Visual C++ 6 Programming Tutorial

This tutorial contains a beginner's guide to Visual C++ 6, introducing the programming environment and the use of MFC classes to implement a Windows 32 user interface, defining key terms and introducing exercises to demonstrate the five control structures (sequence, selection: binary and multiway, iteration: pre-and post-test).

Syllabus outcome

H5.3 A student selects and applies appropriate software to facilitate the design and development of software solutions

Students learn to implement a fully tested and documented software solution in a methodical manner. (SDD syllabus, p.51)

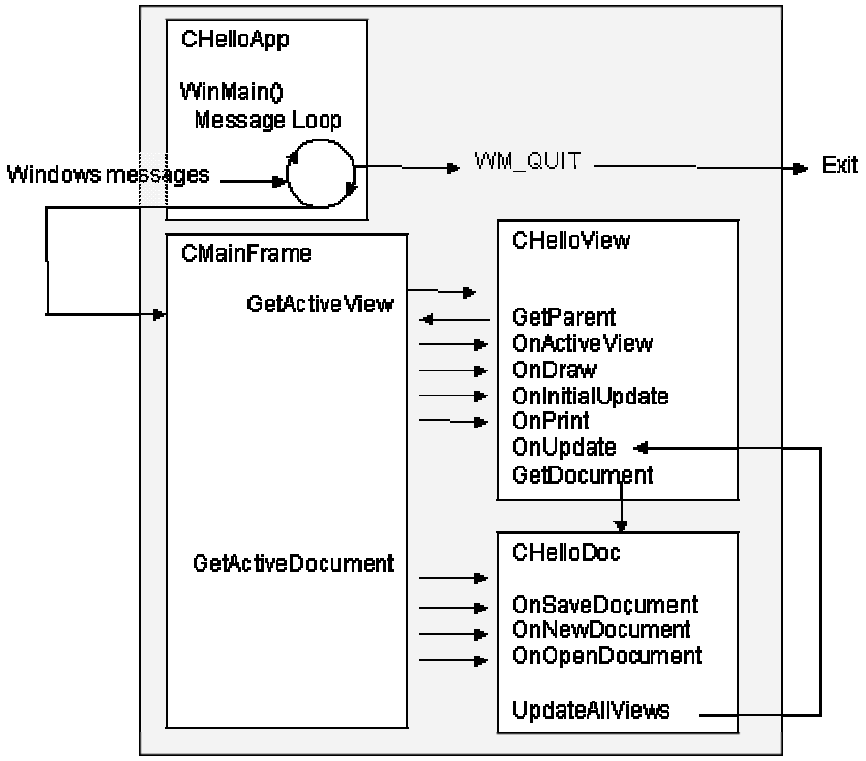
The following headings may help you navigate:

Organising your first project

The first step is to use Microsoft Foundation classes to create a Visual C++ program. There are four basic parts in a Visual C++ program.

- The **application object** (*xx*Welcome.h and *xx*Welcome.cpp) is what Windows actually runs first by calling the WinMain() method to get the program started. This object has three tasks: starting the program, placing the main window on the screen and passing Windows messages from the operating system to the main window. These messages include WM_QUIT if the user quits, WM_SIZE if the user resizes the window, WM_SETFONT and many others.
- The **main window object** (MainFrm.h and MainFrm.cpp) is responsible for the title bar, menu bar, toolbar and status bar.
- The **view object** (*xx*WelcomeView.h and *xx*WelcomeView.cpp) displays data in the client area of the window. The OnDraw() method in the WelcomeView.cpp file allows us to display messages and other objects on the screen.
- The **document object** (*xx*WelcomeDoc.h and *xx*WelcomeDoc.cpp) is where the program's data is stored.

These four objects communicate as the program runs.



Hello.exe
Shows the relationship between all the linked files in the executable version.

Activity 1: Creating a welcome screen and adding a menu item

- Open Visual C++
- Click New in the File menu, select the MFC AppWizard (exe) entry under the projects tab in the New dialog box.
- Give the new program the name *your initials***Welcome** in the Project name box, choose a location and click OK to start the MFC AppWizard.
- In Step 1, select Single document. Then accept the defaults for all the other steps.
- Use the FileView tab in the Workspace window to look for the files belonging to each of the four objects described above.
- Start by declaring two string variables welcome_string and enjoy_string in the document's header file. Open *xx*WelcomeDoc.h by double clicking in the Workspace view, find this section of the code and **add the two lines in bold font**.

```
// xxWelcomeDoc.h : interface of the xxWelcomeDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_xxWELCOMEDOC_H__9442116A_C7A7_11D3_BBC0_0000E883F891__INCLUDED_)
#define AFX_xxWELCOMEDOC_H__9442116A_C7A7_11D3_BBC0_0000E883F891__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class xxWelcomeDoc : public CDocument
{
protected: // create from serialization only
    xxWelcomeDoc();
    DECLARE_DYNCREATE(xxWelcomeDoc)
    CString welcome_string;
    CString enjoy_string;
```

- Now initialize the welcome_string and enjoy_string objects in the documents class's constructor in the file *xx*WelcomeDoc.cpp by adding the bold lines of code. Don't forget to be very fussy about copying exactly and placing the ; at the end of each line as shown.

```
// xxWelcomeDoc construction/destruction

xxWelcomeDoc:: xxWelcomeDoc ()
{
    welcome_string = "Welcome to Visual C++";
    enjoy_string = "Enjoy!";

    // TODO: add one-time construction code here
}
```

