this access, a mutex variable is used within the data structure to lock the entire array and serialize the access to the data.

The main thread first initializes the data structure and the mutex variable. It then sets a level of concurrency and creates the worker threads. The main thread then blocks by joining all the threads. When all the threads have exited, the main thread prints the results.

The worker threads modify the shared data structure from within a loop. Each time the threads need to modify the shared data, they lock the mutex variable associated with the shared data. After modifying the data, the threads unlock the mutex, allowing another thread access to the data.

This example may look quite simple, but it shows how important it is to control access to a simple, shared data structure. The results can be quite different if the mutex variable is not used.

The source to `array.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* sample array data structure */
struct {
        mutex_t data_lock[5];
        int     int_val[5];
        float   float_val[5];
        } Data;

/* thread function */
void *Add_to_Value();

main()
{
int i;

/* initialize the mutexes and data */
for (i=0; i<5; i++) {
        mutex_init(&Data.data_lock[i], USYNC_THREAD, 0);
        Data.int_val[i] = 0;
        Data.float_val[i] = 0;
        }

/* set concurrency and create the threads */
thr_setconcurrency(4);

for (i=0; i<5; i++)
    thr_create(NULL, 0, Add_to_Value, (void *)(2*i), 0, NULL);

/* wait till all threads have finished */
for (i=0; i<5; i++)
        thr_join(0,0,0);

/* print the results */
printf("Final Values.....\n");
```

```
for (i=0; i<5; i++) {
        printf("integer value[%d] =\t%d\n", i, Data.int_val[i]);
        printf("float value[%d] =\t%.0f\n\n", i, Data.float_val[i]);
        }

return(0);
}


/* Threaded routine */
void *Add_to_Value(void *arg)
{
int inval = (int) arg;
int i;

for (i=0;i<10000;i++){
    mutex_lock(&Data.data_lock[i%5]);
       Data.int_val[i%5] += inval;
       Data.float_val[i%5] += (float) 1.5 * inval;
    mutex_unlock(&Data.data_lock[i%5]);
    }

return((void *)0);
}
```

Deadlock

This example demonstrates how a deadlock can occur in multithreaded programs that use synchronization variables. In this example a thread is created that continually adds a value to a global variable. The thread uses a mutex lock to protect the global data.

The main thread creates the counter thread and then loops, waiting for user input. When the user presses the Return key, the main thread suspends the counter thread and then prints the value of the global variable. The main thread prints the value of the global variable under the protection of a mutex lock.

The problem arises in this example when the main thread suspends the counter thread while the counter thread is holding the mutex lock. After the main thread suspends the counter thread, it tries to lock the mutex variable. Since the mutex variable is already held by the counter thread, which is suspended, the main thread deadlocks.

This example may run fine for a while, as long as the counter thread just happens to be suspended when it is not holding the mutex lock. The example demonstrates how tricky some programming issues can be when you deal with threads.

The source to `susp_lock.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Prototype for thread subroutine */
void *counter(void *);

int count;
mutex_t count_lock;

main()
{
char str[80];
thread_t ctid;

/* create the thread counter subroutine */
thr_create(NULL, 0, counter, 0, THR_NEW_LWP|THR_DETACHED, &ctid);
```

```
while(1) {
        gets(str);
        thr_suspend(ctid);

        mutex_lock(&count_lock);
        printf("\n\nCOUNT = %d\n\n", count);
        mutex_unlock(&count_lock);

        thr_continue(ctid);
        }

return(0);
}

void *counter(void *arg)
{
int i;

while (1) {
        printf("."); fflush(stdout);

        mutex_lock(&count_lock);
        count++;

        for (i=0;i<50000;i++);

        mutex_unlock(&count_lock);

        for (i=0;i<50000;i++);
        }

return((void *)0);
}
```

Signal Handler

This example shows how easy it is to handle signals in multithreaded programs. In most programs, a different signal handler would be needed to service each type of signal that you wanted to catch. Writing each of the signal handlers can be time consuming and can be a real pain to debug.

This example shows how you can implement a signal handler thread that will service all asynchronous signals that are sent to your process. This is an easy way to deal with signals, because only one thread is needed to handle all the signals. It also makes it easy when you create new threads within the process, because you need not worry about signals in any of the threads.

First, in the main thread, mask out all signals and then create a signal handling thread. Since threads inherit the signal mask from their creator, any new threads created after the new signal mask will also mask all signals. This idea is key, because the only thread that will receive signals is the one thread that does not block all the signals.

The signal handler thread waits for all incoming signals with the sigwait() call. This call unmasks the signals given to it and then blocks until a signal arrives. When a signal arrives, sigwait() masks the signals again and then returns with the signal ID of the incoming signal.

You can extend this example for use in your application code to handle all your signals. Notice also that this signal concept could be added in your existing nonthreaded code as a simpler way to deal with signals.

The source to `thr_sig.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <signal.h>
#include <sys/types.h>

void *signal_hand(void *);
```

```
main()
{
sigset_t set;

/* block all signals in main thread.  Any other threads that are
   created after this will also block all signals */

sigfillset(&set);

thr_sigsetmask(SIG_SETMASK, &set, NULL);

/* create a signal handler thread.  This thread will catch all
   signals and decide what to do with them.  This will only
   catch nondirected signals.  (I.e., if a thread causes a SIGFPE
   then that thread will get that signal. */

thr_create(NULL, 0, signal_hand, 0, THR_NEW_LWP|THR_DAEMON|THR_DETACHED, NULL);

while (1) {
        /*
        Do your normal processing here....
        */
        }  /* end of while */

return(0);
}

void *signal_hand(void *arg)
{
sigset_t set;
int sig;

sigfillset(&set); /* catch all signals */

while (1) {
        /* wait for a signal to arrive */

        switch (sig=sigwait(&set)) {

          /* here you would add whatever signal you needed to catch */
          case SIGINT : {
                        printf("Interrupted with signal %d, exiting...\n", sig);
                        exit(0);
                        }

          default : printf("GOT A SIGNAL = %d\n", sig);
          } /* end of switch */
        } /* end of while */

return((void *)0);
} /* end of signal_hand */
```

Another example of a signal handler, `sig_kill.c`:

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
```

```
 *  Users may copy, modify or distribute this file at will.
 *
 *  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
 *  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
 *  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
 *
 *  This file is provided with no support and without any obligation on the
 *  part of SunSoft, Inc. to assist in its use, correction, modification or
 *  enhancement.
 *
 *  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
 *  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
 *  FILE OR ANY PART THEREOF.
 *
 *  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
 *  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
 *  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
 *  DAMAGES.
 *
 *  SunSoft, Inc.
 *  2550 Garcia Avenue
 *  Mountain View, California  94043
 */

/*
  * Rich Schiavi writes:                 Sept 11, 1994
  *
  * I believe the recommended way to kill certain threads is
  * using a signal handler which then will exit that particular
  * thread properly. I'm not sure the exact reason (I can't remember), but
  * if you take out the signal_handler routine in my example, you will see what
  * you describe, as the main process dies even if you send the
  * thr_kill to the specific thread.

  * I whipped up a real quick simple example which shows this using
  * some sleep()s to get a good simulation.
  */

#include <stdio.h>
#include <thread.h>
#include <signal.h>

static  thread_t        one_tid, two_tid, main_thread;
static  void    *first_thread();
static  void    *second_thread();
void            ExitHandler(int);

static  mutex_t         first_mutex, second_mutex;
int     first_active = 1 ;
int     second_active = 1;

main()

{
  int i;
  struct sigaction act;

  act.sa_handler = ExitHandler;
  (void) sigemptyset(&act.sa_mask);
  (void) sigaction(SIGTERM, &act, NULL);

  mutex_init(&first_mutex, 0 , 0);
  mutex_init(&second_mutex, 0 , 0);
```

```
  main_thread = thr_self();

  thr_create(NULL,0,first_thread,0,THR_NEW_LWP,&one_tid);
  thr_create(NULL,0,second_thread,0,THR_NEW_LWP,&two_tid);

  for (i = 0; i < 10; i++){
    fprintf(stderr, "main loop: %d\n", i);
    if (i == 5) {
      thr_kill(one_tid, SIGTERM);
    }
    sleep(3);
  }
  thr_kill(two_tid, SIGTERM);
  sleep(5);
  fprintf(stderr, "main exit\n");
}

static void *first_thread()
{
  int i = 0;

  fprintf(stderr, "first_thread id: %d\n", thr_self());
  while (first_active){
    fprintf(stderr, "first_thread: %d\n", i++);
    sleep(2);
  }
  fprintf(stderr, "first_thread exit\n");
}

static void *second_thread()
{
  int i = 0;

  fprintf(stderr, "second_thread id: %d\n", thr_self());

  while (second_active){
    fprintf(stderr, "second_thread: %d\n", i++);
    sleep(3);
  }
  fprintf(stderr, "second_thread exit\n");
}

void ExitHandler(int sig)
{
  thread_t id;

  id = thr_self();

  fprintf(stderr, "ExitHandler thread id: %d\n", id);
  thr_exit(0);

}
```

Interprocess Synchronization

This example uses some of the synchronization variables available in the threads library to synchronize access to a resource shared between two processes. The synchronization variables used in the threads library are an advantage over standard IPC synchronization mechanisms because of their speed. The synchronization variables in the threads libraries have been tuned to be very lightweight and very fast. This speed can be an advantage when your application is spending time synchronizing between processes.

This example shows how semaphores from the threads library can be used between processes. Note that this program does not use threads; it is just using the lightweight semaphores available from the threads library.

When using synchronization variables between processes, it is important to make sure that only one process initializes the variable. If both processes try to initialize the synchronization variable, then one of the processes will overwrite the state of the variable set by the other process.

The source to `ipc.c`

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <synch.h>
#include <sys/types.h>
#include <unistd.h>

/* a structure that will be used between processes */
typedef struct {
        sema_t mysema;
        int num;
} buf_t;

main()
{
int     i, j, fd;
buf_t   *buf;

/* open a file to use in a memory mapping */
fd = open("/dev/zero", O_RDWR);

/* create a shared memory map with the open file for the data
   structure that will be shared between processes */
buf=(buf_t *)mmap(NULL, sizeof(buf_t), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

/* initialize the semaphore -- note the USYNC_PROCESS flag; this makes
   the semaphore visible from a process level */
sema_init(&buf->mysema, 0, USYNC_PROCESS, 0);

/* fork a new process */
if (fork() == 0) {
        /* The child will run this section of code */
        for (j=0;j<5;j++)
                {
                /* have the child "wait" for the semaphore */

                printf("Child PID(%d): waiting...\n", getpid());
                sema_wait(&buf->mysema);

                /* the child decremented the semaphore */

                printf("Child PID(%d): decrement semaphore.\n", getpid());
                }
        /* exit the child process */
        printf("Child PID(%d): exiting...\n", getpid());
        exit(0);
        }

/* The parent will run this section of code */
/* give the child a chance to start running */

sleep(2);

for (i=0;i<5;i++)
        {
        /* increment (post) the semaphore */
```

```
        printf("Parent PID(%d): posting semaphore.\n", getpid());
        sema_post(&buf->mysema);

        /* wait a second */
        sleep(1);
        }

/* exit the parent process */
printf("Parent PID(%d): exiting...\n", getpid());

return(0);
}
```

The Producer / Consumer Problem

This example will show how condition variables can be used to control access of reads and writes to a buffer. This example can also be thought as a producer/consumer problem, where the producer adds items to the buffer and the consumer removes items from the buffer.

Two condition variables control access to the buffer. One condition variable is used to tell if the buffer is full, and the other is used to tell if the buffer is empty. When the producer wants to add an item to the buffer, it checks to see if the buffer is full; if it is full the producer blocks on the `cond_wait()` call, waiting for an item to be removed from the buffer. When the consumer removes an item from the buffer, the buffer is no longer full, so the producer is awakened from the `cond_wait()` call. The producer is then allowed to add another item to the buffer.

The consumer works, in many ways, the same as the producer. The consumer uses the other condition variable to determine if the buffer is empty. When the consumer wants to remove an item from the buffer, it checks to see if it is empty. If the buffer is empty, the consumer then blocks on the `cond_wait()` call, waiting for an item to be added to the buffer. When the producer adds an item to the buffer, the consumer's condition is satisfied, so it can then remove an item from the buffer.

The example copies a file by reading data into a shared buffer (producer) and then writing data out to the new file (consumer). The Buf data structure is used to hold both the buffered data and the condition variables that control the flow of the data.

The main thread opens both files, initializes the `Buf` data structure, creates the consumer thread, and then assumes the role of the producer. The producer reads data from the input file, then places the data into an open buffer position. If no buffer positions are available, then the producer waits via the `cond_wait()` call. After the producer has read all the data from the input file, it closes the file and waits for (joins) the consumer thread.

The consumer thread reads from a shared buffer and then writes the data to the output file. If no buffers positions are available, then the consumer waits for the producer to fill a buffer position. After the consumer has read all the data, it closes the output file and exits.

If the input file and the output file were residing on different physical disks, then this example could execute the reads and writes in parallel. This parallelism would significantly increase the throughput of the example through the use of threads.

The source to `prod_cons.c`:

```
#define _REEENTRANT
#include <stdio.h>
#include <thread.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>

#define BUFSIZE 512
#define BUFCNT  4

/* this is the data structure that is used between the producer
   and consumer threads */

struct {
        char buffer[BUFCNT][BUFSIZE];
        int byteinbuf[BUFCNT];
```

```
        mutex_t buflock;
        mutex_t donelock;
        cond_t adddata;
        cond_t remdata;
        int nextadd, nextrem, occ, done;
} Buf;

/* function prototype */
void *consumer(void *);

main(int argc, char **argv)
{
int ifd, ofd;
thread_t cons_thr;

/* check the command line arguments */
if (argc != 3)
        printf("Usage: %s <infile> <outfile>\n", argv[0]), exit(0);

/* open the input file for the producer to use */
if ((ifd = open(argv[1], O_RDONLY)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[1]);
        exit(1);
        }

/* open the output file for the consumer to use */
if ((ofd = open(argv[2], O_WRONLY|O_CREAT, 0666)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[2]);
        exit(1);
        }

/* zero the counters */
Buf.nextadd = Buf.nextrem = Buf.occ = Buf.done = 0;

/* set the thread concurrency to 2 so the producer and consumer can
   run concurrently */

thr_setconcurrency(2);

/* create the consumer thread */
thr_create(NULL, 0, consumer, (void *)ofd, NULL, &cons_thr);

/* the producer ! */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        /* check to see if any buffers are empty */
        /* If not then wait for that condition to become true */

        while (Buf.occ == BUFCNT)
                cond_wait(&Buf.remdata, &Buf.buflock);

        /* read from the file and put data into a buffer */
        Buf.byteinbuf[Buf.nextadd] = read(ifd,Buf.buffer[Buf.nextadd],BUFSIZE);

        /* check to see if done reading */
        if (Buf.byteinbuf[Buf.nextadd] == 0) {

                /* lock the done lock */
```

```
                mutex_lock(&Buf.donelock);

                /* set the done flag and release the mutex lock */
                Buf.done = 1;

                mutex_unlock(&Buf.donelock);

                /* signal the consumer to start consuming */
                cond_signal(&Buf.adddata);

                /* release the buffer mutex */
                mutex_unlock(&Buf.buflock);

                /* leave the while looop */
                break;
                }

        /* set the next buffer to fill */
        Buf.nextadd = ++Buf.nextadd % BUFCNT;

        /* increment the number of buffers that are filled */
        Buf.occ++;

        /* signal the consumer to start consuming */
        cond_signal(&Buf.adddata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

close(ifd);

/* wait for the consumer to finish */
thr_join(cons_thr, 0, NULL);

/* exit the program */
return(0);
}


/* The consumer thread */
void *consumer(void *arg)
{
int fd = (int) arg;

/* check to see if any buffers are filled or if the done flag is set */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        if (!Buf.occ && Buf.done) {
           mutex_unlock(&Buf.buflock);
           break;
           }

        /* check to see if any buffers are filled */
        /* if not then wait for the condition to become true */

        while (Buf.occ == 0 && !Buf.done)
                cond_wait(&Buf.adddata, &Buf.buflock);

        /* write the data from the buffer to the file */
```

```
        write(fd, Buf.buffer[Buf.nextrem], Buf.byteinbuf[Buf.nextrem]);

        /* set the next buffer to write from */
        Buf.nextrem = ++Buf.nextrem % BUFCNT;

        /* decrement the number of buffers that are full */
        Buf.occ--;

        /* signal the producer that a buffer is empty */
        cond_signal(&Buf.remdata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

/* exit the thread */
thr_exit((void *)0);
}
```

A Socket Server

The socket server example uses threads to implement a "standard" socket port server. The example shows how easy it is to use `thr_create()` calls in the place of `fork()` calls in existing programs.

A standard socket server should listen on a socket port and, when a message arrives, fork a process to service the request. Since a `fork()` system call would be used in a nonthreaded program, any communication between the parent and child would have to be done through some sort of interprocess communication.

We can replace the `fork()` call with a `thr_create()` call. Doing so offers a few advantages: `thr_create()` can create a thread much faster then a `fork()` could create a new process, and any communication between the *server* and the new thread can be done with common variables. This technique makes the implementation of the socket server much easier to understand and should also make it respond much faster to incoming requests.

The server program first sets up all the needed socket information. This is the basic setup for most socket servers. The server then enters an endless loop, waiting to service a socket port. When a message is sent to the socket port, the server wakes up and creates a new thread to handle the request. Notice that the server creates the new thread as a detached thread and also passes the socket descriptor as an argument to the new thread.

The newly created thread can then read or write, in any fashion it wants, to the socket descriptor that was passed to it. At this point the server could be creating a new thread or waiting for the next message to arrive. The key is that the server thread does not care what happens to the new thread after it creates it.

In our example, the created thread reads from the socket descriptor and then increments a global variable. This global variable keeps track of the number of requests that were made to the server. Notice that a mutex lock is used to protect access to the shared global variable. The lock is needed because many threads might try to increment the same variable at the same time. The mutex lock provides serial access to the shared variable. See how easy it is to share information among the new threads! If each of the threads were a process, then a significant effort would have to be made to share this information among the processes.

The client piece of the example sends a given number of messages to the server. This client code could also be run from different machines by multiple users, thus increasing the need for concurrency in the server process.

The source code to `soc_server.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/uio.h>
#include <unistd.h>
#include <thread.h>

/* the TCP port that is used for this example */
```

```
#define TCP_PORT   6500

/* function prototypes and global variables */
void *do_chld(void *);
mutex_t lock;
int     service_count;

main()
{
        int     sockfd, newsockfd, clilen;
        struct sockaddr_in cli_addr, serv_addr;
        thread_t chld_thr;

        if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                fprintf(stderr,"server: can't open stream socket\n"), exit(0);

        memset((char *) &serv_addr, 0, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port = htons(TCP_PORT);

        if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) <
0)
                fprintf(stderr,"server: can't bind local address\n"), exit(0);

        /* set the level of thread concurrency we desire */
        thr_setconcurrency(5);

        listen(sockfd, 5);

        for(;;){
                clilen = sizeof(cli_addr);
                newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);

                if(newsockfd < 0)
                        fprintf(stderr,"server: accept error\n"), exit(0);

                /* create a new thread to process the incomming request */
                thr_create(NULL, 0, do_chld, (void *) newsockfd, THR_DETACHED,
&chld_thr);

                /* the server is now free to accept another socket request */
                }
        return(0);
}

/*
        This is the routine that is executed from a new thread
*/

void *do_chld(void *arg)
{
int     mysocfd = (int) arg;
char    data[100];
int     i;

        printf("Child thread [%d]: Socket number = %d\n", thr_self(), mysocfd);

        /* read from the given socket */
        read(mysocfd, data, 40);

        printf("Child thread [%d]: My data = %s\n", thr_self(), data);
```

```
        /* simulate some processing */
        for (i=0;i<1000000*thr_self();i++);

        printf("Child [%d]: Done Processing...\n", thr_self());

        /* use a mutex to update the global service counter */
        mutex_lock(&lock);

        service_count++;
        mutex_unlock(&lock);

        printf("Child thread [%d]: The total sockets served = %d\n", thr_self(),
service_count);

        /* close the socket and exit this thread */
        close(mysocfd);
        thr_exit((void *)0);
}
```

Using Many Threads

This example that shows how easy it is to create many threads of execution in Solaris. Because of the lightweight nature of threads, it is possible to create literally thousands of threads. Most applications may not need a very large number of threads, but this example shows just how lightweight the threads can be.

We have said before that anything you can do with threads, you can do without them. This may be a case where it would be very hard to do without threads. If you have some spare time (and lots of memory), try implementing this program by using processes, instead of threads. If you try this, you will see why threads can have an advantage over processes.

This program takes as an argument the number of threads to create. Notice that all the threads are created with a user-defined stack size, which limits the amount of memory that the threads will need for execution. The stack size for a given thread can be hard to calculate, so some testing usually needs to be done to see if the chosen stack size will work. You may want to change the stack size in this program and see how much you can lower it before things stop working. The Solaris threads library provides the `thr_min_stack()` call, which returns the minimum allowed stack size. Take care when adjusting the size of a threads stack. A stack overflow can happen quite easily to a thread with a small stack.

After each thread is created, it blocks, waiting on a mutex variable. This mutex variable was locked before any of the threads were created, which prevents the threads from proceeding in their execution. When all of the threads have been created and the user presses Return, the mutex variable is unlocked, allowing all the threads to proceed.

After the main thread has created all the threads, it waits for user input and then tries to join all the threads. Notice that the `thr_join()` call does not care what thread it joins; it is just counting the number of joins it makes.

This example is rather trivial and does not serve any real purpose except to show that it is possible to create a lot of threads in one process. However, there are situations when many threads are needed in an application. An example might be a network port server, where a thread is created each time an incoming or outgoing request is made.

The source to `many_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <thread.h>

/* function prototypes and global varaibles */
void *thr_sub(void *);
mutex_t lock;

main(int argc, char **argv)
{
int i, thr_count = 100;
char buf;
```

```
/* check to see if user passed an argument
   -- if so, set the number of threads to the value
      passed to the program */

if (argc == 2) thr_count = atoi(argv[1]);

printf("Creating %d threads...\n", thr_count);

/* lock the mutex variable -- this mutex is being used to
   keep all the other threads created from proceeding   */

mutex_lock(&lock);

/* create all the threads -- Note that a specific stack size is
   given.  Since the created threads will not use all of the
   default stack size, we can save memory by reducing the threads'
   stack size */

for (i=0;i<thr_count;i++) {
        thr_create(NULL,2048,thr_sub,0,0,NULL);
        }

printf("%d threads have been created and are running!\n", i);
printf("Press <return> to join all the threads...\n", i);

/* wait till user presses return, then join all the threads */
gets(&buf);

printf("Joining %d threads...\n", thr_count);

/* now unlock the mutex variable, to let all the threads proceed */
mutex_unlock(&lock);

/* join the threads */
for (i=0;i<thr_count;i++)
        thr_join(0,0,0);

printf("All %d threads have been joined, exiting...\n", thr_count);
return(0);
}

/* The routine that is executed by the created threads */

void *thr_sub(void *arg)
{
/* try to lock the mutex variable -- since the main thread has
   locked the mutex before the threads were created, this thread
   will block until the main thread unlock the mutex */

mutex_lock(&lock);

printf("Thread %d is exiting...\n", thr_self());

/* unlock the mutex to allow another thread to proceed */
mutex_unlock(&lock);

/* exit the thread */
return((void *)0);
}
```

Real-time Thread Example

This example uses the Solaris real-time extensions to make a single bound thread within a process run in the real-time scheduling class. Using a thread in the real-time class is more desirable than running a whole process in the real-time class, because of the many problems that can arise with a process in a real-time state. For example, it would not be desirable for a process to perform any I/O or large memory operations while in realtime, because a real-time process has priority over system-related processes; if a real-time process requests a page fault, it can starve, waiting for the system to fault in a new page. We can limit this exposure by using threads to execute only the instructions that need to run in realtime.

Since this book does not cover the concerns that arise with real-time programming, we have included this code only as an example of how to promote a thread into the real-time class. You must be very careful when you use real-time threads in your applications. For more information on real-time programming, see the Solaris documentation.

This example should be safe from the pitfalls of real-time programs because of its simplicity. However, changing this code in any way could have adverse affects on your system.

The example creates a new thread from the main thread. This new thread is then promoted to the real-time class by looking up the real-time class ID and then setting a real-time priority for the thread. After the thread is running in realtime, it simulates some processing. Since a thread in the real-time class can have an infinite time quantum, the process is allowed to stay on a CPU as long as it likes. The time quantum is the amount of time a thread is allowed to stay running on a CPU. For the timesharing class, the time quantum (time-slice) is 1/100th of a second by default.

In this example, we set the time quantum for the real-time thread to infinity. That is, it can stay running as long as it likes; it will not be preempted or scheduled off the CPU. If you run this example on a UP machine, it will have the effect of stopping your system for a few seconds while the thread simulates its processing. The system does not actually stop, it is just working in the real-time thread. When the real-time thread finishes its processing, it exits and the system returns to normal.

Using real-time threads can be quite useful when you need an extremely high priority and response time but can also cause big problems if it not used properly. Also note that this example must be run as root or have root execute permissions.

The source to `rt_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <string.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>

/* thread prototype */
void *rt_thread(void *);

main()
{

/* create the thread that will run in realtime */
thr_create(NULL, 0, rt_thread, 0, THR_DETACHED, 0);

/* loop here forever, this thread is the TS scheduling class */
while (1) {
        printf("MAIN: In time share class... running\n");
        sleep(1);
        }

return(0);
}

/*
        This is the routine that is called by the created thread
*/

void *rt_thread(void *arg)
{
pcinfo_t pcinfo;
```

```
pcparms_t pcparms;
int i;

/* let the main thread run for a bit */
sleep(4);

/* get the class ID for the real-time class */
strcpy(pcinfo.pc_clname, "RT");

if (priocntl(0, 0, PC_GETCID, (caddr_t)&pcinfo) == -1)
        fprintf(stderr, "getting RT class id\n"), exit(1);

/* set up the real-time parameters */
pcparms.pc_cid = pcinfo.pc_cid;
((rtparms_t *)pcparms.pc_clparms)->rt_pri = 10;
((rtparms_t *)pcparms.pc_clparms)->rt_tqnsecs = 0;

/* set an infinite time quantum */
((rtparms_t *)pcparms.pc_clparms)->rt_tqsecs = RT_TQINF;

/* move this thread to the real-time scheduling class */
if (priocntl(P_LWPID, P_MYID, PC_SETPARMS, (caddr_t)&pcparms) == -1)
        fprintf(stderr, "Setting RT mode\n"), exit(1);

/* simulate some processing */
for (i=0;i<100000000;i++);

printf("RT_THREAD: NOW EXITING...\n");
thr_exit((void *)0);
}
```

POSIX Cancellation

This example uses the POSIX thread cancellation capability to kill a thread that is no longer needed. Random termination of a thread can cause problems in threaded applications, because a thread may be holding a critical lock when it is terminated. Since the lock was help before the thread was terminated, another thread may deadlock, waiting for that same lock. The thread cancellation capability enables you to control when a thread can be terminated. The example also demonstrates the capabilities of the POSIX thread library in implementing a program that performs a multithreaded search.

This example simulates a multithreaded search for a given number by taking random guesses at a target number. The intent here is to simulate the same type of search that a database might execute. For example, a database might create threads to start searching for a data item; after some amount of time, one or more threads might return with the target data item.

If a thread guesses the number correctly, there is no need for the other threads to continue their search. This is where thread cancellation can help. The thread that finds the number first should cancel the other threads that are still searching for the item and then return the results of the search.

The threads involved in the search can call a cleanup function that can clean up the threads resources before it exits. In this case, the cleanup function prints the progress of the thread when it was cancelled.

The source to `posix_cancel.c`:

```
 #define _REENTRANT
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

/* defines the number of searching threads */
#define NUM_THREADS 25

/* function prototypes */
```

```
void *search(void *);
void print_it(void *);

/* global variables */
pthread_t  threads[NUM_THREADS];
pthread_mutex_t lock;
int tries;

main()
{
int i;
int pid;

/* create a number to search for */
pid = getpid();

/* initialize the mutex lock */
pthread_mutex_init(&lock, NULL);
printf("Searching for the number = %d...\n", pid);

/* create the searching threads */
for (i=0;i<NUM_THREADS;i++)
        pthread_create(&threads[i], NULL, search, (void *)pid);

/* wait for (join) all the searching threads */
for (i=0;i<NUM_THREADS;i++)
        pthread_join(threads[i], NULL);

printf("It took %d tries to find the number.\n", tries);

/* exit this thread */
pthread_exit((void *)0);
}

/*
        This is the cleanup function that is called when
        the threads are cancelled
*/

void print_it(void *arg)
{
int *try = (int *) arg;
pthread_t tid;

/* get the calling thread's ID */
tid = pthread_self();

/* print where the thread was in its search when it was cancelled */
printf("Thread %d was canceled on its %d try.\n", tid, *try);
}

/*
        This is the search routine that is executed in each thread
*/

void *search(void *arg)
{
int num = (int) arg;
int i=0, j;
pthread_t tid;

/* get the calling thread ID */
tid = pthread_self();
```

```
    /* use the thread ID to set the seed for the random number generator */
    srand(tid);

    /* set the cancellation parameters --
       - Enable thread cancellation
       - Defer the action of the cancellation
    */

    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    /* push the cleanup routine (print_it) onto the thread
       cleanup stack.  This routine will be called when the
       thread is cancelled.  Also note that the pthread_cleanup_push
       call must have a matching pthread_cleanup_pop call.  The
       push and pop calls MUST be at the same lexical level
       within the code */

    /* pass address of `i' since the current value of `i' is not
       the one we want to use in the cleanup function */

    pthread_cleanup_push(print_it, (void *)&i);

    /* loop forever */
    while (1) {
            i++;

            /* does the random number match the target number? */
            if (num == rand()) {

                    /* try to lock the mutex lock --
                       if locked, check to see if the thread has been cancelled
                       if not locked then continue */

                    while (pthread_mutex_trylock(&lock) == EBUSY)
                                    pthread_testcancel();

                    /* set the global variable for the number of tries */

                    tries = i;

                    printf("thread %d found the number!\n", tid);

                    /* cancel all the other threads */
                    for (j=0;j<NUM_THREADS;j++)
                            if (threads[j] != tid) pthread_cancel(threads[j]);

                    /* break out of the while loop */
                    break;
                    }

            /* every 100 tries check to see if the thread has been cancelled
               if the thread has not been cancelled then yield the thread's
               LWP to another thread that may be able to run */

            if (i%100 == 0) {
                    pthread_testcancel();
                    sched_yield();
                    }
            }

    /* The only way we can get here is when the thread breaks out
```

```
   of the while loop.  In this case the thread that makes it here
   has found the number we are looking for and does not need to run
   the thread cleanup function.  This is why the pthread_cleanup_pop
   function is called with a 0 argument; this will pop the cleanup
   function off the stack without executing it */

pthread_cleanup_pop(0);
return((void *)0);
}
```

Software Race Condition

This example shows a trivial software race condition. A software race condition occurs when the execution of a program is affected by the order and timing of a threads execution. Most software race conditions can be alleviated by using synchronization variables to control the threads' timing and access of shared resources. If a program depends on order of execution, then threading that program may not be a good solution, because the order in which threads execute is nondeterministic.

In the example, `thr_continue()` and `thr_suspend()` calls continue and suspend a given thread, respectively. Although both of these calls are valid, use caution when implementing them. It is very hard to determine where a thread is in its execution. Because of this, you may not be able to tell where the thread will suspend when the call to `thr_suspend()` is made. This behavior can cause problems in threaded code if not used properly.

The following example uses `thr_continue()` and `thr_suspend()` to try to control when a thread starts and stops. The example looks trivial, but, as you will see, can cause a big problem.

Do you see the problem? If you guessed that the program would eventually suspend itself, you were correct! The example attempts to flip-flop between the main thread and a subroutine thread. Each thread continues the other thread and then suspends itself.

Thread A continues thread B and then suspends thread A; now the continued thread B can continue thread A and then suspend itself. This should continue back and forth all day long, right? Wrong! We can't guarantee that each thread will continue the other thread and then suspend itself in one atomic action, so a software race condition could be created. Calling `thr_continue()` on a running thread and calling `thr_suspend()` on a suspended thread has no effect, so we don't know if a thread is already running or suspended.

If thread A continues thread B and if between the time thread A suspends itself, thread B continues thread A, then both of the threads will call `thr_suspend()`. This is the race condition in this program that will cause the whole process to become suspended.

It is very hard to use these calls, because you never really know the state of a thread. If you don't know exactly where a thread is in its execution, then you don't know what locks it holds and where it will stop when you suspend it.

The source to `sw_race.c`

`Tgrep`: Threadeds version of UNIX `grep`

`Tgrep` is a multi-threaded version of `grep`. `Tgrep` supports all but the -w (word search) options of the normal `grep` command, and a few options that are only avaliable to `Tgrep`. The real change from `grep`, is that `Tgrep` will recurse down through sub-directories and search all files for the target string. `Tgrep` searches files like the following command:

```
find <start path> -name "<file/directory pattern>" -exec \ (Line wrapped)
        grep <options> <target> /dev/null {} \;
```

An example of this would be (run from this `Tgrep` directory)

```
% find . -exec grep thr_create /dev/null {} \;
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
Running the same command with timex:
real        4.26
```

```
user        0.64
sys         2.81
```

The same search run with `Tgrep` would be

```
% {\tt Tgrep} thr_create
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
Running the same command with timex:
real        0.79
user        0.62
sys         1.50
```

`Tgrep` gets the results almost four times faster. The numbers above where gathered on a SS20 running 5.5 (build 18) with 4 50MHz CPUs.

You can also filter the files that you want `Tgrep` to search like you can with find. The next two commands do the same thing, just `Tgrep` gets it done faster.

```
find . -name "*.c" -exec grep thr_create /dev/null {} \;
and
{\tt Tgrep} -p '.*\.c$' thr_create
```

The -p option will allow `Tgrep` to search only files that match the "regular expression" file pattern string. This option does NOT use shell expression, so to stop `Tgrep` from seeing a file named foobar.cyou must add the "$" meta character to the pattern and escape the real ``." character.

Some of the other `Tgrep` only options are -r, -C, -P, -e, -B, -S and -Z. The -r option stops `Tgrep` from searching any sub-directories, in other words, search only the local directory, but -l was taken. The -C option will search for and print "continued" lines like you find in Makefile. Note the differences in the results of grep and `Tgrep` run in the current directory.

The `Tgrep` output prints the continued lines that ended with the "character. In the case of grep I would not have seen the three values assigned to SUBDIRS, but `Tgrep` shows them to me (Common, Solaris, Posix).

The -P option I use when I am sending the output of a long search to a file and want to see the "progress" of the search. The -P option will print a "." (dot) on stderr for every file (or groups of files depending on the value of the -P argument) `Tgrep` searches.

The -e option will change the way `Tgrep` uses the target string. `Tgrep` uses two different patter matching systems. The first (with out the -e option) is a literal string match call Boyer-Moore. If the -e option is used, then a MT-Safe PD version of regular expression is used to search for the target string as a regexp with meta characters in it. The regular expression method is slower, but `Tgrep` needed the functionality. The -Z option will print help on the meta characters `Tgrep` uses.

The -B option tells `Tgrep` to use the value of the environment variable called TGLIMIT to limit the number of threads it will use during a search. This option has no affect if TGLIMIT is not set. `Tgrep` can "eat" a system alive, so the -B option was a way to run `Tgrep` on a system with out having other users scream at you.

The last new option is -S. If you want to see how things went while `Tgrep` was searching, you can use this option to print statistic about the number of files, lines, bytes, matches, threads created, etc.

Here is an example of the -S options output. (again run in the current directory)

```
% {\tt Tgrep} -S zimzap

----------------- {\tt Tgrep} Stats. --------------------
Number of directories searched:           7
Number of files searched:                 37
Number of lines searched:                 9504
Number of matching lines to target:       0
Number of cascade threads created:        7
Number of search threads created:         20
Number of search threads from pool:       17
Search thread pool hit rate:              45.95%
Search pool overall size:                 20
Search pool size limit:                   58
```

```
Number of search threads destroyed:       0
Max # of threads running concurrenly:     20
Total run time, in seconds.               1
Work stopped due to no FD's:  (058)       0 Times, 0.00%
Work stopped due to no work on Q:         19 Times, 43.18%
Work stopped due to TGLIMITS: (Unlimited) 0 Times, 0.00%
----------------------------------------------------
%
```

For more information on the usage and options, see the man page `Tgrep`

The `Tgrep.c` source code is:

```
/* Copyright (c) 1993, 1994  Ron Winacott                               */
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.              */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#ifdef __sparc
#include <note.h> /* warlock/locklint */
#else
#define NOTE(s)
#endif
#include <dirent.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <thread.h>
#include <synch.h>

#include "version.h"
#include "pmatch.h"
#include "debug.h"

#define PATH_MAX                1024 /* max # of characters in a path name */
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
#define FX_STDIN                0x00800
#define TG_BATCH                0x01000
#define TG_FILEPAT              0x02000
```

```
#define FE_REGEXP               0x04000
#define FS_STATS                0x08000
#define FC_LINE                 0x10000
#define TG_PROGRESS             0x20000

#define FILET                   1
#define DIRT                    2
#define ALPHASIZ                128

/*
 * New data types
 */

typedef struct work_st {
    char                *path;
    int                 tp;
    struct work_st      *next;
} work_t;

typedef struct out_st {
    char                *line;
    int                 line_count;
    long                byte_count;
    struct out_st       *next;
} out_t;

typedef struct bm_pattern {     /* Boyer - Moore pattern                */
        short            p_m;           /* length of pattern string     */
        short            p_r[ALPHASIZ]; /* "r" vector                   */
        short           *p_R;           /* "R" vector                   */
        char            *p_pat;         /* pattern string               */
} BM_PATTERN;


/*
 * Prototypes
 */

/* bmpmatch.c */
extern BM_PATTERN *bm_makepat(char *);
extern char *bm_pmatch(BM_PATTERN *, register char *);
extern void bm_freepat(BM_PATTERN *);
/* pmatch.c */
extern char *pmatch(register PATTERN *, register char *, int *);
extern PATTERN *makepat(char *string, char *);
extern void freepat(register PATTERN *);
extern void printpat(PATTERN *);

#include "proto.h"  /* function prototypes of main.c */

void *SigThread(void *arg);
void sig_print_stats(void);

/*
 * Global data
 */

BM_PATTERN      *bm_pat;  /* the global target read only after main */
NOTE(READ_ONLY_DATA(bm_pat))

PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch */
NOTE(READ_ONLY_DATA(pm_pat))
```

```
mutex_t global_count_lk;
int     global_count = 0;
NOTE(MUTEX_PROTECTS_DATA(global_count_lk, global_count))
NOTE(DATA_READABLE_WITHOUT_LOCK(global_count))  /* see prnt_stats() */

work_t  *work_q = NULL;
cond_t  work_q_cv;
mutex_t work_q_lk;
int     all_done = 0;
int     work_cnt = 0;
int     current_open_files = 0;
int     tglimit = UNLIMITED;    /* if -B limit the number of threads */
NOTE(MUTEX_PROTECTS_DATA(work_q_lk, work_q all_done work_cnt \
                         current_open_files tglimit))

work_t  *search_q = NULL;
mutex_t search_q_lk;
cond_t  search_q_cv;
int     search_pool_cnt = 0;    /* the count in the pool now */
int     search_thr_limit = 0;   /* the max in the pool */
NOTE(MUTEX_PROTECTS_DATA(search_q_lk, search_q search_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(search_pool_cnt)) /* see prnt_stats() */
NOTE(READ_ONLY_DATA(search_thr_limit))

work_t  *cascade_q = NULL;
mutex_t cascade_q_lk;
cond_t  cascade_q_cv;
int     cascade_pool_cnt = 0;
int     cascade_thr_limit = 0;
NOTE(MUTEX_PROTECTS_DATA(cascade_q_lk, cascade_q cascade_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(cascade_pool_cnt))  /* see prnt_stats() */
NOTE(READ_ONLY_DATA(cascade_thr_limit))

int     running = 0;
mutex_t running_lk;
NOTE(MUTEX_PROTECTS_DATA(running_lk, running))

sigset_t set, oldset;
NOTE(READ_ONLY_DATA(set oldset))

mutex_t stat_lk;
time_t  st_start = 0;
int     st_dir_search = 0;
int     st_file_search = 0;
int     st_line_search = 0;
int     st_cascade = 0;
int     st_cascade_pool = 0;
int     st_cascade_destroy = 0;
int     st_search = 0;
int     st_pool = 0;
int     st_maxrun = 0;
int     st_worknull = 0;
int     st_workfds = 0;
int     st_worklimit = 0;
int     st_destroy = 0;
NOTE(MUTEX_PROTECTS_DATA(stat_lk, st_start st_dir_search st_file_search \
                         st_line_search st_cascade st_cascade_pool \
                         st_cascade_destroy st_search st_pool st_maxrun \
                         st_worknull st_workfds st_worklimit st_destroy))

int     progress_offset = 1;
NOTE(READ_ONLY_DATA(progress_offset))
```

```
mutex_t output_print_lk;
/* output_print_lk used to print multi-line output only */
int     progress = 0;
NOTE(MUTEX_PROTECTS_DATA(output_print_lk, progress))

unsigned int    flags = 0;
int     regexp_cnt = 0;
char    *string[MAXREGEXP];
int     debug = 0;
int     use_pmatch = 0;
char    file_pat[255];  /* file patten match */
PATTERN *pm_file_pat; /* compiled file target string (pmatch()) */
NOTE(READ_ONLY_DATA(flags regexp_cnt string debug use_pmatch \
                    file_pat pm_file_pat))


/*
 * Locking ording.
 */
NOTE(LOCK_ORDER(output_print_lk stat_lk))

/*
 * Main: This is where the fun starts
 */

int
main(int argc, char **argv)
{
    int         c,out_thr_flags;
    long        max_open_files = 0l, ncpus = 0l;
    extern int  optind;
    extern char *optarg;
    NOTE(READ_ONLY_DATA(optind optarg))
    int         prio = 0;
    struct stat sbuf;
    thread_t    tid,dtid;
    void        *status;
    char        *e = NULL, *d = NULL; /* for debug flags */
    int         debug_file = 0;
    int         err = 0, i = 0, pm_file_len = 0;
    work_t      *work;
    int         restart_cnt = 10;

    flags = FR_RECUR;  /* the default */

    thr_setprio(thr_self(),127);  /* set me up HIGH */
    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) {
        switch (c) {
#ifdef DEBUG
        case 'd':
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
```

```
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case 'f':
            debug_file = atoi(optarg);
            break;
#endif      /* DEBUG */
        case 'B':
            flags |= TG_BATCH;
            if ((e = getenv("TGLIMIT"))) {
                tglimit = atoi(e);
            }
            else {
                if (!(flags & FS_NOERROR))  /* order dependent! */
                    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
                flags &= ~TG_BATCH;
            }
            break;
        case 'p':
            flags |= TG_FILEPAT;
            strcpy(file_pat,optarg);
            pm_file_pat = makepat(file_pat,NULL);
            break;
        case 'P':
            flags |= TG_PROGRESS;
            progress_offset = atoi(optarg);
            break;
        case 'S': flags |= FS_STATS;    break;
        case 'b': flags |= FB_BLOCK;    break;
        case 'c': flags |= FC_COUNT;    break;
        case 'h': flags |= FH_HOLDNAME; break;
        case 'i': flags |= FI_IGNCASE;  break;
        case 'l': flags |= FL_NAMEONLY; break;
        case 'n': flags |= FN_NUMBER;   break;
        case 's': flags |= FS_NOERROR;  break;
        case 'v': flags |= FV_REVERSE;  break;
        case 'w': flags |= FW_WORD;     break;
        case 'r': flags &= ~FR_RECUR;   break;
        case 'C': flags |= FC_LINE;     break;
        case 'e':
            if (regexp_cnt == MAXREGEXP) {
                fprintf(stderr,"Max number of regexp's (%d) exceeded!\n",
                        MAXREGEXP);
                exit(1);
            }
            flags |= FE_REGEXP;
            if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NULL){
                fprintf(stderr,"tgrep: No space for search string(s)\n");
                exit(1);
            }
            memset(string[regexp_cnt],0,strlen(optarg)+1);
            strcpy(string[regexp_cnt],optarg);
            regexp_cnt++;
            break;
        case 'z':
        case 'Z': regexp_usage();
            break;
        case 'H':
        case '?':
        default : usage();
        }
    }
```

```
    if (!(flags & FE_REGEXP)) {
       if (argc - optind < 1) {
            fprintf(stderr,"tgrep: Must supply a search string(s) "
                    "and file list or directory\n");
            usage();
        }
        if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
            fprintf(stderr,"tgrep: No space for search string(s)\n");
            exit(1);
        }
        memset(string[0],0,strlen(argv[optind])+1);
        strcpy(string[0],argv[optind]);
        regexp_cnt=1;
        optind++;
    }

    if (flags & FI_IGNCASE)
        for (i=0; i<regexp_cnt; i++)
            uncase(string[i]);

#ifdef __lock_lint
    /*
    ** This is NOT somthing you really want to do. This
    ** function calls are here ONLY for warlock/locklint !!!
    */
    pm_pat[i] = makepat(string[i],NULL);
    bm_pat = bm_makepat(string[0]);
    bm_freepat(bm_pat);  /* stop it from becomming a root */
#else
    if (flags & FE_REGEXP) {
        for (i=0; i<regexp_cnt; i++)
            pm_pat[i] = makepat(string[i],NULL);
        use_pmatch = 1;
    }
    else {
        bm_pat = bm_makepat(string[0]); /* only one allowed */
    }
#endif

    flags |= FX_STDIN;

    max_open_files = sysconf(_SC_OPEN_MAX);
    ncpus = sysconf(_SC_NPROCESSORS_ONLN);
    if ((max_open_files - HOLD_FDS - debug_file) < 1) {
        fprintf(stderr,"tgrep: You MUST have at lest ONE fd "
                "that can be used, check limit (>10)\n");
        exit(1);
    }
    search_thr_limit = max_open_files - HOLD_FDS - debug_file;
    cascade_thr_limit = search_thr_limit / 2;
    /* the number of files that can by open */
    current_open_files = search_thr_limit;

    mutex_init(&stat_lk,USYNC_THREAD,"stat");
    mutex_init(&global_count_lk,USYNC_THREAD,"global_cnt");
    mutex_init(&output_print_lk,USYNC_THREAD,"output_print");
    mutex_init(&work_q_lk,USYNC_THREAD,"work_q");
    mutex_init(&running_lk,USYNC_THREAD,"running");
    cond_init(&work_q_cv,USYNC_THREAD,"work_q");
    mutex_init(&search_q_lk,USYNC_THREAD,"search_q");
    cond_init(&search_q_cv,USYNC_THREAD,"search_q");
    mutex_init(&cascade_q_lk,USYNC_THREAD,"cascade_q");
```

```
    cond_init(&cascade_q_cv,USYNC_THREAD,"cascade_q");

    if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
        add_work(".",DIRT);
        flags = (flags & ~FX_STDIN);
    }
    for ( ; optind < argc; optind++) {
        restart_cnt = 10;
        flags = (flags & ~FX_STDIN);
      STAT_AGAIN:
        if (stat(argv[optind], &sbuf)) {
            if (errno == EINTR) { /* try again !, restart */
                if (--restart_cnt)
                    goto STAT_AGAIN;
            }
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't stat file/dir %s, %s\n",
                        argv[optind], strerror(errno));
            continue;
        }
        switch (sbuf.st_mode & S_IFMT) {
        case S_IFREG :
            if (flags & TG_FILEPAT) {
                if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
                    add_work(argv[optind],FILET);
            }
            else {
                add_work(argv[optind],FILET);
            }
            break;
        case S_IFDIR :
            if (flags & FR_RECUR) {
                add_work(argv[optind],DIRT);
            }
            else {
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can't search directory %s, "
                            "-r option is on. Directory ignored.\n",
                            argv[optind]);
            }
            break;
        }
    }

    NOTE(COMPETING_THREADS_NOW)  /* we are goinf threaded */

    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        st_start = time(NULL);
        mutex_unlock(&stat_lk);
#ifdef SIGNAL_HAND
        /*
        ** setup the signal thread so the first call to SIGINT will
        ** only print stats, the second will interupt.
        */
        sigfillset(&set);
        thr_sigsetmask(SIG_SETMASK, &set, &oldset);
        if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
            thr_sigsetmask(SIG_SETMASK,&oldset,NULL);
            fprintf(stderr,"SIGINT for stats NOT setup\n");
        }
        thr_yield(); /* give the other thread time */
#endif /* SIGNAL_HAND */
```

```
    }

    thr_setconcurrency(3);

    if (flags & FX_STDIN) {
        fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
        exit(0);                        /* XXX Need to fix this SOON */
        search_thr(NULL);  /* NULL is not understood in search_thr() */
        if (flags & FC_COUNT) {
            mutex_lock(&global_count_lk);
            printf("%d\n",global_count);
            mutex_unlock(&global_count_lk);
        }
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            prnt_stats();
            mutex_unlock(&stat_lk);
        }
        exit(0);
    }

    mutex_lock(&work_q_lk);
    if (!work_q) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: No files to search.\n");
        exit(0);
    }
    mutex_unlock(&work_q_lk);

    DP(DLEVEL1,("Starting to loop through the work_q for work\n"));

    /* OTHER THREADS ARE RUNNING */
    while (1) {
        mutex_lock(&work_q_lk);
        while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &&
               all_done == 0) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                if (work_q == NULL)
                    st_worknull++;
                if (current_open_files == 0)
                    st_workfds++;
                if (tglimit <= 0)
                    st_worklimit++;
                mutex_unlock(&stat_lk);
            }
            cond_wait(&work_q_cv,&work_q_lk);
        }
        if (all_done != 0) {
            mutex_unlock(&work_q_lk);
            DP(DLEVEL1,("All_done was set to TRUE\n"));
            goto OUT;
        }
        work = work_q;
        work_q = work->next;  /* maybe NULL */
        work->next = NULL;
        current_open_files--;
        mutex_unlock(&work_q_lk);

        tid = 0;
        switch (work->tp) {
        case DIRT:
            mutex_lock(&cascade_q_lk);
```

```
            if (cascade_pool_cnt) {
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_cascade_pool++;
                    mutex_unlock(&stat_lk);
                }
                work->next = cascade_q;
                cascade_q = work;
                cond_signal(&cascade_q_cv);
                mutex_unlock(&cascade_q_lk);
                DP(DLEVEL2,("Sent work to cascade pool thread\n"));
            }
            else {
                mutex_unlock(&cascade_q_lk);
                err = thr_create(NULL,0,cascade,(void *)work,
                                 THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                                 ,&tid);
                DP(DLEVEL2,("Sent work to new cascade thread\n"));
                thr_setprio(tid,64);  /* set cascade to middle */
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_cascade++;
                    mutex_unlock(&stat_lk);
                }
            }
            break;
        case FILET:
            mutex_lock(&search_q_lk);
            if (search_pool_cnt) {
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_pool++;
                    mutex_unlock(&stat_lk);
                }
                work->next = search_q;  /* could be null */
                search_q = work;
                cond_signal(&search_q_cv);
                mutex_unlock(&search_q_lk);
                DP(DLEVEL2,("Sent work to search pool thread\n"));
            }
            else {
                mutex_unlock(&search_q_lk);
                err = thr_create(NULL,0,search_thr,(void *)work,
                                 THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                                 ,&tid);
                thr_setprio(tid,0);  /* set search to low */
                DP(DLEVEL2,("Sent work to new search thread\n"));
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_search++;
                    mutex_unlock(&stat_lk);
                }
            }
            break;
        default:
            fprintf(stderr,"tgrep: Internal error, work_t->tp no valid\n");
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Cound not create new thread!\n");
            exit(1);
        }
    }
```

```
 OUT:
    if (flags & TG_PROGRESS) {
        if (progress)
            fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads ar parked */
    if (flags & FC_COUNT) {
        mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}


/*
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;

    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;

    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        mutex_unlock(&stat_lk);
    }
    mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
```

```
 */
void *
search_thr(void *arg) /* work_t *arg */
{
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
    long        byte_count;
    long        next_line;
    int         show_line;  /* for the -v option */
    register int slen,plen,i;
    out_t       *out = NULL;    /* this threads output list */

    thr_setprio(thr_self(),0);  /* set search to low */
    thr_yield();
    wt = (work_t *)arg; /* first pass, wt is passed to use. */

    /* len = strlen(string);*/  /* only set on first pass */

    while (1) {  /* reuse the search threads */
        /* init all back to zero */
        line_count = 0;
        byte_count = 0l;
        next_line = 0l;
        show_line = 0;

        mutex_lock(&running_lk);
        running++;
        mutex_unlock(&running_lk);
        mutex_lock(&work_q_lk);
        tglimit--;
        mutex_unlock(&work_q_lk);
        DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));

        if ((fin = fopen(wt->path,"r")) == NULL) {
            if (!(flags & FS_NOERROR)) {
                fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",
                        strerror(errno),wt->path);
            }
            goto ERROR;
        }
        setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */
        DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));
        while ((fgets(rline,127,fin)) != NULL) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_line_search++;
                mutex_unlock(&stat_lk);
            }
            slen = strlen(rline);
            next_line += slen;
            line_count++;
            if (rline[slen-1] == '\n')
                rline[slen-1] = '\0';
            /*
            ** If the uncase flag is set, copy the read in line (rline)
```

```
        ** To the uncase line (cline) Set the line pointer to point at
        ** cline.
        ** If the case flag is NOT set, then point line at rline.
        ** line is what is compared, rline is waht is printed on a
        ** match.
        */
        if (flags & FI_IGNCASE) {
            strcpy(cline,rline);
            uncase(cline);
            line = cline;
        }
        else {
            line = rline;
        }
        show_line = 1;  /* assume no match, if -v set */
        /* The old code removed */
        if (use_pmatch) {
            for (i=0; i<regexp_cnt; i++) {
                if (pmatch(pm_pat[i], line, &pm_len)) {
                    if (!(flags & FV_REVERSE)) {
                        add_output_local(&out,wt,line_count,
                                         byte_count,rline);
                        continue_line(rline,fin,out,wt,
                                      &line_count,&byte_count);
                    }
                    else {
                        show_line = 0;
                    } /* end of if -v flag if / else block */
                    /*
                    ** if we get here on ANY of the regexp targets
                    ** jump out of the loop, we found a single
                    ** match so, do not keep looking!
                    ** If name only, do not keep searcthing the same
                    ** file, we found a single match, so close the file,
                    ** print the file name and move on to the next file.
                    */
                    if (flags & FL_NAMEONLY)
                        goto OUT_OF_LOOP;
                    else
                        goto OUT_AND_DONE;
                } /* end found a match if block */
            } /* end of the for pat[s] loop */
        }
        else {
            if (bm_pmatch( bm_pat, line)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,byte_count,rline);
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
                    show_line = 0;
                }
                if (flags & FL_NAMEONLY)
                    goto OUT_OF_LOOP;
            }
        }
      OUT_AND_DONE:
        if ((flags & FV_REVERSE) && show_line) {
            add_output_local(&out,wt,line_count,byte_count,rline);
            show_line = 0;
        }
        byte_count = next_line;
```

```
        }
      OUT_OF_LOOP:
        fclose(fin);
        /*
        ** The search part is done, but before we give back the FD,
        ** and park this thread in the search thread pool, print the
        ** local output we have gathered.
        */
        print_local_output(out,wt);  /* this also frees out nodes */
        out = NULL; /* for the next time around, if there is one */
    ERROR:
        DP(DLEVEL5,("Search done for %s\n",wt->path));
        free(wt->path);
        free(wt);

        notrun();
        mutex_lock(&search_q_lk);
        if (search_pool_cnt > search_thr_limit) {
            mutex_unlock(&search_q_lk);
            DP(DLEVEL5,("Search thread exiting\n"));
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_destroy++;
                mutex_unlock(&stat_lk);
            }
            return(0);
        }
        else {
            search_pool_cnt++;
            while (!search_q)
                cond_wait(&search_q_cv,&search_q_lk);
            search_pool_cnt--;
            wt = search_q;  /* we have work to do! */
            if (search_q->next)
                search_q = search_q->next;
            else
                search_q = NULL;
            mutex_unlock(&search_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Continue line: Speacial case search with the -C flag set. If you are
 * searching files like Makefiles, some lines may have escape char's to
 * contine the line on the next line. So the target string can be found, but
 * no data is displayed. This function continues to print the escaped line
 * until there are no more "\" chars found.
 */
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
              int *lc, long *bc)
{
    int len;
    int cnt = 0;
    char *line;
    char nline[128];

    if (!(flags & FC_LINE))
        return(0);

    line = rline;
```

```
  AGAIN:
    len = strlen(line);
    if (line[len-1] == '\\') {
        if ((fgets(nline,127,fin)) == NULL) {
            return(cnt);
        }
        line = nline;
        len = strlen(line);
        if (line[len-1] == '\n')
            line[len-1] = '\0';
        *bc = *bc + len;
        *lc++;
        add_output_local(&out,wt,*lc,*bc,line);
        cnt++;
        goto AGAIN;
    }
    return(cnt);
}

/*
 * cascade: This thread is started by the main thread when directory names
 * are found on the work Q. The thread reads all the new file, and directory
 * names from the directory it was started when and adds the names to the
 * work Q. (it finds more work!)
 */
void *
cascade(void *arg)  /* work_t *arg */
{
    char        fullpath[1025];
    int         restart_cnt = 10;
    DIR         *dp;

    char        dir_buf[sizeof(struct dirent) + PATH_MAX];
    struct dirent *dent = (struct dirent *)dir_buf;
    struct stat   sbuf;
    char        *fpath;
    work_t      *wt;
    int         fl = 0, dl = 0;
    int         pm_file_len = 0;

    thr_setprio(thr_self(),64);  /* set search to middle */
    thr_yield();  /* try toi give control back to main thread */
    wt = (work_t *)arg;

    while(1) {
        fl = 0;
        dl = 0;
        restart_cnt = 10;
        pm_file_len = 0;

        mutex_lock(&running_lk);
        running++;
        mutex_unlock(&running_lk);
        mutex_lock(&work_q_lk);
        tglimit--;
        mutex_unlock(&work_q_lk);

        if (!wt) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
            goto DONE;
        }
        fpath = (char *)wt->path;
```

```
        if (!fpath) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
            goto DONE;
        }
        DP(DLEVEL3,("Cascading on %s\n",fpath));
        if (( dp = opendir(fpath)) == NULL) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't open dir %s, %s. Ignored.\n",
                        fpath,strerror(errno));
            goto DONE;
        }
        while ((readdir_r(dp,dent)) != NULL) {
            restart_cnt = 10;  /* only try to restart the interupted 10 X */

            if (dent->d_name[0] == '.') {
                if (dent->d_name[1] == '.' && dent->d_name[2] == '\0')
                    continue;
                if (dent->d_name[1] == '\0')
                    continue;
            }

            fl = strlen(fpath);
            dl = strlen(dent->d_name);
            if ((fl + 1 + dl) > 1024) {
                fprintf(stderr,"tgrep: Path %s/%s is too long. "
                        "MaxPath = 1024\n",
                        fpath, dent->d_name);
                continue;  /* try the next name in this directory */
            }
            strcpy(fullpath,fpath);
            strcat(fullpath,"/");
            strcat(fullpath,dent->d_name);

          RESTART_STAT:
            if (stat(fullpath,&sbuf)) {
                if (errno == EINTR) {
                    if (--restart_cnt)
                        goto RESTART_STAT;
                }
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can't stat file/dir %s, %s. "
                            "Ignored.\n",
                            fullpath,strerror(errno));
                goto ERROR;
            }

            switch (sbuf.st_mode & S_IFMT) {
            case S_IFREG :
                if (flags & TG_FILEPAT) {
                    if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
                        DP(DLEVEL3,("file pat match (cascade) %s\n",
                                    dent->d_name));
                        add_work(fullpath,FILET);
                    }
                }
                else {
                    add_work(fullpath,FILET);
                    DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
                                fullpath));
                }
                break;
            case S_IFDIR :
```

```
                DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
                add_work(fullpath,DIRT);
                break;
            }
        }

      ERROR:
        closedir(dp);
      DONE:
        free(wt->path);
        free(wt);
        notrun();
        mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt > cascade_thr_limit) {
            mutex_unlock(&cascade_q_lk);
            DP(DLEVEL5,("Cascade thread exiting\n"));
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_cascade_destroy++;
                mutex_unlock(&stat_lk);
            }
            return(0); /* thr_exit */
        }
        else {
            DP(DLEVEL5,("Cascade thread waiting in pool\n"));
            cascade_pool_cnt++;
            while (!cascade_q)
                cond_wait(&cascade_q_cv,&cascade_q_lk);
            cascade_pool_cnt--;
            wt = cascade_q;  /* we have work to do! */
            if (cascade_q->next)
                cascade_q = cascade_q->next;
            else
                cascade_q = NULL;
            mutex_unlock(&cascade_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searching
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;
    int         print_name = 1;

    pp = out;
    mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
        if (progress >= progress_offset) {
            progress = 0;
            fprintf(stderr,".");
        }
    }
    while (pp) {
```

```
        out_count++;
        if (!(flags & FC_COUNT)) {
            if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
                if (!printed) {
                    printed = 1;
                    printf("%s\n",wt->path);
                }
            }
            else {  /* We are printing more then just the name */
                if (!(flags & FH_HOLDNAME))  /* do not print name ? */
                    printf("%s :",wt->path);
                if (flags & FB_BLOCK)
                    printf("%ld:",pp->byte_count/512+1);
                if (flags & FN_NUMBER)
                    printf("%d:",pp->line_count);
                printf("%s\n",pp->line);
            }
        }
        op = pp;
        pp = pp->next;
        /* free the nodes as we go down the list */
        free(op->line);
        free(op);
    }
    mutex_unlock(&output_print_lk);
    mutex_lock(&global_count_lk);
    global_count += out_count;
    mutex_unlock(&global_count_lk);
    return(0);
}

/*
 * add output local: is called by a search thread as it finds matching lines.
 * the matching line, it's byte offset, line count, etc are stored until the
 * search thread is done searching the file, then the lines are printed as
 * a group. This way the lines from more then a single file are not mixed
 * together.
 */
int
add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
{
    out_t       *ot,*oo, *op;

    if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
        goto ERROR;
    if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
        goto ERROR;

    strcpy(ot->line,line);
    ot->line_count = lc;
    ot->byte_count = bc;

    if (!*out) {
        *out = ot;
        ot->next = NULL;
        return(0);
    }
    /* append to the END of the list, keep things sorted! */
    op = oo = *out;
    while(oo) {
        op = oo;
        oo = oo->next;
    }
```

```
    op->next = ot;
    ot->next = NULL;
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Output lost. No space. "
                "[%s: line %d byte %d match : %s\n",
                wt->path,lc,bc,line);
    return(1);
}

/*
 * print stats: If the -S flag is set, after ALL files have been searched,
 * main thread calls this function to print the stats it keeps on how the
 * search went.
 */
void
prnt_stats(void)
{
    float a,b,c;
    float t = 0.0;
    time_t  st_end = 0;
    char    tl[80];

    st_end = time(NULL); /* stop the clock */
    fprintf(stderr,"\n---------------- Tgrep Stats. -------------------\n");
    fprintf(stderr,"Number of directories searched:           %d\n",
            st_dir_search);
    fprintf(stderr,"Number of files searched:                 %d\n",
            st_file_search);
    c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start);
    fprintf(stderr,"Dir/files per second:                     %3.2f\n",
            c);
    fprintf(stderr,"Number of lines searched:                 %d\n",
            st_line_search);
    fprintf(stderr,"Number of matching lines to target:       %d\n",
            global_count);

    fprintf(stderr,"Number of cascade threads created:        %d\n",
            st_cascade);
    fprintf(stderr,"Number of cascade threads from pool:      %d\n",
            st_cascade_pool);
    a = st_cascade_pool; b = st_dir_search;
    fprintf(stderr,"Cascade thread pool hit rate:             %3.2f%%\n",
            ((a/b)*100));
    fprintf(stderr,"Cascade pool overall size:                %d\n",
            cascade_pool_cnt);
    fprintf(stderr,"Cascade pool size limit:                  %d\n",
            cascade_thr_limit);
    fprintf(stderr,"Number of cascade threads destroyed:      %d\n",
            st_cascade_destroy);

    fprintf(stderr,"Number of search threads created:         %d\n",
            st_search);
    fprintf(stderr,"Number of search threads from pool:       %d\n",
            st_pool);
    a = st_pool; b = st_file_search;
    fprintf(stderr,"Search thread pool hit rate:              %3.2f%%\n",
            ((a/b)*100));
    fprintf(stderr,"Search pool overall size:                 %d\n",
            search_pool_cnt);
    fprintf(stderr,"Search pool size limit:                   %d\n",
            search_thr_limit);
```

```
    fprintf(stderr,"Number of search threads destroyed:       %d\n",
            st_destroy);

    fprintf(stderr,"Max # of threads running concurrenly:     %d\n",
            st_maxrun);
    fprintf(stderr,"Total run time, in seconds.               %d\n",
           (st_end - st_start));

    /* Why did we wait ? */
    a = st_workfds; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to no FD's:  (%.3d)       %d Times, %3.2f%%\n",
           search_thr_limit,st_workfds,c);
    a = st_worknull; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to no work on Q:         %d Times, %3.2f%%\n",
           st_worknull,c);
#ifndef __lock_lint  /* it is OK to read HERE with out the lock ! */
    if (tglimit == UNLIMITED)
        strcpy(tl,"Unlimited");
    else
        sprintf(tl,"   %.3d   ",tglimit);
#endif
    a = st_worklimit; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n",
           tl,st_worklimit,c);
    fprintf(stderr,"Work continued to be handed out:          %3.2f%%\n",
            100.00-t);
    fprintf(stderr,"----------------------------------------------------\n");
}

/*
 * not running: A glue function to track if any search threads or cascade
 * threads are running. When the count is zero, and the work Q is NULL,
 * we can safly say, WE ARE DONE.
 */
void
notrun (void)
{
    mutex_lock(&work_q_lk);
    work_cnt--;
    tglimit++;
    current_open_files++;
    mutex_lock(&running_lk);
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (running > st_maxrun) {
            st_maxrun = running;
            DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
        }
        mutex_unlock(&stat_lk);
    }
    running--;
    if (work_cnt == 0 && running == 0) {
        all_done = 1;
        DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
    }
    mutex_unlock(&running_lk);
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
}
```

```
/*
 * uncase: A glue function. If the -i (case insensitive) flag is set, the
 * target strng and the read in line is converted to lower case before
 * comparing them.
 */
void
uncase(char *s)
{
    char        *p;

    for (p = s; *p != NULL; p++)
        *p = (char)tolower(*p);
}


/*
 * SigThread: if the -S option is set, the first ^C set to tgrep will
 * print the stats on the fly, the second will kill the process.
 */

void *
SigThread(void *arg)
{
    int sig;
    int stats_printed = 0;

    while (1) {
        sig = sigwait(&set);
        DP(DLEVEL7,("Signal %d caught\n",sig));
        switch (sig) {
        case -1:
            fprintf(stderr,"Signal error\n");
            break;
        case SIGINT:
            if (stats_printed)
                exit(1);
            stats_printed = 1;
            sig_print_stats();
            break;
        case SIGHUP:
            sig_print_stats();
            break;
        default:
            DP(DLEVEL7,("Default action taken (exit) for signal %d\n",sig));
            exit(1);  /* default action */
        }
    }
}

void
sig_print_stats(void)
{
    /*
    ** Get the output lock first
    ** Then get the stat lock.
    */
    mutex_lock(&output_print_lk);
    mutex_lock(&stat_lk);
    prnt_stats();
    mutex_unlock(&stat_lk);
    mutex_unlock(&output_print_lk);
    return;
}
```

```
/*
 * usage: Have to have one of these.
 */
void
usage(void)
{
    fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Where:\n");
#ifdef DEBUG
    fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\n");
    fprintf(stderr,"Debug     -f = block fd's from use (-f #)\n");
#endif
    fprintf(stderr,"          -b = show block count (512 byte block)\n");
    fprintf(stderr,"          -c = print only a line count\n");
    fprintf(stderr,"          -h = do not print file names\n");
    fprintf(stderr,"          -i = case insensitive\n");
    fprintf(stderr,"          -l = print file name only\n");
    fprintf(stderr,"          -n = print the line number with the line\n");
    fprintf(stderr,"          -s = Suppress error messages\n");
    fprintf(stderr,"          -v = print all but matching lines\n");
#ifdef NOT_IMP
    fprintf(stderr,"          -w = search for a \"word\"\n");
#endif
    fprintf(stderr,"          -r = Do not search for files in all "
                                "sub-directories\n");
    fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
    fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n");
    fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stderr\n"
                   "               for each file it finds, -P 10 prints a DOT\n"
                   "               on stderr for each 10 files it finds, etc...\n");
    fprintf(stderr,"          -e = expression search.(regexp) More then one\n");
    fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n");
    fprintf(stderr,"          -S = Print thread stats when done.\n");
    fprintf(stderr,"          -Z = Print help on the regexp used.\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Notes:\n");
    fprintf(stderr,"      If you start tgrep with only a directory name\n");
    fprintf(stderr,"      and no file names, you must not have the -r option\n");
    fprintf(stderr,"      set or you will get no output.\n");
    fprintf(stderr,"      To search stdin (piped input), you must set -r\n");
    fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
    fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n");
    fprintf(stderr,"      if you supply a directory name.\n");
    fprintf(stderr,"      If you do not supply a file, or directory name,\n");
    fprintf(stderr,"      and the -r option is not set, the current \n");
    fprintf(stderr,"      directory \".\" will be used.\n");
    fprintf(stderr,"      All the other options should work \"like\" grep\n");
    fprintf(stderr,"      The -p patten is regexp, tgrep will search only\n");
    fprintf(stderr,"      the file names that match the patten\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"      Tgrep Version %s\n",Tgrep_Version);
    fprintf(stderr,"\n");
    fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
    fprintf(stderr,"\n");
    exit(0);
}

/*
 * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
 */
int
```

```
regexp_usage (void)
{
    fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
            "<{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"metachars:\n");
    fprintf(stderr,"    . - match any character\n");
    fprintf(stderr,"    * - match 0 or more occurrences of pervious char\n");
    fprintf(stderr,"    + - match 1 or more occurrences of pervious char.\n");
    fprintf(stderr,"    ^ - match at begining of string\n");
    fprintf(stderr,"    $ - match end of string\n");
    fprintf(stderr,"    [ - start of character class\n");
    fprintf(stderr,"    ] - end of character class\n");
    fprintf(stderr,"    ( - start of a new pattern\n");
    fprintf(stderr,"    ) - end of a new pattern\n");
    fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
    fprintf(stderr,"    | - match either pattern\n");
    fprintf(stderr,"    \\ - escape any special characters\n");
    fprintf(stderr,"    \\c - escape any special characters\n");
    fprintf(stderr,"    \\o - turn on any special characters\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"To match two diffrerent patterns in the same command\n");
    fprintf(stderr,"Use the or function. \n"
            "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
            "This will match any line with \"pat1\" or \"pat2\" in it.\n");
    fprintf(stderr,"You can also use up to %d -e expresions\n",MAXREGEXP);
    fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley\n");
    exit(0);
}

/*
 * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
 * how to get tgrep to print debug info on different threads.
 */
#ifdef DEBUG
void
debug_usage(void)
{
    int i = 0;

    fprintf(stderr,"DEBUG usage and levels:\n");
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"Level                   code\n");
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"0                 This message.\n");
    for (i=0; i<9; i++) {
        fprintf(stderr,"%d                 %s\n",i+1,debug_set[i].name);
    }
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"You can or the levels together like -d134 for levels\n");
    fprintf(stderr,"1 and 3 and 4.\n");
    fprintf(stderr,"\n");
    exit(0);
}
#endif
```

Multithreaded Quicksort

The following example `tquick.c` implements the quicksort algorithm using threads.

