

Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

Would you like to discuss interesting topics with intelligent people? If so, you might be interested in [Colloquy](#), the world's first internet-based high-IQ society. I'm one of the Regents of this society, and am responsible for the email list and for membership applications. Application instructions are available [here](#).

Is this book for you? If you're a programmer in a language other than C++, and want to upgrade your skills, then the answer is yes. But what if you have no previous programming experience? In that case, here's a little quiz that may help you decide:

1. Do you want to know how the programs in your computer work inside, and how to write some of your own?
2. Are you willing to exert yourself mentally to learn a complex technical subject?
3. Do you have a sense of humor?

If you've answered yes to these questions and follow through with the effort required, then you will get a lot out of this book.

The common wisdom states that programming is a difficult subject that should be reserved for a small number of specialists. One of the main reasons that I have written this book is that I believe this attitude is wrong; it is possible, and even desirable, for you to learn how programs work and how to write them. Those who don't understand how computers perform their seemingly magical feats are at an increasing disadvantage in a society ever more dependent on these extraordinary machines.

Regardless of the topic, I can see no valid reason for a book to be stuffy and dry, and I've done everything possible to make this one approachable. However, don't let the casual tone fool you into thinking that the subject is easy; there is no "royal road" to programming, any more than there is to geometry. Especially if you have no prior experience in programming, this book will stretch your mind more than virtually any other subject you could study.

One of the reasons that this book is different from other books is the participation of Susan, my primary "test reader", whose account of her involvement in this project immediately follows this Preface. I recommend that you read that account before continuing with the technical material following it, as it explains how and why she contributed to making your task easier and more enjoyable.

Speaking of Susan, here is a bit of correspondence between us on the topic of how one should read this book, which occurred after her first reading of the chapters on hardware and programming basics:

Susan: Let me say this: to feel like I would truly understand it, I would really need to *study* this about two more times. Now, I could do this, but I am not sure you would want me to do so. I think reading a chapter once is enough for most people.

Steve: As a matter of fact, I would expect the reader of my book to read and study this chapter several times if necessary; for someone completely new to programming, I imagine that it **would** be necessary. Programming is one of the most complex human disciplines, although it doesn't take the mathematical skills of a subject such as nuclear physics, for example. I've tried to make my explanations as simple as possible, but there's no way to learn programming (or any other complex subject) without investing a significant amount of work and thought.

After she had gone through the text a number of times and had learned a lot from the process, we continued this discussion as follows:

Susan: Well then, maybe this should be pointed out in a Preface or something. Of course, it would eventually be obvious to the reader as it was to me, but it took me awhile to come to that conclusion. The advantage of knowing this in advance is that maybe I would not be so discouraged that I was not brilliant after one read of a chapter.

Steve: I will indeed mention in the Preface that the reader shouldn't be fooled by the casual tone into thinking that this is going to be a walk in the park. In any event, please

don't be discouraged. It seems to me that you have absorbed a fair amount of very technical material with no previous background; that's something to be proud of!

We'll be hearing from Susan many more times in the course of the book. She will be checking in frequently in the form of extracts from the e-mail discussion we engaged in during the testing and revising process. I hope you will find her comments and my replies add a personal touch to your study of this technical material.

I am always happy to receive correspondence from readers. If you wish to contact me, the best way is to visit [my WWW home page](#).

If you prefer, you can [email me](#).

In the event that you enjoy this book and would like to tell others about it, you might want to write an on-line review on Amazon.com, which you can do [here](#).

Whenever I refer to "the software from the CD in the back of the book", I am talking about the DJGPP compiler, written and copyrighted by DJ Delorie, which is available [here](#), and RHIDE, an integrated development environment for the DJGPP compiler, written and copyrighted by Robert Hoehne, which is available [here](#). The source code for the examples is available [here](#).

I should also tell you how the various typefaces are used in the book. `HelveticaNarrow` is used for program listings, for terms used in programs, and for words defined by the C++ language. *Italics* are used primarily for technical terms that are found in the glossary, although they are also used for emphasis in some places. The first time that a particular technical term is used, it is in **bold** face; if it is a term defined in the C++ language, it will be in **HelveticaNarrowBold**.