- To display the messages on the screen, open the *xx*WelcomeView.cpp file. Add the lines as shown. This code will centre your text (getting the window dimensions with the GetWindowRect() method and the text with the GetDocument() method), colour half the screen blue and add an ellipse filled with greeny-blue cross hatching.

```
// xxWelcomeView drawing

void xxWelcomeView::OnDraw(CDC* pDC)
{
    xxWelcomeDoc* pDoc = GetDocument();      //Declare a pointer variable
    ASSERT_VALID(pDoc);                      //An error handler macro
    CRect rect;

    GetWindowRect(&rect);                    //Find the Windows dimensions
    int x = rect.Width()/2;                  //and create some variables to
    int xx = rect.Width()/2;                 //position text
    int xxx = rect.Width()/2;
    int y = rect.Height();
    int yy = rect.Height()/2;
    int yyy = rect.Height()- 20;
                                             // Construct brushes to do the
    CBrush myBrush;                          cross hatching and background
    CBrush backBrush;                        colour
                                             // Then initialize and use them.
    if(myBrush.CreateHatchBrush(HS_DIAGCROSS, RGB(0, 200, 240)))
    {
        CBrush* pOldBrush = pDC->SelectObject( &myBrush );
```

```
        pDC->Ellipse(CRect(50, 100, 250, 250));
        pDC->SelectObject( pOldBrush );
    }
    if(backBrush.CreateSolidBrush(RGB(0,0,250)))
    {
        CBrush* pOlderBrush = pDC->SelectObject (&backBrush);
        pDC->Rectangle (x*2,y,x,0);
        pDC-> SelectObject(pOlderBrush);
    }
    //Centre and display welcome_string
    CSize size = pDC->GetTextExtent(pDoc->welcome_string);
    x -= size.cx/2;
    pDC->TextOut(x, 0, pDoc->welcome_string);

    // TODO: add draw code for native data here

    //Position and display enjoy_string
    CSize size2 = pDC->GetTextExtent(pDoc->enjoy_string);
    xx += (size2.cx/2 + 50);
    pDC->TextOut(xx, yy, pDoc->enjoy_string);
}
```

- Build your program with the F7 function key. if you have any errors, double click to take you to the line of code and check carefully for missing brackets, semicolons, miss typing, etc. and build again. Run you program with red exclamation mark icon OR execute from the Build Menu.
- To add a menu item, use the menu editor. Click the resources tab in the Workspace window, click the + beside menu, then double click IDR_MAINFRAME. Click on the File menu to open and grab the box at the bottom (use Insert key to add a box if necessary) and drag it up to between Print Preview and Print Setup menu items. Double click to open the Menu Item Properties box, place the caption `Print Welcome` in the caption box and close the box. This gives the new menu item the ID `ID_FILE_PRINTWELCOME`.
- Use the ClassWizard (found under the View menu) to connect the new menu item to an event handler. Make sure *xx*WelcomeView is the class selected in the Class name box. Find and select ID_FILE_PRINTWELCOME in the Objects Ids list, then double click Command in the Messages box. This brings up a message box suggesting the name for the event handler of OnFilePrintWelcome(). Click OK to accept it.
- When the user clicks our new menu item, we will display "Welcome to menus" centered on the bottom of the screen. To do this we need to declare and initialise a variable called StringData to store the string. Open the file *xx*WelcomeDoc.h and add the code in bold font to declare the string object.

```
public:
    virtual ~CShwelcomeDoc();
    CString StringData;
```

- Then open *xx*WelcomeDoc.cpp and initialize the string object in the document's constructor.

```
// xxWelcomeDoc construction/destruction

xxWelcomeDoc:: xxWelcomeDoc ()
{
     welcome_string = "Welcome to Visual C++";
     enjoy_string = "Enjoy!";
     StringData = "";
     // TODO: add one-time construction code here

}
```

- To add the string "Welcome to menus in the StringData object when the user clicks Print Welcome, we add the following code in *xx*WelcomeView.cpp.

```
void xxWelcomeView::OnFilePrintwelcome()
{
     xxWelcomeDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);

     pDoc->StringData = "Welcome to menus!";

     //Invalidate calls the OnDraw()method to display the new string
     Invalidate();

}
```

- Finally we add the code to the end of OnDraw() in *xx*WelcomeView.cpp to display the text in StringData

```
void xxWelcomeView::OnDraw(CDC* pDC)
{
      . // This represents existing code
      .
      CSize menuSize = pDC->GetTextExtent(pDoc->StringData);
      xxx -= menuSize.cx/2;
      yyy -= 20 ;
      pDC->TextOut(xxx, yyy, pDoc->StringData);

}
```

- Build your code with the F7 function key. When it is free of errors, run your code. What happens when you click on the Print Welcome menu item?

Event handlers and scroll bars

Objects and **classes** are two fundamental concepts in object-oriented languages like C++.

An **object** is a thing — anything that appears on the screen and contains functions and variables wrapped up together to perform a discrete task. This grouping together is called ***encapsulation***. For example, all the screen handling parts of a program might be put together in an object called screen. This screen object could include both the data to be displayed on the screen and the functions (or methods) to handle that data, like OnDraw() or drawLine().

A **class** is a description (or template) that is used to create an object. In the class definition, data and methods for handling the data are In C++ programming, classes are defined and objects of that class (which can be thought of as variables of that class's type) are created in the source code (.cpp) files. Visual C++ comes complete with a library of predefined classes, the MFC library, which saves a great deal of work. When we create labels or command buttons, we are creating objects of those types.