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
*  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
*  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any obligation on the
*  part of SunSoft, Inc. to assist in its use, correction, modification or
*  enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
*  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
*  FILE OR ANY PART THEREOF.
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
*  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
*  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
*  DAMAGES.
*
*  SunSoft, Inc.
*  2550 Garcia Avenue
*  Mountain View, California  94043
*/

/*
 * multiple-thread quick-sort.  See man page for qsort(3c) for info.
 * Works fine on uniprocessor machines as well.
 *
 *  Written by:  Richard Pettit (Richard.Pettit@West.Sun.COM)
 */

#include <unistd.h>
#include <stdlib.h>
#include <thread.h>

/* don't create more threads for less than this */
#define SLICE_THRESH   4096

/* how many threads per lwp */
#define THR_PER_LWP       4

/* cast the void to a one byte quanitity and compute the offset */
#define SUB(a, n)      ((void *) (((unsigned char *) (a)) + ((n) * width)))

typedef struct {
  void    *sa_base;
  int      sa_nel;
  size_t   sa_width;
  int    (*sa_compar)(const void *, const void *);
```

```
} sort_args_t;

/* for all instances of quicksort */
static int threads_avail;

#define SWAP(a, i, j, width) \
{ \
  int n; \
  unsigned char uc; \
  unsigned short us; \
  unsigned long ul; \
  unsigned long long ull; \
 \
  if (SUB(a, i) == pivot) \
    pivot = SUB(a, j); \
  else if (SUB(a, j) == pivot) \
    pivot = SUB(a, i); \
 \
  /* one of the more convoluted swaps I've done */ \
  switch(width) { \
  case 1: \
    uc = *((unsigned char *) SUB(a, i)); \
    *((unsigned char *) SUB(a, i)) = *((unsigned char *) SUB(a, j)); \
    *((unsigned char *) SUB(a, j)) = uc; \
    break; \
  case 2: \
    us = *((unsigned short *) SUB(a, i)); \
    *((unsigned short *) SUB(a, i)) = *((unsigned short *) SUB(a, j)); \
    *((unsigned short *) SUB(a, j)) = us; \
    break; \
  case 4: \
    ul = *((unsigned long *) SUB(a, i)); \
    *((unsigned long *) SUB(a, i)) = *((unsigned long *) SUB(a, j)); \
    *((unsigned long *) SUB(a, j)) = ul; \
    break; \
  case 8: \
    ull = *((unsigned long long *) SUB(a, i)); \
    *((unsigned long long *) SUB(a,i)) = *((unsigned long long *) SUB(a,j)); \
    *((unsigned long long *) SUB(a, j)) = ull; \
    break; \
  default: \
    for(n=0; n<width; n++) { \
      uc = ((unsigned char *) SUB(a, i))[n]; \
      ((unsigned char *) SUB(a, i))[n] = ((unsigned char *) SUB(a, j))[n]; \
      ((unsigned char *) SUB(a, j))[n] = uc; \
    } \
    break; \
  } \
}

static void *
_quicksort(void *arg)
{
  sort_args_t *sargs = (sort_args_t *) arg;
  register void *a = sargs->sa_base;
  int n = sargs->sa_nel;
  int width = sargs->sa_width;
  int (*compar)(const void *, const void *) = sargs->sa_compar;
  register int i;
  register int j;
  int z;
  int thread_count = 0;
  void *t;
```

```
void *b[3];
void *pivot = 0;
sort_args_t sort_args[2];
thread_t tid;

/* find the pivot point */
switch(n) {
case 0:
case 1:
  return 0;
case 2:
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  return 0;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
    SWAP(a, 2, 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
    SWAP(a, 1, 0, width);
  }
  return 0;
default:
  if (n > 3) {
    b[0] = SUB(a, 0);
    b[1] = SUB(a, n / 2);
    b[2] = SUB(a, n - 1);
    /* three sort */
    if ((*compar)(b[0], b[1]) > 0) {
      t = b[0];
      b[0] = b[1];
      b[1] = t;
    }
    /* the first two are now ordered, now order the second two */
    if ((*compar)(b[2], b[1]) < 0) {
      t = b[1];
      b[1] = b[2];
      b[2] = t;
    }
    /* should the second be moved to the first? */
    if ((*compar)(b[1], b[0]) < 0) {
      t = b[0];
      b[0] = b[1];
      b[1] = t;
    }
    if ((*compar)(b[0], b[2]) != 0)
      if ((*compar)(b[0], b[1]) < 0)
        pivot = b[1];
      else
        pivot = b[2];
  }
  break;
}
if (pivot == 0)
  for(i=1; i<n; i++) {
    if (z = (*compar)(SUB(a, 0), SUB(a, i))) {
```

```
            pivot = (z > 0) ? SUB(a, 0) : SUB(a, i);
            break;
          }
        }
      if (pivot == 0)
        return;

      /* sort */
      i = 0;
      j = n - 1;
      while(i <= j) {
        while((*compar)(SUB(a, i), pivot) < 0)
          ++i;
        while((*compar)(SUB(a, j), pivot) >= 0)
          --j;
        if (i < j) {
          SWAP(a, i, j, width);
          ++i;
          --j;
        }
      }

      /* sort the sides judiciously */
      switch(i) {
      case 0:
      case 1:
        break;
      case 2:
        if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
          SWAP(a, 0, 1, width);
        }
        break;
      case 3:
        /* three sort */
        if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
          SWAP(a, 0, 1, width);
        }
        /* the first two are now ordered, now order the second two */
        if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
          SWAP(a, 2, 1, width);
        }
        /* should the second be moved to the first? */
        if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
          SWAP(a, 1, 0, width);
        }
        break;
      default:
        sort_args[0].sa_base          = a;
        sort_args[0].sa_nel           = i;
        sort_args[0].sa_width         = width;
        sort_args[0].sa_compar        = compar;
        if ((threads_avail > 0) && (i > SLICE_THRESH)) {
          threads_avail--;
          thr_create(0, 0, _quicksort, &sort_args[0], 0, &tid);
          thread_count = 1;
        } else
          _quicksort(&sort_args[0]);
        break;
      }
      j = n - i;
      switch(j) {
      case 1:
        break;
```

```
  case 2:
    if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
      SWAP(a, i, i + 1, width);
    }
    break;
  case 3:
    /* three sort */
    if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
      SWAP(a, i, i + 1, width);
    }
    /* the first two are now ordered, now order the second two */
    if ((*compar)(SUB(a, i + 2), SUB(a, i + 1)) < 0) {
      SWAP(a, i + 2, i + 1, width);
    }
    /* should the second be moved to the first? */
    if ((*compar)(SUB(a, i + 1), SUB(a, i)) < 0) {
      SWAP(a, i + 1, i, width);
    }
    break;
  default:
    sort_args[1].sa_base          = SUB(a, i);
    sort_args[1].sa_nel           = j;
    sort_args[1].sa_width         = width;
    sort_args[1].sa_compar        = compar;
    if ((thread_count == 0) && (threads_avail > 0) && (i > SLICE_THRESH)) {
      threads_avail--;
      thr_create(0, 0, _quicksort, &sort_args[1], 0, &tid);
      thread_count = 1;
    } else
      _quicksort(&sort_args[1]);
    break;
  }
  if (thread_count) {
    thr_join(tid, 0, 0);
    threads_avail++;
  }
  return 0;
}

void
quicksort(void *a, size_t n, size_t width,
          int (*compar)(const void *, const void *))
{
  static int ncpus = -1;
  sort_args_t sort_args;

  if (ncpus == -1) {
    ncpus = sysconf( _SC_NPROCESSORS_ONLN);

    /* lwp for each cpu */
    if ((ncpus > 1) && (thr_getconcurrency() < ncpus))
      thr_setconcurrency(ncpus);

    /* thread count not to exceed THR_PER_LWP per lwp */
    threads_avail = (ncpus == 1) ? 0 : (ncpus * THR_PER_LWP);
  }
  sort_args.sa_base = a;
  sort_args.sa_nel = n;
  sort_args.sa_width = width;
  sort_args.sa_compar = compar;
  (void) _quicksort(&sort_args);
}
```

Dave Marshall
1/5/1999

Subsections

Remote Procedure Calls (RPC)

This chapter provides an overview of Remote Procedure Calls (RPC) RPC.

What Is RPC

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler (Chapter 33) clients transparently make remote calls through a local procedure interface.

How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 32.1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

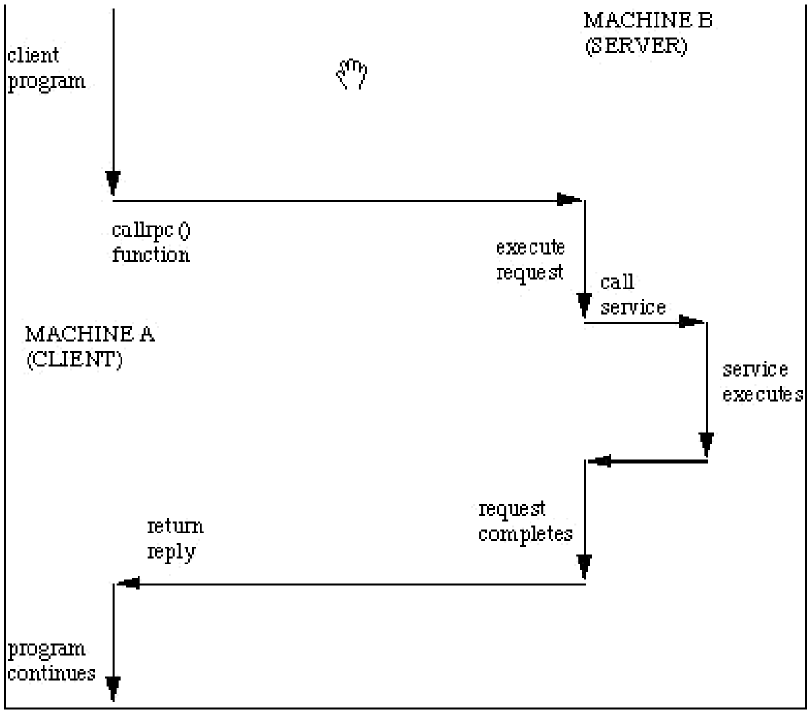


Fig. 32.1 Remote Procedure Calling Mechanism A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a a number of procedures that can be called remotely. Each procedure has a procedure number.

RPC Application Development

Consider an example:

A client/server lookup in a personal database on a remote machine. Assuming that we cannot access the database from the local machine (via NFS).

We use UNIX to run a remote shell and execute the command this way. There are some problems with this method:

- the command may be slow to execute.
- You require an login account on the remote machine.

The RPC alternative is to

- establish an server on the remote machine that can repond to queries.
- Retrieve information by calling a query which will be quicker than previous approach.

To develop an RPC application the following steps are needed:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program

The programs will be compiled seperately. The communication protocol is achieved by generated stubs and these stubs and rpc (and other libraries) will need to be linked in.

Defining the Protocol

The easiest way to define and generate the protocol is to use a protocol complier such as `rpcgen` which we discuss is Chapter 33.

For the protocol you must identify the name of the service procedures, and data types of parameters and return arguments.

The protocol compiler reads a definitio and automatically generates client and server stubs.

`rpcgen` uses its own language (RPC language or RPCL) which looks very similar to preprocessor directives.

`rpcgen` exists as a standalone executable compiler that reads special files denoted by a `.x` prefix.

So to compile a RPCL file you simply do

```
rpcgen rpcprog.x
```

This will generate possibly four files:

- `rpcprog_clnt.c` -- the client stub
- `rpcprog_svc.c` -- the server stub
- `rpcprog_xdr.c` -- If necessary XDR (external data representation) filters
- `rpcprog.h` -- the header file needed for any XDR filters.

The external data representation (XDR) is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.

Defining Client and Server Application Code

We must now write the the client and application code. They must communicate via procedures and data types specified in the Protocol.

The service side will have to register the procedures that may be called by the client and receive and return any data required for processing.

The client application call the remote procedure pass any required data and will receive the retruned data.

There are several levels of application interfaces that may be used to develop RPC applications. We will briefly disuss these below before exapnading thhe most common of these in later chapters.

Compliling and running the application

Let us consider the full compilation model required to run a RPC application. Makefiles are useful for easing the burden of compiling RPC applications but it is necessary to understand the complete model before one can assemble a convenient makefile.

Assume the the client program is called `rpcprog.c`, the service program is `rpcsvc.c` and that the protocol has been defined in `rpcprog.x` and that `rpcgen` has been used to produce the stub and filter files: `rpcprog_clnt.c, rpcprog_svc.c, rpcprog_xdr.c, rpcprog.h`.

The client and server program must include (`#include "rpcprog.h"`

You must then:

- compile the client code:

  ```
  cc -c  rpcprog.c
  ```
- compile the client stub:

  ```
  cc -c  rpcprog_clnt.c
  ```
- compile the XDR filter:

  ```
  cc -c  rpcprog_xdr.c
  ```
- build the client executable:

  ```
  cc -o rpcprog rpcprog.o rpcprog_clnt.o rpcprog_xdr.c
  ```
- compile the service procedures:

  ```
  cc -c  rpcsvc.c
  ```
- compile the server stub:

```
cc -c  rpcprog_svc.c
```

- build the server executable:

```
cc -o rpcsvc rpcsvc.o rpcprog_svc.o rpcprog_xdr.c
```

Now simply run the programs `rpcprog` and `rpcsvc` on the client and server respectively. The server procedures must be registered before the client can call them.

Overview of Interface Routines

RPC has multiple levels of application interface to its services. These levels provide different degrees of control balanced with different amounts of interface code to implement. In order of increasing control and complexity. This section gives a summary of the routines available at each level. Simplified Interface Routines

The simplified interfaces are used to make remote procedure calls to routines on other machines, and specify only the type of transport to use. The routines at this level are used for most applications. Descriptions and code samples can be found in the section, Simplified Interface @ 3-2.

Simplified Level Routine Function

`rpc_reg()` -- Registers a procedure as an RPC program on all transports of the specified type.

`rpc_call()` -- Remote calls the specified procedure on the specified remote host.

`rpc_broadcast()` -- Broadcasts a call message across all transports of the specified type. Standard Interface Routines The standard interfaces are divided into top level, intermediate level, expert level, and bottom level. These interfaces give a developer much greater control over communication parameters such as the transport being used, how long to wait beforeresponding to errors and retransmitting requests, and so on.

Top Level Routines

At the top level, the interface is still simple, but the program has to create a client handle before making a call or create a server handle before receiving calls. If you want the application to run on all transports, use this interface. Use of these routines and code samples can be found in Top Level Interface

`clnt_create()` -- Generic client creation. The program tells `clnt_create()` where the server is located and the type of transport to use.

`clnt_create_timed()` Similar to `clnt_create()` but lets the programmer specify the maximum time allowed for each type of transport tried during the creation attempt.

`svc_create()` -- Creates server handles for all transports of the specified type. The program tells `svc_create()` which dispatch function to use.

`clnt_call()` -- Client calls a procedure to send a request to the server.

Intermediate Level Routines

The intermediate level interface of RPC lets you control details. Programs written at these lower levels are more complicated but run more efficiently. The intermediate level enables you to specify the transport to use.

`clnt_tp_create()` -- Creates a client handle for the specified transport.

`clnt_tp_create_timed()` -- Similar to `clnt_tp_create()` but lets the programmer specify the maximum time allowed. `svc_tp_create()` Creates a server handle for the specified transport.

`clnt_call()` -- Client calls a procedure to send a request to the server.

Expert Level Routines

The expert level contains a larger set of routines with which to specify transport-related parameters. Use of these routines

`clnt_tli_create()` -- Creates a client handle for the specified transport.

`svc_tli_create()` -- Creates a server handle for the specified transport.

`rpcb_set()` -- Calls rpcbind to set a map between an RPC service and a network address.

`rpcb_unset()` -- Deletes a mapping set by `rpcb_set()`.

`rpcb_getaddr()` -- Calls rpcbind to get the transport addresses of specified RPC services.

`svc_reg()` -- Associates the specified program and version number pair with the specified dispatch routine.

`svc_unreg()` -- Deletes an association set by `svc_reg()`.

`clnt_call()` -- Client calls a procedure to send a request to the server.

Bottom Level Routines

The bottom level contains routines used for full control of transport options.

`clnt_dg_create()` -- Creates an RPC client handle for the specified remote program, using a connectionless transport.

`svc_dg_create()` -- Creates an RPC server handle, using a connectionless transport.

`clnt_vc_create()` -- Creates an RPC client handle for the specified remote program, using a connection-oriented transport.

`svc_vc_create()` -- Creates an RPC server handle, using a connection-oriented transport.

`clnt_call()` -- Client calls a procedure to send a request to the server.

The Programmer's Interface to RPC

This section addresses the C interface to RPC and describes how to write network applications using RPC. For a complete specification of the routines in the RPC library, see the `rpc` and related `man` pages.

Simplified Interface

The simplified interface is the easiest level to use because it does not require the use of any other RPC routines. It also limits control of the underlying communications mechanisms. Program development at this level can be rapid, and is directly supported by the `rpcgen` compiler. For most applications, rpcgen and its facilities are sufficient. Some RPC services are not available as C functions, but they are available as RPC programs. The simplified interface library routines provide direct access to the RPC facilities for programs that do not require fine levels of control.

Routines such as `rusers` are in the RPC services library `librpcsvc`. `rusers.c`, below, is a program that displays the number of users on a remote host. It calls the RPC library routine, `rusers`.

The `program.c` program listing:

```
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <stdio.h>

/*
* a program that calls the
* rusers() service
*/

main(int argc,char **argv)
```

```
{
int num;
if (argc != 2) {
   fprintf(stderr, "usage: %s hostname\n",
   argv[0]);
   exit(1);
   }

if ((num = rnusers(argv[1])) < 0) {
   fprintf(stderr, "error: rusers\n");
   exit(1);
  }

fprintf(stderr, "%d users on %s\n", num, argv[1] );
exit(0);
}
```

Compile the program with:

```
cc program.c -lrpcsvc -lnsl
```

The Client Side

There is just one function on the client side of the simplified interface `rpc_call()`.

It has nine parameters:

```
int
rpc_call (char *host /* Name of server host */,
    u_long prognum /* Server program number */,
    u_long versnum /* Server version number */,
    xdrproc_t inproc /* XDR filter to encode arg */,
    char *in /* Pointer to argument */,
    xdr_proc_t outproc /* Filter to decode result */,
    char *out /* Address to store result */,
    char *nettype /* For transport selection */);
```

This function calls the procedure specified by `prognum, versum,` and `procnum` on the host. The argument to be passed to the remote procedure is pointed to by the `in` parameter, and `inproc` is the XDR filter to encode this argument. The `out` parameter is an address where the result from the remote procedure is to be placed. `outproc` is an XDR filter which will decode the result and place it at this address.

The client blocks on `rpc_call()` until it receives a reply from the server. If the server accepts, it returns `RPC_SUCCESS` with the value of zero. It will return a non-zero value if the call was unsuccessful. This value can be cast to the type `clnt_stat`, an enumerated type defined in the RPC include files (<rpc/rpc.h>) and interpreted by the `clnt_sperrno()` function. This function returns a pointer to a standard RPC error message corresponding to the error code. In the example, all "visible" transports listed in `/etc/netconfig` are tried. Adjusting the number of retries requires use of the lower levels of the RPC library. Multiple arguments and results are handled by collecting them in structures.

The example changed to use the simplified interface, looks like

```
#include <stdio.h>
#include <utmp.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

/* a program that calls the RUSERSPROG
* RPC program
*/

main(int argc, char **argv)

{
   unsigned long nusers;
```

```
    enum clnt_stat cs;
    if (argc != 2) {
      fprintf(stderr, "usage: rusers hostname\n");
      exit(1);
     }

    if( cs = rpc_call(argv[1], RUSERSPROG,
           RUSERSVERS, RUSERSPROC_NUM, xdr_void,
           (char *)0, xdr_u_long, (char *)&nusers,
           "visible") != RPC_SUCCESS ) {
               clnt_perrno(cs);
               exit(1);
             }

    fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
    exit(0);
}
```

Since data types may be represented differently on different machines, `rpc_call()` needs both the type of, and a pointer to, the RPC argument (similarly for the result). For `RUSERSPROC_NUM`, the return value is an unsigned long, so the first return parameter of `rpc_call()` is `xdr_u_long` (which is for an unsigned long) and the second is `&nusers` (which points to unsigned long storage). Because `RUSERSPROC_NUM` has no argument, the XDR encoding function of `rpc_call()` is `xdr_void()` and its argument is `NULL`.

The Server Side

The server program using the simplified interface is very straightforward. It simply calls `rpc_reg()` to register the procedure to be called, and then it calls `svc_run()`, the RPC library's remote procedure dispatcher, to wait for requests to come in.

`rpc_reg()` has the following prototype:

```
rpc_reg(u_long prognum /* Server program number */,
        u_long versnum /* Server version number */,
        u_long procnum /* server procedure number */,
        char *procname /* Name of remote function */,
        xdrproc_t inproc /* Filter to encode arg */,
        xdrproc_t outproc /* Filter to decode result */,
        char *nettype /* For transport selection */);
```

`svc_run()` invokes service procedures in response to RPC call messages. The dispatcher in `rpc_reg()` takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered. Some notes about the server program:

- Most RPC applications follow the naming convention of appending a `_1` to the function name. The sequence `_n` is added to the procedure names to indicate the version number `n` of the service.
- The argument and result are passed as addresses. This is true for all functions that are called remotely. If you pass `NULL` as a result of a function, then no reply is sent to the client. It is assumed that there is no reply to send.
- The result must exist in static data space because its value is accessed after the actual procedure has exited. The RPC library function that builds the RPC reply message accesses the result and sends the value back to the client.
- Only a single argument is allowed. If there are multiple elements of data, they should be wrapped inside a structure which can then be passed as a single entity.
- The procedure is registered for each transport of the specified type. If the type parameter is `(char *)NULL`, the procedure is registered for all transports specified in `NETPATH`.

You can sometimes implement faster or more compact code than can `rpcgen`. `rpcgen` handles the generic code-generation cases. The following program is an example of a hand-coded registration routine. It registers a single procedure and enters `svc_run()` to service requests.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

void *rusers();
```

```
main()
{
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
        RUSERSPROC_NUM, rusers,
        xdr_void, xdr_u_long,
        "visible") == -1) {
           fprintf(stderr, "Couldn't Register\n");
            exit(1);
          }
   svc_run(); /* Never returns */
  fprintf(stderr, "Error: svc_run returned!\n");
  exit(1);
}
```

`rpc_reg()` can be called as many times as is needed to register different programs, versions, and procedures.

Passing Arbitrary Data Types

Data types passed to and received from remote procedures can be any of a set of predefined types, or can be programmer-defined types. RPC handles arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a standard transfer format called external data representation (XDR) before sending them over the transport. The conversion from a machine representation to XDR is called serializing, and the reverse process is called deserializing. The translator arguments of `rpc_call()` and `rpc_reg()` can specify an XDR primitive procedure, like `xdr_u_long()`, or a programmer-supplied routine that processes a complete argument structure. Argument processing routines must take only two arguments: a pointer to the result and a pointer to the XDR handle.

The following XDR Primitive Routines are available:

```
xdr_int() xdr_netobj() xdr_u_long() xdr_enum()
xdr_long() xdr_float() xdr_u_int() xdr_bool()
xdr_short() xdr_double() xdr_u_short() xdr_wrapstring()
xdr_char() xdr_quadruple() xdr_u_char() xdr_void()
```

The nonprimitive `xdr_string()`, which takes more than two parameters, is called from `xdr_wrapstring()`.

For an example of a programmer-supplied routine, the structure:

```
struct simple {
   int a;
   short b;
  } simple;
```

contains the calling arguments of a procedure. The XDR routine `xdr_simple()` translates the argument structure as shown below:

```
#include <rpc/rpc.h>
#include "simple.h"

bool_t xdr_simple(XDR *xdrsp, struct simple *simplep)

{
   if (!xdr_int(xdrsp, &simplep->a))
      return (FALSE);
   if (!xdr_short(xdrsp, &simplep->b))
      return (FALSE);
   return (TRUE);
}
```

An equivalent routine can be generated automatically by `rpcgen` (See Chapter 33).

An XDR routine returns nonzero (a C TRUE) if it completes successfully, and zero otherwise.

For more complex data structures use the XDR prefabricated routines:

```
xdr_array() xdr_bytes() xdr_reference()
xdr_vector() xdr_union() xdr_pointer()
xdr_string() xdr_opaque()
```

For example, to send a variable-sized array of integers, it is packaged in a structure containing the array and its length:

```
struct varintarr {
int *data;
int arrlnth;
} arr;
```

Translate the array with `xdr_array()`, as shown below:

```
bool_t xdr_varintarr(XDR *xdrsp, struct varintarr *arrp)

{
    return(xdr_array(xdrsp, (caddr_t)&arrp->data,
             (u_int *)&arrp->arrlnth, MAXLEN, sizeof(int), xdr_int));
}
```

The arguments of `xdr_array()` are the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum array size, the size of each array element, and a pointer to the XDR routine to translate each array element. If the size of the array is known in advance, use `xdr_vector()` instread as is more efficient:

```
int intarr[SIZE];

bool_t xdr_intarr(XDR *xdrsp, int intarr[])
{
   return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int), xdr_int));
}
```

XDR converts quantities to 4-byte multiples when serializing. For arrays of characters, each character occupies 32 bits. `xdr_bytes()` packs characters. It has four parameters similar to the first four parameters of `xdr_array()`.

Null-terminated strings are translated by `xdr_string()`. It is like `xdr_bytes()` with no length parameter. On serializing it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

`xdr_reference()` calls the built-in functions xdr_string() and `xdr_reference()`, which translates pointers to pass a string, and struct simple from the previous examples. An example use of `xdr_reference()` is as follows:

```
struct finalexample {
    char *string;
    struct simple *simplep;
   } finalexample;

bool_t xdr_finalexample(XDR *xdrsp, struct finalexample *finalp)

{  if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (FALSE);
   if (!xdr_reference( xdrsp, &finalp->simplep, sizeof(struct simple), xdr_simple))
       return (FALSE);
    return (TRUE);
}
```

Note that`xdr_simple()` could have been called here instead of `xdr_reference()` .

Developing High Level RPC Applications

Let us now introduce some further functions and see how we develop an application using high level RPC routines. We will do this by studying an example.

We will develop a remote directory reading utility.

Let us first consider how we would write a local directory reader. We have seem how to do this already in Chapter 19.

Consider the program to consist of two files:

- `lls.c` -- the main program which calls a routine in a local module `read_dir.c`

```
/*
 * ls.c: local directory listing main - before RPC
 */
#include <stdio.h>
#include <strings.h>
#include "rls.h"

main (int argc, char **argv)

{
        char    dir[DIR_SIZE];

        /* call the local procedure */
        strcpy(dir, argv[1]);   /* char dir[DIR_SIZE] is coming and going... */
        read_dir(dir);

        /* spew-out the results and bail out of here! */
        printf("%s\n", dir);

        exit(0);
}
```

- `read_dir.c` -- the file containing the *local* routine `read_dir()`.

```
/* note - RPC compliant procedure calls take one input and
   return one output. Everything is passed by pointer.  Return
   values should point to static data, as it might have to
   survive some while. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>     /* use <xpg2include/sys/dirent.h> (SunOS4.1) or
        <sys/dirent.h> for X/Open Portability Guide, issue 2 conformance */
#include "rls.h"

read_dir(char    *dir)
   /* char dir[DIR_SIZE] */
{
        DIR * dirp;
        struct direct *d;
                      printf("beginning ");

        /* open directory */
        dirp = opendir(dir);
        if (dirp == NULL)
                return(NULL);

        /* stuff filenames into dir buffer */
        dir[0] = NULL;
        while (d = readdir(dirp))
                sprintf(dir, "%s%s\n", dir, d->d_name);

        /* return the result */
                  printf("returning ");
        closedir(dirp);
        return((int)dir);  /* this is the only new line from Example 4-3 */
}
```

- the header file `rls.h` contains only the following (for now at least)

```
#define DIR_SIZE 8192
```

Clearly we need to share the size between the files. Later when we develop RPC versions more information will need to be added to this file.

This local program would be compiled as follows:

```
cc lls.c read_dir.c -o lls
```

Now we want to modify this program to work over a network: Allowing us to inspect directories of a remote server accross a network.

The following steps will be required:

- We will have to convert the `read_dir.c`, to run on the server.
  - We will have to register the server and the routine `read_dir()` on the server/.
- The client `lls.c` will have to call the routine as a remote procedure.
- We will have to define the protocol for communication between the client and the server programs.

Defining the protocol

We can can use simple `NULL`-terminated strings for passing and receivong the directory name and directory contents. Furthermore, we can embed the passing of these parameters directly in the client and server code.

We therefore need to specify the program, procedure and version numbers for client and servers. This can be done automatically using `rpcgen` or relying on prdefined macros in the simlified interface. Here we will specify them manually.

The server and client must agree *ahead of time* what logical adresses thney will use (The physical addresses do not matter they are hidden from the application developer)

Program numbers are defined in a standard way:

- 0*x*00000000 - 0*x*1*FFFFFFF*: Defined by Sun
- 0*x*20000000 - 0*x*3*FFFFFFF*: User Defined
- 0*x*40000000 - 0*x*5*FFFFFFF*: Transient
- 0*x*60000000 - 0*xFFFFFFFF*: Reserved

We will simply choose a *user deifnined value* for our program number. The version and procedure numbers are set according to standard practice.

We still have the `DIR_SIZE` definition required from the local version as the size of the directory buffer is rewquired by bith client and server programs.

Our new `rls.h` file contains:

```
#define DIR_SIZE 8192
#define DIRPROG ((u_long) 0x20000001)   /* server program (suite) number */
#define DIRVERS ((u_long) 1)    /* program version number */
#define READDIR ((u_long) 1)    /* procedure number for look-up */
```

Sharing the data

We have mentioned previously that we can pass the data a simple strings. We need to define an XDR filter routine `xdr_dir()` that shares the data. Recall that only one encoding and decoding argument can be handled. This is easy and defined via the standard `xdr_string()` routine.

The XDR file, `rls_xrd.c`, is as follows:

```
#include <rpc/rpc.h>

#include "rls.h"

bool_t xdr_dir(XDR *xdrs, char *objp)
```

```
{ return ( xdr_string(xdrs, &objp, DIR_SIZE) ); }
```

The Server Side

We can use the original `read_dir.c` file. All we need to do is register the procedure and start the server.

The procedure is registered with `registerrpc()` function. This is prototypes by:

```
registerrpc(u_long prognum /* Server program number */,
        u_long versnum /* Server version number */,
        u_long procnum /* server procedure number */,
        char *procname /* Name of remote function */,
        xdrproc_t inproc /* Filter to encode arg */,
        xdrproc_t outproc /* Filter to decode result */);
```

The parameters a similarly defined as in the `rpc_reg` simplified interface function. We have already discussed the setting of the parametere with the protocol `rls.h` header files and the `rls_xrd.c` XDR filter file.

The `svc_run()` routine has also been discussed previously.

The full `rls_svc.c` code is as follows:

```
#include <rpc/rpc.h>
#include "rls.h"

main()
{
        extern bool_t xdr_dir();
        extern char * read_dir();

        registerrpc(DIRPROG, DIRVERS, READDIR,
                        read_dir, xdr_dir, xdr_dir);

        svc_run();
}
```

The Client Side

At the client side we simply need to call the remote procedure. The function `callrpc()` does this. It is prototyped as follows:

```
callrpc(char *host /* Name of server host */,
    u_long prognum /* Server program number */,
    u_long versnum /* Server version number */,
    char *in /* Pointer to argument */,
    xdrproc_t inproc /* XDR filter to encode arg */,
    char *out /* Address to store result */
    xdr_proc_t outproc /* Filter to decode result */);
```

We call a local function `read_dir()` which uses `callrpc()` to call the remote procedure that has been registered `READDIR` at the server.

The full `rls.c` program is as follows:

```
/*
 * rls.c: remote directory listing client
 */
#include <stdio.h>
#include <strings.h>
#include <rpc/rpc.h>
#include "rls.h"

main (argc, argv)
```

```
int argc; char *argv[];
{
char    dir[DIR_SIZE];

        /* call the remote procedure if registered */
        strcpy(dir, argv[2]);
        read_dir(argv[1], dir); /* read_dir(host, directory) */

        /* spew-out the results and bail out of here! */
        printf("%s\n", dir);

        exit(0);
}

read_dir(host, dir)
char   *dir, *host;
{
        extern bool_t xdr_dir();
        enum clnt_stat clnt_stat;

        clnt_stat = callrpc ( host, DIRPROG, DIRVERS, READDIR,
                        xdr_dir, dir, xdr_dir, dir);
        if (clnt_stat != 0) clnt_perrno (clnt_stat);
}
```

Exercise

Exercise 12833

Compile and run the remote directory example `rls.c` *etc*. Run both the client ande srever locally and if possible over a network.

Dave Marshall
1/5/1999

Subsections

Protocol Compiling and Lower Level RPC Programming

This chapter introduces the rpcgen tool and provides a tutorial with code examples and usage of the available compile-time flags. We also introduce some further RPC programming routines.

What is `rpcgen`

The `rpcgen` tool generates remote program interface modules. It compiles source code written in the RPC Language. RPC Language is similar in syntax and structure to C. `rpcgen` produces one or more C language source modules, which are then compiled by a C compiler.

The default output of rpcgen is:

- A header file of definitions common to the server and the client
- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

`rpcgen` can optionally generate (although we *do not* consider these issues here -- see `man` pages or receommended reading):

- Various transports
- A time-out for servers
- Server stubs that are MT safe
- Server stubs that are not main programs
- C-style arguments passing ANSI C-compliant code
- An RPC dispatch table that checks authorizations and invokes service routines

`rpcgen` significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the rpcgen output.

An `rpcgen` Tutorial

`rpcgen` provides programmers a simple and direct way to write distributed applications. Server procedures may be written in any language that observes procedure-calling conventions. They are linked with the server stub produced by rpcgen to form an executable server program. Client procedures are written and linked in the same way. This section presents some basic rpcgen programming examples. Refer also to the `man rpcgen` online manual page.

Converting Local Procedures to Remote Procedures

Assume that an application runs on a single computer and you want to convert it to run in a "distributed" manner on a network. This example shows the stepwise conversion of this program that writes a message to the system console.

Single Process Version of `printmesg.c`:

```
/* printmsg.c: print a message on the console */
#include <stdio.h>
main(int argc, char *argv[])

{
   char *message;
   if (argc != 2) {
      fprintf(stderr, "usage: %s <message>\n",argv[0]);
      exit(1);
    }
   message = argv[1];
   if (!printmessage(message)) {
     fprintf(stderr,"%s: couldn¹t print your message\n",argv[0]);
     exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}

/* Print a message to the console.
* Return a boolean indicating whether
* the message was actually printed. */

printmessage(char *msg)

{
   FILE *f;
   f = fopen("/dev/console", "w");
   if (f == (FILE *)NULL) {
     return (0);
    }
   fprintf(f, "%s\n", msg);
   fclose(f);
   return(1);
}
```

For local use on a single machine, this program could be compiled and executed as follows:

```
$ cc printmsg.c -o printmsg
```

```
$ printmsg "Hello, there."
Message delivered!
$
```

If the `printmessage()` function is turned into a *remote procedure*, it can be called from anywhere in the network. `rpcgen` makes it easy to do this:

First, determine the data types of all procedure-calling arguments and the result argument. The calling argument of `printmessage()` is a string, and the result is an integer. We can write a protocol specification in RPC language that describes the remote version of printmessage. The RPC language source code for such a specification is:

```
/* msg.x: Remote msg printing protocol */
program MESSAGEPROG {
   version PRINTMESSAGEVERS {
     int PRINTMESSAGE(string) = 1;
   } = 1;
} = 0x20000001;
```

Remote procedures are always declared as part of remote programs. The code above declares an entire remote program that contains the single procedure `PRINTMESSAGE`.

In this example,

- `PRINTMESSAGE` procedure is declared to be:
  - the `procedure 1`,
  - in version 1 of the remote program
- `MESSAGEPROG`, with the program number 0x20000001.

Version numbers are incremented when functionality is changed in the remote program. Existing procedures can be changed or new ones can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

Note: that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow. Note also that the argument type is string and not char * as it would be in C. This is because a char * in C is ambiguous. char usually means an array of characters, but it could also represent a pointer to a single character. In RPC language, a null-terminated array of char is called a string.

There are just two more programs to write:

- The remote procedure itself

  Th RPC Version of `printmsg.c`:

  ```
  /*
  * msg_proc.c: implementation of the
  * remote procedure "printmessage"
  */

  #include <stdio.h>
  #include "msg.h" /* msg.h generated by rpcgen */

  int * printmessage_1(char **msg, struct svc_req *req)

  {
     static int result; /* must be static! */
     FILE *f;

     f = fopen("/dev/console", "w");
     if (f == (FILE *)NULL) {
  ```

```
          result = 0;
          return (&result);
        }
      fprintf(f, "%s\n", *msg);
      fclose(f);
      result = 1;
      return (&result);
}
```

Note that the declaration of the remote procedure `printmessage_1` differs from that of the local procedure printmessage in four ways:

- It takes a pointer to the character array instead of the pointer itself. This is true of all remote procedures when the `'-' N` option is not used: They always take pointers to their arguments rather than the arguments themselves. Without the `'-' N` option, remote procedures are always called with a single argument. If more than one argument is required the arguments must be passed in a struct.
- It is called with two arguments. The second argument contains information on the context of an invocation: the program, version, and procedure numbers, raw and canonical credentials, and an `SVCXPRT` structure pointer (the `SVCXPRT` structure contains transport information). This information is made available in case the invoked procedure requires it to perform the request.
- It returns a pointer to an integer instead of the integer itself. This is also true of remote procedures when the `'-' N` option is not used: They return pointers to the result. The result should be declared static unless the `'-' M` (multithread) or `'-' A` (Auto mode) options are used. Ordinarily, if the result is declared local to the remote procedure, references to it by the server stub are invalid after the remote procedure returns. In the case of `'-' M` and `'-' A` options, a pointer to the result is passed as a third argument to the procedure, so the result is not declared in the procedure.
- An `_1` is appended to its name. In general, all remote procedures calls generated by rpcgen are named as follows: the procedure name in the program definition (here `PRINTMESSAGE`) is converted to all lowercase letters, an underbar (`_`) is appended to it, and the version number (here `1`) is appended. This naming scheme allows multiple versions of the same procedure.

- The main client program that calls it:

```
/*
* rprintmsg.c: remote version
* of "printmsg.c"
*/

#include <stdio.h>
#include "msg.h" /* msg.h generated by rpcgen */

main(int argc, char **argv)

{
  CLIENT *clnt;
  int *result;
  char *server;
  char *message;

  if (argc != 3) {
     fprintf(stderr, "usage: %s host
     message\n", argv[0]);
     exit(1);
    }
```

```
        server = argv[1];
        message = argv[2];

        /*
         * Create client "handle" used for
         * calling MESSAGEPROG on the server
         * designated on the command line.
         */

        clnt = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS, "visible");

        if (clnt == (CLIENT *)NULL) {
         /*
          * Couldn't establish connection
          * with server.
          * Print error message and die.
          */

         clnt_pcreateerror(server);
         exit(1);
         }

        /*
         * Call the remote procedure
         * "printmessage" on the server
         */

         result = printmessage_1(&message, clnt);
         if (result == (int *)NULL) {
           /*
            * An error occurred while calling
            * the server.
            * Print error message and die.
            */

           clnt_perror(clnt, server);
           exit(1);
          }

         /* Okay, we successfully called
          * the remote procedure.
          */

         if (*result == 0) {

         /*
          * Server was unable to print
          * our message.
          * Print error message and die.
          */

          fprintf(stderr, "%s: could not print your message\n",argv[0]);
          exit(1);
          }

         /* The message got printed on the
          * server's console
```

```
    */

    printf("Message delivered to %s\n", server);
    clnt_destroy( clnt );
    exit(0);
}
```

Note the following about Client Program to Call printmsg.c:

- First, a client handle is created by the RPC library routine `clnt_create()`. This client handle is passed to the stub routine that calls the remote procedure. If no more calls are to be made using the client handle, destroy it with a call to `clnt_destroy()` to conserve system resources.
- The last parameter to `clnt_create()` is visible, which specifies that any transport noted as visible in `/etc/netconfig` can be used.
- The remote `procedure printmessage_1` is called exactly the same way as it is declared in `msg_proc.c`, except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result.
- The remote procedure call can fail in two ways. The RPC mechanism can fail or there can be an error in the execution of the remote procedure. In the former case, the remote `procedure printmessage_1` returns a `NULL`. In the latter case, the error reporting is application dependent. Here, the error is returned through `*result`.

To compile the remote `rprintmsg` example:

- compile the protocol defined in `msg.x`: `rpcgen msg.x`.

  This generates the header files (`msg.h`), client stub (`msg_clnt.c`), and server stub (`msg_svc.c`).
- compile the client executable:

  ```
  cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
  ```
- compile the server executable:

  ```
  cc msg_proc.c msg_svc.c -o msg_server -lnsl
  ```

The C object files must be linked with the library `libnsl`, which contains all of the networking functions, including those for RPC and XDR.

In this example, no XDR routines were generated because the application uses only the basic types that are included in `libnsl`. Let us consider further what `rpcgen` did with the input file `msg.x`:

- It created a header file called `msg.h` that contained `#define` statements for `MESSAGEPROG`, `MESSAGEVERS`, and `PRINTMESSAGE` for use in the other modules. This file**must** be included by both the client and server modules.
- It created the client stub routines in the `msg_clnt.c` file. Here there is only one, the `printmessage_1` routine, that was called from the `rprintmsg` client program. If the name of an `rpcgen` input file is `prog.x`, the client stub's output file is called `prog_clnt.c`.
- It created the server program in `msg_svc.c` that calls `printmessage_1` from `msg_proc.c`. The rule for naming the server output file is similar to that of the client: for an input file called `prog.x`, the output server file is named `prog_svc.c`.

Once created, the server program is installed on a remote machine and run. (If the machines are homogeneous, the server binary can just be copied. If they are not, the server source files must be copied to and compiled on the remote machine.)

Passing Complex Data Structures

`rpcgen` can also be used to generate XDR routines -- the routines that convert local data structures into XDR format and vice versa.

let us consider `dir.x` a remote directory listing service, built using `rpcgen` both to generate stub routines and to generate the XDR routines.

The RPC Protocol Description File: `dir.x` is as follows:

```
/*
* dir.x: Remote directory listing protocol
*
* This example demonstrates the functions of rpcgen.
*/

const MAXNAMELEN = 255; /* max length of directory entry */

typedef string nametype<MAXNAMELEN>; /* director entry */

typedef struct namenode *namelist; /* link in the listing */

/* A node in the directory listing */

struct namenode {
   nametype name; /* name of directory entry */
   namelist next; /* next entry */
  };

/*
* The result of a READDIR operation
*
* a truly portable application would use
* an agreed upon list of error codes
* rather than (as this sample program
* does) rely upon passing UNIX errno's
* back.
*
* In this example: The union is used
* here to discriminate between successful
* and unsuccessful remote calls.
*/

union readdir_res switch (int errno) {
   case 0:
     namelist list; /* no error: return directory listing */
   default:
     void; /* error occurred: nothing else to return */
   };

/* The directory program definition */

program DIRPROG {
   version DIRVERS {
    readdir_res
    READDIR(nametype) = 1;
```

```
    } = 1;
} = 0x20000076;
```

You can redefine types (like `readdir_res` in the example above) using the `struct, union`, and `enum` RPC language keywords. These keywords are not used in later declarations of variables of those types. For example, if you define a `union`, `my_un`, you declare using only `my_un`, and not union `my_un`. `rpcgen` compiles RPC unions into C structures. Do not declare C unions using the union keyword.

Running `rpcgen` on `dir.x` generates four output files:

- the header file, `dir.h`,
- the client stub, `dir_clnt.c`,
- the server skeleton, `dir_svc.c` ,and
- the XDR routines in the file `dir_xdr.c`.

This last file contains the XDR routines to convert declared data types from the host platform representation into XDR format, and vice versa. For each RPCL data type used in the `.x` file, `rpcgen` assumes that `libnsl` contains a routine whose name is the name of the data type, prepended by the XDR routine header `xdr_` (for example, `xdr_int`). If a data type is defined in the `.x` file, `rpcgen` generates the required `xdr_` routine. If there is no data type definition in the `.x` source file (for example, `msg.x`, above), then no `_xdr.c` file is generated. You can write a `.x` source file that uses a data type not supported by `libnsl`, and deliberately omit defining the type (in the `.x` file). In doing so, you must provide the `xdr_` routine. This is a way to provide your own customized `xdr_` routines.

The server-side of the `READDIR procedure`, `dir_proc.c` is shown below:

```
/*
* dir_proc.c: remote readdir
* implementation
*/

#include <dirent.h>
#include "dir.h" /* Created by rpcgen */

extern int errno;

extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(nametype *dirname, struct svc_req *req)

{
  DIR *dirp;
  struct dirent *d;
  namelist nl;
  namelist *nlp;

  static readdir_res res; /* must be static! */

  /* Open directory */
  dirp = opendir(*dirname);

 if (dirp == (DIR *)NULL) {
    res.errno = errno;
   return (&res);
  }
```

```
  /* Free previous result */
  xdr_free(xdr_readdir_res, &res);

  /*
   * Collect directory entries.
   * Memory allocated here is free by
   * xdr_free the next time readdir_1
   * is called
   */

   nlp = &res.readdir_res_u.list;
   while (d = readdir(dirp)) {
     nl = *nlp = (namenode *)
     malloc(sizeof(namenode));
     if (nl == (namenode *) NULL) {
       res.errno = EAGAIN;
       closedir(dirp);
       return(&res);
      }
    nl->name = strdup(d->d_name);
    nlp = &nl->next;
  }

  *nlp = (namelist)NULL;

  /* Return the result */
  res.errno = 0;
  closedir(dirp);
  return (&res);
}
```

The Client-side Implementation of implementation of the READDIR procedure, `rls.c` is given below:

```
/*
* rls.c: Remote directory listing client
*/

#include <stdio.h>
#include "dir.h" /* generated by rpcgen */

extern int errno;

main(int argc, char *argv[])


{
  CLIENT *clnt;
  char *server;
  char *dir;
  readdir_res *result;
  namelist nl;

  if (argc != 3) {
     fprintf(stderr, "usage: %s host
     directory\n",argv[0]);
     exit(1);
   }
```

```
    server = argv[1];
    dir = argv[2];

    /*
     * Create client "handle" used for
     * calling MESSAGEPROG on the server
     * designated on the command line.
     */

    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");

    if (clnt == (CLIENT *)NULL) {
       clnt_pcreateerror(server);
       exit(1);
      }

    result = readdir_1(&dir, clnt);

    if (result == (readdir_res *)NULL) {
       clnt_perror(clnt, server);
       exit(1);
     }

    /* Okay, we successfully called
     * the remote procedure.
     */

    if (result->errno != 0) {
      /* Remote system error. Print
       * error message and die.
       */

      errno = result->errno;
      perror(dir);
      exit(1);
     }

    /* Successfully got a directory listing.
     * Print it.
     */

    for (nl = result->readdir_res_u.list;
         nl != NULL;
         nl = nl->next) {
           printf("%s\n", nl->name);
       }

    xdr_free(xdr_readdir_res, result);
    clnt_destroy(cl);
    exit(0);
  }
```

As in other examples, execution is on systems named local and remote. The files are compiled and run as follows:

```
remote$ rpcgen dir.x
```

```
remote$ cc -c dir_xdr.c
remote$ cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
remote$ cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
remote$ dir_svc
```

When you install `rls` on system local, you can list the contents of `/usr/share/lib` on system remote as follows:

```
local$ rls remote /usr/share/lib
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
local$
```

`rpcgen` generated client code does not release the memory allocated for the results of the RPC call. Call `xdr_free()` to release the memory when you are finished with it. It is similar to calling the `free()` routine, except that you pass the XDR routine for the result. In this example, after printing the list, `xdr_free(xdr_readdir_res, result);` was called.

Note - Use `xdr_free()` to release memory allocated by `malloc()`. Failure to use `xdr_free to()` release memory results in memory leaks.

Preprocessing Directives

`rpcgen` supports C and other preprocessing features. C preprocessing is performed on `rpcgen` input files before they are compiled. All standard C preprocessing directives are allowed in the `.x` source files. Depending on the type of output file being generated, five symbols are defined by `rpcgen`. `rpcgen` provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly to the output file, with no action on the line's content. Caution is required because `rpcgen` does not always place the lines where you intend. Check the output source file and, if needed, edit it.

The following symbols may be used to process file specific output:

`RPC_HDR`
: -- Header file output

`RPC_XDR`
: -- XDR routine output

`RPC_SVC`
: -- Server stub output

`RPC_CLNT`
: -- Client stub output

`RPC_TB`
: -- Index table output

The following example illustrates tthe use of `rpcgen`Œs pre-processing features.

```
/*
* time.x: Remote time protocol
*/
program TIMEPROG {
```

```
   version TIMEVERS {
     unsigned int TIMEGET() = 1;
    } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

cpp Directives

`rpcgen` supports C preprocessing features. `rpcgen` defaults to use `/usr/ccs/lib/cpp` as the C preprocessor. If that fails, `rpcgen` tries to use `/lib/cpp`. You may specify a library containing a different cpp to `rpcgen` with the `'-' Y` flag.

For example, if /usr/local/bin/cpp exists, you can specify it to rpcgen as follows:

```
rpcgen -Y /usr/local/bin test.x
```

Compile-Time Flags

This section describes the `rpcgen` options available at compile time. The following table summarizes the options which are discussed in this section.

Option	Flag	Comments
C-style	`'-' N`	Also called Newstyle mode
ANSI C	`'-' C`	Often used with the -N option
MT-Safe code	`'-' M`	For use in multithreaded environments
MT Auto mode	`'-' A`	-A also turns on -M option
TS-RPC library '	`-' b`	TI-RPC library is default
`xdr_inline` count	`'-' i`	Uses 5 packed elements as default,
		but other number may be specified

Client and Server Templates

`rpcgen` generates sample code for the client and server sides. Use these options to generate the desired templates.

Flag	Function
`'-' a`	Generate all template files
`'-' Sc`	Generate client-side template
`'-' Ss`	Generate server-side template

`'-' Sm`	Generate makefile template

The files can be used as guides or by filling in the missing parts. These files are in addition to the stubs generated.

Example rpcgen compile options/templates

A C-style mode server template is generated from the `add.x` source by the command:

```
rpcgen -N -Ss -o add_server_template.c add.x
```

The result is stored in the file `add_server_template.c`.

A C-style mode, client template for the same `add.x` source is generated with the command line:

```
rpcgen -N -Sc -o add_client_template.c add.x
```

The result is stored in the file `add_client_template.c`.

A make file template for the same `add.x` source is generated with the command line:

```
rpcgen -N -Sm -o mkfile_template add.x
```

The result is stored in the file `mkfile_template`. It can be used to compile the client and the server. If the `'-' a` flag is used as follows:

```
rpcgen -N -a add.x
```

`rpcgen` generates all three template files. The client template goes into `add_client.c`, the server template to `add_server.c`, and the makefile template to makefile.a. If any of these files already exists, `rpcgen` displays an error message and exits.

Note - When you generate template files, give them new names to avoid the files being overwritten the next time rpcgen is executed.

Recommended Reading

The book *Power Programming with RPC* by John Bloomer, O'Reilly and Associates, 1992, is the most comprehensive on the topic and is essential reading for further RPC programming.

Exercises

Exercise 12834

Use `rpcgen` the generate and compile the `rprintmsg` listing example given in this chapter.

Exercise 12835

Use `rpcgen` the generate and compile the `dir` listing example given in this chapter.

Exercise 12836

Develop a Remote Procedure Call suite of programs that enables a user to search for specific files or filtererd files in a remote directory. That is to say you can search for a named file *e.g. file.c* or all files named `*.c` or even `*.x`.

Exercise 12837

Develop a Remote Procedure Call suite of programs that enables a user to `grep` files remotely. You may use code developed previously or unix system calls to implement `grep`.

Exercise 12838

Develop a Remote Procedure Call suite of programs that enables a user to *list* the contents of a named remote files.

Dave Marshall
1/5/1999

Subsections

Writing Larger Programs

This Chapter deals with theoretical and practical aspects that need to be considered when writing larger programs.

When writing large programs we should divide programs up into modules. These would be separate source files. `main()` would be in one file, `main.c` say, the others will contain functions.

We can create our own library of functions by writing a *suite* of subroutines in one (or more) modules. In fact modules can be shared amongst many programs by simply including the modules at compilation as we will see shortly..

There are many advantages to this approach:

- the modules will naturally divide into common groups of functions.
- we can compile each module separately and link in compiled modules (more on this later).
- UNIX utilities such as **make** help us maintain large systems (see later).

Header files

If we adopt a modular approach then we will naturally want to keep variable definitions, function prototypes *etc.* with each module. However what if several modules need to share such definitions?

It is best to centralise the definitions in one file and share this file amongst the modules. Such a file is usually called a **header file**.

Convention states that these files have a `.h` suffix.

We have met standard library header files already *e.g*:

```
#include <stdio.h>
```

We can define our own header files and include then our programs via:

```
#include ``my_head.h''
```

NOTE: Header files usually ONLY contain definitions of data types, function prototypes and C preprocessor commands.

Consider the following simple example of a large program (Fig. 34.1) .

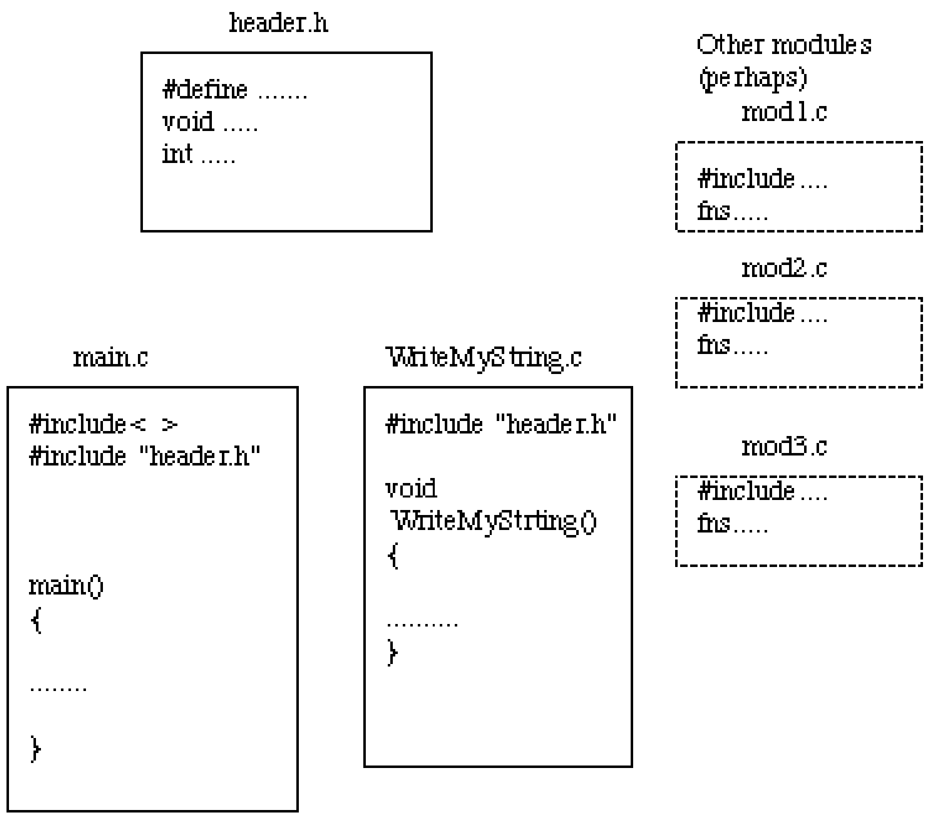


Fig. **Modular structure of a C program** The full listings `main.c`, `WriteMyString.c` and `header.h` as as follows:

main.c:

```
/*
 *      main.c
 */
#include "header.h"
#include <stdio.h>

char    *AnotherString = "Hello Everyone";
```

```
main()
{
        printf("Running...\n");

        /*
         *      Call WriteMyString() - defined in another file
         */
        WriteMyString(MY_STRING);

        printf("Finished.\n");
}
```

WriteMyString.c:

```
/*
 *      WriteMyString.c
 */
extern char     *AnotherString;

void WriteMyString(ThisString)
char    *ThisString;
{
        printf("%s\n", ThisString);
        printf("Global Variable = %s\n", AnotherString);
}
```

header.h:

```
/*
 *      header.h
 */
#define MY_STRING "Hello World"

void WriteMyString();
```

We would usually compile each module separately (more later).

Some modules have a `#include ``header.h''` that share common definitions.

Some, like *main.c*, also include standard header files also.

`main` calls the function `WriteMyString()` which is in *WriteMyString.c* module.

The function prototype `void` for `WriteMyString` is defined in *Header.h*

NOTE that in general we must resolve a tradeoff between having a desire for each `.c` module to have access to the information it needs solely for its job and the practical reality of maintaining lots of header files.

Up to some moderate program size it is probably best to one or two header files that share more than one modules definitions.

For larger programs get UNIX to help you (see later).

One problem left with module approach:

SHARING VARIABLES

If we have global variables declared and instantiated in one module how can pass knowledge of this to other modules.

We could pass values as parameters to functions, BUT:

- this can be laborious if we pass the same parameters to many functions and / or if there are long argument lists involved.
- very large arrays and structures are difficult to store locally -- memory problems with stack.

External variables and functions

``Internal'' implies arguments and functions are defined inside functions -- **Local**

``External'' variables are defined outside of
functions -- they are potentially available to the whole program (Global) but **NOT necessarily**.

External variables are always permanent.

NOTE: That in C, all function definitions are external. We CANNOT have embedded function declarations like in PASCAL.

Scope of externals

An external variable (or function) is not always totally global.

C applies the following rule:

The scope of an external variable (or function) begins at its point of declaration and lasts to the end of the file (module) it is declared in.

Consider the following:

```
main()
   { .... }

int what_scope;
float end_of_scope[10]
```

```
void what_global()
                 { .... }

char alone;

float fn()
                 { .... }
```

main cannot see what_scope or end_of_scope but the functions what_global and fn can. ONLY fn can see alone.

This is also the one of the reasons why we should *prototype* functions before the body of code *etc.* is given.

So here main will not know anything about the functions what_global and fn. what_global does not know about fn but fn knows about what_global since it is declared above.

NOTE: The other reason we *prototype* functions is that some checking can be done the parameters passed to functions.

If we need to refer to an external variable before it is declared <u>or</u> if it is defined in another module we must declare it as an extern variable. *e.g.*

```
extern int what_global
```

So returning to the modular example. We have a global string AnotherString declared in main.c and shared with WriteMyString.c where it is declared extern.

BEWARE the extern prefix is a *declaration* <u>NOT</u> a *definition*. *i.e* **NO STORAGE** is set aside in memory for an extern variable -- it is just an announcement of the property of a variable.

The actual variable <u>must</u> only be defined once in the whole program -- you can have as many extern declarations as needed.

Array sizes must obviously be given with declarations but are not needed with extern declarations. *e.g.*:

```
main.c:    int arr[100]:

file.c:    extern int arr[];
```

Advantages of Using Several Files

The main advantages of spreading a program across several files are:

- Teams of programmers can work on programs. Each programmer works on a different file.
- An object oriented style can be used. Each file defines a particular type of object as a datatype and operations on that object as functions. The implementation of the object can be kept private from the rest of the program. This makes for well structured programs which are easy to maintain.
- Files can contain all functions from a related group. For Example all matrix operations. These can then be accessed like a function library.
- Well implemented objects or function definitions can be re-used in other programs, reducing development time.
- In very large programs each major function can occupy a file to itself. Any lower level functions used to implement them can be kept in the same file. Then programmers who call the major function need not be distracted by all the lower level work.
- When changes are made to a file, only that file need be re-compiled to rebuild the program. The UNIX make facility is very useful for rebuilding multi-file programs in this way.

How to Divide a Program between Several Files

Where a function is spread over several files, each file will contain one or more functions. One file will include main while the others will contain functions which are called by others. These other files can be treated as a library of functions.

Programmers usually start designing a program by dividing the problem into easily managed sections. Each of these sections might be implemented as one or more functions. All functions from each section will usually live in a single file.

Where objects are implemented as data structures, it is usual to to keep all functions which access that object in the same file. The advantages of this are:

- The object can easily be re-used in other programs.
- All related functions are stored together.
- Later changes to the object require only one file to be modified.

Where the file contains the definition of an object, or functions which return values, there is a further restriction on calling these functions from another file. Unless functions in another file are told about the object or function definitions, they will be unable to compile them correctly.

The best solution to this problem is to write a header file for each of the C files. This will have the same name as the C file, but ending in .h. The header file contains definitions of all the functions used in the C file.

Whenever a function in another file calls a function from our C file, it can define the function by making a `#include` of the appropriate `.h` file.

Organisation of Data in each File

Any file must have its data organised in a certain order. This will typically be:

- A preamble consisting of `#define`d constants, `#include`d header files and `typedef`s of important datatypes.
- Declaration of global and external variables. Global variables may also be initialised here.
- One or more functions.

The order of items is important, since every object must be defined before it can be used. Functions which return values must be defined before they are called. This definition might be one of the following:

- Where the function is defined and called in the same file, a full declaration of the function can be placed ahead of any call to the function.
- If the function is called from a file where it is not defined, a prototype should appear before the call to the function.

A function defined as

```
float find_max(float a, float b, float c)
{  /* etc ... ... */
```

would have a prototype of

```
float find_max(float a, float b, float c);
```

The prototype may occur among the global variables at the start of the source file. Alternatively it may be declared in a header file which is read in using a `#include`.

It is important to remember that all C objects should be declared before use.

The Make Utility

The *make* utility is an intelligent program manager that maintains integrity of a collection of program modules, a collection of programs or a complete system -- does not have be programs in practice can be any system of files (*e.g.* chapters of text in book being typeset).

Its main use has been in assisting the development of software systems.

Make was originally developed on UNIX but it is now available on most systems.

NOTE: Make is a programmers utility not part of C language or any language for that matter.

Consider the problem of maintaining a large collection of source files:

```
main.c f1.c ......... fn.c
```

We would normally compile our system via:

```
cc -o main main.c f1.c ......... fn.c
```

However, if we know that some files have been compiled previously and their sources have not changed since then we could try and save overall compilation time by linking in the object code from those files say:

```
cc -o main main.c f1.c ... fi.o .. fj.o ... fn.c
```

We can use the C compiler option (Appendix) `-c` to create a `.o` for a given module. For example:

```
cc -c main.c
```

will create a `main.o` file. We do not need to supply any library links here as these are resolved at the linking stage of compilation.

We have a problem in compiling the whole program in this *long hand* way however:

- It is time consuming to compile a .c module -- if the module has been compiled before and not been altered there is no need to recompiled it. We can just link the object files in. However, it will not be easy to remember which files are in fact up to date. If we link in an old object file our final executable program will be wrong.

- It is error prone and laborious to type a long compile sequence on the command line. There may be many of our own files to link as well as many system library files. It may be very hard to remember the correct sequence. Also if we make a slight change to our system editing command line can be error prone.

If we use the **make** utility all this control is taken care by make. In general only modules that have older object files than source files will be recompiled.

Make Programming

Make programming is fairly straightforward. Basically, we write a sequence of commands which describes how our program (or system of programs) can be constructed from source files.

The construction sequence is described in
makefiles which contain *dependency rules* and *construction rules*.

A dependency rule has two parts - a left and right side separated by a :

```
left side : right side
```

The `left side` gives the names of a *target(s)* (the names of the program or system files) to be built, whilst the `right side` gives names of files on which the target depends (eg. source files, header files, data files)

If the *target* is **out of date** with respect to the constituent parts, *construction rules* following the dependency rules are obeyed.

So for a typical C program, when a make file is run the following tasks are performed:

1. The makefile is read. Makefile says which object and library files need to be linked and which header files and sources have to be compiled to create each object file.
2. Time and date of each object file are checked against source and header files it depends on. If any source, header file later than object file then files have been altered since last compilation **THEREFORE** recompile object file(s).
3. Once all object files have been checked the time and date of all object files are checked against executable files. If any later object files will be recompiled.

NOTE: Make files can obey any commands we type from command line. Therefore we can use makefiles to do more than just compile a system source module. For example, we could make backups of files, run programs if data files have been changed or clean up directories.

Creating a makefile

This is fairly simple: just create a text file using any text editor. The *makefile* just contains a list of file dependencies and commands needed to satisfy them.

Lets look at an example makefile:

```
prog: prog.o f1.o f2.o
  c89 prog.o f1.o f2.o -lm etc.
```

```
prog.o: header.h prog.c
                 c89 -c prog.c

f1.o: header.h f1.c
                 c89 -c f1.c



f2.o: ---
                 ----
```

Make would interpret the file as follows:

1.
   prog depends on 3 files: `prog.o, f1.o and f2.o`. If any of the object files have been changed since last compilation the files must be relinked.
2.
   prog.o depends on 2 files. If these have been changed prog.o must be recompiled. Similarly for `f1.o` and `f2.o`.

The last 3 commands in the makefile are called *explicit rules* -- since the files in commands are listed by name.

We can use *implicit rules* in our makefile which let us generalise our rules and save typing.

We can take

```
f1.o: f1.c
  cc -c f1.c

f2.o: f2.c
                 cc -c f2.c
```

and generalise to this:

```
.c.o:   cc -c $<
```

We read this as .source_extension.target_extension: `command`

`$< is shorthand for file name with .c extension.`

`We can put comments in a makefile by using the # symbol. All characters following # on line are ignored.`

`Make has many built in commands similar to or actual UNIX commands. Here are a few:`

```
   break     date        mkdir
```

```
> type            chdir           mv (move or rename)
                  cd              rm (remove)             ls
                  cp (copy)               path
```

There are many more see manual pages for make (online and printed reference)

Make macros

We can define *macros* in make -- they are typically used to store source file names, object file names, compiler options and library links.

They are simple to define, *e.g.*:

```
SOURCES                 = main.c f1.c f2.c
CFLAGS          = -g -C
LIBS            = -lm
PROGRAM                 = main
OBJECTS                 = (SOURCES: .c = .o)
```

where `(SOURCES: .c = .o)` makes .c extensions of SOURCES .o extensions.

To reference or invoke a macro in make do $(macro_name).*e.g.:*

```
$(PROGRAM) : $(OBJECTS)
$(LINK.C) -o $@ $(OBJECTS) $(LIBS)
```

NOTE:

- $(PROGRAM) : $(OBJECTS) - makes a list of dependencies and targets.
- The use of an internal macros *i.e.* $@.

There are many internal macros (see manual pages) here a few common ones:

$*

    -- file name part of current dependent (minus .suffix).

$@

    -- full target name of current target.

$<

    -- .c file of target.

An example makefile for the WriteMyString modular program

```
discussed in the above is as follows:

#
# Makefile
#
SOURCES.c= main.c WriteMyString.c
INCLUDES=
CFLAGS=
SLIBS=
PROGRAM= main

OBJECTS= $(SOURCES.c:.c=.o)

.KEEP_STATE:

debug := CFLAGS= -g

all debug: $(PROGRAM)

$(PROGRAM): $(INCLUDES) $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS) $(SLIBS)

clean:
        rm -f $(PROGRAM) $(OBJECTS)
```

Running Make

Simply type `make` from command line.

UNIX automatically looks for a file called `Makefile` (note: capital M rest lower case letters).

So if we have a file called `Makefile` and we type make from command line. The `Makefile` in our current directory will get executed.

We can override this search for a file by typing `make -f make_filename`

e.g. `make -f my_make`

There are a few more `-options` for makefiles -- see manual pages.

Dave Marshall
1/5/1999

Next Up Previous
Next: hello.c **Up:** Programming in C **Previous:** Time

Program Listings

Here we give complete program listings that illustrate points in the course.

- hello.c
- printf.c
- swap.c
- args.c
- arg.c
- average.c
- cio.c
- factorial
- power.c
- ptr_arr.c
- Modular Example
  - main.c
  - WriteMyString.c
  - header.h
  - Makefile
- static.c
- malloc.c
- queue.c
- bitcount.c
- lowio.c
- print.c

- cdir.c
- list.c
- list_c.c
- fork_eg.c
- fork.c
- signal.c
- sig_talk.c
- Piping
  - plot.c
  - plotter.c
  - externals.h
- random.c
- time.c
- timer.c

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: printf.c **Up:** Program Listings **Previous:** Program Listings

hello.c

```
#include <stdio.h>
main()
{
    (void) printf("Hello World\n");
    return (0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

printf.c

```
#include <stdio.h>

char char1;     /* first character */
char char2;     /* second character */
char char3;     /* third character */

main()
{
    char1 = 'A';
    char2 = 'B';
    char3 = 'C';
    (void)printf("%c%c%c reversed is %c%c%c\n",
        char1, char2, char3,
        char3, char2, char1);
    return (0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

swap.c

```
/*      exchange values */

#include        <stdio.h>

void swap(float *x, float *y);

main()
{
        float   x, y;

        printf("Please input 1st value: ");
        scanf("%f", &x);
        printf("Please input 2nd value: ");
        scanf("%f", &y);
        printf("Values BEFORE 'swap' %f, %f\n", x, y);
        swap(&x, &y);   /*      address of x, y */
        printf("Values AFTER  'swap' %f, %f\n", x, y);
        return 0;
}

/*      exchange values within function */

void swap(float *x, float *y)
{
        float   t;

        t = *x; /*      *x is value pointed to by x     */
        *x = *y;
        *y = t;
        printf("Values WITHIN 'swap' %f, %f\n", *x, *y);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: arg.c **Up:** Program Listings **Previous:** swap.c

args.c

```
#include <stdio.h>

main(int argc, char **argv)

{ /* program to print arguments from command line */

     int i;

     printf("argc = %d\n\n",argc);
     for (i=0;i<argc;++i)
      printf("argv[%d]: %s\n",i, argv[i]);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

arg.c

```
\* program to read command line input and open files specified */


#include <stdio.h>

main(argc, argv)
int argc;
char **argv;
{
    int c;
    FILE *from, *to;

    /*
     * Check our arguments.
     */
    if (argc != 3) {
        fprintf(stderr, "Usage: %s from-file to-file\n", *argv);
        exit(1);
    }

    /*
     * Open the from-file for reading.
     */
    if ((from = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]);
        exit(1);
    }

    /*
     * Open the to-file for appending.  If to-file does
     * not exist, fopen will create it.
     */
    if ((to = fopen(argv[2], "a")) == NULL) {
        perror(argv[2]);
        exit(1);
    }

    /*
     * Now read characters from from-file until we
```

```
     * hit end-of-file, and put them onto to-file.
     */
    while ((c = getc(from)) != EOF)
        putc(c, to);

    /*
     * Now close the files.
     */
    fclose(from);
    fclose(to);
    exit(0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: cio.c **Up:** Program Listings **Previous:** arg.c

average.c

```
#include <stdio.h>

float data[5];  /* data to average and total */
float total;    /* the total of the data items */
float average;  /* average of the items */

main()
{
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 45.0;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average =  total / 5.0;
    (void)printf("Total %f Average %f\n", total, average);
    return (0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: factorial **Up:** Program Listings **Previous:** average.c

cio.c

```
\* program to echo keyboard input to screen */

#include        <stdio.h>

/*      copy input to output    */

main()
{
        int     c;

        while ((c = getchar()) != EOF)
                putchar(c);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: power.c **Up:** Program Listings **Previous:** cio.c

factorial

```
\* e.g use of functions factorials *\
\* fact(n) = n*(n-1)*....2*1 *\

#include <stdio.h>

main()
{
    int n, m;

    printf("Enter a number: ");
    scanf("%d", &n);

    m = fact(n);
    printf("The factorial of %d is %d.\n", n, m);
    exit(0);
}

fact(n)
int n;
{
    if (n == 0)
        return(1);

    return(n * fact(n-1));
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next | Up | Previous
Next: ptr_arr.c **Up:** Program Listings **Previous:** factorial

power.c

```
#include <stdio.h>

int     power (int m, int n);

main () {
    int     i;

    printf ("power\t   2^power\t -3^power\n");
    for (i = 0; i < 10; ++i)
        printf ("%5d \t%8d \t%8d\n", i, power (2, i), power (-3, i));
    return 0;
}

int     power (int base, int n) {
    int     i,
            p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p *= base;
    return p;
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

ptr_arr.c

```
#define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "012345678";

main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; index++) {
        (void)printf(
            "&array[index]=0x%x (array+index)=0x%x array[index]=0x%x\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: main.c **Up:** Program Listings **Previous:** ptr_arr.c

Modular Example

We list here three C modules that comprise of the large program example. The Makefile is also included.

* main.c
* WriteMyString.c
* header.h
* Makefile

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: WriteMyString.c **Up:** Modular Example **Previous:** Modular Example

main.c

```
/*
 *      main.c
 */
#include "header.h"
#include <stdio.h>

char    *AnotherString = "Hello Everyone";

main()
{
        printf("Running...\n");

        /*
         *      Call WriteMyString() - defined in another file
         */
        WriteMyString(MY_STRING);

        printf("Finished.\n");
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: header.h **Up:** Modular Example **Previous:** main.c

WriteMyString.c

```
/*
 *      WriteMyString.c
 */
extern char     *AnotherString;

void WriteMyString(ThisString)
char    *ThisString;
{
        printf("%s\n", ThisString);
        printf("Global Variable = %s\n", AnotherString);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

header.h

```
/*
 *       header.h
 */
#define MY_STRING "Hello World"

void WriteMyString();
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: static.c **Up:** Modular Example **Previous:** header.h

Makefile

```
#
# Makefile
#
SOURCES.c= main.c WriteMyString.c
INCLUDES=
CFLAGS=
SLIBS=
PROGRAM= main

OBJECTS= $(SOURCES.c:.c=.o)

.KEEP_STATE:

debug := CFLAGS= -g

all debug: $(PROGRAM)

$(PROGRAM): $(INCLUDES) $(OBJECTS)
	$(LINK.c) -o $@ $(OBJECTS) $(SLIBS)

clean:
	rm -f $(PROGRAM) $(OBJECTS)
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

static.c

```
#include <stdio.h>

void stat();

main() {
    int counter;    /* loop counter */


    for (counter = 0; counter < 5; counter++) {
        stat();
    }

}

void stat()

{ int temporary = 1;
  static int permanent = 1;

        (void)printf("Temporary %d Permanent %d\n",
            temporary, permanent);
        temporary++;
        permanent++;
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: queue.c **Up:** Program Listings **Previous:** static.c

malloc.c

```
#include <stdlib.h>     /* using ANSI C standard libraries */
#include <malloc.h>

main()
{
    char *string_ptr;

    string_ptr = malloc(80);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: bitcount.c **Up:** Program Listings **Previous:** malloc.c

queue.c

```
/* Corrected  19/3/90 - nolonger leaves queue in memory!     */
/* Note UNIX would clear the dynamically allocated memory    */
/* when the program ends                                     */
/*                                                           */
/* queue.c                                                   */
/* Demo of dynamic data structures in C                      */

#include <stdio.h>

#define FALSE 0
#define NULL 0

typedef struct {
    int     dataitem;
    struct listelement *link;
}               listelement;

void Menu (int *choice);
listelement * AddItem (listelement * listpointer, int data);
listelement * RemoveItem (listelement * listpointer);
void PrintQueue (listelement * listpointer);
void ClearQueue (listelement * listpointer);

main () {
    listelement listmember, *listpointer;
    int     data,
            choice;

    listpointer = NULL;
    do {
        Menu (&choice);
        switch (choice) {
            case 1:
                printf ("Enter data item value to add  ");
                scanf ("%d", &data);
                listpointer = AddItem (listpointer, data);
                break;
            case 2:
                if (listpointer == NULL)
                    printf ("Queue empty!\n");
                else
                    listpointer = RemoveItem (listpointer);
                break;
            case 3:
                PrintQueue (listpointer);
                break;

            case 4:
                break;

            default:
                printf ("Invalid menu choice - try again\n");
                break;
```

```
        }
    } while (choice != 4);
    ClearQueue (listpointer);
}                               /* main */

void Menu (int *choice) {

    char    local;

    printf ("\nEnter\t1 to add item,\n\t2 to remove item\n\
\t3 to print queue\n\t4 to quit\n");
    do {
        local = getchar ();
        if ((isdigit (local) == FALSE) && (local != '\n')) {
            printf ("\nyou must enter an integer.\n");
            printf ("Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n");
        }
    } while (isdigit ((unsigned char) local) == FALSE);
    *choice = (int) local - '0';
}

listelement * AddItem (listelement * listpointer, int data) {

    listelement * lp = listpointer;

    if (listpointer != NULL) {
        while (listpointer -> link != NULL)
            listpointer = listpointer -> link;
        listpointer -> link = (struct listelement  *) malloc (sizeof (listelement));
        listpointer = listpointer -> link;
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return lp;
    }
    else {
        listpointer = (struct listelement  *) malloc (sizeof (listelement));
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return listpointer;
    }
}

listelement * RemoveItem (listelement * listpointer) {

    listelement * tempp;
    printf ("Element removed is %d\n", listpointer -> dataitem);
    tempp = listpointer -> link;
    free (listpointer);
    return tempp;
}

void PrintQueue (listelement * listpointer) {

    if (listpointer == NULL)
        printf ("queue is empty!\n");
    else
        while (listpointer != NULL) {
            printf ("%d\t", listpointer -> dataitem);
            listpointer = listpointer -> link;
        }
    printf ("\n");
}
```

```
void ClearQueue (listelement * listpointer) {

    while (listpointer != NULL) {
        listpointer = RemoveItem (listpointer);
    }
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: lowio.c **Up:** Program Listings **Previous:** queue.c

bitcount.c

```
/* binary counting exampple -counts bits set to 1 in an 8 bit number */


/* acc -o bitcount bitcount.c on SUNS */

/* c89 -o bitcount bitcount.c on DECS */

#include <stdio.h>

unsigned char bitcount(unsigned char); /* prototype */

main()

{  unsigned char i8,count;
   int i;

   printf("Enter number (0 - 255 decimal)\n");
   scanf("%d",&i);

   if (( i < 0 ) || (i > 255))
     { printf("Error:Number out of range = %d\n", i);
       exit(1);
     }

   i8 = (unsigned char) i;

   count = bitcount(i8);

   printf("\n\nNumber of bits set to 1 in %d = %d\n",i,count);

}

unsigned char bitcount(unsigned char x)

{  unsigned char count;


   for (count = 0; x!=0; x>>=1)
     if ( x & 01 )
       ++count;

   return count;
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

lowio.c

```
/*********************************************** lowio.c ********/

#include <fcntl.h>
#include <stdio.h>
#define PERMS 0600      /* r,w permission owner only  (octal no.)*/

void inputtext (char *buf, int fd);
void display (char *buf, int fd);

main () {
    char    buf[BUFSIZ];
    int     fd1,
            fd2,
            t;

    if ((fd1 = creat ("iotest", PERMS)) == -1) {
        printf ("Cannot open file with creat\n");
        exit (1);
    }

    inputtext (buf, fd1);

    close (fd1);

    if ((fd2 = open ("iotest", 0, O_RDONLY)) == -1) {
        printf ("Cannot open file\n");
        exit (1);
    }
    display (buf, fd2);
    close (fd2);
}

void inputtext (char *buf, int fd1) {
    register int    t;

    printf ("Enter lines of text, end with quit\n");
    do {
        for (t = 0; t < BUFSIZ; t++)
            buf[t] = '\0';
        gets (buf);
```

```
        if (write (fd1, buf, BUFSIZ) != BUFSIZ) {
            printf ("Error in writing\n");
            exit (1);
        }
    } while (strcmp (buf, "quit"));
}

void display (char *buf, int fd2) {
    for (;;) {
        if (read (fd2, buf, BUFSIZ) == 0)
            return;
        printf ("%s\n", buf);
    }
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: cdir.c **Up:** Program Listings **Previous:** lowio.c

print.c

```
/********************************************************
 * print -- format files for printing                   *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>      /* ANSI Standard only */

int verbose = 0;         /* verbose mode (default = false) */
char *out_file = "print.out";   /* output filename */
char *program_name;      /* name of the program (for errors) */
int line_max = 66;       /* number of lines per page */

main(int argc, char *argv[])
{
    void do_file(char *); /* print a file */
    void usage(void);     /* tell user how to use the program */

    /* save the program name for future use */
    program_name = argv[0];

    /*
     * loop for each option.
     *   Stop if we run out of arguments
     *   or we get an argument without a dash.
     */
    while ((argc > 1) && (argv[1][0] == '-')) {
        /*
         * argv[1][1] is the actual option character.
         */
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
            /*
             * -o<name>  output file
             *     [0] is the dash
             *     [1] is the "o"
             *     [2] starts the name
```

```
                 */
                case 'o':
                    out_file = &argv[1][2];
                    break;
                /*
                 * -l<number> set max number of lines
                 */
                case 'l':
                    line_max = atoi(&argv[1][2]);
                    break;
                default:
                    (void)fprintf(stderr,"Bad option %s\n", argv[1]);
                    usage();
            }
            /*
             * move the argument list up one
             * move the count down one
             */
            argv++;
            argc--;
        }

        /*
         * At this point all the options have been processed.
         * Check to see if we have no files in the list
         * and if so, we need to process just standard in.
         */
        if (argc == 1) {
            do_file("print.in");
        } else {
            while (argc > 1) {
              do_file(argv[1]);
              argv++;
              argc--;
            }
        }
        return (0);
    }
    /********************************************************
     * do_file -- dummy routine to handle a file            *
     *                                                      *
     * Parameter                                            *
     *      name -- name of the file to print               *
     ********************************************************/
    void do_file(char *name)
    {
        (void)printf("Verbose %d Lines %d Input %s Output %s\n",
            verbose, line_max, name, out_file);
    }
```

```
/********************************************************
 * usage -- tell the user how to use this program and   *
 *              exit                                    *
 ********************************************************/
void usage(void)
{
    (void)fprintf(stderr,"Usage is %s [options] [file-list]\n",
                                program_name);
    (void)fprintf(stderr,"Options\n");
    (void)fprintf(stderr,"  -v          verbose\n");
    (void)fprintf(stderr,"  -l<number>  Number of lines\n");
    (void)fprintf(stderr,"  -o<name>    Set output filename\n");
    exit (8);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

cdir.c

```
/* cdir.c program to emulate unix cd command */

/* cc -o cdir cdir.c */


#include<stdio.h>
/* #include<sys/dir.h> */

main(int argc,char **argv)
{
        if (argc < 2)
          { printf("Usage: %s <pathname>\n",argv[0]);
            exit(1);
          }

        if (chdir(argv[1]) != 0)
          { printf("Error in \"chdir\"\n");
            exit(1);
          }
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

list.c

```
/* list.c -  C version of a simple UNIX ls utility */

/* c89 list.c -o list */

/* need types.h and dir.h for definitions of scandir and alphasort */
#include <sys/types.h>
#include <sys/dir.h>

/* definition for getwd ie MAXPATHLEN etc */
#include <sys/param.h>

#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE

/* prototype std lib functions */
extern  int alphasort();

/* variable to store current path */
char pathname[MAXPATHLEN];

main()
{ int count,i;
  struct direct **files;
  int file_select();

  if (getwd(pathname) == NULL )
   { printf("Error getting path\n);
     exit(1);
   }
  printf("Current Working Directory = %s\n",pathname);

  count =
    scandir(pathname, &files, file_select, alphasort);


  /* If no files found, make a non-selectable menu item */
  if (count <= 0) {
    printf("No files in this directory\n");
    exit(0);
  }
```

```
  printf("Number of files = %d\n",count);

  for (i=1;i<count+1;++i)
   { printf("%s   ",files[i-1]->d_name);
     if ( (i % 4) == 0) printf("\n");
   }

  printf("\n"); /* flush buffer */
}


int
file_select(struct direct   *entry)
{  /* ignore . and .. entries */
      if ((strcmp(entry->d_name, ".") == 0) ||
            (strcmp(entry->d_name, "..") == 0))
                return (FALSE);
     else
       return (TRUE);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

list_c.c

```
/* list_c.c -  list C realted files ie .c .o .h files */

/* c89 list_c.c -o list_c */

#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE

extern  int alphasort();
char *rindex(char *s, char c);

char pathname[MAXPATHLEN];

main()
{ int count,i;
  struct direct **files;
  int file_select();

(char *) getwd(pathname);
  printf("Current Working Directory = %s\n",pathname);

  count =
    scandir(pathname, &files, file_select, alphasort);


  /* If no files found, make a non-selectable menu item */
  if (count <= 0) {
    printf("No files in this directory\n");
    exit(0);
  }

  printf("Number of files = %d\n",count);

  for (i=0;i<count;++i)
   { printf("%s ",files[i]->d_name);
     if ( (i % 4) == 0) printf("\n");
   }

  printf("\n"); /* flush buffer */
}


int
file_select(struct direct   *entry)
{
        char            *ptr;
        char            tmp[MAXPATHLEN];
```

```
      if ((strcmp(entry->d_name, ".") == 0) ||
            (strcmp(entry->d_name, "..") == 0))
                return (FALSE);

        /* Check for .c or .o or .h filename extensions */
        ptr = rindex(entry->d_name, '.');
        if ((ptr != NULL) &&
            ((strcmp(ptr, ".c") == 0) || (strcmp(ptr, ".h") == 0) || (strcmp(ptr,
".o") == 0) ))
                return (TRUE);

}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: fork.c **Up:** Program Listings **Previous:** list_c.c

fork_eg.c

```
/* fork_eg.c --- simple eg of fork in UNIX */

main()
{ int return_value;

  printf("Forking process\n");
  fork();
  printf("The process id is %d and return value is %d\n",getpid(),return_value);
  execl("/bin/ls/","ls","-l",0);
  printf("This line is not printed\n");
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: signal.c **Up:** Program Listings **Previous:** fork_eg.c

fork.c

```
/* fork.c - example of a fork in a program */
/* The program asks for UNIX commands to be typed and inputted to a string*/
/* The string is then "parsed" by locating blanks etc. */
/* Each command and sorresponding arguments are put in a args array */
/* execvp is called to execute these commands in child process */
/* spawned by fork() */

/* c89 -o fork fork.c */

#include <stdio.h>

main()
{
    char buf[1024];
    char *args[64];

    for (;;) {
        /*
         * Prompt for and read a command.
         */
        printf("Command: ");

        if (gets(buf) == NULL) {
            printf("\n");
            exit(0);
        }

        /*
         * Split the string into arguments.
         */
        parse(buf, args);

        /*
         * Execute the command.
         */
        execute(args);
    }
}

/*
 * parse--split the command in buf into
 *         individual arguments.
 */
parse(buf, args)
char *buf;
char **args;
{
    while (*buf != NULL) {
        /*
```

```
         * Strip whitespace.  Use nulls, so
         * that the previous argument is terminated
         * automatically.
         */
        while ((*buf == ' ') || (*buf == '\t'))
            *buf++ = NULL;

        /*
         * Save the argument.
         */
        *args++ = buf;

        /*
         * Skip over the argument.
         */
        while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
            buf++;
    }

    *args = NULL;
}

/*
 * execute--spawn a child process and execute
 *          the program.
 */
execute(args)
char **args;
{
    int pid, status;

    /*
     * Get a child process.
     */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);

        /* NOTE: perror() produces a short  error  message  on  the  standard
           error describing the last error encountered during a call to
           a system or library function.
       */
    }

    /*
     * The child executes the code inside the if.
     */
    if (pid == 0) {
        execvp(*args, args);
        perror(*args);
        exit(1);

       /* NOTE: The execv() vnd execvp versions of execl() are useful when the
          number  of  arguments is unknown in advance;
          The arguments to execv() and execvp()  are the name
          of the file to be executed and a vector of strings  contain-
          ing  the  arguments.   The last argument string must be fol-
          lowed by a 0 pointer.
```

```
            execlp() and execvp() are called with the same arguments  as
            execl()  and  execv(),  but duplicate the shell's actions in
            searching for an executable file in a list  of  directories.
            The directory list is obtained from the environment.
          */
    }

    /*
     * The parent executes the wait.
     */
    while (wait(&status) != pid)
        /* empty */ ;
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: sig_talk.c **Up:** Program Listings **Previous:** fork.c

signal.c

```
#include <signal.h>

main()
{
    signal(SIGINT, SIG_IGN);

    /*
     * pause() just suspends the process until a
     * signal is received.
     */
    pause();
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next | Up | Previous
Next: Piping **Up:** Program Listings **Previous:** signal.c

sig_talk.c

```
/* sig_talk.c --- Example of how 2 processes can talk */
/* to each other using kill() and signal() */
/* We will fork() 2 process and let the parent send a few */
/* signals to it`s child  */

/* acc sig_talk.c -o sig_talk  on SUNS */
/* c89 sig_talk.c -o sig_talk  on DECS */

#include <stdio.h>
#include <signal.h>

void sighup(); /* routines child will call upon sigtrap */
void sigint();
void sigquit();

main()
{ int pid;

  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

   if (pid == 0)
     { /* child */
       signal(SIGHUP,sighup); /* set function calls */
       signal(SIGINT,sigint);
       signal(SIGQUIT, sigquit);
       for(;;); /* loop for ever */
     }
  else /* parent */
     {  /* pid hold id of child */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
```

```
        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid,SIGINT);
        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid,SIGQUIT);
        sleep(3);
      }
}

void sighup()

{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint()

{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit()

{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: plot.c **Up:** Program Listings **Previous:** sig_talk.c

Piping

Three modules make up a program that pipes output to a graphdrawing package, `gnuplot` To Run this system you must have gnupolt installed.