Now that those preliminaries are out of the way, let's proceed. The next voice you will hear is that of Susan, my test reader. I hope you get as much out of her participation in this book as I have.

Now, on with the show!

[Dedication](#)

[Acknowledgements](#)

[Letter from a Novice](#)

[Foreword](#)

[Prologue](#)

[Hardware Fundamentals](#)

[Basics of Programming](#)

[More Basics](#)

[Functional Literacy](#)

[Taking Inventory](#)

[Stringing Along](#)

[Down the Garden Path](#)

[Tying up Loose Ends](#)

[About the Author](#)

[Comment on this book!](#)

You can order a hardcopy version of the new, greatly expanded version of this book by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Index of /whos/code

	Name	Last modified	Size	Description
	Parent Directory	06-Oct-2002 11:18	-	
	ages.cc	06-Oct-2002 11:18	1k	
	ages.gdt	06-Oct-2002 11:18	1k	
	ages.gpr	06-Oct-2002 11:18	9k	
	basic00.cc	06-Oct-2002 11:18	1k	
	basic00.gdt	06-Oct-2002 11:18	1k	
	basic00.gpr	06-Oct-2002 11:18	10k	
	basic01.cc	06-Oct-2002 11:18	1k	
	basic01.gdt	06-Oct-2002 11:18	1k	
	basic01.gpr	06-Oct-2002 11:18	9k	
	basic02.cc	06-Oct-2002 11:18	1k	
	basic02.gdt	06-Oct-2002 11:18	1k	
	basic02.gpr	06-Oct-2002 11:18	9k	
	basic03.cc	06-Oct-2002 11:18	1k	
	basic03.gdt	06-Oct-2002 11:18	1k	
	basic03.gpr	06-Oct-2002 11:18	9k	
	basic04.cc	06-Oct-2002 11:18	1k	
	basic04.gdt	06-Oct-2002 11:18	1k	
	basic04.gpr	06-Oct-2002 11:18	9k	
	basic05.cc	06-Oct-2002 11:18	1k	
	basic05.gdt	06-Oct-2002 11:19	1k	
	basic05.gpr	06-Oct-2002 11:19	9k	
	basic06.cc	06-Oct-2002 11:19	1k	

 basic06.gdt	06-Oct-2002 11:19	1k
 basic06.gpr	06-Oct-2002 11:19	9k
 basic07.cc	06-Oct-2002 11:19	1k
 basic07.gdt	06-Oct-2002 11:19	1k
 basic07.gpr	06-Oct-2002 11:19	9k
 basic08.cc	06-Oct-2002 11:19	1k
 basic08.gdt	06-Oct-2002 11:19	1k
 basic08.gpr	06-Oct-2002 11:19	9k
 basic09.cc	06-Oct-2002 11:19	1k
 basic09.gdt	06-Oct-2002 11:19	1k
 basic09.gpr	06-Oct-2002 11:19	9k
 birthday.cc	06-Oct-2002 11:19	1k
 birthday.gdt	06-Oct-2002 11:19	1k
 birthday.gpr	06-Oct-2002 11:19	9k
 birthprt.cc	06-Oct-2002 11:19	1k
 birthprt.gdt	06-Oct-2002 11:19	1k
 birthprt.gpr	06-Oct-2002 11:19	9k
 calc1.cc	06-Oct-2002 11:19	1k
 calc1.gdt	06-Oct-2002 11:19	1k
 calc1.gpr	06-Oct-2002 11:19	9k
 calc1.out	06-Oct-2002 11:19	1k
 code.new	06-Oct-2002 11:19	1k
 code.old	06-Oct-2002 11:19	1k
 common.h	06-Oct-2002 11:19	1k
 common.in	06-Oct-2002 11:19	1k
 count1.cc	06-Oct-2002 11:19	1k
 count1.gdt	06-Oct-2002 11:19	1k