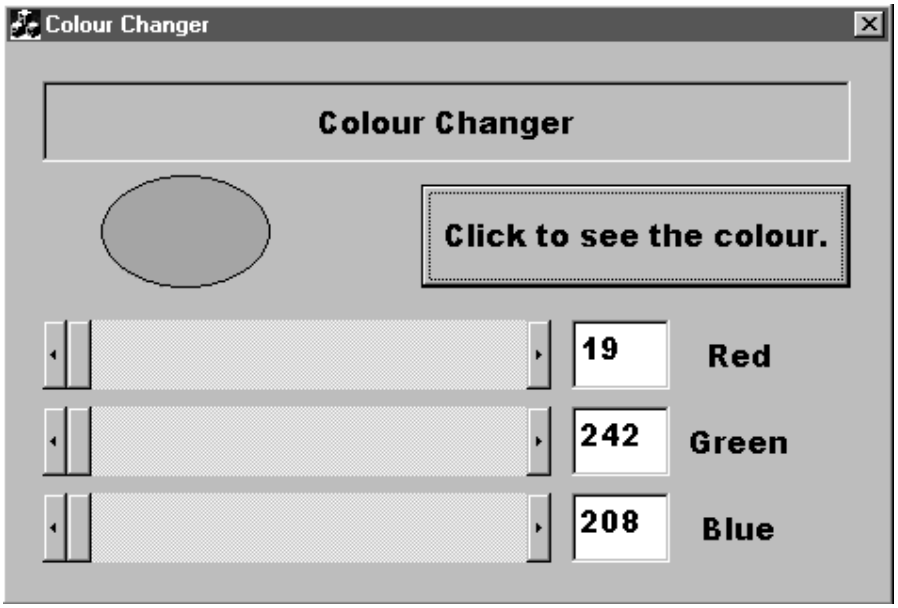
Visual C++ also supports ***inheritance.*** Using inheritance, a derived class can inherit all the properties of its base class and have extra features added.

Access modifiers are keywords used to set the scope of data members and functions within classes. The *private* keyword means that only objects of that class have access to that data member or function. It is usual to declare all data members as private. The *public* keyword means that that member is available to all other parts of the program. The *protected* keyword is used to give access to objects of that class or derived from that class by inheritance.

Events are things that happen on the screen. **Event handlers** transfer data to **methods** that complete the task.

Activity 2: Colour Changer

- Open a new project in a new workspace using the MFC AppWizard (exe) and call it ColourChanger.



- In AppWizard Step 1, select dialog based and accept the defaults in the other steps.

- Click on the TODO .. and delete. Then add three horizontal scroll bars from the toolbox. Select all and use the layout menu to make them the same size, align them and arrange them evenly. Next, add three edit boxes beside them and a label across the top. Arrange them neatly as shown in the diagram. Leave some space between the top label and the scroll bars as shown. The Layout menu will help with this.

- Right click each label to bring up the properties box and for each (under the styles tab), align text:centre, centre vertically and give them a border. Give the top label the caption `Colour Changer` and the other three `Red`, `Green` and `Blue` respectively. Right click the background of the dialog box to bring up its Properties and use this to select a font. Arial Black 12 works well. This will apply to each label on the dialog box.

- Use the View menu to open the Class Wizard. Make sure the Message Maps tab is selected and that CColourChangerDlg is the class name selected. Then scroll down the Messages until you find WM_HSCROLL. Double click to add OnHScroll to the member functions. Note that this does not relate to a particular Scroll Bar.

- Now change to the Member Variables tab in Class Wizard. Click on IDC_EDIT1 in the Control Ids box, then Add Variable. Name it m_text1, accept the default (value) category and choose CString for your variable type. Repeat this step for IDC_EDIT2 (using m_text2) and IDC_EDIT3 (with m_text3).

- Still in the Member Variables tab of the Class Wizard, give a name to each scroll bar, using Add Variable. Name them m_slide1, m_slide2 and m_slide3 respectively, choose Control for the Category and CscrollBar for the Variable type. This will be used to find which scroll bar is moved each time.

- Add a button and change its Caption to Click to see the colour by right clicking and opening Properties. Then open the Class Wizard again, select the Message Maps tab, make sure CColourChangerDlg is the class selected, select IDC_BUTTON1 and double click the message BN_CLICKED to add the OnButton1 member function.

- Now add some code to the CColourChangerDlg.cpp source code file. Find the class initialisation and **add the code in bold**.

```
BOOL CColourChangerDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

        m_slide1.SetScrollRange(1,255, true);    //Set scroll bar range and
        m_text1 = "1";                           //intitialise edit boxes to 1
        m_slide2.SetScrollRange(1,255, true);
        m_text2 = "1";
        m_slide3.SetScrollRange(1,255, true);    //This updates data from the
        m_text3 = "1";                           //variable to the edit box.
        UpdateData(false);                       //(True) updates data from
                                                 //the textbox to the variable.
        n_red = 1;
        n_green = 1;                             //Declare and initialise three
        n_blue = 1;                              //integer variables.

        // Add "About..." menu item to system
        menu.
```