- plot.c
- plotter.c
- externals.h

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

plot.c

```
/* plot.c - example of unix pipe. Calls gnuplot graph drawing package to draw
   graphs from within a C program. Info is piped to gnuplot */
/* Creates 2 pipes one will draw graphs of y=0.5 and y = random 0-1.0 */
/* the other graphs of y = sin (1/x) and y = sin x */
/*  c89 -o plot plot.c plotter.c  - ON DECS */
/*  acc -o plot plot.c plotter.c  - ON SUNS */

#include "externals.h"
#include <signal.h>

#define DEG_TO_RAD(x) (x*180/M_PI)

double drand48();
void quit();

FILE *fp1, *fp2, *fp3, *fp4, *fopen();

main()
{   float i;
    float y1,y2,y3,y4;

    /* open files which will store plot data */
    if ( ((fp1 = fopen("plot11.dat","w")) == NULL) ||
           ((fp2 = fopen("plot12.dat","w")) == NULL) ||
            ((fp3 = fopen("plot21.dat","w")) == NULL) ||
             ((fp4 = fopen("plot22.dat","w")) == NULL) )
              { printf("Error can't open one or more data files\n");
                exit(1);
              }

    signal(SIGINT,quit); /* trap ctrl-c call quit fn */
    StartPlot();
    y1 = 0.5;
    srand48(1); /* set seed */
    for (i=0;;i+=0.01) /* increment i forever use ctrl-c to quit prog */
      { y2 =  (float) drand48();
        if (i == 0.0)
           y3 = 0.0;
       else
           y3 = sin(DEG_TO_RAD(1.0/i));
        y4 = sin(DEG_TO_RAD(i));

        /* load files */
        fprintf(fp1,"%f %f\n",i,y1);
        fprintf(fp2,"%f %f\n",i,y2);
        fprintf(fp3,"%f %f\n",i,y3);
        fprintf(fp4,"%f %f\n",i,y4);

        /* make sure buffers flushed so that gnuplot reads up to data file */
        fflush(fp1);
```

```
        fflush(fp2);
        fflush(fp3);
        fflush(fp4);

        /* plot graph */
        PlotOne();
        usleep(250); /* sleep for short time */
      }
}

void quit()
{  printf("\nctrl-c caught:\n Shutting down pipes\n");
   StopPlot();

   printf("closing data files\n");
   fclose(fp1);
   fclose(fp2);
   fclose(fp3);
   fclose(fp4);

   printf("deleting data files\n");
   RemoveDat();
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: externals.h **Up:** Piping **Previous:** plot.c

plotter.c

```
/* plotter.c module */
/* contains routines to plot a data file produced by another program  */
/* 2d data plotted in this version                                    */
/*********************************************************************/

#include "externals.h"

static FILE *plot1,
       *plot2,
       *ashell;

static char *startplot1 = "plot [] [0:1.1]'plot11.dat' with lines, 'plot12.dat'
                           with lines\n";

static char *startplot2 = "plot 'plot21.dat' with lines, 'plot22.dat' with lines\n";
static char *replot = "replot\n";
static char *command1= "/usr/local/bin/gnuplot> dump1";
static char *command2= "/usr/local/bin/gnuplot> dump2";
static char *deletefiles = "rm plot11.dat plot12.dat plot21.dat plot22.dat";
static char *set_term = "set terminal x11\n";

void
StartPlot(void)
 { plot1 = popen(command1, "w");
   fprintf(plot1, "%s", set_term);
   fflush(plot1);
   if (plot1 == NULL)
      exit(2);
   plot2 = popen(command2, "w");
   fprintf(plot2, "%s", set_term);
   fflush(plot2);
   if (plot2 == NULL)
      exit(2);
 }

void
RemoveDat(void)
 { ashell = popen(deletefiles, "w");
   exit(0);
 }

void
StopPlot(void)
 { pclose(plot1);
   pclose(plot2);
 }

void
PlotOne(void)
 { fprintf(plot1, "%s", startplot1);
   fflush(plot1);

   fprintf(plot2, "%s", startplot2);
   fflush(plot2);
```

```
 }

void
RePlot(void)
 { fprintf(plot1, "%s", replot);
   fflush(plot1);
 }
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: random.c **Up:** Piping **Previous:** plotter.c

externals.h

```
/* externals.h */
#ifndef EXTERNALS
#define EXTERNALS

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* prototypes */

void StartPlot(void);
void RemoveDat(void);
void StopPlot(void);
void PlotOne(void);
void RePlot(void);
#endif
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: time.c **Up:** Program Listings **Previous:** externals.h

random.c

```
/* random.c - simple example of setting random number seeds with time */

/* c89 random.c -o random */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{ int i;
  time_t t1;

  (void) time(&t1);

   srand48((long) t1); /* use time in seconds to set seed */


  printf("5 random numbers (Seed = %d):\n",(int) t1);
  for (i=0;i<5;++i)
    printf("%d ", lrand48());
  printf("\n\n"); /* flush print buffer */


  /* lrand48() returns non-negative  long  integers
     uniformly distributed over the interval (0, ~2**31)
  */
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: timer.c **Up:** Program Listings **Previous:** random.c

time.c

```
#include <sys/types.h>
#include <sys/times.h>

main()
{
    struct tms before, after;

    times(&before);

    /* ... place code to be timed here ... */

    times(&after);

    printf("User time: %ld seconds\n", after.tms_utime -
        before.tms_utime);
    printf("System time: %ld seconds\n", after.tms_stime -
        before.tms_stime);

    exit(0);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

timer.c

```
/* timer.c - simple example of timing a piece of code */

/* c89 timer.c -o timer */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{ int i;
  time_t t1,t2;

  (void) time(&t1);

   for (i=1;i<=300;++i) printf("%d %d %d\n",i, i*i, i*i*i);


   (void) time(&t2);

   printf("\nTime to do 300 squares and cubes= %d seconds\n", (int) t2-t1);
}
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Introduction **Up:** Ceilidh - On Line C Tutoring System **Previous:** Ceilidh - On Line C Tutoring System

Why Use CEILIDH ?

CEILIDH provides the following:

- On line course notes
- Automatic Assessment of C programs
- Template programs are provided to start you on an exercise. This means less typing.
- Automatic Compilation of programs
- Programs can be run against test data and user specified data
- CEILIDH will be used to help mark your coursework.
- You are allowed to resubmit your program for marking by CEILIDH. This lets you try to improve your mark.

PLEASE NOTE:

- CEILIDH marks a program in many ways: it analyses style, efficiency, `prettiness' and output.
- It is fussy about its output. TO GET FULL MARKS you will need to emulate the output almost exactly as the question requests. So read the questions CAREFULLY.
- Get plenty of practice using CEILIDH and submitting and marking exercise before your first assessments are due.
- If used properly CEILIDH should be very useful in helping you learn C.
- A C++ module is also available.(Not covered by this lecture course).

What follows is a modified version of student notes provided with system from Nottingham University. The Xwindow bits are new.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Using Ceilidh as a Student **Up:** Ceilidh - On Line C Tutoring System
Previous: Why Use CEILIDH ?

Introduction

Ceilidh is an on-line coursework administration and auto-marking facility designed to help both students and staff with programming courses. It helps students by informing them of the coursework required of them, and by permitting them to submit their work on the computer, instead of having to print things out and hand them in. It also marks programs directly, and informs the student and teacher of the mark awarded. The marking uses a comprehensive variety of static and dynamic metrics to assess the quality of submitted programs, of which details are in the paper by Zin and Foxley[1] (a copy of which may be stored on-line in Ceilidh, see below). Ceilidh also provides students with on-line access to notes, examples and solutions, and provides tutors with extensive course monitoring and tracking facilities.

This document is a guide for student users of the Ceilidh system.

The Ceilidh system acts in a number of ways for students, tutors and teachers, and can support a variety of different courses.

There are different facilities for students (reading notes and coursework definitions, looking at examples, developing programs, submitting and marking work), and tutors (observing submitted work and marks) and teachers (amending course material, setting up exercises, performing plagiarism tests). The appropriate facilities are offered to appropriate users by the Ceilidh system itself, which takes note of the login identification of the user and compares it with lists of authorised users.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Using Ceilidh as a Student

There are two ways of calling the Ceilidh system. Ceilidh may be used to support several courses in your department. You can either enter the system at a general level, and then choose the particular course you are studying, or you can enter directly into the particular course you are interested in.

To enter the system at the general level, the appropriate command (which should have been set up by your computer systems administrator) is

```
xceilidh (Xwindows version --- recommended)
```

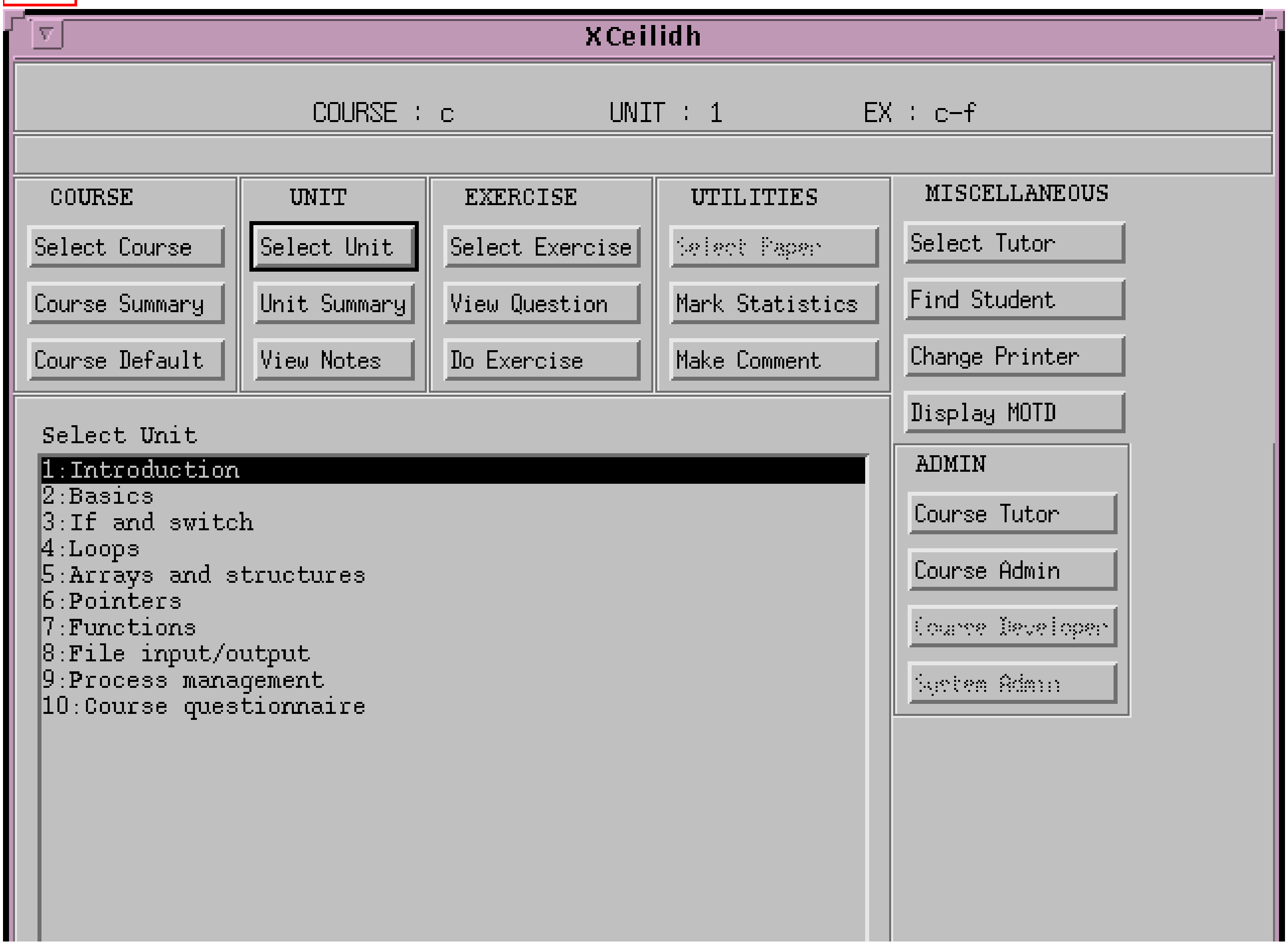
or

```
ceilidh (text based version)
```

Upon issuing the command xceilidh (or ceilidh) you will be greeted with the menu shown in figure .

Note: Example menus are shown in this document. Menus seen in practice may vary slightly from those shown, since the actual menu you are offered reflects only those facilities available at the time.

xceilidh is the X-Windows based version which you should use as it is easier and more intuitive. However, the text based version ceilidh does have a few more features. These are not that important though. I will list features of both. ceilidh uses abbreviations for commands. xceilidh has buttons to press




```
(text version is like this)
```

```
CEILIDH system level menu
 lc    list course titles          | sc   move to named course
 vp    view papers                 | pp   print papers
 clp   change printer              | h    for more help
 co    make a comment to teacher   | q    quit this session
 fs    find student                | ft   find tutees
============================================
System level command:
```

Fig. System Level Ceilidh Menu

This is the "system" level of Ceilidh and represents a department wide view of the system.

The commands which are available at this point are as follows.

lc

(text ceilidh only) in X the courses available appear on main window.This command tells you which courses are available and supported by the Ceilidh system, their full title and their abbreviation.

vp

(text ceilidh only) If you are interested, you can use the vp command to view various papers describing the workings of the Ceilidh system. A typical response to this command would be The stored papers are:

```
ASQA   : Automated Software Quality Assessment
CAL   : The Ceilidh Courseware System
CLI   : The command line interface ceilidh
Courseware : Courseware to support the teaching of programming
Install : Installer's Guide
Oracle  : The "oracle" output recogniser
Qu-ans  : The question/answer marking program
Student : Student Guide to Ceilidh
Teacher : Teacher Guide to Ceilidh
```

Choose a paper :

which lists a selection of the available papers. If you reply with the short name of the paper (the first word on the line), the paper will be shown on the screen a page at a time through a paging command such as "more". Diagrams may not appear correctly.

It is possible to print a given paper which looks interesting using the pp command. Some papers containing diagrams may not view or print nicely on devices without appropriate facilities.

h

(Help button) The h command offers a little more information on the significance of the different commands available to you in the Ceilidh system. This command is available at various points when you are using Ceilidh, and should give help relevant at the time.

q

(Exit button) This is the "quit" command to leave the Ceilidh system, and to return to your ordinary UNIX shell.

For courses with student registers, the following commands are also available

fs
: (Find Student Button) To find details about any student registered on any of the courses supported by the system.

ft
: (Find Tutor) To find details of the tutees of a specified tutor.

See below for discussion of the clp and co commands, both of which occur at many places in Ceilidh.

In general you will wish to move fairly soon to work on a specific course which you are studying. A particular course is entered using the sc (select course) command highlighting c course in window or by typing for example

```
sc
```

followed by return to enter the course "c" in text based version. (You must use lower case/upper case (small and capital) letters exactly as requested. If the given name is not a valid course, all available course names will be listed and a valid one should be selected.)

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next | Up | Previous
Next: The exercise level **Up:** Using Ceilidh as a Student **Previous:** Using Ceilidh as a Student

The course and unit level

In the X based version these are both in the main window

In the text based version: When you have selected a particular course, the menu shown in figure should now be displayed on the screen.

```
-------------------------------------------------------------------------
Course and unit menu for course "pr1" unit "1"
 lu      list unit titles           | su       set unit code
 lx      list unit exercise titles  | sx       move to named exercise (1)
 lux     list units and exercises   | state    current exercise state
 vn      view notes on the screen   | pn       print notes on letter13
 csum    read course summary        | usum     read unit summary
 vm      view all marks             |
 clp     change printer             | h        for more help
 co      make a comment to teacher  | q        quit
==========================================
Unit command:
-------------------------------------------------------------------------
```

Figure 17.1: Unit and Course Level Ceilidh Menu (text version only)

This menu is identical whether it is obtained from the system level of Ceilidh using the sc command, or by entering Ceilidh with a `-c` argument.

We are now in a chosen course. The various possible commands have the following significance.

lu

(text only - list unit :the units are automatically listed in X version once the course has been selected)

Each course is divided into a number of units, rather like the chapters of a book. This option lists the name of each unit, giving you a brief outline of the course as a whole. Typical output might be

```
Units in course pr1
  Unit 1: Background
  Unit 2: Elementary programming
  Unit 3: Conditionals
  Unit 4: Loops
  Unit 5: Functions
  Unit 6: Miscellany
  Unit 7: Arrays and structures
  Unit 8: File input and output
  Unit 9: Pointers
```

lux

(text only) This command lists all units and exercises within these units.

csum

(Course Summary button) If the teacher remembers to keep the information up-to-date, this command gives you a summary of the timetable for your course, with

details of the courseworks to be set, and the hand-in dates for each one.

state

(text only - See message of the day MOTD) As a course progresses exercises are opened, made late and then closed. This command gives a summary of the state of each exercise.

Select Unit Button (su)

This command enables you to select a chosen unit of the course. The menu remains the same, apart from the currently selected unit number which is included at the top of the menu. Commands below which relate to a specific unit use the currently selected unit number.

Unit Summary Button (usum)

This will list a brief summary of the currently selected unit, usually at the level of section headings in the notes.

View Notes Button (vn)

This command (view notes) allows you to view on-line the notes for the current unit of the course.

q

(text only) This is the command to quit the system. If you entered Ceilidh at the course level with a command such as

```
ceilidh -c pr1
```

the quit will return you to your shell. If you entered the course level from the system level using first

```
ceilidh
```

and then

```
sc pr1
```

for example, the quit returns you to the system level of Ceilidh, and you will need another quit to return to your shell.

Your current unit and exercise will be noted, so that when you re-enter Ceilidh, you will default to the same unit and exercise as when you left. If you wish to quit without saving your current state, use q!

instead.

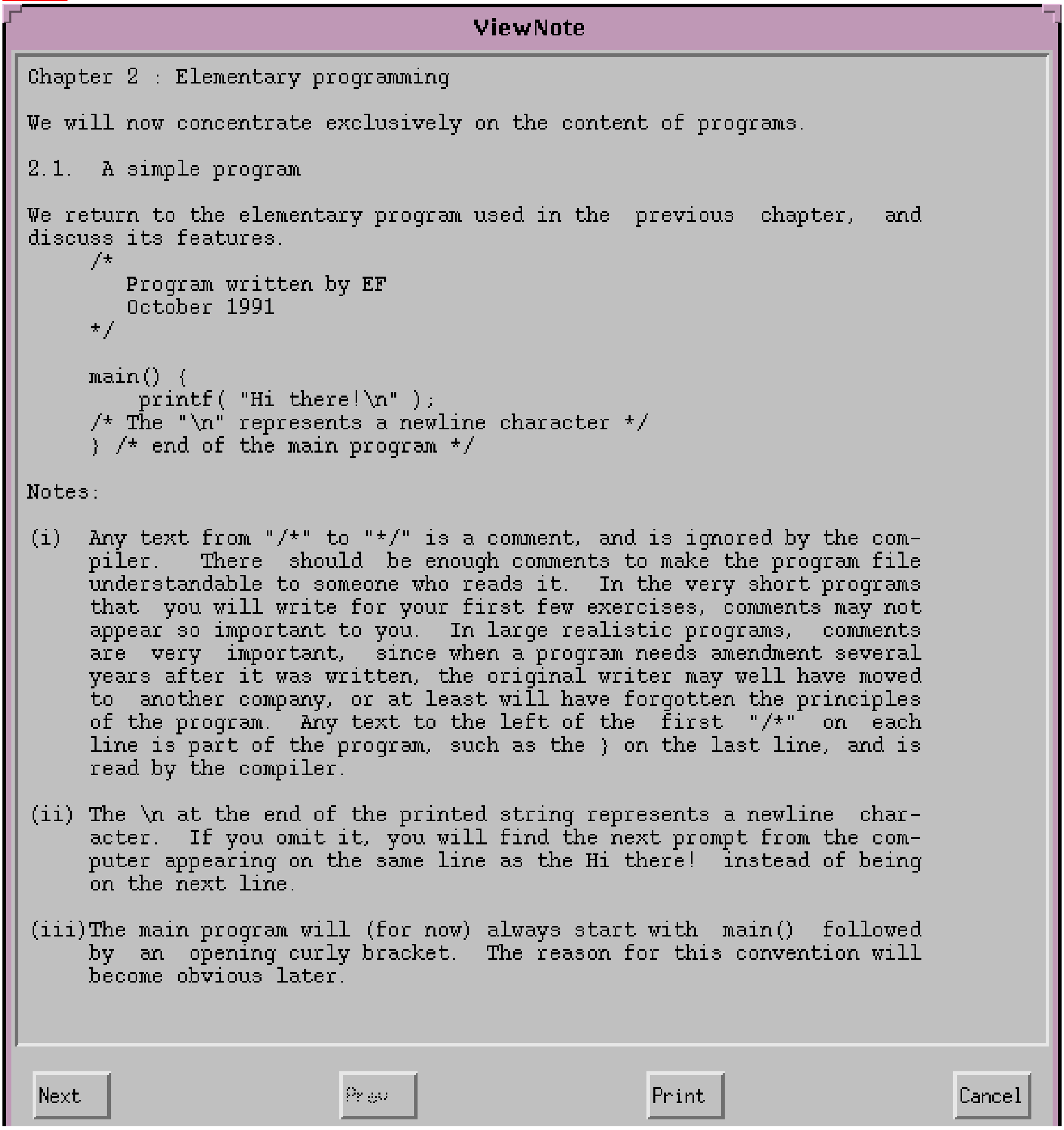
Make Comment Button (co)

At many points in the Ceilidh system, the system allows you to make comments to the course teacher. Comments are always welcome. Comments may be a request for help ("What do you mean by .... in this week's question?"), a criticism of the system ("I think the mark it gave me was not fair"), or an apology for the late hand-in of work ("Sorry but I had an examination ..."). Please feel free to use this facility; the teacher will try to answer most queries. The comments are sent using email to the teacher in charge of the course.

Change Printer (clp)

- may not work !! Whenever you use a command which involves printing some information, the computer chooses the printer which it thinks is most convenient. This is done by looking at where you are on campus. Sometimes the computer chooses the wrong printer (it cannot always tell exactly where you are on a network), so there is a facility for you to choose a particular printer by name. You will be told appropriate printer names in class.

To work on your coursework, you will need to move from the "unit" level of Ceilidh into the "exercise" level.




```
Chapter 2 : Elementary programming

We will now concentrate exclusively on the content of programs.

2.1.  A simple program

We return to the elementary program used in the  previous  chapter,  and
discuss its features.
     /*
        Program written by EF
        October 1991
     */

     main() {
         printf( "Hi there!\n" );
     /* The "\n" represents a newline character */
     } /* end of the main program */

Notes:

(i)  Any text from "/*" to "*/" is a comment, and is ignored by the com-
     piler.   There  should  be enough comments to make the program file
     understandable to someone who reads it.  In the very short programs
     that  you will write for your first few exercises, comments may not
     appear so important to you.  In large realistic programs,  comments
     are  very  important,  since when a program needs amendment several
     years after it was written, the original writer may well have moved
     to  another company, or at least will have forgotten the principles
     of the program.  Any text to the left of the  first  "/*"  on  each
     line is part of the program, such as the } on the last line, and is
     read by the compiler.

(ii) The \n at the end of the printed string represents a newline  char-
     acter.  If you omit it, you will find the next prompt from the com-
     puter appearing on the same line as the Hi there!  instead of being
     on the next line.

(iii)The main program will (for now) always start with  main()  followed
     by  an  opening curly bracket.  The reason for this convention will
     become obvious later.
```

Fig. Example of X version view notes window

Next Up Previous
Next: The exercise level **Up:** Using Ceilidh as a Student **Previous:** Using Ceilidh as a Student

---

Dave.Marshall@cm.cf.ac.uk

Wed Sep 14 10:06:31 BST 1994

The exercise level

If, for a given coursework, you are asked to solve a nominated coursework exercise in a this week's unit of the course, you will perhaps first select the appropriate unit using the, Select Unit (su) command, then list the names of all the exercises in this unit appear in the main window (or using the command

```
lx
```

at the course/unit level, and then enter the required exercise using

```
sx 2
```

for example, to select exercise 2 of the current unit.)

In X highlight the exercise you want and press Select Unit button.

IN X: Do Exercise moves you to a new level and a new window: The exercise level.

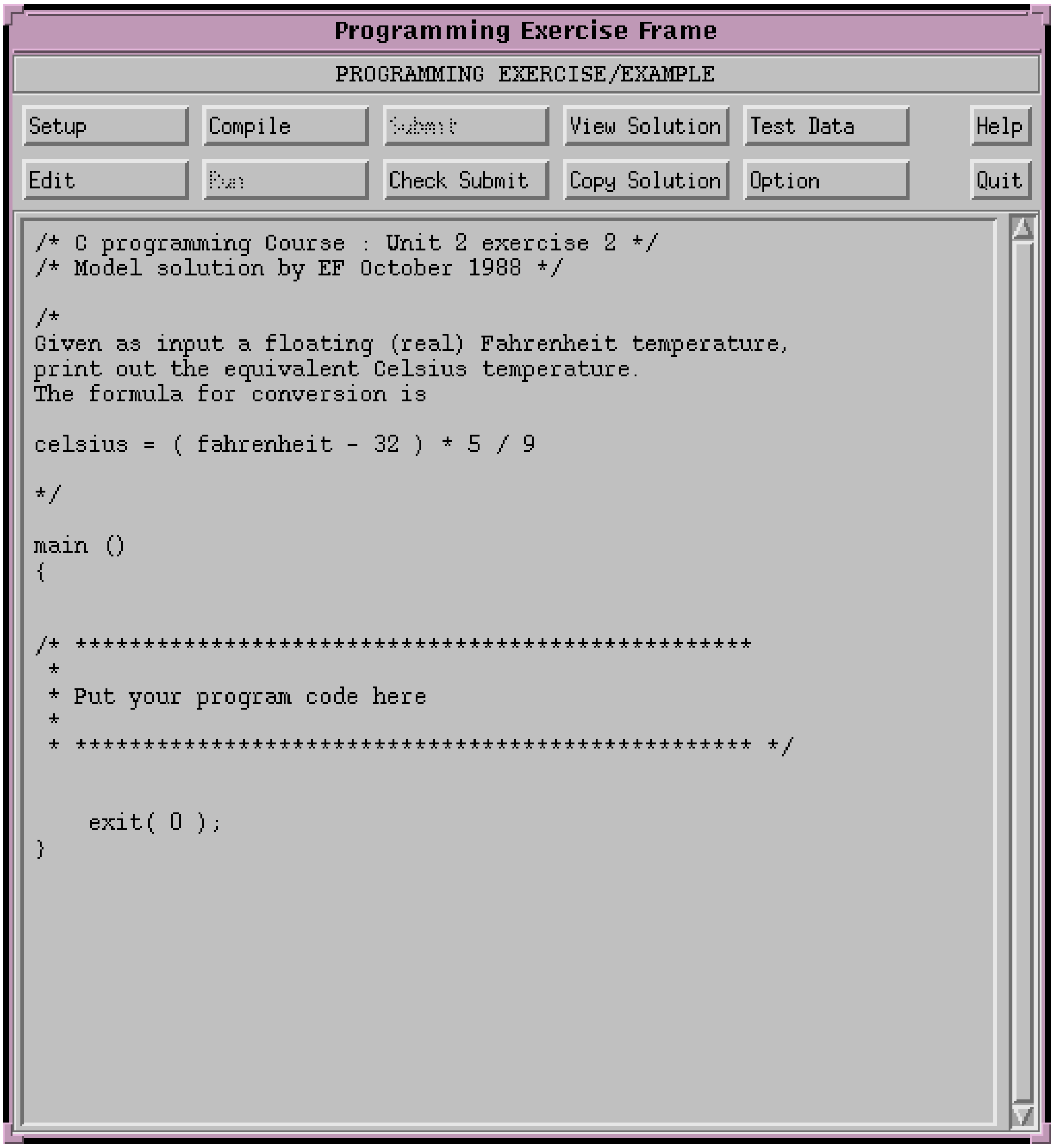
IN TEXT VERSION: It is worth noting that at the course level, while the sx (select a particular exercise) command moves you to another level, the "exercise level" with another menu, the su (select a unit) command leaves you at the course level with the same menu. You can move around the different units in a course at will without changing your level in the system. To attempt exercises you must enter the exercise level, which has different menus depending on the type of exercise you are asked to complete. These exercises include compiled language exercises, interpreted language exercises, question/answer exercises and text submission (essay) exercises. For the moment will will consider the compiled language exercise menu.

If you type

```
sx 1
```

to select exercise 1 in the current unit of the course you will see the menu given in Fig. .

This is the level at which most of your work will be undertaken. Each exercise will have been set up by the teacher, and will include a question, a skeleton solution, and all the necessary testing information.



(text version ceilidh looks like this)

```
Compiled language menu for course "pr1" unit "1" exercise "1"
  vq     view question on the screen    | pq     print question on draft13
  co     make a comment to teacher      | set    set up coursework
  h      for context help               | H      for general help
  q      to return to calling menu      |
  ed     edit your program              | cm     compile your program
  cv     compile verbose                | cks    check whether submitted OK
  run    run your executable            | rut    run yours against test data
  sub    submit/mark your program       | std    look at the test data
  vs     view solution program          | ps     print sol'n program on draft13
  cp     get copy of solution           |
  rex    run solution executable        | rxt    run sol'n against test data
 ===========================================
 Type compiled language command:
```

Fig. Exercise Level Ceilidh Menu

Your normal sequence of activity at this level might be as follows. First use view question (vq) to look at the question, or print question (pq) to print it out. You may need to study the question for a while before attempting its solution on the computer. It may be sensible to view or print it at least a day before the laboratory session during which you solve the problem.

You will then use

```
        setup (set)
```

to set up a skeleton solution. This command typically puts an outline of the required program into your directory, to give you a flying start in solving the problem. In more complex exercises later in the course, it may set up other data files as well.

A textedit window will be brought up to edit it.

At this stage you can start to develop your program, using the commands

Edit (ed)

> to edit your program,

Compile (cm)

> to compile it (if the compilation fails, go back to ed to correct the error with the editor, and then try compilation again), and

Run

> to try running your program. It is up to you to think of appropriate tests when running your program, to convince yourself that it is running correctly.

cv

> (Text only - see Options button in X Version to set verbose compilation) This command is given as an alternative to the cm command. When used it will compile your program more verbosely, giving compiler
>
> warning messages which can help identify problems in your solution.

db

> (text only) If this option has been set up by the course developer, it offers debugging facilities to you.

Note: Not all of the options in the menu will appear on the at all times; if there is no executable, for example, the running options will not appear or appear ghosted in XCEILIDH. If you have not executed set to obtain an outline program, the ed command for editing your program will not be shown.

Once you have successfully compiled your program and tested it to your satisfaction, the system is ready to mark and submit it. It does this by looking at your program source code (checking that it is indented correctly, for example), and running your compiled program against various sets of test data and seeing that it produces the correct results. At this stage you may wish to use the following commands.

rut

> (text only - see OPTIONS button to set this in X version) This runs your compiled program against the first set of test data used by the marking process, and enables you to see whether it appears to produce sensible answers.

std (show test data)

> This shows you each set of test data being used by the marking process. The teacher reserves the right to change the test data at any time, since your program should generally work on absolutely any data which it receives.

When you have performed enough tests to convince you that your program is correct (and only then) you should ask the system to mark and submit it using the

```
        submit button (sub command).
```

The computer's response will be something like that shown in figure .

```
---------------------------------------------------
        Analysis of Dynamic Correctness
                                   item   mark   out of
                            Simple test
                      Negative distance
                     Check "feet" "ins"
                            Inches > 12
                        Negative inches
        Score for Dynamic Correctness is       .0%

        Mark summary
                               category   mark   out of
                    Dynamic correctness
                  C++ typographic style
                 C++ complexity measure
                   C++ program features
        Overall mark awarded

---------------------------------------------------
```

Figure 17.2: Output from the marking command

The significance of this output is as follows.

Firstly your compiled program is run against several sets of test data. The system looks in the output generated by your program for evidence that you have produced the correct answer; this can be a non-trivial operation if your program does not print its results clearly! Each test produces one line of output, giving you a brief summary of the test, and the score you have been awarded. Different tests will be marked out of different totals, depending on the importance of the test.

The marks from these runs against test data are then combined into a single "dynamic test" result for your program. This result is then scaled out of a particular value, and the next few lines give marks for various "static tests" (tests performed by looking at your program source, rather than by executing it) such as "typographic style" (your program layout, choice of identifiers, use of comments, etc, see the ASQA paper[1] for details, a copy is stored on the Ceilidh system) "complexity" (the complexity of your program is compared with the complexity of the course developer's model solution; the two should not differ by too large a factor) and lastly "features" (the computer looks for specific good or bad programming features associated with this particular coursework).

All these marks are then combined with their weightings into a single mark which you are awarded. The Ceilidh system retains a copy of your program and of the mark awarded for future reference.

If you are happy with the mark awarded, you can quit at this stage. Alternatively, you may try to improve your mark and try again. It is your last mark which is recorded as your actual mark for this coursework.

To check that the mark has been correctly stored by the computer, use the command

```
check submission  button (cks)
```

which will show you what the computer has recorded. You should always use this checking facility after every exercise.

There is also a command at the course/unit level vm which lets you view ALL your marks submitted so far.

Note:

- Do not waste hours trying to obtain an extra mark ar two. It is a misguided waste of your time. Once you have achieved a good overall mark, leave the Ceilidh system and work on your other courses!
- Do not use the system to find bugs in your program. Design and test your program thoroughly yourself before you submit it to Ceilidh for marking.

Other commands at this level are:

View Solution (vs), Print Solution (ps), Copy Solution (cp)

: These commands are available only after the hand-in date of the coursework, and let you view the solution (vs) to the coursework, print the solution (ps), and copy the solution into your own directory (cp) so that you can try it out yourself.

rex, rxt

(text only - see OPTIONS Button): These commands allow you to run the course developer's compiled program interactively (rex) to see that it works the way you expected, and to run it against the first set of test data (rxt) to see the output which it gives. This may give you ideas on how to layout your output. These options may not exist if there is insufficient space on the disc for the teacher to store executable versions of all the solutions.

When you quit (q) from the exercise level of Ceilidh, you return to the course level of Ceilidh, where you may perform other activities, or execute another quit to leave Ceilidh completely.

Next Up Previous

Next: Interpreted language exercises **Up:** Using Ceilidh as a Student **Previous:** The course and unit level

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Question/answer exercises **Up:** Using Ceilidh as a Student **Previous:** The exercise level

Interpreted language exercises

The menu and process for interpreted language exercises is similar to the compiled language menu described in the previous section. The compilation commands are, of course, excluded.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Question/answer exercises

The exercise level menu for these exercises is completely different from that of the Compiled Language menu shown above.

For Question/Answer exercises you are given the following menu.

```
Question/answer exercise menu for course "tst" unit "1" exercise "qu":
vq     view questions                     | pq      print questions
ans    answer questions and submit        | cks     check submitted
h      help                               | q       return to calling menu
Type question/answer command:
```

The X windows one is similar We will not use this much in our course.

The options have significance as follows.

vq
: This allows you to view the questions before attempting to answer them. The pq command can then be used to obtain a printout of these questions.

ans
: When you are happy you know the answers to the questions set, you can enter your solutions using the ans command. This will then ask you the questions one at a time and read your response. Answers may be a choice between a few options, a word or a short sentence. To quit the exercise before answering all the questions type q as your answer.

cks
: This command allows you to check that your mark has been submitted correctly, and to check your answers.

Some question/answer exercises are purely for collecting answers, such as those to the end-of-course questionnaire. Other will involve answers which are marked. The questions should make clear which of these cases holds.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

The command line interface (TEXT CEILIDH ONLY)

This is a completely new interface in which, instead of using menus, each Ceilidh facility is represented by a UNIX command. It can be used on any terminal. Because there are no menus in this system, it is recommended that you use it only after some experience of the menu system.

To use this facility, there are two things you must do. First execute

```
~ceilidh/bin.cli/set.env
```

to set up an appropriate environment. You will need to check with your teacher just where the ~ceilidh directory is on the machine. This needs to be done once only (unless at a later stage you wish to reset your environment).

In order to use these commands, the directory containing them must be included in your PATH variable. To do this, type

```
source ~ceilidh/bin.cli/source.csh
```

at the start of each logged-on session during which you wish to use Ceilidh.

From here on, type

```
commands
```

to get a list of Ceilidh commands currently available, or

```
status
```

to show the currently set course, unit and exercise. The commands follow generally the pattern of the menu commands, but a few have had to be renamed to avoid clashes with existing commands. A typical starting sequence might be

```
Command          Purpose
```

```
commands       See commands available
set.cse pr1    Select course "pr1"
commands       See extra course commands
lu             List unit titles
set.unit 4     Set a particular unit
lx             List exercise titles
set.ex 4       Select exercise to solve
vq             View question
setup          Set up program skeleton
ep             Edit program
cm             Compile program
run            Run program
sub            Submit
cks            Check submitted
```

- Advantages of the command line interface
- General points

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: General points **Up:** The command line interface (TEXT CEILIDH ONLY)
Previous: The command line interface (TEXT CEILIDH ONLY)

Advantages of the command line interface

With this interface, you can execute other non-Ceilidh commands or even log out at any point. When you resume, the course, unit and exercise will remain set just as when you last issued a Ceilidh command (although you may choose to execute "status" to check the settings). This interface will be particularly useful for the "pr2" course, in which you need to perform all compilations yourself.

With this interface there is never any need to use "q" to quit the various levels of Ceilidh.

At any time, type

```
commands
```

to remind yourself of the commands currently available. The command

```
status
```

shows the currently set course, unit and exercise.

Typing

```
~ceilidh/bin.cli/set.env
```

will clear out the currently set values for course, unit and exercise. You will then need to use "set.cse", "set.unit" etc to reset them to the values you require.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Conclusions **Up:** The command line interface (TEXT CEILIDH ONLY)
Previous: Advantages of the command line interface

General points

At certain times, the teacher may close a complete course, or a unit, or an exercise. These perhaps represent parts of the course which are under development, or which must be kept unmodified for administrative reasons.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: How Ceilidh worksCeilidh Course Notes, User **Up:** Ceilidh - On Line C Tutoring System **Previous:** General points

Conclusions

The Ceilidh system is an essential part of your learning process; learn to make good use of it.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: References **Up:** Ceilidh - On Line C Tutoring System **Previous:** Conclusions

How Ceilidh works, Ceilidh Course Notes, User Guides etc.

- Ceilidh Licence Details
- Ceilidh papers --- How Ceilidh works, marks etc.
- Ceilidh C Course Notes --- Alternative to what you have been reading.
- Ceilidh User Guides

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Common C Compiler Options **Up:** Ceilidh - On Line C Tutoring System **Previous:** How Ceilidh worksCeilidh Course Notes, User

References

1. Abdullah Mohd Zin and Eric Foxley, "Automatic Program Quality Assessment System", Proceedings of the IFIP Conference on Software Quality, S P University, Vidyanagar, INDIA (March 1991).

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

About this document ...

Hands On: C/C++ Programming and
Unix Application Design:
UNIX System Calls and Subroutines using C,
Motif, C++

This document was generated using the **LaTeX**2HTML translator Version 97.1 (release) (July 13th, 1997)

The command line arguments were:
latex2html `-split 3 -no_navigation C`.

The translation was initiated by Dave Marshall on 1/5/1999

Dave Marshall
1/5/1999

Next Up Previous
Next: The Minimum C Program **Up:** Programming in C **Previous:** Exercises - Using X WindowsEditing and

The C Program

In this Chapter we will look at the basic elements of C programming. We will firstly look at the basic C program structure and then how to compile and run programs.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

...NAME="84">

The Meta key is an abstraction of the X Window System which is usually `alt` on most systems. However some systems may not posses such a key. Apple Macintoshes use the *Apple* key instead, for example. On Sun Type 4 keyboards the Meta key is the diamond shape key *next to* the `alt` key (*not* the `alt` key). Local X implementation should be consulted for further clarification. In this book we will simply refer to the `Meta` key.

.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.

...together.

Even though we deal with UNIX and C nearly all the forthcoming discussions are applicable to MSDOS and other operating systems

.
.

Dave Marshall
1/5/1999

Next Up Previous
Next: Program Listings **Up:** C Standard Library Functions **Previous:** String Manipulation

Time

#include <time.h>

char *asctime (struct tm *time) - Convert time from `struct tm`to string.

clock_t clock(void) - Get elapsed processor time in clock ticks.

char *ctime(time_t *time) - Convert binary time to string. double difftime(time_t time2, time_t time1) - Compute the difference between two times in seconds.

st ruct_tm *gmtime (time_t *time) - Get Greenwich Mean Time (GMT) in a `tm structure`.

struct tm *localtime(time_t *time) - Get the local time in a `tm structure`.

time_t time(time_t *timeptr) - Get current times as seconds elapsed since 0 hours GMT 1/1/70.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Exercises

1. Write a program to print the lines of a file which contain a word given as the program argument (a simple version of `grep` UNIX utility).

   View Solution

   (unit8:File Input/Output:ex.grp)
2. Write a program to list the files given as arguments, stopping every 20 lines until a key is hit.(a simple version of `more` UNIX utility)

   View Solution
3. Use `popen()` to pipe the `rwho` (UNIX command) output into `more` (UNIX command) in a C program.

   View Solution
4. Setup a two-way communication between parent and child processes in a C program. i.e. both can send and receive signals.

   View Solution
5. Write a C program to emulate the `ls -l` UNIX command that prints all files in a current directory and lists access privileges etc. DO NOT simply `exec ls -l` from the program.

   View Solution
6. Write a C program to produce a series of floating point random numbers in the ranges (a) 0.0 - 1.0

   View Solution

   (b) 0.0 - n where n is any floating point value. The seed should be set so that a unique sequence is guaranteed.

   View Solution
7. Write a C program that times a fragment of code in milliseconds.

   View Solution

8. Write a program that will list all files in a current directory and all files in subsequent sub directories.

   View Solution

9. Write a program that will only list subdirectories in alphabetical order.

   View Solution

10. Write a program that shows the user all his/her C source programs and then prompts interactively as to whether others should be granted read permission; if affirmative such permission should be granted.

    View Solution

11. Write a program that gives the user the opportunity to remove any or all of the files in a current working directory. The name of the file should appear followed by a prompt as to whether it should be removed.

    View Solution

Next Up Previous
Next: Ceilidh - On Line C Tutoring System **Up:** UNIX and C **Previous:** Times Up!!

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

...c89

`c89` is the name of the Dec ANSI C compiler. Other compilers exist for example: `acc` - SUN's ANSI Compiler, `cc` - non-ANSI compiler, `gcc` - Gnu C compiler and whole host of proprietary compilers (`tcc` - TURBO C)

...together.

Even though we deal with UNIX and C nearly all the forthcoming discussions are applicable to MSDOS and other operating systems

...CEILIDH

A ceilidh (pronounced Kay-Lee) is an informal gathering for conversation, music, dancing, songs and stories. Concise OED.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

CEILIDH GENERAL LICENSE
Version 2.2, Feb 1994

1. This system is distributed with the proviso that it may not be used for
   commercial gain.

2. CEILIDH is not proprietary, but it is not in the public domain.
   The upshot of all this is that anyone can get a copy of the release
   and do anything they want with it (subject to condition 1 above), but no
   one takes any responsibility whatsoever for any (mis)use.

3. Any alterations to the code should be distinguished from the original
   system code.

4. The authors do not accept any responsibility in the use of the system
   and any consequences of using the system.

5. This system is provided as is and although every effort will be made to
   support the system, support cannot be guaranteed.

6. Any changes made to the code to overcome problems and/or tailor operation
   to the local site should be reported back to the authors at
   Nottingham University (email ltr@cs.nott.ac.uk).

THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.
SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY
SERVICING, REPAIR OR CORRECTION.

Please fill in the details below, sign and return a paper copy to the address
given at the bottom of the page.

Name .... Position ....

Organisation..

Signed .... Date ....

Neil Gutteridge, Department of Computer Science, University of Nottingham,
University Park, Nottingham, NG7 2RD, England

Ceilidh Papers

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Ceilidh Notes

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Ceilidh Guides

- General Overview of Ceilidh
- Student's Guide to CEILIDH
- Course developer's Guide to CEILIDH
- Installer's Guide to CEILIDH
- Question/answer exercises in Ceilidh
- Teacher's Guide to CEILIDH
- Tutor's Guide to CEILIDH
- Ceilidh System Changes

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next | Up | Previous

Next: Compiler Options **Up:** Programming in C **Previous:** References

Common C Compiler Options

Here we list common C Compiler options. They can be tagged on to the compiler directive. Some take an additional argument.

E.g.

```
c89 -c -o prog prog.c
```

The `-o` option needs an argument, `-c` does not.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Buffer Manipulation **Up:** Programming in C **Previous:** Compiler Options

C Standard Library Functions

Listed below are nearly all the ANSI C standard library functions.

The header file where related definitions are stored are given. These may vary on some systems so check local reference manuals.

A brief description is include with all parameter types. More info can be obtained from online man calls or reference manuals.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

String Manipulation

#include <string.h>

char *stpcpy (char *dest, char *src) - Copy one string into another.

int strcmp(char *string1, char *string2) - Compare string1 and string2 to determine alphabetic order.

char *strcpy(char *string1, char *string2) - Copy string2 to stringl.

char *strerror(int errnum) - Get error message corresponding to specified error number.

int strlen(char *string) - Determine the length of a string.

char *strncat(char *string1, char *string2, size_t n) - Append n characters from string2 to stringl.

int strncmp(char *string1, char *string2, size_t n) - Compare first n characters of two strings.

char *strncpy(char *string1, char *string2, size_t n) - Copy first n characters of string2 to stringl .

char *strnset(char *string, int c, size _t n) - Set first n characters of string to c.

char *strrchr(char *string, int c) - Find last occurrence of character c in string.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

About this document ...

This document was generated using the **LaTeX**2`HTML` translator Version 0.5.3 (Wed Jan 26 1994) Copyright © 1993, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

The command line arguments were:
latex2html `CE.tex`.

The translation was initiated by Dave.Marshall@cm.cf.ac.uk on Wed Sep 14 10:06:31 BST 1994

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Advantages of using UNIX with C **Up:** Programming in C **Previous:** Running Make

UNIX and C

There is a very close link between C and most operating systems that run our C programs. Almost the whole of the UNIX operating system is written in C. This Chapter will look at how C and UNIX interface together.

We have to use UNIX to maintain our file space, edit, compile and run programs *etc*. (Appendix).

However UNIX is much more useful than this:

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Exercises **Up:** UNIX and C **Previous:** Receiving signals - signal()

Times Up!!

The last topic we will at in this course is how we can access the clock time with UNIX system calls.

There are many more time functions - see `man` pages and handouts.

Uses of time functions include:

- telling the time.
- timing programs and functions.
- setting random number seeds.

`time_ttime(time_t*tloc)` - returns the time since `00:00:00 GMT, Jan. 1, 1970`, measured in seconds.

If `tloc` is not NULL, the return value is also stored in the location to which tloc points.

`time()` returns the value of time on success.

On failure, it returns `(time_t) -1`. `time_t` is typedefed to a long (int) in <sys/types.h> and <sys/time.h> header files.

`int ftime(struct timeb *tp)` - fills in a structure pointed to by `tp`, as defined in <sys/timeb.h>:

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Day light Saving time applies locally during the appropriate part of the year.

On success, `ftime() returns no useful value. On failure, it returns -1.`

`Two other functions defined` *`etc.`* `in #include <time.h>`

`char *ctime(time_t*clock), char *asctime(struct tm *tm)`

`ctime() converts a long integer, pointed to by clock, to` a 26-character string of the form produced by asctime(). It first breaks down clock to a tm structure by calling localtime(), and then calls asctime() to convert that tm structure to a string.

asctime() converts a time value contained in a tm structure to a 26-character string of the form:

Sun Sep 16 01:03:52 1973

asctime() returns a pointer to the string.

Example 1: Time (in seconds) to perform some computation:

Example 2: Set a random number seed

lrand48() returns non-negative long integers uniformly distributed over the interval (0, 2**31).

A similar function drand48() returns double precision numbers in the range [0.0,1.0).

srand48() sets the seed for these random number generators. It is important to have different seeds when we call the functions otherwise the same set of pseudo-random numbers will generated. time() always provides a unique seed.

Next Up Previous

Next: Exercises **Up:** UNIX and C **Previous:** Receiving signals - signal()

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

The Ceilidh System

A General Overview

Steve Benford, Edmund Burke, Eric Foxley
Neil Gutteridge, Abdullah Mohd Zin

Learning Technology Research
Computer Science Department
Nottingham University

email : ltr @ cs.nott.ac.uk

1. Overview

The three main areas involved in what we refer to as courseware are

- The administration of courses
  - Under this heading we include
    - Monitoring student progress
    - Monitoring overall course progress
    - Informing tutors of relevant information
    - Detecting defaulting students

- The assessment of student achievement
  - Marking student work in various forms such as
    - Computer programs
      - in various languages
    - Multiple choice questionnaires
    - Question/answer exercises
    - Essays or reports

- The presentation of information to students
  - The traditional role of CAL has been in the presentation of information to a student, with the speed of progress determined by the student, and with different routes being followed depending on the student's choice and on the system's assessment of the student's progress.

The Ceilidh project aims eventually to cover all these areas, but at present covers only the first two. The third area is being actively pursued as part of the current project, but at present the system gives administration and marking support for a course of lectures.

We must distinguish between the Ceilidh system itself, and the courses which run under it.

Use of the present system

The Ceilidh system has been in use at Nottingham since 1988,

assisting
in both C and C++ courses to classes of up to 160 students. It runs
on
UNIX systems.

It is now distributed to about 20 UK sites, and 4 overseas sites.

1.1. Marking and assessment

The original version of Ceilidh was developed at Nottingham mainly
for
marking programs written in C. The student would (on-line) read
this
week's question (ASCII text in a named file), obtain a skeleton
outline
of the solution program (and any associated header files etc), develop
a
solution program, and submit the program for marking.

The last two steps could be repeated, so that the student could
have
several attempts at a solution, with the system providing feedback
about
the weaknesses in submitted work.

The marking was done using a number of metrics

```
     o Static:
         program layout
           indentation
           choice of identifiers
         program structure
           use of denotations
           complexity metrics
         "lint" warnings
           suspicious constructs

     o Dynamic:
         run against test data sets
           or using shell scripts
         program output validated
           using anoracle
         efficiency monitored
```

Originally (when machines were not so powerful as now) the marking
was
done overnight; the student submitted the work during the day,
the
marking was done (and results emailed to the student) overnight.
This
implied at most one attempt per day.

Now we mark interactively. This provides instant feedback to the
stu-
dent on the mark awarded, and on the major areas where they have
lost
marks. The marks are made available also to the course teacher and
the
student's tutor, and the student program is stored for possible

future
reference.

1.2. Course administration

The system was then extended to assist the teacher more in the
adminis-
tration of the course, and to broaden the scope of the student
activi-
ties. Marks generated by staff can be entered by hand; these marks
may
be either amendments to existing marks (overriding the computer
assess-
ments), or marks for associated work (such as essays or reports)
which
would be marked by hand. In addition work in the form of
essays/reports
can be submitted on-line by the students (having been generated using
a
word processor), stored on the system, marked by hand on or off
the
machine, and the marks then entered by hand.

The teacher can then look at the mark statistics for a class,
exercise
or student, find who hasn't submitted (and perhaps email them
and/or
their tutors), look at overall class program metrics, and check for
pla-
giarism in submitted work. The overall metrics for a given exercise
are
useful in keeping the teacher in touch with the current performance
of
the class as a whole; this is more important when the teacher is
not
hand marking student work. The plagiarism pattern over a series
of
exercises can be significant; in general the known presence of
plagiar-
ism tests acts as a considerable deterrent to copying.

A tutor can look at the progress of tutees, and look at their
submitted
work.

2. The TLTP involvement

The original system was developed and used locally at Nottingham in
both
the Computer Science (supporting C++ teaching) and Mathematics
(support-
ing C teaching) departments. The Computer Science Department
then
received a grant from the local "Enterprise in Higher Education"
initia-
tive. This was used to support a student over the summer of 1992,
to
arranged the distribution of the system with C and C++ courses to

```
other
departments and sites.

A consortium of Polytechnics and Universities then jointly  produced
an
application for funds to the TLTP, and was successful in obtaining
fund-
ing for a 3-year project.  Most of the funding is  for  the
integration
into  Ceilidh of software which has been (or will be) developed at
vari-
ous sites using other funding arrangements.

3.  Current developments

The main areas of development taking place at the moment (including
TLTP
work,  the TLTP project year is given brackets, [1] should imply
availa-
bility in September 1993, [2] September 1994, [3] September 1995) are:

o    Course administration:
          Teacher information
              statistics [2]
          Teacher control of exercises
              open, late, close
          Course developer menus
              develop marking schemes

o    Other courses:
          Pascal [1] (Royal Holloway)
          Modula 2 [1]
          SML [2] (Heriot-Watt)
          ADA [2] (Lancaster)
          Software Engineering [3] (Lancaster)
          Alternative C courses
          A possible ULISP course
     A second C course has already been submitted.

o    Other platforms:

          X-windows [1] (Nottingham)
          PC network [1] (Lancaster)
          VAX VMS [2]
          PC network windows [2] (Lancaster)
          Macintosh (Loughborough)
          PC standalone

o    Other exercise types:
          Semantic assessment of answers [2]
          Essay marking [3] (Nottingham Trent University)
          Interpreted languages, support for
              PROLOG
              shell programming

o    Other information delivery systems:
          Hypertext system using an authoring language [2]
              (Manchester Metropolitan University)
```

Acrobat (Nottingham)
Cajun

4. The current state

The first system and course release, Release 1.1, was June 1992, before TLTP funding. It included the system, a C course, and a first C++ course. The second release 1.2 was in December 1992. The system changes involved mainly bug fixes, and an additional "C++ programming in the large" course. Both system releases were for dumb terminals.

5. Future developments

Future releases should be as above, plus anything else that people give us. There is much work to be done and incorporated.

6. Evaluations

Questionnaires completed by the students (every course has an on-line questionnaire as the last "exercise" of the course) do not necessarily produce unbiassed results. The results analysed so far indicate that they agree overwhelmingly system is helpful.

In addition to the questionnaires, we have had discussions with students at the end of each course. The general impression given is that they find the system very supportive.

Some students express concern at having difficulty in finding the last few percentage marks to raise their total from the mid-nineties to the high nineties. Others say that they use Ceilidh to develop their mark up to a certain figure (80%?) and then leave the problem completely.

7. Other considerations

Educational problems

There are many areas of educational interest.

o How much feedback should we give to the student ?

o How should we control the number of attempts?
    Minimum interval
    Maximum number
    Marks lost per attempt

Practice would be different if Ceilidh were being used for training/quality control in industry.

Reading programs

Some teachers attach importance to teaching programming through the reading of programs.

The skeleton programs from which the students start provide this experience. The skeletons for some exercises form an almost complete program; they may provide a complete module for which a linking module has to be written. All this gives good experience in reading code.

In addition, a number of exercise themes follow through the course, and students are obliged to read and develop code they wrote earlier in the course.

Availability

We would like to see a wider availability of

- o platforms
- o courses
- o interfaces

Distance learning

There are a number of distinct scenarios for the use of Ceilidh.

(i) Centralised.
All processes run on a central system. Students perform their development in the system under Ceilidh, so that Ceilidh can monitor all processes. This is the way we run the early stages of our introductory course.

(ii) Local.
We use a central system, but program development is not necessarily done under Ceilidh. The student reads the question and obtains a skeleton answer under Ceilidh, leaves Ceilidh to develop a solution, and returns to Ceilidh to submit and mark. This is the system we use in our second C++ course.

(iii) Close coupled.

A close coupled system relies on the use of a network. All of the notes, questions and marking are centralised, but student development work is done on the peripheral machines. These machines could be UNIX systems, or based on other platforms such as PCs or Macintoshes.

(iv) Loose coupled.
We could have a loose client-server approach, in which the server supplies the requested information and marking facilities, but does not control or co-ordinate the marks.

(v) Standalone.
There are no centralised functions, and thus Ceilidh becomes a self-paced self-teaching tool. There are no secret solutions, or overall marking and control.

Testing operator practice

At the moment, the marking process runs from input, through a program, checking the resulting output for validity. The program is the item under test.

It is possible to imagine a scenario in which the program is fixed, but the student's task is to create the input to it.

- The program might be a database; the problem might be accessing certain information.

- The program might be a reservation system; the problem is to perform a certain series of transactions.

All this could be monitored by a modified version of Ceilidh, and marked. It could be used to assess trainee computer terminal operators.

Courses completely assessed by Ceilidh

Some early C and C++ courses at Nottingham are completely assessed by Ceilidh. The results of the Ceilidh exercise assessments are weighted, scaled and submitted as the returned mark for the module. There are certain implications when this happens.

(i) A complete dump of the file system is taken at the end of the course, to be made available to external examiners, or in the event of an appeal. Any later changes to the on-line information will not affect the archive.

(ii) The marks and submitted programs must be made available to the student. The students cannot then claim that "That isn't the version I submitted". There are two Ceilidh commands to assist in this: "cks" allows students to check all of their submitted marks and programs as seen by the system; and "vm" lets them view their marks, complete with weighting and scaling factors which will be applied as part of the examining process.

8. Student participation

Students are encouraged to send comments to the teacher at many points in the system. These comments vary from

I think my program is perfect, why doesn't Ceilidh give it 100%

I'm sorry it was late, I was unavoidably detained.

It is hoped to develop a help desk within the Ceilidh system.

References

1. Steve Benford, Edmund Burke, and Eric Foxley, "A System to Teach Programming in a Quality Controlled Environment", The Software Quality Journal 2, pp.177-197 (1993).

2. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and Abdullah Modh Zin, "Early experiences of computer-aided assessment and administration when teaching computer programming", Association for Learning Technology Journal 1(2), pp.55-70 (1993).

3. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and Abdullah Modh Zin, The Ceilidh Courseware System, Proceedings of the International Conference on Computer Technologies in Education 1993, Kiev, Ukraine, September 1993.

4. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and
Abdullah Modh Zin, Experience using the Ceilidh System, Proceedings
of the All Ireland Conference on delivering the Computer Curricu-
lum, Dublin, September 1993.

5. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and
Abdullah Modh Zin, Integrating Software Quality Assurance into the
Teaching of Programming, Amsterdam"".if 1>0 , Proceedings of the
CSR 10th Annual Workshop "Applications of Software Metrics and
Quality Assurance in Industry", Amsterdam, 1993.

6. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and
Abdullah Mohd Zin, Ceilidh: A course administration and marking
system, Proceedings of the International Conference on Computer
Based Learning in Science, Vienna, December 1993.

7. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and
Abdullah Mohd Zin, Experiences with the Ceilidh System, Proceedings
of the International Conference on Computer Based Learning in Sci-
ence, Vienna, December 1993.

8. Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and
Abdullah Mohd Zin, CEILIDH - a management and marking system,
Assessment in Higher Education Computing, York, June 1993.

AUTOMATIC PROGRAM ASSESSMENT SYSTEM

Abdullah Mohd Zin
Dr Eric Foxley

Department of Computer Science
University of Nottingham
Nottingham UK

Abstract

The problem of assessing the quality of a program is a difficult but important task. This paper discusses a number of computer-implementable approaches to the problem, and describes a system in operation at Nottingham University for automatically assessing the quality of student coursework. The system also provides feedback to the teacher on the overall strengths and weaknesses of the work being submitted.

1. INTRODUCTION

Program quality assessment is of fundamental importance in any area of software quality control. Tools for assessing quality are useful in two distinct areas, both to assist an individual programmer to improve the quality of his/her programs in a systematic way, and to help a manager to maintain quality controls and uniform standards for a project team. Such tools are also of use in two comparable roles in education. Since we are anxious to teach the concepts of software quality control, the availability of a tool would help students improve their own software quality, and would give them the experience of working in a quality controlled environment; this corresponds to the use of tools by programmers above. In addition, these tools can again be of use to the management, in this case the teacher, but now in two distinct ways.

Firstly, such tools can provide valuable feedback to the teacher about the strengths and weaknesses of the class as a whole, and indicate an emphasis which should be made in the continuing teaching.

Secondly, these tools can assist in marking student work, normally an arduous task performed unsatisfactorily by hand. In some

universi-
ties, an introductory or intermediate course in programming may be taken
by as many as 200 students. During the course, every student may be
required to submit one or two programs every week. The marking is nor-
mally done by asking the students to submit printed listings of the pro-
grams and of the results obtained from running them. These papers are
then handled to various graders who mark them. This type of arrangement
is unsatisfactory for several reasons.

(a) With large classes, the volume of papers to be handled is incon-
veniently large.

(b) The programs are not really tested, so we cannot be sure whether
the results obtained are the real results produced by the programs.

(c) If several people are involved in the marking process, it is diffi-
cult to maintain a uniform marking standard.

In this paper, we discuss various approaches to program assessment
which have been published in relation to the development of an automatic
program assessment system. We then discuss the approach which has been
taken at Nottingham to develop our system, called analyse.

2. SOFTWARE QUALITY FACTOR

Any assessment system should provide information in two distinct areas.

(a) It should provide an overall measurement of the quality of a pro-
gram.

(b) It should give comments to indicate the areas where the quality of
the program can be improved.

Before we can measure the quality of the program, we have to have a
clear understanding of what is meant by program or software quality.

The most common way of defining software quality is by looking at
many different factors which affect quality. The overall quantification
of software quality can then be performed by measuring a large number of

factors separately and combining them. McCall et al[19] have proposed a
useful categorisation of factors that affect software quality. These
software quality factors are grouped to focus on three important aspects
of a software product:

(a) product operations: correctness, reliability, efficiency, integrity
and usability;

(b) product revision: maintainability, flexibility and testability;
and

(c) product transition: portability, reusability and interoperability.

The factors proposed by McCall are very comprehensive. However,
the relative importance of different factors may vary in different
environments.[32] For example, in a small software project, Burgess pro-
posed that only four factors should be considered.[4] These factors are:
correctness, maintainability, usability and efficiency. However, the
definition of maintainability as proposed by Burgess is the combination
of maintainability and flexibility in McCall's software quality factors.

In our present study, we will consider only the factors of correct-
ness, maintainability and efficiency. Correctness is of course very
important because the main aim of programming is produce a correct and
working program. Maintainability is also very important because all
programs will have to be maintained. At the moment, the cost of mainte-
nance activities in a software engineering project forms a very high
proportion of the total costs, 40% according to Boehm,[3] 50% according
to Glass and Noiseux[8] and 67% according to Zelkowitz.[33] Program
dynamic efficiency (execution speed) is a desirable goal because it

normally reflects the type of algorithms which has been used in the pro-
gram.[23]

3. MEASURING PROGRAM CORRECTNESS AND EFFICIENCY

The most commonly used method to check program correctness is program testing. Since program correctness testing involves execution of the program, such tests can also be used to measure program efficiency. We will discuss the approaches which can be taken to measure program correctness and program efficiency based on program testing.

3.1. Types of program testing

Program testing can be divided into two types: static analysis and dynamic testing.

Static analysis involves the examination of program source code without execution. The program source code structure and syntax are inspected so as to highlight static errors and produce statistical information for the programmer.[5] Although compilation is a form of static analysis, the term `static analysis' is normally used for activities intended to pick up other type of errors or potential error conditions, such as infinite loops, unreachable statements, conflicting conditions, improperly nested loops and unused variables.

Dynamic testing involves executing the program by using test data. The main aim of dynamic testing is to uncover execution errors in a program. The most ideal dynamic testing would be to test a program with all possible elements from the input domain, but this is obviously impossible in any real situation. Thus testing can be done in practice only by using a small subset of data from the possible input domain. To ensure that as many as possible of the potential errors can be detected the test data must be chosen carefully. Many techniques for selecting test data have been proposed for example by using Data Flow Information,[24] Cn coverage measures,[20] TERn measures,[31] and boundary-interior testing.[14]

Running a program against test data will produce output. Most testing processes are based on the assumption that an oracle is present.

An oracle is a mechanism by which the correctness of the output can be
checked.[30] Construction of an oracle is a non-trivial problem in any
situation where the program output format is not exactly specified. In
a simple student problem such as `write a program to read a number of
centimetres and print the equivalent distance in feet and inches', the
number of possible outputs for the same input data include examples such
as

```
1 foot 3.6 inches
1 ft 3.59 ins
one foot four inches
ins 3.6 ft 1
```

All these must be correctly interpreted by the oracle.

In some examples, we may be testing a procedure rather than a com-
plete program. The problems of an oracle still exist, but are consider-
ably simplified.

Another possible output from dynamic testing is some information
about the program execution. This information, called a program profile,
can be useful for the programmer. For example, it can be used for iden-
tification of dynamically dead code, checking the correct number of loop
iterations and to help in the optimisation of the most frequently exe-
cuted code segments. It cannot, of course, distinguish between code
which is dead simply because the chosen test data does not call for its
execution, and code which is logically non-executable.

The testing process thus involves the following activities:

(a) program compilation;

(b) static analysis;

(c) for each set of test data
- execute the program against the data
- compare the output with the expected result
- analyse the output of the program profile.

3.2. Measurements of program correctness

We have mentioned that the main aim of program testing is to uncover

errors, so that a measure of program correctness can be made by consid-
ering the number and condition of errors in the program.

One method which can be used is to invoke the concept of verifica-
tion level. Conway[6] for example, lists eight different verification
levels.

1 The program contains no syntactic errors.

2 The program contains no compilation errors, or system detected
faults during execution.

3 There exists a set of test data for which the program gives
the correct answers.

4 For several typical sets of test data, the program gives the
correct answers.

5 For carefully chosen difficult sets of test data, the program
gives correct answers.

6 For all possible sets of data which are valid with respect to
the program specification, the program gives correct answers.

7 For all possible sets of valid test data and all likely condi-
tions of erroneous input, the program gives correct answers.

8 For all possible input, the program gives correct answers.

The higher the position of a program in this verification level, the
better its quality is considered to be.

Another method of measuring program correctness is by assigning
appropriate weight to each of the activities in the testing process. The
result of each activity is evaluated and a score is given. So the meas-
ure of program correctness can be calculated as

dwisi

where wi and si are respectively the weight and score for activity i.

3.3. Measuring program efficiency

In the teaching of programming, we are generally not concerned with marginal efficiencies in program size or speed. However, it was felt that such concerns may need to be introduced for two reasons.

In most problems gross differences (orders of magnitude) in execution times or program size might need to be considered; an example on sorting should be expected to do better than an Oqn2w timing. These differences essentially reflect the use of a different algorithm. In addition, the teacher might wish to set particular exercises in which efficiency might be a major criterion.

It was therefore decided that any software quality assessment should include a measure of program efficiency. This is most easily based on dynamic profiling of student program, since dynamic profilers are already available for most language systems. These can be used to find the maximum execution count of any statement in the program, which is usually the key factor in running speed. Other aspects of efficiency such as the size of the compiled program are not currently included in our system, but may be added in due course.

4. MEASURING PROGRAM MAINTAINABILITY

Software maintenance is a broad-based activity, and a prerequisite activity is to understand the software to be maintained.[32] Based on this fact, one common method to measure maintainability of a program is to measure its understandability.

There are few models which have been proposed so far for measuring a program's understandability.

4.1. Complexity model

The first proposal is based on the complexity measure. In relation to a fixed problem, Van Verth proposed that the program understandability is in "inverse" relation to its complexity.[29] To implement this we would thus need to measure the program complexity.

Several software complexity measures have already been proposed.

The first and most widely known measure is the one proposed by Halstead[10] called the software science. The Halstead measures are functions of the number of operators and operands in a program. The major components of software science are

n1 the number of distinct operators
n2 the number of distinct operands
N1 the number of operators
N2 the number of operands

Halstead shows that the overall program length N can be estimated as

N = n1logn1+n2logn2

and the program difficulty D as

2nD = qn1*N1___________

The second complexity measure is the cyclomatic number developed by McCabe.[18] McCabe considers the program as a directed graph in which the edges are the lines of control flow and the nodes are the line segments of code. The cyclomatic number represents the number of linearly independent connection paths through the program.

Halstead and McCabe measures treat a program as a single body of code. Henry and Kafura[13] present a measure which is sensitive to the decomposition of the program into procedures and functions. The measure depends on the size and the flow of information into procedures (the fan-in) and out of procedures (the fan-out). Henry and Kafura define the complexity of a program as

length*qfan-in*fan-outw

The program assessment system developed by Van Verth uses the complexity model as proposed by Oviedo[22] which involves the control flow measure and the data flow measure of the program. Both of these measures depend upon breaking down the program into maximal atoms called blocks, and then constructing the flow graph of the program treating the blocks as vertices.

4.2. Program difficulty model

The difficulty model, proposed by Bern[1] attempts to analyse how the dynamic portion of a program manipulates and controls the static elements.

The difficulty of each element in a program can be represented by assigning a weight to each syntactic element. These syntactic elements are items such as parameter, variable, array, function statement, subroutine etc. In addition, factors may be assessed for syntactic

attributes such as implicit definition, name in common, number of aliases, value changed, data type and dummy assignment. The various executable statement types in a language also have a hierarchy of difficulty of understanding, so that in the end each element can be assigned a weight that is representative of its difficulty.

One example of a software tool which is based on this model is called The Maintenance Analysis Tool[1] which analyses programs written in VAX-11 Fortran, a superset of Fortran 77.

4.3. Programming style model

The programming style model evaluates the program's understandability based on the style of the program. In this model, a program which has a better programming style is assumed to be more readable. The concept of programming style has been discussed by many people, for example by Kernighan and Plauger.[17]

The first effort to measure programming style was proposed by Rees[27] who described a Pascal source code style grader based on ten factors: average line length, comments, indentation, identifier length, use of label and gotos, blank lines, embedded spaces, modularity, variety of reserved words and variety of identifier names. Berry and Meeking[2] proposed a style grader for C, which calculates its results using similar factors to those of Rees.

In the approach proposed by Rees, the measurement of each aspect of the programming style is done as follows:

Diagram goes here, see printed notes

Figure 1

First, the score for each factor is collected from the program. Based on each separate score a mark is given. The calculation for the mark is based on the scheme as shown in figure 1, where the points have the following significance.

L: the point below which no mark is obtained.
S: the starting point of the `ideal' range.
F: the end point of the `ideal' range.
H: the point above which no mark is obtained.

Thus scores between S and F obtain maximum mark, those between L and S and between F and H are calculated by interpolation according to their exact position within the range, and those outside the range (L, H) receive no marks. The values for L, S, F, H and the maximum mark for each factor is given as part of the assessment program.

Another form of programming style model was proposed by Redish and Smyth,[25] where the marking of the programming style is based on a model program. They have tested their model by implementing an assessment system called AUTOMARK. AUTOMARK measures the programming style of FORTRAN programs by using 33 factors. These factors can be grouped as

follows: commenting (4 distinct factors), indentation (1 factor), block size (2), labels and formats (7), count of names and statements (6), array (2), control flow (7), blank lines (1), operator count (1), operand count (1) and parameterisation (1). The AUTOMARK marking[26] is based on the following six vectors:

F: the non-negative values of the factors computed for the model program
T: the non-negative tolerances for the factor values F
X: the maximum mark available for each of the factors
W: the non-negative weight assigned to the mark for each factor
L, G: indicators taking one of the three values -1, 0, +1

The calculation of mark for factor i is shown in figure 2.

Diagram goes here, see printed notes

Figure 2

5. The analyse program assessment system

In this section, we will describe the automatic assessment system used at Nottingham, called analyse. This system is intended to be used for marking students' C programs in an introductory or intermediate programming course.

5.1. Design considerations

In order to design a good assessment system, there are two points which have to be considered.

First, we have mentioned that the quality of a program is environment dependent, so that it is very important for the system to be able to adapt to the user's environment. For example in an academic environment, an teacher may, depending on the particular stage of the course being given, or the nature of audience, or the relationship of the course to the overall programme of instruction, place more or less emphasis on one measure or another. One week the emphasis might be on dynamic efficiency, another week on handling data errors. A user adaptable environment can be achieved by allowing the teacher to specify the values of certain variables (weights) which control the calculation of the quality of a program.

The second point is that all programs are written as a solution to a problem, and the way the program is written is dependent on the type of problem to be solved. The evaluation of the program quality must also take into consideration the problem specification. We choose to base some aspects of the evaluation on a model program supplied by the teacher; the teacher presumably thinks that this is the best solution which reflects the problem specification.

Apart from the (student) program to be analysed, the system must therefore also accept two more inputs, the set of values of the control variables defining the importance of different factors in the

assessment, and the model program produced by the teacher. For the purpose of dynamic testing, the system must also be supplied with sets of test data (as many sets as are deemed appropriate for testing this exercise) or test data specification (a range from which random data may be chosen) and the appropriate oracle.

The input and output of an assessment system is shown in figure 3.

Diagram goes here, see printed notes

Figure3

5.2. Computing program quality

There are five main components used in analyse to compute the score for program quality

maintainability,
structural weakness,
dynamic correctness,
dynamic efficiency and
program complexity.

The weights attached to the separate components are determined by the teacher. In addition, failure of the program to compile, or catastrophic failure during execution cancels all scores except that for maintainability.

5.2.1. Measurement of program maintainability

For maintainability measurement, analyse uses the programming style model. Following Oman and Cook,[21] we divide the programming style into two categories: those pertaining to the typographic arrangement, and those measuring the structural content of the code. The latter is used partly in this section, and partly in the section on program complexity below. Typographic style describes the way a program source code is presented. In analyse, factors belonging to the typographic arrangement include

% of indented lines
% of blank lines
average characters per line
average spaces per line
average module length
% of modules which have a good length
average identifier length
% of identifiers which have a good length
% of #define's
% of comments.

Since the typographic style is concerned with the program source

code presentation, it is independent of the problem specification. In analyse, the awarding of scores for each aspect of typographic style is done by using a similar technique to that proposed by Rees.[27] In order to allow the teacher to adapt each assessment to its environment, analyse allows values of L, S, F, H and the maximum mark for each factor to be defined separately for each assignment.

5.2.2. Measurement of structural weakness

For this section, we rely heavily on the standard Unix C utility lint, which comments on C program source code. For our static analysis, the score is based on the occurrence of static problems in the programs. These warnings include the following: variable declared but never used, variable assigned but never used, value returned by a function never used, value returned by a function sometimes used, statement not reached, and variable used before set.

5.2.3. Measurement of program dynamic correctness

For dynamic correctness testing, the score is awarded based on the correctness of the output when the program is run by using several sets of test data provided by the teacher. The oracle which is used to check the program correctness involves a number of regular expressions to define the structures which it expects to find in the student program's output. For each set of test data the teacher provides a set of regular expressions to recognise required features of the output. For example, in a simple example of a program to convert centimetres to feet and inches, if the correct output value is 3 feet 4.69 inches, there might be two regular expressions. One would search for

"3" followed by "feet" or "ft"

and the other for

"4.69" or "4.7" or "5" followed by "inches" or "ins"

The precise definitions are difficult to design; in the first example above, the "3" must be delimited by white space, if the number were "1" then "foot" should be an alternative to "feet". With experience, most of the regular expressions are now determined correctly. The oracle is much easier when we are testing procedures which compute values rather than programs which print output.

The score from a regular expression based oracle may be any value from 0 to 100%.

Each program would typically undergo several dynamic tests; each test would typically involve several regular expressions. Each test has an overall score given by the teacher, and within each test each expression has an associated weight.

In more complex cases, instead of using test data files, the dynamic tests are driven from shell scripts supplied by the teacher. This enables, for example, arguments to be supplied to the student's

program, or data to be piped between processes. The same technique of sets of regular expressions is used to analyse the output from the shell scripts.

5.2.4. Measurement program dynamic efficiency

During the execution of the program against sets of test data to check the correctness of the output, the program is profiled using the tcov system available on our SUN Unix machines. The programs are compiled with a special flag, and can then be run as many times as required against different sets of test data. The tcov output then gives

an execution count for every section of the program;
a summary of the sections with the maximum counts;
a summary of any unexecuted sections.

The results are compared with those for the model solution. Equivalent profiling systems are available for most languages on most Unix systems.

For the model solution, it would be expected that there were no unexecuted sections of code, since the sets of test data are devised using the tcov command to ensure that all code is executed at least once. There may be justified sections of unexecuted code in student programs if they have tested for additional possible invalid data sets.

We concern ourselves at the moment primarily with the maximum execution count. Other efficiency related measures may be considered at a later date.

5.2.5. Measurement of program complexity

This category includes static analysis of the frequency of occurrence of

gotos
reserved words
include files
operators
loops
conditional statements
assignment statements
function calls
complexity of expressions
library functions
literals
methods of types and data declarations

in the program. Each one is marked relative to the corresponding counts for the model program, and again with the scoring technique shown in figure 1.

5.3. Relative and absolute measures

The various factors involved in the maintainability measurements are absolute; they involve the measuring of a number of criteria concerned

with the program source.

The factors concerned with program complexity, on the other hand, are scored relative to a model program supplied by the teacher, in a way similar to the method proposed by Redish and Smyth[26] since the structure of a program depends on the type of problem to be solved. Before we can compare the structural style of two programs, we have to make sure that both programs are equivalent, i.e they are solving the same problem. We therefore consider the structural style only if the program analysed is equivalent to the model program, i.e. only if the program gives `correct' results, if the oracle has awarded more than some defined minimum score.

The nature of the scores for the major divisions can be summarised as follows.

maintainability: absolute
weaknesses: absolute
correctness: absolute, but depends on the particular problem and data
efficiency: relative to the execution of the model program
complexity: relative to the complexity of the model program

5.4. Implementation

The system is implemented under the UNIX operating system. It consists of two subsystems:

(a) student program assessment; and

(b) teacher facilities.

Before a particular problem becomes available to the students, the teacher must first use the teacher facility menu to collect together

the problem definition in English
the model solution program
sets of test data as required, each with a set of regular expressions for the oracle
the overall scores and weights for marking
the detailed components for each factor as in the graph of figure 1

All these files are kept in a single directory, and the name of this
directory must be specified when using the system. The files are all
collected into one directory using a menu system to ensure that all
items have been supplied, and that the model program functions correctly
on the chosen test data. The command runmodel is then run once only by
the teacher. This consolidates information about the program such as
its complexity measures and dynamic profile.

The command analyse can then be used by the student or the teacher
to assess the quality of any given program relative to the installed
model. The students normally work through a menu-driven system allowing
them options such as

read this week's coursework questions (typically two/week);
obtain a skeleton solution;
edit, compile, and run their program;
run the program against given test data;
run the model solution program against the same test data;
ask to see other similar coursework examples and solutions (typically eight/week);
ask for help from other on-line notes;
submit their program for assessment.

The test data referred to above is not necessarily the same as that used
in the quality assessment process; the dynamic correctness testing
should use data not previously seen by the student.

The teacher facilities subsystem provides facilities for the
teacher to set up the analysis system for a particular coursework as
described above. It also includes facilities for marking submitted work
in a number of ways. Mark sheets for the whole class are straightfor-
ward to produce. Of more interest is the production of the average
marks for the whole class in each of the basic scoring components. The
lecturer can thus be made aware that, for example, a significant number
of student programs are failing on a particular set of test data, or
show weakness in a particular structural aspect of their programs. The
teaching can then be reinforced or modified to remedy the weaknesses.

The system is written using a combination of Bourne shell scripts and programs written in C and awk. The static analysis uses lint and the compilation uses the C compiler provided on the SUN machines. We also rely on the SUN tcov program for profiling. The calculation of the typographic style is based on the program written by Berry and Meek- ing.[2]

6. CONCLUSION

The experience with analyse shows that the development of automatic assessment system is feasible and useful. This system can be a great help for anybody who has to assess the quality of programs, especially for teachers in programming courses.

However, the techniques currently used in analyse restrict its area of application in a number of ways.

Firstly, the use of an automatic testing technique to determine program correctness is not always suitable.

(a) The testing technique can never show the absolute correctness of a program. This fact is stated by Dijkstra as: "testing can show only the presence of errors, never their absence". We may there- fore be faced with a situation of coincidental correctness in which an incorrect program is considered to be correct because it appears to execute a particular set of test data correctly.

(b) In the above discussion, we have assumed that an oracle is always present, but this is not necessarily so. There are many types of

problems where an oracle does not exist.[30] Without the oracle, dynamic testing cannot be automated. Even if the oracle does exist, checking the correctness by using an oracle is a difficult task.

Secondly, the models for measuring program maintainability which have been mentioned are not necessarily the best. The suitability of the programming style model which we used to represent program understandability has been challenged by many people, for example[11,12,21]. Similarly, the program complexity model has also been criticised, for example by Bern[1] and Kearney.[16]

As we have stated before, the assessment of program quality is environment dependent, and in particular it is also dependent on the user's requirement. In the present technique, the user requirement is represented by a model program. The program to be analysed is compared with the model program for correctness and maintainability. The use of a model program to represent user requirement is the cause of problems stated above.

Another method for measuring program quality which should be considered is to compare the program directly with a user's requirement specification. In the case of program correctness, the user's requirement is normally given in the form of system specification. In this case it should be possible to check for program correctness by comparing the program code with the given specification. In order to do this automatically, the specification must be presented in a formal notation. At the moment there are many formal form of program specification have been proposed, for example the Z,[28] VDM[15] and OBJ[9] specification languages.

In the case of program maintainability, the user's requirement can be given in the form of programming standard. The use of programming standards is proven to be a better approach to enhance maintainability since it can ease the program understanding[7] and can avoid the problem of style clash.[8] Since each organisation will be using its own (different) programming standard, the user must be able to inform the system of details of the standard it is using. In order to check this automatically, the standard must also be presented in a formal notation. Since at the moment there is no proposal for a way to represent this standard formally, we consider this as part of our on-going research.

Acknowledgments

We would like to thank colleagues for helpful comments during the development of this project. Another vital contribution has come from the students attending the programming courses on which this system has been used.

References

1. G. M. Bern, "Assessing Software Maintainability", Comm. ACM 27(1), pp.14-23 (Jan 1984).

2. R. E. Berry and B. A. E. Meekings, "A Style Analysis of C Programs", Communication of the ACM 28(1), pp.80-88 (Jan 1985).

3. B. W. Boehm, "Software and its Impact : A quantitative assessment", Datamation, pp.48-59 (May 1973).

4. R. S. Burgess, An Introduction to Program Design using JSP, Hutchinson & Co. (Publishers) Ltd. 1984.

5. M. Coleman and S. Pratt, Software Engineering for Students, Chartwell-Bratt Ltd. 1986.

6. R. Conway, A Primer on Discipline Programming, Winthrop Publishers, Cambridge, Mass. 1978.

7. J. M. Einbu, "An Architectural Approach to Improved Program Maintainability", Software - Practice and Experience 18(1), pp.51-62 (Jan 1988).

8. R. L. Glass and R. A. Noiseux, Software Maintenance Guidebook, Prentice-Hall, Inc. 1981.

9. J.A. Goguen and J. J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications.", in Proc. Specification of Reliable Software Conf. (1979). Cam-

bridge, Mass.

10. M. Halstead, Elements of Software Science, North Holland, 1977.

11. G. Hannemyr, "Automatic assessment aid for Pascal programs - Rats!", Sigplan Notices 18(4) (April 1983).

12. W. Harrison and C. R. Cook, "A Note on the Berry-Meeking Style Metric", Communication of the ACM 29(2), pp.123-125 (Feb. 1986).

13. S. Henry and D. Kafura, "The evaluation of software system's structure using quantitative software metrics", Software - Practice and Experience 14(6), pp.561-573 (June 1984).

14. W. E. Howden, "Methodology for the generation of test data", IEEE Trans. Computing 24, pp.554-560 (May 1975).

15. C. B. Jones, Systematic Software Development using VDM, Prentice-Hall, Inc. 1986.

16. J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, "Software Complexity Measurement", Communication of the ACM 29(11), pp.1044- (Nov 1986).

17. B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, New York, 1974.

18. T. McCabe, "A Software Complexity Measure", IEEE Trans. Software Engineering 2(12), pp.308-320 (Dec 1976).

19. J. McCall, P. Richards, and G. Walters, Factors in Software Quality, 3 Vols., NTIS AD-A049-014,015,055, 1977.

20. E. Miller, "Coverage Measure definitions reviewed", Testing Tech. Newsletter 3, p.6 (Nov 1980).

21. P. W. Oman and C. R. Cook, "A Paradigm for programming style research", SIGPLAN Notices 23(12), pp.69-78 (Dec 1988).

22. E. I. Oviedo, "Control flow, data flow and program complexity", Proc. IEEE COMPSAC, pp.146-152 (1980).

23. R. S. Pressman, Software Engineering : A Practitioner's Approach,

McGraw-Hill, Inc. 1987.