- Add the following code to the void CColourChangerDlg::OnHScroll method to read the scrollbar value into the edit boxes and into the three variables, n_red, n_green and n_blue.

```
void CColourChangerDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
     if(nSBCode == SB_THUMBPOSITION)
     {
         if (pScrollBar == &m_slide1){
             m_text1.Format("%ld", nPos);
             UpdateData(false);
             n_red = nPos; }
         if (pScrollBar == &m_slide2){
             m_text2.Format("%ld", nPos);
             UpdateData(false);
             n_green = nPos;}
         if (pScrollBar == &m_slide3){
             m_text3.Format("%ld", nPos);
             UpdateData(false);
             n_blue = nPos;}
     }
     else
     {
         CDialog::OnHScroll(nSBCode, nPos,
         pScrollBar);
     }
}
```

- The last step is to add code to the void CColourChangerDlg::OnButton1() method. In this code, we construct and destruct a new device context (pDC) to draw to and declare a new object of the CBrush class to paint the colours defined in the RGB macro and determined by the n_red, n_green and n_blue variables declared earlier. Again add the code in bold.

```
void CColourChangerDlg::OnButton1()
{
    CClientDC* pDC = new CClientDC(this);
    CBrush backBrush;

    if(backBrush.CreateSolidBrush(RGB(n_red,n_green,n_blue)))
    {
        CBrush* pOlderBrush = pDC->SelectObject (&backBrush);
        pDC-> Ellipse(50,70, 140,130);

        pDC-> SelectObject(pOlderBrush);
    }
    delete pDC;
    // TODO: Add your control notification handler code here
}
```

- Now build (F7) and execute your code.

Variables, datatypes and functions

Variables are named memory locations of a size determined by the declared datatype. The following table lists some of the simple datatypes available in C++.

Datatype	Description and Range
bool	One of two values only. e.g. True or False
unsigned char	Positive numeric values without decimals from 0-255
unsigned short (OR int)	Integers from 0 – 65,535 (but the older int is system dependent).
short int	Integers from –32,768 to 32,767
unsigned long int	Integer values from 0 to 4,294,967,295
long int	Integer values from –2,147,483,648 to 2,147,483,647
float	Real numbers to 3.4E +/- 38 (7 digits)
double	Real numbers to 1.7E +/- 308 (15 digits) (You can also have a long double at 1.2E +/- 4932 (19 digits)
CString	Not really a simple datatype. A predefined C++ class for handling strings of alphanumeric characters
enum	For defining datatypes based on predefined datatypes

A **function** is a segment of code that accepts zero, one or more arguments and returns a single result. Some perform basic mathematical tasks, e.g.

```
//NB This code depends on using the #include <CMath> header file
// at the beginning of this code.

unsigned long myValue = 2;
unsigned long cubed = cube(myValue)          //cubed = 8
unsigned long squared = square(cubed)        //squared = 64
```

Classes combine data and related functions (called methods of the class).

An **argument** is a value you pass to a function so the function has data to work with. myValue and cubed are both examples of arguments.

A function is **declared** in a function prototype using the following syntax:

return_type function_name ([type[parameterName]] …);

For example, `int CalculateArea(int length, int width);`

A function prototype tells the compiler the return type, name and parameter list. Functions are not required to have parameters but must contain the parentheses even if empty. A function may take the return type `void` if no value is to be returned. A prototype always ends with a semicolon.

The function **definition** tells the compiler how the function works using this syntax:

```
return_type function_name ([type parameterName]…)
{
      statements;
}
```

For example,

```
int CalculateArea(int l, int w)
{
      return l*w;
}
```

A function definition must agree with its prototype in return type and parameter list. It must provide names for all the parameters, and braces must surround the body of the function. All statements within the braces must end in semicolons, but the function itself is not ended with a semicolon.

Activity 3: Dialog boxes

In this program we will use a dialog box to allow a user to input data and use that data to display a personalised message on the main client area of the screen.

- Start a new program by selecting New in the File menu, select the MFC AppWizard (exe) entry under the projects tab in the New dialog box.
- Give the new program the name **ComputerConversation** in the Project name box, choose a location and click OK to start the MFC AppWizard. In Step 1, select Single document. Then accept the defaults for all the other steps.
- To create an input dialogue box, select the Resource item from the Insert menu. This opens the Insert Resource box. Select the Dialog entry and click the new button. This opens the Dialog box editor.
- Add a label, right click and open its properties. Change the caption to "Type in your name please."
- Add an edit box underneath and a button (with the caption "Talk to me!").
- Choose Class Wizard from the View menu to create a new dialog box class. Choose create a new class and name it Dlg. Click OK. To connect an event handler to the button, select IDC_BUTTON1 and double click the BN_CLICKED entry in the Messages box. This creates the new member function OnButton1().
- Now change to the Member Variables tab of the Class Wizard, select IDC_EDIT1 and click the Add Variable button. Create the variable m_text of the value category as type CString.
- Connect m_text to the IDC_EDIT control using the DDX_text() method 0f the DoDataExchange() class. Add the bold code to dlg.cpp.

```
void Dlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
```

```
    //{{AFX_DATA_MAP(Dlg)
    DDX_Text(pDX, IDC_EDIT1, m_text);
    //}}AFX_DATA_MAP
}
```

- Get the user name from the edit box using UpdateData(false) which reads from the edit box to the variable. Add "Hello there," with string concatenation, display a message box and display back to the main screen with UpdateData(true).

```
void Dlg::OnButton1()
{
    UpdateData(true);
    m_text = "Hello there, "  + m_text + "!";
    UpdateData(false);
    AfxMessageBox(m_text);
    // TODO: Add your control notification handler code here
}
```

- The next step is displaying the dialog box. To do this we have to include the support for our Dlg class in the view class by including the Dlg.h header file in ComputerConversationView.cpp.

```
// ComputerConversationView.cpp : implementation of the
// ComputerConversationView class
//

#include "stdafx.h"
#include "ComputerConversation.h"

#include "ComputerConversationDoc.h"
#include "ComputerConversationView.h"
#include "Dlg.h"
```

- Now we can make use of our Dlg class by creating a new object of that class and displaying it on the screen with the DoModal() method, which returns an integer value. [NB: If you add a Show Dialog item to the Program's file menu and connect that menu choice to the view class's method OnFileShowdialog() you can add these same lines of code to this method to enable the user to show the dialog box again after it is closed.] Use the ClassWizard to add the OnInitialUpdate method. [Make sure CcomputerConversationView is the class selected and doubleclick OnInitialUpdate in the messages box]

```
void CComputerConversationView::OnInitialUpdate()
{
    CView::OnInitialUpdate();

    Dlg dlg;
    int display = dlg.DoModal();
    // TODO: add construction code here
}
```

- The last step is to add the message in m_text to the client screen after the dialog box is closed with the OK button. Use the ClassWizard to add the onOK() method to Dlg.cpp. [Find IDOK, and doubleclick BN_CLICKED]. Now add code.

```
void Dlg::OnOK()
{
    // TODO: Add extra validation here
    // to make sure m_text is holding the text from the edit box
    UpdateData(true);
    //to close the dialog box and return the value IDOK.
```

```
        CDialog::OnOK();
    }
```

- Add code to CComputerConversationView.cpp to update the display.

```
    void CComputerConversationView::OnInitialUpdate()
    {
        CView::OnInitialUpdate();

        Dlg dlg;
        int display = dlg.DoModal();
        // TODO: add construction code here
        if(display ==IDOK){
            CComputerConversationDoc* pDoc = GetDocument();
            ASSERT_VALID(pDoc);

            pDoc->StringData = dlg.m_text;
            Invalidate();
        }
    }
```

- Add a CString object named StringData to CComputerConversationDoc.h.

```
    class CComputerConversationDoc : public CDocument
    {
    protected: // create from serialization only
        CComputerConversationDoc();
        DECLARE_DYNCREATE(CComputerConversationDoc)
    // Attributes
    public:
    CString StringData;
    // Operations
```

- To complete the program, draw the text from the dialog box in the view's OnDraw() method.

```
    void CComputerConversationView::OnDraw(CDC* pDC)
    {
         CComputerConversationDoc* pDoc = GetDocument();
         ASSERT_VALID(pDoc);
         // TODO: add draw code for native data here
         pDC->TextOut(0,0,pDoc->StringData);
    }
```

- The program is complete. Build it (debug any errors carefully) and run it.

Remarks are added in code to explain the purpose of a section of code or to add information for code maintenance. If // is placed in front of the remark, that line of code is ignored completely.

Activity 4: Create a calculator and avoid division by zero

Use a dialog box to create a calculator that can add, subtract, multiply and divide two numbers given by the user.

- Open a new project in a new workspace using the MFC AppWizard (exe) and call it Calculator.

- In AppWizard Step 1, select dialog based and accept the defaults in the other steps.

- Click on the TODO .. and delete. Add two edit boxes and line them up using tools in Layout menu. Use the Class Wizard from the View menu to connect each to a float, value number1 or number2. Add another edit box at the bottom and connect to a float value answer.

- Add 4 small buttons in a line and right click each to access Properties and change their captions to Add, Subtract, Multiply and Divide. Using Class Wizard again, double click their BN_CLICKED event but change the member function names to OnAdd, OnSubtract, OnMultiply and OnDivide.

- Open CalculatorDlg.cpp and add the following code to the OnAdd method. Using cut and paste, add the same code to OnSubtract, OnMultiply and OnDivide, changing the operator in the second line appropriately.

```
void CCalculatorDlg::OnAdd()
{
    UpdateData(true);
    answer = number1 + number2;
    UpdateData(false);
    / /TODO: Add your control notification handler code here
}
```

- Now to ensure that the program does not attempt division by zero. From the Insert menu, Resource.. add a new dialog. Place a static label across the top, right click to access the properties, change the caption to read "You cannot divide by zero! Please enter another number". [Remember, you can access font size by right clicking the dialog box background and selecting its properties.] Double click the new dialog box and accept the defaults to create a new class called dlg.

- Open CalculatorDlg.cpp again and add code to access the new dialog box.

```
// CalculatorDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Calculator.h"
#include "CalculatorDlg.h"

#include "dlg.h"
```

- Use the binary selection control structure if (condition) { [statements;] …} else { [statements;]… } to prevent division by zero in the OnDivide method.

```
void CCalculatorDlg::OnDivide()
{
     UpdateData(true);

     if (number2 == 0){
         dlg error;
         int display = error.DoModal();
     }
     else {
         answer = number1 / number2;
         UpdateData(false);
     }
     // TODO: Add your control notification handler code here
}
```

- Build and run your program. Add a **remark** (put // at the beginning of the line) at the top of your code which includes your name and the date.