24. S. Rapps and E. J. Weyuker, "Selecting software test data using Data Flow Information", IEEE Trans. Software Engineering 11(4), pp.367-375 (April 1985).

25. K. A. Redish and W. F. Smyth, "Program style analysis : A Natural By-product of program compilation", Communication of the ACM 29(2), pp.126-133 (Feb 1986).

26. K. A. Redish and W. F. Smyth, "Evaluating Measures of Program Quality", The Computer Journal 30(3) (1987).

27. M. J. Rees, "Automatic Assessment Aid for Pascal Programs", SIGPLAN Notices 17(10), pp.33-42 (Oct 1982).

28. J. M. Spivey, The Z Notation : A Reference Manual, Prentice-Hall, Inc. 1989.

29. P. B. Van Verth, A System for Automatically Grading Program Quality, SUNY (Buffalo) Technical Report, 1985.

30. E. W. Weyuker, "On Testing non-testable program", The Computer Journal 25(4), pp.465-470 (Nov 1982).

31. M. R. Woodward, D. Hedley, and M. A. Hennel, "Experience with path analysis and testing of programs", IEEE Trans. Software Engineering 6, pp.278-286 (May 1980).

32. S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance", IEEE Trans. Software Engineering 6(6), pp.545-552 (Nov 1980).

33. M. V. Zelkowitz, "Perspective on Software Engineering", ACM Computer Survey 10(2), pp.197-216 (June 1978).

Appendix

Typical output from a run of the system on a student program is as follows.

Analysis of /staff/ef/notes/c/analyse/prog44/sue.c   Susan G Bloggs

Analysis of Typographic Style

item	score	mark	out of
Average character per line	10.9	10.0	10
% indentation	0.0	0.0	10
% blank lines	33.3	6.7	10
Average spaces per line	4.5	10.0	10
Average module length	16.0	0.0	0
% good module	100.0	0.0	0
Average identifier length	5.2	10.0	10
% names with good length	60.0	0.0	0
% define's	0.0	10.0	10
% comments	292.3	0.0	10

Score for Typographic Style is 66.7%

|

|

Analysis of Structural Weakness

item	score	mark	out of
Used before set	0.0	0.0	-5
Defined/Set but not used	0.0	0.0	-10
Variable/Argument unused	0.0	0.0	-10
Return value sometimes ignored	0.0	0.0	-5
Return value always ignored	1.0	-5.0	-5
Statement not reached	0.0	0.0	-20
Return and return(e)	0.0	0.0	-20
Function has variable no of args	0.0	0.0	-20

Score for Structural Weakness is -5.3%

|

|

Analysis of Dynamic Correctness

item	score	mark	out of
test 1	1.0	15.0	15
test 2	1.0	20.0	20
test 3	1.0	30.0	30

Score for Dynamic Correctness is 100.0%

|

|

Analysis of Dynamic Efficiency

item	score	model	mark	out of
Max execution count	37.0	37.0	20.0	20
Coverage %	100.0	87.0	22.0	20
Average count	16.0	15.0	9.0	10
Score for Dynamic Efficiency is			102.0%	

|

|

Analysis of Program Complexity

item	score	model	mark	out of
Number of reserved words	9.0	9.0	20.0	20
Number of include files	0.0	0.0	10.0	10
Goto statements	0.0	0.0	30.0	30
Conditional statements	1.0	2.0	20.0	20
Loops	1.0	1.0	20.0	20
Opss	0.0	0.0	20.0	20
Score for Program Complexity is			100.0%	

|

|

Summary

item	score	weight	mark
Typographic Style	66.7%	25	16.7
Structural Weakness	-5.3%	100	-5.3
Dynamic Correctness	100.0%	40	40.0
Dynamic Efficiency	102.0%	10	10.2
Program Complexity	100.0%	25	25.0
Total score for sue.c is			86.6%

|

|

The command line interface ceilidh

S D Benford, E K Burke, E Foxley

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

There is now an additional interface to Ceilidh, which may prove useful under certain circumstances.

Instead of using a menu system, each possible Ceilidh action is available directly as a shell command. Some changes to command names have been essential to avoid clashing with names of existing commands.

A great advantage is that you can logout of the UNIX system, and return at any time. Your Ceilidh commands will not be affected. A disadvantage is that you need to remember the command names, instead of having a menu to prompt you every time.

Once you have set up the system (see below), you might type

```
     set.cse pr1 : to set up the "pr1" course
     set.unit 5  : to set yourself into unit "2"
     vn       : to view the notes
     set.ex 5    : to set into exercise 5
     setup         : to set up the skeleton
     ep       : (note change) to edit the program
```

At this stage, you could log out, then log in and carry on with

```
     cm       : to compile it
     sub      : (note change) to mark and submit it
     commands    : list the possible commands
     info     : more information (the old "help")
```

There is NEVER any need to quit. All sensible commands are available at all times.

Some command names are a little different from the menu options, to avoid clashes with existing commands.

Some exercise commands may behave strangely if no exercise has been set.

Using the new interface

You need to have the directory

```
     ~ceilidh/bin.cli
```

```
in your path.  Your teacher will tell you where  this  directory  is
on

your system.  Either insert this directory in your path this by hand,
or

     if you use the C-shell, execute
          source ~ceilidh/source.csh

     and put this line into your .login file to save having to  type
it
     each time you log in;  or

     if you use a Bourne shell, execute
          . ~ceilidh/source.sh

     and put this line into your .profile file, so  that  you  will
not
     need to execute it by hand each time you log in.

Then type
     set.env

to set up the necessary files.

You can then start from
     set.cse pr1

as shown above.
```

Courseware to support the teaching of programming

Steve Benford, Edmund Burke, Eric Foxley

Learning Technology Research
Department of Computer Science
University of Nottingham
NOTTINGHAM NG7 2RD, UK

e-mail: ltr @ cs.nott.ac.uk

Abstract

We describe the Ceilidh system, developed at Nottingham to support the teaching of programming languages in a controlled environment. The core of the system is an on-line coursework submission and auto-marking facility, the latter using a comprehensive variety of static and dynamic metrics to assess the quality of submitted work. Ceilidh also provides access to on-line, notes, exercises and solutions as well as extensive course monitoring and tracking facilities. We discuss the motivation for and functionality of the system, give an overview of its implementation and then present our experiences of using it.

1. Introduction

The teaching of computer programming often involves very large numbers and has to be aimed at students of widely varying experience and abilities. All the indications are that in future years the expansion of student numbers will further exacerbate the problems.

It is not necessarily the preparation of lectures that causes problems but the marking of coursework. In many universities and polytechnics the number of students attending programming courses is so large that it becomes extremely impractical for one person to mark a piece of weekly coursework. Having a team of people marking inevitably introduces inconsistencies. Yet, in a programming course the coursework is arguably more important than the lectures. Programming cannot be learned without extensive "hands-on" experience. Many students, used to lecture based teaching, do not really appreciate this fact and a significant

number
find it difficult to make the transition between the theory presented in
lectures and the creative process of designing and implementing pro-
grams. It is also the case that due to the delays involved in manually
marking so many programs, feedback is often given too late so that weak
students experience severe problems. It is clear that, with ever
increasing numbers, a more radical approach is required..

We describe a system called Ceilidh that, among other things, has pro-
vided this approach. Ceilidh stands for Computer Environment for
Interactive Learning in Diverse Habitats. As has been indicated, the
core of the system is an automatic coursework submission and marking
component which can give instantaneous feedback on students programs
from perspectives such as dynamic correctness, typographic style and

program complexity. Beyond this, Ceilidh also provides on-line access
to all notes, examples, exercises and solutions as well as progress mon-
itoring for tutors and general course administration.

Another way to approach this work is to consider the issue of quality
assessment, an area currently receiving much attention. Program qual-
ity assessment is of fundamental importance in any area of software
quality control. Tools for assessing quality are useful in two distinct
areas, both to assist an individual programmer in improving the quality
of his/her programs in a systematic way, and to help a manager to main-
tain quality controls and uniform standards for a project team. Ceilidh
can therefore be viewed as a quality control tool which aims to raise
students awareness of quality issues and to give them the experience of
working in a quality controlled environment. In addition, it can pro-
vide valuable feedback to the teacher about the strengths and weaknesses
of the class as a whole, and can indicate an emphasis which should be
made in the continuing teaching.

We have emphasised that the main goal of Ceilidh is to assist teachers in marking student work, normally an arduous task performed unsatisfactorily by hand. In some universities and polytechnics, an introductory or intermediate course in programming may be taken by as many as 200 students. During the course, every student may be required to submit one or two programs every week. The marking is normally done by asking the students to submit printed listings of the programs and of the results obtained from running them. These papers are then handed to various graders who mark them. This type of arrangement is unsatisfactory for several reasons.

(a) With large classes, the volume of papers to be handled is inconveniently large. Archiving old papers introduces even more inconvenience.

(b) The programs submitted on paper are not really tested, so we cannot be sure whether the results listed are the real results produced by the programs.

(c) If several people are involved in the marking process (if the marking is distributed between a number of graduate students for example), it is difficult to maintain a uniform marking standard.

(d) The possibilities for comparison and analysis of marks are extremely limited.

Many of these problems can be removed, or at least reduced, through the use of an on-line submission and marking system.

This paper is intended primarily as a practical guide to the functionality offered by Ceilidh (section 2) and a discussion of our observations of how the system is used and how it changes the learning process (section 4). However, for those interested in the more theoretical aspects of program assessment, we also provide a brief overview of its implementation (section 3). A much more detailed description of the underlying theory can be obtained from the authors on request.

Finally, it should be noted that various components of our system have been in use at Nottingham to support the teaching of the C language for three years and of C++ for one year.

2. Functionality of Ceilidh

We describe the functionality of Ceilidh by considering, in some detail, its user interface. The current user interface is menu-based and operates with VT-100 compatible terminals (the terminals used for teaching at Nottingham). Three top level menus, the student menu, staff menu and teacher menu, are used to group together student access functions, student progress monitoring functions and course set-up and management functions respectively. Comprehensive help facilities are also available.

Each course is divided into a number of units (chapters) representing different topics and then into different exercises within each unit. Users are able to browse through notes at will, giving the ability to read ahead and also revise past topics (browsing is, of course, subject to access control by teachers). In addition, teachers can specify default notes and exercises to be available at any particular time.

The student menu

For students, the major menu items are as follows.

(i) It allows the students to view general temporal course information (hand-in times for coursework) and more permanent information such as lecture notes (to be viewed on-line or to be printed). Several hundred pages of detailed notes for both C and C++ are currently available, including working versions of all of the examples used in lectures.

(ii) It provides access to the specifications of the programs they have to write for coursework. These specifications form a reasonably

precise definition of the program (or other object) which the stu-
dent has to write. The following is the specification for the
second assessment:

Write a program to read a number of centimetres (float) and print
the equivalent length in feet (integer) and inches (float).
Print your result in a form such as

```
20 cms is equivalent to 2 ft 3.5 inches
```

Print a message containing the word "error" if the input value is
negative.

Later in the course, specifications may run to several pages,
including C++ class definitions to be implemented or used and
detailed descriptions of exception states. The goal at this stage
is to develop the students' ability to work to specification. More
advanced tasks might typically involve implementing a C++ class to
work with a given interface program, writing a program to use a
given class, or writing a program which combines several given
classes.

(iii)It offers outline program source, and associated modules and header
files where appropriate, to assist in the solution of the course-
work. The student may be given a program skeleton outline, perhaps
including a declaration such as

```
const inches_per_cm = 2.54;
```

for use in the program.

(iv) It allows them to edit, compile and test run their program. The
extent to which compilation details are hidden from the student is
determined by the teacher; in early stages, all details may be
hidden; in intermediate stages the system may show the student the
commands which are being executed; in later stages the student may
have to complete all the compilation and linking commands outside
this system.

```
(v)  They can ask the system to mark their program. This uses a
variety
     of standard software metrics, combining dynamic and static
analysis
     techniques.  A summary of the marks is made available to  the
stu-
     dents  to  help  them to assess their program quality.  The
factors
     involved in the marking process, the level of detailed
information
     given  to  the  student,  and  the speed of response (on-line or
e-
     mailed overnight), are all determined by the teacher.

(vi) The system allows the student to submit completed  work  for
their
     assignment,  perhaps  together  with  mark details, all of which
is
     then retained on the computer for further analysis (notice that
the
     system  retains  copies of both the source code and the marks).
At
     present the system maintains a history of marks obtained, with
the
     most  recent being taken as the mark gained.  This history
informa-
     tion allows teachers to monitor the use  of  the  marking  tool
to
     identify  particular patterns of work (e.g. students who are
merely
     tweaking programs and re-marking without going away to think
about
     the problem).

     It is our view that the  ability  for  students  to  instantly
and
     repeatedly  re-mark  programs  and so work towards a target
quality
     level represents a key innovation in assessment techniques.

(vii)It allows them to view a model solution, to run this  solution,
to
     view  test  data  and  to run both their own solution and the
model
     solution against the test data.  The model solutions are
available
     only after the deadline for submission has passed.

(viii)It allows them to comment on a specific exercise or on the
system
     as  a  whole.   Comments are stored for later browsing by
teachers.
     In particular, whenever a student submits some work they are
asked
     if  they wish to comment on the marks obtained (e.g. to disagree
or
     sometimes to make an excuse for lateness!).  For formally
assessed
```

work (i.e. work which counts towards end of year marks) we are con-
sidering introducing a challenge system where the student may
challenge the system mark and demand human marking instead. How-
ever, in this case, they must accept the new mark whether it is
higher or lower than the original.

(ix) It offers help facilities including an overview of the marking
metrics employed by the system (including a good programming style
guide).

The staff menu

An additional menu available to staff, tutors and demonstrators offers
all of the above student facilities, plus the following.

(i) List details of the work submitted by all of the staff member's
tutees. For each item of coursework the listing gives a summary of
the marks awarded and the time at which it was submitted (in par-
ticular whether it was early or late). Details of the work such as
the program source code and a more detailed breakdown of the marks
can be inspected if requested.

(ii) List the names of tutees who have not submitted work, or who have
submitted late.

(iii)List the marks awarded to tutees for a particular exercise (useful
immediately before a weekly tutorial), or for all weeks so far this
semester.

(iv) Check attendance at laboratory sessions against laboratory lists.

The teacher's menu

An additional menu for the teacher offers the following facilities.

(i) Set new coursework, or amend existing coursework. The system
prompts the teacher to ensure that all the necessary data items
have been input. This phase is quite demanding of the teacher the
first time that a new coursework is implemented. The teacher

```
must
     supply
          a specification of the coursework,
          a model solution,
          an outline skeleton if required,
          associated headers and object modules if required,
          techniques for dynamically testing the student solutions, and
          a marking scheme (the weights to be attached to each metric).
     The weights in the marking scheme can be  varied  to  reflect
dif-
     ferent emphases on different courseworks.

(ii) Each student's work can be marked, and the marks  e-mailed  to
the
     student,  and/or grouped and e-mailed to tutors.  Typically,
teach-
     ers reserve the right to change  the  dynamic  tests  for  a
given
     coursework  (the  test data files or the test shell scripts) at
any
     time.  Additional tests may  then  disclose  additional
weaknesses
     after the student has submitted the coursework.

(iii)The teacher can also obtain a summary of each software  metric
for
     the  whole  class,  thus  giving feedback on the main strengths
and
     weaknesses of the class (e.g. are these students poor at
indenting
     programs properly).

(iv) The teacher can conduct a plagiarism test on all the submitted
pro-
     grams.

(v)  Teachers can browse the comments file.  At present we are using
an
     informal  convention  for marking to file to show which teacher
has
     dealt with which comments.

3.  Implementation

The section gives a brief overview of the Ceilidh system
implementation.
Given the space available, it is not possible to describe the details
of
the marking metrics we employ.  However, recognising that assessment
of
programs is an area of interest for many researchers, we identify
previ-
ous research which has motivated our approach  and  discuss  a  few
key
issues.   People  who  are particularly interested in this aspect of
the
system are  referred to a more  detailed  research  paper[17]  which
is
```

available from the authors.

Ceilidh runs under the UNIX operating system, running on SUN hardware.
It is written as a combination of shell, C, C++ and awk programs. Wherever possible it uses existing software tools (such as lint in supporting the C language) already available on the system; the C and C++ versions therefore employ slightly different metrics reflecting the language tools already available on the operating system.

The major headings for the assessment were originally developed by Abdullah Mohd Zin and Eric Foxley,[17] and are dynamic correctness, dynamic efficiency, typographic analysis, complexity analysis and structural weakness.
The overall mark is calculated as a weighted average of the marks under these five headings, the weights being specified by the teacher.

3.1. Dynamic correctness

Dynamic testing involves executing the program against several files of test data, and perhaps with different program arguments. The main aim of dynamic testing is to uncover execution errors in a program. To ensure that as many as possible of the potential errors can be detected the test data must be chosen carefully. Many techniques for selecting test data have been proposed for example by using Data Flow Information,[11] Cn coverage measures,[8] TERn measures,[15] and boundary-interior testing.[5]

Running a program against test data will produce output. Most testing processes are based on the assumption that an oracle is present to check the correctness of output.[14] Construction of an oracle is a non-trivial problem in any situation where the program output format is not exactly specified. In a simple student problem such as `write a program to read a number of centimetres and print the equivalent distance in

feet and inches', the number of possible outputs for the same input data include examples such as 1 foot 3.6 inches; 1 ft 3.59 ins; one foot four inches; and ins 3.6 ft 1. All these must be correctly interpreted by the oracle.

In our system the teacher provides for each coursework a number of dynamic tests which may be given as files of test data, or as UNIX shell scripts (e.g. if we need to run the program against a large file of data such as a dictionary). For each test, the teacher provides a maximum mark. To check the correctness of the output, the teacher provides for the oracle a number of regular expressions, each with a sub-mark attached. If that expression is found in the output, the sub-mark is awarded. The sum of the marks for all the regular expressions of the test which have been found is then scaled to reflect the total mark to be awarded for this test.

3.2. Dynamic efficiency

Program profiles represent another possible output from dynamic testing. These may be useful for identification of dynamically dead code, checking the correct number of loop iterations and to help in the optimisation of the most frequently executed code segments. Profiling cannot, of course, distinguish between code which is dead simply because the chosen test data does not call for its execution, and code which is logically non-executable.

In the teaching of programming, we are generally not concerned with marginal efficiencies in program size or speed. However, it was felt that such concerns may need to be considered in later courses which specifically focus on programming efficiency (e.g. a comparison of algorithms).

It was therefore decided that any software quality assessment should include a measure of program efficiency. This is most easily based on dynamic profiling of student program, since dynamic profilers are already available for most language systems. These can be used to

find
the maximum execution count of any statement in the program, which
is
usually the key factor in running speed. Other aspects of
efficiency
such as the size of the compiled program are not currently included
in
our system, but may be added in due course.

3.3. Typographic analysis

The metrics used here are those described in the research literature
as
being relevant to program readability and maintainability.
Software
maintenance is a broad-based activity, and a prerequisite activity is
to
understand the software to be maintained.[16] Based on this fact,
one
common method to measure maintainability of a program is to measure
its
understandability. There are few models which have been proposed so
far
for measuring a program's understandability including the Program
Diffi-
culty Model,[1] proposed by Bern. An example of a software tool
which
is based on this model is called The Maintenance Analysis Tool[1]
which
analyses programs written in VAX-11 Fortran, a superset of Fortran
77.
Another approach is to develop a Programming style model which
evaluates
the program's understandability based on the style of the program.
The

concept of programming style has been discussed by many people,
for
example by Kernighan and Plauger.[6] The first effort to measure
pro-
gramming style was proposed by Rees[12] who described a Pascal
source
code style grader. Berry and Meeking[2] proposed a style grader for
C,
which calculates its results using similar factors to those of Rees.

For maintainability measurement, our system uses the programming
style
model. Following Oman and Cook,[9] we divide the programming style
into
two categories: those pertaining to the typographic arrangement,
and
those measuring the structural content of the code. Typographic
style
describes the way a program source code is presented. We measure
abso-
lute metrics including the number of blank lines and comments;
lengths

of identifiers and comments; ratio of white space to other characters;
average length of modules and `correctness' of indentation.

3.4. Complexity analysis

A number of techniques have been proposed for measuring program complexity by researchers such as Van Verth[13] ,Halstead[3] ,McCabe[7] ,Henry and Kafura[4] and Oviedo.[10] In our system we have chosen to use as metrics a static analysis of the frequency of occurrence of

gotos, reserved words, include files
operators, loops, conditional statements
assignment statements, depth of loops
function calls, complexity of expressions
library functions, literals

These are compared against an analysis of the model solution provided by the teacher.

3.5. Structural weakness

For this section of the C support system we rely heavily on the standard Unix C utility lint, which comments on C program source code. For our structural weakness analysis, the score is based on the occurrence of static problems in the programs. These warnings include the following:

variable declared but never used,
variable assigned but never used,
value returned by a function never used,
value returned by a function sometimes used,
statement not reached, and
variable used before set.

For C++ support, we rely on warnings from the GNU C++ compiler using the

```
g++ -Wall
```

option. Further extension in this area is planned.

4. Experiences with Ceilidh

The Ceilidh system has been developed and used at Nottingham over the past three years. This year the system was used to support a two term C++ course for 150 novice programmers. Consequently, as a result of this practical experience, we have gained several key insights into how

such courseware is used and how it changes the learning environment.

First, although initially aimed at replacing the traditional paper-based marking mechanism, it turns out that auto-marking dramatically changes the nature of the assessment process itself. The ability for students to work towards a target and to argue with marks as they are given them has been a new innovation and we feel that it has increased discussion and feedback on the course. This leads to perhaps our most significant observation. Ceilidh has been very effective at consciousness raising. Even those students who disagree with the marker are more aware of quality control issues and this year, style and correctness have been high on everybody's agenda. Indeed, it has been encouraging to receive such criticism of our own examples and model solutions. It is therefore important to put the auto-marker into perspective; in particular it does not have to be perfect, only good enough to stimulate and support the learning process. Interestingly this beneficial cultural change in the learning process is something that might have been overlooked by a purely theoretical consideration of automatic program assessment. Of course, even in its early state, Ceilidh is already much better at marking 150 scripts a week than the course lecturers would ever claim to be.

Another major improvement has been the ability to identify students who are struggling with the course at an early stage. Tutors have been able to obtain up to the minute details of the progress of each student and although Ceilidh is not always 100% accurate, you can be sure that a student is having problems if they are consistently obtaining low marks or not handing in any work at all.

Judging from student feedback and our own comparison with previous years' teaching, students seem to have found the system to be of significant benefit. However, we have observed several problems:

o Some students seem to feel that they must obtain 100% for each exercise. Although perhaps understandable for dynamic

correctness,
this is not reasonable for typographic analysis. The result has
often been thrashing around the marking system many times to gain
just a few marks (one person managed fifty iterations to put their
mark up by only 2% when it was already in the eighties to begin
with!).

o Some students have complained that the typographic and complexity
marks can sometimes appear almost arbitrary. In particular they
have requested more feedback about where they are going wrong and
how marks are being calculated.

The first point might best be addressed by giving the students a clearer understanding of how the marker is intended to be used (i.e. as a guide-line to assessment). The same misunderstanding underlies the belief that "if my program satisfies the dynamic marker then it must be correct". However, we did also introduce the technical innovation of enforcing a minimum time delay between submissions (this is configurable by the teacher).

The second point is more difficult to deal with. First, we note that

these complaints tended to come from those students who were coping better and whose marks were already quite satisfactory. We suspect that several of these had looked at the system to see how it worked and, when they did, realised how it could be fooled. As a result they were unimpressed (e.g. comments like "if I add two extra nonsense comments my style mark goes up by 3%"). It is probably fortunate that they can't peer into human markers heads in this way! Chosing an appropriate level of feedback is a difficult issue. At first glance it might seem desir-able to give as much as possible (this is certainly what the students want). However, our goal is to encourage them to think for themselves and to learn to solve their own problems. Thus, we believe it is not always appropriate to give maximum feedback at all times. Of course, this is a case of us claiming to know what is best for the students better than they do themselves. In the end, we decided to make the level of support configurable and to let further experience be the judge.

Finally, it is worth noting a number of smaller changes that were made as a result of our experiences.

- The comment facilities were a relatively late addition to the system.

- Initially the system presented the typographic mark followed by the dynamic because the former was quicker to calculate and we wanted to give something to look at while dynamic tests were being run. We later changed this to give the dynamic mark first to reflect its greater significance and to reduce the focus on the typographic mark.

- We added a check submission facility at the student's request so that they could confirm when the system thought they had last submitted their solution. Interestingly, they seemed to believe the new facility better than the original command, even though it had been built by the same people.

5. Future developments

Overall, we have been pleased with the success of Ceilidh on our own courses. Our next wish, beyond a few immediate improvements, is to make it available to other organisations in a usable form. In particular, we would like to separate the general course management facilities from the C++ specific features. This would enable other departments to develop courses in a standard format which could then be plugged into the system. To achieve this goal we will have to extend the system in several major ways:

(i) Port the system to different operating environments (e.g. networked
PCs).

(ii) Provide assessment support for other programming languages.

(iii)Develop installation and system management tools.

(iv) Better document the system.

(v) Provide alternative forms of assessment which could support other
types of course. Examples might include multiple-choice and screen
based script annotation.

(vi) Develop better user interfaces based on different technologies
(e.g. X-windows and windows for the PC).

(vii)Provide networked and email interfaces in order to support distance
learning.

(viii)Rewrite in a compiled form to increase operating speed.

(ix) Carry out a proper evaluation of the ability of the system to
enhance the learning process.

We have recently acquired an initial grant to make a start on this work.
Beyond this, we are looking to establish a consortium interested in
further development of the system and in integrating it with other work
in the field.

Acknowledgments

We would like to express our thanks to Abdullah Mohd Zin for early work
on the C language system, and to all the students whose constructive
comments have been of great help.

References

1. G. M. Bern, "Assessing Software Maintainability", Comm. ACM 27(1),
pp.14-23 (Jan 1984).

2. R. E. Berry and B. A. E. Meekings, "A Style Analysis of C Programs", Communication of the ACM 28(1), pp.80-88 (Jan 1985).

3. M. Halstead, Elements of Software Science, North Holland, 1977.

4. S. Henry and D. Kafura, "The evaluation of software system's struc-
ture using quantitative software metrics", Software - Practice and
Experience 14(6), pp.561-573 (June 1984).

5. W. E. Howden, "Methodology for the generation of test data", IEEE
Trans. Computing 24, pp.554-560 (May 1975).

6. B. W. Kernighan and P. J. Plauger, The Elements of Programming
Style, McGraw-Hill, New York, 1974.

7. T. McCabe, "A Software Complexity Measure", IEEE Trans. Software
Engineering 2(12), pp.308-320 (Dec 1976).

8. E. Miller, "Coverage Measure definitions reviewed", Testing Tech.
Newletter 3, p.6 (Nov 1980).

9. P. W. Oman and C. R. Cook, "A Paradigm for programming style
research", SIGPLAN Notices 23(12), pp.69-78 (Dec 1988).

10. E. I. Oviedo, "Control flow, data flow and program complexity",
Proc. IEEE COMPSAC, pp.146-152 (1980).

11. S. Rapps and E. J. Weyuker, "Selecting software test data using
Data Flow Information", IEEE Trans. Software Engineering 11(4),
pp.367-375 (April 1985).

12. M. J. Rees, "Automatic Assessment Aid for Pascal Programs", SIGPLAN
Notices 17(10), pp.33-42 (Oct 1982).

13. P. B. Van Verth, A System for Automatically Grading Program Qual-
ity, SUNY (Buffalo) Technical Report, 1985.

14. E. W. Weyuker, "On Testing non-testable program", The Computer
Journal 25(4), pp.465-470 (Nov 1982).

15. M. R. Woodward, D. Hedley, and M. A. Hennel, "Experience with path
analysis and testing of programs", IEEE Trans. Software Engineering
6, pp.278-286 (May 1980).

16. S. S. Yau and J. S. Collofello, "Some stability measures for
software maintenance", IEEE Trans. Software Engineering 6(6),

pp.545-552 (Nov 1980).

17. Abdullah Mohd Zin and Eric Foxley, "Automatic Program Quality Assessment System", Proceedings of the IFIP Conference on Software Quality, S P University, Vidyanagar, INDIA (March 1991).

The Design Document for Ceilidh Version 2

S D Benford, E K Burke, E Foxley, N Gutteridge, A Mohd Zin

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

1. Introduction

Ceilidh is a course management system. The main objective of Ceilidh is to support teaching and learning through computer. As a course management system, it also provides some facilities for the course teacher to organise the teaching and to monitor the progress of the students attending the course.

Historically, Ceilidh was designed for teaching and learning of programming through computer.[1] The main motivation for Ceilidh was to solve the problem of teaching a large programming class. It has long been realised that programming cannot be learned without extensive hands-on experience. It is also necessary that all programs written by students should be marked and checked by the teacher, and the feedback should be given to the student as soon as possible. However, handling and marking 150 or 200 student programs every week is impossible to perform quickly and fairly by hand. Ceilidh allows students to develop their programs on the terminal, and each program will be marked automatically.

The experience with the first version of Ceilidh has motivated us to explore further. During this present stage in the development of Ceilidh, efforts are being made to provide a more comprehensive system, covering other types of courses. This document describes the design of Ceilidh version 2, due for release mid-1993.

2. Usage of Ceilidh

2.1. Users

The users of Ceilidh can be divided into five categories:

o student: someone using the system as a learning tool

o course tutor: those with access to student progress monitoring

o course teacher/administrator: the person in charge of managing a course

o course developer: the person in charge of the creating of a particular course

o system manager/administrator: the person in charge of the running of the whole system

Facilities provided by Ceilidh for each group of users are as follows:

Students:
- reading documents about Ceilidh.
- selecting course and reading course summary.
- selecting unit and reading unit notes for a particular course.
- answering exercise, submitting answer, and in some cases getting automatic feedback.
- make comments to course teacher.
- getting some information about the progress of the course.

Tutors:
- view students' work.
- mark students' work, automatically or by hand.
- enter students' mark, in certain cases.
- view students' mark.

Course Administrator:
- edit course "motd", "summary", "weights", and "scales".
- set exercise as "open", "late" and "closed".
- register students.
- search for missing and unknown students.
- informing students about the progress of their work.
- gathering of overall exercise metrics.
- search for plagiarism.

Course Developer:
- creating new unit and edit notes for that unit.
- creating new exercise and edit question for that exercise.
- set solution for each exercise.

System Manager:
- install system and users.
- install and setup New Course
- Edit system "motd", "staff list", "tutor list", and "help" files.
- Monitoring the progress of the whole system.

The permission for students are controlled mostly by the standard UNIX facilities. For the tutors and teachers facilities there is extensive use of SUID programs to permit access to confidential areas of the system. For developers and managers, they must login as ceilidh to perform their activities.

2.2. Basic concepts

Some basic concepts of Ceilidh are described here:

System

We must distinguish at all times between the system and the courses which run underneath it.

Papers and documents

Within the Ceilidh system, there is a collection of research papers and documents related to Ceilidh which can be read by those who want further information about Ceilidh.

Message of the day, "motd"

Following the UNIX style, the system manager can make an urgent message available to all users of the system by using the Ceilidh system "message of the day" (motd) file. Similarly, the course teacher can inform students of urgent news by using the course or unit motd file.

Course

A course in Ceilidh is given a course name (up to three characters) and course title (one line). Each course is divided into a number of units. Within each unit there are unit notes and a number of exercises.

Exercise

Exercises can be divided into three types:

- programming exercise
- question/answer exercises
- text submission (essay)

Programming exercises can be divided to two types, depending on the type of language.

- Imperative programming language: for example Pascal, Ada, FORTRAN and C. To write a program in this type of language requires an "edit, compile and run" development cycle.

- Interpretive language: for example shell programming, Awk, BASIC, LISP, and Prolog. In this type of language, after the program is written, it is directly executed by the interpreter. The development cycle is thus "edit, run".

The question/answer exercises allows for exercises involving[2] a number of questions, where the answer to each may be

- multiple choice (a single character from a restricted set);

- a numeric value (or sequence of values, with tolerance specified by the teacher);

- simple words (recognised by a regular expression oracle provided by the developer); and soon

- simple sentences (with semantic recognition). For copyright reasons, this cannot be distributed at present.

The text submission exercises act merely for the collection of files from the students. This can save endless hassle in the collection of paper scripts.

2.3. System overview

The Ceilidh system is divided into three levels: system level, course and unit level, and exercise level.

System level

Facilities provided by Ceilidh at this level are:

- select course.

- reading papers.

- getting some information about the system.

Course and unit level

Facilities provided by Ceilidh for reading notes:

- select unit.
- select exercise.
- reading the notes for a particular unit.

Exercise level

Facilities provided for each exercise depend on the type of exercise.
In general these facilities include:

- read exercise.
- prepare the solution for the exercise.
- submit answer for the exercise.
- look at the teacher's solution - after certain date.

3. Design objectives

The design objective of Ceilidh can be described as follows:

- To support multiple courses.
- To support automatic feedback.
- To support multiple interfaces.
- To support remote learning.
- To allow for extensibility.
- To allow for portability.

3.1. Support for multiple courses

Ceilidh is designed to support the learning and teaching of many courses. To date most of the effort has been geared towards supporting programming courses. Text submission (essay) oriented courses have now also been included within the system. In future, graphics courses and some mathematically based courses should also be considered for inclusion in the system.

3.2. Support for automatic feedback

Feedback from the teacher is important for students because it helps them to improve the quality of their work.

3.3. Support for multiple interfaces

Different computer system supports different types of terminals. To
enable Ceilidh to be used as widely as possible, it is designed to sup-
port three types of interface:

- o X windows interface
- o Dumb terminal interface
- o Command line interface

3.4. Support for remote learning

Remote learning is an important learning technique in the future.
Ceilidh is designed to support remote learning by allowing access
through:

- o Email system
- o FTP system
- o Client-server system

The degree of distribution of facilities is potentially unlimited.

3.5. Extensibility

Ceilidh is designed to be open, so that new courses can be added without
major changes to the system. We are gradually making more facilities
tailorable, so that typographic, compilation and oracle facilities can
be added on a per-course basis.

3.6. Portability

Portability is very important so that Ceilidh can be used as widely as
possible. Apart from security aspects (which rely on UNIX SUID facili-
ties) all other aspects are portable.

To ease the process of porting Ceilidh into PC, all file names should
now conform to MS-DOS standards.

4. Ceilidh structure

In line with the design of most software, the structure of the Ceilidh
system is divided into three layers: user interface, basic tools and
data base. The advantage of this approach is that we can change one

layer without affecting the other parts of the system.

The structure of the system is described in figure 1.

Diagram goes here, see printed notes

Figure 1: The Ceilidh data base

4.1. User interface

The dumb terminal menu interface is a set of shell scripts stored in the directory ~ceilidh/bin.mnu.

The command line interface is also a set of shell scripts stored in the directory ~ceilidh/bin.cli.

The X windows interface consists of a set of executable programs stored in the directory ~ceilidh/bin.x. The source programs are stored in ~ceilidh/bin.x/SOURCE.

4.2. Basic tools

Basic tools are software tools that are integrated with the system. For example tools for setting up an exercise, submitting an exercise, viewing students mark etc. These tools are divided into two categories:

- o The system wide tools are stored in the directory ~ceilidh/Tools . These tools are programs and shell scripts. The program sources are stored in ~ceilidh/Tools/SOURCE together with a Makefile.

- o The course related tools (for example the compiler, static analyser, program marker, essay marker, etc.) are stored in the course/bin directory for the course concerned.

Our strategy is to look for all available tools first. If the required tools are not available, then we have to develop our own tool. We have used UNIX tools such as awk and sed when appropriate. Note that the full pathname of the tools must be included in the code to avoid the substitution of student rogue versions.

4.3. Data base

The Ceilidh data base is simply a collection of files. This data base
consists of directories for help, papers, lib and courses.

Help

The help information for the system is stored in two directories:

```
     help
          The help directory contains all the text  files  displayed by
          the  various help commands in the menu and command line inter-
          face versions.  Each filename in general starts with
                sys   for system level help
                cse   for course/unit level help
                ex    for exercise level
          and ends with
                usr   for student level help
                tut   for tutor level help
                tch   for teacher level help
                dev   for developer help
          The "exercise-user" level is further split, so that different
          help is offered depending on the stage the user has reached in
          the solution of a given exercise; does the user have a source
          or executable in the directory?

     xhelp
          This directory has a help file corresponding to each button on
          the X windows screen.
```

Papers

A number of papers and documents relevant to Ceilidh are stored here.
It is our belief that students should have access to all information
regarding the workings of the system, and should be encouraged to read
around the topic.

Each paper occurs as three files

```
      name.ttl   a one-line title
      name.cat   a version suitable for viewing through "cat".
      name.ps    a PostScript version
```

The originals (in troff format) are stored at Nottingham. New papers
could be added and existing ones removed at your discretion.

"lib"

This directory contains a few miscellaneous text files such as the roff

macros used in preparing the papers (one set of macros for the .cat ver-
sion, one for the .ps version, and one for the .ohp version used with
course notes), and a default Makefile for compilation exercises.

Courses

Below the ceilidh directory, each course has a directory such as
course.pr1 for the course "PR1" (C++ programming in semester 1). The
layout of files under a typical course directory is shown in figure 2.
Below the course directory is a bin directory containing any shell
scripts or executable programs (sources and Makefile are in a directory
such as ~ceilidh/course.pr1/bin/SOURCE directory) specific to this
course. These may be compilation, debugging or typographic commands,
for example. These commands will take preference over any system com-
mands of the same name.

Below the course directory are directories for each unit of the course,
for example unit.3 to contain the third unit or the "PR1". The unit
name is "numeric", and they will appear in numeric order when listed.
Below this are directories for each item of coursework, such as ex.1 for
the first exercise. Exercise names can be any string of up to three
characters.

All course, unit and exercise directories contain a file "title" con-
taining a one-line title for the course, unit or exercise. The course
directories should also contains a summary file containing a summary of
the lectures, times, courseworks set and hands-in dates. The summary
files in the unit directories are a brief summary of the content of that
unit.

The content of the exercise directories depends on the type of the exer-
cise. All exercise directories should contains the following files:

```
type        information about exercise type, suffix, compiler
late.dat    date when the exercise is considered late
late        lists of all late submissions
```

close.dat   date when the exercise is considered closed
model.q   the exercise question

It also include a directory solns which contains all submitted solutions from students together with a marks file, listing all students' marks, one line per student submission.

Diagram goes here, see printed notes

Figure 2: A course filestore layout

5. Implementation considerations

Ceilidh is designed to run under the UNIX Operating System environment. UNIX has many facilities which have enabled Ceilidh to be developed without much difficulty.

Security

One of the major weaknesses of the present UNIX file system is the problem of access control.

Access to files in UNIX is controlled by dividing users into three categories: the owner, the owner's group and other people. For each category, there are three types of access: read, write and execute. This type of access control mechanism is not sufficient for the Ceilidh system. Some of the files requires access control which allows append only. For example Ceilidh programs can only append entries to the mark files. They are not allowed to change other parts of the file. Another type of access control is read thing relevant to you only. For example for the mark file, a student user is only allowed to see only the lines giving their own marks, not other people's. For the copies of submitted programs, there is no general public read access, but a student must be permitted to read their own solution.

To solve this problem, we have to use UNIX SUID (set-user-id) and SGID (st-group-id) techniques. We use a number of ceilidh owned SUID program for, for example, the marking. Such a program can read files in the ceilidh file-system which have no public access, and can write a copy

```
of
the student's program within ceilidh and ensure that it  has  no
public
read access.  Such a program must open the Ceilidh files within the
pro-
gram, after the ceilidh UID has become effective.  The  student
program
may not be readable to an SUID program (the student may not allow
public
read access to their program) so that in general the student will be
fed
to  the  SUID program as standard input.  We hope that this problem
will
eventually be solved  when  the  new  UNIX  V.4ES  is  available,
where
enhanced security features such as ACLs are provided.

The major SUID programs are

     o    ccef whose main  purpose  is  to  copy  information  into
the
          Ceilidh area, and

     o    efread whose main purpose is to read confidential files
within
          the Ceilidh area.

Typical files  within  Ceilidh  which  must  not  be  publicly
readable
include:

     student solutions
          These are owned be Ceilidh, with access mode  600.   They
are
          named
               <student logname>.<suffix>
          in general.  The student must be able to read their  own
file
          on  demand,  which  is  controlled by the program efread
which
          acts as a controlled version of cat.

     marks files
          These are the files
               <course>/<unit>/<ex>/solns/marks
          in each exercise directory.  Ceilidh never overwrites a
marks
          file, each new mark is appended.  These have no public
access,
          and a particular student must be able to  read  on  their
own
          entries  (lines) from the file.  This is done within efread
by
          using a grep command.

     Oracle files
          The files for checking the accuracy of program output must
not
```

be publicly readable. The oracle program itself[3] will be called from the SUID marking program, so will run with UID as ceilidh, and will be able to read the files.

Archiving

With traditional coursework and examinations, it is usual that all relevant documentation (exam papers and scripts, student projects etc) has to be kept for a fixed time after the examination.

We have built in an archiving command, so that an archive copy of the course (using shar) can be taken at the end of a course for two reasons:

- o to be kept as required above for regulatory reasons; and

- o to retain a copy of the course at the time of completion, ignoring any later changes that may occur, for access at a later time by the external examiner.

Auditing

The latest release of Ceilidh includes facilities for maintaining audit trails in the system.

An audit trail can be set up to record either

- o all activities of a given student within Ceilidh; or

- o all usage of a particular Ceilidh facility (e.g. compiling, or printing).

Each audit trail is collected in a separate file, under an archive directory, with one line appended per archive event. The line is of the form

<course>:<unit>:<ex>:<student>:<date>:<student>:<facility>:<comment>

The facilities for analysing these trails are not yet developed. The files can become very large, so any which are set up must be checked regularly.

There must always be an audit file for Error messages.

```
Appendix 1 : The shell scripts in ~ceilidh/Tools

CAddSt

     Called as:
          Tools/CAddSt <course>

     Description:
          Command for staff to add student to register for a particular
course
          Arg1 is the <course>

CAudit

     Called as:
          Tools/CAudit

     Description:
          Audit trail facilities menu for system administrator
          No arguments

CComment

     Called as:
          Tools/CComment system
          Tools/CComment ex <source> <executable>
          Tools/CComment course
          Tools/CComment unit
          Tools/CComment mk <course> <unit> <exercise> <suffix>

     Description:
          Send comment to first named teacher in home/staff.lst for
system comments
          in course/staff.lst for course etc comments
          flag -a to force append of program source to message

CCompile

     Called as:
          CCompile <source> <executable>

     Description:
          Compile a student source program
          Use a Makefile in the exercise if one exists and has an entry
such as
                 prog72 : ?????
          Use .o files if they exist
          The system first looks for a <course>/bin/CCompile

CCopyout

     Called as:
          Tools/CCopyout <source> <destination>

     Description:
          Copy <source> (usually in Ceilidh) to <destination> (student)
          Replace "$USER" "$NAME" "$DATE" by student logname, full
name, date
```

```
            and "PROG" by e.g. prog32

      CCrsSumm

            Called as:
                  Tools/CCrsSumm <course>

            Description:
                  Produce a course summary
                  Arg1 is the course, e.g. pr1

      CDynCorr

            Called as:
                  Tools/CDynCorr -v2 <course> <unit> <exercise> <executable>

            Description:
                  Execute dynamic correctness tests
                  Execute$0 [ -v2 ] course unit ex executable
                  flag -v<n> for verbosity
                  flag -c    to permit compilation
                  Expects ~ceilidh/cse/unit/ex/test data and recogniser files

      CEdNotes

            Called as:
                  Tools/CEdNotes <course> <unit>

            Description:
                  Edit notes of given course and unit
                  Not for public use

      CEdWts

            Called as:
                  Tools/CEdWts 2
                  Tools/CEdWts 1
                  Tools/CEdWts 0

            Description:
                  Amend entry in weights file
                  Arg1 is 0 to indicate an open exercise, 1 for late, 2 for
      closed

      CEnterMk

            Called as:
                  Tools/CEnterMk <course> <unit> <exercise>

            Description:
                  For teacher/tutor to enter marks directly

      CFeature

            Called as:

                  CFeature <course> <unit> <exercise> <executable>
```

```
     Description:
          Use the features oracle file,
          call as
              $0 course unit ex
          To be called from "mark.act" file as in
              10 Feature: CFeature $C $U $E prog$U$E

CFindSt

     Called as:
          Tools/CFindSt <name>

     Description:
          Find a student, <name> is name being searched for.

CFindTu

     Called as:
          Tools/CFindTu -l
          Tools/CFindTu <name>

     Description:
          Find given tutors tutees, <name> is tutor's login.
          If "-l" is given then list all tutors.

CListCrs

     Called as:
          Tools/CListCrs -t
          Tools/CListCrs -t -s

     Description:
          List courses

CListEx

     Called as:
          Tools/CListEx <course> <unit>
          Tools/CListEx <course>
          Tools/CListEx -b <course> <unit>

     Description:
          List exercises in a unit

CListPap

     Called as:
          Tools/CListPap -t

     Description:
          List paper titles

CListUnt

     Called as:
          Tools/CListUnt <course>
          Tools/CListUnt -b <course>
```

```
          Description:
               List units in a course.
               Flag -b for brief.

     CMailMks

          Called as:
               Tools/CMailMks -[stv]

          Description:
               Mail results to teachers, tutors and/or students
               flag -s for students, -t for tutors, -v to view

     CMetrics

          Called as:
               Tools/CMetrics <course> <unit> <exercise> <suffix>

          Description:
               Display overall metrics for one exercise for the whole class
               dynamic test results should be added somehow

     CMissSt

          Called as:
               Tools/CMissSt -s <course> <unit> <exercise>
               Tools/CMissSt -u <course> <unit> <exercise>

          Description:
               Find missing students.
               Flag -s for missing, -u for unknown, -m to mail results

     CNewWeek

          Called as:
               Tools/CNewWeek

          Description:
               For teacher to set up new weeks exercise state
               No args

     COutput

          Called as:
               Tools/COutput

          Description:
               View or print file.

               Filename is $1,  $2 is "p" to print

     CPlag

          Called as:
               Tools/CPlag <course> <unit> <exercise> <suffix>

          Description:
```

```
            Check for plagiarism
            $0 course unit exercise suffix

CQAMark

      Called as:
            Tools/CQAMark <source> <unit> <ex> <user>

      Description:
            archive QA questions
            archive solns/fred.q<digits> into solns/fred.mc
            remove fred.q<digitd> if flag -r
            make if flag -m

CReMark

      Called as:
            Tools/CReMark $STUD_SRC $STUD $SRC <executable>

      Description:
            Re mark a submitted program.

CReMark1

      Called as:
            Tools/CReMark $STUD_SRC $STUD $SRC <executable>

      Description:
            Re mark a submitted program

CReMarkI

      Called as:
            Tools/CReMarkI $STUD_SRC $STUD $SRC <executable>

      Description:
            Re-mark a submitted program
            For interpreted languages, not tested

CRegAll

      Called as:
            Tools/CRegAll <course>

      Description:
            Register all students for course given as arg1

CRemvSt

      Called as:
            Tools/CRemvSt <course>

      Description:
            Remove students from a particular course given as arg1

CRoffcat

      Called as:
```

```
            Tools/CRoffcat <notes> > notes.cat

       Description:
            Messy script to roff nroff files ending ".ms"
            It produces ".cat" output suitable for $PAGER

  CRoffdvi

       Called as:
            Tools/CRoffdvi <notes> > notes.dvi

       Description:
            Messy script to roff nroff files ending ".ms" into .dvi
            for ditroff viewer such as xditview

  CRoffohp

       Called as:
            Tools/CRoffohp <notes> > notes.ohp

       Description:
            Messy script to roff nroff files ending ".ms" for OHP

  CRoffps

       Called as:
            Tools/CRoffps <notes> > notes.ps

       Description:
            Messy script to roff nroff files ending ".ms"
            Produces PostScript

  CRunInt

       Called as:
            Tools/CRunInt -t -d <course> <unit> <exercise>
            Tools/CRunInt -u -d <course> <unit> <exercise>
            Tools/CRunInt -t <course> <unit> <exercise>
            Tools/CRunInt -u <course> <unit> <exercise>
            Tools/CRunInt -s <course> <unit> <exercise>

       Description:
            Run an interpreted program, not tested

  CRunProg

       Called as:
            Tools/CRunProg -t -d <course> <unit> <exercise>
            Tools/CRunProg -u -d <course> <unit> <exercise>
            Tools/CRunProg -t <course> <unit> <exercise>
            Tools/CRunProg -u <course> <unit> <exercise>
            Tools/CRunProg -s <course> <unit> <exercise>

       Description:
            Run a compiled progrsm
            -t : run teacher's program
            -u : run user's program
            -d : against test data
```

```
          -s : show test data

    CSeeMks

         Called as:
              Tools/CSeeMks -x <course> <unit> <exercise>

         Description:
              Inspect marks of student, exercise or course
              call by
                 $0 -s aar prg1
                 $0 -x prg1 2 5
                 $0 -c prg1

    CSetNW

         Called as:
              Tools/CSetNW -d <course> <unit> <exercise>
              Tools/CSetNW -o <course> <unit> <exercise>
              Tools/CSetNW -l <course> <unit> <exercise>
              Tools/CSetNW -c <course> <unit> <exercise>
              Tools/CSetNW -r <course> <unit> <exercise>
              Tools/CSetNW -p <course> <unit> <exercise>

         Description:
              Set exercise as default, open, late, close, private or
    public.

    CSetProg

         Called as:
              Tools/CSetProg <course> <unit> <exercise> <suffix>

         Description:
              Setup, args are course, unit, exercise, suffix

    CSetWgt

         Called as:
              Tools/CSetWgt -t <course> <unit> <exercise> <suffix>
              Tools/CSetWgt -c <course> <unit> <exercise> <suffix>

              Tools/CSetWgt -f <course> <unit> <exercise> <suffix>
              Tools/CSetWgt -d <course> <unit> <exercise>

         Description:
              Args art course, unit, exercise, suffix

    CSetWgtI

         Called as:
              Tools/CSetWgtI -t <course> <unit> <exercise> <suffix>
              Tools/CSetWgtI -c <course> <unit> <exercise> <suffix>
              Tools/CSetWgtI -f <course> <unit> <exercise> <suffix>
              Tools/CSetWgtI -d <course> <unit> <exercise> <suffix>

         Description:
              For interpreted programs, not tested
```

```
CShlPrcs

     Called as:
          Tools/CShlPrcs

     Description:
          Various shell functions for use elsewhere

CSubInt

     Called as:
          Tools/CSubInt -s <course> <unit> <exercise> <suffix>
          Tools/CSubInt -c <course> <unit> <exercise> <suffix>

     Description:
          Submit interpreted program
          args are course, unit, exerise, suffix
          Not tested

CSubProg

     Called as:
          Tools/CSubProg -s <course> <unit> <exercise> <suffix>
          Tools/CSubProg -c <course> <unit> <exercise> <suffix>

     Description:
          Submit compiled program
          "-s" to submit.
          "-c" to check submission.

CSumStud

     Called as:
          Tools/CSumStud

     Description:
          Summarise one student

CVCompil

     Called as:
          CVCompil <source> <executable>

     Description:
          As for CCompile, but compile a  student  source  program
ver-
          bosely

CViewNts

     Called as:
          Tools/CViewNts <course> <unit>

     Description:
          View notes

CViewPap
```

```
     Called as:
          Tools/CViewPap <paper>

     Description:
          View paper.

CViewPrt

     Called as:
          Tools/CViewPrt $QUEST $ANS

     Description:
          View or print file
          filename is $1,  $2 is "p" to print

CViewReg

     Called as:
          Tools/CViewReg -t -P$PRINTER <course>
          Tools/CViewReg -v -P$PRINTER <course>

     Description:
          View register
          "-v" to view all students
          "-t" to view tutees only

CViewWrk

     Called as:
          Tools/CViewWrk -s <course> <unit> <exercise> $STUD

     Description:
          View students work
          "-s" to view all student
          "-t" to view tutees only

CZapCse

     Called as:
          Tools/CZapCse <course>

     Description:
          Shell script to reset an existing course to empty. It removes
all student
          solutions, marks files, and the student register and resets
all permissions.
          The course to be deleted is passed to the script as an
argument.

Appendix 2 : Programs in ~ceilidh/Tools

audit

     Called as:
          Tools/audit -e Error "%s logname %s",
          Tools/audit -e Error "%s open",
          Tools/audit <category> <message>
```

```
            Tools/audit <cse> <unit> <ex> <user> <audit cat> <audit mess>
            Tools/audit Admin "TchrCse"
            Tools/audit Audit "$NF"
            Tools/audit Comment "$TYPE"
            Tools/audit Copyout ""
            Tools/audit CrsSumm ""
            Tools/audit Develop "Comp"

       Description:
            SUID program to audit CEILIDH command.
            Call as either
                 $C_HOME/Tools/audit <audit category> <audit message>
            e.g.
                 $C_HOME/Tools/audit Compile verbose
            to audit category "Compile" appending "marks"-type record

  <cse>:<unit>:<ex>:<user>:<date>:<user>:<category>:<message>
            to the file
                 ~ceilidh/audit/Compile
            if it exists.

  ccef

       Called as:
            Tools/ccef -Ac $FILE <course>
            Tools/ccef -Ac closed <course>
            Tools/ccef -Ac students <course>
            Tools/ccef -Ac weights <course>
            Tools/ccef -Ar closed <course> <unit> <exercise>
            Tools/ccef -Ar closed.dat <course> <unit> <exercise>
            Tools/ccef -Ar late <course> <unit> <exercise>
            Tools/ccef -Ar late.dat <course> <unit> <exercise>
            Tools/ccef -Ax late <course> <unit> <exercise>
            Tools/ccef -a <course> <unit> <exercise> $USER mk
            Tools/ccef -o <course> <unit> <exercise> $USER <suffix>
            Tools/ccef -Ac default <course>
            date | Tools/ccef -Ax closed.dat <course> <unit> <exercise>
            date | Tools/ccef -Ax late.dat <course> <unit> <exercise>
            Tools/ccef -Ac motd <course>

       Description:
            SUID program to copy from standard input to the CEILIDH
  filesystem.
            (i) Call with flag -o to overwrite, or -a to append,
  compulsory
                 or -t for time delay test
                Reads text from standard input, then
                   arg1 is course(e.g. prg1)
                   arg2 is unit(e.g. 2)

                   arg3 is exercise (e.g. 5)
                   arg4 is their logname
                   arg5 is the extension, (e.g. "C" or "pas")
                If file ~ceilidh/course../unit../ex../late exists,
                 query user and offer late submissions,
                 append username to file
                If file ~ceilidh/course../unit../ex../closed exists,
                 offer nothing
```

```
                option flag -f forces whether or not "late" file exists
                option flag -t just tests the time gap
                option flag -g120
                 to set minimum inter-submission gap to 120 seconds
                    or recompile with MIN_GAP reset
                option flag -s10000
                 to set maximum size of saved file to 10000 bytes
                    or recompile with MAX_CHARS reset

            (ii) Flag -A? for various course administrator functions
                 checks user's login name is on "teacher.lst" of the
course
                 -Ac = course admin (motd, weights, scales)
                 -Au = unit admin
                 -Ax = exercise admin (late, closed)
                 arg1 = filename, arg2 = course, arg3 = unit arg3 = ex

compl

     Called as:
          Tools/compl -m $SRC
          Tools/compl -m < $SRC
          Tools/compl -v2 -x <exercise directory>/model.cm
$TMPDIR/*.<suffix>
          cat $SRC | Tools/compl -v2 -x <exercise>D/model.cm

     Description:
          Complexity analysis of C++ source on standard input
          Uses model figures in file "-x mval" (default "model.cm"),
essential
          Uses weightings in file "-f cv" if supplied
          Flag -v2 for verbosity 2 etc
              Verbosity0 : only single result
               2 : each metric
               3 : give other data values

          Flag -w : write a default "cv" file
          Flag -p : write an tweaked "cv" to give 100% (minimum
tweaking)
          Flag -o : write an tweaked "cv" to give 100% mark
          Flag -m : write away as model metrics in model.cm
          Flag -f : read parameters from next arg
          Flag -x : read model metrics from next arg

copyin

     Called as:
          Tools/copyin < $SOLN

     Description:

          Copy user file to system area,
          substitute appropriate "#include ..." and ".so ..." lines

dyncorr

     Called as:
          for
```

```
                all dynamic tests
            done | Tools/dyncorr $FLAG $DVAL

        Description:
            Read output from dynamic mark script,
                Tools/CDynCorr
            and print nicely.
            Should really have more formats

    efread

        Called as:
            Tools/efread <filename> | tail -1

        Description:
            SUID program to allow user to read their file
            Permit read if user is staff or tutor or filename starts with
    logname
            Or read your own entry from a combined "marks" file

    extract

        Called as:
            Tools/extract < prog$2$3.C | $C_HOME/Tools/oracle
    <exercise>D/model.ft
            cat $SRC | Tools/extract | $C_HOME/Tools/oracle -v2 $FEAT

        Description:
            Extract from standard input all (or most!) mode, char
    denotations
            and comments
            Then they can be analysed for program structure, and some
    quick
            statistics.
            Flag -s extract only the strings, one per line
                -p only the primed bits
                -c only the C comments
                -C only the C++ comments

    mark

        Called as:
            Tools/mark -v1 -m -s <course> <unit> <exercise> $PROG
            Tools/mark -v1 -m -s <course> <unit> <exercise> $PROG
    .<suffix>
            Tools/mark -v1 -m <course> <unit> <exercise> $PROG ""
            Tools/mark -v1 -m <course> <unit> <exercise> $PROG .<suffix>
            Tools/mark -v1 -mm -q <course> <unit> <exercise> <executable>
    $STUD
            Tools/mark -v1 -q <course> <unit> <exercise>
            Tools/mark -v1 -q <course> <unit> <exercise> <executable>
    .<suffix>
            Tools/mark -v1 -u $STUD -m -q <course> <unit> <exercise>
    <executable>
            Tools/mark -w

        Description:
            Program to call marking commands and saving commands
```

```
          as required by the teacher.
          Call by
               mark pr1 2 1
          and it will expect to find files such as
               "prog21.C" (for typographic tests)
               "~ceilidh/course../unit../ex../model.cm" and "prog21.C"
               (for complexity)
               executable "prog21" & various oracle files (for dynamic
tests)
                  "mark.act" contains e.g.
          Use flag
               mark -p ef prog21 :  to use local source ef.C with
prog21
               mark -s prog21 :  to do the saves.
               mark -g5 :  set gap of 5 seconds
               mark -u <logname> :  stores marks with the username
field
                                          set to <logname>

mchoice

     Called as:
          mchoice <file>

     Description:
          Oracle for marking multiple choice answers
          argv[1] = file (no public read) containing e.g.
              10:a
              5:b
              0:c
              5:d
          Standard input is the student answer.
          Read one visible char, score it as it matches a given answer.
          Score zero if it doesn't.
          Print e.g.
              Score 67
          as a percentage.

mmulti

     Called as:
          Tools/mmulti <course> <unit> <exercise> <logname>

     Description:
          Program for multi choice marking.
          It reads the student's archived solutions from
               ~ceilidh/course.../unit.../ex.../solns/<logname>.mc
          and the script from file
               ~ceilidh/course.../unit.../ex.../model.mc
          which should be in the format
          We expect the script to be non-public readable, and this
program
          to be SUID in normal use.
          Flags are
               -v<integer> verbosity level
                   level 0: print nothing

                   level 1: print total marks
```

```
                        level 2: print each questions marks
                        level 3: print lots
                -q  just list the questions

numeric

     Called as:
          numeric <file>

     Description:
          Numeric oracle for the question/answer system
          argv[1] = file (no public read) containing e.g.
               5:10
               12 14
               1 2 3
               12
               5 6 7 8
          The meaning is as follows:
               10 marks per number read unless a mark (e.g. 5: above)
is specified
               If one number on the line: exact match required
               If 2 numbers: student value must lie between the two.
               If 3 numbers: interpolate, full marks for middle value,
zero at the
                     others
               If four numbers: full marks between the middle two etc
          Standard input is the student answer.

oracle

     Called as:
          Tools/extract < prog$2$3.C | $C_HOME/Tools/oracle
<exercise>D/model.ft
          cat $SRC | Tools/extract | $C_HOME/Tools/oracle -v2 $FEAT

     Description:
          Reads from arg1 a file of regular expressions (REs), to act
as an
          oracle. They are converted to an "awk" program, which then
reads text
          from standard input.
          Flags are
               "-v1" causes awk program to print subtotals.
               "-v2" prints the REs as well
          A numeric first argument
               oracle 20
          sets the default mark to 20.
          The program will normally be SUID to enable the REs to be
hidden from
          the user.

qmulti

     Called as:
          Tools/qmulti <course> <unit> <exercise>
          Tools/qmulti -q <course> <unit> <exercise>
          Tools/qmulti -Q1 <course> <unit> <exercise>
```

```
     Description:

          Program to display question and get answer for QA exercises.
          It works from file
               ~ceilidh/course .../unit.../ex.../model.mc
          We expect the script to be non-public readable, and this
program
          to be SUID in normal use.
          Flags are
               -v<integer> verbosity level
                   level 2: print each questions marks
               -qjust list the questions
                  -Q2     just print question 2

qu-a-co

     Called as:
          Tools/qu-a-co -m <course> <unit> <exercise> $USER

     Description:
          Calls
               Tools/CQAMark
          with SUID

register

     Called as:
          Tools/register -f -v <course> "$LOG" "$NAME"
          Tools/register -f -v <course> $LOG
          Tools/register -v -f <course> $LOG
          Tools/register <course> $USER

     Description:
          SUID program to register student to given course in the
CEILIDH filesystem.
          e.g.
               register <course> <logname>
          to register on course

run

     Called as:
          Tools/run /bin/sh < $SHLL > $PROGOUT 2>&1
          Tools/run <executable> > $PROGOUT 2>&1 < $DATA

     Description:
          Type
               run <executable prog>
          to run the named program, and kill it after a default number
of
          seconds if it is still running.  Default is 5 seconds.
          Type
               run -10 <executable>
          to kill after 10 seconds.

setup

     Called as:
```

```
          Tools/setup -v3 -f <exercise directory>/setup.act <course>
<unit> <exercise>

          Tools/setup -v3 <course> <unit> <exercise>

     Description:
          Program to call marking commands and saving commands
          as required by the teacher.
          Call by
               setup prg1 2 5
          and it will expect to find file setup.act.
          The following entry in setup.act
                  Copy prog.sk prog.C
                  Copy header.h
          will copy "prog.sk" in the coursework directory to "prog.C"
          in the user's directory, and "header.h" to "header.h".
          A default "copies" file is searched for in the coursework
directory.

typog

     Called as:
          cat $SRC | Tools/typog -v2
          cat $TMPDIR/* | Tools/typog -v3

     Description:
          Typographic analysis of C++ source on standard input
          Uses weightings in file "argv[1]" is supplied
          Verbosity0,1 : only single result
               2 : each metric with non-zero weight
               3 : all metrics
               4 : give other data values
          Feedback1 : show position of indent_errors
               2 : show metric of max loss of marks
          Optional argument is name of a "tv" file in format such as
          Flag -w : write a default "tv" file
          Flag -p : write an tweaked "tv" to give 100% (minimum
tweaking)
          Flag -o : write an better tweaked tweaked "tv" to give 100%
mark

vmarks

     Called as:
          Tools/vmarks $VERB -c<course> $USER | $PAGER
          Tools/vmarks -c<course> -v1 $STUD
          Tools/vmarks -v2 -c<course> $STUD | $PAGER

     Description:
          View marks.
          Call with "-v1" flag for verbose, "-v3" for more so
              -v0 : just summaries
              -v1 : unit/ex marks
              -v2 : every attempt mark
              -v3 : ups/downs summaries
          Other flags
              vmarks -cpr1all studs course "pr1" 1 line per student
              vmarks -cpr1 zvcjust student zvc 1 line summary
```

```
              vmarks -cpr1 -v1 zvcjust student zvc each exercise
              vmarks -cpr1 -v2 zvcjust student zvc all attempts

              vmarks -cpr1 8 3just exercise 8 3 for all students
              vmarks -cpr1 -v2 8 3just exercise 8 3 all attempts
              vmarks -cpr1 -g zvcgnuplot file for nhg (marks) for
student zvc
              vmarks -cpr1 -g -n zvc  gnuplot file for nhg (natts) for
student zvc
          Needs "weights" and "scales" files
```

Appendix 3 : Shell scripts in ~ceilidh/bin.mnu

CEILIDH
This is the main calling script at the top level. When the user
issues
a sc (set course) command it calls

CStudCse
This is the general course level menu. The command sx leads to

CStudEx
which looks at the type of exercise and calls one of

CStudComp (compiled language)
CStudInt (interpreted language)
CStudQA (question/answer)
CStudEs (essay)

depending.
The command tutor (if available) leads into

CTutrCse
This has no subsidary menus.
The command teach leads into

CTchrCse
which again has no subsidary menus.
The command develop leads into

CDevlCse
Within this, the command sx (set exercise) leads into

CDevlEx
which looks at the exercise type and passes control
to
one of

CDevlCom
CDevlInt
CDevlQA
CDevlEs

as appropriate.

CHelp

CSetType

```
CSetUp

CShlPrcs

CShlVars

               Appendix 4 : Source code for xceilidh: in
~ceilidh/bin.x

               CEILIDH
                    This the main calling script at the top level
which
                    set  the  environment  variables  for the system
and
                    then calls xceilidh.

               The  source  code  which  build  xceilidh  is  stored
in
               ~ceilidh/bin.x/SOURCE.   The   files  in  this
directory
               includes:

               xceilidh.h
                    Contains all the global declaration for xceilidh.

               xceilidh.c
                    The main program for xceilidh and  manage  the
main
                    menu for the system.

               xcomp.c
                    Manage the "do exercise" menu if the  exercise
type
                    is either COMP or EXAMPLE.

               xessay.c
                    Manage the "do exercise" menu if the  exercise
type
                    is ESSAY.

               xmulti.c
                    Manage the "do exercise" menu if the  exercise
type
                    is QA.

               xadmin.c
                    Manage the system admin's menu.

               xcdev.c
                    Manage the course developer's menu.

               xcadmin.c
                    Manage the course admin's menu.

               xctutor.c
                    Manage the course tutor's menu.
```

xhelp.c
Manage the help facilities throughout the system.
The help window for each help button within the sys-
tem is built by "BuildHelp".

xmsg.c
Manage the message facilities throughout the system.
There are three types of messages:

simple message:which is handled by the function
"Message".

message which requires which is handled by
function "Message2".

multi pages message:which is handled by func-
tion "MessageF".

xinit.c
Initialise the system. These includes functions:

SetCeilidh:Set the system.

SetCourse:Set a course. This is called either
by "SetCeilidh" or by "SelectBoxCB" when
another course is selected.

SetUnit:Set a unit. This is called either by
"SetCourse" or by "SelectBoxCB" when
another unit is selected.

SetExercise:Set an exercise. This is called
either by "SetUnit" or by "SelectBoxCB"
when another exercise is selected.

xutiliti.c
Manage low-level utilities for the system.

References

1. Steve Benford, Edmund Burke, and Eric Foxley, Cour-
seware to support the teaching of programming, TLTP
Conference, University of Kent at Canterbury, 1992.

2. Eric Foxley, Question/answer exercises in
Ceilidh,
LTR Report, Computer Science Dept,
Nottingham
University, 1993.

3. Abdullah Mohd Zin and Eric Foxley, The oracle
pro-
gram, LTR Report, Computer Science Dept,
Nottingham
University, 1992.

The "oracle" program

Abdullah Mohd Zin
Eric Foxley

Department of Computer Science
University of Nottingham
NOTTINGHAM NG7 2RD, UK
email: amz, ef @ cs.nott.ac.uk

Introduction

A program which recognises whether a given piece of text contains a particular required meaning is called an "oracle".[3] This type of activity is vital in several areas of CEILIDH. It is used in CEILIDH to check that the output from the dynamic tests of a program represent "correct" output,[1] that program sources do or do not contain particular features, and that the answers to multiple choice questions contain appropriate valid words.[2]

Implementation

The implementation uses the UNIX concept of a Regular Expression or "RE". REs are involved in many UNIX commands, text editors, the "sed" stream editor, the "grep" family of commands, and "awk".

To check program output, the teacher provides a number of regular expressions which will be searched for in the text. These REs will be given one-per-line in a file whose name is supplied as the first argument in a call of the oracle program.

The implementation of the "oracle" program uses "awk" to detect the REs, so exact details of the RE formats are to be found in "awk" documentation.

The program "oracle" reads from its first argument a file of regular expressions (REs), to act as an oracle. They are converted to selection strings for an "awk" program, which then reads the text to be analysed from its standard input.

The output of the "awk" program is of the form

```
     Score  93
```

which is a percentage of the mark awarded out of the max possible mark
depending which REs were found.

The REs must be awk-type REs, see "awk" documentation for exact details.

Each line of the file of REs is of the form

```
     RE          default 10 marks awarded if RE found at least once,

                 no marks awarded of not found.

     Examples

     127         the string "127"
     127.3       the string "127"
                 followed by any one character,
                 followed by a "3"
     127\.3      the string "127.3"
     cat|dog     the string "cat" or the string "dog"
     [Ff]e*t     the string "Ft", "ft", "Feet" or "feet"
     ^cat        a line starting with the string "cat"
     cat$         a line ending with the string "cat"
     ^cat$       a line consisting of exactly the string "cat"
```

Extensions to the REs

The RE may be preceded by colon separated fields such as

```
 5:RE        5 marks awarded if RE found.
             The default is 10 marks.

 15:>5:RE    15 marks awarded if > 5 occurrences found,
             zero marks if <= 5.

 10:>=5:RE   10 marks awarded if >= 5 occurrences found,
             zero marks if < 5.

 <5:RE       Default marks awarded if < 5 occurrences found,
             zero marks if >= 5.

 <=5:RE      Default marks awarded if <= 5 occurrences found,
             zero marks if > 5.

 ==1:RE      Default marks awarded if exactly 1 occurrence found.

 !=1:RE      Default marks awarded if more or less than 1 occurrence.

 >=4-10:RE   Default marks if >=10, zero marks if <=4,
             interpolated marks if between 4 and 10.

 +:RE        This RE must occur AFTER the previous RE.

 -:RE        The previous RE must NOT have been found before this one.

 ~10:RE      If the RE is found, 10 marks are taken away.
             The 10 marks are not included in the maximum total
```

```
             out of which the mark gained is scaled as a percentage.
             Marks will never go negative.

Any colon required in the RE must be escaped.

The maximum mark (out of which the awarded percentage is calculated)
is
the  total  of all the positive possible marks, default 10 per RE
unless
otherwise specified.

If there is a line such as
     :50,20;80,40;90,60

in the oracle file (starting  with  a  colon,  consisting  of  pairs
of
integers) the percentage will be piecewise linear scaled, so that
  0   goes to   0
 50   goes to   20
 80   goes to   40
 90   goes to   60
100   goes to   100
with linear interpolation between these points.  If  there  are  10
REs
consisting of error messages which must be absent such as
     ==0:delivers a random

you may wish to scale the marks so that the presence of any one of
them
immediately reduces the mark to, say 50%.

Any prime "'" in an RE is currently converted to  a  dot  "."  to
avoid
shell  script  problems  driving  the  "awk" program.  We should be
more
clever and escape it properly.

Flags to the oracle program

Flag "-v1" causes the "awk" program to print subtotals.

Flag "-v2" prints the actual REs it is searching for, and then for
each
one  tabulates  the  required  minimum and maximum number of
occurrences
demanded by the oracle file, the actual number of occurrences found,
the
awarded  score, the maximum score (out of), and the accumulated score
so
far, and the marks lost on this RE.

Flag "-v3" prints the "awk" program which  has  been  generated  to
the
screen.

A numeric first non-flag argument such as
     oracle 20 RE_FILE
```

```
sets the default mark per RE to 20.

SUID facility

The program will normally be SUID to enable the file of  REs  (which
is
opened  from  within  the  program) to be kept hidden from the user.
The
files of REs will generally have no public read permission.

The use of oracles in CEILIDH

Oracles are used in four separate areas of CEILIDH as a convenient
means
of checking general text.

1: Dynamic test output

The program's output must be examined to see if the student  has
solved
the  given  problem correctly. The more precisely the question
specifies
the output format, the easier the oracle.  Generating  flexible  REs
to
allow flexible output formats requires thought and experience.  To
avoid
problems, you may choose to specify the output format very  exactly,
or
to  give  the  necessary  print  commands  to the student in the
program
skeleton.

For "Convert feet and inches to  centimetres"  I  have  several
dynamic
tests.

     The oracle for the first test looks simply for the correct
numeric
     output value, the RE is perhaps
          134\.57

     (but beware of rounding errors and numbers of  decimal  places,
so
     perhaps the RE should be
          134\.(6|57)|135

     Later tests look additionally for the text "cms"  or
"centimetres"
     which any civilised output should contain.

     Later tests check that the input values have been printed as in
the
     output, as in
          12 feet 3.4 inches converts to 12345.67 cms

     so checks also for the input values and text such as "fe*t".

     We may also wish to check for error messages in appropriate cases:
```

```
          20:[Ee]rror

     and to ensure their absence in outher cases:
          ==0:[Ee]rror

     to ensure that the student does not always print "Error"!
```

The standard and error outputs are currently combined for the oracle.

Flexibility causes the teacher more trouble, but is educationally much better.

2: Program features

For particular programs, the program features oracle will check that, for example, constants such as 2.54 (convert centimetres to inches) occur exactly once in the source, using the RE line

```
     ==1:2\.54
```

(where the "==1:" prefix requires exactly one occurrence) in the "model.ft" oracle.

This oracle is set by the "sf" teacher option, and should represent any features which the lecturer would look for "by eye" in hand marking.

The contents of comments and strings are NOT passed to this oracle.

3: Program structure

In the structure marking feature, the C programs are run through the "lint" checker, and the output searched by an oracle for particular warning messages. For C++ we use the "g++ -Wall" option. A typical oracle would be

```
     ==0:Undefined symbol.*referenced from text segment
     ==0:undeclared
     ==0:redeclared as different kind of symbol
     ==0:assignment of integer from pointer lacks a cast
     ==0:comparison is always 1 due to limited range of data type
     ==0:data definition lacks type or storage class
     ==0:defined but not used
     ==0:float or double assigned to integer data type
     ==0:previous declaration of
     ==0:statement with no effect
     ==0:unused variable
     ==0:value computed is not used
     :90,40;100,100
```

to give 100 marks for no warnings, 40 marks if there is 1 warning, less if there are more. The particular messages depend, of course, on

your
program analyser.

4: Multi-choice questions

The script for a multi-choice problem include the text displayed to the
student, followed after each "page" displayed by an oracle line to
recognise the output.

If the text says "Type a, b, c or d:" then you should recognise the
answer "b" using the RE

```
^b$
```

rather than simply

```
b
```

to ensure that the student cannot type the reply

```
abcd
```

Most student answers are one-line replies. If the multi-line option is
used, the complete text is put onto one line before the oracle is
applied.

If you are looking for a given word, give all possible alternatives,
such as

```
(ferrous|iron) *oxide
```

Other oracles are available for the question-answer system, see its
document for details.

References

1. Steve Benford, Edmund Burke, and Eric Foxley, Teacher's Guide to the Ceilidh System, LTR Report, Computer Science Dept, Nottingham University, 1992.

2. Eric Foxley, Question/answer exercises in Ceilidh, LTR Report, Computer Science Dept, Nottingham University, 1993.

3. E. W. Weyuker, "On Testing non-testable program", The Computer Journal 25(4), pp.465-470 (Nov 1982).

Department of Computer Science

Policy on Plagiarism and Late Handing in of Work

In order to help you successfully pass your programming courses, we need
to make a few basic ground rules clear. These rules concern plagiarism
(i.e. copying other people's work) and late handing-in of work. The pla-
giarism policy applies to non-programming courses as well.

Plagiarism

This means copying work and pretending that it is yours. Plagiarism is
not allowed. Any student who plagiarises the work of others will be
reported to the Academic Offences Committee via the Registrar and Head
of Department.

The following actions are considered to be plagiarism:

1 copying paragraphs or programs from a textbook;

2 copying another person's work either with or without their
knowledge;

3 working together in groups of two or more to produce a single pro-
gram or essay and then each member of the group submitting a copy
of this as their own work.

You can stop other people copying your work by checking that your file
permissions are set properly and by not leaving printouts lying around
on or near the printers.

Working in groups is acceptable, and encouraged for some subjects, pro-
vided that the group work does not lead to a finished program or essay.
It is important to understand that a program or essay which is submitted
as coursework must contain a sufficient amount of your own effort to
make it your work rather than the group's work. More specifically, you
may want to discuss possibilities for algorithms and data structures
that might be appropriate to some particular piece of coursework, and
this is perfectly acceptable. But when you go to the machine you must

code up your *own* solution to the problem. You must *not* use a
friend's program as a template or a short cut towards your own solution.
In other words, any work you hand in should clearly be your own and
should not be a joint effort. Essays can include paragraphs copied from
a text book providing you acknowledge the source and list all references
in the essay; this shows that you are reading round the subject as you
should.

We also need to tell you that we have an AUTOMATIC PLAGIARISM DETECTOR
which is capable of comparing all students programs across an entire
course. The detector is very sophisticated and is capable of spotting
programs that are identical, nearly identical or just similar. We always
check your work with this tool and have spotted may cases of plagiarism

in previous years.

Late Handing in of Work

Our policy for late programming work is different to the standard
university policy. The standard policy reduces marks by 5% for each day
a piece of work is late. Our policy is much stricter. If you hand in
your programming solutions late then you will get 0 marks for them.
This is because we make model solutions available at the end of exer-
cises, which would allow you to copy them. Please, do not hand in work
late. If you think that you are going to be late for a good reason,
then tell the course lecturer and your tutor BEFORE the deadline.

In summary, don't copy and don't hand in late. We are sorry to be so
blunt, but it is important that everybody understands the rules we are
working under. This is particularly true on coursework assessed modules.
Remember, consult a member of staff if in doubt.

Question/answer exercises in Ceilidh

Eric Foxley

Department of Computer Science
University of Nottingham
NOTTINGHAM NG7 2RD, UK
email: ef @ cs.nott.ac.uk

Introduction

The primary method of student assessment in the Ceilidh system is by the automatic marking of programs usinxg a variety of static and dynamic tests. The Ceilidh system also includes a general question/answer type of exercise, allowing the teacher to set simple "question/answer" tests easily. Such exercises can include multiple-choice questions, questions with a single word as the answer, questions with numeric answers, and questions whose answer is a simple sentence. The exercise can be used for marking, or purely as an information gathering system (as in the provision of an end-of-course questionnaire).

The controlling program for question/answer marking lives with the other Ceilidh commands in the directory ~ceilidh/bin , and is called multi .
As part of its operation, it calls a variety of other programs to check student responses in particular ways, at present mchoice (to check straightforward multiple choice answers), numeric (for numerical answers), oracle (for single word answers), and semantic (for single sentence answers).

The multi program is driven from a controlling script which must have been set up within the exercise directory as part of the Ceilidh course structure. The program is called with the course, unit and exercise numbers as arguments, as in

```
multi <course> <unit> <exercise>
```

This will operate by reading a controlling script, which is stored in the file ~ceilidh/course.../unit.../ex.../model.mc . This script file contains both the information and questions to be presented to the student, and the information for the marking processes to be used in analysing the student responses. The general format of a script file is typified by

```
>>>>
Lesson 1
What is 1 + 2 ?
Type 1 2 3 4 or 5:
<<<< 10 M
10:3
>>>>
Lesson 2
What is the middle letter of cat ?
Type a b c d or e:
<<<< 10 M

10:a
>>>>
Lesson 3
Is EF more handsome than SDB?
Type yes:
<<<< 10 M
10:yes
>>>>
```

Generally, information from a >>>> line up to the next <<<< line is presented to the student, while information from the <<<< line to the next >>>> line is for the marking process. Further information for the marking process appears on the <<<< line. The exact format of these scripts is detailed in the next section.

1. The format of the scripts

The format of the driving script file is as follows.

Text from the line starting

>>>>

up to the line

<<<<

will contain information for the student, and will end with a question. This information will be displayed*

===

Note: * A facility for the specification of one of a variety of presentation programs may be introduced later.

===

on the user's screen exactly as written. The final <newline> is not printed, and a colon is appended to the last line, on the assumption it represents a prompt for the student.

Text from the line starting

```
<<<<
```

ending the lesson up to the next

```
>>>>
```

is the data to be used by the marking process to mark the student response. Additional information on the <<<< line specifies a mark out of which this question is to be marked (if no mark is specified, the question is not marked), and a letter defining the type of marking to be used. The total mark awarded at the end is the total of the marks awarded divided by the maximum total possible mark.

2. The marking programs

There are a number of different marking programs available, the particular one to be called being specified by a single letter on the <<<< line. The possible letters and programs are as follows.

M : multiple-choice

If the multiple choice marker is indicated by lines such as

```
...
Type a, b, c, d or e:
<<<< 10 M
```

it will expect marking data immediately following the <<<< line in the form

```
100:c
50:d
```

This indicates that 100 marks are to be awarded for the answer "c", and 50 for the answer "d". There are no marks for other answers. This section of the script file will thus be

```
... Details of question ...
Type a, b, c, d or e:
<<<< 10 M
100:c
50:d
>>>>
Next displayed information ...
```

The number following the <<<< and preceding the "M" causes the mark awarded to be scaled out of (in this example) 10 towards the exercise total. If no number is present, a default of 10 is assumed.

A perfect answer is assumed to be represented by a mark of 100 within the marking information.

O : oracle

The "oracle" program is used[1] to recognise a correct answer in cases where a single word or short phrase is required in answer to the question. The data between the <<<< and >>>> lines is now a set of regular expression recognisers for the oracle program. One might have typically:

```
... murder story ...
Who killed the victim?
<<<< 10 O
100:[Ee]ric
40:[Ss]teve
>>>>
Next question ...
```

N : numeric

If the answer required is numeric, a numeric recogniser can be requested. The student input now is a series of numbers. The teacher input between the <<<< and >>>> lines is now a series of lines representing the values expected. Each line can contain up to four numbers:

One number: this represents the exact value required from the student. The student achieves full marks for the correct value, zero marks for any other.

Two numbers: full marks are awarded for any value typed by the student which lies between the two specified values.

Three numbers: first value (or any value below it) is awarded zero marks, the middle value is awarded perfect marks, the third value (or above) is awarded zero marks again. Between the outer values and the middle value marks are interpolated linearly.

Four values: as for the software metrics, zero marks up to the first value, interpolated up to the second value, full marks up to the third value, interpolated up to the fourth value, and zero above that.

student value	teacher values				mark awarded	comment
10	10				100	must be 10
11	10				0	
10	9	11			100	must be between 9 and 11
12	9	11			0	
10	6	10	14			100
8	6	10	14		50	interpolate between (6,0) and (10,100)
15	6	10	14		0	
20	10	20	30	40	100	20 to 30 is perfect
15	10	20	30	40	50	between 10 and 20 is interpolated
50	10	20	30	40	0	zero marks outside [10,40]

|

|

|

Values from the student must be in the correct sequence.

S : semantic

The most general purpose recogniser is the semantic recogniser. The teacher provides between the <<<< and >>>> lines a single sentence representing the correct answer to a question:

The object required must be small, brown and hairy.

The student's reply will be a sentence, whose semantic content is compared with that of the teacher's sentence. The semantic recogniser awards a mark of 100 for a perfect match, and zero for no apparent semantic correlation.

The last exercise

The script will normally end with

```
Last question ...
<<<< ....
Last oracle
>>>>
Thank you, that is the end of this exercise.
<<<< E
```

with an "E" on the final line to indicate the end of script.

3. Defining the student end-of-data

An additional letter on the <<<< line indicates the way the student is to end his/her data input. Normal answers will be terminated by end-of-line, expressed as

```
<<<< 10 M L
```

If the student is to permitted several lines of input (which will be concatenated before being passed to the marking program) use

```
<<<< 10 M B
```

for the student to type to a blank line.

Finally, of interest only to multiple choice questions, use

```
<<<< M R
```

for the student to just type a single character in cbreak mode. Since most multiple-choice answers will be a single character, the notation for multiple-choice answers will be

```
<<<< R a b c d e
10:d
5:b
>>>>
```

to indicate that the answer is "a" or "b" or ... or "e", and the student will be repeatedly prompted until one of these answers is given.

4. General

All lines in the script file starting "#" are completely ignored.

5. Implementation

The program qmulti in Tools is called as in

```
qmulti pr1 1 q
```

to answer course "pr1" unit "1" exercise "q". The student's answers to the questions will be stored in files

```
~ceilidh/course../unit../ex../solns/<logname>.q1
~ceilidh/course../unit../ex../solns/<logname>.q2
```

etc.

To obtain a single answer from the student to a single question, use

```
qmulti -Q3 pr1 1 q
```

for the third question of the exercise.

To consolidate the separate answer files into a single <logname>.mc

file
(in archive format), execute the CQAMark shell script as in
     ~ceilidh/Tools/CQAMark pr1 1 q <logname>

This consolidates the question files for the specified student in the specified exercise into one archive file, and marks the result, appending the marks record to the course../unit../ex../solns/marks file. The marking is this case is done by the mmulti program.

6. SUID facilities

We expect the script to be non-public readable (since it contains the answers to the questions), and this program to be SUID in normal use.

7. Marks

The marks will be stored in the usual Ceilidh format, i.e. in a file
     ~ceilidh/course.../unit.../ex.../solns/marks

There is one line per student marked, in the Ceilidh format of
     <course>:<unit>:<ex>:<student>:<date>:<entered by>:<total %>:<details>

```
pr1:2:5:prg:92.04.14-11.01.05:prg: 95:10:abs:30
ist:2:6:efx:92.04.11-12.14.08:efx: 75:10:10:10: 0:
```

The <details> fields may be the oracle mark (if a maximum mark has been specified) or the student response otherwise. For a long response, only the first 20 characters will be stored.

I haven't worked out a general result analysis command yet.

8. Program flags

Flags at the call level of" qmulti" are

```
-v<integer> verbosity level
     level 0: print nothing
     level 1: print to total mark
     level 2: print each question's marks
     level 3: print more

-q  just list the questions
-Q <number> : list the numbered question
```

9. Note

For reasons of copyright, the "semantic" program is not currently available on general release.

References

1. Abdullah Mohd Zin and Eric Foxley, The oracle program, LTR Report,
Computer Science Dept, Nottingham University, 1992.

Ceilidh Statistics Package

S D Benford, E K Burke, E Foxley
C A Gibbon, N H Gutteridge

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

1. Introduction

Ceilidh is an on-line coursework administration and auto-marking facil-
ity designed to help both students and staff with programming courses.
It helps students by informing them of the coursework required of them,
and by permitting them to submit their work on-line. When a student
submits a solution program the system automatically awards a mark for
the program. This mark is fed back to the student and stored in the
system for later reference. Facilities are available to teachers and
tutors to summarise the progress of students throughout a course in the
form of lists of marks.

In addition to these facilities a statistical analysis package has been
developed. The package runs on X terminals and displays information
graphically in the form of histograms.

2. The Package

On typing the command /cs/ceilidh/bin/stats the package will prompt the
user for a course name. The currectly active course in Computer Science
is

pr1

Once this has been entered the package will load all information on that
course into memory. The number of exercises to be stored and the number
of exercises loaded into memory are displayed as below.

```
Enter course name :pr1
Loading details on course "pr1" : Please Wait

Exercise   ##############
Done       #########
```

```
Once all information is loaded into memory, the following menu  will
be
displayed.
```

```
CEILIDH system: Statistics on Course "pr1"
 s   student statistics    | ls   list students
 x   exercise statistics   | lx   list exercises
 c   course statistics     | q    quit
Type your choice:
```

|
|
|
|
|
|
|
|
|

|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|

|

The statistics produced by the package are split into three
categories,
Course, Exercise and Student statistics.

```
     s    will take you into the student statistics menu.   The
package
          will prompt the user for a student username and then allow
the
          user to view statistics on that student (see Section 2.2).

     ls   will list details on all students studying the chosen
course.
          This  option  will  use  the  pager  specified  by  your
PAGER
          environment variable.

     x    will take you into the exercise statistics menu.  The
package
          will  prompt  the  user for a unit and exercise and then
allow
          the user to view statistics  on  that  exercise  (see
Section
          2.3).

     lx   will list all exercises stored in memory.

     c    will take you into the course  statistics  menu  (see
Section
          2.4).

     q    to quit out of the package.
```

2.1. Student Statistics

The histograms produced at this level allow the user to monitor the
pro-
gress of a student throughout a course. The user will be presented
with
the following menu:

```
 CEILIDH system: Statistics on Student "zvc" on Course pr1
  m    marks and class averages for each exercise
  a    number of attempts at each exercise
  r    development ratio for each exercise
  p    mark profile for an exercise
  pr   print student report
  q    quit
 Type your choice:
```

|

|

```
     m    will display the student's marks and class averages  for
each
          exercise.  This allows the user to view a students progress
in
```

```
          relation to the rest of the class and identify exercises
that
          the  student  had  problems with.  For each exercise, the
left
          hand bar (red?) shows the student mark,  the  right  hand
bar
          (blue?)  the class average.  It is easy to see how the
stident
          compares with the average.

     a    will display the student's number of attempts  at  each
exer-
          cise.  A large number of attempts at a particular exercise
may
          indicate the student was struggling with that exercise.

     r    will display the development ratio  for  each  exercise.
The
          development  ratio  for  an exercise indicates how the
student
          worked towards the final solution (see section 4).

     p    displays the mark profile for a named exercise.  It asks for
a

          unit  and exercise, and displays the marks awarded by the
sys-
          tem for each attempt the student made at the  exercise.
This
          allows the user to examine in more detail how a student
worked
          towards a final solution.

     pr   will print a one page report on the current students
progress.
          This report consists of the first three histograms and will
be
          sent to the printer specified by the user's LASER
environment
          variable.

     q    returns to the previous menu.

2.2.  Exercise Statistics

The histograms produced at this level allow the user to examine
progress
on  a  particular  exercise in more detail. On entering an exercise,
the
user will be presented with the following menu:
   _________________________________________________________________
    CEILIDH system: Statistics on Course pr1 Unit 1 Exercise 1
     m    Distribution of Marks
     ml   list students with marks less than
     mg   list students with marks greater than
     a    Number of students vs Number of Attempts
     ns   list Students who have not submitted this exercise
     ag   list Students who have made attempts greater than
```

```
     pr   print exercise report
     q    quit
    Type your choice:
```

```
     m    will display the distribution of marks for an exercise.
This
          allows  the user to observe the spread of marks awarded for
an
          exercise.

     ml   allows the user to identify students on  the  previous
histo-
          gram.  The package will ask for a mark, and will list all
stu-
          dents who obtained a mark less than or equal to this value
for
          the current exercise.

     mg   is similar to the previous command except that it  lists
stu-
          dents who achieved a mark greater than or equal to a
specified
          value.

     a    will display the number of students vs number of  attempts
at
          an  exercise.   This  allows  the  user  to view the number
of
          attempts people have been making at the current exercise.

     ns   lists students who have not submitted the current exercise.

     ag   prompts the user for a number of attempts and lists  all
stu-
          dents  who  have  made  attempts greater than or equal to
this
          value.  A large number of attempts for an exercise  may
indi-
          cate that the student is having dificulties.

     pr   will print a one page report on the  current  exercise.
This
          report  consists of both of the histograms mentioned above
and

          will be sent to the printer  specified  by  the  user's
LASER
          environment variable.

     q    returns to the initial menu.
```

2.3. Course Statistics

The statistics produced at this level give an overview of the

progress
of a course as a whole. On entering a course the user will be
presented
with the following menu

```
 CEILIDH system: Class Statistics for Course "pr1"
  m    class averages and standard deviations
  a    average number of attempts
  r    average development ratio
  s    number of students to submit each exercise
  d    distribution of average student marks
  al   list students with averages less than
  ag   list students with averages greater than
  pr   print course report
  q    quit
 Type your choice:
```

|

|

m will display the class average and standard deviation for
each
exercise in a course. This graph helps identify exercises
for
which the students are finding difficult to achieve
reasonable
marks.

a will display the average number of attempts for each
exercise
in a course. This can be used to identify exercises for
which
the students are making large numbers of submissions.
This
would indicate problems with that exercise.

r will display the average development ratio for each
exercise
in a course. Again this helps the user pinpoint problem
exer-
cises.

s will display the number of students who have submitted
each
exercise. This enables the user to determine exercises
few
students have attempted. This may affect the above
histo-
grams, since if few students (presumably the brightest)
have
submitted, the average may be unexpectedly high.

d will display the distribution of final student marks for
the
course. This takes into consideration weightings for
each
exercise as set by the course teacher.

```
     al   allows the user to identify students on  the  previous
histo-
          gram.   The  package will prompt the user for a mark, and
will
          list all students with average marks less  than  or  equal
to
          this mark.

     ag   is similar to the previous command except that it  lists
stu-
          dents with marks greater than or equal to a specified value.

Please note that all averages only consider students who have
submitted
work.

3.  How to run the Statistics Package

To run the package type
     /cs/ceilidh/bin/stats

from an X-terminal.  The package uses the  following  environment
vari-
ables:

     TERM to check the terminal is an X terminal;

     PAGER which should be set to the users preferred paging command;

     LASER should be set to the  name  of  the  PostScript  printer
any
     reports are to be sent.

4.  Development Ratio

One of the problems with providing on-line marking is that some
students
may  use this facility to blindly "tweak" their programs to achieve
max-
imum marks.  To help detect this problem we have defined  the
"develop-
ment ratio".

Development Ratio = (No. of times mark  increases  -  no.  of  times
it
decreases) / Number of attempts at an exercise

Range: 1 to -1

Development ratios above 0.5 suggest the student is working well
towards
a  final  solution,  around  0 suggest the student has been tweaking
the
solution, and below 0 suggests the student is having difficulty.
```

```
       Ceilidh System Fourth Release Version 2.2 : February 1994

System changes

     General
          Courses or individual exercises can now include in a "type"
file
              MAXSUB=20  to limit the number of submissions each
student may make
              MINGAP=30  to set the minimum seconds between submissions
              OUTOF=10   to cause all mark totals to be scaled out of
10
              VDATA=no   to inhibit viewing of the test data
              SAVEOUT=yes to cause the user's test output to be stored
for analysis
                    in solns/<user>.out
              ARFILES="hotel.C header.h booking.C"
                    causes all the named files to be saved in an
archive
                    format file in the solutions directory in a file
                        <user>.ar
              C_ORACLE=myoracle
                    to specify a different oracle
          Mark actions:
              Marking can now include "structure" marking, using "lint"
output for C,
               or "g++ -Wall" output for C++
              Marking can now include program run-time execution counts
using
               profiling on SUN C compilers
              The "mark.act" mark actions file can now specify more
complex
               storage of student results, including use of RCS to
               store all the submissions, and storage of executables.
              Question-answer marking uses "ar" to combine the answers.
          A new exercise type "MARK" for exercises used purely for mark
entry
              purposes.
          A new statistics package for staff/tutor use is released,
              using X-windows and graphics for the displays.
          There are new features built into the oracle for scaling
marks,
              and for subtracting marks for unwanted features.
              The oracle does not now require an "awk" supporting
functions.
          Better "default course unit exercise" system:
              Each user has a default course, and a separate default
              unit/exercise for each course they use.
          More efficient, less use of "<<" in the shell scripts.
          The installation script is simpler to use.

     Menu version
          Minor changes to the items available on the menus.
          More portability (but still not to COSIX).
          Better error handling
          Tutor menu: added
              mark and submit student work after exercise closure date
              remark a program already submitted
```

```
          The course developer menu is now more helpful
              Setting up exercises
              Editing notes if you have the master copy

     CLI version
          No changes

     X version
          First release, for SUNOS only, using X11r4 and Motif-v1.1.x.
It has not been tested under X11r5 or Motif-v1.2.x but there should not
be any problems with any system using these more up-to-date versions.

Course changes

C Course
     Many more exercises
     Execution count marking
     Structure marking
     Corrections to the notes

C++ course pr1
     Many more exercises
     Structure marking
     Minor corrections to the notes

C++ course pr2

Pascal course

===============================================
          Ceilidh System Third Release Version 2.1 : July 1993

Major changes are as follows.

There has been a complete rewrite of the system.  We now have

     A dumb terminal menu version in directory ~ceilidh/bin.mnu
control-
     ling  script  in file CEILIDH menus for students, tutors,
teachers,
     developers

     A command line  interface  version  in  directory
~ceilidh/bin.cli
     which  directory  must  be  in  your path if you wish to use it
See
     document "CLI" for details.

     An  X-window  interface  in  directory  ~ceilidh/bin.x
controlling
     script in file CEILIDH as a separate release.

The tools (called by all 3 interfaces) are in
     directory Tools/C* (shell scripts)
     directory Tools/[a-z]* (executables)
     directory Tools/SOURCES/*.[ch] (sources for executables)

The function of all tools is described in the "Design" document.
```

```
All file names have been adjusted for  MS-DOS  compatibility,  up  to
8
characters  before  the  dot,  at  most 3 afterwards. There may be a
few
exceptions still!

There is more/better documentation, including a design document.

An additional experimental course "tst" is distributed with examples
of
exercises of type
     question/answer

     text submission
     compiled languages
     interpreted languages
to encourage users to develop courses.

System changes:

The total system contents have been split as above.

Users are now split into more categories, each with their own
documena-
tion and menu.
     user = student
     tutor as before, read access to all marks
     teacher = course admin facilities,
        list of permitted users set up by the system administrator
     developer = course creation/editing facilities
        such a person must login to "ceilidh"
     system administrator

Every exercise now has a file "type" specifying the  type  of
exercise,
and the compiler, file suffix etc to be used, containing e.g.
     TYPE=COMP
     CC=g++
     SUFF=C

New exercise facilities include
     Interpreted language programs.
     Mark without submitting.
     User specified oracles.
     Course specific compilation, typography etc programs
         can now be stored in the course/bin directory.
     The marking actions file can now be more specific.
     The use for on-line submission of text documents
         by students is encouraged,
         to be marked and marks entered by hand.

See document "Develop" for details.

Question/answer improved There is a variety of  possible  other
oracles
for multi-choice and numeric answers. The new Oracle features also
help.
```

See document "Qu-ans" for details.

Extra Oracle features for sequencing values. See document "Oracle" for
details.

Closer teacher control over exercises. The teacher "open"s particular
exercises, then makes them "late", then "close"s them. The teacher is
encouraged to obtain overall metrics and plagiarism results for late and
closed exercises. Exercise results are given weighting factors in
overall calculations. Overall mark totals can then be piecewise scaled.
All this makes the display of marks much faster. Individual units of the
course can be closed or opened. See document "Teacher" for details.

Audit trail facilities have been added. The system administrator can

keep audit trails of particular commands (e.g. setup, mark ...) and/or
particular students (all activities) and inspect the records. Errors are
logged as an audit trail, not by email. See the document "Develop" for
details.

Courses:

C course:
- All exercises renumbered with mnemonics
- Additional exercises
- Expanded notes
- The course is still weak from unit 8 onwards, volunteers welcome!

Intro to C++ (pr1) course:
- Expanded notes, more exercises.

Further C++ (pr2) course:
- Added two exercises: the Pie Machine (assessed) and the Robot (essay).
- Bug fixes in the Hotel and Rational exercises to solve some minor problems.
- Correcting some mistakes in the notes.

Course "tst"
- This is just to demonstrate exercises of all types, so that you can start from an existing exercise. There are awk and shell programming exercises, question/answer, and text submission.

===

Ceilidh System Second Release Version 1.2 : Christmas 1992

```
The major changes to the first release are detailed below.

System:
     Many small changes and bug fixes
     Student commands:
     new student command "vm" (view marks, very slow!)
     Tutor commands:
     new command "vo" to view secret oracle files
     Teacher menu:
     improved teacher facilities for setting new exercises
     menu commands made more uniform with student menu
     Papers made more uniform,
     teacher guide expanded greatly
     student guide added

General:

     The "co" facility now works through email to the first named
person
     in  the  "staff.list" file for that course.  This requires that
the
     "staff.list" file be set up properly.

     Some defaults have been modified in the light of  experience.
max
     seconds  before  a  user process is killed from 5 to 10 seconds
max

     size of a saved text file from 20k to 50k minimum time between
mark
     submissions  from  60 to 300 seconds typographic metric ranges
have
     been widened

     Simple multiple choice exercise is now used for  student
question-
     naires.

     Essay type exercises can now be used for on-line  essay
submission
     (but no automatic marking!).

Courses:

     All courses now include a Ceilidh student questionnaire at the
end,
     please ask your students to complete it, and email us the
resulting
     "marks" file.

     prg1: C++ first course
     Notes in each unit greatly extended in "notes.cat" (for "more")
and
     "notes.ps" (PostScript).
     PostScript files "notes.ohp" added, suitable for overheads
     Many more exercises
     All skeletons mark more clearly where the student  must  amend
```

```
the
     skeleton program

     prg2: Further C++
     First release, covers classes and program design

     c:  Many small improvements, mostly due to  Heriot  Watt
comments:
     thank you!
     PostScript files "notes.ohp" added, suitable for overheads
     Some more exercises
     All skeletons mark more clearly where the student  must  amend
the
     skeleton program
     Still weak on exercises in later units (EF will be using it
locally
     next semester; expect improvements!)

===============================================
             Version 1.1 : First public release, June 1992.

System:
     As seen

Courses:
     prg1 (C++ introduction)
     c (C)
```

Mathematics Department

Introduction to Programming in C

Brief Summary Notes by Eric Foxley

Chapter 1 : Background

You will be given duplicated notes each week as part of the programming course. You should supplement these by reading your own books, or books in the library.

1.1. High level languages

There are many languages for programming computers; you may perhaps have used one already, at school or on a home computer. Which one have you used at school? BASIC? Pascal? None?

Old-fashioned serial languages include

BASICoften used by beginners on home computers,

FORTRANold but still popular with some scientists and engineers, new versions are introduced every few years,

Algol
an elegant little-used internationally designed language, whose features are being incorporated into other languages,

COBOL
a widely used and well standardised language used in commerce,

APL
an interactive scientific language with a very mathematical notation,

PL/I
a failed attempt by IBM at achieving an all-purpose language, now almost dead,

Pascal
good for beginners, often taught as a first language,

Modula2
a development of Pascal to make it more realistic for large programs, and to enable modern program design techniques to be used,

```
C
     a practical language, see below,

C++
     a development of C, see below, and

ADA
     a USA Department  of  Defense  standard,  now  adopted  by  the
UK
     Ministry  of  Defence also, aimed at safe programming for
real-time
     embedded systems.
```

There are now also many fourth generation languages or 4GLs, aimed at retrieving information from modern databases.

These languages are all essentially the same, they just involve different syntactic sugar , i.e. they are written using different formats, punctuation and grammars. The differences exist because the languages are aimed at different classes of users. For example those aimed at commercial users (COBOL, PL/I) will use more words and less symbols; we might see

```
     rate_per_hour multiplied by hours_worked gives gross_pay
```

rate_per_hour multiplied by hours_worked gives gross_pay

in COBOL, compared with

```
     pay = rate * hours
```

in a language for engineers.

Languages aimed at scientific numerical users (FORTRAN, Pascal) offer facilities for simplifying the handling of vectors and matrices, and for performing very accurate arithmetic.

In addition to being aimed at different categories of user, languages may have other different objectives, such as aiming particularly at beginners (BASIC, Pascal), or being particularly safe for large projects (ADA, Modula2).

All of these languages are a formal notation for you to give sequential instructions to the computer. The instructions in all of the above languages are sequential ("Do this, then do that ...") and explicit.

There are other very different types of languages such a Prolog (taught at Nottingham to Computer Science 2nd years) which are of a completely different nature. In Prolog for example you essentially describe your problem, and leave the computer to decide how best to solve it.

1.2. Compilation versus interpretation

A typical BASIC or APL program is interpreted; that is, each line is decoded and interpreted by the computer each time it is executed. An instruction which occurs inside a repeated loop may have to be interpreted many times. This wastes computer time, and causes the program to run relatively slowly. It has the questionable advantage that parts of the program which are not executed do not need to be interpreted, so that an incorrectly typed line may not be detected until it comes to be executed.

Programs in most of the other languages mentioned above are compiled; that is, the whole program is first analysed and digested by a compiler , which converts it into a machine executable form. This machine executable form runs much faster than an interpreted program, since all of the analysis of the program statements has been completed before any execution starts. If you request it, the compiler will spend additional time making the compiled program as efficient as possible.

The compilation is often in two distinct stages, first compiling your program into an object module, and then loading the object module into an executable program. At the loading stage, items from a library to perform certain standard operations (to handle networks, or to draw pictures, for example) may be combined with the program object module when it is loaded.

Language interpreters usually include some simple form of editor, such as the line numbering system in BASIC. Every line of the program starts with a number; the number determines the choice of a line to change, and the sequence in which lines are executed. Compiled languages use

programs which have been stored in ordinary text files. You can use any
editor to create the original text source file; the most commonly
available general UNIX text editor is vi , the most commonly used editor
on SUN computers is emacs . In the Nottingham University Mathematics
Department the preferred editor is the local extension by Dr Walker of
the ed editor. Your environment variable EDITOR should be set to refer
to your preferred editor.

1.3. A brief history of C and C++

Brian Kernighan et al (as they developed the UNIX computer operating
system) required a language for writing a computer system. At that time
(about 1970) most systems programs were written in assembly language for
reasons of efficiency. This involves expressing the problem in terms of
very low-level machine operations on a particular type of computer. The
resulting program is long, tedious, error prone, non-portable and
difficult to change. Brian Kernighan and his colleagues realised that
the use of Assembler was to be avoided at all costs.

A language called BCPL had been developed in the UK specifically for
writing computer systems. Brian Kernighan's first attempt at a language
was based on BCPL, adding various features to make it more useful, and
was called "B".

After some more experience, they decided that a new language was needed.
A new language was therefore developed, and was called "C". This was
used to write the next version of UNIX system software, which eventually
became the world's first portable operating system.

C has now become a widely used professional language for various
reasons.

1: It has high-level constructs.
2: It can handle low-level activities.
3: It produces efficient programs.
4: It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection.

The standard for C programs was originally the features set by Brian Kernighan. In order to make the language more internationally acceptable, an international standard was developed, ANSI C (American National Standards Institute).

Another group developed C to reflect modern developments in program design, in particular object-oriented programming. This language became "C++". C++ may be considered in several ways.

1: An extension of C.
2: A "data abstraction" improvement on C.
3: A base for "object-oriented" programming.

1.4. A minimal C program

It is high time that we stopped talking about programming languages, and saw an actual C program; the simplest possible program might be as follows.

```
/* Sample minimal program */
/* from EF's C notes */

main() {
    printf( "Hi there!\n" );
    exit ( 0 );
} /* end of program */
```

The text of the program as shown here would be stored in a file.

1.5. Creating, compiling and running your program

The stages of developing your C program are as follows.

1.5.1. Creating the program

Create a file containing the complete program, such as the above example, using any ordinary editor with which you are familiar such as vi or emacs or ed .

The filename must by convention end ".c" (full stop, lower case c), e.g. myprog.c or progtest.c . The contents must be as in the above example, starting with the line

```
/* Sample ....
```

or a blank line preceding it, and ending with the line

```
} /* end of program */
```

or a blank line following it.

1.5.2. Compilation

Compile your program with the command

```
cc program.c
```

where program.c is the name of the file.

If there are obvious errors in your program (such as typing

```
main((
```

instead of

```
main()
```

or misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them. The compiler will tell you the number of the line where it detected the error; this may not be the line on which the error occurs, it is often the following line! Usually only the first reported error is significant; later error messages may be spuriously generated by the first genuine error. If errors occur, you must correct them using the editor, and then call the compiler again, and repeat this process until no errors are reported.

The compiler's error messages contain the word Error. There may be other messages from the compiler containing the word Warning. These represent constructions in your program which the compiler thinks suspicious. You do not have to correct them, but you should make sure that they are not significant.

There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations.

When the compiler has successfully digested your program, the compiled version, or executable , is left in a file called a.out . After a successful compilation, execute the command

```
ls -l
```

to see that a file a.out exists and has execute permission. Observe its size, and compare it with that of the original program source.

1.5.3. Running the program

The next stage is to actually run your executable program. To run an

executable in UNIX, you simply type the name of the file containing it,
in this case

```
a.out
```

or perhaps

This executes your program, printing any results to the screen. At this
stage there may be run-time errors, such as division by zero, or it may
become evident that the program has produced incorrect output. If so,
you must return to edit your program source, and recompile it, and run
it again. For any serious program, the testing process must be
thorough, and will involve careful planning of the number of tests
required, and the test data chosen to exercise each part of the
program.

General points

Once you have an executable program in the file a.out , you can use
a.out as just another UNIX command.

Any input requested by the program must be typed at the keyboard; there
is no built-in prompt as in some other languages; the program just
halts until you have typed the required input, terminated by end-of-line.

To take your input data from a file called input_file , use the symbol
"<" in the shell and type for example

```
a.out < input_file
```

Any program output normally comes to the screen; to send output to a
file output_file use the symbol ">" in the shell, and type for example

```
a.out > output_file
```

This overwrites any information which was previously in the file
output_file with the new output. To append the new output to any
information already in the file without overwriting, use the ">>"
operation as in

```
a.out >> output_file
```

To pipe output into another program use for example

```
a.out | wc
```

to see how many lines, words and characters are in the program output.

It may be more convenient to use a "-o" and filename in the compilation
as in

```
cc -o program program.c
```

which puts the compiled program into the file program (or any file you
name following the "-o" argument) instead of putting it in the file
a.out .

You can then run it using the command

```
program
```

Alternatively you could have used the earlier compilation command, and
then executed

```
mv a.out program
```

to rename the a.out file.

On some machines, instead of

```
cc prog.c
```

you can use

```
make prog
```

to compile the program in prog.c. This leaves the executable program in
a file prog instead of in a.out. You then type

```
prog
```

to run it. For full use of the make command see elsewhere.

You can now keep several different compiled programs. Beware though,
they all occupy disc space, and you have an upper limit on your total
disc occupation. It is best to delete executables that you do not
require; they can always be quickly regenerated by compilation if you
need them.

C compilers

A compiler is itself just a program, albeit a large and complex one.
Bear in mind that there may be minor differences between our compiler
and ones on, for example, Computing Centre machines, or on your own PC.

1.6. The Ceilidh- system, and the special commands

==
Note: - "Ceilidh" stands for Computer Environment for

Integrated
Learning in Diverse Habitats.
==

Programming is a practical subject, and can be learnt only by practical
experience. It is no use simply reading and listening how to do it, you
MUST get your hands dirty and actually do it. The Ceilidh system was
developed to make the practical work of programming courses as effective
and productive as possible.

There will be at least one programming exercise every week, which you
may complete at any time you like within the set time limit. Use the
"course summary" csum to keep yourself informed of exercises and
submission dates. All of these exercises will be assessed, and the
results form the basis of your end-of-course mark.

You will be expected to use the Ceilidh system for reading the
coursework definition each week, and for marking your results. The
intervening operations of editing, compilation and test runs can be
performed either inside or outside the Ceilidh system.

The simplest option is, at least early in the course, to perform all
your programming work inside Ceilidh. In Ceilidh, each course is
divided into a number of units. Each week you will be asked to complete
certain exercises in certain units. Inside the Ceilidh system, you
perform all operations by choosing items from menus, and will typically
work in the following sequence.

```
     Set the required course ("c") and unit ("1" for now).
     Select the required exercise ("hi" for now).
     View the coursework question ("vq").
     Setup your outline program ("set").
     Edit the program ("ed").

     Compile it ("cm").
     Run it ("run").
     Repeat the previous three steps until you are satisfied.
     Ask the system to mark and submit it ("sub").
     Check that it has been submitted ("cks").
     Quit ("q").
```

To work outside the Ceilidh system, first use Ceilidh to read the

coursework definition ("vq") and to set up your skeleton program
("set"). Then leave the system ("q"), and use vi or emacs to edit, a
command such as cc -o prog11 prog11.c to compile, and prog11 to run your
program, directly in your UNIX shell as required. Then return to
Ceilidh to issue your marking ("mk") command to mark and submit the
work. At some stage of the course, you may be asked to show your
working program to a member of staff.

You must ALWAYS at some stage use the mark and submission command mk in
Ceilidh to show that you have completed the work. If you forget this,
the teacher will have no evidence that you have done the work. Do NOT
hand sheets of paper to anyone, all work is monitored on-line.

1.7. Books

The favourite book for keenies is that by the designers of C, Brian
Kernighan and Dennis Ritchie[2] but make sure you get the second
edition. It assumes that you have programmed in other languages. For
beginners, there are other books which take the subject a little more
gently, such as those by Barclay[1] or Tizzard[5] The two books
published by the Que Corporation[3,4] are expensive, but very thorough,
and full of worked examples.

We will assume that you are familiar with UNIX.

Choose any book with which you feel happy. The notes supplied with this
course should be fairly comprehensive.

1.8. Advice

You may need to contact the course teacher at some point. Eric Foxley
works jointly between the Computer Science and Mathematics departments;
he may therefore be difficult to locate! He may be available in his
Computer Science office (Tower building floor 11 room 1102, internal
phone 4210) part of the time, or in his Mathematics department office
(Maths/Physics building top floor room C107, internal phone 4953). To

find which office he is in at any given time, you will of course use

```
rwho | grep ef
```

to see if ef is logged on, to which machine, and from where. It may be more convenient to use electronic mail to ef or the co (comment) facility in Ceilidh.

Coursework

Details of this week's coursework are available from the "course summary" csum facility in Ceilidh.

1.9. The C preprocessor

This section is for future information.

The C preprocessor is a separate program (usually in a file such as /lib/cpp), which is quite useful in its own right. The compiler passes your program through this preprocessor before compiling it. The facilities of the preprocessor are detailed below.

1.9.1. File inclusion in the source

```
#include "file.h"
    /* include the named source file here */
#include "sub.c"
    /* ".c" for C source */
    /* ".h" for header file */
#include <stdio.h>
    /* include from /usr/include/stdio.h */
```

The form used in the third example <...> searches a standard system area for the file. The other "..." form searches first the current directory, then the include directories.

Convention is that ".h" files (header files) include no code generation; see later for the significance of this.

Include directories can also be specified at the command level (so that different versions can be compiled with different included files).

Examples are

```
cc -o prog -I/usr/lib/include prog.c
cc -o prog.exe -I/usr/vms/include prog.c
```

1.9.2. Macro definitions

Both fixed and parameterised macros are available.

```
/* a fixed definition */
#define VAT    0.15
#define MAX    100
/* every future occurrence of "VAT"
        is substituted by "0.15" */

tax = price * VAT;
/* compiler sees "tax = price * 0.15;" */

if( n > MAX ) ...
/* compiler sees "n > 100" */
```

Warning

Beware of

```
#define COST base + part
/* COST will be replaced by "base + part" */

price = COST * number
/* compiler sees "base + part * number" */
```

Use

```
#define COST (base + part)
/* to avoid operator problems */
```

1.9.3. Parameterised macros

```
/* a parameterised definition */
#define MAX( X, Y )    ( X > Y ? X : Y )

p = MAX( x / y, y / x );
/* the compiler sees
    "p = ( x / y > y / x ? x / y : y / x );" */
```

The same problems arise as mentioned above. A better definition is

```
#define MAX( X, Y ) ( (X)>(Y) ? (X) : (Y) )
```

Note

Note the brackets in the following.

```
#define PRINT printf X ;
```

The statements

```
PRINT( ( "hello" ) )
PRINT( ( "%d", x ) )
```

are now expanded to

```
printf( "hello" )
printf( "%d", x )
```

If we redefine PRINT as empty by

```
#define PRINT
```

the statements expand to nothing, i.e they vanish.

1.9.4. Use of defines for code selection

Somewhere near the start we may have

```
     #define 68040
```

Later on we may now use

```
     #ifdef 68040
      ... /* this code included
              only if 68040 is defined */

     #else
      ... /* this included otherwise */
     #endif

     #ifndef 68040
      ... /* this code included
          only if 68040 NOT defined */
     #endif
```

To cancel a definition

```
     #undef 68040
             /* to unset the definition */
```

1.9.5. Other uses

For two versions of a program (a standard version and a master version
with extra facilities) use

```
     #ifdef MASTER
      ...
     #endif
```

Programs wishing to be portable should set standard defines such as
"SUN3", "UNIX", "VAX" etc., usually one for the processor, one for the
system, e.g.

```
     #ifdef SUN3 && UNIX
```

1.9.6. Defines at the command level

Defines of both the above types are available at the command parameter
level as follows:

```
     cc -DVAT=0.15 prog.c
     cc -D68040 prog.c
```

1.10. Compiler options

These are not important at the moment, but you may find them useful
later.

```
      cc *.c                text in several files
      cc -o progex prog.c   named executable file
                             executable goes into "progex"
      cc -O ...             capital letter O, optimise
                             takes longer to compile
                             but produces more efficient program
      cc -S ...             leave assembler code in .s
      cc -s ...             strip relocation data
      cc -c ...             compile only, to .o
```

```
                              then use the "ld" command
       cc ... -lm            look in library m
```

Examples of complete commands

```
cc -o prog -O -s prog.c -lm
cc -c *.c   # compile several to .o files
ld *.o        # link and load them to a.out
cc prog.c *.o # compile prog.c, link with *.o
```

1.11. Miscellaneous points

Useful UNIX commands related to C programming include

lint prog.c	comment on C source
cc -p prog.c	run-time profiling
xref -c prog.c	cross ref'ce listing
make	system maintenance
SCCS	Source code control system
RCS	SUN Revision Control System
adb	debugging
diff	textual differences
grep	searching for text patterns
cb	C program beautifier
indent	ditto

References

1. Kenneth A Barclay, C Problem Solving and Programming, Prentice-Hall, 1989.

2. Brian W Kernighan and Dennis M Ritchie, The C Programming Language 2nd Ed, Prentice-Hall, 1988.

3. Jack J Purdon, C Programming Guide 3rd Ed, Que Corporation.

4. Sobleman and Krekleberg, Advanced C, Que Corporation.

5. Keith Tizzard, C for Professional Programmers, Ellis Horwood, 1986.

Chapter 2 : Elementary programming

We will now concentrate exclusively on the content of programs.

2.1. A simple program

We return to the elementary program used in the previous chapter, and
discuss its features.

```
/*
   Program written by EF
   October 1991
*/

main() {
    printf( "Hi there!\n" );
/* The "\n" represents a newline character */
} /* end of the main program */
```

Notes:

(i) Any text from "/*" to "*/" is a comment, and is ignored by the compiler. There should be enough comments to make the program file understandable to someone who reads it. In the very short programs that you will write for your first few exercises, comments may not appear so important to you. In large realistic programs, comments are very important, since when a program needs amendment several years after it was written, the original writer may well have moved to another company, or at least will have forgotten the principles of the program. Any text to the left of the first "/*" on each line is part of the program, such as the } on the last line, and is read by the compiler.

(ii) The \n at the end of the printed string represents a newline character. If you omit it, you will find the next prompt from the computer appearing on the same line as the Hi there! instead of being on the next line.

(iii)The main program will (for now) always start with main() followed by an opening curly bracket. The reason for this convention will become obvious later.

(iv) The body of the program (the actual instructions to be executed

```
by
     the computer) is contained between the { and } symbols.  These
sym-
     bols compare with the use of the keywords BEGIN and END  in
Pascal
     and  other  languages;  in  many aspects C uses features from
other
     languages, but is always as terse as possible.

(v)  Every statement which is an instruction to the computer to do
some-
     thing  must  end  with  a  semi-colon.  The symbols { and } and
the
     main() are not instructions to "do something", but are part of
the
     layout  of the program;  they do not have to be followed by a
semi-

     colon.  (This again is different from Pascal and other languages).

(vi) No particular layout  with  newlines,  tab  characters  and
spaces
     (sometimes called "white space") is enforced, but you MUST make
the
     program easily readable.  The only place that extra space must
NOT
     be inserted is inside words such as "main" and "printf", and
within
     the actual text to be printed (which will  be  printed  exactly
as
     given  in  the  program, and must not contain a newline
character).
     The end of line does not terminate a particular instruction;
there
     must be a semi-colon present.

The earlier example program could be written
     main(){printf("Hi there!0);}

DO NOT WRITE LIKE THIS!

2.2.  Output from the program

The identifier printf is used in combination with the parameters
follow-
ing it in printing instructions as follows.
     /* Program written by EF */
     /* October 1991 */

     main() {

     /* to print several lines of text */
       printf(
         "My room is\n1102 Tower\nUninott\n"
       );

     /* and to combine values and text */
       printf( "One seventh is %f\n", 1.0 / 7 );
```

```
     } /* end main */
```

Text between double quotes is printed exactly as given,
arithmetic
values are evaluated and printed as numbers where a "%" appears in
the
format string , see details below.

2.3. Variables

We will wish to have identifiers to represent variables which will
be
used for storing intermediate results during the running of the
program.

For each identifier that we use, we must tell the computer the type
of
object which we wish to use it for.

```
     /* Program written by EF */
     /* October 1991 */

     #define pi     3.14159
     #define seven  7

     main() {

     /* A integer variable, initialised */
     /* Its value may be reassigned */
       int number = seven;

     /* A float variable, initialised */
     /* It can contain non-integral values */
       float reciprocal = 1.0 / number;

     /* An integer variable, not initialised */
     /* to any particular value */
       int square;

     /* Now to print some results */
       printf( "The reciprocal of %d is %f\n",
         number, reciprocal );

       printf(
         "The area of a unit circle is %f\n",
         pi
       );

       square = number * number;
       printf( "Square is %d\n", square );

     } /* end main */
```

The first printf instruction will print the text

```
     The reciprocal of 7 is 0.142857
```

The second prints

```
     The area of ... is 3.14159
```

The third will print

Square is 49.

Each print instruction in the above examples ends with a newline charac-
ter. You will generally want to end with a newline, unless you are
printing a prompt requesting the user to enter a reply.

Identifiers

The identifiers you choose to represent variables or #define items must
satisfy various rules and recommendations.

- They must start with a letter, which can be followed by any number of letters, digits and underscores.

- They should be meaningful. In general, you should not use single characters such as "x", "y", "i" and "j"; these terse identifiers give no feel for the significance of the values they represent.

- Your identifiers must not clash with certain special words. If you use "float", for example, the compiler will become very confused.

- Upper and lower case characters (capital and small letters) in identifiers are distinct.

Examples of identifiers are:

```
fred    total_91    total_92
Fred    Prog_Week_1A
PayPerHr       pay_per_hr
```

It is common practice to use the underscore symbol to separate the com-
ponent parts of an identifier.

Declarations

Declarations are program statements which tell the compiler which iden-
tifiers we intend to use. If an identifier occurs which we have not
declared (either we forgot to declare it, or, more likely, we mistyped
an identifier) the compiler will print an error message. Declarations
such as those in the above program do two things:

- They tell the compiler about an identifier/type pair, so that the compiler can interpret later statements correctly.

o At run time, the running program must set aside the necessary space
to store the specified type of object.

At a later stage, we may need to distinguish between these two effects
by separating a declaration into two parts.

You may combine a declaration with the assignment of an initial value.
If you do not initialise a variable, you cannot rely on it containing
any particular value before you use it.

At this stage, we will insist that you put #define lines before the
main() line, and variable declarations after it, before any executable
program code.

2.4. Basic types

The basic variable types in C are given below, together with their sizes
in bytes on some implementations.

type	PDP	68000	VAXVMS
char	1	1	1
int	2	4	4
long int	4	8	4
short int	2	2	2
float	4	4	4
double	8	8	8

char:This can contain one character, a letter or digit or punctuation
character.

int: This can contain integral (whole-number) values. There is a limit

to the size of the largest positive and negative numbers which can
be stored, which depends on how many bytes are occupied.

long int:This also contains integral values, and may use more space
than, and hence be able to store larger numbers than, an int vari-
able. It may actually be no larger than an ordinary int variable.

short int:This also contains integral values, and may use less space
than, and hence be able only to store smaller numbers than, an int
variable. It may actually be no smaller than an ordinary int vari-

able, but you may need to conserve space if possible.

float:This type of variable can store all numeric values, not just integral values. The accuracy to which they are stored, and the maximum and minimum values, depend of the particular hardware/software being used.

double:This type of variable is intended for storing more accurate numeric values than float variables.

2.5. Denotations

A denotation is a representation of a particular value. Typical denotations for constants of various types are as follows.

int: Typical denotations might be

```
25 -15 +99
0177 /*leading zero denotes octal */
0XFF /*leading zero and X denotes hexadecimal */
```

long int:The integer denotations must be followed by the letter "L" to indicate a long int denotation.

```
1999L 0L
```

char:Single character as values are always written between prime symbols, so that they are not confused with identifiers.

```
'a' 'Z' '+' '.' ' '
'\t' /* tab */
'\n' /* newline */
'\a' /* alert = bleep */
'\177' /* ASCII octal code */
'\0'
'\\' /* backslash */
'\'' /* prime */
```

float:Float values are typed as numbers with a decimal point, and can be optionally followed by the letter "E" (for exponent) and an integer. The value before the "E" is assumed to be multiplied by 10 raised to the power of the integer after the "E".

```
111.222 1.2345E6 -1E6 -1E-6
```

The latter denotations represents the values 1234500, 1000000 and 0.000001 respectively.

2.6. Comments

Comments are from "/*" to the next "*/".

```
/* ... comment ... */

/*
     This is
     a long
     comment
 */

float velocity; /* velocity in kph */
```

For long comments, some people use

```
 /*
  * This is
  * a long
  * comment
  */
```

to look pretty.

2.7. Program layout

Lay out your program carefully, with plenty of white space, indented
as
appropriate.   I don't mind which convention you use, but be
consistent!
As programs become larger, program layout becomes more and more
impor-
tant.

2.8. Input to the program

We use printf for program output, and a similar function scanf for
input
to the program.

Example

```
/* Program written by EF */
/* October 1991 */

main() {
  int number;
  float reciprocal;

  printf( "Type a number: " );

/* "scanf( "%d", ...)" expects a value valid */
/* for the given type, %d for integer, %f for floating. */
    scanf( "%d", &number );

/* If you use "1" instead of "1.0" */

/* it would be an integer result. */
  reciprocal = 1.0 / number;

  printf( "Reciprocal of %d is %f\n",
    number, reciprocal );
```

```
} /* end main */
```

Note that the prompt does not have a newline character, and has a space
before the final closing quote.

The scanf has a format string as its first parameter, and a variable
name preceded by an ampersand (&) sign as its second argument. The
ampersand sign is essential, its omission may cause strange failures
with messages such as

```
Bus error - core dumped
```

which leaves a (possibly huge) file called core in your directory. You
should remove this with the UNIX command

```
rm core
```

Example

This second program reads two values.

```
/* Program written by EF */
/* Calculate the area of a rectangle */

main() {

/* Declare three float variables */
  float length, breadth, area;

  printf( "Type length and breadth: " );

/* Read two values */
  scanf( "%f", &length );
  scanf( "%f", &breadth );

/* Compute area */
  area = length * breadth;

/* Print results */
  printf(
    "Length %f, breadth %f, \
area is %f\n",
    length, breadth, area );

} /* end main */
```

The escaped end-of-line causes the end-of-line to be ignored.

You could avoid the variable "area" all together, and print the results

with

```
printf( "Area is %f\n", length * breadth );
```

but this is not good practice.

It is good practice at this stage of your course to print out all of the values you have read in, so that the output gives a complete picture of what has been calculated.

You may loose a few marks on the dynamic testing if you do not do this. You will certainly loose marks if you do not put explanatory text into your output.

You can now write simple programs to read in some values, evaluate a formula, and print the result.

Formatted output

You can use formatted output to print output more nicely. Above we used "%d" for decimal printing, and "%f" for floating. You can also use:

```
     %o   /* for octal */
     %x   /* for hexadecimal */
     %c   /* for a single character */
     %3d  /* decimal, allow for 3 digits */
     %03d /* ditto, but zero fill on left */
     %-5d /* allow 5 but left justify */
     %10.3f  /* float, width 10, 3 DPs */
     %s   /* string, see later */

     printf(
         "hours %d\nmins %d\nsecs %d\n",
         h, m, s );
     printf( "time %2d:%02d:%02d",
         h, m, s );
```

The last statement would print in the format

```
     time 15:02:05
     time  5:05:59
```

2.9. Operators in more detail

We will now have a thorough tour of all of the available operators and their significance.

2.9.1. Arithmetic operators

The simple arithmetic operators you would expect to see are

```
     + - * /
```

representing addition, subtraction, multiplication and division, respectively. As everywhere in C, types are important here. Between two "int"s the result is an "int", otherwise (between two "float"s, or between an "int" and a "float") the result is a "float". This is fairly

obvious for addition, subtraction and multiplication.

Beware of "/" between integers; the result is an "int", which may not be what you expected. It gives the quotient as an integer rounded down towards zero if the result is positive. If you type

```
     float x = 1 / 7;
     float y = 6 / 7;
```

you may not get the expected result; both x and y will be set to zero!
If the result would be negative, the language does not define exactly what will happen, for example whether "10 / -7" is -1 (which you would get if you rounded towards zero) or -2 (if you rounded down).

The operator "%" between integers gives the remainder when the first is divided by the second. Thus "72 % 10" evaluates to 2, and "30 % 13" evaluates to 4. If either of the operands are negative there are ambi-guities similar to those in division. The value of "10 % -7" might be -3 or +4. The official definition says that the value of the expression

```
     ( a / b ) * b + a % b
```

must always equal the value of a.

There is no operator in C for exponentiation (for raising any number to a given power).

Example programs

In the example programs given from now on, we may give only the text within the "{" and "}" of the main program. To run the program, you would have to add the lines up to main(){ and the final "}". Further, we may not include the niceties of prompts and input/output if we are really demonstrating other types of statement.

(i) Convert Fahrenheit temperature to Celsius

```
          float fahrenheit, celsius;
          printf( "Type Fahrenheit temperature: " );
          scanf( "%f", &fahrenheit );
          celsius = (fahrenheit - 32) * 5 / 9;
          printf( "Celsius is %f\n", celsius );
```

(ii) I have a certain number of bicycle spokes; I need 44 to make one
wheel; how many wheels can I make? How many spokes will I have
left over?

```
#define  spokes_per_wheel 44

int spokes, wheels, left_over;
printf( "How many spokes? " );
scanf( "%d", &spokes );
wheels = spokes / spokes_per_wheel;
left_over = spokes % spokes_per_wheel;
```

(iii)We know the number of football matches won, drawn and lost by a

given team; how many points do they have (3 for a win, 1 for a
draw)?

```
integer won, drawn, lost, points;
scanf( "%d", &won );
scanf( "%d", &drawn );
scanf( "%d", &lost );
points = won * 3 + drawn;
```

2.9.2. Comparison operators

There are many other operators besides those used for arithmetic evalua-
tions. We will need these in the next chapter for use in "if ... then"
constructs.

```
a > b     /* greater than */
a < b     /* less than */
a >= b  /* g.t. or equal to */
a <= b  /* l.t. or equal to */
a == b  /* equals */
a != b  /* not equals */
/* watch for the double "=" sign */
```

These can be between (almost) any types of object. The result delivered
is zero for FALSE, one for TRUE. In appropriate places in C generally,
zero is always interpreted as meaning FALSE, while non-zero is inter-
preted as TRUE.

Note that testing for equality "==" between floats or doubles is not
sensible, because of the possibility of rounding errors. The compiler
will permit you to do it, but it is considered bad practice. You should
instead look for a small absolute difference between the two values.

Examples

(i) Is the Fahrenheit temperature above freezing point?

```
          fahrenheit > 32
```

(ii) Do I have enough spokes for 2 bicycle wheels?

```
          spokes >= 2 * spokes_per_wheel
```

2.9.3. Logical operators

For combining the results of comparisons, we need general logical opera-
tors. In fact, we are not limited to combining the results of comparis-
ons; we can combine any values, and any zero value will be interpreted
as FALSE, and non-zero value as TRUE. We use "&&" for the logical "and"
and "||" for "or".

```
     int number;

     /* set "number" to some value ... */

     number >= 0 && number < 10

     /* "&&" is "and", */
     /* so true if the number is from 0 to 9 inclusive */
     /* false otherwise */

     number < 0 || number >= 10
     /* "||" is "or" (inclusive or) */
     /* one or the other or both */
     /* so true if the number is outside the range 0 to 9 */

     ! ( number >= 0 && number < 10)
     /* "!" is logical "negation" */
     /* this expression has the same value as the previous one */
     /* a monadic operator */
```

Examples

(i) Is the Celsius temperature today within the expected band for this
     time of year, say 5 to 15 degrees?

```
          5 < celsius && celsius < 15
```

(ii) Can I construct at least 2 wheels with less than 10 spokes left
     over?

```
          spokes >= 2 * spokes_per_wheel
            && spokes % spokes_per_wheel < 10
```

2.9.4. Incremental operators

These are unique to C and C++. Their real significance and use will
become apparent later.

```
     ++i  /* increment, deliver new value */
     i++  /* increment, deliver old value */
     --i  /* decrement, deliver new value */
```

```
     i--  /* decrement, deliver old value */
```

The word "increment" means "increment by a suitable value". Compare

```
     i = 0; printf( "%d\n", i++ );
```

which prints the value 0, with

```
     i = 0; printf( "%d\n", ++i );
```

which prints the value 1. In both cases, "i" takes the value 1
after
the instruction.

Note that

```
     j = p + i++;
```

is equivalent to

```
     j = p + i; i = i + 1;
```

where

```
     j = p + ++i;
```

or

```
     j = p + (++i);
```

is equivalent to

```
     i = i + 1; j = p + i;
```

You will often see the free-standing increment

```
     i++; /* increment i, could be ++i */
```

to add 1 to i, used instead of writing

```
     i = i + 1;
```

You will see examples of these operators later.

2.9.5. Assignment

The values being assigned will be cast or coerced (their types will
be
changed and their values converted between types) as required.
Examples
of assignments include the following.

```
     int i, j;
     char c, lc;

     i = ( j + 2 ) / 3; /* integer divide */
     c = 'X';
     j = c - 'A';
         /* j is ordinal of char c */
     lc = c - 'A' + 'a';
         /* lc is lower case char */
         /* for upper case char c */
     i += 3;
         /* "plus-and-becomes" */
     i -= j;
         /* "minus-and-becomes */
     i *= 10;
         /* "times-and-becomes" */
```

Delivered Result

Assignment is an operator, and delivers as its result the value just
assigned. Thus

```
     i = ( c = getchar() ) - 'A';
```

is equivalent to

```
     c = getchar();
     i = c - 'A';
```

To assign the same value to several variables use

```
     i = j = k = 0;
```

Note that in initialising declarations, you MUST still write in full

```
     int i = 0, j = 0, k = 0;
```

Examples

These operations could all be performed as two separate instructions;
express them as two statements if you feel happier that way.

(i) How many bicycle wheels can I make, and how many spokes will I have
used?

```
          spokes_used =
            ( wheels = spokes / spokes_per_wheel )
            * spokes_per_wheel;
```

A warning

Beware of using "=" instead of "==", such as writing accidentally

```
     if ( i = j ) ....
```

with a single equals sign. This copies the value in "j" into "i", and
delivers this value, which will then be interpreted as TRUE if is is
non-zero.

2.9.6. The comma operator

The real significance and use of this operator will appear later. An
expression consisting of statements separated by commas, as in

```
     statement1,
     statement2,
     statement3
```

causes each statement to be executed in turn; the result finally
delivered is that delivered by the last statement. Thus you can write

```
     i = ( c = getchar(), j = i - 'A' ) + k;
```

or

```
     if (
        c = getchar(),
        i = c - '0',
        c != '\n'
     ) { ...
```

The effect of this last statement is exactly the same as if you had typed

```
     c = getchar();
     i = c - '0';
     if (
        c != '\n'
     ) { ...
```

2.9.7. Operator precedence

It is necessary to define carefully the meaning of such expressions as

```
     a + b * c
```

to define the effect as either

```
     (a + b ) * c
```

or

```
     a + ( b * c )
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that

```
     a - b - c
```

is evaluated as

```
     ( a - b ) - c
```

as you would expect.

Exact details are in any book on C. There are many operators here that we have not yet met, but they are all entered here for completeness.

If you are ever in doubt, use extra parentheses to ensure the correct order of evaluation, and (equally important) to ensure the easy readability of the program.

From high priority to low priority the order is

```
     ( )  [  ]  -> .
     !  ~  -  *  & sizeof cast ++ --
         (these are right->left)
     *  /  %
     +  -
     <  <= >= >
     == !=
     &
     ^
```

```
     |
     &&
     ||
     ?:         (right->left)
     =  +=  -= (right->left)
     ,   (comma)
```

Thus

```
     a < 10 && 2 * b < c
```

is interpreted as

```
     ( a < 10 ) && ( ( 2 * b ) < c )
```

and

```
     a =

       b =
         c =
           spokes / spokes_per_wheel
            + spares;
```

as

```
     a =
       ( b =
         ( c =
           ( spokes / spokes_per_wheel )
            + spares
         )
        );
```

Chapter 3 : Conditionals

We now get down to some typical program statements. This chapter is
concerned with constructs which cause the program to take different
paths depending on the values which have been assigned to program vari-
ables during the running of the program.

3.1. If statements

3.1.1. Simple "if" statements

The simplest form of "if" statement causes selected statements to be
executed if a certain condition holds at that point in the program.

```
     int radius, result = 0;

     scanf( "%d", &radius );

     if ( radius > 0 ) {
         result = radius * radius;
         printf( "radius is positive\n" );
     } /* end if radius > 0 */

     printf( "radius %d, result %d\n", radius, result ) );
```

The condition to be tested appears in parentheses after the word "if",
and the statement or statements whose execution is determined by the
condition are contained within curly braces. If the condition does not
hold, everything up to the next "}" is ignored, and program execution
continues after the "}". In this example, if the radius value read in
is positive, the program then executes in order the statements

```
     result = radius * radius;
     printf( "radius is positive\n" );
     printf( "%d %d\n", radius, result );
```

If the value read in is not positive, the statement in the braces fol-
lowing the "if" condition will be ignored, and the program will execute
the single statement

```
     printf( "%d %d\n", radius, result ) );
```

If there is a numeric expression in the parentheses, as in

```
     if ( total ) {
```

then a value of zero is considered as FALSE, non-zero as TRUE.

3.1.2. "If ... else ..." statements

If the condition does not hold, we may wish to execute some

different
statements as alternatives to the "if" statements. We add the
keyword
"else" and the additional statements contained between curly braces.

```
int money;
int deposits = 0;

int transactions = 0;
int withdrawals = 0;

scanf( "%d", &money );

if ( money > 0 ) {
    printf( "Deposit.\n" );
    deposits += money;
    transactions++;
} else { /* if money <= 0 */
    printf( "Withdrawal.\n" );
    withdrawals -= money;
    transactions++;
} /* end if else money > 0 */

printf( "Transaction noted.\n" );
```

If the quantity read is positive, the program then executes

```
printf( "Deposit.\n" );
deposits += money;
printf( "Transaction noted.\n" );
```

If the quantity is zero or negative, the program executes

```
printf( "Withdrawal.\n" );
withdrawals -= money;
printf( "Transaction noted.\n" );
```

The careful layout of programs becomes more important as the
programs
become more structured. Always indent the statements between
curly
braces more than the lines containing the braces. Some people prefer
to
put the braces on separate lines as in

```
if ( radius > 0 )
{
    ....;
    ....;
} /* end of radius > 0 */
else
{ /* if radius <= 0 */
    ....;
    ....;
} /* end if else  */
```

Choose a method of laying out programs which you feel happy with,
and
keep to it. In legal jargon, I would say that the closing curly
brace
must line up with the first visible character on the line containing
the

opening curly brace.

3.1.3. Further alternatives

We can add further alternatives to an "if" statement if we require.

```
     if ( radius > 100 ) {

     /* radius > 100 */
         ....;
     } else if ( radius > 10 ) {
     /* radius > 10 and radius <= 100 */
         ....;
     } else if ( radius > 1 ) {
     /* radius > 1 and radius <= 10 */
         ....;
     } else {
     /* radius <= 1 */
         ....;
     } /* end if radius various values */
```

The tests in the if statements are executed exactly in the order in
which they are encountered. The first test here is "radius > 100"; if
this is false, the second test if executed, testing whether "radius >
10", so that this is effectively the test "radius <= 100 && radius >
10", since we know that the first test is false.

3.1.4. Possible conditions (logical expressions)

We looked at the format of conditions in chapter 2, when we were dis-
cussing expressions in general.

```
     /* Test equality, use double equals   */
     if ( i == 3 ) ....

     /* Two tests ANDed together */
     if ( radius >= 0 && radius <= 10 ) ....

     /* Two tests ORed together */
     if ( i == 0 || i == 1 ) ....
```

It is generally bad practice to compare two float values for equality or
inequality. Because of rounding errors, the results of a floating
expression may not be exactly what you think. The exception to this
rule is to read in a float value using scanf and immediately compare it
with zero, for example.

Lazy operators

The AND and OR operators above are lazy. The word "lazy" has a proper

meaning in the compilation of programs. The "and" and "or" logical
operators ("&&" and "||") are lazy, i.e. are evaluated from left-to-
right, and stop as soon as the result is determined.

```
i >= 0 && i < 10 && funct( i ) == 0
/* evaluated left to right "lazy" */
/* stop as soon as "false" is encountered */

ok( "Overwrite file?" ) && ok( "Sure?" )
/* Second prompt only if first ok  */

x < 0 || x >= 10 || ...
```

The "and" operator stops as soon as a false expression is encountered;
the "or" stops as soon as a true expression is encountered.

In C you can safely write

```
if ( x+y > 0 && sqrt( x + y ) ... )
```

since the sqrt will not evaluate unless the first condition holds.

Remember all the points on conditions from chapter 2, in particular the
interpretation of the value zero as FALSE, and non-zero as TRUE; the
danger of using assignment instead of equality; and the danger of test-
ing equality between "float"s. operators.

3.1.5. The comma operator

You may find the comma operator useful when the test you wish to perform
involves several calculations, as in

```
if ( radius = 1.0 / i, circc = 2 * pi * radius, circ > 5 ) {
    ....;
}
```

To make the program more readable, you may choose to lay this out on
more lines as

```
if (
  radius = 1.0 / i,
  circ = 2 * pi * radius,
  circ > 5
) {
    ....;
}
```

The first two statements could be written before the "if" if you prefer,
so that we could also write

```
radius = 1.0 / i;
circ = 2 * pi * radius;
if ( circ > 5 ) {
    ....;
```

```
     }
```

You may not find the comma operator helpful at first. There are many
philosophical arguments about programs and their structure, and the
argument in favour of the comma here is that all the calculations
involved in the test should be grouped together within the "if" condi-
tion.

3.1.6. Nesting "if" statements

Your "if" statements can be nested to your heart's content if that is
what the program logic requires.

```
     if ( radius > 0 ) {
       if ( result > 0 ) {

     /* radius > 0 and result > 0 here */
         ....;
       } else {
     /* radius > 0 and result <= 0 here */
         ....;
       } /* end if else result > 0 */
     } else { /* not radius > 0 */
       if ( result > 0 ) {
     /* radius <= 0 and result > 0 here */
         ....;
       } else {
     /* radius <= 0 and result <= 0 here */
         ....;
       } /* end if else result > 0 */
     } /* end if else radius > 0 */
```

3.1.7. Ambiguity

We have specified above that you must always use curly braces after an
"if" condition and after the "else". The C official definition states
that the curly braces are essential only if there is more than one
statement to be executed as a result of the condition. Many commercial
users of C insist, as we do, that the curly braces should always be
there.

If you don't use curly braces, the following is ambiguous.

```
     if ( radius > 0 )
         if ( result > 0 )
         xxx;
       else
         yyy;
```

The above could mean either

```
     if ( radius > 0 ) {
       if ( result > 0 ) {
         xxx;
       } else {
         yyy;
       }
     }
```

or

```
     if ( radius > 0 ) {
       if ( result > 0 ) {
         xxx;
       }
     } else {
       yyy;
     }
```

These two have quite different effects. In the case when, for example,
"radius > 0" and not "result > 0", the first will execute "yyy", the

second will have no effect. If we have "radius <= 0" (the first condi-
tion is FALSE) and "result > 0", the first example will have no effect,
while the second would execute "yyy". The two are thus quite different
in their effect.

It is a good principle always to use curly braces, and to put a comment
after any closing curly brace which is not close to its opening partner.
An example might be

```
     if ( radius > 0 ) {
        ...;
        ...;
     } /* end if radius > 0 */
```

Our rule (enforced by the Ceilidh marking system) will be that a closing
curly brace must have a comment if it is more than 10 lines after its
opening curly brace. This ensures that you see a comment on the com-
puter terminal screen if the complete "if" statement is unlikely to fit
onto one screenful of information.

3.2. Switch statements

The "if" statement essentially gives a choice between two alternatives.
We may sometimes need a choice between a larger number of possibilities.
The "switch" construct illustrated below allows for any number of dif-

ferent actions to be taken dependent of the value of an integer calcula-
tion.

3.2.1. Switch example

```
int input_value;

scanf( "%d", &input_value );

switch ( input_value ) {

  case 0 :
/* if "input_value" is 0 */
    do_this();
    break;

  case 3 :
  case 4 :
/* if "input_value" is 3 or 4 */
    do_that();
    break;

  case 7 :
/* if "input_value" is 7 */
    do_the_other();
    break;

  default :
/* if "input_value" is anything else */
    yet_else();

} /* end switch ( input_value ) */
```

The value after the word "case" must be a constant, you could not put

```
case j:
```

where "j" is an "int" variable. You must put either an explicit con-
stant (the numeric value 7), or a #define constant (where you have
declared for example

```
#define ins_per_ft 12
```

).

Two "case" labels can be adjacent. In this case, the two specified
values will cause the same code to be executed.

Don't forget the "break;" statements if you need them. You will nor-
mally want control to leave the "switch" statement at the end of each
separate "case".

The default: entry is optional, but will most often be included.

The value in the "switch" parentheses must be an integer variable or
expression. It cannot be a "float" value.

3.2.2. Character switch example

```
char command_char;

scanf( "%c", &command_char );

switch( command_char ) {
/* Perhaps 'e' for "edit" */
  case 'e' :
    edit();
    break;

/* Perhaps 'l' for list/print */
  case 'l' :
  case 'p' :
    print();
    break;

/* Some other character */
  default :
    printf( "Don't understand \"%c\"\n",
       command_char );

} /* end switch ( command_char ) */
```

Note the printing of the single character, and the quotes round it. The
quotes are written \" within the format string.

Note again the careful indentation, and the comment after the closing
curly brace.

Think also of omitting the break statements on the rare occasions when
you wish to cause code to follow through as in this example.

```
case 'e' :
  edit();
case 'c' :
  compile();
case 'r' :
  run();
  break;
```

In this case,

```
 'r' causes "run"
 'c' causes "compile" then "run"
 'e' causes "edit" then "compile" then "run"
```

3.2.3. Notes

The expression in brackets after "switch" must deliver an integral
value, not a float or double. Integers, characters and long

integers
are permitted.

All the time as we learn new constructs, programs become more complex,
and the neat layout of programs becomes more important. Our automatic
marking system will check your layout.

Chapter 4 : Loops

In the programs we have written so far, the statements in the program have been executed in sequence, from the start of the program to the end, omitting sections of "if" and "switch" constructs which have not been selected. The real power of computers comes from their ability to execute given sets of statements many times.

The repetition may be required for several reasons.

(i) We wish to repeat the calculation once for each one of a number of items of data. We may wish to compute the profit for a number of different months of a company's sales. We may wish to compute the stress in each part of the structure of a bridge. We may wish to look at each word in a file of text. In some cases, we may know in advance exactly how many times the calculation will need to be repeated.

(ii) We may wish to repeat a certain operation until a particular condition is satisfied. For example, we may read numbers from a keyboard, analyse each one and perform some action on it, until a particular value is typed. Another example is that many computer programs keep reading commands and executing them until the user types "quit". In this case, we do not know in advance how many times we will have to go round the loop.

(iii)We may wish to keep attempting a certain operation (such as obtaining data from a remote computer over a network) until either we succeed (all is well, we have obtained the data, so we proceed to use that data), or until a specified number of attempts have failed. (In this case the whole process must be abandoned.)

To repeat sets of instructions there are three main loop constructs in C.

4.1. "while" loops

A "while" loop repeats a given set of instructions until a given condi-
tion holds. The notation is as follows.

```
     int number = 10;

     while ( number >= 0 ) {
       printf( "Value of number is %d\n", number );
       number--;
     } /* end while number >= 0 */

     printf( "Loop ended\n" );
```

The condition to be tested is contained in parentheses (round brackets) after the word "while", and the body of the loop is in curly braces after the condition. There is no semi-colon after the closing curly

brace.

The sequence of operations in the loop (after the initialisation of the value of "number" to 10, for example) is

(i) Test whether "number >= 0".

(ii) If it is FALSE, abandon the loop, and continue with the statements after the closing curly brace. In this case, the next statement to be executed would print out the "Loop ended" message.

(iii)If the result of "number >= 0" is TRUE, execute the statements in the body of the loop (between the curly braces, in this case first print a message, and then decrement the value of "number" by 1), and then return to step (i) above.

The test is TRUE to continue the loop, FALSE to leave it. The test occurs before the loop is executed; the loop may not be executed at all if the test result is FALSE the very first time that it is encountered.
The pattern of execution is thus

```
     test;
or
     test; loop; test;
or
     test; loop; test; loop, test;
```

and so on. The last test on each line must have delivered the result FALSE; earlier tests must have delivered the result TRUE.

When the loop finishes, control passes to the next instruction in

```
the
program, following the closing curly brace of the loop.

Some simple examples of "while" loops

To execute a loop for an integer variable "number" taking the values
1,
2,  ...,  10  you may use any one of the following four possible
"while"
loop constructions.

Version 1
     int number = 1;

     while ( number <= 10 ) {
       ....;
       number++;
     }

Version 2
     int number = 0;

     while ( number < 10 ) {
       number++;
       ....;
     }

Version 3
     int number = 0;

     while ( number++ < 10 ) {
       ....;
     }

Version 4
     int number = 0;

     while ( ++number <= 10 ) {
       ....;
     }

In  the  first  two  examples,  it  is  immaterial  whether  you
write
"number++;" or "++number;";  to increment to value of "number" either
of
these will do.  In the third and fourth examples, you must use  "++"
as
shown.

Some points to observe

1    If what you really want is to execute the loop 10 times, write
the
     condition (as above)
          number < 10

     and not as
          number <= 9
```

The numeric denotations (actual numeric values) appearing in your program should be exactly the numbers you would talk about in describing what the program is required to do. In this case, the value 9 should not appear. If you are converting seconds to minutes and hours, the value 59 should not appear, only the value 60. The Ceilidh automatic marking system checks program features such as this.

2 In general, specific values such as "10" should not appear within the body of your program. They would probably be needed in more than one place, since you may have several loops processing the same number of data items. You should therefore declare them as "const"s at the top of the program as typified by the example

```
#define int repeats 10

main () {
  ....
  while( number < repeats ) {
    ....
  } /* end the loop */

} /* end main program */
```

The Ceilidh marking system checks that values other than, for example "0" and "1" do not appear in the body of the program, and

appear exactly once in a "const" declaration. My examples in the notes may not follow this rule.

3 In C generally you would more likely want to loop not from 1 to 10, but from 0 to 9. All counting in C tends to start at zero rather than one. This is a convention that most C programmers adopt.

More examples of "while" loops

To add together the sequence

1 + 1/2 + 1/4 + 1/8 + ...

until the terms we are adding together are smaller than 0.00001 the program might be as follows.

```
/* We need float variables */
float term = 1.0, total = 0.0;
```

```
     while ( term > 0.00001 ) {

     /* Add the next term to the total */
         total += term;
     /* Halve the term */
         term /= 2.0;

     } /* end while term > 0.00001 loop */

     printf( "Total %d\n", total );
```

The value of "term" is halved each time round the loop; each of
these
values is added to the total.

To read positive integer numbers in, terminated by a zero, and print
the
biggest one.

```
     int next_number = 1, biggest = 0;

     while ( next_number != 0 ) {
       scanf( "%d", &next_number );
       if ( biggest < next_number ) {
         biggest = next_number;
       }
     } /* end while next_number non-zero */

     printf( "Biggest was %d\n", biggest );
```

Note that the terminating zero is still processed by the statements
in
the second part of the loop. In this example, it will have no effect
on
the final result. If we were counting how many positive numbers we
had
read, we would have to be careful not to include the terminating zero.

If we were printing the square of each number read in, we would
probably
not want to print the square of the terminating zero. We would
then
need

```
     int next_number = 1, biggest = 0;

     while ( next_number != 0 ) {
       scanf( "%d", &next_number );
       if ( next_number != 0 ) {
         if ( biggest < next_number ) {
           biggest = next_number;
         } /* end of "if giggest < ..." */
       } /* end of the "if next != 0" */
     } /* end while next_number non-zero */

     printf( "Biggest was %d\n", biggest );
```

Note the following possible alternative coding, which has the
drawback

that the input instruction has to be repeated.

```
int next_number, biggest = 0;

scanf( "%d", &next_number );

while ( next_number != 0 ) {
    if ( biggest < next_number ) {
        biggest = next_number;
    }
    scanf( "%d", &next_number );
} /* end while next_number non-zero */

printf( "Biggest was %d\n", biggest );
```

Within the loop, we read the next number at the end of the loop. We
must read the very first number before we enter the loop. We show a
better solution to this problem later in this chapter.

4.2. "do" loops

The "while" loops above performed the test first, and then executed the
loop. Sometimes you may wish to test at the end of the loop, after the
execution of the statements in the body of the loop (and hence to exe-
cute the loop body always at least once). In C we use what is referred
to as a "do" loop, written as follows.

```
int number = 1;

do {
  ....;
  number++;
} while ( number <= 10 );
```

In this case the value of "number" would be 1 the first time round the
loop, and 10 the last time.

The condition (exactly as in a "while" loop) is still contained in round
brackets, and is still TRUE to continue with another execution of the
loop body, and FALSE to leave the loop. Remember that there is a semi-
colon after the condition, terminating the whole statement. We have one
more semi-colon overall than the equivalent "while" loop.

The pattern of execution in this case can be summarised as follows.

```
loop; test
loop; test; loop; test
loop; test; loop; test; loop; test
```

The code in the above programming example could also be written

```
int number = 1;

do {
  ....;
} while ( ++number <= 10 );
```

This is the form of combined "increment and test" that most C
program-
mers would use. The "++" must, of course, be in front of the
"number"
in this case.

It is generally safer to test at the start of a loop; "while" loops
are
generally safer and more common than "do" loops.

More examples of "do" loops

To read in positive numbers until a zero is encountered, and print
the
biggest one.

```
int next_number, biggest = 0;

do {
    scanf( "%d", &next_number );
    if ( biggest < next_number ) {
        biggest = next_number;
    }
} while ( next_number != 0 );

printf( "Biggest %d\n", biggest );
```

The use of exit

We may wish to abandon the program from within the body of the loop
if
some error condition occurs.

```
int next_number;

do {
  scanf( "%d", &next_number );
  if ( next_number < 0 ) {
    printf( "Error, negative number\n" );
  printf( "Value %d\n", next_number );
    exit( 0 );
  }
  .. process the number ..
  .. which must be >= 0 ..
} while ( next_number > 0 );
```

Use of the comma operator

You can use the comma operator in the condition and write
statements
such as

```
int this_one = 10, that_one = 0;

while ( this_one--, that_one++, this_one > that_one ) {
```

```
        ....;
      } /* while this_one > that_one */
```

At the start of the loop we decrement "this_one", then increment
"that_one", and then test whether "this_one > that_one" before proceed-
ing with the body of the loop.

It would perhaps be clearer to lay this out as

```
      int this_one = 10, that_one = 0;

      while (
        this_one--,
        that_one++,
        this_one > that_one
      ) {
        ....;
      } /* while this_one > that_one */
```

This shows each of the statements and tests on separate lines for clar-
ity.

This type of construction in which the "while" condition involves
several statements, effectively gives us a loop which exits in the mid-
dle. A simple "while" loop tests and exits at the top, a simple "do ...
while" loop tests and exits at the bottom, and a "while" loop with com-
mas tests and exits in the middle of the loop.

Examples of the comma operator

The cleanest way to, for example, read integer values until a zero is
encountered, is to use the comma operator in the following construct.

```
      while (
          scanf( "%d", &next_number ),
          next_number != 0
      ) {
         ....
      } /* end while next value non-zero */
```

With this program structure, the terminating zero is not processed;
this is generally what we require.

Note the layout of the program; we treat parentheses rather like curly
braces. If a matching opening and closing parentheses fit onto one
line, that is fine. If they do not, then they should line up with each
other and be indented in the same way as curly braces. The Ceilidh

automatic marking system checks that program layout conforms to this
pattern.

4.3. "for" loops

There is a third type of loop in C, called a "for" loop. It is written
as follows.

```
int counter, number;

for ( counter = 0; counter < 10; counter++ ) {
  ....;
}

for ( number = 10; number > 0; number-- ) {
  ....;
}

#define float e 0.00001
float term;

for ( term = 1.0; term > e; term *= 0.5 ) {
  ....;
}
```

The general form of a "for" loop is

```
for ( initialise; test; execute after loop ) {
  ....;
}
```

The initialise statement is carried out once only, at the start of the
very first time that the loop is entered. The test is executed before
each execution of the body of the loop. The first time will be immedi-
ately after the initialisation, and hence there will be perhaps no exe-
cutions of the loop body if the test fails at this stage. The third
expression is a statement executed after every execution of the loop
body, before the next test.

The sequence is now

```
init; test;
init; test; loop; incr; test;
init; test; loop; incr; test; loop; incr; test;
```

Again note that the increments in the examples above could be written
with the "++" before or after the variable identifier; in this case it
does not matter.

Readability

One of the important advantages of a "for" loop is its readability. All
of the essential loop control is grouped together at the top of the
loop. We can see at a glance the initial values which are set up, the
test to be satisfied for loop exit, and the main variable increments.
You should make maximum use of this readability.

The "for" loop could be written as a "while" loop in the form

```
     initialise;
      ....

     while ( test ) {
       ....;
       incr;
     }
```

In this layout, the loop control is not so clearly seen.

Defaults

Defaults are obvious; any or all of the three control statements can be
omitted. The construct

```
     for ( ; ; ) {
       ....;
     }
```

gives no initialisation, assumes a TRUE test result, and performs no
incrementing.

The dreaded comma again

You may find the comma operator useful again, particularly in the ini-
tialisation and increment parts of the loop control.

```
     for (
       this = 10, that = 0;
       this > that;
       this--, that++
     ) {
       ....;
     }
```

4.4. General points on loops

4.4.1. Nesting of loops

Loops may, of course, be nested to any depth in any combination as
required.

```
     int month, year;

     for ( year = 1900; year < 2000; year++ ) {
```

```
       for ( month = 0; month < 12; month++ ) {
         ....; /* execute 1200 times ... */
       } /* end month loop for each year */

     } /* end year loop */
```

The loop executes with "month" and "year" taking the pairs of values
[1900,0], [1900,1], [1900,2], ..., [1900,11], [1901,0], [1901,1], ...,
[1901,11], ..., [1999,11] in turn in that order.

4.4.2. The "break" statement

In any of the above loops, the special statement "break" causes the loop
to be abandoned, and execution continues following the closing curly
brace.

```
     while ( i > 0 ) {
       ....;
       if ( j == .... ) {
         break; /* abandon the loop */
       }
       ....;
     } /* end of the loop body */

     printf( "continues here ...\n" );
```

The program continues after the end of the loop.

Within a nested loop, "break" causes the innermost loop to be abandoned.

4.4.3. The "continue" statement

In any of the above loops, the statement "continue" causes the rest of
the current round of the loop to be skipped, and

o a "while" or "do" loop moves directly to the next test at the head
  or foot of the loop, respectively; and

o a "for" loop moves to the increment expression, and then to the
  test.

4.4.4. Example of "break" and "continue"

We wish to write a loop processing integer values which we have read in.
If the value we have read is negative, we wish to print an error message
and abandon the loop. If the value read is great than 100, we wish to
ignore it and continue to the next value in the data. If the value

is
zero, we wish to terminate the loop.

```
     while ( scanf( "%d", &value ) == 1 && value != 0 ) {

       if ( value < 0 ) {
         printf( "Illegal value\n" );
         break;
     /* Abandon the loop */
       }

       if ( value > 100 ) {
         printf( "Invalid value\n" );
         continue;
     /* Skip to start loop again */
       }

     /* Process the value read */
     /* guaranteed between 1 and 100 */
       ....;

       ....;
     } /* end while value != 0 */
```

4.4.5. Comments

It is good practice to comment the end closing curly brace of any
loop
which extends over more than a few lines. The Ceilidh system expects
a
comment after every closing brace which appears more than 10 lines
from
its opening curly brace.

4.4.6. Curly braces

If there is only a single statement in the loop body, the curly
braces
are not obligatory in C. It is however recommended that you always
use
them.

4.4.7. Empty loop bodies

You may find examples of programs in which the complete work of a
loop
is performed within the parentheses of the test; such a loop may
have
an empty body.

```
     while (
       total += term,
       term *= 0.5,
       term > 0.00001
     ) {
       ;
     }
```

This is quite legal; it may not be clear to a reader at first
glance

exactly what is happening.

4.5. Other general loop examples

4.5.1. Sum, minimum, and maximum of data

Read positive "float" values from the input, and print their sum, how
many numbers were read, the largest value and the smallest value.

```
     int count = 0;
     float maximum = 0.0, minimum = 1e6;
     float total = 0.0, number;

     printf( "Type positive values ended by zero or negative: " );
     while ( scanf( "%d", &number ) == 1 && number > 0 ) {

     /* add numbers together */
       total += number;

     /* check minimum so far */
       if ( minimum > number ) {
         minimum = number;
       }

     /* check maximum */
       if ( maximum < number ) {
         maximum = number;
       }

     /* count numbers */
       count++;
     } /* while number > 0 */

     printf( "tot %d, count %d\n", total, count );
     printf( "min %d, max %d\n", minimum, maximum );
```

Be careful if you wish to print the average of the numbers, and use

```
     if ( count > 0 ) {
         printf( "Ave %d\n", total / count );
     } else {
         printf( "No data to average\n" );
     } /* end if count > 0 */
```

You must ensure that you can never attempt a division by zero. Wherever
there is a division sign in a program you must be able to prove that the
denominator cannot be zero.

4.5.2. Sum of squares

To sum the square of all the integer values from 1 squared up to 99
squared.

```
     int next_val, sum = 0;

     /* add squares up to 99 squared */
     for ( next_val = 1; next_val < 100; next_val++ ) {
```

```
    sum += next_val * next_val;
  }

  printf( "Sum %d\n", sum );
```

4.5.3. Decreasing powers of 2

To print (in decimal) the decreasing powers of 2 (1, 1/2, 1/4, 1/8,
...)
you would write:

```
  int count = 1;
  float x = 1.0;

  while ( x > 0.0001 ) {
   printf( "Count %3d, x %10.4f\n", count, x );
    x *= 0.5;
    count++;
  }
```

It may be clearer to write

```
  int count = 1;

  float x;

  for ( x = 1.0; x > 0.0001; x *= 0.5 ) {
   printf( "Count %3d, x %10.4f\n", count, x );
    count++;
  }
```

4.5.4. Read a sentence of text

To read text until a full stop (period) is encountered, and to count
the
number of occurrences of the letter "e", you could write:

```
  char ch;
  int count = 0;

  while (
      scanf( "%c", &ch ) == 1
      && ch != '.'
  ) {
    if ( ch == 'e' ) {
      count++;
    }
  } /* end while ch != '.' */

  printf( "There were %d letter e's\n", count );
```

4.5.5. Increasing and decreasing integers

To read a series of integers, terminated by a zero, and print how
many
times an integer was larger than its predecessor, and how many times
it
was smaller. Ignore the terminating zero.