Sequence algorithms

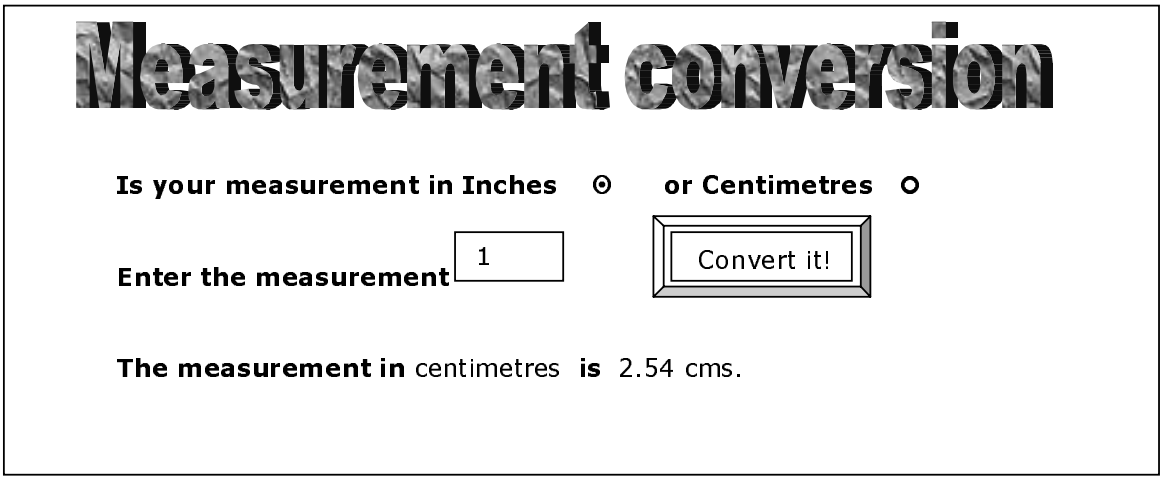
The programs in Activities 1 — 4 were mainly constructed from **sequence** algorithm constructs. Each line of code followed another with only one possible pathway for each event. So, for each **member function,** the algorithm would consist of input, output and process steps, e.g.

```
void CCalculatorDlg::OnAdd()
{
    UpdateData(true);             // Input user numbers
    answer = number1 + number2;   // Calculate answer
    UpdateData(false);            // Output answer
}
```

Binary Selection algorithms

We have also used the second algorithm construct — **selection.** Selection allows multiple pathways for any event and allows for choices to be made. Selection constructs can be **Binary** (two way) or **Multiway** (multiple choices). You have used binary selection in your calculator to prevent a user dividing by zero.

Activity 5: Measurement Converter (Binary Selection)



- Create a new program (again dialog based) called Converter to convert inches to centimetres OR centimetres to inches (using the conversion 1 inch = 2.54 centimetres).
- Use option buttons (from the toolbox) for the user to indicate whether the conversion is inches to centimetres or centimetres to inches.
- Use If statements to determine which formula to use based on which option button is selected. **Option buttons** are mutually exclusive, only one can be selected at a time. i.e.
- Use an edit box (connected to a float named `measurement` by the ClassWizard) to obtain the user input. Add two radio buttons labelled "cms to inches" and "inches to cms" and another edit box (connected to a float named `answer`). Change the caption of the OK button to `Convert` and the caption of the cancel button to `Close`.

- Use the ClassWizard to create OnRadio1(), OnRadio2() and OnOK() in ConverterDlg.cpp.
- Open ConverterDlg.h and add a boolean variable named flag.

```
// CConverterDlg dialog

class CConverterDlg : public CDialog
{
// Construction
public:
      CConverterDlg(CWnd* pParent = NULL);      // standard constructor
      bool flag;
```

- Open ConverterDlg.cpp and add the following code in bold font.

```
void CConverterDlg::OnRadio1()
{
     flag = true;
}
void CConverterDlg::OnRadio2()
{
     flag = false;
}
void CConverterDlg::OnOK()
{
     UpdateData(true);
     if (flag == false) {
          answer = float(measurement * 2.54);
          unit = "cms";
          UpdateData(false);
     }
     else {
          answer = float(measurement / 2.54);
          unit = "inches";
          UpdateData(false);
     }
     //CDialog::OnOK();
}
```

- Build and run the application to ensure that it is working correctly.
- Use your calculator to verify the results. Try it out with some **test data** including very large numbers, very small numbers, zero, negative numbers, 0.000000987654.

Multiway selection

In Activity 5, we looked at an example of binary selection. If the selection involves more than two alternatives, you can use nested If statements but this is complicated and can lead to hard-to-read code. It is sometimes better to use **Switch** statements. Here is the syntax for multiple selection through **Switch** statements.

```
switch (integralOrEnumExpression) Statement
{
       case ConstantExpression:      Statement;
                                     break;
       .
       .
       .
       default:                      statement
}
```

Activity 6: Parcel Post

The post office has the following charges for parcels based upon different weights.

Weight (gram)	Cost
0 – 50	$1.40
51 – 100	$2.70
101 – 250	$4.00
251 – 500	$7.50

Parcels which are heavier than 500g are calculated by weight*0.02

Design a project (dialog based) that allows a user to enter the weight in an edit box (connected to a float value named *weight*). Calculate the postage when the user clicks a button and display the cost in an edit box (with a connected float variable called *cost*) according to the information given above. Use **Switch** statements in your code to practise the technique. Note that only integers (or their equivalent, e.g. char) can be used for the condition in a switch statement.

```
void CParcelPostDlg::OnCalculate()
{
     UpdateData(true);
     int type = 0;

     if (weight < 51)type = 1;
     else if (weight < 101)type = 2;
     else if (weight < 251)type = 3;
     else if (weight < 501)type = 4;

     switch (type)
     {
         case < 1:      cost = float(1.40);
                        break;
         case < 2:      cost = float(2.70);
                        break;
         case < 3:      cost = float(4.00);
                        break;
         case < 4:      cost = float(7.50);
                        break;
         default:       cost = float(weight * 0.02);
     }
     UpdateData(false);
}
```

Switch statements are rarely implemented in C++. A series of ordered -else if -statements does the same job more elegantly. Switch statements are sometimes used to distinguish between members of an enumerated list.

Iterations

Iterations or loops are structures that allow a statement or group of statements to be carried out repeatedly while some condition remains true (or for a counted number of times). **Each iteration MUST contain a way of stopping the looping.** There are 2 basic iteration structures:

- **Pre-test iterations:** In these loops, the condition to be met occurs at the beginning of the loop and ***the code will not run at all if the condition is never met***.

- **Post-test iterations:** in these loops, the condition to be met is at the end of the loop so that ***the code always runs at least once***.

Activity 7: Pre-test loop Beeper

In this project, we will create an ActiveX Control to cause a beep, then use a pre-test loop to sound as many beeps as the number the user inputs.

- First create our ActiveX control. Under New..Projects, select MFCActiveX ControlWizard and name the Project Beep.

- Accept the defaults for Step 1, but in step 2, select BUTTON in the drop down menu under the question "Which window class, if any, should this control subclass?" Click Finish and OK to create the Beep class.

- Give your button a caption. Open BeepCtl.cpp and insert the following code.

```
CBeepCtrl::CBeepCtrl()
{
     InitializeIIDs(&IID_DBeep, &IID_DBeepEvents);

     // TODO: Initialize your control's instance data here.
     SetText("Click Me!");
}
```

- Now we need to add a click event to the button. Open ClassWizard in the View menu and select the ActiveX events tab. Select Add event. Select Click for the external name (giving an internal name of FireClick) and check that the implementation is Stock and OK. Still in the class wizard, select the Message Maps tab and add the OnLButtonDown() (by double clicking WM_LBUTTONDOWN in the messages box) event handler to the ActiveX control. Add code to fire the click event when the user clicks the button to our ActiveX Control in BeepCtl.cpp. This code causes the control to fire a click event each time the button is clicked, and programs that use this control will be able to set up a Click event-handler to handle such clicks.

```
void CBeepCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
     // TODO: Add your message handler code here and/or call default
     FireClick();
     COleControl::OnLButtonDown(nFlags, point);
}
```

- To finish our ActiveX control, add the method, Beep() to our control. Open ClassWizard, Automation tab and select Add Method. Name the method Beep and make its return type void. Click OK and close the ClassWizard. Add the code to make the computer beep, using the MFC MessageBeep() method.

```
void CBeepCtrl::Beep()
{
     // TODO: Add your dispatch handler code here
     MessageBeep(MB_OK);
}
```

- Now build Beep to create Beep.ocx. This also registers that control with Windows.

- Create a new dialog-based program with AppWizard now, naming this program Beeper.

- Add your Beep control to the dialog editor's toolbox using the Project menu (Add to Projects, Components and Controls Gallery). Open the Registered Controls folder, find

Beep and Insert. Accept default name CBeep and close. You will now have an OCX control in your toolbox called Beep. Insert on your form, add an edit box to accept user input and a label captioned "Enter an integer between 1 and 10, then click to hear that number of beeps."

- Use ClassWizard to connect a member variable, m_beeper, to the new Beep control and a variable called counter of type int to the textbox. Set the minimum and maximum values to 0 and 10. Now we can use the Beep() method using m_beeper.Beep().

- Use ClassWizard to add an event-handler to the Beep control's Click event. In Message Maps select IDC_BEEPCTRL1 and double click Click to add the new event handler OnClickBeepctrl1().

- Add code to OnClickBeepctrl1() to get number from the edit box and call the Beep method. Use a pre-test loop to call Beep() the asked for number of times.

```
void CbeeperDlg::OnClickBeepctrl1()
{
     UpdateData(true);
     while (counter > 0)                //A pre-test loop based on a
     {                                  //counter condition.
            m_beeper.Beep();
            counter--;                  //Changes loop condition to
                                        //terminate loop
     }
}
```

- The only remaining problem is that most modern computers run so fast that the beeps are continuous and sound like one beep. One way to solve this problem is to give the processor another task to slow it down. Add an edit box, right click to Properties and deselect Visible. Use ClassWizard to add a variable called pause of type long. Now add the following code to OnClickBeepctrl1(), using a For loop to display value of pause to the invisible edit box to slow the computer down, add code to limit the user to an integer between 1 and 10 and set the edit box back to zero for next use.

```
void CbeeperDlg::OnClickBeepctrl1()
{
     UpdateData(true);
     while ((counter > 0) && (counter < 11))
            m_beeper.Beep();
            counter--;
            for (int pause = 0; pause<=20000; pause++) UpdateData(false);
     }
     counter = 0;
     UpdateData(false);
}
```

- Build and run your program.

Activity 8: Random number generator

Use a new function, rand() **to generate a random number**. You will also need to use srand() to provide an initial seed value so that rand() does not always start at the same place. We will tie srand() to the current time (in seconds past midnight) so that it has a different seed value each time it is used.

- Design an ActiveX control button (called Dice) to generate a random number between 1 and 6. Use the MFC ActiveX ControlWizard and base it on the BUTTON class in Step 2.