```
  int next, previous;
  int bigger = 0, smaller = 0;
```

```
     /* Set up the first value */
     scanf( "%d", &previous );

     /* Read data up to a zero */
     while ( scanf( "%d", &next ) == 1 ) {

     /* Was it bigger than the previous? */
       if ( next > previous ) {
         bigger++;
       }
     /* Was it smaller? */
       if ( next < previous ) {
         smaller++;
       }
     /* Store this value for next loop */
       previous = next;
     }

     printf( "Bigger %d\n", bigger );
     printf( "Smaller %d\n", smaller );
```

Chapter 5 : Arrays and structures

So far we have declared variables one at a time. We will often need many variables in a program, and "array"s are a technique for declaring many variables of the same type in one statement, and for being able to consider the whole collection as a single object for appropriate operations.

A "structure" is a means for combining several related items which may be of different types as a single object, while still being able to refer to the separate individual components if we wish to.

5.1. Arrays

5.1.1. Examples of the requirement for arrays

We are analysing the annual rainfall over a period of 90 years; we require 90 float variables to store the information.

We are studying a piece of English text; we require 10000 char variables to store the characters.

We are recording the numbers of students in each department of the university; we need (depending on how many departments there are) 100 int variables.

5.1.2. Declaration of arrays

The above examples would be declared as

```
     /* 90 float variables */
     float annual_rain[ 90 ];

     /* 10000 char variables */
     char text[ 10000 ];

     /* 100 int variables */
     int stud_nos[ 100 ];
```

The number of elements requested in the declaration must be fixed at compile time.

5.1.3. Calls of array elements

To get at a particular variable (element) in an array, we write the identifier of the array followed by the subscript in square brackets.

In C if we declare

```
     float annual_rain[ 90 ];
```

then the subscripts run from 0 (zero) to 89 inclusive; this gives us
the required number (90) of variables.

To access the 23rd of these variables, we would put the subscript in

square brackets after the array identifier, and use

```
      ... annual_rain[ 22 ] ...
```

The subscript can be any integer expression.

To store a value in this variable we may use

```
     annual_rain[ 22 ] = ....;
```

or perhaps

```
     scanf( "%d", &annual_rain[ 22 ] );
```

and to use the value stored there

```
     if ( annual_rain[ 22 ] > minimum ) {
       total = total + annual_rain[ 22 ];
     }
```

The real use of arrays is when we refer to each element of the array not
by a specific constant subscript, but access all elements in turn or
choose a particular element dynamically. To access the variables in
turn, for example to read 90 values from the data into the 90 locations,
we would use

```
     int year;

     for( year = 0; year < 90; year++ ) {
     /* Variable "year" goes from 0 to 89 */
       scanf( "%d", &annual_rain[ year ] );
     } /* for year */
```

The 90 values would have to appear in the data stream in the correct
order, separated by "white space", i.e. spaces, tabs or newlines.

Having read the 90 values in, we may wish to calculate the total and
average rainfall over these 90 years. For this we would use

```
     float total = 0; /* declaration to follow "main" */

     for( year = 0; year < 90; year++ ) {
     /* Each array element is a "float" */
       total += annual_rain[ year ];
     }  /* for year */

     printf( "Total %d\n", total );
     printf( "Average %f\n", total / 90 );
```

In the above examples, the constant value 90 keeps appearing. We should
really take this out as a constant, so that the actual value appears at
only one point in the program. The program now becomes as follows.

```
/* Global constant */
#define duration 90

main() {
  float annual_rain[ duration ];

  int year;
  float total = 0;

  for( year = 0; year < duration; year++ ) {
    scanf( "%d", &annual_rain[ year ] );
  } /* for year */

  for( year = 0; year < duration; year++ ) {
    total += annual_rain[ year ];
  } /* for year */

  printf( "Total %d\n", total );
  printf( "Avege %f\n", total / duration );

} /* end main */
```

We will not now have problems when the number of years duration of the
rainfall analysis needs to be changed; all values of the duration will
change in step together when we change the value of the global constant.
If we had not done this, we might have forgotten to change some of the
occurrences of "90" to another value.

Note that the value of the #define denoting the number of elements in
the array must be a genuine integer constant! When the compiler is dig-
esting your program must know exactly how many array elements are
required. The constant must not be one which is determined as the
result of some calculation while the program is running.

Note also the typical C loop, starting at zero, and limited by "counter
strictly less than number of elements". This gives us the range of
values 0, ..., 89 if there are 90 elements. There is NO element with
subscript 90.

5.1.4. Other array examples

To count the number of occurrences of the letter 'e' in a piece of text,
we will write a program to read the characters of the text into an array
of characters, and then search the array for occurrences of the letter
'e'. We will need an array of char elements. We will read characters
from the input until we encounter a full stop.

```
     /* Set maximum number of characters */
     #define max 100

     /* Grab space for 100 characters */
     char sentence[ max ];

     int i = 0;
     /* Read up to a full stop */
     while(
       scanf( "%c", &sentence[ i ] ) == 1 &&
       sentence[ i ] != '.'
     ) {
       if ( ++i >= max ) {
         fprintf( stderr, "Error sentence overflow\n" );

         exit( 1 );
       }
     } /* while read character not full stop */
```

We should declare the array long enough to hold all likely sentences,
and check that the data does not overflow it.

If we ran the program, and typed

```
     The cat sat on the mat.
```

at the terminal, the code would set

```
     sentence[ 0 ]   to the value   'T'
     sentence[ 1 ]   to the value   'h'
     sentence[ 2 ]   to the value   'e'
     sentence[ 3 ]   to the value   ' '
```

and so on.

To search through the array once it has been read in looking for
occurrences of the letter 'e', we use

```
     int count = 0;

     /* Now go through the array counting */
     for( i = 0; sentence[ i ] != '.'; i++ ) {

       if( sentence[ i ] == 'e' ) {
         count++;
       } /* end if letter is e */

     } /* end search sentence */

     printf( "Number of e's is %d\n", count );
```

We again control the loop by looking for the full stop '.'. It would be
possible instead to count the total number of characters as they are
read in, and use this count to control the upper limit of the loop.

Sorting numbers

We wish to read integers into an array (of int variables), sort them
into ascending order by swapping, and then print them out. We will
assume that the data is terminated by a zero value. First the declara-
tions:

```
     /* the maximum number of ints */
     #define max_n 100

     int array[ max_n ];
```

Then we read the data in:

```
     int i = 0;
     int number;
     while(

       scanf( "%d", &array[ i ] ) == 1 &&
       array[ i ] != 0
     ) {
       if ( ++i >= max_n ) {
         fprintf( stderr, "Error array overflow\n" );
         exit( 1 );
       }
     } /* while read up to zero */

     /* note how many we read in */
     number = i;
```

Then we sort the numbers into ascending order by swapping:

```
     for( i = 0; i < number; i++ ) {

       for( j = 0; j < i; j++ ) {
         if( array[ j ] > array[ i ] ) {
     /* swap if out of order */
           swap = array[ i ];
           array[ i ] = array[ j ];
           array[ j ] = swap;
         } /* if out of order */
       } /* for j up to i-1 */

     } /* for i in the array */
```

Then we might print the results:

```
     for( i = 0; i < number; i++ ) {
       printf( "%d %d\n", i, array[ i ] );
     } /* for i */
```

To print the results ten entries per line

```
/* npl = number per line */
#define npl 10

for( i = 0; i < number; i++ ) {

  printf( "%d ", array[ i ] );
  if ( (i+1) % npl == 0 ) {
    printf( "\n" ); /* newline */
  }

} /* for i */

printf( "\n" );
```

The complete program

The above program when put together becomes as follows.

```
#include < stream.h>

/* the maximum number of ints */
#define max_n 100

/* npl = number printed per line */
#define npl 10

main() {

  int array[ max_n ];
  int i = 0, number;
  int j, swap;

  while(
    scanf( "%d", &array[ i ] ) == 1 &&
    array[ i ] != 0
  ) {
    if ( ++i >= max_n ) {
      fprintf( stderr, "Error array overflow\n" );
      exit( 0 );
    } /* if check array overflow */
  } /* while read up to zero */

  /* note how many numbers we read in */
  number = i;

/* Now order them */
  for( i = 0; i < number; i++ ) {

    for( j = 0; j < i; j++ ) {

      if( array[ j ] > array[ i ] ) {
/* swap if out of order */
     swap = array[ i ];
     array[ i ] = array[ j ];
     array[ j ] = swap;
      } /* if out of order */

    } /* for j up to i-1 */
```

```
  } /* for i in the array */

/* Now print them */
  for( i = 0; i < number; i++ ) {
    printf( "%d %d\n", i, array[ i ] );
    if ( (i+1) % npl == 0 ) {
      printf( "\n" ); /* newline */
    }

  } /* for i */

  printf( "\n" );

} /* end main program */
```

5.1.5. Array elements and indexes

Always be careful to distinguish between the operation

     find the value of the largest element in the array ...

and

     find the position of the largest element ...

To find the maximum value in an array you may write

```
float values[100];
/* Assume that the array is set up */
/* with values terminated by a zero. */

int sub; /* Subscript */
float max = values[0];

for ( sub = 1; values[ sub ] >= 0; sub++ ) {

  if ( max < values[ sub ] ) {
    max = values[ sub ];
  }

} /* for sub in array */
```

We start by assuming that the first element is the largest, and
compare
each of the remaining elements (subscripts from 1 upwards) with it
in
turn.

We now have the largest value in the "float" variable "max".   The
type
of the variable "max" will be the same as the type of the array
ele-
ments.

To find the position (the "index") of the largest element, write

```
float values[100];
/* Assume the array is set up */
/* as before. */
```

```
int sub;
int maxpos = 0;

for ( sub = 1; values[ sub ] >= 0; sub++ ) {

  if ( values[ maxpos ] < values[ sub ] ) {
    maxpos = sub;

  }

} /* for sub in array */

printf( "Largest value %f\n", values[ maxpos ] );
printf( "Position %d\n", maxpo );
```

We assume the position of the largest element is zero, and compare each other element with it in turn. We now have the position of the largest element in the "int" variable "maxpos". The type of the variable "max-pos" will always be int.

Note that there is always a unique maximum value for an element of an array; there may not be a unique position of the largest element if the are several equal largest values.

5.1.6. Searching for words

To search for the occurrences of a particular word such as "the" in the text, we must search for occurrences of the 3 characters 't', 'h' and 'e' in adjacent positions in the array.

```
count = 0;
/* now go through the array counting */

for( i = 0; sentence[ i ] != '.'; i++ ) {
  if ( i < 2 ) {
    continue;
  }
  if(
    sentence[ i-2 ] == 't' &&
    sentence[ i-1 ] == 'h' &&
    sentence[ i   ] == 'e'
  ) {
    count++;
  } /* end if word "the" */
} /* end search sentence */
```

Note that in this case we start the subscript at 2 rather than zero.

We could move the subscript from zero upwards using

```
count = 0;
/* now go through the array counting */
if ( sentence[0] != '.' && sentence[1] != '.' ) {
```

```
        for( i = 0; sentence[ i+2 ] != '.'; i++ ) {
          if(
            sentence[ i   ] == 't' &&
            sentence[ i+1 ] == 'h' &&
            sentence[ i+2 ] == 'e'
          ) {
            count++;
          } /* end if word "the" */
        } /* end search sentence */
      }
```

In either case, we must be careful not to run off either end of the array.

If we were actually looking for the word "the" for serious linguistic reasons, we would need to check that there were non-letters at both ends of the string "the" wherever we find it, using perhaps

```
      if (
        ( i < 3 || sentence[ i-3 ] not a letter )
          &&
        sentence[ i+1 ] not a letter
      ) { ...
```

There is a standard library function is_alpha to check whether a character given as parameter represents a letter. In addition we would probably accept either a leading upper case 'T' or lower case 't'. using a construction such as

```
      if (
        sentence[ i-2 ] == 'T'
          || sentence[ i-2 ] == 't'
      ) { ...
```

Instead of looking for a specific word such as "the", we may wish to look for a general word (string). We would then store the word we are looking for in a second character array. To search our stored sentence for a word of length "l_word" stored in a character array "word" (assuming that the value in l_word has been set up to equal the length of the word stored in the character array word) the second part of the program would have to be turned into a loop to compare characters in the word with characters in the array. The code might be:

```
      count = 0;
      int found;

      /* Now go through the sentence counting */
```

```
for(
  i = l_word-1; /* Start at length of word */
  sentence[ i ] != '.'; /* Stop at full stop */
  i++
) {

  found = 1; /* 1 means true */
  for ( j = 0; j < l_word; j++ ) {
    if(
      sentence[ i-l_word+j+1 ] != word[ j ]
   ) {
      found = 0; /* 0 means false */
    } /* end if next letter found */
  } /* end for all chars in word */

  if( found ) { /* if ( found == 1 ) would do */
    count++;
  } /* end if found */

} /* end search sentence */
```

The above example would be a little more easily readable if we declared

```
#define TRUE 1
#define FALSE 0
```

and use the values TRUE or FALSE later in the program.

String functions

Any operation like this will be much more easily performed by using the
string library functions, perhaps "strncmp" in this case:

```
count = 0;

/* Now go through the array counting */
for( i = 0; sentence[ i ] != '.'; i++ ) {

  if( strncmp( word, sentence + i, l_word ) == 0 ) {
    count++;
  } /* end if found */

} /* end search sentence */
```

For serious programming, always use library functions whenever they are
available.

All of the string library functions assume that the characters stored in
the array are terminated by a null (zero) character. Such arrays could
be printed by "printf", which will print characters from a character
array until a zero element is encountered.

5.1.7. Initialised arrays

If you wish the values of an array to be initialised, this can be

done
ONLY FOR GLOBAL DECLARATIONS before the line containing "main". You can
write

```
     int primes[ ] = { 1, 2, 3, 5, 7, 11 };
```

with the initial values separated by commas, within curly braces. You
need not put a value for the length of the array between the square
brackets, since the compiler can count how many elements you have
declared! This would set up an array of six elements starting at suffix
zero with

```
     primes[0] = 1
     primes[1] = 2
     etc
```

If you do put a value between the square brackets, as in

```
     int primes[ 10 ] = { 1, 2, 3, 5, 7, 11 };
```

an array of 10 elements will be declared, with the remaining elements (4
in this case) not initialised. The number given between the square

brackets must be greater than or equal to the number of initialising
values given.

To initialise a character array, you could write of course

```
     char word[ ] = { 'E', 'r', 'i', 'c' };
```

An alternative notation has been devised because initialised strings are
required so often. You can also write

```
     char word[ ] = "John";
```

This actually initialises a 5-character array, with an additional zero
element at the end; this is provided so that programs using the array
can keep looping until they find the zero element. The form of a loop
using this array of characters (an array with a terminating zero ele-
ment) is now

```
     int sub; /* Our subscript */

     for( sub = 0; word[ sub ] != 0; sub++ ) {
       ....;
     }
```

We could, of course, omit the "!= 0".

Notice that "word" is a straightforward array of character elements. If we execute

```
     word[2] = 'a';
```

the word becomes "Joan".

The size of an initialised array

To operate on the elements of an initialised array you will need to know
how many elements it has. It is bad practice to have a separate global
constant giving the number of elements, declared separately from the
initialised array declaration, since there is always a possibility that
the length value might not agree with the actual length.

One possibility is to use the compile-time operator sizeof (deliver the
size in bytes of an object) which was mentioned earlier, and write

```
     #define arr_length \
       sizeof annual_rain / sizeof annual_rain[0];
```

Alternatively you can put a special marker element at the end of the
array, and loop through the elements until you encounter that special
value. For example

```
     int primes[ ] = { 1, 2, 3, 5, 7, 11, 0 };

     for( i = 0; primes[ i ] != 0; i++ ) {
       ....;
     } /* for i loop */
```

Adding extra prime numbers into the declaration at a later stage will
not affect the correct functioning of the loop, as long as the last

element is always a zero.

This is exactly the way that strings are scanned by library functions;
you may well see the lazy version written as

```
     for( i = 0; sentence[ i ]; i++ ) {
       ....;
     }  /* for i loop */
```

where the " != 0 " is omitted.

Efficiency of array initialisation

To initialise a globally declared array take NO time while the program
is running. The appropriate locations have already been initialied to
the correct values in your executable file.

Arrays and variables declared in global, and not otherwise initialised,
are all initialised to zero.

5.1.8. Two-dimensional arrays

Declaration

The above arrays are one-dimensional, and can store a single "row" or
"column" or "vector" of values. Suppose we wish to store a table (two-
dimensional) of values; these might be the rainfall for each of 12
months for each of 90 years, or the IQs of each of the 11 people in each
of 22 football teams, or the marks for each of 20 exercises for each of
100 students, or the names (30 characters long) of each of 100 students.
We now need two subscripts for each element, and would write

```
/* 90 * 12 variables */
#define n_yrs 90
#define n_mths 12
float month_rain[ n_yrs ][ n_mths ];

int IQ[ 22 ][ 11 ];

int mark[ 100 ][ 20 ];

char names[ 100 ][ 30 ];
```

Use of 2-dimensional arrays

To read rainfall data in, we might use

```
int year, month; /* for subscripts */

/* Read in all the 12 * 90 values */
for( year = 0; year < n_yrs; year++ ) {
  for( month = 0; month < n_mths; month++ ) {
    scanf( "%d", &month_rain[ year ][ month ] );
  } /* Month loop */
} /* Year loop */
```

The numbers in the data must be given in the correct order required by
the program; if the loops are nested as above, we would require the
twelve numbers for the first year, then the twelve numbers for the
second year, ...

If the loops had been nested the other way round (just interchange the
two "for" lines) the data would have had to consist of the 90 January
figures, then the 90 February figures, ...

Accessing the elements

To calculate the annual totals for each of the 90 years we might write

```
/* Calculate the year totals */
float total;

for( year = 0; year < n_yrs; year++ ) {

  total = 0;

  for( month = 0; month < n_mths; month++ ) {
    total += month_rain[ year ][ month ];
  } /* for month */

  annual_rain[ year ] = total;

} /* for year */
```

We could have used "annual_rain[year]" instead of "total" for our
addition above; it would be marginally slower on the computer, since it
would have to look up the array subscript each time.

To print the average rainfall over the 90 years for each of the 12
months separately, we might write

```
/* print monthly averages */
for( month = 0; month < n_mths; month++ ) {

  total = 0;

  for( year = 0; year < n_yrs; year++ ) {
    total += month_rain[ year ] [ month ];
  } /* for year */

  printf( "%d %f\n", month, total / n_yrs );

} /* for month */
```

To find the first student whose name starts with the letter 'S', we
might use

```
int student = -1, i;

for ( i = 0; i < n_students; i++ ) {
  if ( names[ i ][ 0 ] == 'S' ) {
    student = i;

    break;
  }
}
```

leaving student set to -1 if no such student was found.

Initialising 2-dimensional arrays

Arrays can be initialised only when declared in global. You could

write
for example

```
     int table[ ][ ] = {
       { 1, 2, 3, 4, 5 },
       { 5, 4, 3, 2, 1 },
       { 1, 3, 5, 3, 1 }
     };
```

This would give us a 3 by 5 array of integers.

There is an alternative "flattened" form

```
     int table[3][5] = {
       1, 2, 3, 4, 5,
       5, 4, 3, 2, 1,
       1, 3, 5, 3, 1
     };
```

In the flattened form we have to insert the bounds, since the compiler
could not know whether we intended a 3 by 5 array or a 5 by 3 array.

The particular case of initialised two-dimensional character arrays
which will be explained more fully later, but is introduced here since
you may find it useful. We are concerned with initialising an array of
words. When looking at C programs, we may wish to search for all key-
words. We would declare

```
     char *keywords[] = {
       "int",
       "float",
       "double",
       "char"
       ""
     };
```

This is effectively a 2-dimensional array of characters. Each row of
this array is of a different length, the length of that keyword + one
for the terminating zero which is always added to a string.

We could now have code to look for each keyword in turn, the element
"keyword[i][j]" is the j-th character of the i-th keyword. We search
along each keyword until we encounter the terminating zero.

This subject is dealt with more fully later under the subject of
pointers.

5.1.9. Higher dimensional arrays

You can, of course, declare and use 3-dimensional, 4-dimensional and higher dimensioned arrays by extending the above notation.

In three dimensions

```
     float mass[10][10][10];

     int x_co_ord, y_co_ord, z_co_ord;

     for ( x_co_ord = 0; x_co_ord < 10; x_co_ord++ ) {
       for ( y_co_ord = 0; y_co_ord < 10; y_co_ord++ ) {
         for ( z_co_ord = 0; z_co_ord < 10; z_co_ord++ ) {
           if ( mass[x_co_ord][y_co_ord][z_co_ord] >= min ) ...
         } /* z loop */
       } /* y loop */
     } /* x loop */
```

Beware that it is very easy to eat up huge amounts of storage (the above example declares 1000 variables) with many-dimensioned arrays.

5.2. More array examples

5.2.1. Reflect lines

We read text from standard input until end-of-file is encountered, printing each line as it is read in reflected from left to right. We use the function "getchar()" to read characters which reads spaces and newline characters as such. The "getchar()" function returns a negative value when it reaches end-of-file, so that the loop is controlled by a "while ... > 0" mechanism.

```
     #include <iostream.h>
     #include <stdio.h>

     main() {
     /* Store each line as it is read */
       char line[100];
       char ch;
       int i = 0;

       while ( ch = getchar(), ch > 0 ) {

         if ( ch == '\n' ) {
     /* End of line, print in reverse */
           while( --i >= 0 ) {
          printf( "%c", line[ i ] );
           }
           printf( "\n" );
           i = 0;
         } else {
     /* Ordinary character, store it */
           line[ i++ ] = ch;

         } /* end if else end of line */
```

```
       } /* end while not end of file */

     } /* end main */
```

Observe that we have a single loop to read the characters, which takes
a
special action when it encounters a newline character.

Observe that, if the compiled program is called "reflect", then
typing
the command

```
     cat any_file | reflect | reflect
```

or

```
     reflect < any_file | reflect
```

should show the original file.

5.3. Structures

The purpose of structures is to group together a number of
related
items. The items do NOT need to be of the same type.

5.3.1. Examples of the requirement for structures

(i) We are dealing with payroll in a company. For each person
employed
     we need their
          name (string)
          address (string)
          works number (integer)
          tax code (integer?)
          rate of pay / hour (integer)
          hours worked this week (float?)
          total pay so far this year (integer)

(ii) We are analysing the properties of a proposed bridge design.
For
     each component of the structure we need its
          strength (float)
          dimensions (3 floats?)
          weight (float?)
          cost (integer)
          name (string)

(iii)We are working with geographical data. Each item of data
consists
     of
          a data value of type integer
          two position co-ordinates of type float

(iv) We are working with train timetables. For each entry in the
timet-
     able we need
          departure time (int)
          arrival time (int)

special features (Fridays only?, Buffet car?)

5.3.2. Declarations of structure types

We will declare first the layout (components) required of a particular
type of structure; the declaration of the structure objects themselves
will come later elsewhere.

A bridge component structure type might need

```
     struct element {
       float strength;
       float length, breadth, height;
       int cost;
       float weight;
       char name[50];
       int stock;
     };
```

Note that this merely defines a structure type. It does not define an
object of any sort.

A general purpose structure for handling dates might be

```
     struct date {
       int day_no, week_no, month_no;
       char name[4];
       long int secs;
     };
```

A structure for integer data values each of which is associated with an
(x,y) co-ordinate (the x and y could represent geographical latitude and
longitude or Ordnance survey co-ordinates) might be

```
     /* The structure type */
     struct data_item {
         float x, y;
         int value;
     };
```

A structure for railway timetable entries might be

```
     struct t_table {
       int depart;
       int arrive;
       int Fri_only;
       int buffet;
     };
```

The separate components of the structure are called its fields.

5.3.3. Declaration of structure objects

The above declarations are of structure types, not of structure vari-
ables for storing data values. To actually declare objects of the above

structures, we might write

```
     struct data_item this, that, result[200];
```

Here we have declared two single structures called this and that and an
array of 200 structures; the whole array is called result .

Do not confuse the structure type declaration (uses no space, gives an
identifier to the type of structure) and the variable declarations (they
occupy space in the program's memory when the program is running). Typ-
ically the structure type declarations would go into a header file.

5.3.4. Accessing elements of a structure

In order to access individual fields of a structure we use the structure
variable identifier, followed by a full stop, then the field identifier
from the structure type declaration.

```
     /* "girder" is a "element" structure */
     struct element girder;
     /* "all_stock" is an array of 100 structures *
     struct element all_stock[ 100 ];

     girder.name    = "Box girder type 10A";
     girder.strength = 25.93;
     girder.stock   = 0;
     if ( all_stock[ 10 ].stock < minimum ) {
         all_stock[ 10 ].required = minimum;
     }

     tot_requd = 0;
     for ( stock = 0; stock < 100; stock++ ) {
         tot_requd += all_stock[ stock ].required;
     } /* for stock loop */
```

The compiler will not be confused by the fact that stock is used both as
a structure field identifier, and as an array identifier. The field
identifier will always follow a full stop.

```
     /* "nott_london" is an array of "t_table" */
     /* for the Nottm to London timetable */

     for( i = 0; i < NTTS; i++ ) {
       if ( nott_london[ i ].depart < .... ) {
         ....;

     /* "result" is an array of "data_item"s */
     /* add data values for all points within */
     /* given "radius" circle */

     int sum = 0;
```

```
     for ( i = 0; ...... ) {
       if (
         result[ i ].x * result[ i ].x
           + result[ i ].y * result[ i ].y
           < radius * radius

       ) {
         sum += result[ i ].value;
       }
     } /* for i loop */
```

To copy a whole structure use:

```
     all_structs[0] = girder;
```

or if you require merely to copy certain fields use:

```
     all_stock[0].strength  = girder.strength;
     all_stock[0].length  = girder.length;
     all_stock[0].breadth  = girder.breadth;
     all_stock[0].height  = girder.height;
```

5.3.5. Initialising structures

To initialise structures use a similar format to that for arrays with
values between curly braces, as in

```
     struct element girder = {
       1234.56,
       15.7, 32,6, 99,9,
       7600,
       50.05,
       "Box girder"
     };
```

For an array of structures, use either the nested braces as in

```
     struct data_item value[] = {
       { 1.0, 2.0, 1323 },
       { 1.5, 1.0, 4523 },
       { 0.0, 2.9, 4373 },
     };
```

or use the flattened form as in

```
     data_item value[] = {
       1.0, 2.0, 1323,
       1.5, 1.0, 4523,
       0.0, 2.9, 4373,
     };
```

The length of (the number of elements in) an initialised array of struc-
tures can be calculated (a compile-time constant) by using the sizeof
function to divide the total size of the array by the size of one ele-
ment, as in

```
     #define nvalues \
       (sizeof value / sizeof value[0]);
```

Do not forget always to divide by the size of a single element of

the
array.

For a simplified BritRail timetable, use perhaps

```
     struct t_table {
       int depart, arrive ;
     };
```

for the structure type, and

```
     struct t_table nott_lond[] = {
       {  537,  727 },
       {  738,  909 },
       {  837, 1023 },
       { 1037, 1216 },
       { 1237, 1412 },
       { 1437, 1623 },
       { 1637, 1819 },
       { 2037, 2229 },
       { 2214,  512 }
     };
```

for the initialised declaration. You will also need

```
     #define num_trains \
       ( sizeof nott_lond / sizeof nott_lond[0] );
```

In this case the structure type "t_table" represents a single
timetable
entry, whereas the array of structures "t_table" represents a
whole
timetable.

5.3.6. Bit-fields in structures (keenies only)

Ordinary mortals do not need to know that structure notation can be
used
to break a single computer word down into small bit-fields for
storing
small data items.

```
     struct line_type {
       int i;
       float f;
       unsigned line :  5;
       unsigned col  :  6;
       unsigned mode :  2;
       unsigned t   :  2;
     };
```

The integers against each field represent the number of bits
allocated.
All the fields must be of unsigned type.

5.3.7. The use of arrays of structures

Whenever you have related items, they should be grouped in a
structure.
If a program contains two arrays of the same length, it can usually
be
inferred that the corresponding elements are related.

You should replace

```
     float x[100], y[100];
     int value[100];

     for ( i = 0; i < 100; i++ )
       printf( "%f %f\n", x[i], y[i] );
```

by

```
     struct data {
       float x, y;
       int value;
     };

     struct data point[100];
```

and use by

```
     for ( i = 0; i < 100; i++ )
       printf( "%f %f\n", point[i].x, point[i].y );
```

Program style checkers look for arrays of the same length, and recommend
that they be turned into a structure.

5.4. More suggestions for #defines

You could try

```
     #define MAX 100
     #define ALLI i = 0; i < MAX; i++
     #define EVER ;;

     for( ALLI ) {
         a[ i ] = ...
     }

     for( EVER ) {
         ...
         if ( ... ) break;
         ...
     }
```

These can make code more readable.

5.5. The use of Typedef

Typedef is like a "#define" for data types. To define a type `addr'
which is equivalent to `int' on this machine use

```
     typedef int addr;
```

and then declare

```
     addr fred, jim;
```

On another machine, if could be typedef'd to another type.

```
     struct disk = { ... };

     typedef struct disk DISK;
```

```
     DISK a, b[10];
```

Chapter 6 : Pointers

6.1. Simple pointers

```
     int i, *pointer;
        /* "pointer" is a variable
         for storing pointer to int */
     pointer = &i;
       /* "pointer" assigned
         the address of i */
     *pointer = 2;
       /* "int" pointed at by "pointer"
             is assigned the value 2 */
```

The (monadic) operator "&" is "address of", while "*" is "object pointed
at by". The variable name is "pointer" (no star).

Pointers are typed by the object they point at; you can't assign the
address of a long int into a pointer to an int.

Note well

When you declare a pointer, it is not initialised to point at anything,
unless you set it. To declare

```
     int *point;
```

and to then use

```
     *point = ....
```

will generate an error. You must first initialise the pointer with

```
     point = &variable;
```

or initialise it in the declaration with

```
     int total;
     int *point = &total;
```

is OK.

Note also ...

The notation for an initialised pointer declaration

```
     int *point = &total;
```

is perfectly acceptable with the meaning of

```
     int *point; point = &total;
```

but can be confusing to read. In the abbreviated version, we are
declaring "*point" but assigning to "point". Although it is written
"*point = ..." it actually means "point = ...".

6.1.1. Pointers and arrays

```
     int array[ 10 ], *arr_pt;
```

```
arr_pt = &array[ 0 ];
arr_pt = array; /* identical,
   since "array" is pointer to "array[0]" */
/* *arr_pt is now equivalent to array[0]. */

arr_pt++;
/* increment arr_pt
   to point at next object */

*arr_pt = 2; /* assigns to array[1] */

*( arr_pt + 3 ) = 2; /* assigns to array[4] */
```

In general " pointer plus-or-minus integer " gives a pointer; and
" pointer minus pointer " gives an integer.

6.1.2. Pointers for loops

To sum the elements of an initialised array, use a terminating zero.

```
int array[] = { 1, 2, 3, 5, 7, 11, 0 };
int *point, sum = 0;

for(
    point = array;
    *point != 0 ;
    point++
) {
/* the "*point" means "*point != 0" */
    sum += *point;
}
printf( "total %d\n", sum );
```

To count characters in an array of characters terminated by a '.' use:

```
char buffer[ 100 ], *pc;
int e_count = 0;

for( pc = buffer; *pc != '.'; pc++ ) {
    if ( *pc == 'e' ) {
     e_count++;
    }
}
```

6.1.3. Priorities

The expression "*p++" is interpreted as "*(p++)", i.e. increment "p",
deliver what it previously pointed at.

Whereas "(*p)++" increments whatever "p" points at. Given

```
char c, *p, a[10];
p = a; /* point p at a[0] */
```

then

```
c = *p++;
```

is equivalent to

```
{ c = *p; p++; }
/* c is a[0], p points at a[1] */
```

but

```
c = (*p)++;
```

is equivalent to

```
{ c = a[0]; a[0]++; }
/* p still points at a[0] */
```

Some pointer examples

We will give two simple example programs, each in two forms, firstly
using subscripts in the arrays, then using pointers.

To read two arrays of floats, and form their scalar product.

```
/* scalar product */

float a[10], b[10];
float scalar_prod = 0.0;

main() {
    int i;

    for ( i = 0; i < 10; i++ ) {
     scanf( "%f", &a[i] );
    }
    for ( i = 0; i < 10; i++ ) {
     scanf( "%f", &b[i] );
    }

    for ( i = 0; i < 10; i++ ) {
     scalar_prod += a[i] * b[i];
    }
    printf( "Scalar prod %5.1f\n",
     scalar_prod
    );
}

/* scalar product */

float a[10], b[10];
float scalar_prod = 0.0;

main() {

    int i;
    float *pa, *pb;

    for ( i = 0, pa = a; i < 10; i++ ) {
     scanf( "%f", pa++ );
    }
    for ( i = 0, pb = b; i < 10; i++ ) {
     scanf( "%f", pb++ );
    }
```

```
        for (
         i = 0, pa = a, pb = b;
         i < 10;
         i++
        ) {
         scalar_prod += *pa++ * *pb++;
        }
        printf( "Scalar prod %5.1f\n",
         scalar_prod
        );
    }
```

To print the first few lines of a Pascal triangle.

```
    /* Print Pascal's triangle */
    #define MAX    100

    int old[ MAX ], new[ MAX ], i, l;
    int n = 5;

    main ()
    {
        old[0] = 1;
        for ( l = 1; l < n; l++ ) {
         new[0] = 1;
         for ( i = 1; i <= l; i++ ) {
             new[i] = old[i-1] + old[i];
         }
         for ( i = 0; i <= l; i++ ) {
             old[i] = new[i];
             printf ( "%4d", new[ i ] );
         }
         printf ( "\n" );
        } /* for l = 0 to n */
    }

    /* Print Pascal's triangle */
    #define MAX    6

    int old[ MAX ], new[ MAX ], l;

    main ()
    {

        int *pold, *pnew;
        *old = 1;
        for ( l = 1; l < MAX - 1; l++ ) {
         *new = 1;
         for (
             pold = old+1, pnew = new+1;
             *pnew++ = *(pold-1) + *pold;
             pold++
         ) {
             ;
         }
         for ( pold = old, pnew = new; *pnew; ) {
             printf ( "%4d", *pnew );
             *pold++ = *pnew++;
         }
```

```
          printf ( "\n" );
         } /* for l = 0 to n */
     }
```

6.1.4. Double pointers (pointers to pointers)

We will assume that we are given an int variable "i" and an array "a" of
9 ints

```
     int i, a[9];
```

We then declare an array "pa" of 3 pointers to ints, either as

```
     int *pa[] = { a, a + 3, a + 6 };
```

or exactly equivalent

```
     int *pa[] = { &a[0], &a[3], &a[6] };
```

We then declare a single pointer to pointer to int, either as

```
     int **ppa; ppa = pa;
```

or exactly equivalent

```
     int **ppa; ppa = &pa[0];
```

Now add two more declarations of general point to int and pointer to
pointer to int.

```
     int *pi = a; /* i.e. pi = a */
     int **ppi = ppa;
```

The above data can be represented pictorially as follows.

Diagram goes here, see printed notes

I have laid them out so that the top row are all "int"s, the next
row "pointers to int" or "int *", the bottom row "pointers to
pointers to int" or "int **".

We can now get the value of "pa[0]" by either "*ppa" or "*ppi".

Using double pointers, we can refer to "a[0]" as "**ppa" or "**ppi"
or "*pa[0]".

If we execute

```
     ppi++;
     pi++;
```

we get the revised picture

Diagram goes here, see printed notes

The value of "*ppi" is now "pa[1]", "**ppi" is now "a[3]", and
"*pi" is now "a[1]".

Note the importance of order of evaluation in

```
     pi = *ppi++;

which is equivalent to
     pi = *ppi; ppi++;

and
     i = *++pi;

which is equivalent to
     pi++; i = *pi;

and
     pi = (*ppi)++;

which is equivalent to
     pi = pa[1]; pa[1]++;
```

Ragged Arrays

Another way of obtaining a structure similar to the above is as follows.

```
     int a0[] = { 1, 4, 2, 6, 0 };
     int a1[] = { 9, 3, 0 };
     int a2[] = { 7, 6, 5, 4, 3, 2, 0 };
     int *t[] = { a0, a1, a2, 0 };
     /* t can be used like a [3][?] array */

     int *p, **q;
     p = t[1];
       /* p points to the start of a1 */
     q = t;
       /* q points to value of a0 */
```

Diagram goes here, see printed notes

```
          *p = ...
            /* *p is a1[0] */

          **q = ...
            /*  *q is t[0] or a0 */
            /* **q is a0[0] */
```

If we execute

```
          p++;
            /* *p is now a1[1] */
           .. = *q++;
            /* delivers t[0] */
            /* but *q is now t[1] */
            /* **q is now a1[0] */
```

Diagram goes here, see printed notes

```
               *(*q++)++;
                 /* **q is now t[2][1] */
```

A typical for loop might now be:

```
               for( q = t; *q; q++ )
```

```
/* scan q through t */
  for( p = *q; *p; p++ )
    /* scan *p over elements of t */
    ...
```

6.2. Strings

Strings are handled specially in C.

The denotation "eric" (including the quote signs)
is
valid anywhere (not just in global), and delivers
a
pointer to a preassigned string of characters

```
{ 'e', 'r', 'i', 'c', 0 }
```

Note the terminating zero. Thus we can have

```
char *name;
name = "eric";
```

or

```
char *name = "eric";
```

Because of the terminating zero convention we
can
write

```
char *p;

for( p = name; *p; p++ )
    { ... *p ... }
```

6.2.1. Strings in C

All strings in C are assumed to be character
arrays
which end with a null, for example in

```
printf( "His name is %s", name );
```

the variable "name" must be a character pointer,
and
characters are printed until a null is
found.
(What's the difference between "printf(name)"
and
"printf("%s", name)"?)

To copy the string pointed at by "q" into the
area
pointed at by "p" (they must both be of type
"char
*"), we use

```
while ( *p++ = *q++ )
    ;
```

To compare two strings (until a null is
encountered)
pointed at by p and q

```
while( *p )
```

```
              if ( *p++ != *q++ )
                return 0;
              /* return 0 is FALSE return */
            if ( *q )
              return 0;
              /* if not at end of string "q" */
            return 1;
              /* return 1 is TRUE exit */
```

6.2.2. String libraries

The library functions for strings include

```
     strcmp(a, b ) /* compare two strings */
     strcpy(a, b ) /* copy b to a */
     strlen(a)     /* deliver the length */
     strcat(a, b ) /* concatenate b onto end of a
*/
     strncmp(a, b, n) /* compare at most n
characters */
     etc
```

See the on-line manual "man string" for
further
details.

6.2.3. The "system" routine (Unix only)

The call

```
     system( "who" );
```

causes the program to be suspended while the
command
"who" is executed (with its standard input and
out-
put connected to the terminal), after which
program
execution continues.   The "system" call
actually
calls a shell to interpret the string.

```
     system( "a.out" ); /* infinitely recursive,
nasty! */

     printf( "The date is " ); system( "date" );

     system( "cc prog.c; a.out < data; rm a.out" );

     system( "rm *.o" );

     strcpy( p, "ed " ); strcat( p, file );
     system( p ); /* edit the named file */

     system( "stty cbreak" );
      ...
     system( "stty -cbreak" );

     system( "cd /tmp; ....; ...." );
```

6.2.4. Arrays of strings

Arrays of strings (of assorted lengths) are often necessary, in looking up command names, people's names, names, names of months, ... They are declared as

```
char *months[] = {
     "January",
     "February",
     "March",
     "April",
     "May",
     "June",
     0 };
```

This sets up an array of pointers to characters, with a null pointer at the end of each string and at the end of the array of pointers. Now add

```
char **q, *p:

q = months; p = *q;
```

Diagram goes here, see printed notes

Boxes at right are "char".
Boxes in middle are "pointer to char" or "char *".
Box at left are "pointer to pointer to char" or "char **".

6.2.5. Using the "char *[]"

We can do the following:

```
/* print the names */
 q = months;
 while( *q )

    printf( "%s\n", *q++ );

/* look for a name */
  while( *q && strcmp( "April", *q++ ) )
    ;
  if ( *q == 0 ) ... /* not found */
  else
    { q--; /* *q is the one found */ ...
}

q = months;
  /* *q points to "January",
    **q is 'J' */
q++;
  /* *q points to "February",
    **q is 'F' */
```

```
     (*q)++;
       /* *q points to "ebruary",
         **q is 'e' */
```

6.2.6. sprintf and sscanf

The same as printf and scanf, but an
extra
first parameter, a "char *", used instead
of
actual i/o.

```
     sprintf( s, "value is %d", rate );
     /* sets in s "value is 123" */
     /* s must point at a large enough space
*/

     sscanf( s, "%d", &i );
     /* looks for a decimal value in s */

     sprintf( s, "cc %s.c", progname );
     system( s );
```

6.3. Parameters to the program.

When a command (calling a program) is
issued
with parameters, as in

```
     a.out this that other
```

the system generates an integer argc
counting
the arguments (four in this case) and an
object
argv set up to represent the command line,
as
in

```
     int argc = 4;
     char *argv[] =
       { "a.out",
         "this",
         "that",
         "other",
         0
       };
```

("argc" means argument count, "argv"
means
argument values.) Both of these data items
can
be picked up from the main program.

6.3.1. Accessing the parameters

Use

```
     main( argc, argv )
         int argc; char *argv[];
     {
         ... etc
```

The parameter "argv" is the assembled array of strings (including the command name, "a.out" in this case), while "argc" is the number of (non-null) entries. ("argc" is not essential, since its value could be found by scanning through "argv"; it is just convenient; you must include it.) The data in the variables is set up by the system.

```
if ( argc > 1 ) {
    printf( "arg 1 is %s\n",
     argv[1]
    );
}
printf( "Program is called %s\n",
    argv[0]
);
```

6.3.2. The echo command

To print out the arguments (which is what the "echo" command does), the program would be

```
main( argc, argv )
  int argc; char *argv[];
{
  int i;
  for ( i = 1; i < argc; i++ )
    printf( "Arg no %d is %s\n",
      i, argv[ i ] );
}
```

or, with a more pointer-oriented method,

```
main( argc, argv )
  int argc; char *argv[];
{
  argv++;
  while ( *argv ) {
    printf( "Next arg is %s\n",
      *argv++ );
  }
}
```

6.3.3. Unix-type flags

To work out the code to pick up the flags in "cc -s -O prog.c", we first look at the "argv" data as set up by the system.

Diagram goes here, see printed notes

The code to look for flags would be:

```
     while( argc > 1 && argv[1][0] == '-' )
     {
       switch( argv[1][1] )
       {
         case 's' : strip = 1; break;
         case 'O' : optimise = 1; break;
         default :
           printf( "Don't recognise ..." );
       }
       argc--; argv++;
     }
```

You could, of course, do all this with pointers instead of subscripts.

After a single "argc--; argv++;" the picture becomes

Diagram goes here, see printed notes

After the loop is finished we have the picture

Diagram goes here, see printed notes

6.3.4. Other flags

The above code handles separate flags such as "cc -s -c prog". If the flags occur several together, as in "ls -lrt" use

```
     if( argc>1 && argv[1][0] == '-' )
     {
       i = 1;
       while( c = argv[1][i++] )
         switch( c ) ... as above
```

6.3.5. Values from arguments

To read numeric values from arguments, use the "atoi" (ASCII to integer) function as follows.

```
     if ( argc > 2 ) {
       rate = atoi( argv[1] );
       hours = atoi( argv[2] );
```

```
     }
```

or

```
     argv++; argc--;
     if ( argc > 0 ) {
         rate = atoi( *argv++ ); argc--;
     }
     if ( argc > 0 ) {
         hours = atoi( *argv++ ); argc--;
     }
```

6.3.6. The environment

A third parameter of the same type as "argv" can be used to pick up your UNIX environment ...

```
     main( argc, argv, envp )
       int argc; char *argv[], *envp[];
```

where "envp" contains pointers to strings such as "USER=ef", "TERM=vt100", and so on, and is null terminated. The library function "getenv("USER")" then delivers a pointer to the string following "USER=" if it can find one starting "USER=".

6.4. Pointers and structures

6.4.1. Basics

Pointers can be used with structures, using all the techniques explained above. There is one significant extension relat-ing to the operator "->".

```
     struct date toady, week[7], *p;
     p = &today;
     p = week;
     (*p).name = "Mon";
     p -> name = "Mon";
     p++;
     p = week + 6;

     p = (struct date *) 0177756; /* cast */
```

What is meant to look like an arrow is formed of a '-' and a '>'. It is preceded by a pointer to a structure, and

followed by a fieldname.

6.4.2. Structures for lists

```
struct cell
    { int data; struct cell *next; };

for( p = head; p != NULL; p = p->next )
    { .... }
```

6.4.3. Planting Structures

```
struct disc {
   int drive, sector, block;
    char error, mode;
    unsigned data;
};

struct disc *d;
d = (struct disc *) 01777756;

if ( d->error < 0 ) ...
d->sector = ...
```

Chapter 7 : Functions

7.1. Motivation: the need for functions.

The basic property of functions is that they enable us to break a program down into a number of smaller units. There are several different reasons why programs should be broken down in this way. Different users attach different importance to these reasons; some of the considerations below may be considered irrelevant by some practitioners.

(i) One approach starts from the problem which the program is intended to solve. Most problems naturally break down into sub-problems. Many methods for problem solving or system analysis rely on this property. A large problem is broken down into sub-problems; each non-trivial sub-problem is broken down into smaller sub-sub-problems ... Each solution of a sub-problem will be represented by an appropriate piece of program called a function. This may be called "top-down" problem analysis.

(ii) Most problems in the real world of industry or commerce are large and complex. Large problems must be implemented by a team, not an individual. The problem therefore needs to be broken down in a way appropriate for team implementation. This may be done by looking at the different objects involved in the problem and their associated operations; this may be called an "object-oriented" approach to programming.

(iii)When you write a program, you will often find that similar or identical code is required at several places in the program. The use of a function allows this code to be written just once, and to be called up wherever it is required.

Re-usability

If a solution has been created for a particular sub-problem (an implementation of the solution involving means for storing the related data items, and the means of accessing them in an appropriate way) may be useful in more than one problem. An example is the sorting of a number of items into a particular order; sorting is a basic operation which is needed at many points in many computer problems.

Units which solve frequently occurring sub-problems can therefore be re-used in appropriate places in a variety of larger problems.

(i) They may be used in order to save effort in re-solving that particular problem, and in re-programming the solution.

(ii) They may be used in order to produce better quality systems, on the assumption that the solution being re-used was written and tested (by someone else) to a high quality standard when it was originally written.

In this C course we teach just the programming techniques needed for producing useful re-usable code; we are not emphasising just WHY you may choose to break down your problem into these particular units. We need what are called "functions" and "structures" in C; on other courses you may learn techniques for using functions as the basis of a particular methodology for the design of programs.

In this section of the course, we teach the techniques for implementing functions. For those who have used other programming languages, the concept of a "function" in C and C++ corresponds to a "procedure" or "function" elsewhere.

7.2. Functions with no parameters

7.2.1. A simple example

First we look at a simple example.

```
/* First the declaration and definition */
/* of two functions */
void dothis () {
    printf( "Hello\n" );
} /* end of dothis */

void dothat () {
    printf( "Goodbye\n" );
} /* end of dothat */
```

```
/* Now the program calls them */
main () {
/* These are the calls */
    dothis ();
    ....;
    dothat ();
} /* end of main */
```

The parentheses following the function identifier in the function
calls
in the main program indicate that "dothis" and "dothat" are
identifiers
representing functions to be executed. The functions may, of course,
be
called as many times as you wish within the program.

7.2.2. Declaration versus definition

For each function the compiler needs two distinct items of information.

(i) A declaration, to define (to tell the compiler) what a call of
the
     function will look like, and the identifier to be used.

(ii) A definition, to define exactly what is to be done (the actions
to
     be executed) whenever the function is called.

This should be compared with the declarations of variables, where
again
there are two distinct aspects to any declaration, one to inform
the
compiler about the type and identifier for compile-time, and one
to

arrange to claim space at run-time.

In the above simple example, declaration and definition are combined.

These two parts (declaration and definition) of a function can
be
separated in C as in the example

```
/* declarations */
void dothis ();
void dothat ();

main () {
/* calls */
    dothis ();
    ....;
    dothat ();
} /* end main */

/* definitions */
void dothis () {
    printf( "Hello\n" );
} /* end of dothis */
```

```
     void dothat () {
         printf( "Goodbye\n" );
     } /* end of dothat */
```

The first declaration void dothis(); is to warn the compiler what
to
expect as calls of the function, for example by introducing the
identif-
ier to be used. The compiler will not now be surprised to see the
calls
of the function when it encounters them in the main program.
Although
the compiler knows that dothis is the identifier of a function, not of
a
variable, it still requires parentheses after the identifier in
function
calls in the program.

The later definition void dothis() { ... } with executable
statements
within curly braces will define between those curly braces the code
to
be executed whenever the function is called. It will involve
executable
C code.

7.2.3. More examples of simple functions

Variables which need to be accessed within both the function(s) and
the
main program must now be declared before both.

Our rates of pay exercise earlier becomes

```
     int rate, hours, pay;

     void calc_pay() {
        if ( hours ... ) {
         pay = rate * hours;
       } else if ( hours <= ... ) {
         pay = ...;
       } else {

         pay = ....;
     } /* end calc_pay function */

     main() {
       printf( "Type rate and hours: " );
       scanf( "%d%d", &rate, &hours );
       calc_pay();
       printf( "Pay is %f\n", pay );
     } /* end main */
```

To compute the area of a circle of given radius, a program might be

```
     #define pi 3.14159

     float radius;
     float area;
```

```
void calc_area() {
    area = pi * radius * radius;
} /* end cal_area */

main() {
  printf( "Type a value: " );
  scanf( "%f", &radius );
  calc_area();
  printf( "Area is %f\n", area );
} /* end program */
```

To calculate the volume of a cylinder using the formulae

```
volume = height * are of end
area of end = pi * radius * radius
```

we would use functions as in the definition and write

```
#define pi 3.14159

float height, radius, area, volume;

void calc_area() {
  area = pi * radius * radius;
} /* end calc_area */

void cal_volume() {
  calc_area();
  volume = height * area;
} /* end calc_volume */

main() {
  printf( "Type height, radius: " );
  scanf( "%f%f", &height, &radius );
  calc_volume();
  printf( "Volume %f\n", volume );
} /* end main */
```

7.2.4. Local variables

The functions or main program can also have variables declared
within
them, at the start of the code. These are called local variables
and
can be referred to only from within that function or program. The
Hal-
berstam loop eaxmple earlier might be written

```
int value;
int result;

void halberstam() {
  int counter = 0;
  while value > 1 ) {
    if ( value % 2 == 0 ) {
/* even */
      value %= 2;
    } else {
      value = 3 * value + 1;
    }
    counter++;
```

```
    } /* end while loop */
  } /* end function */

  main() {
    printf( "Type the number: " );
    scanf( "%d", &number );
    halberstam();
    printf( "Result is %d\n", result );
  } /* end main */
```

The variable "counter" can be referred to only from within the halber-
stam function.

7.2.5. Local identifiers

If a local identifier is the same as a global one,

o references to that identifier from within the function in which it is declared refer to the local variable;

o references to that identifier from elsewhere refer to the gloablly declared variable.

Generally, and identifier is first searched for as a local variable; if no local declaration of that identifier is found, a global declaration is used.

Function identifiers themselves count a global identifiers. They cannot also be used as global variable identifiers. If they are used as a local variable identifier in a second function, then that function can-
not be called from within that second function.

7.2.6. Warning

The use of global variables as shown above is generally not a good way to develope functions. See later!

7.3. Functions with parameters

The above examples were very simplistic. In more significant examples, we will wish to pass certain values into a function when it is invoked, and may wish to obtain results directly from the call. For example, if we are performing mathematics and require a "square root" function, we will wish

o   to pass to it the value of the number whose square root require,
so
     that a call looks like
          sqrt(2.71828);

     or
          sqrt(2 * x);

o   and instead of putting the result into a global variable, we
would
     like to write
          y = sqrt(2.71828);

     or
          if (sqrt(2 * x) > y) ...

     and use the result directly.

7.4. The function declaration

The function declaration now needs to inform the compiler of
     1: the type of result to be returned;
     2: the identifier of the function;
     3: the number and types of any parameters.

1   Some functions will return a result, some will not.   Functions
to
     compute the square root of a number will return a numeric
result
     for use in the program requesting the square root.   The
examples
     shown earlier, or a function to print output will not return
a
     numeric value.

2   The identifier of a function follows exactly the same rules
as
     identifiers for variables.   It must not clash with the
special
     words of the language, or with global variable identifiers.

3   We will probably need to pass values into functions when we
call
     them.   With the "square root" example above, each time we call
it
     we must give the value of the number whose square root we
require.
     The number which we pass to it is sometimes called a parameter
and
     sometimes an argument. In C the number and types of the
parameters
     are specified in the function definition, not its declaration.
In
     C++, they are specified in the declaration.

Examples of function declarations

A function to compute the square root of a given value.

```
/* identifier "sqrt" */
/* takes one double parameter */
/* gives double result */
double sqrt ( );
```

A function to compute the larger of two integer values.

```
/* identifier "max" */
/* takes two integer parameters, */
/* gives integer result */
int max ( );
```

A function to compute the volume of a cylinder.

```
/* identifier "cyl_vol" */
/* takes two float parameters */
/* delivers float result */
float cyl_vol();
```

A function to print an error message involving an integer value.

```
/* identifier "error" */
/* takes integer parameter */
/* gives no returned result */
void error ( );
```

Examples of function calls

The above functions would be called from elsewhere (perhaps from the main program, perhaps from another function) by statements such as the following.

Square root would take a "double" parameter and return a double result.

```
double y = sqrt ( 2.0 );

if ( sqrt ( x * 5 ) > 2 ) ....

printf( "sqrt 2 is %f\n", sqrt( 2.0 ) );
```

We would normally expect to use the returned result as a value.

Maximum of two integers

```
int this, that;
scanf( "%d%d", &this, &that );

biggest = max( this, that );

printf( "Larger value is %d\n",  max ( this, that ) );

non_negative = max( this, 0 );
```

Volume of a cylinder

```
mass = cyl_vol( hgt, rad ) * density;

if ( cyl_vol( height, radius ) > 100 ) {
```

Reporting an error

```
if ( n < 0 ) {
  error ( 5 ); /* no result */
```

```
     }
     if ( n > 100 ) {
       error ( 6 ); /* no result */
     }
```

The syntax of a function declaration

A function declaration consists of

```
     <returned type> <identifier> ( );
```

Do not forget the semi-colon!

The returned type may be any ordinary C type such as int , float and so on, or the type void if no value is being returned to the main program.

In some older programming languages, the word "function" was used only when a result was to be delivered; the word "procedure" or "function" was used is there was no result to deliver. In C and C++ all such objects are called functions.

7.4.1. The function definition

The function definition needs to inform the compiler of the actions to be taken whenever the function is called. In the declaration we described only the type of the result. The compiler needs parameter type information to enable it to handle the parameters correctly wher-ever the function is called. In order to describe the actions to be taken when the function is called, we need to identify each of the parameters, so that we can refer to them and use them from within the code which forms the body of the function.

The "max" function

We want a function to deliver the larger of the values of the two values given as parameters.

```
     /* deliver the larger of the 2 params */
     int max ( i, j ) int i, j; {
         if ( i > j ) {
             return i;
         }
         return j;
     } /* end max */
```

The "return" keyword acts both to specify the particular value to be returned by the call of the function, and to cause dynamic exit from

the
function.

If the function returns type void we leave the function using the
state-
ment

```
     return;
```

with no value specified; the function will leave anyway at the end
of
its code. If the function returns a non-void type, it must always
use
return ... to exit, it cannot just leave at the end of the code,
and
the return must be followed by a value which the compiler can force
into
the required type.

Notice that we do not need an "else" statement in this example
because
of the dynamic exit caused by the return i; statement.

The cylinder volume function

We will declare and define two functions for this.

```
     /* Two declarations */
     float circ_area();
     float cyl_vol();

     float circ_area( radius ) float radius; {
       return pi * radius * radius;
     } /* end circ_area definition */

     float cyl_vol( height, radius ) float height, radius; {
       return height * circ_area( radius );
     } /* end cyl _vol definition */
```

The Halberstam function

The declaration would be

```
     int halberstam();
```

and the definition

```
     int halberstam( number ) int number; {
       int counter = 0;
       while value > 1 ) {
         if ( value % 2 == 0 ) {
           value %= 2;
         } else {
           value = 3 * value + 1;
         }
         counter++;
       } /* end while loop */
       return counter;
     } /* end function */
```

The "error" function

We want a function to print an error message and then abandon the
pro-
gram.

```
     void error ( errno ) int errno; {
         printf( "Error number %d has occurred\n", errno );
         printf( "Program abandoned\n" );
         exit ( 1 );
     } /* end error */
```

In this example, the "exit" causes the whole program to terminate;
if
we merely wished to report the error and continue the program, we
would
use a

```
     return;
```

statement with no value given. If the function has its delivered
type
declared as void its code does not return values, and the function
is
left by using

```
     return;
```

7.5. The exit function

The exit function causes the completer program to terminate.
Convention
with the exit function is that

```
     exit( 0 ); /* indicates normal exit */
     exit( 1 ); /* indicates error exit */
     exit( 2 ); /* indicates error exit */
```

The UNIX shell can use the value returned by a terminating program
to
determined whether it terminated successfully or not.

However, with the Ceilidh system, all programs must use

```
     exit( 0 );
```

since any other exit assumes that some accidental error has occurred.

7.6. More examples

The sqrt function

We need to calculate the square root of the given parameter value,
and
return it. Please forgive the crude method!

The declaration would be

```
     double sqrt();
     /P}
     The definition could be
          double sqrt( double x ) {
          /* code not guaranteed, I'm in a hurry! */
            double accuracy = 0.00000001;
            double low = 1, high = x, mid = x / 2;
            if ( x < 0 ) {
```

```
        error( 5 ); /* error if negative parameter */

      }
      if ( x < 1 ) {
        low = 0;
        high = 1;
      }
      while( high - low > accuracy ) {
        if ( mid * mid > x ) {
          high = mid;
        } else {
          low = mid;
        } /* end if guess too large */
        mid = ( low + high ) / 2;
      } /* end while accuracy not yet reached */
      return mid;
    } /* end sqrt */
```

The syntax of a function definition

A definition consists of

```
     <returned type> <identifier> (
         <param type> <param ident>,
         <param type> <param ident>,
         ....
     ) { /* No semi-colon after the ")" */
         ....; /* body */
     } /* usually an end comment here */
```

7.7. Returned types

The compiler needs to know the type of the value to be returned by the function so that it can be treated correctly in the call of the function. If no value is to be returned (cf "error" or "dothis" above) then the return type is given as "void".

The particular value to be returned is specified by the line

```
     return <value>;
```

With a void function write simply

```
     return;
```

The compiler will do type conversions where necessary. If the function declaration says that it returns a float and the function includes return 1; then the compiler knows to sort this out.

7.8. Parameter types

The parameter type sequence in the definition and the calls must agree. The compiler will make the necessary type conversions to the delivered result in a call of the function.

The parameters in the function definition are called the formal parameters. The parameter values supplied in calls of the function are called actual parameters.

7.9. Program structure

The main program and function definitions can appear in any order in your program. It is normally clearest to read and understand if the main program appears before the function definitions. Reading the main program gives an overview of the sequence of operations.

Function declarations MUST appear before the function is called. They are usually put together before the main program.

7.10. Local variables

We can declare local variables at the start of our function code if we require additional variables for computations within the function. The declarations appear after the opening curly brace, just as they do in the main program.

If a local identifier clashes with a global constant, or with the name of another function, then that global constant or other function becomes inaccessible within this function. The local object will be referred to by the identifier.

7.11. Parameters called by value

When a function is called, the values of the (actual) parameters at the point where the function is called are calculated, and passed to the function definition for execution. Within the function, the parameters act like variables, initialised to the value of the corresponding actual parameter at the call, and their values can be changed by ordinary assignment.

Thus if a function definition is

```
int fred( int number ) {
  ....;
  number = number + p;
```

```
    ....;
} /* end function fred */
```

the identifier "number" can be considered as a local variable to the function. Changing its value inside the function as shown in the above example has no effect on the outside world.

If the call of the function is

```
int counter = ...;

fred( counter );
```

the value of the variable counter in the calling program will not be changed. We could also call the function by

```
fred( 23 );
```

which would pass the value 23 to the formal parameter. This way of passing parameters is referred to as "passing by value".

7.12. Jargon reminder

The parameters as specified at the start of the function definition are referred to as "formal parameters".

The parameters substituted in any particular call of the function are referred to as "actual parameters".

7.13. A complete example

The complete program with function declaration, main program and function definition will look roughly as follows. We use an example with the Halberstam function.

```
/* Program example */
/* Halberstam function being declared */
/* then used */
/* then defined */

/* Declare the function */
int halberstam( );

/* The main program */
main()
{

    int numb = 0, calcs = 0;

/* Loop reading integers until we meet a zero */
    while (
     printf( "Enter the number now please: " ),
     scanf( "%d, &numb ),
```

```
             numb != 0
            ) {

        /* This is the call */
             calcs = halberstam( numb );
             printf( "numb %d result %d\n", numb, calcs );

            } /* end while read number > 0 */

        } /* end main program */

        /* Now for the definition */
        int halberstam( numb ) int numb; {
            int calcs = 0;

        /* Error exit */
            if ( numb <= 0 ) {
             return -1;
            }

        /* Loop counting how many times */
            while ( numb != 1 ) {
                if ( numb % 2 ) { /* number is odd */
                    numb *= 3;
                    numb++;
                } else { /* number is even */
                    numb /= 2;
                }
                calcs++;
            }       /* End of while loop */
            return calcs;

        } /* end function halberstam */
```

7.14. Examples

In general functions will perform operations on data, and not per-
form their own input output except

(i) if the main purpose of the function is for input/output; or

(ii) to print error messages.

7.14.1. Circle radii and areas

We show here two functions taking float parameters, and giving
float results. The first takes a radius length, and delivers the
area of a circle of that radius. The second takes an area, and
delivers the radius of a circle with that area.

```
        const float pi = 3.14159;

        float area ( float radius ) {
          return pi * radius * radius;
        } /* end radius to area */
```

```
float radius ( float area ) {
  if ( area < 0 ) {
    cerr << "Sqrt invalid param " << area << "\n";
/* you might choose to exit here with "exit( 1 );" */
    return -1;
  }

  return sqrt ( area / pi );
} /* end area to radius */
```

Calls of these functions might be

```
float a1, r1, a2, r2;

scanf( "%f", &r1 );
a1 = area( r1 );

scanf( "%f", &a2 );

r2 = radius( a2 );

if ( r2 < 0 ) { ...
```

7.14.2. Summation

This function sums the series

1 + x + x*x/2 + x*x*x/2.3 + x*x*x*x/2.3.4 + ...

for a given value of "x". We keep adding terms until we reach a term whose value is less than 0.001.

```
float expl( float x ) {
/* sum the exponential series */
  float total = 0;
  float term = 1;
  int count = 0;

/* now loop */
  while ( term > 0.001 ) {
    total += term;
    count++;
    term *= x/count;
  }

/* return the total */
  return total;

} /* end expl */
```

Calls of this function might be

```
printf( "Value is %f\n", expl( 1 ) );
```

7.14.3. The OK function

We require a function to print out a given message as a question, and return TRUE if the user replies "y" and FALSE otherwise. The type of a message (a string of text contained within double

quotes)
is char * (the reasons for this will appear later), and the definition of the function might be:

```
int ok( message ) char *message; {
  char ch;
  printf( "%s [Type y or n]? ", message );
  scanf( "%c", &ch );
  return ch == 'y';
}
```

In a real situation, the function body might repeat the question until the given answer is "y" or "n". In the above example, we respond with TRUE if the answer is "y" and FALSE otherwise.

Note that we do not need to say

```
if ( ch == 'y' ) return 1;
return 0;
```

We can return the comparison result directly.

Calls of this function might be:

```
if ( ok( "Was that correct" ) ) {
  ....;
} else {
  ....;
}

if (
  ok ( "Overwrite the file" )
    && ok ( "Are you sure" )
) {
  ....;
}
```

Observe the subtlety of the last example. The program first asks

Overwrite the file [Type y or n]?

and, because the "&&" operator is lazy, only if the reply to the first question is "y" asks for confirmation

Are you sure [Type y or n]?

If the reply to the first question is "n", the second question is not asked.

7.15. Arrays as parameters

7.16. Program modularity

Chapter 8 File Input and Output

Introduction

There are already ways in which you can use files for input and output
without using any new programming features. For example, you can output
to a file using

```
     program1 > file1
```

to redirect (all of) the program's standard output into the file file1
and you can read those results from the file using

```
     program2 < file1
```

However, this precludes any user interaction with the program, which
might involve displaying questions or a menu, and reading user replies.
Further, it does not allow for multiple files to be used.

There are two levels for using file in C; one uses the basic UNIX system calls, and one the stdio (standard input/output) library. We will
describe the stdio system first, which is the simplest system for
straightforward use.

8.1. The C standard file i/o system

All file input/output works on the open a file, access it, close it
principle.

8.1.1. Opening a file

You need a line

```
     #include "stdio.h"
```

at the top of your program.

This is the simplest general file input/output system. File descriptors
are of type FILE * where FILE is a type which has been #defined in
"stdio.h". Don't forget the "*".

To open a file, use the function fopen as in:

```
     #include "stdio.h"

     FILE *fd;

     fd = fopen( "filename", "w" );
     /* first parameter is file name
      * as string or "char *"
```

```
      * second parameter is
      * "r" = read, "w" = write
      * "a" = append */
```

The call of fopen checks that your have permission to access the file
in

the mode that you have requested. If the fopen fails, it delivers
a
NULL pointer. You should therefore follow the fopen statement by
code
such as

```
     if ( fd == NULL ) {
         ... /* error, open failed */ ...
         exit( 1 );
     }
```

Always check the returned value for NULL before proceeding. The
value
of NULL is #defined in the header file stdio.h.

When opening for reading, the file must exist and you must have
read
access to it.

When opening for writing, if the file exists it is emptied, and you
must
have write access. If it does not exist, it is created; you must
have
write access to the directory in which it is to be created.

```
     fprintf( fd, "Value of i is %d\n", i );
```

This is just like printf, but with an extra file descriptor parameter
at
the start. It returns the value EOF (another value #defined in
stdio.h
) if there was an error (e.g. filesystem full).

You might choose to use the returned value from fprintf, [2~since
the
print is more likely to fail when performed into a file. Use

```
     if ( fprintf( fd, "Value of i is %d\n", i ) == EOF ) {
          .... error ....
     }
```

8.1.2. Reading from a file

To read from an opened file (opened for reading, file descriptor
"fd"),
use code such as:

```
     fscanf( fd, "%d", &i );
```

Note that fscanf returns an integer value as for scanf, or EOF to
indi-
cate an error.

8.1.3. Closing the file

When you have finished with the file, you must close it. Do this using

```
     fclose( fd );
```

Any files you forget to fclose will be closed for you when the program terminates.   Hoever, you cannot have more than a certain number of open files at any time, so it is good practice to close each file when you have finished accessing it.

To read from file, and write amended values to another, use the outline

```
     FILE *input, *output;

     input = fopen( ..., "r" );
     output = fopen( ..., "w" );

     while(
         fscanf( input, .... ) != EOF
     ) {
         ....
         fprintf( output, .... );
     }

     fclose( input );
     fclose( output );
```

8.1.4. Standard input and output

The three standard channels are now stdin for reading keyboard input, stdout for standard output, and stderr for error messages (which will not be redirected in the shell). You do not need to open these three streams. Error messages should be sent to the stderr stream.

```
     if ( ( fd = fopen( file, "r" ) ) == NULL ) {
         fprintf( stderr, "Cannot open file %s\n", file );
         exit( 1 );
     }
```

8.1.5. Appending to a file

It is often useful to append new information to existing data in a file.
If you open a file for writing, its contents are lost completely.

To append to a file, use

```
     fd = fopen( "filename", "a" );
```

This will fail if the named file does not already exists, or you do not have write permission to it. Any data already in the file remains. Any further fprintf's which you perform will be appended to the end of the file.

8.1.6. Pipes in STDIO

In the sdtio library, you can open processes for reading from and writ-
ing to. I find this a very useful extension. For input, we could have

```
     FILE *pipout, *pipin;

     pipin = popen( "who", "r" );
     /* we can now read from output of "who" command
         using fscanf */
```

For output, we could have

```
     pipout = popen( "lpr", "w" );
     /* output using fprintf goes straight to lpr */
```

or even

```
     pipout =
       popen( "sort | pr | lpr -Panadex", "w" );
     /* output goes to sort, then pr etc */
```

We could now copy from input to output as in

```
     char ch;
     while (
         fscanf( pipin, "%c", &ch ) > 0
     ) {
         fprintf( pipout, "%c", ch );
     }
```

At the end, you MUST

```
     pclose( pipin ); pclose( pipout );
```

These pclose calls cannot be ignored like those of fclose, since it is
essential that we send an EOF to the output stream, and that we wait for
that process to terminate.

8.1.7. Moving around inside a file

You can move the read/write pointer around within a file using the fseek
routine.

```
     long posn, offset;

     posn = fseek( fd, offset, 0 );
```

The first parameter is an opened file descriptor. The second is a byte
offset. The returned value is the absolute value of the new position in
the file. The last parameter is

```
     0 = set new position at "offset" bytes
     1 = ... at current + "offset"
     2 = ... at end-of-file + "offset"
```

Thus to move back to the start of the file (perhaps to read it again
without closing and opening it) use:

```
     fseek( fd, 0L, 0 );
```

To skip to the end of the file (perhaps to append additional information, or to find its size) use:

```
     long int size;
     size = fseek( fd, 0L, 2 );
```

If the file is composed of (fixed length) records of a particular type of struct, to skip to the start of the (i-1)-th record use

```
     fseek( fd, i * sizeof ???, 0 );
     fscanf( fd, ... ); /* to read it */
```

8.1.8. Other stdio routines

There are many; I can't remember them all! See the on-line manual man stdio, man fprintf etc for details.

fgetc(fd)
     delivers the next character.

fgets(p, n, fd)
     reads a string into char *p to EOF, newline or n chars, whichever occurs first.

ferror(fd)
     checks whether an error has occurred on that stream.

fread(p, s, n, fd)
     reads s * n characters from stream fd into the char * given by p.

8.2. The use of files in general

Updating files is the essence of commercial programming. A file will contain details of

     all personnel, pay to date, tax to date, tax codes, etc

     all stock in the warehouse, current and minimum levels, etc

     all bank accounts, the owner, the balance, the maximum debt, etc

     all flights by the airline, booked and free seats, destination, timing, etc

In commerce, each set of related data (one person's record, the data for one type of stock item in the warehouse) is referred to as a "record". Each complete set of related data and the means for accessing it from within a program would be represented by a structure inside a C++ program.

Each day or week or month (or instantly on receipt of an interactive transaction) the file will be updated, and a new file produced. For security reasons, a firm will keep a limited number of old copies of the file together with details of all subsequent transactions, so that the latest file can be re-created if it gets corrupted.

The information inside most files will be held in a definite order, e.g. ordered by personnel works numbers, warehouse stock number, bank customer account number, flight departure time, etc.

8.2.1. Updating small files

A typical program to update a file would, if the file is small,

1 read the whole of the latest master file into an array

2 interact with the user (or use information stored in a data file) to update the various entries in the array (to add this week's pay, to decrement or increment the current warehouse stock values, to change the current credit in the bank accounts, to reserve a seat on a flight)

3 write the updated information stored in the array into a new file.

If all has gone smoothly, the new file is now the master copy, the previous master file becomes the backup copy.

The program sequence might be

```
     Declare a structure type suitable for
       the information in each record
     Declare a big enough array of
       these structures
     Open the existing master file
       for reading
     Read all the information from the
       existing file into the array
     Close the file
Then interact with the user using:
     while (
       Ask "Any more updates? ",
       Reply isn't no
     ) {
       Ask "person? ", read person
       Find array subscript for this person
       Ask "details? ", read details
```

```
    Amend entry values in array of structures
  } /* end while more updates */
```

Now finish off with

```
  Open a new file for writing
  Write the complete array to the
    new file name
  Close the new file
  Print any summaries as required
```

8.2.2. Updating large files

For larger files, it may not be possible to read the whole file into memory. The program would first order the transactions so that they are in the same order as the entries in the master file; we would assume that the transactions are now held in a file rather than input from a keyboard. We then read the existing master file one entry at a time, see if that entry needs updating, and write that entry to the new master file. In this case both old and new files (and the file of transactions) are open, and only one record is held in the program at a time.
The program outline might be as follows.

```
  Declare a structure type for each record
  Declare one structure variable
  Open existing master file for reading

  Open new file for writing

  while (
    Not at end of transaction file
  ) {
    Read next transaction from transaction file
    Read records from master file,
      copying to new master file
      until this person's record found
    Check transaction details
    Amend record values
    Write this person's new record to
      the new master file
  }

  Copy the remainder of old master
    file to new master file
  Both files could be closed here

  Print any summaries ...
```

Alternatively, the while loop could be controlled by the reading from the input file, as in

```
  Declare structure type for each record
```

```
Declare one structure variable
Open existing file for reading
Open new file for writing
Ask "First person to amend? "
while (
  Not reached end-of-input-file
) {
  Read record from existing file
  if ( not person we're looking for ) {
    Write record to output file
    continue
  }
  Ask "details? ", read details
  Amend record values
  Write this person's record to output file
  Ask "Next person to search for? "
} /* end loop to end of file */

Both files could be closed here
Print summary ...
```

8.2.3. Interactive transactions

For interactive transactions (such as airline bookings) there must be a way of locking an individual record; is must not be possible for two customers to simultaneously request a spare seat, find that there is one, and attempt to both occupy the same single remaining seat! We are then into a new level of complexity.

8.3. Basic system calls for file input/output

We now look at the lower level facilities for file handling. These are provided by system calls to the UNIX kernel.

8.3.1. Opening a file

We now look at the basic system-provided input-output. File descriptors are integers. To open a file

```
int  fds; /* file descriptor */
char *file = "/tmp/eric";
  /* the filename */

fds = open( file, 0 );
  /* 0 = read, 1 = write, 2 = both */
if ( fds < 0 )
  exit( 1 );
  /* -1 returned if error */
```

The file must already exist and have read access. (Note that in the standard file i/o system earlier, if the file did not exist, it was

created; that is not the case here.)
The integer returned must be retained for future use.
The integer returned is the lowest available channel.
To create a file,

```
fds = creat( file, 0755 );
```

The second parameter is the file access mode, usually written in octal.
If the file already exists with write access, this call will empty it,
and point to its start.
If the file doesn't exist, it is created with mode 0755 octal.
If it exists but does not have write access, the function fails, and
returns the value -1.
If a file is created with access mode 0 (no access permissions), this
program can still write to it and read from it. No other program will
be able to access it.

8.3.2. Reading from the file

```
int  n;
char buffer[100];

n = read( fds, buffer, 100 );
```

The integer fds is the file descriptor (integer) returned from the open,
the second parameter is a pointer (of type "char *") to where the data
is to go, the third parameter is the number of bytes requested.

The returned integer (stored in n in this example) is the number of
bytes actually read.

```
100 normally,
< 100 if near end of file,
0 if at end of file,

-1 if error
```

To read a character at a time, use

```
char ch;

while ( read( fds, &ch, 1 ) == 1 )
   { ... ... ... }
```

The loop terminates at error, or at end of file.

To (binary) read a structure,

```
read( fds, &object, sizeof object );
```

8.3.3. Writing

The write call parallels the read call.

```
n = write( fds, buffer, 100 );
```

To copy one file to another

```
while(
  ( n = read( fdsin, buffer, 100) )
    > 0
) {
  write( fdsout, buffer, n );
}
```

8.3.4. Close

When you have finished with a data stream

```
close( fds );
```

This returns -1 if error. All files are closed on program
termination
anyway.

8.3.5. Random Access to a File

The lseek routine lets you move around a file.

```
long posn, offset;

posn = lseek( fds, offset, 0 );
```

The returned value is the absolute position in the file.   The
last
parameter is

```
0 = to position at `offset' bytes
1 = ... at current + `offset'
2 = ... at end-of-file + `offset'
```

Thus to move back to the start of the file

```
lseek( fds, 0L, 0 );
```

To skip to the end of the file

```
lseek( fds, 0L, 2 );
```

If the file is composed of (fixed length) records, use

```
lseek( fds, i * sizeof ???, 0 );
```

8.3.6. Removing a File

These are UNIX system calls.

```
unlink( "/tmp/effile" );
```

8.3.7. Creating a link

```
link( "oldname", "newname" );
```

8.3.8. Invisible Temporary Files

For workfiles invisible to the outside world, use

```
fds = creat( "/tmp/junk", 0 );
unlink( "/tmp/junk" );

write( fds ... );
read( fds, ... );
```

```
     close( fds );
```

8.3.9. Lock files

Some programs use the presence or absence of a lock-file to indicate the availability or otherwise of a resource (e.g. a direct link to another machine).

Problem : minimise time between

a) Does the file exist?

b) If not, create it.

```
     char *lock = "/tmp/vaxlock";

     fds = creat( lock, 0 );
     if ( fds < 0 ) {
       ... exists ...
       ... wait or exit ...
     } else {
       /* didn't exist, but does now */
       ... do work ...
       unlink( lock );
     }
```

There are problems with root. Use link instead of creat.

8.3.10. Default input/output channels

Three channels are already open.

```
     0 is standard input
     1 is standard output
     2 is error output
     write( 1, "message:\n", 9 );
     write( 2, "error", 5 );
```

8.3.11. Diverting Standard Input

Diverting input to an unopened file.

```
     close( 0 );
     fds = open( "new_file", 0 );
     /* returns zero */
     getchar() ... /* from file */
     scanf( ... ) ... /* from file */
```

Diverting input to an already opened file:

```
     fds = open( "file", 0 );
      ...
     close( 0 );
     fds1 = dup( fds );
       /* result is zero */
     close( fds );
      ...
     getchar() ...
       /* from current place
```

```
          in file */

     O Eric Foxley 1993
```

Chapter 9 Process Management

9.1. Fork, exec and wait

To initiate a new process, first `fork' to produce a duplicate of the current process.

```
     int pid; /* process identifier */

     pid = fork();
     if ( pid < 0 )
         { ... error, exit ... }
     if ( pid == 0 )
         { ... child ... }
     else
         { ... parent,
           pid is child's pid ... }
```

A typical child might be

```
     exec( program );
     /* overlay with new program */
```

A typical parent might be

```
     wait();
     /* wait for child to finish */
```

9.1.1. Variants of `exec'

```
     execl( "/bin/pr", "pr", "-2", "file", 0 );

     execv( "/bin/pr", argv ); /* see 6.3.1 ? */

     execve( "/bin/pr", argv, arge );

     execlp( "pr", "pr", "-2", "file", 0 );
```

The value of "argv[0]" is up to you.

Exec calls do not return. They are therefore typically followed by

```
     execl( ... );
     fprintf( stderr, "Error exec failed0 );
     exit( 1 );
```

9.1.2. Wait

wait returns pid of dying process:

```
     pid = wait( &i );
```

If you give a parameter (`int *'), wait returns exit status in i.

To control two children

```
     int p1, p2;
     p1 = fork();
     if ( p1 == 0 ) {
       ... exec child1 ...
       ... printf error ...
```

```
    ... exit( 1 );
  }
  p2 = fork();
  if ( p2 == 0 ) {
    ... exec child2 ...
    ... error etc ...
  }

  p = wait();  if ( p == p1 ) {
    ... first child died ...  } if ( p == p2 ) {
    ... second child died ...  }
```

9.2. Pipes (i) Parent writes to child

```
int pdes[2];

pipe( pdes );

if ( fork() == 0 ) { /* child */
  close( pdes[1] );
  read( pdes[0], .... );
} else { /* parent */
  close( pdes[0] );
  write( pdes[1], ... );
}
```

9.2.1. (ii) Parent read child's standard output

```
if ( fork() == 0 ) { /* child */
   close( pdes[0] );
   close( 1 );
   dup( pdes[1] );
   close( pdes[1] );
   exec( childprog ... )
   ... error exit etc ...
}
/* else parent */
close( pdes[1] );
read( pdes[0], ... );
```

9.2.2. (iii) Two-way communication

```
int pc[2], /* parent to child */
    cp[2]; /* child to parent */

/* open both pipes */

/* child */
{   close ( cp[0] );
    close( pc[1] );
    ... read pc[0], write to cp[1] ...
}

/* parent */
{   close( cp[1] );
    close( pc[0] );
    ... read cp[0], write to pc[1] ...
}
```

9.3. Interrupts (signals)

We need two facilities: one to send a signal to another process (dif-
ferent signals, numbered 0 to 15, are possible), and another to say
(early in a program) "if signal number so-and-so is received, do this".

```
     #include <signal.h>
```

defines mnemonics for signals.

9.3.1. Sending signals

To send to given process a given signal

```
     kill( pid, signalno );
     kill( pid, SIGINT );
     kill( pid, SIGALRM );
```

To send signal SIGALRM to current process in 60 seconds use

```
     alarm( 60 );
```

9.3.2. Receiving signals

On receipt of the numbered signal, `sigproc' is executed.

```
     sigproc()
     { ... to be executed on receipt
         of signal ...
     }

     main()
     { ...
       signal( signalno, sigproc );
       ...
     }
```

9.3.3. To ignore a given signal

```
     signal( signo, 0 );
      /* to reset to previous sigproc */
```

9.3.4. To save and reset the previous signal proc

```
     newproc() { ... the new procedure }

     int (*oldproc)() = signal( SIGINT, newproc );
     /* signal delivers pointer to old proc */

     signal( SIGINT, oldproc );
     /* reset old proc */
```

9.3.5. Long jumps

To control interrupts so that, for example in a menu driven system, an
interrupt brings you back to the menu, we need jumps (goto's!) back to
the start of the loop if an interrupt occurs.

```
#include "signal.h"
#include "setjmp.h"

jmp_buf inttrap;

settrap() {
     signal( SIGINT, settrap );
     signal( SIGQUIT, settrap );
     next_prompt( "Type ? for help" );
     longjmp( inttrap, 1 );
}

main() ...
   while (
     setjmp( inttrap ),
     ...,
     1
    ) {
     signal( SIGINT, settrap );
     signal( SIGQUIT, settrap );
     ...
    }
```

Chapter 10
Miscellaneous

10.1. Unions

Unions allow two ALTERNATIVE data items to share storage.

```
     union number {
             int i;
             float f;
             } x;
```

x can now be either an integer, or a float. Use

```
      ... x.i /* the integer */
      ... x.f /* the float */
```

Beware of assigning to `x.i' and then using the value of `x.f'.
There
are no machine checks.

You may typically have a marker to tell you the type of object
currently
stored.

```
     struct header {
         int type;
         ... various other fields ...
         union {
             struct {
                 ... this lot ...
             } s1;
             struct {
                 ... that lot ...
             } s2;
         } un;
     } hd;
```

The `type' field keeps an indicator of the type of structure stored.

```
     switch( hd.type ) {
       case 1:
         hd.un.s1.s1field = ...;
         break;
       case 2:
         hd.un.s2.s2field = ...;
         break;
       default:
         ... error ...
     }
```

10.2. Enumerated types

This will be familiar to Pascal freaks.

```
     enum { spade, heart, club, dia }
         suit;
     suit = heart;

     if ( suit == club ) { ... }

     enum suitype { spade, ht, club, dia };
     enum suitype suit;
```

```
     suit = (enum suitype) 2; /* club? */
```

Student's Guide to CEILIDH -

S D Benford, E K Burke, E Foxley

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

Introduction

Ceilidh is an on-line coursework administration and auto-marking facility designed to help both students and staff with programming courses. It helps students by informing them of the coursework required of them, and by permitting them to submit their work on the computer, instead of having to print things out and hand them in. It also marks programs directly, and informs the student and teacher of the mark awarded. The marking uses a comprehensive variety of static and dynamic metrics to assess the quality of submitted programs, of which details are in the paper by Zin and Foxley[1] (a copy of which may be stored on-line in Ceilidh, see below). Ceilidh also provides students with on-line access to notes, examples and solutions, and provides tutors with extensive course monitoring and tracking facilities.

This document is a guide for student users of the Ceilidh system.

==
Note: - A ceilidh is an informal gathering for conversation, music, dancing, songs and stories. Concise OED.
==

The Ceilidh system acts in a number of ways for students, tutors and teachers, and can support a variety of different courses.

There are different facilities for students (reading notes and coursework definitions, looking at examples, developing programs, submitting and marking work), and tutors (observing submitted work and marks) and teachers (amending course material, setting up exercises, performing plagiarism tests). The appropriate facilities are offered to appropriate users by the Ceilidh system itself, which takes note of the login identification of the user and compares it with lists of authorised users.

Using Ceilidh as a student

There are two ways of calling the Ceilidh system. Ceilidh may be used
to support several courses in your department. You can either enter the
system at a general level, and then choose the particular course you are
studying, or you can enter directly into the particular course you are
interested in.

To enter the system at the general level, the appropriate command (which

should have been set up by your computer systems administrator) is
ceilidh
Upon issuing the command ceilidh you will be greeted with the menu shown
in figure 1. --

```
CEILIDH system level menu
 lc    list course titles         | sc   move to named course
 vp    view papers                | pp   print papers
 clp   change printer             | h    for more help
 co    make a comment to teacher  | q    quit this session
 fs    find student               | ft   find tutees
===========================================
System level command:
```

Figure 1 : System Level Ceilidh Menu

==
Note: -- Example menus are shown in this document. Menus seen in prac-
tice may vary slightly from those shown, since the actual menu you are
offered reflects only those facilities available at the time.
==

This is the "system" level of Ceilidh and represents a department-wide
view of the system.

The commands which are available at this point are as follows.

lc

This command tells you which courses are available and supported by
the Ceilidh system, their full title and their abbreviation.

vp

If you are interested, you can use the "vp" command to view various
papers describing the workings of the Ceilidh system. A typical

response to this command would be

```
The stored papers are:
   ASQA   : Automated Software Quality Assessment
   CAL    : The Ceilidh Courseware System
   CLI    : The command line interface ceilidh
   Courseware : Courseware to support the teaching of
programming
   Install : Installer's Guide
   Oracle  : The "oracle" output recogniser
   Plag   : Departmental plagiarism and late work policy
   Qu-ans  : The question/answer marking program
   Student : Student Guide to Ceilidh
   Teacher : Teacher Guide to Ceilidh
Choose a paper :
```

which lists a selection of the available papers. If you reply with the short name of the paper (the first word on the line), the paper will be shown on the screen a page at a time through a paging command such as "more". Diagrams may not appear correctly.

It is possible to print a given paper which looks interesting using the "pp" command. Some papers containing diagrams may not view or print nicely on devices without appropriate facilities.

h

The "h" command offers a little more information on the significance of the different commands available to you in the Ceilidh system. This command is available at various points when you are using Ceilidh, and should give help relevant at the time.

q

This is the "quit" command to leave the Ceilidh system, and to return to your ordinary UNIX shell.

For courses with student registers, the following commands are also available

fs

To find details about any student registered on any of the courses supported by the system.

ft

To find details of the tutees of a specified tutor.

See below for discussion of the "clp" and "co" commands, both of which occur at many places in Ceilidh.

In general you will wish to move fairly soon to work on a specific

course which you are studying. A particular course is entered using
the
"sc" (select course) command by typing for example
"sc pr1"
followed by return to enter the course "pr1". You must use
lower
case/upper case (small and capital) letters exactly as requested.
If
the given name is not a valid course, all available course names will
be
listed and a valid one should be selected.

The system level can be bypassed by calling Ceilidh to enter a
particu-
lar course directly with a flag argument "-c" as in the example
"ceilidh -c pr1"
which will start up Ceilidh in the course "pr1" directly. Since
you
will be making extensive use of the Ceilidh system, you will
generally
find it simplest if you abbreviate this command using the "alias"
facil-
ity of the C-shell as in
alias pr1 "ceilidh -c pr1"
This will enable you to type

pr1

to enter the course directly. This "alias" command should be added
to
your .cshrc file (you will be given more details in lectures) if it
is
not already there.

The first time you call up a particular course you will be asked if
you
wish to register for the course. This assists the administration of
the

course.

The course and unit level

When you have selected a particular course, the menu shown in figure
2
should now be displayed on the screen.

```
 Course and unit menu for course "pr1" unit "1"
  lu     list unit titles            | su      set unit code
  lx     list unit exercise titles   | sx      move to named exercise
(1)
  lux    list units and exercises    | state   current exercise state
  vn     view notes on the screen    | pn      print notes on letter13
  csum   read course summary         | usum    read unit summary
  vm     view all marks              |
  clp    change printer              | h       for more help
  co     make a comment to teacher   | q       quit
 ===========================================
 Unit command:
```

|

|

Figure 2 : Unit and Course Level Ceilidh Menu

This menu is identical whether it is obtained from the system level
of
Ceilidh using the "sc" command, or by entering Ceilidh with a "-c"
argu-
ment.

We are now in a chosen course. The various possible commands have
the
following significance.

lu
Each course is divided into a number of units, rather like
the
chapters of a book. This option lists the name of each unit,
giv-
ing you a brief outline of the course as a whole. Typical
output
might be

```
Units in course pr1
  Unit 1: Background
  Unit 2: Elementary programming
  Unit 3: Conditionals
  Unit 4: Loops
  Unit 5: Functions
  Unit 6: Miscellany
  Unit 7: Arrays and structures
  Unit 8: File input and output
  Unit 9: Pointers
```

lux
This command lists all units and exercises within these units.

csum
If the teacher remembers to keep the information up-to-date,
this
command gives you a summary of the timetable for your course,
with
details of the courseworks to be set, and the hand-in dates
for
each one.

state
As a course progresses exercises are opened, made late and
then
closed. This command gives a summary of the state of each
exer-
cise.

su
This command enables you to select a chosen unit of the
course.
The menu remains the same, apart from the currently selected
unit
number which is included at the top of the menu. Commands
below
which relate to a specific unit use the currently selected

```
unit
     number.

usum
     This will list a brief summary of the currently selected unit,
usu-
     ally at the level of section headings in the notes.

vn
     This command (view notes) allows you to view on-line the notes
for
     the  current  unit  of  the  course.   The command "pn" gives you
a
     printed copy of the notes.  In general you  will  have  been
given
     duplicated copies of the notes, so that printing repeated copies
is
     to be discouraged;   it  wastes  paper  and  depletes  the
world's
     forests.

q
     This is the command to quit the system.  If you entered Ceilidh
at
     the course level with a command such as

          ceilidh -c pr1

     the quit will return you to your shell.  If you entered the
course
     level from the system level using first

          ceilidh

     and then

          sc pr1

     for example, the quit returns you to the system level  of
Ceilidh,
     and you will need another quit to return to your shell.

Your current unit and exercise will be noted, so that when you
re-enter
Ceilidh,  you  will  default  to  the same unit and exercise as when
you
left.  If you wish to quit without saving your current state, use

     q!

instead.

co
     At many points in the Ceilidh system, the system allows you to
make
     comments  to  the  course  teacher.   Comments  are always
welcome.

     Comments may be a request for help ("What do you mean  by  ....
in
     this  week's  question?"),  a criticism of the system ("I think
the
```

mark it gave me was not fair"), or an apology for the late hand-in of work ("Sorry but I had an examination ..."). Please feel free to use this facility; the teacher will try to answer most queries. The comments are sent using email to the teacher in charge of the course.

clp

Whenever you use a command which involves printing some information, the computer chooses the printer which it thinks is most convenient. This is done by looking at where you are on campus. Sometimes the computer chooses the wrong printer (it cannot always tell exactly where you are on a network), so there is a facility for you to choose a particular printer by name. You will be told appropriate printer names in class.

To work on your coursework, you will need to move from the "unit" level of Ceilidh into the "exercise" level.

The exercise level

If, for a given coursework, you are asked to solve a nominated coursework exercise in a this week's unit of the course, you will perhaps first select the appropriate unit using the, "su" command, then list the names of all the exercises in this unit using the command

"lx"

at the course/unit level, and then enter the required exercise using

"sx 2"

for example, to select exercise 2 of the current unit.

It is worth noting that at the course level, while the "sx" (select a particular exercise) command moves you to another level, the "exercise level" with another menu, the su (select a unit) command leaves you at the course level with the same menu. You can move around the different units in a course at will without changing your level in the system. To attempt exercises you must enter the exercise level, which has different menus depending on the type of exercise you are asked to complete. These exercises include compiled language exercises, interpreted language exercises, question/answer exercises and text submission (essay) exercises. For the moment we will consider the

compiled
language exercise menu.

If you type
"sx 1"
to select exercise 1 in the current unit of the course you will see the
menu given in figure 3.

This is the level at which most of your work will be undertaken. Each
exercise will have been set up by the teacher, and will include a ques-
tion, a skeleton solution, and all the necessary testing information.

Your normal sequence of activity at this level might be as follows.
First use

```
 Compiled language menu for course "pr1" unit "1" exercise "1"
  vq    view question on the screen   | pq    print question on draft13
  co    make a comment to teacher     | set   set up coursework
  h     for context help              | H     for general help
  q     to return to calling menu     |
  ed    edit your program             | cm    compile your program
  cv    compile verbose               | cks   check whether submitted
OK
  run   run your executable           | rut   run yours against test
data
  sub   submit/mark your program      | std   look at the test data
  vs    view solution program         | ps    print sol'n program on
draft13
  cp    get copy of solution          |
  rex   run solution executable       | rxt   run sol'n against test
data
 ===========================================
 Type compiled language command:
```

|

|

Figure 3 : Exercise Level Ceilidh Menu

vq

(view question) to look at the question, or "pq" to print it out. You
may need to study the question for a while before attempting its solu-
tion on the computer. It may be sensible to view or print it at least a
day before the laboratory session during which you solve the problem.

You will then use
"set"
to set up a skeleton solution. This command typically puts an outline
of the required program into your directory, to give you a flying start

in solving the problem. In more complex exercises later in the course,
it may set up other data files as well.

At this stage you can start to develop your program, using the commands

ed
to edit your program,

cm
to compile it (if the compilation fails, go back to "ed" to correct
the error with the editor, and then try compilation again), and

run
to try running your program. It is up to you to think of appropri-
ate tests when running your program, to convince yourself that it
is running correctly.

cv
This command is given as an alternative to the "cm" command. When
used it will compile your program more verbosely, giving compiler
warning messages which can help identify problems in your solution.

db
If this option has been set up by the course developer, it offers
debugging facilities to you.

Note: Not all of the options in the menu will appear on the screen at

all times; if there is no executable, for example, the running options
will not appear. If you have not executed "set" to obtain an outline
program, the "ed" command for editing your program will not be shown.

Once you have successfully compiled your program and tested it to your
satisfaction, the system is ready to mark and submit it. It does this
by looking at your program source code (checking that it is indented
correctly, for example), and running your compiled program against vari-
ous sets of test data and seeing that it produces the correct results.
At this stage you may wish to use the following commands.

rut
This runs your compiled program against the first set of test data
used by the marking process, and enables you to see whether it
appears to produce sensible answers.

```
std
     (show test data) This shows you each set of test data being used
by
     the  marking process.  The teacher reserves the right to change
the
     test data at any time, since your program should generally work
on
     absolutely any data which it receives.
```

When you have performed enough tests to convince you that your program is correct (and only then) you should ask the system to mark and submit it using the "sub" command.

The computer's response will be something like that shown in figure 4.

```
Analysis of Dynamic Correctness
                              item   mark   out of
                       Simple test
                 Negative distance
                Check "feet" "ins"
                       Inches > 12
                   Negative inches
Score for Dynamic Correctness is       .0%

Mark summary
                          category   mark   out of
               Dynamic correctness
             C++ typographic style
            C++ complexity measure
              C++ program features
Overall mark awarded
```

Figure 4 : Output from the marking command

The significance of this output is as follows.

Firstly your compiled program is run against several sets of test data. The system looks in the output generated by your program for evidence that you have produced the correct answer; this can be a non-trivial operation if your program does not print its results clearly! Each test produces one line of output, giving you a brief summary of the test, and the score you have been awarded. Different tests will be marked out of different totals, depending on the importance of the test.

The marks from these runs against test data are then combined

```
into
     a single "dynamic test" result for your program.

     This result is then scaled out of a particular value, and the
next
     few lines give marks for various "static tests" (tests performed
by
     looking at your program source, rather than by executing  it)
such
     as "typographic style" (your program layout, choice of
identifiers,
     use of comments, etc, see the ASQA paper[1] for details, a copy
is
     stored  on the Ceilidh system) "complexity" (the complexity of
your
     program is compared with the complexity of the  course
developer's
     model  solution;   the two should not differ by too large a
factor)
     and lastly "features" (the computer looks for specific good or
bad
     programming features associated with this particular coursework).

     All these marks are then combined with their weightings into a
sin-
     gle  mark which you are awarded.  The Ceilidh system retains a
copy
     of your program and of the mark awarded for future reference.

If you are happy with the mark awarded, you  can  quit  at  this
stage.
Alternatively,  you  may  try to improve your mark and try again.  It
is
your last mark which is recorded as your actual mark  for  this
course-
work.

To check that the mark has been correctly stored by  the  computer,
use
the command
     "cks"
(check submission) which will show you what the computer  has
recorded.
You should always use this checking facility after every exercise.

There is also a command at the course/unit level  "vm"  which  lets
you
view ALL your marks submitted so far.

Note:

(i)  Do not waste hours trying to obtain an extra mark or two.  It is
a
     misguided  waste  of  your  time.   Once  you  have achieved a
good
     overall mark, leave the Ceilidh  system  and  work  on  your
other
     courses!

(ii) Do not use the system to find bugs in  your  program.   Design
and
     test  your  program  thoroughly  yourself  before  you submit it
```

```
to
     Ceilidh for marking.

Other commands at this level are:

vs, ps, cp:
     These commands are available only after the  hand-in  date  of
the
     coursework,  and  let you view the solution ( "vs" ) to the
course-
     work, print the solution ( "ps" ), and copy the solution into
your

     own directory ( "cp" ) so that you can try it out yourself.

rex, rxt:
     These commands allow you to run  the  course  developer's
compiled
     program  interactively  (  "rex" ) to see that it works the way
you
     expected, and to run it against the first set of test data (
"rxt"
     ) to see the output which it gives.  This may give you ideas on
how
     to layout your output.  These options may not  exist  if  there
is
     insufficient  space on the disc for the teacher to store
executable
     versions of all the solutions.

When you quit ( "q" ) from the exercise level of Ceilidh, you return
to
the  course level of Ceilidh, where you may perform other activities,
or
execute another quit to leave Ceilidh completely.

Interpreted language exercises

The menu and process for interpreted language exercises  is  similar
to
the  compiled language menu described in the previous section.  The
com-
pilation commands are, of course, excluded.

Question/answer exercises

The exercise level menu for these exercises is completely different
from
that of the Compiled Language menu shown above.

For Question/Answer exercises you are given the following menu.
```

```
 Question/answer exercise menu for course "tst" unit "1" exercise "qu":
 vq    view questions                  | pq      print questions
 ans   answer questions and submit     | cks     check submitted
 h     help                            | q       return to calling menu
 Type question/answer command:
```

```
|


|
```

The options have significance as follows.

vq
    This allows you to view the questions before attempting to answer
    them. The "pq" command can then be used to obtain a printout of
    these questions.

ans
    When you are happy you know the answers to the questions set, you
    can enter your solutions using the "ans" command. This will then
    ask you the questions one at a time and read your response.
    Answers may be a choice between a few options, a word or a short
    sentence. To quit the exercise before answering all the questions
    type "q" as your answer.

cks
    This command allows you to check that your mark has been submitted
    correctly, and to check your answers.

Some question/answer exercises are purely for collecting answers, such as those to the end-of-course questionnaire. Other will involve answers

which are marked. The questions should make clear which of these cases holds.

Text submission exercises

For text submission (essay) type exercises, the system gives the following menu.

```
 ____________________________________________________________________
 Text submission menu for course "tst" unit "2" exercise "1": Type
 vq    view question                  | pq     print question
 set   setup a skeleton document      | ed     edit document
 sub   Submit document                | cks    check submission
 h     help                           | q      return to calling menu
 Type text submission command:
 ____________________________________________________________________
|                                                                    |
```

vq
    To view the question,

set
    to copy the skeleton solution to your directory,

ed
    to edit the skeleton solution, producing your essay or report, and

sub
to submit your file. Note that when you submit a text file, an entry of zero is recorded in the marking file purely for administrative reasons, so that the marking process has evidence of submission.

cks
In this case allows you to check your submitted work.

Miscellaneous additional Ceilidh commands

At most points in Ceilidh, there are three additional commands available.

cd directory
This permits you to change directory within Ceilidh. It does not change the directory you will be in when you leave Ceilidh.

!ls -l
Any command starting with an exclamation mark is treated as a UNIX command, and executed.

EDITOR=vi
You can temporarily reset your environment variables temporarily using this construction. There must be no spaces in the command. The setting is lost when you leave Ceilidh, or when you return to a higher level.

As you become a more experienced programmer, you may find the system becomes restrictive. Courses that teach modular programming and hence require more complicated compilation techniques often require the

student to leave the system to compile their solutions. To overcome these problems an alternative interface to the system has been developed.

The command line interface

This is a completely new interface in which, instead of using menus, each Ceilidh facility is represented by a UNIX command. It can be used on any terminal. Because there are no menus in this system, it is recommended that you use it only after some experience of the menu system.

To use this facility, there are two things you must do. First execute

```
~ceilidh/bin.cli/set.env
```

to set up an appropriate environment. You will need to check with your teacher just where the "~ceilidh" directory is on the machine. This needs to be done once only (unless at a later stage you wish to reset your environment).

In order to use these commands, the directory containing them must be included in your PATH variable. To do this, type

```
source ~ceilidh/bin.cli/source.csh
```

at the start of each logged-on session during which you wish to use Ceilidh.

From here on, type

```
commands
```

to get a list of Ceilidh commands currently available, or

```
status
```

to show the currently set course, unit and exercise.

The commands follow generally the pattern of the menu commands, but a few have had to be renamed to avoid clashes with existing commands.

A typical starting sequence might be

Command	Purpose
commands	See commands available
set.cse pr1	Select course "pr1"
commands	See extra course commands
lu	List unit titles
set.unit 4	Set a particular unit
lx	List exercise titles
set.ex 4	Select exercise to solve
vq	View question
setup	Set up program skeleton
ep	Edit program
cm	Compile program
run	Run program
sub	Submit
cks	Check submitted

Advantages of the command line interface

With this interface, you can execute other non-Ceilidh commands or even log out at any point. When you resume, the course, unit and exercise

will remain set just as when you last issued a Ceilidh command (although
you may choose to execute "status" to check the settings). This inter-
face will be particularly useful for the "pr2" course, in which you need
to perform all compilations yourself.

With this interface there is never any need to use "q" to quit the vari-
ous levels of Ceilidh.

At any time, type

    commands

to remind yourself of the commands currently available. The command

    status

shows the currently set course, unit and exercise.

Typing

    ~ceilidh/bin.cli/set.env

will clear out the currently set values for course, unit and exercise.
You will then need to use "set.cse", "set.unit" etc to reset them to the
values you require.

General points

At certain times, the teacher may close a complete course, or a unit, or
an exercise. These perhaps represent parts of the course which are
under development, or which must be kept unmodified for administrative
reasons.

Conclusions

The Ceilidh system is an essential part of your learning process; learn
to make good use of it.

References

1. Abdullah Mohd Zin and Eric Foxley, "Automatic Program Quality
    Assessment System", Proceedings of the IFIP Conference on Software
    Quality, S P University, Vidyanagar, INDIA (March 1991).

Course developer's Guide to CEILIDH

S D Benford, E K Burke, E Foxley

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

1. Introduction

The users of the Ceilidh system fall into the classes

users	those using the system as a learning tool
tutors	those with access to student progress monitoring they essentially have read access to student marks and work
teacher administrators	course management, setting up message of the day opening and closing exercises, etc they have write access to selected items
course developers	course material creation and management they essentially have write access to all material

In addition, there is a system support level, usually provided by the staff who run the computing service on which Ceilidh is running. This level is not at present distinguished from developer facilities.

This guide is for course developers, and should be read in conjunction with the other Ceilidh guides, the student guide[2] , the tutor guide[3] , the teacher guide[4] and the design document[5] .

The Course Developer

The course developer may want merely to set up new exercises in an existing course (a relatively straightforward operation once you are familiar with the principles involved), or to add new marking tools to an existing course, or set up a completely new course (a complete new set of marking tools will need to be written).

To use the developer's facilities, you will have to log in as "ceilidh." It is recommended that this is normally be done indirectly using "rlogin" or "xlogin" rather than directly by using a password.

At the moment, there is no distinction between different users logged in as "ceilidh." Course developer facilities and system

administration
facilities will both be available, and both are described in this
docu-
ment. This will be rectified at a later stage.

Most administrative and developer facilities are covered by menu
items
as described below. However, some specialised operations (sych as
the
installation of new marking programs) will need to be done by
hand
through the normal UNIX filestore mechanisms.

The support provided by the Ceilidh system falls into the four
distinct
levels

system	department-wide view
course	one university teaching module
unit	documentation and exercises for one course unit of a course
exercises	assessment definition for one item of coursework

Filestore layout

The current filestore structure is headed by a directory which we
will
refer to as "~ceilidh."   The menu commands (dumb terminal menus,
one
command for each category of user at each relevant level) are in
the
directory "~ceilidh/bin.mnu," the general tools (shell scripts and
pro-
grams called by menus, command line interface commands and X versions
of
Ceilidh) in the directory "~ceilidh/Tools," the command line
interface
commands themselves in the directory "~ceilidh/bin.cli," help
informa-
tion in the directory "~ceilidh/help," Ceilidh guides and
research
papers in the directory "~ceilidh/papers," miscellaneous data files
in
"~ceilidh/lib," and audit trails in the directory "~ceilidh/audit."

Below the "~ceilidh" directory each course has a directory such
as
"course.pr1" for the course "pr1" (Introductory C++ programming).
Below
the course directory are directories for each course unit,
typically
"unit.3" to contain the third unit of the "pr1" course. The unit
name
is always numeric, and indicates the sequencing of the units.
Below
this are directories for each item of coursework, such as "ex.1" for
the
first exercise or example. Exercise names can be any string of up

to
three characters*

==
Note: * At the moment we generally use numeric names in the C++
courses, and mnemonic strings in the C course, but mnemonics appear more
convenient since they can reflect relationships between exercises in
different units.
==

and exercises are not necessarily in a particular sequence.

All course, unit and exercise directories include a file "title" con-
taining a one-line summary of the course, unit or example. The course
directory should also contain a "summary" file containing a summary of
the lectures, times, courseworks set and hand-in dates, kept up-to-date
by the teacher using the teacher administrative facilities.

The unit summaries are assumed to be set up by the course developer, and
the course summary to be the responsibility of the teacher administra-
tor. The course directories also contain their own "bin" directory for
commands specific to that particular course.

The contents of the directories and the commands available at each of
these levels are described below.

2. Developer commands at the system level

In response to the command

     ceilidh

the menu for a developer or system administrator should appear as fol-
lows.

```
Ceilidh system top level menu:
lc    list course titles          | sc      move to named course
vp    view papers                 | pp      print papers
clp   change printer              | h       for more help
co    make a comment to teacher   | q       quit this session
fs    find student                | ft      find tutees
===========================================
Additional tutor menu:
ss    summarise one student
===========================================
Additional administrator menu:
```

```
  em    edit ceilidh system motd    | eh      edit help files
  nc    create new course           | audit   administer audit trails
  est   edit system staff.lst       | etu     edit system tutor.lst
  rdc   reset default course
  ===========================================
  System level command:
```

|

|

The ordinary user items (see the Student Guide to Ceilidh) are followed by additional menu items, one for the tutor (see the Tutor Guide), and some relevant to the system administrator, some to the developer. These two categories of user are not currently distinguished.

If the extra items do not appear, you are not logged in as the special username allocated for this purpose, normally "ceilidh," but possible another name set during the initialisation process. The additional administrative facilities include the following.

Edit system message-of-the-day (~ceilidh/motd)
Edit help files (all of the help files are offered)
Edit the top level staff and tutor lists
Create a new course
Set up a default course
Perform auditing operations

Default course

Using the "rdc" command, the system administrator can set up the default course which will be set when a new user calls Ceilidh at the top level for the first time. Thereafter the user will pick up the course they most recently used.

Within a course, it is the teacher (as part of the "new week" "nw" command) who sets up the default unit and exercise numbers.

Edit help files ("eh" command)

The help files live in the directory "~ceilidh/help." The names of help files are of the form

<category>.<level>

where the category is

sys	for system level help

cse	for course and unit level help
ex...	for exercise level help

and the level is

dev	for developer's help
tch	for teacher's help
tut	for tutor's help
usr	for user (student) help

Thus the teacher's course level help is in the file "tch.cse."   At the student exercise level, there are several files

expr1.usr	help when no compilable program source exists
exin1.usr	help when no interpretable program source exists
expr2.usr	help when no executable exists
expr3.usr	help if source and executable exist
exes.usr	student help for text submisison exercises
extxt.dev	developer help for text submisison exercises
exmc.usr	student help for text question-answer exercises
exqa.dev	developer help for text question-answer exercises

Miscellaneous help files include

sys.aud	audit facilities
en.dev	edit notes
enms.dev	edit master notes

Maintaining audit trails ("audit" command)

There are minimal elementary auditing facilities built into the system from release 2.1. Although this facility really intended for the system administrator, it will be described here, since the system administration and course development facilities are somewhat intermingled. There are no audit trail analysis facilities provided, just their generation.
Suggestions would be welcome.

Separate trails can be kept of all use of certain Ceilidh operations, or
all use by a particular user. These separate trails can each be
switched on or off at will. If a trail file of the appropriate name
exists, the trail information will be appended to it whenever necessary.
It must be remembered that such facilities can result in the collection
of large quantities of data; this must be archived and removed regu-
larly. The files containing the trails are stored in the directory
"~ceilidh/audit."

The audit administrator option calls up a separate auditing menu. The
auditing menu allows the developer/administrator to perform the follow-
ing functions.

o Check the state of the currently set audit trails. This command
gives the name and size of the audit trail file for each facility
and student currently switched on. Sizes and dates can be checked
to see which files need to be archived. We could give the number
of entries in each trail if required. There are also separate com-
mands for showing separately the feature trails set, and the

student trails set.

o Show a list of those facilities not currently switched on. These
are the facilities which could be added.

o Switch on the auditing of a new trail (either a particular facility
or a named user). An empty trail file of appropriate name is
created.

o Switch off the auditing of a particular facility or student. The
named trail file will be removed. The user will first be invited
to save any existing stored information. If required, it will be
appended to a trail in the directory "~ceilidh/audit/archive."

o A particular audit file can be viewed using the standard pager defined by the environment variable "$PAGER." Each record is a single line of the form

```
course:unit:exercise:user:date-time:user:facility:information
```

These follow the same format as mark information. If the appropriate trail files exist, an entry will be appended to both the facility file and the student file whenever that event occurs.

- The user can edit an audit trail file, perhaps to remove unwanted information after it has been inspected, but leave other entries intact.

- An existing trail can be archived, i.e. appended to a file of the same name in the directory "~ceilidh/audit/archive." To use a different archive directory, edit the setting of the "ARCHIVE" environment variable in the command "~ceilidh/Tools/CAudit."

- The archived trails can be viewed or edited as for the current trails.

The "Error" audit should always be switched on, and should be checked regularly.

Create a new course ("nc" command)

This command requests a course abbreviation suitable for use as a directory suffix (eg. the C++ programming course "pr1" is contained in the directory "~ceilidh/course.pr1") and a one-line course title. The course name should be at most three characters, with no spaces, to conform to MS-DOS practice. The command then requests the login names and full names of staff who will be permitted to administrate this course.

It sets up the course directory, its "bin" and "bin/SOURCE" subdirectories, and the "title" and "staff" and "tutor" files. The course "bin" directory may eventually contain compilation and marking shell scripts and programs specific to this course, with their source and "Makefile" in "bin/SOURCE" .

If there is a command "CCompile" in the course "bin" directory, it is used for all compilations in preference to the main command in "~ceilidh/Tools/CCompile." It is called with two arguments as in

```
CCompile <source> <executable>
```

and expects a "true" exit to indicate successful compilation. The actual compiler called is defined by the

```
CC=
```

environment variable set in the exercise "type" file.

Similar considerations apply to "CVCompile" (the verbose compilation option).

If you require a debugging command for the students, there must be a "CDebug" command in the "bin" directory. This will be called as

```
<course directory>/bin/CDebug <course> <unit> <exercise>
```

When the marking process (see later) requests a program such as "typog_c" for C typographic metrics (or "typog_C" for C++) it searches first the course "bin" directory for an executable "typog_c," and if this is absent uses the "~ceilidh/Tools/typog_c" version.

The "nc" command then moves into the newly created course, and offers the developer's course menu, see below for details.

When the user finishes creating the new course, there is an option to remove the course. This is really to ease practising setting up a new course.

3. Developer commands at the course and unit level

The developer may either move to this level from the system level by typing

```
sc pr1
```

or will more normally enter the course "pr1" for example directly, by calling

```
ceilidh -c pr1
```

either directly or using a preset alias. We normally set an alias "pr1"
for this command. There will always be a currently set default unit and
exercise name.

To use course developer facilities, choose the

     develop

option on the menu*.

===
Note: * If this option does not appear, you are not logged in as
"ceilidh," or the main login username has not been set correctly in the
installation process.
===

You can use instead the command or alias

     ceilidh -c pr1 -x develop

to enter the developer's menu directly on the course "pr1". We normally
set an alias "pr1d" for this command.

The menu is now

```
 Developer's course (pr1) and unit (1) level menu:
 ect   edit course title       | eut   edit unit title
 eus   edit unit summary       | en    edit notes
 cx    make copy of exercise   | mx    rename exercise
 lu    list units              | lx    list exercises
 su    set unit number         | sx    move to exercise (1)
 nu    create new unit         | nx    create new exercise
 ar    archive course          | zap   remove student work
 h     help                    | q     return to user menu
 ===========================================
 Type developer course command:
```

The menu options can be divided into course-related items, unit-related
items, and exercise-related items.

Course related menu options

ect
     Edit the course one-line title file. All edit operations are, of
     course, implemented using the editor specified in the "$EDITOR"
     environment variable.

```
lu
     List all units in the course.

ar   This makes an archive copy of the course directory and all its
con-
     tents  (including student solutions and mark files, and weights
and
     mark scaling files), using a standard  UNIX  "shar"  command.
The
     resulting file will be large if there is much coursework
submitted.
     The Ceilidh group at Nottingham would appreciate  receiving
copies
     of any such files for statistical analysis purposes.

zap  This removes all student submitted work and marks from the
current
     course,  and resets permissions as though the course had been
rein-
     stalled.  You will normally have archived the  material  first
for
     future reference.

Unit related menu options

su
     Set the currently selected unit number.

eut
     Edit the one-line title of the currently  set  unit,  in  the
file
     "title" in the unit directory.

eus
     Edit the unit summary file "summary" in the unit directory.

en
     Edit notes.  See the section on notes below for details.

nu
     Create a new unit for the course.  This requests a numeric
identif-
     ier for the unit, and a one-line title in a similar way to the
"new
     course" command.  It sets up a new unit directory "unit.<unit>"
and
     "title"  file under it.  At the end you will be offered the
removal
     of the new unit just created.

Exercise related menu options

mx   Move an existing exercise in the current unit.  This  requests
the
     exercise identifier, and a new unit number and exercise
identifier.
     The exercise and all its files are moved to the new  position,
```

```
and
     any files internal to the exercise are edited to reflect the
change
     where possible.  You will need to perform hand edits if  the
notes
     or  other  exercises cross refer to "exercise .. of unit .."
which
     has been renamed.

cx   Copy an exercise which already exists in the  current  unit.
This
     requests  the  exercise identifier, and a new unit number and
exer-
     cise identifier.  The exercise and all its files are copied to
the
     new position, and edited as required.  This is often an easy way
of
     setting up a sequence of related exercises in different units.

nx   Create a new exercise for the current unit.  This will  request
an
     identifier  (up  to  three  characters) for the new exercise, and
a
     one-line   title.    It   then   requests    an    exercise
type
     (question/answer, text submission, programming) and details such
as
     the compiler to be used, and the file  suffix  required.   It
will
     then  move into the exercise menu (see below) for the new
exercise.
     When you leave the exercise menu, it will offer deletion of the
new
     exercise.

lx   List all exercises in the currently set unit.

sx   Move to a named exercise in current unit.  This gives the
exercise
     menu, see below.

Notes

All supporting notes for a course are divided among  the  course
units.
For courses we distribute, in each unit there are files

      notes.cat   ASCII copy, viewable by "$PAGER" (e.g. "more")
      notes.ps    PostScript notes for printing or viewing
      notes.ohp   PostScript notes for overhead projectors

      |
                                                            |

Any diagrams will not appear correctly in the ".cat" version.

We generate these from a master copy kept at  Nottingham,  which  is
```

in
"roff" format. Copies could perhaps be distributed if volunteers were
prepared to improve them significantly.

We also keep at Nottingham ".dvi" files in device independent troff
(ditroff) form if required. These can be viewed through "xditview" or a
similar program.

The student generally accesses the ASCII notes on-line through the "vn"
option (which calls the user's command defined by the "PAGER" environ-
ment variable) to view the ".cat" version of the notes on the screen.
It would be easy to add an X viewer for viewing the PostScript file if
required, or a "ditroff" viewer.

The option "pn" prints the PostScript notes "notes.ps" to a default
printer.

The overhead projector slides are in file "notes.ohp" in PostScript, and
must be printed by UNIX commands if required.

The "en" developer option displays an extra menu, which allows the fol-
lowing operations.

You can edit the ASCII notes in "notes.cat."

You can copy new named files of your own into the "notes.cat" and
"notes.ps" files.

You can view or print either version of the notes.

If you have the master "roff" copy of the notes in file "notes.ms" you
will obtain a different menu allowing editing of the master file, and
the re-creation of all the other files.

Annual tidying of courses

Two of the above commands ("ar" and "zap") relate to this activity.
They are currently in the developer's course level menu, but should
perhaps eventually be in the teacher's menu. UNIX file permissions make
this difficult.

4. Commands at the exercise level

To work on a given exercise, use the command

```
sx 5
```

for example, to work on exercise 5 of the current unit.

Each exercise is stored in a directory such as

```
~ceilidh/course.pr1/unit.2/ex.5
```

for course "pr1", unit "2", exercise "5". The "sx" command moves you into the exercise directory, and offers you a new menu. In addition to the commands offered on the menu, the usual additional facilities of replying with an exclamation mark followed by a UNIX command can be use- ful here, for example

```
!ls -l
```

to see the names of the files in the exercise, or a command such as

```
!emacs model.q
```

to edit one of them directly.

The exercise may currently be of several basic types:
- Question/answer exercise
- Text submission (essay) exercise
- Compiled programming exercise
- Interpretive programming exercise
- Programming example (no marking)

The type is defined by a file "type" in the exercise directory, in which the various details of the exercise type and requirements are specified using Bourne shell notation.

After submission of work by students, the subdirectory "ex.<exercise>/solns" will contain the students' submitted work in files starting with their username (program sources in files named "<username>.c" or "<username>.C" for programming exercises, perhaps files "<username>.txt" etc for essays), and their mark histories (in a single file "marks" containing all marks for that exercise) for any marked exercises.

The "marks" file consists of one line entries with colon-separated fields. Each line is of the form

```
     <course>:<unit>:<ex>:<student>:<date>:<entered by>:<total
     mark>:<other marks>
```

All marks are kept, each date- and author-stamped by the person who
entered it. Only the last entry for a particular student is used in
summaries. The "entered by" field contains the student username for
work submitted by the student, or the tutor's username for work remarked
by the tutor.

Submitted text files (such as program sources, essays ...) are normally
overwritten at each submission. There are now facilities to keep all
submissions using RCS or SCCS, see later for details.

At the exercise level, the user and developer see one of several dif-
ferent menus depending on the type of the exercise, and the contents of
the directory.

Question/answer exercises

The "type" file here will be simply

```
     TYPE=QA
     SUFF=mc
```

This type of exercise is useful both for marked exercises, and for end-
of-course questionnaires where the results are collected for analysis,
but not marked.

The exercise directory will contain (in addition to the usual "title"
and "model.q" question files) a question/answer script file "model.mc"
containing the vital information for display to the student, and for
checking the responses. The key program here is the question/answer
program (see the Ceilidh document[7] for more details)
"~ceilidh/Tools/multi."

The answers may be checked using a variety of recognisers, for checking
multi-choice answers ("mchoice") or numeric answers ("numeric") or
general regular expression recognisers ("oracle" , see the document by
Zin and Foxley[9] for details). There is also a semantic answer
recogniser at Nottingham, restricted for copyright reasons, which it

is
hoped will be distributed at some stage.

The student answers to the questions are combined into a single file
using the standard UNIX "ar" (archive) command. When a student uses
"cks" to check submitted information, the separate responses are
extracted from the archive file.

Note that the script file should have no public read access, since it
may also contain the oracles which check the student's answers to the
questions.

The developer's exercise level menu for QA exercises offers
editing of the script file,
editing the question and title files,
and rerunning the "multi" program in a verbose marking mode.

Text submission exercises

The file "type" here will be of the form

```
TYPE=ESSAY
SUFF=txt
```

The menu options for the student are
read or print question

obtain skeleton of essay in local file
edit local essay file
submit local essay file

The exercise directory will contain the title file, the question file
"model.q" defining the problem to be solved, and an essay skeleton in a
file such as "model.sk." The exercise is purely (at present) for text
submission purposes. When the texts have been submitted, they are
stored in the usual way in the exercise solutions directory, and the
developer can (directly under UNIX) print out the files, or look at them
on-line. It would be expected that the marks would then be entered by
hand using the teacher's "em" (enter marks) option. On-line viewing,
annotation and marking of essays may be provided later.

A student can edit text files (program source or essays), of course,
outside the Ceilidh system if preferred. The use of Ceilidh to adminis-

ter essay questions can considerably simplify the collection and administration of work, even though there are no formally offered essay marking facilities yet. Any submitted essays appear in the "solns" directory, under names such as "<username>.txt" for each student.

The developer's menu offers editing of the question or the skeleton, or of the setup actions file "setup.act." The latter file (used as the basis when the student types "set" at the beginning of solving an exercise) will be of the form

```
Copy model.sk prog56.txt
```

This will copy the file "model.sk" in the exercise directory to "prog56.txt" in the student directory. We use the skeleton file as an essay outline, with major subheadings for the student report or essay indicated.

There will always be a "mark.act" file, which in text submission exercises will consist of the single line

```
Save o txt
```

to save a copy of the student file "prog56.txt" for unit 5 exercise 6 in the file "<username>.txt" in the solutions directory.

Example exercises

If there are no program dynamic test files, it is assumed that this directory represents an example without marking facilities. The "type" file will be set to for example

```
TYPE=EXAMPLE
SUFF=C
CC=g++
```

to specify the program suffix and compiler command. It will typically be a non-handed-in informal coursework or programming example. The student menu is then

```
view or print question
obtain copy of solution program
```

Programming exercises

For program development (our major application so far) the user and
developer menus are considerably larger. The developer's menu is now

```
CEILIDH developer's exercise menu (Course pr1, unit 1, ex 1):
et    to edit title                  | ety   to edit the "type" file
ep    to edit program                | eq    to edit question
es    to edit skeleton               | em    to edit Makefile
esa   to edit setup actions file     | ema   to edit mark actions file
cm    to compile                     | run   to run
sd    to set dynamic mark scheme     | sf    to set "features" mark
st    to set typographic weights     | sc    to set complexity weights
se    to set efficiency test         | ss    to set structure test
mk    to mark the model soln         | mv    to mark more verbosely
mks   to mark a studs soln           | mka   to remark all studs
ch    to do a complete check         |
h     help                           | q     to return to outer level
===========================================
Type developer's exercise command:
```

|

|

The "type" file will typically be of the form

```
TYPE=COMP
SUFF=C
CC="g++"
```

specifying the exercise type as "compiled programming", the program
source file suffix, and the compiler command. There is a number of
optional extra entries which can be specified in the "type" file on a
per-exercise basis.

"MAXSUB=20"

This will limit any student's maximum number of submissions to
20. The student will be warned at each submission of the
upper limit, and of the number of submissions made so far. A
value such as -5 sets no limit, but produces messages after 5
submissions telling the student how many submissions have been
made.

"MINGAP=300"

This sets the minimum time gap between submissions to 300

```
          seconds.

     "OUTOF=10"

          This sets all mark totals seen  by  the  student  (except
for
          individual  dynamic  tests) to be scaled out of 10, instead
of
          as a percentage.  Such coarser granularity of  marking
should
          reduce the student temptation to tweaking.  One day we hope
to

          enable coarse granularities such as

               OUTOF=2

          to be expressed to the student as letters (e.g. "A", "B",
"C"
          instead of numeric values 2, 1, 0 respectively).

     "SAVEOUT=yes"

          This will cause the output from the runs of the  student
pro-
          gram  against test data to be saved in a file
"<username>.out"
          in the solutions directory.

     "VDATA=no"

          The "view test data" student option lists all of the test
data
          files in turn to the screen, and can be turned off by putting

               VDATA=no

          in the "type" file.  An intelligent student could, of
course,
          write a program which echoes its input to a file, and find
the
          data that way by submitting the program.

     "C_ORACLE=my_oracle"

          to specify a particular oracle.  This program will be
searched
          for  first  in  the "~ceilidh/course.<...>/bin" directory,
and
          then in the "~ceilidh/Tools" directory.

The compilation options on all menus will call "CCompile" in the
course
"bin"  directory if it exists, or in the "~ceilidh/Tools" directory
oth-
erwise.  The standard Ceilidh "CCompile" shell script uses "$CC" for
its
compilations.
```

5. Setting up programming exercises

This important activity will now be described in detail.

For programming assignments the student will generally start by first
calling a setup option to set up any skeleton or header files in the
local directory, will then develop the program for the solution of the
given problem (which can be done outside this system if required), and
will then mark and submit it.

5.1. Setup

The student executes the setup command once before solving the exercise;
the command uses a file in the exercise directory "setup.act" containing
lines such as

```
Copy model.sk prog72.C
Copy header.h
```

Any line with two filenames causes the exercise directory file such as
"model.sk" in the above example to be copied to the student's directory
under the name "prog72.C." If the line contains only one name, the
source and target names are assumed to be the same. The setup actions
file is edited using the "esa" developer option, and the skeleton file
"model.sk" using the "es" option.

The "esa" command gives a default file which may be adequate in simple
cases. Execution of the setup function by the student will not
overwrite existing files in their directory. On any copying it does,
the string "$USER" in the source file is replaced by the user's login
name as the file is copied, "$DATE" by the current date, and "$NAME" by
the user's full name.

The "es" command starts the skeleton with a copy of the model solution;
it is sensible to set up the skeleton after the model solution has been
completely developed. It is assumed that the developer will take the
solution program, and then edit out all the parts which the student has

to create. There may be a file such as "../../box" (in the course directory) containing a comment box such as

```
////////////////////////////////////////////
//
// Put your program instructions here
//
////////////////////////////////////////////
```

for C++ programs, which you may choose to insert with your editor at appropriate points. Consistent use of features such as this makes the student's understanding of what they have to do much clearer.

5.2. The question

The command

```
eq
```

allows you to edit the student question. Experience has shown the value of using a standard structure outline for the question file, such as

```
Question:
Method:
Notes:
Typical input:
Typical output:
```

You can, of course, make the question as helpful or unhelpful as you wish.

5.3. Program development

This stage of student activity can be performed outside the Ceilidh system if so desired; we anticipate that the student will work completely within Ceilidh during the early stages of the course, but outside during

the later stages.

The student will have a source such as "prog57.C" and executable such as "prog57." In the exercise directory, the source will be linked to "model.C" and the executable to "model."

The options available to the student for development of their programs with Ceilidh include the following.

The edit option calls their chosen editor as defined by

```
the
     environment  variable  "EDITOR"  with  the user's program source
as
     argument.

     The   compile   options    use    the    Ceilidh    shell
script
     "~ceilidh/Tools/CCompile" to compile the source from a file such
as
     "prog72.C" into the corresponding  executable  file  "prog72."
It
     first  looks  for  a  "Makefile"  in the exercise directory.  If
it
     finds a "Makefile" containing a dependence such as

          prog72 :

     in the unit 7 exercise 2 directory, for example, it compiles
using
     that entry with a

          make prog72

     command.  (The "Makefile" can also contain an entry

          model : ...

     to assist the course developer;  this will not  be  usable  by
the
     student.)   If there is no "Makefile" in the exercise directory,
or
     no

          prog72 :

     entry in it, the system uses a default "CCompile" shell scripts
and
     the "CC" command, which would typically default to

          $CC -o prog72 prog72.C

     If there is  no  appropriate  "Makefile"  entry,  and  the
example
     involves  more  than  one  module, then the student must create
the
     executable by hand outside Ceilidh.

     There is a "Makefile" available which will put the user back into
a
     shell for compilation.  This is offered to the developer when
using
     the

          em

     (edit makefile) command, and takes the form

          $PROG : $PROG.C
```

```
                rm -f $PROG
                cat $$C_LIB/NoComMsg
                $$SHELL
```

where the file "NoComMsg" contains

```
          There are no automatic compilation facilities in this
exercise.
          The
          system will now put you into a shell. Execute the required
          compilation
          commands, then quit the shell. To do this, if you are using
the
          C-shell, type "exit", if you are using the Bourne shell, type
          "control-D".
          You will then be returned to the point where you left the
Ceilidh
          system.
          Now wait for the shell prompt .....
```

The "run" option runs the student's executable such as "prog57" interac-
tively from the terminal.

The student can also use a "rut" option to run their executable against
test data supplied by the developer (it uses the first of the data sets
provided for the marking process, see below).

Students have a "vtd" command to enable them to view the files of test
data used in dynamic tests during the marking process. This can be
turned off by adding the line

```
     VDATA=no
```

to the "type" file.

If the developer has left an executable program in the exercise direc-
tory, the student can use further options to run the developer's execut-
able interactively, and to run the developer's executable against the
same test data. This allows the students to compare their answers with
those presumably expected by the developer. When a course is set up (by
un-shar-ing the distributed shar file) there are no executables. If you
require executables to specific exercises to enable students to run
them, you must use the developer's compile command to create them as
required. Generally the C++ executables we get are quite large (around

300kb), so they are created only when required.

The extent to which we should assist the student's program development
(by repeated submission, by releasing the test data, by offering an exe-
cutable, etc) is very open to debate; comments and suggestions are
always welcome. The system can easily be adapted to offer more or less
help on different exercises.

5.4. Marking and submission

The student calls the "sub" option to mark the program which she or he
has developed. The default submission/marking process is driven by a
file "mark.act" in the exercise directory, and expects to find in the
current (student's) directory a source in a file such as "prog72.c" or
"prog72.C" and a corresponding executable*

==
Note: * Earlier versions of the marking process recompiled the source
before marking to produce a guaranteed up-to-date executable; the
extension to numbers of object modules made this difficult and undesir-
able.
==

in a file "prog72" . The submission/marking option generally submits a
one-line summary of the marks and a copy of the current version of the
student program source to the "ex.<exercise>/solns" directory; if the
lecturer wishes, lines of the user's source code starting "#include"
which refer to a local file may be substituted by that file if it is
local and readable. In general, only the latest copy of the source is
retained.

The mark.act file

If a line such as

```
Save o C
```

exists in the "mark.act" file, the intention is that the student' source
(ending ".C", the last argument of the "Save" line) will be saved at

```
each submission, overwriting (the "o" specifies "overwrite", specify
"a"
if  each  submission is to be appended rather than overwritten)
previous
copies.

The following generalisation is an enhancement in release 2.2.

A line such as "Save o C" will first  search  the
"course.<course>/bin"
and "~ceilidh/Tools" directories for an executable "CSav_C" (the name
is
"CSav_" followed by the given suffix).  If such a file is found,  it
is
executed with arguments

     CSav_C -o <course> <unit> <exercise>

If it is not found, the student's source is saved directly as
described
earlier.  It is thus possible now to write special save commands such
as
"CSav_..."  as shell scripts if required.  Each  such  command  needs
a
corresponding  "CVw_..."   command which is called when the student
exe-
cutes a "cks" command to view their submitted work.

Three new "CSav_..."  commands are provided in the "Tools" directory
of
the current release as follows.

ar   A line

          Save o ar

     in conjunction with a line

          ARFILES="prog56.C header.h hotel.C"

     in the "type" file will cause the  named  files  from  the
student
     directory to be combined using the UNIX "ar" archive command into
a
     single stored file

          <username>.ar

     in the "solns" directory.  This facility is implemented by the
com-
     mand  "CSav_ar"  in the "Tools" directory for performing the
saves,
     and "CVw_ar" to enable the student to view the stored files.

     The command "CSav_ar" has code included such that  if  the
student
     has  a  file  "save.lst" containing the names of files to be
saved,
```

this list is used in preference to the "ARFILES" list.

rcs A line

```
Save o rcs
```

saves the student source in a file "<username>.rcs" in the solutions directory using the RCS (Berkeley "Revision Control System", a source code control program) command. All versions of the program submitted by the student are thus available to the teacher afterwards; the student just sees the most recent version during a "cks" (check submission) command. Viewing the RCS files has not been built-in as a teacher command; the demand for it is uncertain. The RCS file would enable a teacher or researcher to follow the student's progress through an exercise. The shell script to perform the RCS collection is in the file "~ceilidh/Tools/CSav_rcs."

exe The line

```
Save o exe
```

in the "mark.act" file causes the student executable to be saved in the "solns" directory under the name "<username>.exe" The command "CSav_exe" looks for either of the files "prog<unit><exercise>" or "prog<unit><exercise>.exe" in the student directory. Beware that executables can be very large; storing large numbers of them can consume vast areas of disc.

The marks (one line) are appended to the file "marks" at each submission. The submission/marking process cannot be called by one user more frequently that a general time interval set by the developer; this is currently set to 600 seconds, and is set in the source of the controlling program "ccef.c" (in file "~ceilidh/Tools/SOURCE/ccef.c") as the "#define MIN_GAP" constant (in seconds). You must then re-make the program. The value can be reset on a per-exercise basis by a line

```
     MINGAP=60
```

(in seconds) in the exercise "type" file.

The number of submissions can be limited by a line

```
     MAXSUB=10
```

in the "type" file. The student will receive messages informing how
many submissions have already been made, and what the upper limit is,
typically

```
     You have now made 4 out of a maximum of 6 submissions.
```

If a negative value is set as in

```
     MAXSUB=-5
```

the student will be informed of the number of submissions made after (in
this case) the 5th, but no upper limit is set.

The overall mark awarded is made up from a number of sub-marks, the
weights and sub-tests being specified in the "mark.act" file. A typical
"mark.act" file might be

```
     60 Dynamic
     25 Typographic: typog_C -v1 $X/model.tv < prog$U$E.C
     15 Complexity: compl_C -v1 -x $X/model.cm -f $X/model.cv <
prog$U$E.C
     Save o C
```

The "Save" line has already been explained. The other lines have signi-
ficance as follows.

The first three lines specify that the overall mark is to be calculated
from three tests, dynamic, typographic and complexity. The dynamic
tests are to be scaled out of 60, the typographic test out of 25, and
the complexity test out of 15. If these marks do not add up to 100,
they are further scaled during the marking process.

Each sub-test will also include weights for its sub-components, of which
details will be given later.

The overall mark awarded is formed from a number of components, of which
the following are provided.

```
     Dynamic correctness
```

Dynamic efficiency (C on SUNs only)
Typographic style
Complexity level
Features of the program
Structure of the program

Others could be added as required. These components are then combined to form a single percentage mark.

The "mark.act" file is set up when the exercise is created initially with one line such as

```
Save o C
```

for saving the program source. Other lines are added as the tests are set up by the developer.

The marking commands are normally specified in the "mark.act" file. Any marking program must produce output ending with a line such as

```
Score 75
```

as a percentage. The variables introduced by dollar symbols have significance as follows.

$C	course abbreviation
$U	unit abbreviation
$E	exercise abbreviation
$X	the exercise directory

|

|

The dynamic test is an exception. If it is abbreviated as in the example above, the marks for the individual sub-tests will be displayed to the student. If the test is given explicitly as

```
60 Dynamic: CDynCorr -v1 $C $U $E prog$U$E
```

only the total overall dynamic score will be shown.

5.4.1. Dynamic correctness

These are the most difficult tests to set up.

The student executable program is run against various sets of test data (or shell scripts) provided by the developer, and an "oracle" searches the program output for signs of correctness in each test. For details of

how the "oracle" program works, and how its marking weights are set, see
the oracle[9] document. All the tests and associated information can be
set up using the developer's "sd" (set dynamic) option.

Each test must have a short title, and a total mark. These are speci-
fied in the "model.dv" file, which might take the form

```
Dynamic Correctness
25 Simple data
20 Zero denominator
15 No data
35 Longer test
```

This specifies four tests (the first line must be there, but can other-
wise be ignored): the first test (worth 25 marks) using simple data,
the second (worth 20 marks) involving care over a zero denominator, the
third (15 marks) requiring a check for data, and the last (35 marks)
being a substantial test. If the marks do not add up to 100, the final
total is scaled appropriately. This percentage total is then scaled for
the overall mark awarded as specified in the "mark.act" file. The

chosen titles can be as helpful (as above) or unhelpful ("Test 1", "Test
2") as you require. The test data for the first test will be in the
file "model.d1," for the second in "model.d2" etc. The oracle data (a
set of regular expressions to search for the required information in the
output of the program under test, see below) for the first test will be
in the file "model.k1," for the second in "model.k2" etc.

The default oracle program is in the "Tools" directory; another oracle
can be specified by inserting, for example

```
C_ORACLE=search
```

in the "type" file.

The default oracle works (by calling "awk") using sets of regular
expressions (REs) devised by the developer; these will usually be
designed to recognise alternative output formats which are close to what
the developer asked for, but cannot in the end recognise every

semantic
possibility. Each RE can be preceded by a mark (it is otherwise given a
default mark of 10), and/or a limitation on the number of occurrences
(full marks are otherwise awarded if the RE is found at least once in
the program output). The percentage mark awarded for each dynamic test
is based on the sum of the marks for the REs found divided by the total
of all possible marks.

Suppose we wish to create a file of REs for the oracle to check output
which should read roughly

    A temperature of 124.6 degrees F converts to 32.7 degrees C

The "oracle" file of REs might be

```
10:32.7
5:124.6
2:degree
1:temp
==0:20:[Ee]rror
~10:Fail
```

This awards 10 marks if the result contains the string "32.7," a further
5 marks if it contains the string "124.6" (at this stage students were
asked to print out copies of all entered data), and so on. If the out-
put contained the string "Error" or "error" then 20 marks would not be
awarded (read the "==0:" as "there must be exactly 0 occurrences of
..."). The last line will subtract 10 marks if a line containing "Fail"
is found. The total awarded is then scaled out of the total of the pos-
sible positive marks, 38 in this case, and given as a percentage. The
full stops in the above REs (representing decimal points) should
strictly be expressed as

```
10:32\.7
5:124\.6
```

since a full stop in an "awk" RE represents any one character, whereas

an escaped full stop represents a full stop.

The REs could be further extended (they are passed to an awk program) if the lecturer so wishes, to specify that the numeric values are surrounded by non-numerics, for example, using the standard awk RE notation such as

```
[^0-9]18[^0-9]
```

and alternatives given as in

```
ins|inches
```

The possibility of rounding errors may necessitate the RE

```
3\.141[56]
```

to allow for either "3.1416" or "3.14159". See the original "awk" reference[1] or the relevant section in a book such as Bourne's[6] for exact details of permitted REs, and the "oracle" document for further details of the oracle program.

The marks for each dynamic test can thus be built up as a series of weighted RE scores.

If the program under test fails catastrophically at run time (core dump, or stuck in a loop) useful results may be difficult to guarantee. The system is designed to award zero for that dynamic test, and continue trying all remaining dynamic tests. The program is killed after 5 seconds of processing if it is still running, using the "~ceilidh/Tools/run" command (source in "Tools/SOURCE/run.c" if required). This is a configurable parameter; to change it, edit the defined value for "MAX" in the source in the source file "run.c" and remake.

Each dynamic test needs a file of REs for the oracle, the files being named "model.k1," "model.k2" etc for successive tests. Each test is from a file of test data, named "model.d1," "model.d2" etc, or a shell script, named "model.s1," "model.s2" etc. Other oracles can be used by assigning

```
C_ORACLE=my_oracle
```

in the exercise "type" file, which will look for a program
"~ceilidh/Tools/my_oracle" and will call it with the keywords file as
argument, and the student output as standard input. It will expect the
output of the oracle to be of the form

    Score 95

out of 100.

5.4.2. Dynamic efficiency

The C system on SUNs (not the C++ system) examines the number of times
each line of the code is executed (using tcov on the SUNs), and compares
the maximum line execution count with those for the developer's model
solution. If this test has been set up using the

    se

(set efficiency) developer's command, the maximum line execution count
on the student program is compared with that on the model program. If
the student's count is less than the model count, 100% is awarded; if
it is equal, 99% is awarded; if it is greater,

    100 * model count / student count

is awarded. See the shell script "Tools/CEff_c" for details.

5.4.3. Typographic analysis

The program "typog_c" reads C program source ("typog_C" reads C++
source) and computes various statistics associated with maintainability
and readability of the source, such as

```
Average characters per line   % blank lines
Average spaces per line
Average function length       % good function
Average identifier length     % names with good length
% define's
% number comments             % chars in comments
% indentation                 % indent errors
```

For further discussion of the typographic and complexity metrics, see
Zin and Foxley[8]

The students can obtain the system definition of features such as good
indentation from the help facility, where it is defined as

```
     Either

          the opening and closing curlies are on the same line,

     or the following three conditions must hold.

     (i)  Each closing curly bracket must relate to the line
containing
          the corresponding opening curly, and the closing curly must
be
          in a column between the first visible character on  that
line
          and the actual opening curly.

     (ii) A line containing the opening curly must not be indented
more
          than the one following it.  and

     (iii)A line containing the closing curly must not be indented
more
          than the one preceding it.

The present system also  insists  (since  it  is  recommended  in
early

lectures)  that closing curly brackets must be followed on the same
line
by a comment if more than  10  lines  after  the  corresponding
opening
curly.

Each of the general metric factors is measured, and  compared  with
set
values.   There  will  be  a range of values within which full marks
are
awarded, and a wider range within which  part  marks  will  be
awarded.
Beyond the outer range, no marks are awarded.

The marks and range parameters for each of the factors  involved
either
take  on  default  values  from  data  initialisations  built  into
the
"typog_c" etc programs (to change the  defaults,  edit  the  source
and
remake),  or  can  be  supplied  from  a file "model.tv" in the
exercise
directory using the call

     typog_c -f <exercise directory>/model.tv

to name  the  file  containing  the  typographic  weights.   A
skeleton
"model.tv" file can be generated in the current directory by calling

     typog_c -w

The developer may choose to vary these parameters to emphasize
```

different
aspects of the marking at different points in the course. Each line
of
the "model.tv" file is of the form

```
     AIDL   10    1    4   10   15 Average identifier length
     %NGL    1    0   45  100  100 % names with good length
```

where the entries are
     A 4-letter code
     The maximum mark for this feature
     The next four numbers define five ranges
       If the program value is less than the first number,
         no marks are awarded.
       If it is between the first and second numbers,
         part interpolated marks are awarded.
       If it is between the second and third numbers,
         full marks are awarded.
       If it is between the third and fourth numbers,
         part interpolated marks are awarded.
       If it is above the fourth number,
         no marks are awarded.
     A title for this metric

The developer's

```
     st
```

(set typographic) command shows the default score for the
developer's
source, and if the mark awarded is not 100% (it rarely is!)
offers
repeatedly editing of either the program source, or of a local file
of
typographic parameters.

If this is the first developer's call of "st" the system will ask for
a
total mark to be awarded, and will add an appropriate entry to
the
"mark.act" file.

To change the default values for the whole course, edit the source
in
"Tools/SOURCE/typog_c.c," where there is a table of the above
values
held in a structure array, and remake.

The student sees only the overall typographic mark. Tutors and
teachers
can see a more detailed breakdown if they wish using the "mt"
(mark
typographic) tutor command.

A call of

```
     cat source.C | typog_c -p
```

will produce a local "model.tv" file with parameters tweaked to give the
named program a typographic score of 100%.

5.4.4. Complexity analysis

This follows a similar pattern to the typographic analysis, but in this
case the metrics for the student program are not compared with absolute
values, but with those for the developer's model. The factors involved
include the following.

```
      Number of reserved words        Number of includes
      Number of gotos                 Number of conditionals
      Number of loops                 Depth of loops
      Number of operators
      Number of braces                Max depth of braces
      Number of square brackets       Max depth of square brackets
      Number of round brackets        Max depth of round brackets
      Number of function calls
      Number of numeric denotations
```

The metrics of the model program set up by the developer must first be
created in file "model.cm" in the exercise directory using

```
cat source.C | compl_c -m
```

When the student's source is analysed by

```
cat source.C | compl_c <model-metrics-file>
```

the student program metrics must be within factors for each metric set
by the developer (usually between 50% and 200%) of those of the model
solution*.

==

Note: * The developer's model program will thus always be awarded 100%
for complexity.

==

The marks and parameters for each of the factors operate in the same way
as the typog_c weights, with each line containing a maximum mark, and

four values defining the five ranges of marks awarded. The command
either uses default values built into the "complexity" program source,
or can be supplied from a file "model.cv" in the example directory
called by

```
cat source.C | compl_c -f model.cv <model-metrics-file>
```

The developer's command

    sc

(set complexity) operates in a similar way to the "set typographic" com-
mand.

If there are significantly different possible solution programs to a
given problem (e.g. a recursive and an iterative solution), it should be
possible for the developer to supply more than one model, and for the
system to compare the student program with all of these models, and to
choose the best fit. This is not yet implemented.

5.4.5. Features of the program

When assessing program sources by hand/eye, one typically looks at the
general layout (the typographic marks above), and for particular problem
dependent features such as

    occurrences of particular numeric denotations within the code which
    should really be set as constants (in C++) or "#define"s (in C);

    the use of "<= 59" where "< 60" is better practice;

    the use of 64 as a specific ASCII code rather than the correct
    character denotation 'A'.

The marking system allows the setting up of a file of REs in the exer-
cise directory, in a file "model.ft" which will then be marked using the
oracle against the program source with comments and strings removed.
This file can be set up using the "sf" (set features) developer's
option. The command performing this marking is "Tools/CFeature" shell
script.

Extra features of the oracle useful at this point are the limitations on
the number of occurrences of each RE. A line for the oracle can be of
the form

    <=1:3.141[56]

meaning that the RE after the colon must occur less than or equal

to
once. To require that a program involving the conversion factor 60
uses
that factor only once (presumably declared as a constant), and the
value
59 not at all, use the oracle file

```
==1:60

==0:59
```

Experience can be gained by looking critically by eye at student
solu-
tions to a given problem, and the features oracle can then be
improved
for future years.

5.4.6. Program structure

To look for structural weaknesses in the student's program, it can
be
re-compiled in the present C++ compiler with

```
g++ -c -Wall
```

or for C programs we can use the lint command. Certain warning
messages
(such as variables declared and not used) are picked up by an
oracle,
and a score awarded.

The developer command

```
ss
```

(set structure) allows the developer to set up such an oracle
file
"model.st" in the current exercise. If it is not set up, a
standard
file "model.st" in the course directory is used. The standard file
also
includes scaling, so that a few errors can produce a significant drop
in
the marks awarded.

The commands performing the work are "Tools/CStr_C" for C++
and
"Tools/CStr_c" for C.

5.4.7. Summary of files in the exercise directory

In a simple one module C++ programming exercise, the files which must
be
created in the exercise directory before the students access the
exer-
cise would be

```
model.q     the question
```

```
    prog83.C    the model source (no public read permission)
                  linked to "model.C"
    model.sk    the skeleton program for the user to start from
    prog83      the executable (no public read)(optional)
                  linked to "model"
    setup.act   the files to be copied before the student starts
    mark.act    the components of the marking
    model.d1    the first file of test data
    model.k1    the oracle for "model.d1" output (no public read)
    model.d2    the second file of test data (optional)
    model.k2    the oracle for "model.d2" (optional, no public read)
    model.d3    further tests and oracles as required (optional)
    model.k3
    model.dv    the dynamic test weights and titles
    model.tv    the typographic weights (optional)
    model.cm    the model complexity metrics
    model.cv    the complexity weights (optional)
    model.ft    the source code features oracle (optional, no public
    read)
    model.st    the structure oracle (optional)
```

There may be other sources, objects and header files and perhaps a "Makefile" in more complex examples.

6. Setting up a new programming exercise

The developer will normally use commands from the developer's menu to set up a new exercise in approximately the following order:

```
                    ep    edit program source
                    cm    compile program
                    run   test run program
                    ema   edit mark actions
                    sd    set up dynamic marking
                    st    set up typographic metrics
                    sc    set up complexity metrics
                    sf    set program feature oracle
                    ss    set structure marking
                    mk    for complete marking
                    eq    edit question
                    esa   edit setup actions
                    es    edit skeleton
```

With the "sd" option, typically type in order

```
   c    create new test
   d    data file rather than shell script
         then edit the data file
         observe program output
         edit oracle file
           default initial value is a copy of the program output
         observe the oracle score
   y    if oracle and data OK
         if not, alter data or oracle (you are invited to edit both)
        Then give a mark and a title for this test
   q    quit creating tests
   mk   to perform complete test marking
```

We now summarise in more detail the actions needed to set up a new exer-

cise. Suppose that the chosen exercise is a program to convert seconds
into minutes and seconds, within unit 2 of the course "pr1". All the
appropriate files could be set up by hand. We assume that the developer
exercise menu will be used.

Choose an identifier for the new exercise, such as "9".

Login as "ceilidh", and move to the defined existing course and unit and
go into "developer" mode. Type

```
sc pr1
su 2
develop
```

Then create the new exercise with "nx", which will ask for the exercise
number and title.

```
Command?     nx
Ex number?
Title?       Seconds to minutes
```

We are now in the new exercise, in developer mode. We have to set up
the following.

```
     model solution program
       source "prog29.c"
       and executable "prog29"
     mark actions
     dynamic tests
       test data
       oracles
     typographic metrics
     complexity metrics
     features metrics
     program skeleton
     setup actions
     question
```

The exact order is not vital, but the above is suggested.

Model solution program

Use "ep" to edit the program; you can edit, compile and run the program
to develop it under Ceilidh using "cm" and "run", or outside if you
wish. You would do the latter by moving to the directory

```
~ceilidh/course.pr1/unit.2/ex.9
```

or by simply obtaining a shell within Ceilidh by typing

```
!
```

The source should be in "model.C" (or "model.c" ...) linked to

"prog29.C" (or ".c" ...), and the executable should be in "model" linked
to "prog29".

The marking process

The dynamic tests are the most complex to develop. This stage requires
an executable program in the exercise directory. Type "sd" (set
dynamic), which will offer you the following sequence.

Another test?
Type "c" to create a new test. Each dynamic test requires a file
of test data (or a driving shell script), and an oracle file of
recognisers (regular expressions) for the output. The sequence of
prompts for a new test is:

Data file or shell script?
reply "d" if you wish to run the test from a data file. The
editor is then called automatically; use it to create an
appropriate data file (or shell script). The data file in
this case might be just a single integer

```
2345
```

If you chose a shell script, it might be

```
prog29 << ++++
2345
++++
```

The system then runs the program against this data file, and
its output is displayed. You are then invited to edit the
file of regular expressions (REs) for the oracle which seeks
to find out whether the output is correct. The file is ini-
tialised to a copy of the program output, so that you can sim-
ply edit out the unimportant parts. This will then be entered
into the oracle output recogniser file. It is suggested that
the file should be edited in the following stages.

For a simple test, just leave the essential output values
which should be in the output, each on a separate line.

If the number in the output should be "123", it is pru- dent to give the RE as

```
[^0-9]123[^0-9]
```

With floating point values, allowance should be made for rounding, or for different precisions. The value "1.42857" might be recognised using

```
[^0-9]1\.428[56]
```

If the output is expected to be two numbers such as "12\ 23", observe that the single expression

```
[^0-9]12  *[^0-9]23[^0-9]
```

will offer only full marks or no marks. If you use instead

```
[^0-9]12[^0-9]
[^0-9]23[^0-9]
```

the student will get half marks if only one of the two expressions is found.

For later tests, enter also recognisers for ancillary text, to recognise "12 minutes" use

```
[^0-9]12[^0-9]
[Mm]ins|[Mm]inutes
```

or a variant thereof, permitting the user to use "Mins", "minutes" etc as alternative words. To give more weight to the numeric value than to the surrounding text, use the mark specification facility of the oracle as in

```
10:[^0-9]12[^0-9]
5:[Mm]in(ute|)s
```

in which each RE is preceded by a numeric mark and a colon. It is unwise to form the complete requirement into a single RE, since if you specify, for example

```
[^0-9]12[^0-9] *[Mm]in(ute|)s
```

```
                the student obtains zero or full marks, with no
possible
                intermediate values.

          When you leave the editor after creating the oracle  RE
file,
          the  system  runs  the program against this oracle, and
prints
          the mark awarded as a percentage.

          You are then asked if you want to repeat the cycle
               edit test data
               edit oracle REs
               look at the resulting mark
          The default is to repeat the cycle if the mark  is  not
100%.
          If you indicate the end of this test, you are asked for

          Marks for this test?

               Reply with the maximum marks for this test.  It  is
con-
               venient if the total of the marks for all tests add up
to
               100.

          Title of test?

               This is the title which will appear on the student
mark-
               ing output.

          Edit model.dv file?

               This file contains the marks and titles  of  all  of
the
               tests   (following   an  initial  title  line,  which
is
               ignored).  The file can be amended if you wish to
adjust
               the marks or titles of the tests.

     Control then returns to the "Create another dynamic  test?"
query.
     Type  "q" to quit the loop.  You will then be asked for the
overall
     "dynamic" total to be entered into the "mark.act" file.

     If you re-enter the "sd" option, you can skip the earlier tests
by
     typing "s" .

Typographic marks

Use the command "st" to set the typographic marking.  If  the
immediate
result  is  not  100%,  you  may choose to stick with the system
default
```

metrics, and edit your program to improve its typographic marks relative to the built-in default values (the test shows you a breakdown of each individual metric), or set up an exercise-specific metrics "model.tv" file and insert adjusted metrics; the latter should not often be necessary. In each case the cycle

     look at metrics
     edit either the program or the weights

repeats until you are satisfied with the mark awarded.

The system then asks for a total typographic mark for the "mark.act" file.

Set complexity marks

The "sc" command to set complexity operates in a similar way to the typographic marking, except that a file storing the complexity metrics of the model program must be created. This is done each time that the "sc" command is entered.

The command then follows the pattern of the "set typographic" command.

Set features mark

If you wish, an oracle can be set up to recognise particular good or bad features which might appear in the program source for this particular problem. This facility is typically used for features such as one might look for by eye if hand marking program sources. This oracle usually requires the extra facility for specifying limits on the number of occurrences of given features.

To forbid the appearance of a number such as 59 in a problem essentially involving the value 60, use

```
==0:59
```

to specify zero occurrences of "59".

To insist that a particular constant is used only once, and is not

repeated at several points in the program, use

```
<=1:2\.54
<=1:0\.39
```

or perhaps

```
==1:2\.54|0\.39
```

to specify exactly one occurrence of either "2.54" or its inverse.

The text in comments and strings in a program are extracted before it is
passed to this oracle. See the oracle document for further examples.

This command cycles through the sequence

```
edit features oracle
look at features mark
```

until the score is 100% and/or you wish to leave. You are then asked
for a total features mark.

Complete marking

A complete marking should then be performed with the "mk" command. The
ratios of the subtotals for the separate aspects of the marking are
adjusted using the "ema" (edit marking actions) command. Check the
weights, and delete any lines corresponding to marking components which
you wish to ignore.

Setup actions

The "esa" ("edit setup actions") command allows actions which take place
when the user types "set" at the start of solving a particular exercise.
A typical setup actions file is

```
Copy model.sk prog29.C
Copy header.h
```

This causes the skeleton program to be copied to the user's area as
"prog29.C", and the header file "header.h" to be copied to the same name
in the user area. An appropriate default file will be created automati-
cally. During copying, the string "$USER" is replaced by the user's
name, and "$DATE" by the current date.

The question

Finally, the question file should be edited and checked. It is often
easy at this stage to insert an example of the model program's output
(by running the model against the first test data) and perhaps of

an
input file (by inserting the first test data). The question should
be
reasonably specific in all aspects.

Leaving the exercise mode

When you type "q" to quit the exercise mode as a developer, checks
are
made that the essential files are all there.

7. A complete example from Ceilidh

For the C++ "pr1" unit 2 exercise 5 the non-obvious files are as
fol-
lows:

Program ("model.C"):

```
#include <stream.h>

// Convert feet and inches to centimetres
// Written by EF 1988

const int INSPERFT = 12;
const float CMSPERIN = 2.54;

main ()
{
    float cms, ins;
    int   ft;

    cout << "Type feet and inches: ";
    cin >> ft >> ins;

    if ( ins < 0 || ins >= INSPERFT ) {
        cout << "Error: Inches out of range\n";
        exit( 0 );
    }

    if ( ft > 0 ) {
// Avoid explicit constants anywhere
        cms = ( ft * INSPERFT + ins ) * CMSPERIN;
    } else {
        cms = ( ft * INSPERFT - ins ) * CMSPERIN;
    }

    cout << ft << " feet "
        << ins << " inches is "
        << cms << " cms\n";
}
```

Skeleton ("model.sk"):

```
#include <stream.h>

// Convert feet and inches to centimetres
// Written by EF 1988
```

```
const int INSPERFT = 12;
const float CMSPERIN = 2.54;

main ()
{
    float cms, ins;
    int   ft;

    cout << "Type feet and inches: ";
    cin >> ft >> ins;

//////////////////////////////////////////
//
// Put your program instructions here
//
//////////////////////////////////////////

    cout << ft << " feet "
        << ins << " inches is "
        << cms << " cms\n";
}
```

Test data 1 ("model.d1"):

```
12 9.5
```

Oracle RE 1 ("model.k1"):

```
389.89
```

Test data 2 ("model.d2"):

```
-27 6.5
```

Oracle RE 2 ("model.k2"):

```
-839.470
```

Test data 2 ("model.d3"):

```
9 1.2
```

Oracle RE 2 ("model.k3"):

```
9
ft|feet
1.2
ins|inches
277.368
```

Overall marking ("model.dv"):

```
Dynamic Correctness
50 Simple test
0 Negative distance
50 Check "feet" "ins"
```

Features ("model.ft"):

```
==1:12
==1:2.54
```

We have no "model.tv" or "model.cv" files, but are using the default metrics.

Setup actions ("setup.act")

```
Copy model.sk prog25.C
```

Marking actions ("mark.act")

```
60 Dynamic
25 Typographic: typog_C -v1 $X/model.tv < prog$U$E.C
15 Complexity: compl_C -v1 -x $X/model.cm -f $X/model.cv < prog$U$E.C
Save o C
```

The complexity metrics file "model.cm" is set up by the system.

References

1. Alfred V Aho, Brian W Kernighan, and Peter J Weinberger, Awk - A Pattern Scanning and Processing Language, Bell Laboratories.

2. Steve Benford, Edmund Burke, and Eric Foxley, Student's Guide to the Ceilidh System, LTR Report, Computer Science Dept, Nottingham University, 1992.

3. Steve Benford, Edmund Burke, and Eric Foxley, Tutor's Guide to the Ceilidh System, LTR Report, Computer Science Dept, Nottingham University, 1993.

4. Steve Benford, Edmund Burke, and Eric Foxley, Teacher's Guide to the Ceilidh System, LTR Report, Computer Science Dept, Nottingham University, 1993.

5. Steve Benford, Edmund Burke, and Eric Foxley, The Design Document for Ceilidh Version 2, LTR Report, Computer Science Dept, Nottingham University, 1993.

6. Steve R Bourne, Unix System V, Addison-Wesley, 1989.

7. Eric Foxley, Question/answer exercises in Ceilidh, LTR Report, Computer Science Dept, Nottingham University, 1993.

8. Abdullah Mohd Zin and Eric Foxley, "Automatic Program Quality Assessment System", Proceedings of the IFIP Conference on Software Quality, S P University, Vidyanagar, INDIA (March 1991).

9. Abdullah Mohd Zin and Eric Foxley, The oracle program, LTR Report, Computer Science Dept, Nottingham University, 1992.

Installer's Guide to CEILIDH

S D Benford, A N Bullock, E K Burke, E Foxley, N Gutteridge, A Mohd Zin

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

Abstract

Ceilidh is an on-line coursework submission and auto-marking facility for programming courses. The automatic marking uses a comprehensive variety of static and dynamic metrics to assess the quality of submitted programs. Ceilidh also provides students with on-line access to notes, exercises and solutions, and provides tutors with extensive course monitoring and tracking facilities. Since coursework marks are involved, tight security is essential. The use of a special username together with SUID programs to control access to information is extensive.

This document acts as installation notes for the maintainer of the system. Details are given of how to install the Ceilidh system at your local site.

1. Requirements

To install and run Ceilidh as it stands the following are required

- Any recent System V or BSD system (e.g. Sun architecture running SunOS)

- For the system itself, a C compiler "cc". In the future, the system may also need a C++ compiler for some software.

- For the C++ courses, there must be a C++ compiler, AT&T C++ version 2.1 or GNU g++ version 1.39.1

Other compilers may be used for the C++ module but the given example programs may require some minor tweaking .

The software is intended to be used in a multi-user environment, where each student logs onto the system with a unique login name. The login names are used for keeping track of the student's work and marks.

2. Installing and Setting Up Ceilidh

2.1. Initial prerequisites

You will need root privileges in order to install the Ceilidh system.

It is recommended that the first thing to be done is to create a new account and associated directory called "ceilidh." A separate account

makes administration far easier and more reliable. This user should have "*" as the password entry (so that no-one can login to it directly), and a ".rhosts" file should be created with entries for each teacher who is going to maintain the system, or create work in the system. The home directory for this account should be the top level ceilidh directory after installation (i.e. the installation path ending "/ceilidh").

Ceilidh should be initially installed and tested on one machine only, and any file systems referenced should be local to that machine. The partition onto which the source is mounted needs to be suid and write-permit mounted to allow programs within Ceilidh which run SUID to write to the filestore. The "ceilidh" filestore should be integral, and not broken internally by cross-device links.

For other machines to run Ceilidh, the partition on which the source resides must be mounted with both write permission and SUID enabled.

In the following notes the expression "~ceilidh" is purely a typographic one and simply refers to the home directory of the Ceilidh source tree.

If there are compelling reasons, and only one person is going to be involved, Ceilidh can be set up under an existing username.

2.2. The Ceilidh Distribution

Ceilidh is being distributed as a collection of shar files. One of these is the Ceilidh system itself, while the others are the individual course modules for use with the system. In the current distribution there

are
five shar files. These are

file name	content	approx size
sys.sh	system	2.5Mb
x.sh	X-windows interface	0.3Mb
pr1.sh	first C++ course	2.4Mb
pr2.sh	follow-up C++	1.8Mb
c.sh	C programming	1.6Mb

These files are freely available to all academic institutions via
anonymous FTP.

==
Note: The X version is imminent, but not yet available.
==

3. Installing the Ceilidh system

You should first set up the "ceilidh" account with its associated direc-
tory, and log into it. The home directory for the system should have
the permissions "drwxr-xr-x." Move to the username and directory and
unpack the ceilidh system files using the command

```
sh sys.sh
```

This places the source, shell scripts, help files etc in various sub-
directories.

The Ceilidh home directory is referred to in the shell scripts as
"$C_HOME;" we will refer to it in this document as "~ceilidh."

The Ceilidh system is based on a collection of executable C programs and
shell scripts which are located in the top level directories "bin.mnu"
(controlling the menus), "bin.cli" (for the command line interface), and
"Tools" (for the shell scripts and programs that do the work). A
detailed summary of the names of all shell scripts and programs can be
found in the design documents "Design.*" in "~ceilidh/papers"." To
install the system, move into the "~ceilidh/Install" directory and exe-
cute the command

```
CInstSys
```

This script will ask you to enter the following site specific

```
variables
and use your answers to modify the system.