- In ClassWizard-> ActiveX events tab-> Add Event, select Click as external name and OK. In ClassWizard-> Automation tab-> Add method, add a method called Random giving it the return type short. In ClassWizard-> Message Maps connect CDiceCtrl to WM_LBUTTONDOWN to create the OnLButtonDown() event-handler.

- Open DiceCtl.cpp and add the following lines of code.

```
CDiceCtrl::CDiceCtrl()
{
     InitializeIIDs(&IID_DDice, &IID_DDiceEvents);

     // TODO: Initialize your control's instance data here.
     // This line is to change the button's caption.
     SetText("Roll the dice!");
}
.
.
void CDiceCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
     // TODO: Add your message handler code here and/or call default
     FireClick();

     COleControl::OnLButtonDown(nFlags, point);
}
short CDiceCtrl::Random()
{
     // TODO: Add your dispatch handler code here
     int i;

     /* Seed the random-number generator with current time so
     that the numbers will be different every time */
     srand( (unsigned)time( NULL ) );

     i = rand()% 6 + 1;
     return i;
}
```

- Start a new dialog based application (RollTheDice) and add the new Dice Control (in Projects, Add to projects, Components and Controls). Place on the dialog box along with an edit box to display the result each time the Dice is clicked.

- Using ClassWizard->Member Variables, add m_text (as an int) to IDC_EDIT1 and m_button (of the CDice class) to IDC_DICECTRL1. Change to Message Maps tab and connect IDC_DICECTRL1 to the Click event to create the event-handler OnClickDicectrl1().

- Add the bold code to RollTheDiceDlg.cpp.

```
     void CRollTheDiceDlg::OnClickDicectrl1()
     {
          // TODO: Add your control notification handler code here
          /*Use char() to typecast integer returned by Random() so that it
          can be assigned to the CString variable m_text. */
          m_text = char (m_button.Random());
          UpdateData(false);
     }
```

- Build the program, debug any errors and run it.

Activity 9: Post-test loop to display Fibonacci numbers

Create a program to generate the first 20 Fibonacci numbers. Use a counter to control the number of iterations in a post-test loop.

- Start a new single-document project called Numbers. Open NumbersDoc.h (from the header files) and add three CString objects (data, data1 and data2) in the public section.

- Open NumbersDoc.cpp and initialise the objects.

```
CNumbersDoc::CNumbersDoc()
{
     // TODO: add one-time construction code here
     data = "The first 20 Fibonacci numbers are:";
     data1 = "";
     data2 = "";
}
```

- Open NumbersView.cpp and add the code to display the Fibonacci numbers in the OnDraw() method.

```
void CNumbersView::OnDraw(CDC* pDC)
{
     CNumbersDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     // TODO: add draw code for native data here
     short x = 0;
     short y = 1;
     int counter = 1;
                                                  //Declare variables to
     CString new1 = "";                           //format the values as
     CString new2 ="";                            //text.

     do                                           //Use post-test loop
     {
          new1.Format("%ld", x);                  //Format values of x, y
          new2.Format("%ld", y);

          pDoc->data1 += new1;                    //Concatenate to string
          pDoc->data1 += "   ";                   //Add 3 spaces
          pDoc->data1 += new2;
          pDoc->data1 += "   ";

          x+=y;                                   //Change value of x,y
          y+=x;
          counter +=2;                            //Changes loop
     }                                            //condition to
     while  (counter < 10);                       //terminate loop.

     do                                           //This post-test loop
     {                                            //concatenates values
          new1.Format("%ld", x);                  //into data2
          new2.Format("%ld", y);

          pDoc->data2 += new1;
          pDoc->data2 += "  ";
          pDoc->data2 += new2;
          pDoc->data2 += "  ";

          x+=y;
          y+=x;
          counter +=2;
```

```
        }
        while ((counter < 20) && (counter >= 10));    //Displays strings on
                                                      //screen – each string
        pDC->TextOut(0, 0, pDoc->data);               //on a new line.
        pDC->TextOut(0, 20, pDoc->data1);
        pDC->TextOut(0, 40, pDoc->data2);
    }
```

- Build and run.

Activity 10: Nested FOR Loops

Write a program that uses nested **For** loops to display the times tables from 1 – 12 with a new line for each times table.

- Open a new single project with MFCAppWizard(exe) named LearnTables.
- Declare a CString variable, `line`, in LearnTablesDoc.h.
- Intialize in learnTablesDoc.cpp (`line =" ";`) and add the following code to LearnTablesView.cpp.

```
void ClearnTablesView::OnDraw(CDC* pDC)
{
     CLearnTablesDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     // TODO: add draw code for native data here
     int i;
     int j;
     int y = 0;
     CString convert = "";

     for(i=0; i<12; i++)
     {
          pDoc->line = "";
          for(j=0; j<12; j++)
          {
               convert.Format("%d", ((i+1)*(j+1)));
               pDoc->line+= convert;
               pDoc->line+="  ";
          }
          pDC->TextOut(0,y, pDoc->line);
          y+=20;
     }
}
```

Bibliography

Holzner, S (1998). *Visual C++ in record time.* San Francisco: Sybex.

Further Resources

Dale, N and others. (2000). *Programming and Problem Solving with C++.* Sudbury, Massachusetts: Jones and Bartlett.

Free-Ed Net Course Catalog: http://www.free-ed.net/catalog.htm

This work was prepared by

Beverley Sampford