Ceilidh username

     If you are Installing Ceilidh under a username other than
"ceilidh"
     enter  that  username. The default is set to your "$USER" or
"$LOG-
     NAME" environment variable.

Ceilidh home directory path

     This is the path of the directory the system  is  stored  in.
The
     default  is  the  "Install" directory's parent directory
determined
     using the output from a "pwd" command.  You may wish to express
the
     path in a different way.

Ceilidh Manager's username

     This should reflect the user to  whom  system  comments  should
be
     emailed.   Course comments will be mailed to the first name user
in
     the course's "staff.lst" file.

Your default printer

     This represents the name of the printer to be used in all hard
copy
     operations.   You  should  change  this  to  the  name of the
local
     printer you wish to be the default when within Ceilidh.

Your print command

     This defines your local print  command.   The  default  is  set
to
     "lpr"."   If  you  use "enscript," "lp" etc, please enter that
com-
     mand.

Your default editor

     Please enter the name of the editor your students use.

Default pager

     Please enter the pager you use at your site.

The script will display the answers you have given. Type "y" if they
are
correct and the script will start installing the system.

The script will add your defaults to the system, and inserts  full
```

path
names for calls to system commands such as "awk, grep" and "mail"
by
looking through the currently set "PATH" environment variable.
The
"echo" commands are checked to see that they are in the correct
format
for your site (if not, they are changed) and the source code is
com-
piled.

For a listing of the output expected from this command see Appendix A.

Please keep us informed of any additional source changes required;
we
hope that the system is now reasonably portable.

4. Adding the X-windows Interface

This can only be done once the Ceilidh system itself has been
unpacked
and installed.

Move to "~ceilidh" and unpack the file "x.sh" by executing the
command
"/bin/sh x.sh" This will add the X-windows source into a
directory
"~ceilidh/bin.x."

The code is then installed by executing the command "CInstX" from
within
"~ceilidh/Install."

"CInstX" attempts to fully automate the installation of the
X-Interface
to Ceilidh. It does this by providing a pre-written Makefile,
tested
thoroughly on the SunOS 4.1.x operating system. However, if
this
Makefile cannot successfully compile up the X-Interface an Imakefile
is
provided to generate an alternative system dependent Makefile
tailored
to your specific system requirements. If your system architecture
is
comparatively orthogonal in nature to the SunOS suite then by
issuing
the "-i" flag to the "CInstX" command will prevent the
pre-written
Makefile from being used.

If the X-Interface fails to compile with either the pre-written
Makefile
or the imake facilities do not work or are absent. Then please give
a
concise account to where the problem lies and your specific system
con-
figuration with version numbers.

```
You need to edit the shell file "CEILIDH" to replace the value  for
the
"MANAGER,"  "C_ORACLE,"  "C_DEBUG"  and  the exact pathname for the
UNIX
commands used in the system.  You should also change the value  for
the
"FONT" by the suitable font provided by your system.  It is advisable
to
use the "fix" font to produce a better  quality  display  of  notes
and
other messages.

If the system is not installed under the "ceilidh"  username,  you
must
also change the value of "ceilidh" into the actual login name.

To run XCeilidh type "~ceilidh/bin.x/CEILIDH."

5.  Installing a Course

The Ceilidh system should now be installed and ready to  use,  but
does
not support any courses as yet.

Three courses should have  been  included  with  this  distribution
and
further  courses  will  become  available  in  the  fullness of time.
To
install a course the following steps should be taken:

o    To add the course content execute the command "/bin/sh
<course>.sh"
     .  This should create a directory "course.<coursename>" under
which
     all files relating to that course are kept.

o    Each course is installed by executing

          CInstCse <coursename>

     from within "~ceilidh/Install," for example.  The argument given
is
     the  name of the course to be installed.  This sets correct
permis-
     sions on all files within the course, and may  take  some  time
(3
     minutes).   Set  up  a file "~ceilidh/course.<course>/staff.lst"
to
     contain at least one teacher's name in the form

          username:Full name

     The first name in the file will receive the course  comments
using
     email.   All the users on this list will be able to use the
teacher
     administrative facilities on this course.
```

Carry out the above steps for each course required, and upon completion
the Ceilidh system is ready for use.

Initial tests

To run ceilidh simply execute the shell script
"~ceilidh/bin.mnu/CEILIDH." However, before trying out the system
please read the next two subsections.

6. Developers, teachers and tutors

Ceilidh supports at least four levels of access. Student users are
ordinary users of the system. Tutors have read access to various infor-
mation, and can access certain marks and submitted programs. Teachers
(teacher administrators) can perform all operations necessary for the
running of an existing course, opening and closing exercises, emailing
marks to absentees, etc. Course developers can amend exercises, and
create new ones.

The idea of having distinct teacher and tutor roles within Ceilidh stems
from the fact that, at Nottingham, whilst members of staff give the lec-
tures, the tutorial support for courses is quite often provided by post-
graduate students. Hence within Ceilidh course developers have full
access to the everything (through the use of the "ceilidh" login name),
whereas teacher administrators and tutors have limited access enforced
by stored data and SUID programs.

Two files in "~ceilidh/course.<course>" will need editing to enable the
correct operation of the system by teachers and tutors. Teachers and
Tutors in Ceilidh have special privileges and so there must be restric-
tions on the ability to play these roles. This is achieved through the
use of two files located in "~ceilidh/course.<course>" for each course.
These are "staff.lst" and "tutor.lst." The format for each of these is
one line per entry, each line being of the form

```
     username:Full name
```

eg.

```
     ef:Eric Foxley
     anb:Adrian Bullock
```

Ceilidh searches through these files for the login name of the person
currently using the system to see if they have the privileges to be a
tutor or a teacher. If someone appears in the teacher list they will
also have tutor privileges as a consequence. Access for tutors and
teachers is also restricted through the normal Unix file access permis-
sions.

It should also be noted that the "co" facility for students to send com-
ments on courses, units or exercises uses the "staff.lst" file for the
course concerned. The comment made by the student will be emailed to
the first named person in that file. It is vital that the file exist
for every course with at least one valid entry! Comments sent at the
system level (from the CEILIDH command before the user has selected a
particular course) are emailed to the system manager.

7. Aliases

There are two ways to invoke the ceilidh system. The first is to simply
type

```
     "~ceilidh/bin.mnu/CEILIDH"
```

and enter at the system level menu; all courses are now available. To
simplify access, we suggest that a hard link from
"/usr/local/bin/ceilidh" or equivalent be set up to
"~ceilidh/bin.mnu/CEILIDH" so that users have to type simply "ceilidh"
to access the system. The same thing can be done for the X-windows
interface by having a hard link from "/usr/local/bin/xceilidh" to
"~ceilidh/bin.x/CEILIDH"

Alternatively the name of a particular course can be given as an option
on the command line and Ceilidh is entered straight into the course
level, e.g. by typing

```
     "ceilidh -c pr1"
```

To make it easier for students to use the system aliases for courses

are
recommended. If the course "pr1" is running, the system alias
     "alias pr1 ceilidh -c pr1"
enables students to enter the course by simply typing "pr1."
Students
should be recommended to enter the system using such aliases as the
"-c"
flag reduces the number of processes, and hence the load on the system.

A further option supported by Ceilidh is the command line option
"-x"
which allows an initial command from a menu within Ceilidh to be
issued
from the command line. For example the command
     "ceilidh -c pr1 -x tutor"
will place the user at the tutor level menu for course "pr1"
(assuming
appropriate privileges).   Teachers and tutors may find this type
of
alias particularly useful, and may wish to use an alias for it.

The aliasing of such options by the system manager will greatly ease
the
use of Ceilidh by its general users.

To use the command line interface   to   the   system,   the
path
"~ceilidh/bin.cli" must first be added to a user's path.

8. Audit trails

The system manager now has a facility to keep audit trails of
certain
facilities.   When the system is installed, an Error audit category
will
be created. This logs any system errors that occur during the
systems
use.   This should be checked on a regular basis. Details are given
in
the developer's guide.

9. Moving the Source

If you need to move your source to another location, it is simplest
to
re-install from the original distribution. Otherwise the source can
be
moved providing that the "CInst..."   files in "~ceilidh/Install"
are
altered to show the directory being replaced.

These shell scripts automatically replace references to the
Nottingham
file store with local ones and will need altering to replace
references
to your old file store with the new local ones.

10. Appendix A

```
Expected Output from the CInstSys Command

Ceilidh Installation Script
===========================

Enter the Ceilidh username (<cr> for "ceilidh"):
No username specified....CEILIDH set to "ceilidh"

Enter the Ceilidh home directory path (<cr> for
"/ltr/user/nhg/CEILIDH2.2"):

No path specified....C_HOME set to "/ltr/user/nhg/CEILIDH2.2"

Enter the Ceilidh Manager's username: ef

Please enter your default printer: draft14

Please enter your print command (<cr> for "lpr"):
No print command specified....LPR set to "lpr"

Enter your default editor (<cr> for "vi"): emacs

Enter your default pager (<cr> for "more"):
No pager specified....PAGER set to "more"

CEILIDH set to  "ceilidh"
C_HOME  set to  "/ltr/user/nhg/CEILIDH2.2"
MANAGER set to  "ef"
PRINTER set to  "draft14"
LPR     set to  "lpr"
EDITOR  set to  "emacs"
PAGER   set to  "more"

OK ? [yn]: y

Editing "/ltr/user/nhg/CEILIDH2.2/bin.mnu/CShlVars"
550
562

Editing "/ltr/user/nhg/CEILIDH2.2/bin.mnu/CDirVars"
260
258

Editing "/ltr/user/nhg/CEILIDH2.2/bin.mnu/CShlPrcs"
850
850
Adding ceilidh to Tutor and Staff lists
Install Tools
/ltr/user/nhg/CEILIDH2.2/Tools/CCloseEx: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
/ltr/user/nhg/CEILIDH2.2/Tools/CDynCorr: Old path found

C_HOME=/ltr/user/nhg/CEILIDH2.2
/ltr/user/nhg/CEILIDH2.2/Tools/CMarkCom: Old path found
    C_HOME=/ltr/user/nhg/CEILIDH2.2
/ltr/user/nhg/CEILIDH2.2/Tools/COpenEx: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
```

```
/ltr/user/nhg/CEILIDH2.2/Tools/CProfC: Old path found
if test -z "$C_HOME" ; then C_HOME=/ltr/user/nhg/CEILIDH2.2 ; fi
/ltr/user/nhg/CEILIDH2.2/Tools/CRoffcat: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
REF=/ltr/user/nhg/CEILIDH2.2/lib/Ref
/ltr/user/nhg/CEILIDH2.2/Tools/CRoffdvi: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
REF=/ltr/user/nhg/CEILIDH2.2/lib/Ref
/ltr/user/nhg/CEILIDH2.2/Tools/CRoffohp: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
REF=/ltr/user/nhg/CEILIDH2.2/lib/Ref
/ltr/user/nhg/CEILIDH2.2/Tools/CRoffps: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
REF=/ltr/user/nhg/CEILIDH2.2/lib/Ref
/ltr/user/nhg/CEILIDH2.2/Tools/CSav_ar: Old path found
 # C_EXD=/ltr/user/nhg/CEILIDH2.2/course.pr1/unit.1/ex.1 export C_EXD
/ltr/user/nhg/CEILIDH2.2/Tools/CSav_exe: Old path found
 # C_EXD=/ltr/user/nhg/CEILIDH2.2/course.pr1/unit.1/ex.1 export C_EXD
/ltr/user/nhg/CEILIDH2.2/Tools/CTestAll: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
/ltr/user/nhg/CEILIDH2.2/Tools/CZapCse: Old path found
# C_HOME=/ltr/user/nhg/CEILIDH2.2
Check #includes
No problem with #include <time.h>
OK uid_t defined in types.h
Install Source
First some full pathnames for security
Searching for "grep", local executable is in "/bin/grep"
grep found in efread.c
         "/bin/grep
         "/bin/grep
           "/bin/grep
grep found in register.c
     "/bin/grep
     "/bin/grep
     (void) sprintf( dir, "/bin/grep
grep found in setcse.c
     "/bin/grep
grep found in vmarks.c
     "/bin/grep
Searching for "awk", local executable is in "/usr/local/bin/awk"
awk found in o-oracle.c
    (void) sprintf( commp, "/usr/local/bin/awk    '" ); skip();
awk found in oracel1.c
    (void) sprintf( commp, "/usr/local/bin/awk
'0 );
awk found in oracle.c
    (void) sprintf( commp, "/usr/local/bin/awk
'0 );
awk found in p-oracle.c
    (void) sprintf( commp, "/usr/local/bin/awk
'0 );

Searching for "nawk", local executable is in "/bin/nawk"
Searching for "ps", local executable is in "/bin/ps"
ps found in old-run.c
    (void) system( "/bin/ps " );
ps found in run.c
```

```
    (void) system( "/bin/ps " ); /* space there so that we can replace
Searching for "stty", local executable is in "/bin/stty"
stty found in qmulti.c
                 system( "/bin/stty cbreak" );
                 system( "/bin/stty -cbreak" );
Searching for "cat", local executable is in "/bin/cat"
cat found in efread.c
     (void) sprintf( file, "/bin/cat %s", argv[1] );
cat found in mmulti.c
system( "/bin/cat /tmp/oracle.ceilidh" );
cat found in qmulti.c
system( "/bin/cat /tmp/oracle.ceilidh" );
Searching for "rm", local executable is in "/bin/rm"
rm found in mmulti.c
    (void) sprintf( command, "/bin/rm -f %s", oracle );
rm found in multi.c
    (void) sprintf( command, "/bin/rm -f %s", oracle );
rm found in qmulti.c
    (void) sprintf( command, "/bin/rm -f %s", oracle );
Now replace Ceilidh pathnames by local ones
#define home   "/ltr/user/amz/ceilidh"
#define bin    "/ltr/user/amz/ceilidh/Tools"
dir.h: Path found
cc   -target sun4 -c  ccef.c
cc   -target sun4 -c  setprocs.c
cc -o ccef ccef.o setprocs.o
strip ccef
chmod 711 ccef
chmod ug+s ccef
rm -f ../ccef
ln ccef ../.
ls -l ccef
-rws--s--x  2 ceilidh     32768 Feb  1 11:20 ccef
cc   -target sun4 -c  dyncorr.c
cc -o dyncorr dyncorr.o
strip dyncorr
rm -f ../dyncorr
ln dyncorr ../.
ls -l dyncorr
-rwxr-xr-x  2 ceilidh     16384 Feb  1 11:20 dyncorr
cc -c typog.c
cc -c statdata.c
cc -c statproc.c
cc -c getwts.c
cc -c srchline.c
cc -o typog typog.o statdata.o statproc.o getwts.o  srchline.o
strip typog
rm -f ../typog ../typog_c ../typog_C
ln typog ../typog

ln typog ../typog_c
ln typog ../typog_C
ls -l typog
-rwxr-xr-x  4 ceilidh     32768 Feb  1 11:20 typog
cc -c compl.c
cc -o compl compl.o statdata.o statproc.o getwts.o  srchline.o
strip compl
rm -f ../compl ../compl_c ../compl_C
```

```
ln compl ../compl
ln compl ../compl_c
ln compl ../compl_C
ls -l compl
-rwxr-xr-x  4 ceilidh     32768 Feb  1 11:20 compl
cc   -target sun4 -c  oracle.c
cc -o oracle oracle.o
strip oracle
chmod 711 oracle
chmod ug+s oracle
rm -f ../oracle
ln oracle ../.
ls -l oracle
-rws--s--x  2 ceilidh     24576 Feb  1 11:20 oracle
cc   -target sun4 -c  run.c
cc -o run run.o
strip run
rm -f ../run
ln run ../.
ls -l run
-rwxr-xr-x  2 ceilidh     16384 Feb  1 11:20 run
cc   -target sun4 -c  mark.c
cc -o mark mark.o setprocs.o
strip mark
rm -f ../mark
ln mark ../mark
ls -l mark
-rwxr-xr-x  2 ceilidh     24576 Feb  1 11:21 mark
cc   -target sun4 -c  setup.c
cc -o setup setup.o setprocs.o
strip setup
rm -f ../setup
ln setup ../.
ls -l setup
-rwxr-xr-x  2 ceilidh     24576 Feb  1 11:21 setup
cc   -target sun4 -c  copyin.c
cc -o copyin copyin.o
strip copyin
rm -f ../copyin
ln copyin ../.
ls -l copyin
-rwxr-xr-x  2 ceilidh     16384 Feb  1 11:21 copyin
cc   -target sun4 -c  copy.c
cc -o copy copy.o setprocs.o
strip copy
rm -f ../copy

ln copy ../.
ls -l copy
-rwxr-xr-x  2 ceilidh     24576 Feb  1 11:21 copy
cc   -target sun4 -c  register.c
cc -o register register.o
strip register
chmod 711 register
chmod ug+s register
rm -f ../register
ln register ../.
ls -l register
```

```
-rws--s--x  2 ceilidh     16384 Feb  1 11:21 register
cc   -target sun4 -c  extract.c
cc -o extract extract.o
strip extract
rm -f ../extract
ln extract ../.
ls -l extract
-rwxr-xr-x  2 ceilidh     16384 Feb  1 11:21 extract
cc   -target sun4 -c  efread.c
cc -o efread efread.o
strip efread
chmod 711 efread
chmod ug+s efread
rm -f ../efread
ln efread ../.
ls -l efread
-rws--s--x  2 ceilidh     16384 Feb  1 11:21 efread
cc   -target sun4 -c  setcse.c
cc -o setcse setcse.o
strip setcse
chmod 711 setcse
chmod ug+s setcse
rm -f ../setcse
ln setcse ../.
ls -l setcse
-rws--s--x  2 ceilidh     16384 Feb  1 11:21 setcse
cc   -target sun4 -c  multi.c
cc -o multi multi.o setprocs.o
strip multi
chmod 711 multi
chmod ug+s multi
rm -f ../multi
ln multi ../.
ls -l multi
-rws--s--x  2 ceilidh     24576 Feb  1 11:21 multi
cc   -target sun4 -c  qmulti.c
cc -o qmulti qmulti.o setprocs.o
strip qmulti
chmod 711 qmulti
chmod ug+s qmulti
rm -f ../qmulti
ln qmulti ../.
ls -l qmulti

-rws--s--x  2 ceilidh     24576 Feb  1 11:21 qmulti
cc   -target sun4 -c  mmulti.c
cc -o mmulti mmulti.o setprocs.o
strip mmulti
chmod 711 mmulti
chmod ug+s mmulti
rm -f ../mmulti
ln mmulti ../.
ls -l mmulti
-rws--s--x  2 ceilidh     24576 Feb  1 11:21 mmulti
cc   -target sun4 -c  vmarks.c
cc -o vmarks vmarks.o
strip vmarks
chmod 711 vmarks
```

```
chmod ug+s vmarks
rm -f ../vmarks
ln vmarks ../.
ls -l vmarks
-rws--s--x  2 ceilidh     32768 Feb  1 11:21 vmarks
cc   -target sun4 -c  mchoice.c
cc -o mchoice mchoice.o
strip mchoice
chmod ug+s mchoice
rm -f ../mchoice
ln mchoice ../.
ls -l mchoice
-rwsr-sr-x  2 ceilidh     16384 Feb  1 11:21 mchoice
cc   -target sun4 -c  numeric.c
cc -o numeric numeric.o
strip numeric
chmod ug+s numeric
rm -f ../numeric
ln numeric ../.
ls -l numeric
-rwsr-sr-x  2 ceilidh     16384 Feb  1 11:21 numeric
cc   -target sun4 -c  ndate.c
cc -o ndate ndate.o
strip ndate
rm -f ../ndate
ln ndate ../.
ls -l ndate
-rwxr-xr-x  2 ceilidh     16384 Feb  1 11:22 ndate
cc   -target sun4 -c  audit.c
cc -o audit audit.o
strip audit
chmod 711 audit
chmod ug+s audit
rm -f ../audit
ln audit ../.
ls -l audit
-rws--s--x  2 ceilidh     16384 Feb  1 11:22 audit
cc   -target sun4 -c  qu-a-co.c
cc -o qu-a-co qu-a-co.o
strip qu-a-co

chmod 711 qu-a-co
chmod ug+s qu-a-co
rm -f ../qu-a-co
ln qu-a-co ../.
ls -l qu-a-co
-rws--s--x  2 ceilidh     16384 Feb  1 11:22 qu-a-co
cc   -target sun4 -c  difftime.c
cc -o difftime difftime.o
strip difftime
rm -f ../difftime
ln difftime ../.
ls -l difftime
-rwxr-xr-x  2 ceilidh     16384 Feb  1 11:22 difftime
Install Menu
/ltr/user/nhg/CEILIDH2.2/bin.mnu/CEILIDH: Old path found
C_HOME=/ltr/user/nhg/CEILIDH2.2
ceilidh
```

```
Whoami exists on this machine
Edit Postscript viewers out
PSVIEW=
DITVIEW=
Install CLI
C_HOME=/ltr/user/amz/ceilidh
set.env: Path found
setenv PATH /ltr/user/amz/ceilidh/bin.cli:$PATH
/ltr/user/nhg/CEILIDH2.2/source.csh: Path found
PATH=/ltr/user/amz/ceilidh/bin.cli:$PATH
/ltr/user/nhg/CEILIDH2.2/source.sh: Path found
Install Test
char *ceilidh = "/ltr/user/amz/ceilidh/bin.mnu/CEILIDH";   /* which
ceilidh */
get_script.c: Path found
char *ceilidh = "/ltr/user/amz/ceilidh/bin.mnu/CEILIDH";   /* which
ceilidh */
run_script.c: Path found
Install Echo
echo -n OK on this system, no edits needed
```

Output may differ for different reasons. If your version of the bourne shell uses does not use "echo -n" these will be replaced. On some machines various warning messages are displayed eg.

```
        cc -O -c vmarks.c
"vmarks.c", line 235: warning: trigraph sequence replaced
"vmarks.c", line 294: warning: trigraph sequence replaced
"vmarks.c", line 322: warning: trigraph sequence replaced

chmod: WARNING: Execute permission required for set-ID on execution for
mchoice
```

These do not affect the compilation of the system. If you know why these messages are produced, please let us know.

11. Appendix B

Possible Problems and Solutions

Here we hope to outline some problems which may be encountered installing and using Ceilidh and to give answers to them.

QuestionCEILIDH was unpacked and the installation appeared to go smoothly, but when I move to a course using the "sc" command the message "Failed" appears, but I am still placed at the exercise menu.

AnswerWhen you move to a course for the first time Ceilidh tries to register you for that course. To do this Ceilidh calls a SUID pro-

```
     gram called register to write your details  away  to  the
relevant
     file  in  the  Ceilidh  source.   If  you  are running Ceilidh on
a
     machine which NFS  mounts  the  partition  upon  which  the
source
     resides  then  this  partition  must  be  mounted  with suid
access
     allowed and with write permission enabled.

QuestionWhen I try to submit a comment to the system it  complains
that
     there is a comment with the wrong username.

AnswerCeilidh uses two mechanisms to check a user's identity.   For
the
     most part the value held in the $USER environment variable is
used,
     but for the more critical tests (eg. submitting work  for
marking,
     making comments) the system program "whoami"
      is used.  These two must be in agreement with each other.

QuestionI try to compile a C++ program but I am told the compiler is
not
     found.

AnswerThe compiler to be used for the  C++  course  is  defined  in
the
     "type"  files  in  each exercise.  Either replace the compiler
men-
     tioned here by a local C++ compiler, or install it.   It  is
worth
     noting  that  if  you use a different C++ compiler the example
pro-
     grams may require some alteration for them  to  compile  and
score
     good marks.

QuestionMy Bourne shell does not support functions.

AnswerReplace the reference at the beginning of all shell  scripts.
To
     do  this move to "~ceilidh/Install" and execute the "CChChell"
com-
     mand.

QuestionMy Bourne shell doesn't like functions.

AnswerFInd one that does (a System V shell?)  and change "#! /bin/sh"
in
     all  the  shell  scripts (bin.mnu/C*, Tools/C*) to "!# /bin/shV"
or
     whatever the shell's pathname is.

QuestionMy awk compiler doesn't like functions.

AnswerChange "/bin/awk" to "/bin/nawk" if you have it.
```

```
QuestionWhen I install and run the system I get the error message:
     ~ceilidh/bin.mnu/CEILIDH: syntax error at line 16:
     `^D^P(M-^@)^E(M-z)^O^?(M-^?)@^F' unexpected

AnswerIf you have files ".CEILIDH" or ".C.dir" in  your  directory,
try
     removing these.

QuestionWhat is the file "~ceilidh/Tools/SOURCE/new-run.c" for?

AnswerThis is a new improved  version  of  the  Ceilidh  tool
"run.c"."
     Unfortunately  this  program is not yet portable. If you would
like
     to try compiling it, copy it to "run.c" (after saving  the
current
     "run.c" ) and type "make."

More will surely follow in time
```

Teacher's Guide to CEILIDH

S D Benford, E K Burke, E Foxley

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

1. Introduction

This guide is to help a teacher administer a course which uses the Ceilidh system. It should be read in conjunction with the student[1] and tutor[2] guides.

The users of the CEILIDH system fall into the following classes.

```
     users          those using the system as a learning tool
     tutors         those with access to student progress monitoring
                    they essential have read access to student marks and work
     teachers       administering a course
     developers     course material creation and amendment
                    they essentially have write access to all material
     system admin   updating the system commands
```

It is to the third of these categories that this document is directed. Any user in this category will automatically have access to all the student and tutor facilities.

The support provided by the CEILIDH system falls into the four distinct levels

```
     system      department-wide view
     course      one university teaching module
     unit        documentation and exercises for one course unit
     exercises   assessment definition for one coursework
```

It is important to control access to the information provided by the system. Access is controlled by UNIX file permissions (set by the system administrator, and relevant to the course developer), and also (for teachers) by the names specified in the file ~ceilidh/course.<course>/staff.lst in the file concerned (only people whose login names occur in this file have access to teacher administration facilities for this course) and ~ceilidh/course.<course>/tutor.lst for tutor commands. Only the main Ceilidh system administrator can change the names in the staff.lst files. Any teacher named in that file can access the teacher facilities, and through these change the names in the corresponding tutor.lst file.

Filestore layout

The current filestore structure is headed by a directory ~ceilidh with
the shell scripts and programs for the commands in the directory
~ceilidh/Tools, help information in the directory ~ceilidh/help, and
research papers in the directory ~ceilidh/papers. See also the

directories ~ceilidh/bin.mnu for the shell scripts giving the course,
student, tutor etc menus, and ~ceilidh/bin.cli for the command line
interface commands. Below the ~ceilidh directory each course has a
directory such as course.pr1 for the Nottingham course "PR1" (C++ pro-
gramming in semester 1), called the "course directory". Below the
course directory are directories for each course unit (we previously
used words such as week or chapter instead of unit), typically unit.3
to contain the third unit of the "pr1" course. The unit name must be
numeric. Below this are directories for each item of coursework, such
as ex.1 for the first exercise. Exercise identifiers can be any set of
up to three characters.

The course directory contains a motd and a summary file containing a
summary of the lectures, times, courseworks set and hand-in dates; this
should be kept up-to-date by the course administrator.

The contents of the directories and the commands available at each of
these levels are described below.

2. Commands at the top level

In response to the command

```
ceilidh
```

the user is offered a standard tutor menu, see the student and tutor
guides for details. There are no special teacher commands at this
level, although the teacher has access to all tutor facilities.

3. Commands at the course and unit level

The user may either move to this level from the system level, or will
normally enter the course "PR1" for example directly, by calling

```
ceilidh -c pr1
```

either directly or using a preset alias. There will always be a currently set default unit number and exercise name set by the teacher.

If you an authorised teacher for the current course, and extra teach entry will appear on the menu as in the example

```
 Course and unit menu for course "pr1" unit "1":
 lu      list unit titles            | su    set unit code
 lx      list unit exercise titles   | sx    move to named exercise (1)
 lux     list units and exercises    state   current exercise state
 vn      view notes on the screen    pn      print notes on beth
 csum    read course summary         usum    read unit summary
 vm      view all marks
 clp     change printer              h       for more help
 co      make a comment to teacher   q       quit
 teach   teacher administration      tutor   tutor administration
 ===========================================
 Course level command:
```

|

|

The teacher administering a course should find a teach option at the foot of the course/unit menu. If this is not the case, then your name

has not been appropriately set in the course staff.lst file by the sys- tem administrator. This can be done only from within the ceilidh user- name. You can enter the teacher menu by typing teach at the course menu, or directly by typing

```
     ceilidh -c pr1 -x teach
```

as a command; an alias to this command is useful for the teacher.

The teacher menu is then displayed.

```
 Teacher's course (tst) unit (2) and exercise (1) level menu:
 ecm   edit course motd                    |  ecs   edit course summary
 ewt   edit course weights                 |  esc   edit course scale
factors
 est   edit student register               |  ety   edit course type file
 cc    close course                        |  oc    open course
 mk    email marks                         |  vm    view marks
 ss    register studs from master list     |  rs    register students by
name
 csa   add unregistered students           |  csd   delete students with
no work
 nw    set new week's defaults             |  etl   edit tutor list
 su    set unit number                     |  sx    set exercise (1)
 h     help                                |  q     return to user menu
 ===========================================
 Type teacher course command:
```

|

|

The teacher menu at this level offers facilities such as

```
edit course motd (message of the day) and summary files
edit the weights and scales files
set up student register for the course
check the register against all submitted work for the course
    report total absentees, offer to delete them from register
    report unknown submissions, offer to add them to register
edit the register
set current unit
move to named exercise in current unit
set the new week's defaults
open or close the course
view student marks
edit the files affecting mark calculations
email mark details to students and/or tutors
```

Details of these operations will now be explained further.

4. Student register

The student register for a particular course is kept in a file students in the course directory. The format of each line of the file is

```
<login>:<Full Name> (<tutor logname>):<anything>
```

Users are prompted to register the first time they use a course, although this may be made automatic later. If the user's login name does not appear in the register, the user is asked to type in the details of their full name (and perhaps tutor). It may be better to set up the register by hand before the course starts.

Other commands related to administration are:

est : edit the student register file directly

This (in common with all other "edit ... file" commands) takes a local copy of the file, allows you to edit it, and then copies the edited version back to the Ceilidh system area.

ss : register studs from master list

We keep a master file of all students, with their login name, full name, course code, tutor, modules being taken, and other information. We then add students to the register from this file by simple selection (performed using grep) command.

rs : register students by name

The system allows the teacher to give the login names, full names and tutor's login name of students to be registered on the course.

```
csa : add unregistered students

     The system searches for all files and marks submitted so  far,
and
     notes  any  login names which are not in the student register.
The
     teacher is offered these one at a time to be added to  the
student
     register.

csd : delete students with no work

     The system searches for any students who are in  the  register
but
     who have not submitted any work, and offers interactively to
delete
     them from the register.

5.  Miscellaneous

ecm : edit course message-of-the-day

     The current motd is copied to the teacher's directory.  The
teacher
     can  then  edit the file, and the amended version is copied back
to
     the Ceilidh system.  This is used for urgent messages such as
lec-
     ture re-scheduling and coursework changes.

ecs : edit course summary

     The current summary is  copied  to  the  teacher's  directory,
the
     teacher  can  edit the file, and the amended version is copied
back
     to the Ceilidh system.  We use this to keep an up-to-date
schedule
     of lectures and work.

ety : edit type file

     Each course can have a "type" file containing definitions such as
          MAXSUB=5
          MINGAP=300

          OUTOF=10

     meaning respectively

             the maximum number of submisisons for any exercise is
limited to 5
             the minimum time gap between submissions is 300 seconds
             all visible marks are to be scaled out of 10 instead of
100

     These parameters apply to the whole course,  unless  overridden
by
     information  in  an  individual  exercise's "type" file (set by
the
     developer).  Ihe MAXSUB value is negative,  say  -5,  there  is
no
     limit  on  the  number  of  submissions, but warnings will be
```

given
after the given number (in this case 5) of submissions.

oc (open course) and cc (close course)

After the use of the cc command, only tutors and teachers can log into the course. At the end of a course, it is wise both to close it, and to take an archive copy of the complete course for reference purposes.

ou (open unit) and cu (close unit)

Individual units can be opened or closed. If you prefer students not to read ahead, you can close all future units, and open them as you reach that material in lectures.

ox (open exercise) and cx (close exercise)

This is normally done during the "new week" (nw) command.

6. Tutors

etl : edit tutor list

The login and full names of all who require access to the tutor facilities should be in the file in colon separated form typified by

```
ef:Eric Foxley
sdb:Steve Benford
```

7. Marking

vm : view marks

This gives a summary of all marks. It first asks whether the marks are to be viewed, stored (in which case it asks for a filename) or printed. It then asks whether the marks requested are those for a particular student (it will request a login name), or for one exercise (the exercise currently set by su and sx), or for the whole (currently set) course.

ewt : edit course weights

The file offered by this command (the original is in a file weights in the course directory) contains for every exercise set on the course one line containing

the unit number
the exercise code

```
              the weight to be attached to this exercise mark
              an indicator 0 = open, 1 = late, 2 = closed

     The total mark awarded by the vm command for the course is then
the
     sum for each late or closed exercise of

              mark awarded x weight factor

     divided by the total of the weights.

NOTE:  It is vital that this file be kept up-to-date in order  that
the
student vm command works.

esc : edit scaling factors

     The marks from Ceilidh will be high relative to normal
examination
     results.   They  will perhaps average 95%.  The marks may
therefore
     need to be scaled down to match other  marks  from  other
courses.
     The  file offered by this command (the original is in a file
scales
     in the course directory) is of the form
                                 ___________
                                    0     0
                                   50    40
                                   70    50
                                   80    60
                                  100   100
                                 ___________
                                 |
                                            |

     The student summary output from the vm command shows the  raw
mark
     awarded  (the sum of each exercise mark times its weighting
factor)
     and the scaled mark using the above piecewise linear  scale.
With
     the file shown above,

              a mark awarded by Ceilidh of  0 would scale to  0,
              a mark awarded by Ceilidh of 50 would scale to 40,
              a mark awarded by Ceilidh of 70 would scale to 50,

     and so on, with linear interpolation between the given  points.
A
     summary at the end of the list of marks gives the marks broken
down
     by scaling interval, so that additional unnecessary entries in
the
     scales  file  can  be  used to provide a more detailed breakdown
of
     marks.

8.  Weekly administration

The recommended practice is for each exercise to be "opened" when it
is
```

```
announced  to  the class, to be made "late" at a certain handing-in
time
(submissions are still accepted, but are considered  late),  and  to
be
"closed"  (no  more  submissions  accepted) still later.  The nw
command
assists in these three operations.

nw : set up a new week

     This command asks in turn for:

     Exercises to close

          The teacher specifies the unit and exercise  numbers  for
any
          exercises  (already  late) to be closed completely.  The
ques-
          tions and solutions for these exercises are concatenated,
and
          left  in a file closed.qus in your home directory.  It is
sug-
          gested that these be made available to the students either
by
          duplication or through the filesystem.  The end of the list
of
          units and exercises to be closed is indicated by a blank
line.
          For  each  exercise closed, an appropriate one-line message
is
          appended to the motd file.   For  each  exercise  closed,
the
          teacher  is  offered  a  file  of class metrics and
plagiarism
          results.  All such computations will be forked off at the
end
          of  the  nw  command, and will hammer the machine!  The use
of
          these facilities is to be encouraged, since the teacher
should
          be aware of class overall metrics.

     Exercises to make late

          The teacher specifies the unit and exercise  numbers  for
any
          exercises  to  be  made  late.  For each exercise made late,
a
          one-line message is appended to the  motd  file.   Again,
the
          teacher  will  be  invited  to  obtain  metrics and
plagiarism
          details.

     Exercises to open

          The teacher specifies the unit and exercise  numbers  for
any
          new exercises to be opened;  an entry for each one is added
to
          the weights file.  A list of the questions  and  skeletons
is
          concatenated  in a file opened.qus in your home directory.
```

It is suggested that this be printed and photocopied to the class.

In all of the above cases, the relevant entry in the weights file is appended or amended, and a one-line entry in the motd file added.

After the nw command, you may still wish to edit the weights file with the ewt command to set the weight (third column) to be attached to that particular exercise.

The teacher is then asked for the default unit and exercise to be set, which students will default to when they log in to Ceilidh.

The above should assist in the administration of the course, and in following up student progress reports.

References

1. Steve Benford, Edmund Burke, and Eric Foxley, Student's Guide to the Ceilidh System, LTR Report, Computer Science Dept, Nottingham University, 1992.

2. Steve Benford, Edmund Burke, and Eric Foxley, Tutor's Guide to the Ceilidh System, LTR Report, Computer Science Dept, Nottingham University, 1992.

Tutor's Guide to CEILIDH

S D Benford, E K Burke, E Foxley, N Gutteridge, A M Zin

ltr @ cs.nott.ac.uk
Learning Technology Research
Computer Science Department
University of Nottingham
NOTTINGHAM NG7 2RD, UK

Introduction

Ceilidh is an on-line coursework administration and auto-marking facility designed to help both students and staff with programming courses. It helps students by informing them of the coursework required of them, and by permitting them to submit their work on the computer, instead of having to print things out and hand them in. It also marks programs directly, and informs the student and teacher of the mark awarded. The marking uses a comprehensive variety of static and dynamic metrics to assess the quality of submitted programs. Ceilidh also provides students with on-line access to notes, examples and solutions, and provides tutors with extensive course monitoring and tracking facilities.

This document is a guide for tutor users of the Ceilidh system. For more details of the user view of the system please see the student guide.

Overview of Ceilidh

The Ceilidh system acts in a number of ways for students, tutors and teachers, and can support a variety of different courses.

There are facilities for students (reading notes and coursework definitions, looking at examples, developing programs, marking programs, submitting work), and tutors (observing submitted work and marks, checking for plagiarism) and for teachers (amending course material, setting up exercises). The appropriate facilities are offered to appropriate users by the Ceilidh system itself, which takes note of the login identification of the user and checks this against lists stored in the system. To obtain access to the tutor facilities described below, your name must have been added to the appropriate lists by the Ceilidh system administrator.

Using Ceilidh as a Tutor

Upon issuing the command ceilidh you will be greeted with the menu shown
in figure 1. -

===
Note: - Example menus are shown in this document. Menus seen in prac-
tice may vary slightly from those shown, since the actual menu you are
offered reflects only those facilities available at the time.
===

```
CEILIDH system - Type
 lc    list course titles          | sc   move to named course
 vp    view papers                 | pp   print papers
 clp   change printer              | h    for more help
 co    make a comment to teacher   | q    quit this session
 fs    find student                | ft   find tutees
===========================================
Additional tutor menu
 ss     summarise one student
===========================================
System level command:
```

|

|

Figure 1 : System Level Ceilidh Menu

At this level tutors are offered the following additional option

ss
This command summarizes work completed by a student on any Ceilidh
courses they are registered for, thus enabling the tutor to follow
the work completed by a student on any of the courses supported by
the system. A sample of the output from this command can be seen
below

```
===== nhg not taking course pas =====

===================================
Course pr1

Work for course pr1 unit 1 ex 1
Mark:   date 92.10.23;  time 17.21.57;  entered by nhg; mark 94:
Mark:   date 92.10.26;  time 07.46.50;  entered by nhg; mark 94:
Mark:   date 92.10.26;  time 07.52.18;  entered by nhg; mark 89:
Mark:   date 92.10.26;  time 07.58.08;  entered by nhg; mark 98:
```

```
          Mark:   date 93.01.22;  time 15.27.38;  entered by nhg; mark
100:
          Soln:   size 142;       date Jan 22 15:27
```

If the tutor then selects a particular course (using the sc command) the
menu shown in figure 2 is displayed.

```
 Course and unit menu for course "pr1" unit "1"
  lu     list unit titles            | su     set unit code
  lx     list unit exercise titles   | sx     move to named exercise (1)
  lux    list units and exercises    | state  current exercise state
  vn     view notes on the screen    | pn     print notes on letter13
  csum   read course summary         | usum   read unit summary
  vm     view all marks              |
  clp    change printer              | h      for more help
  co     make a comment to teacher   | q      quit
  tutor  tutor administration
 ============================================
 Unit command:
```

|

|

Figure 2 : Course and Unit Level Ceilidh Menu

At this level the tutor can access further student monitoring facilities
by typing the command
     tutor

The menu of figure 3 will then be displayed.

```
 Tutor's course (pr1) unit (2) ex (5) menu
  vr    view course register            | pr   print register
  fs    find named student entry        | ft   find tutees' register
entry
  rs    register student
  ss    summarise named student's work  | st   summ tutees' work in
this ex
  vs    view named student's work       | vt   view tutees' work in
this ex
  mi    search for missing students     | uk   search for unknown
students
  sv    save missing etc lists          | vm   view mark summaries
  plag  search for plagiarism (slow)    | met  overall exercise metrics
  su    set unit                        | sx   set exercise
  mt    mark typographic                | mc   mark complexity
  md    mark dynamic                    | mf   mark features
  em    enter marks                     | vo   view oracle files in
this ex
  h     help                            | q    return to calling menu
 ============================================
 Type tutor command:
```

|

|

Figure 3 : Additional Tutor Menu

This menu gives the tutor access to commands which

- o Find student's registered on the current course
- o Allow tutee's work to be monitored
- o Help determine areas of weakness in a tutee's solution program
- o Examine the metrics used to mark student's programs
- o Identify student's who have not submitted work for an exercise
- o Identify student's who have submitted work and are not registered for the course

A summary of these commands is as follows

su
This command allows you to change the currently set unit and

sx
allows you to change the currently set exercise. There is no change of menu after this command.

vr
This allows the tutor to view the course register for the current course. If a paper copy is needed, the pr command can be used to send the register to a printer. Individual students registered on

the course can be found using the

fs
command. This looks for a student in the course register using a name or substring. If the first letter of the substring is a capital letter, the system assumes that this is a student's name and will look for a student whose name contains this substring. If the first letter is lower case, the system assumes that the string is a username.

ft
This is similar to the previous command except that it asks for a tutor's login name and returns the names of his/her tutees.

rs
This command can be used to register students on a course. It asks for a student's login name, full name and tutor's login name.

Once
all students have been entered, type q to quit.

ss
This command summarises a named student's work. It first prints one line for every attempt at every exercise, each line giving the unit and exercise number, the date and time of the attempt, whether it was late, the mark awarded, and whether it was marked by the student or the teacher. Then for each of the formally set exercises, it lists the unit and exercise number, the mark awarded (starred if submitted late), the class average mark, and the weight to be awarded to that exercise in the final total. The summary line at the end gives the weighted total mark, the scaled mark (using scaling factors set by the teacher), the number of exercises submitted out of the total possible (and the number of late submissions in parentheses), and the total number of attempts. An example of the final summary can be seen below.

```
       nhg Student Smith, Fred
     Unit  1, exercise   1: mark 100 , avge  93, weight  1.0
     Unit  1, exercise   2: mark 100*, avge  96, weight  1.0
     Unit  2, exercise   5: mark  96 , avge  95, weight  2.0
     Unit  2, exercise   6: mark  97 , avge  95, weight  2.0
     Unit  2, exercise   7: mark  97 , avge  91, weight  3.0
     Unit  3, exercise   2: mark  96 , avge  90, weight  4.0
     Unit  3, exercise   6: mark 100 , avge  96, weight  3.0
     Unit  3, exercise   7: mark 100 , avge  94, weight  4.0
     Unit  4, exercise   3: mark  99 , avge  90, weight  4.0
     Unit  4, exercise   4: mark  96 , avge  95, weight  4.0
     Unit  4, exercise   5: mark  99*, avge  87, weight  6.0
     Unit  5, exercise   1: mark 100*, avge  96, weight  5.0
     Unit  5, exercise   2: mark  99 , avge  94, weight  6.0
     Unit  5, exercise   5: mark  96 , avge  92, weight  5.0
     Unit  6, exercise   1: mark 100 , avge  94, weight  6.0
     Unit  6, exercise   4: mark 100 , avge  94, weight  6.0
     Unit  6, exercise   6: mark 100 , avge  94, weight 10.0
     Unit  7, exercise   1: mark 100*, avge  94, weight 10.0
     Unit  7, exercise   5: mark 100*, avge  94, weight  9.0
     Unit  7, exercise   6: mark 100*, avge  95, weight  7.0

     Unit  8, exercise   2: mark 100 , avge  88, weight 11.0
     Unit  8, exercise   3: mark  99*, avge  81, weight 20.0
     nhg   Smith, Fred             Mark 99.1, Scaled  95,  Sub'd
22( 7)22, Att's 136
```

Marks marked with an asterisk were late; on the last summary line, the number in parentheses is the number of exercises submitted late.

```
st
     This is similar to the above, except that it summarises the work
of
     all of a specified tutor's tutees.

vs
     Allows the tutor to examine every piece of coursework submitted
by
     a named student.  The student solution to each exercise is shown
in
     turn.  After each exercise, you can continue to view the next
exer-
     cise solution or quit.

vt
     Allows the tutor to examine work by each of his/her tutees  in
the
     currently set unit and exercise.

mi
     Lists the students who have not  submitted  work  for  the
current
     exercise.   A  list  is  printed  of login names of students in
the
     register (the file students in the course directory) but  who
have
     not submitted work.

uk
     Looks for students who have submitted work yet are  not
registered
     on the course (i.e. do not appear in the students file).  The
above
     lists can be saved for later processing using the sv command.

vm
     This command allows you to view or print mark summaries for a
stu-
     dent,  the  current  exercise or the current course.  It first
asks
     whether the marks are to be viewed on the screen, stored (in
which
     case  it asks for a filename) or printed.  It then asks whether
the
     marks requested are those for a particular student (it will
request
     a  login  name), or for one exercise (the exercise currently set
by
     su and sx ), or for the whole (currently set) course.

plag
     This command will carry out a  search  for  plagiarism  within
the
     current exercise.  It is very slow and machine demanding.  It
looks
     for pairs of similar solution programs  among  those  submitted
so
     far.   It lists pairs of login names in order with the most
similar
     pair first.  The dates and sizes of the programs  are  also
```

shown.

Typical out is as follows.

```
CEILIDH system plagiarism check: course pr1 unit 8 ex 3

Fri Jun 11 09:21:01 BST 1993 ,       94 files

***** Difference 1 *****

kjc   Dave Prentice (oklee)     6863 Jan 24 15:32 kjc.C
ktc   Steve Earwicker (prs)     6748 Jan 20 13:00 ktc.C

nhx   Lucy Barwell (oklee)     6855 Jan  7 17:10 nhx.C
bmz   Jane Wilkinson (oklee)     6884 Jan 20 10:51 bmz.C

wcw   Les Shaw (msg)     6833 Jan 24 15:26 wcw.C
ktc   Steve Earwicker (prs)     6748 Jan 20 13:00 ktc.C

***** Difference 2 *****

kjc   Dave Prentice (oklee)     6863 Jan 24 15:32 kjc.C
wcw   Les Shaw (msg)     6833 Jan 24 15:26 wcw.C

***** Difference 4 *****

bmz   Jane Wilkinson (oklee)     6884 Jan 20 10:51 bmz.C
kjc   Dave Prentice (oklee)     6863 Jan 24 15:32 kjc.C

bmz   Jane Wilkinson (oklee)     6884 Jan 20 10:51 bmz.C
ktc   Steve Earwicker (prs)     6748 Jan 20 13:00 ktc.C

bmz   Jane Wilkinson (oklee)     6884 Jan 20 10:51 bmz.C
wcw   Les Shaw (msg)     6833 Jan 24 15:26 wcw.C

***** Difference 5 *****

nhx   Lucy Barwell (oklee)     6855 Jan  7 17:10 nhx.C
kjc   Dave Prentice (oklee)     6863 Jan 24 15:32 kjc.C

nhx   Lucy Barwell (oklee)     6855 Jan  7 17:10 nhx.C
wcw   Les Shaw (msg)     6833 Jan 24 15:26 wcw.C

nlx   Jean Best (mjg)     5018 Jan 25 12:09 nlx.C
cla   Claire L Aspell (sdb)     5317 Jan 20 16:22 cla.C

***** Difference 6 *****

mcs   Mark C Willoughby (pmc/ANO)     4692 Jan 23 14:54 mcs.C
pdr   Paul D Dimbleby (pmc/ANO)     4412 Jan 25 23:02 pdr.C

***** Difference 13 *****

akx   Naji Kumar (pmc/rbh)     2753 Jan 25 11:14 akx.C
pmj   Peter M Jamithorp (leon)     2578 Jan 22 21:05 pmj.C

***** Difference 14 *****

tsx   Tom J Johnson (pmc/ANO)     3325 Jan 25 11:24 tsx.C
pmj   Peter M Jamithorp (leon)     2578 Jan 22 21:05 pmj.C

***** Difference 15 *****
```

```
          akx   Arun Kumar (pmc/rbh)     2753 Jan 25 11:14 akx.C

          tsx   Tom J Johnson (pmc/ANO)     3325 Jan 25 11:24 tsx.C

          ***** Difference 16 *****

          tsx   Tom J Johnson (pmc/ANO)     3325 Jan 25 11:24 tsx.C
          agx   Andrew Bramman (mfd)     2478 Jan 22 14:00 agx.C
```

met
     This produces overall metrics for work submitted for the
current
     exercise.   This is useful in determining in which areas the
class
     is weak. See the mt command below for a description of the
output.

The tutor has the ability to examine the details of marking in
various
areas of a student's program (which must be stored in a file such
as
prog21.C in the current directory) in more detail using the
following
commands.   Students should NOT be encouraged to look at these
detailed
breakdowns, as that will encourage them to tweak results.

mt
     to examine typographic marks in more detail. The result may be

```
          Typographic Analysis
                                  factor:value: mark:out of: lost
             Average characters per line: 17.0: 10.0:  10  :  0.0
                           % blank lines: 23.1: 10.0:  10  :  0.0
                 Average spaces per line:  2.0: 10.0:  10  :  0.0
               Average identifier length:  5.2: 10.0:  10  :  0.0
                % names with good length: 84.6:  5.0:   5  :  0.0
                     % lines as comments: 26.9: 10.0:  10  :  0.0
                     % chars in comments: 27.2: 10.0:  10  :  0.0
                           % indentation: 14.4: 10.0:  10  :  0.0
                         % indent errors:  0.0: -0.0: -10  :  0.0
                      % [] indent errors:  0.0: -0.0: -10  :  0.0
                      % () indent errors:  0.0: -0.0: -10  :  0.0
                         % uncommented }:  0.0: -0.0: -10  :  0.0
          Score for Typographic Analysis is::100.0%
```

     For each significant typographic factor, the columns show the
value
     achieved by the program, the mark awarded, the maximum mark,
and
     the number of marks lost. The last column is the most
useful.
     Metrics marked out of zero are not included.

mc
     This examines the complexity metrics in a similar way.

md
     This shows the details of the dynamic marking. Each dynamic
test
     is performed in turn, and the output inspected by an oracle.
You

should also read the Ceilidh oracle document. Output might be

```
Test 1 : radius
Test 2 : 30
Test 3 : area
Test 4 : 2827.43
test min max cnt mrk oof cum oof lost

  1   1   1    1  10  10  10  10   0
  2   1   1    1  10  10  20  20   0
  3   1   1    1  10  10  30  30   0
  4   1   1    1  10  10  40  40   0
Awarded 40 marks out of max 40 marks
100
Test 1 : [Nn]egative
Test 2 : [Nn]ot
test min max cnt mrk oof cum oof lost
  1   1   1    0   0  10   0  10  10
  2   1   1    1  10  10  10  20   0
Awarded 10 marks out of max 20 marks
50
```

This shows two tests. For each test, first the oracle expressions are shown. Then a table shown what was found, and the marks awarded. The columns in the tables include minimum and maximum number of times the expression should be found, the actual number of occurrences found ("cnt"), the mark awarded ("mrk") out of a maximum ("oof"), cumulative marks so far ("cum") out of cumulative total ("oof"), and the marks lost on this expression. Finally an overall percentage mark for the dynamic test is shown.

The first test here involved searching for the strings "radius", "30", "area", and "2827.43", and all were found, 100% awarded. The second test looked for the string "[Nn]egative" (not found, 10 marks lost), and then "[Nn]ot" (found), 50% awarded.

mf

This performs the "features" marking with the oracle. The oracle output is as described above for the dynamic tests. The "features" oracles are items which the teacher has prescribed relevant to the student's program source. An example is

```
Test 1 : define.*3.14159
Test 2 : =1:3.14159
Test 3 : >3:radius
test min max cnt mrk oof cum oof lost
  1   1  1    1  10  10  10  10   0
  2   1  1    1  10  10  20  20   0
  3   4  4    5  10  10  30  30   0
Awarded 30 marks out of max 30 marks
```

Score 100

The teacher expected the string "define.*3.14159" to occur at least once, the string "3.14159" to occur exactly once, and the string "radius" to occur at least three times.

em

For hand marked examples, this command allows the tutor to enter marks by hand. It assumes the current course and unit, and asks for an exercise number. If the exercise already exists, marks will be appended to the existing marks file. If not, the named new exercise is set up. You can then enter marks in three ways.

- o You type a name, the computer checks for uniqueness, you then type the mark.
- o The computer lists each name from the register in register order.
- o The computer lists just your tutee's names.

vo

Allows a tutor to look at all the oracle files from which the student solutions are marked. You will be shown the oracles for checking the dynamic test output, and the "features" oracle if it exists. To understand these, you will need to look at the Oracle document. See the md (mark dynamic) and mf (mark features) commands above.

h

Gives general help on tutor commands.

q

Returns the user to the previous menu (typically the course level student menu).

Compiler Options

```
-c        Suppress linking with ld(1) and produce a .o  file
          for each source file.  A single object file can be
          named explicitly using the -o option.

-C        Prevent the C preprocessor from  removing
          comments.

-E        Run the source file through  the  C  preprocessor,
          only.  Sends the output to the standard output, or
          to a file named with the -o option.  Includes  the
          cpp line numbering information.  (See also, the -P
          option.)

-g        Produce additional symbol  table  information  for
          dbx(1) and dbxtool(1).  When this option is given,
          the -O and -R options are suppressed.

-help     Display helpful information about compiler.

-Ipathname
          Add pathname to the list of directories  in  which
          to   search   for #include  files  with  relative
          filenames  (not  beginning  with  slash  /).   The
          preprocessor  first searches for #include files in
          the  directory  containing  sourcefile,  then   in
          directories  named  with  -I options (if any), and
          finally, in /usr/include.

-llibrary Link with  object  library  library  (for  ld(1)).
          This option must follow the sourcefile arguments.

-Ldirectory
          Add directory to the list of directories  contain-
          ing  object-library  routines  (for  linking using
          ld(1).

-M        Run only the macro preprocessor  on  the  named  C
          programs,  requesting  that  it  generate makefile
          dependencies and send the result to  the  standard
```

```
                 output  (see  make(1)  for details about makefiles
                 and dependencies).


         -o outputfile
                 Name the output file outputfile.  outputfile  must
                 have  the  appropriate suffix for the type of file
                 to be produced  by  the  compilation  (see  FILES,
                 below).  outputfile  cannot be the same as source-
                 file (the compiler will not overwrite  the  source
                 file).

         -O[level] Optimize the object code.  Ignored when either  -g
                 or  -a  is  used.   -O  with  the level omitted is
                 equivalent to -O2.  level is one of:

                      1    Do postpass assembly-level  optimization
                           only.

                      2    Do global  optimization  prior  to  code
                           generation,   including  loop  optimiza-
                           tions, common subexpression elimination,
                           copy propagation, and automatic register
                           allocation. -O2 does not optimize refer-
                           ences  to  or definitions of external or
                           indirect variables.


                 If the optimizer runs out of memory, it  tries  to
                 recover  by  retrying  the  current procedure at a
                 lower level of optimization and resumes subsequent
                 procedures at the original level.

          -P        Run the source file through  the  C  preprocessor,
                 only.  Puts the output in a file with a .i suffix.
                 Does not include cpp-type line number  information
                 in the output
```

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Buffer Manipulation

#include <memory.h>

void *memchr (void *s, int c, size_t n) - Search for a character in a buffer.

int memcmp (void *s1, void *s2, size_t n) - Compare two buffers.

void *memcpy (void *dest, void *src, size_t n) - Copy one buffer into another .

void *memmove (void *dest, void *src, size_t n) - Move a number of bytes from one buffer lo another.

void *memset (void *s, int c, size_t n) - Set all bytes of a buffer to a given character.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Character Classification and Conversion

#include <ctype.h>

int isalnum(int c) - True if c is alphanumeric.

int isalpha(int c) - True if c is a letter.

int isascii(int c) - True if c is ASCII .

int iscntrl(int c) - True if c is a control character.

int isdigit(int c) - True if c is a decimal digit.

int isgraph(int c) - True if c is a graphical character.

int islower(int c) - True if c is a lowercase letter.

int isprint(int c) - True if c is a printable character.

int ispunct (int c) - True if c is a punctuation character.

int isspace(int c) - True if c is a space character.

int isupper(int c) - True if c is an uppercase letter.

int isxdigit(int c) - True if c is a hexadecimal digit.

int toascii(int c) - Convert c to ASCII .

tolower(int c) - Convert c to lowercase.

int toupper(int c) - Convert c to uppercase.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Directory Manipulation **Up:** C Standard Library Functions **Previous:** Character Classification and Conversion

Data Conversion

#include <stdlib.h>

double atof(char *string) - Convert string to floating point value.

int atoi(char *string) - Convert string to an integer value.

int atol(char *string) - Convert string to a long integer value.

char *itoa(int value, char *string, int radix) - Convert an integer value to a string using given radix.

char *ltoa(long value, char *string, int radix) - Convert long integer to string in a given radix.

double strtod(char *string, char *endptr) - Convert string to a floating point value.

long strtol(char *string, char *endptr, int radix) - Convert string to a long integer using a given radix.

unsigned long strtoul(char *string, char *endptr, int radix) - Convert string to unsigned long.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: File Manipulation **Up:** C Standard Library Functions **Previous:** Data Conversion

Directory Manipulation

#include <dir.h>

int chdir(char *path) - Change current directory to given path.

char *getcwd(char *path, int numchars) - Returns name of current working directory.

int mkdir(char *path) - Create a directory u sing given `path` name.

int rmdir(char *path) - Delete a specified directory.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Input and Output **Up:** C Standard Library Functions **Previous:** Directory Manipulation

File Manipulation

#include <sys/stat.h> and #include <sys/types.h>

int chmod(char *path, int pmode) - Change permission settings of a file.

int fstat(int handle, struct stat *buffer) - Get file status information.

int remove(char *path) - Delete a named file.

int rename(char *oldname, char *newname) - rename a file.

int stat(char *path, struct stat *buffer) - Get file status information of named file.

unsigned umask(unsigned pmode) - Set file permission mask.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Stream 1/0 **Up:** C Standard Library Functions **Previous:** File Manipulation

Input and Output

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Stream 1/0

#include <stdio.h>

void clearerr(FlLE *file_pointer) - Clear error indicator of stream,

int fclose(FlLE *file_pointer) - Close a file,

int feof(FlLE *file_pointer) - Check if end of file occurred on a stream.

int ferror(FlLE *file_pointer) - Check if any error occurred during file I/0.

int fflush(FlLE *file_pointer) - Write out (flush) buffer to file.

int fgetc(FlLE *file_pointer) - Get a character from a stream.

int fgetpos(FlLE *file_pointer, fpos_t current_pos) - Get the current position in a stream.

char *fgets(char *string, int maxchar, FILE *file_pointer) - Read a string from a file.

FILE *fopen(char *filename, char *access_mode) - Open a file for buffered I/0.

int fprintf(FlLE *file_pointer, char *format_string, args) - Write formatted output to a file,

int fputc(int c, FILE *file_pointer) - Write a character to a stream.

int fputchar(int c) - Write a character to `stdout`.

int fputs(char *string, FILE *file_pointer) - Write a string to a stream.

size_t fread(char *buffer, size_t size size_t count, FILE *file_pointer) - Read unformatted data from a stream into a buffer.

FILE *freopen(char *filename, char *access mode, FILE *file_pointer) - Reassign a file pointer to a different file.

int fscanf(FlLE *file_pointer, char *format string, args) - Read formatted input from a `stream`.

int fseek(FlLE *file_pointer, long offset, int origin) - Set current position in file to a new location.

int fsetpos(FlLE *file pointer, fpos_t *current pos) - Set current position in file to a new location.

long ftell(FlLE *file_pointer) - Get current location in file.

size_t fwrite(char *buffer, size_t size, size_t count FILE *file_pointer) - Write unformatted data from a buffer to a stream.

int getc(FlLE *file_pointer) - Read a character from a `stream`.

int getchar(void) - Read a character from `stdin`.

char *gets(char *buffer) - Read a line from stdin into a buffer.

int printf(char *format _string, args) - Write formatted output to `stdout`.

int putc(int c, FILE *file_pointer) - Write a character to a `stream`.

int putchar(int c) - Write a character to `stdout`.

int puts(char *string) - Write a string to `stdout`.

void rewind(FlLE *file_pointer) - Rewind a file.

int scanf(char *format_string, args) - Read formatted input from stdin.

void setbuf(FlLE *file_pointer, char *buffer) - Set up a new buffer for the stream.

int setvbuf(FlLE *file_pointer, char *buffer, int buf_type, size_t buf size) - Set up new buffer and control the level of buffering on a stream.

int sprintf(char *string, char *format_string, args) - Write formatted output to a `string`.

int sscanf(char *buffer, char *format_string, args) - Read formatted input from a `string`.

FILE *tmpfile(void) - Open a temporary file.

char *tmpnam(char *file_name) - Get temporary file name.

int ungetc(int c, FILE *file_pointer) - Push back character into `stream`' s buffer

---

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Mathematics **Up:** Input and Output **Previous:** Stream 1/0

Low level I/O

#include <stdio.h> and may also need some of #include <stdarg.h>, #include <sys/types.h>, #include <sys/stat.h>, #include <fcntl.h>.

int close (int handle) - Close a file opened for unbuffered I/O.

int creat(char *filename, int pmode) - Create a new file with specified permission setting.

int eof (int handle) - Check for end of file.

long lseek(int handle, long offset, int origin) - Go to a specific position in a file.

int open(char *filename, int oflag, unsigned pmode) - Open a file for low-level I/O.

int read(int handle, char *buffer, unsigned length) - Read binary data from a file into a buffer.

int Write(int handle, char *buffer, unsigned count) - Write binary data from a buffer to a file.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Memory Allocation **Up:** C Standard Library Functions **Previous:** Low level I/O

Mathematics

#include <math.h>

int abs (int n) - Get absolute value of an integer.

double acos(double x) - Compute arc cosine of x.

double asin(double x) - Compute arc sine of x.

double atan(double x) - Compute arc tangent of x.

double atan2(double y, double x) - Compute arc tangent of y/x.

double ceil(double x) - Get smallest integral value that exceeds x.

double cos(double x) - Compute cosine of angle in radians.

double cosh(double x) - Compute the hyperbolic cosine of x.

div_t div(int number, int denom) - Divide one integer by another.

double exp(double x - Compute exponential of x.

double fabs (double x) - Compute absolute value of x.

double floor(double x) - Get largest integral value less than x.

double fmod(double x, double y) - Divide x by y with integral quotient and return remainder.

double frexp(double x, int *expptr) - Breaks down x into mantissa and exponent of no.

labs(long n) - Find absolute v alue of long integer n.

double ldexp(double x, int exp) - Reconstructs x out of mantissa and exponent of two.

ldiv_t ldiv(long number, long denom) - Divide one long integer by another.

double log(double x) - Compute log(x).

double log10 (double x) - Compute log to the base 10 of x.

double modf(double x, double *intptr) - Breaks x into fractional and integer parts.

double pow (double x, double y) - Compute x raised to the power y.

int rand (void) - Get a random integer between 0 and 32.

int random(int max_num) - Get a random integer between 0 and max_num.

void randomize(void) - Set a random seed for the random number generator.

double sin(double x) - Compute sine of angle in radians.

double sinh(double x) - Compute the hyperbolic sine of x.

double sqrt(double x) - Compute the square root of x.

void srand(unsigned seed) - Set a new seed for the random number generator (rand).

double tan(double x) - Compute tangent of angle in radians.

double tanh(double x) - Compute the hyperbolic tangent of x.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Process Control **Up:** C Standard Library Functions **Previous:** Mathematics

Memory Allocation

#include <malloc.h>

void *calloc(size_t num elems, size_t elem_size) - Allocate an array and initialise all elements to zero .

void free(void *mem address) - Free a block of memory.

void *malloc(size_t num bytes) - Allocate a block of memory.

void *realloc(void *mem address, size_t news i ze) - Reallocate (adjust size) a block of memory.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Searching and Sorting **Up:** C Standard Library Functions **Previous:** Memory Allocation

Process Control

include <stdlib.h>

void abort(void) - Abort a process.

int execl(char *path, char *argO, char *arg1,..., NULL) - Launch a child process (pass command line).

int execlp(char *path, char *argO, char *arg1,..., NULL) - Launch child (use PATH pass command line).

int execv(char *path, char *argv[]) - Launch child (pass argument vector).

int execvp(char *path, char *argv[]) - Launch child (use PATH, pass argument vector).

void exit(int status) - Terminate process after flushing all buffers.

char *getenv(char *varname) - Get definition of environment variable,

void perror(char *string) - Print error message corresponding to last system error.

int putenv(char *envstring) - Insert new definition into environment table.

int raise(int signum) - Generate a C signal (exception).

void (*signat(int signum, void(*func)(jnt signum [, int subcode])))(int signum) - Establish a signal handler for signal number `signum`.

int system(char *string) - Execute a UNIX (or resident operating system) command.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: String Manipulation **Up:** C Standard Library Functions **Previous:** Process Control

Searching and Sorting

#include <stdlib.h>

void *bsearch(void *key, void *base, size_t num, size_t width, int (*compare)(void *elem1, void *elem2)) - Perform binary search.

void qsort(void *base, size_t num, size_t width, int (*compare)(void *elem1, void *elem2)) - Use the quicksort algorithm to sort an array.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Advantages of using UNIX with C

- **Portability** - UNIX, or a variety of UNIX, is available on many machines. Programs written in *standard* UNIX and C should run on any of them with little difficulty.
- **Multiuser / Multitasking** - many programs can share a machines processing power.
- **File handling** - hierarchical file system with many file handling routines.
- **Shell Programming** - UNIX provides a powerful command interpreter that understands over 200 commands and can also run UNIX and user-defined programs.
- **Pipe** - where the output of one program can be made the input of another. This can done from command line or within a C program.
- **UNIX utilities** - there over 200 utilities that let you accomplish many routines without writing new programs. *e.g.* make, grep, diff, awk, more ....
- **System calls** - UNIX has about 60 system calls that are at the *heart* of the operating system or the *kernel* of UNIX. The calls are actually written in C. All of them can be accessed from C programs. Basic I/0, system clock access are examples. The function `open()` is an example of a system call.
- **Library functions** - additions to the operating system.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next | Up | Previous
Next: File and Directory Manipulation **Up:** UNIX and C **Previous:** Advantages of using UNIX with C

Using UNIX System Calls and Library Functions

To use system calls and library functions in a C program we simply call the appropriate C function (Appendix).

We have already met some system calls when dealing with low level I O - `open(), creat(), read(), write() and close()` are examples.

Examples of standard library functions we have met include the higher level I/O functions - `fopen(), fprintf()`, `sprintf()`, `malloc()` ...

All math functions such as `sin(), cos(), sqrt()` and random number generators - `random(), seed(), lrand48(), drand48()` *etc.* are standard math library functions.

NOTE: most standard library functions will use system calls within them.

For most system calls and library functions we have to include an appropriate header file. *e.g.* `stdio.h, math.h`

Information on nearly all system calls and library functions is available in manual pages. These are available on line: Simply type `man` function name.

e.g. ~`man drand48`

would give information about this random number generator.

All system calls and library functions have been listed in a previous handout.

We have already seen examples of string handling library functions. For the rest of this course we will study the application of a few more system and library functions.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Running Make

Simply type `make` from command line.

UNIX automatically looks for a file called `Makefile` (note: capital M rest lower case letters).

So if we have a file called `Makefile` and we type make from command line. The `Makefile` in our current directory will get executed.

We can override this search for a file by typing `make -f make_filename`

e.g. ~ `make -f my_make`

There are a few more `-options` for makefiles - see manual pages.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Header files **Up:** Programming in C **Previous:** Exercises

Writing Larger Programs

This Chapter deals with theoretical and practical aspects that need to be considered when writing larger programs.

When writing large programs we should divide programs up into modules. These would be separate source files. `main()` would be in one file, `main.c` say, the others will contain functions.

We can create our own library of functions by writing a *suite* of subroutines in one (or more) modules. In fact modules can be shared amongst many programs by simply including the modules at compilation as we will see shortly..

There are many advantages to this approach:

- the modules will naturally divide into common groups of functions.
- we can compile each module separately and link in compiled modules (more on this later).
- UNIX utilities such as **make** help us maintain large systems (see later).

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Running Make **Up:** Writing Larger Programs **Previous:** Creating a makefile

Make macros

We can define *macros* in make - they are typically used to store source file names, object file names, compiler options and library links.

They are simple to define, *e.g.*:

where `(SOURCES: .c = .o)` makes `.c` extensions of `SOURCES .o` extensions.

To reference or invoke a macro in make do `$(macro_name)`. *e.g.:*

NOTE:

- `$(PROGRAM) : $(OBJECTS)` - makes a list of dependencies and targets.
- The use of an internal macros *i.e.* `$@`.

There are many internal macros (see manual pages) here a few common ones:

`$*`

- file name part of current dependent (minus .suffix).

`$@`

- full target name of current target.

`$<`

- .c file of target.

Appendix contains an example makefile for the WriteMyString modular program discussed in the last Chapter.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Directory handling functions **Up:** UNIX and C **Previous:** Using UNIX System Calls and Library Functions

File and Directory Manipulation

There are many UNIX utilities that allow us to manipulate directories and files. `cd, ls, rm, cp, mkdir` *etc.* are examples we have (hopefully) already met.

We will now see how to achieve similar tasks from within a C program.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: File Manipulation Routines **Up:** File and Directory Manipulation **Previous:** File and Directory Manipulation

Directory handling functions

This basically involves calling appropriate functions.

`int chdir(char path)` - changes directory to specified path string.

Example: C emulation of UNIX's `cd` command:

`char getwd(char path)` - get the full pathname of the current working directory. path is a pointer to a string where the pathname will be returned. getwd returns a pointer to the string or NULL if an error occurs.

`scandir(char dirname, struct direct namelist, int (*select)(), int (compar)())` - reads the directory dirname and builds an array of pointers to directory entries or -1 for an error. namelist is a pointer to an array of structure pointers.

(*select))() is a pointer to a function which is called with a pointer to a directory entry (defined in <sys/types> and should return a non zero value if the directory entry should be included in the array. If this pointer is NULL, then all the directory entries will be included.

The last argument is a pointer to a routine which is passed to qsort (see man qsort) - a built in function which sorts the completed array. If this pointer is NULL, the array is not sorted.

`alphasort(struct direct d1, d2)` - alphasort() is a built in routine which will sort the array alphabetically.

Example - a simple C version of UNIX ls utility scandir returns the current directory (.) and the directory above this (..) as well as all files so we need to check for these and return FALSE so that they are not included in our list.

Note: scandir and alphasort have definitions in sys/types.h and sys/dir.h. MAXPATHLEN and getwd definitions in sys/param.h

We can go further than this and search for specific files: Let's write a modified file_select() that only scans for files with a .c, .o or .h suffix:

NOTE: rindex() is a string handling function that returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c does not occur in the string. (index() is similar function but assigns a pointer to 1st occurrence.)

Next Up Previous

Next: File Manipulation Routines **Up:** File and Directory Manipulation **Previous:** File and Directory Manipulation

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: errno **Up:** File and Directory Manipulation **Previous:** Directory handling functions

File Manipulation Routines

`int access(char *path, int mode)` - determine accessibility of file.

`path` points to a path name naming a file. `access()` checks the named file for accessibility according to `mode`, defined in `#include <unistd.h>`:

R_OK
: - test for read permission

W_OK
: - test for write permission

X_OK
: - test for execute or search permission

F_OK
: - test whether the directories leading to the file can be searched and the file exists.

`access()` returns: 0 on success, -1 on failure and sets `errno` to indicate the error. See `man` pages for list of errors.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Process Control and Management **Up:** File and Directory Manipulation **Previous:** File Manipulation Routines

errno

`errno` is a special system variable that is set if a system call cannot perform its set task.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program other wise it simply retains its last value.

`int chmod(char *path, int mode)` change the mode of access of a file. specified by `path` to the given `mode`.

`chmod()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are defined in `#include <sys/stat.h>`

The access mode of a file can be set using predefined macros in `sys/stat.h` - see `man` pages - or by setting the mode in a a 3 digit octal number.

The rightmost digit specifies owner privileges, middle group privileges and the leftmost other users privileges.

For each octal digit think of it a 3 bit binary number. Leftmost bit = read access (on/off) middle is write, right is executable.

So 4 (octal 100) = read only, 2 (010) = write, 6 (110) = read and write, 1 (001) = execute.

so for access mode 600 gives user read and write access others no access. 666 gives everybody read/write access.

NOTE: a UNIX command `chmod` also exists

`int stat(char *path, struct stat *buf), int fstat(int fd, struct stat *buf)`

`stat()` obtains information about the file named by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

`fstat()` obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an `open` call (Low level I/O).

`buf` is a pointer to a stat structure into which information is placed concerning the file. A stat structure is define in #include <sys/types.h>, see `man` pages for more information.

`stat(), and fstat()` return 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are again defined in #include <sys/stat.h>

`int unlink(char *path)` - removes the directory entry named by `path`

`unlink()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors listed in #include <sys/stat.h>

NOTE: There are a few more file manipulation routines (Appendix).

Next Up Previous
Next: Process Control and Management **Up:** File and Directory Manipulation
Previous: File Manipulation Routines

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Running UNIX Commands from C **Up:** UNIX and C **Previous:** errno

Process Control and Management

A *process* is basically a single running program. It may be a ``system'' program (*e.g* login, update, csh) or program initiated by the user (textedit, dbxtool or a user written one).

When UNIX runs a process it gives each process a unique number - a process ID, `pid`.

The UNIX command `ps` will list all current processes running on your machine and will list the pid.

The C function `int getpid()` will return the pid of process that called this function.

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Dave.Marshall@cm.cf.ac.uk

Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: execl() **Up:** Process Control and Management **Previous:** Process Control and Management

Running UNIX Commands from C

We can run commands from a C program just as if they were from the UNIX command line by using the `system()` function. **NOTE:** this can save us a lot of time and hassle as we can run other (proven) programs, scripts *etc.* to do set tasks.

`int system(char *string)` - where string can be the name of a unix utility, an executable shell script or a user program. System returns the exit status of the shell.

Example: Call `ls` from a program

```
system is a call that is made up of 3 other commands:
execl(), wait() and fork()
```


Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: fork() **Up:** Running UNIX Commands from C **Previous:** Running UNIX Commands from C

execl()

`execl` has 5 other related functions - see `man` pages.

`execl` stands for *execute* and *leave* which means that a process will get executed and then terminated by `execl`.

It is defined by:

```
execl(char *path, char *arg0,...,char *argn, 0);
```

The last parameter must always be 0. It is a *NULL terminator*. Since the argument list is variable we must have some way of telling C when it is to end. The NULL terminator does this job.

where `path` points to the name of a file holding a command that is to be executed, `argo` points to a string that is the same as path (or at least its last component.

`arg1 ... argn` are pointers to arguments for the command and 0 simply marks the end of the (variable) list of arguments.

So our above example could look like this also:

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: wait() **Up:** Running UNIX Commands from C **Previous:** execl()

fork()

`int fork()` turns a single process into 2 identical processes, known as the *parent* and the *child*. On success, fork() returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, fork() returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

NOTE: The child process will have its own unique PID.

The following program illustrates a simple use of fork, where two copies are made and run together (multitasking)

The Output of this would be:

NOTE: The processes have unique ID's which will be different at each run.

It also impossible to tell in advance which process will get to CPU's time - so one run may differ from the next.

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child. We can trap any errors if fork returns a -1. *i.e.*:

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

wait()

`int wait (int *status_location)` - will force a parent process to wait for a child process to stop or terminate. `wait()` return the pid of the child or -1 for an error. The exit status of the child is returned to `status_location`.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Piping in a C program **Up:** Running UNIX Commands from C **Previous:** wait()

exit()

`int exit(int status)` - terminates the process which calls this function and returns the exit `status` value. Both UNIX and C (forked) programs can read the status value.

By convention, a status of 0 means *normal termination* any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the `sys/stat.h` header file. We can easily derive our own conventions.

A complete example of forking program is in Appendix and is originally titled `fork.c`

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: popen() - Formatted Piping **Up:** Process Control and Management **Previous:** exit()

Piping in a C program

Piping is a process where the input of one process is made the input of another. We have seen examples of this from the UNIX command line using |.

We will now see how we do this from C programs.

We will have two (or more) `forked` processes and will communicate between them.

We must first open a *pipe*

UNIX allows two ways of opening a pipe.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: pipe() - Low level Piping **Up:** Piping in a C program **Previous:** Piping in a C program

popen() - Formatted Piping

`FILE *popen(char *command, char *type)` - opens a pipe for I/O where the command is the process that will be connected to the calling process thus creating the *pipe*. The type is either ``r'' - for reading, or ``w'' for writing.

`popen()` returns is a stream pointer or NULL for any errors.

A pipe opened by `popen()` should always be closed by `pclose(FILE *stream)`.

We use `fprintf()` and `fscanf()` to communicate with the pipe's `stream`.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Interrupts and Signals **Up:** Piping in a C program **Previous:** popen() - Formatted Piping

`pipe()` - Low level Piping

`int pipe(int fd[2])` - creates a pipe and returns two file descriptors, `fd[0], fd[1]`. `fd[0]` is opened for reading, `fd[1]` for writing.

`pipe()` returns 0 on success, -1 on failure and sets `errno` accordingly.

The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using `read()` and `write()`.

Pipes opened with `pipe()` should be closed with `close(int fd)`.

Example: Parent writes to a child

An example of piping in a C program is `plot.c and subroutines and is detailed in Appendix` .

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous

Next: Sending Signals - kill() **Up:** Process Control and Management **Previous:** pipe() - Low level Piping

Interrupts and Signals

In this section will look at ways in which two processes can communicate. When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

The process uses *signals* which can be numbered 0 to 31. Macros are defined in `signal.h` header file for common signals.

These include:

```
SIGHUP 1 /* hangup */
SIGQUIT 3 /* quit */
SIGABRT 6 /* used by abort */
SIGALRM 14 /* alarm clock */
SIGCONT 19 /* continue a stopped process */
SIGCHLD 20 /* to parent on child stop or exit */
SIGINT 2 /* interrupt */
SIGILL 4 /* illegal instruction */
SIGKILL 9 /* hard kill */
```


Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Sending Signals - `kill()`

`int kill(int pid, int signal)` - send a `signal` to a process, `pid`. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

`kill()` returns 0 for a successful call, -1 otherwise and sets `errno` accordingly.

There is also a UNIX command called kill - see man pages.

NOTE: that unless caught or ignored, the `kill` signal terminates the process. Therefore protection is built into the system.

Only processes with certain access privileges can be killed off.

Basic rule: *only processes that have the same user can send/receive messages*.

The `SIGKILL` signal cannot be caught or ignored and will always terminate a process.

For example`kill(getpid(),SIGINT);` would send the interrupt signal to the id of the calling process.

This would have a similar effect to `exit()` command. Also `ctrl-c` typed from the command sends a `SIGINT` to the process currently being.

`unsigned int alarm(unsigned int seconds)` - sends the signal `SIGALRM` to the invoking process after seconds seconds.

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994

Next Up Previous
Next: Times Up!! **Up:** Interrupts and Signals **Previous:** Sending Signals - kill()

Receiving signals - `signal()`

`int (*signal(int sig, void (*func)()))()` - that is to say the function `signal()` will call the `func` functions if the process receives a signal `sig`. Signal returns a pointer to function `func` if successful or it returns an error to `errno` and -1 otherwise.

`func()` can have three values:

`SIG_DFL`
: - a pointer to a system default function `SID_DFL()`, which will terminate the process upon receipt of `sig`.

`SIG_IGN`
: - a pointer to system ignore function `SIG_IGN()` which will disregard the `sig` action (UNLESS it is `SIGKILL`).

A function address
: - a user specified function.

`SIG_DFLand SIG_IGN` are defined in `signal.h` (standard library) header file.

Thus to ignore a `ctrl-c` command from the command line. we could do:

```
signal(SIGINT, SIG_IGN);
```

TO reset system so that `SIGINT` causes a termination at any place in our program, we would do:

```
signal(SIGINT, SIG_DFL);
```

So lets write a program to trap a `ctrl-c` but not quit on this signal. We have a function `sigproc()` that is executed when we trap a `ctrl-c`. We will also set another function to quit the program if it traps the `SIGQUIT` signal so we can terminate our program:

Finally lets write a program that communicates between child and parent processes using `kill() and signal().`

`fork() creates the child process from the parent. The pid can be checked to decide whether it is the child (== 0) or the parent (pid = child process id).`

The parent can then send messages to child using the pid and kill().

The child picks up these signals with signal() and calls appropriate functions.

An example of communicating process using signals is sig_talk.c in Appendix .

Dave.Marshall@cm.cf.ac.uk
Wed Sep 14 10:06:31 BST 1994