 count1.gpr	06-Oct-2002 11:19	9k
 count2.cc	06-Oct-2002 11:19	1k
 count2.gdt	06-Oct-2002 11:19	1k
 count2.gpr	06-Oct-2002 11:19	9k
 count3.cc	06-Oct-2002 11:19	1k
 count3.gdt	06-Oct-2002 11:19	1k
 count3.gpr	06-Oct-2002 11:19	9k
 count4.cc	06-Oct-2002 11:19	1k
 count4.gdt	06-Oct-2002 11:19	1k
 count4.gpr	06-Oct-2002 11:19	9k
 count5.cc	06-Oct-2002 11:19	1k
 count5.gdt	06-Oct-2002 11:19	1k
 count5.gpr	06-Oct-2002 11:19	9k
 count6.cc	06-Oct-2002 11:19	1k
 count6.gdt	06-Oct-2002 11:19	1k
 count6.gpr	06-Oct-2002 11:19	9k
 cout1.cc	06-Oct-2002 11:19	1k
 cout1.gdt	06-Oct-2002 11:19	1k
 cout1.gpr	06-Oct-2002 11:19	9k
 dangchar.cc	06-Oct-2002 11:19	1k
 dangchar.gdt	06-Oct-2002 11:19	1k
 dangchar.gpr	06-Oct-2002 11:19	9k
 dangchar.out	06-Oct-2002 11:19	1k
 func1.cc	06-Oct-2002 11:19	1k
 func1.gdt	06-Oct-2002 11:19	1k
 func1.gpr	06-Oct-2002 11:19	9k
 inita.cc	06-Oct-2002 11:19	1k

 inita.gdt	06-Oct-2002 11:19	1k
 inita.gpr	06-Oct-2002 11:19	9k
 inita.out	06-Oct-2002 11:19	1k
 initb.cc	06-Oct-2002 11:19	1k
 initb.gdt	06-Oct-2002 11:19	1k
 initb.gpr	06-Oct-2002 11:19	9k
 initb.out	06-Oct-2002 11:19	1k
 initc.cc	06-Oct-2002 11:19	1k
 initc.gdt	06-Oct-2002 11:19	1k
 initc.gpr	06-Oct-2002 11:19	9k
 initc.out	06-Oct-2002 11:19	1k
 initd.cc	06-Oct-2002 11:19	1k
 initd.gdt	06-Oct-2002 11:19	1k
 initd.gpr	06-Oct-2002 11:19	9k
 initd.out	06-Oct-2002 11:19	1k
 inite.cc	06-Oct-2002 11:19	1k
 inite.gdt	06-Oct-2002 11:19	1k
 inite.gpr	06-Oct-2002 11:19	9k
 inite.out	06-Oct-2002 11:19	1k
 initf.cc	06-Oct-2002 11:19	1k
 initf.gdt	06-Oct-2002 11:19	1k
 initf.gpr	06-Oct-2002 11:19	9k
 initf.out	06-Oct-2002 11:19	1k
 invent1.cc	06-Oct-2002 11:19	1k
 invent1.h	06-Oct-2002 11:19	1k
 invent2.cc	06-Oct-2002 11:19	1k
 invent2.h	06-Oct-2002 11:19	1k

 item1.cc	06-Oct-2002 11:19	1k
 item1.h	06-Oct-2002 11:19	1k
 item2.cc	06-Oct-2002 11:19	1k
 item2.h	06-Oct-2002 11:19	1k
 item4.cc	06-Oct-2002 11:19	1k
 item4.h	06-Oct-2002 11:19	1k
 item5.cc	06-Oct-2002 11:19	1k
 item5.h	06-Oct-2002 11:19	1k
 item6.cc	06-Oct-2002 11:19	2k
 item6.h	06-Oct-2002 11:19	1k
 itemtst1.cc	06-Oct-2002 11:19	1k
 itemtst1.gdt	06-Oct-2002 11:19	1k
 itemtst1.gpr	06-Oct-2002 11:19	9k
 itemtst2.cc	06-Oct-2002 11:19	1k
 itemtst2.gdt	06-Oct-2002 11:19	1k
 itemtst2.gpr	06-Oct-2002 11:19	11k
 itemtst2.out	06-Oct-2002 11:19	5k
 itemtst3.cc	06-Oct-2002 11:19	1k
 itemtst3.gdt	06-Oct-2002 11:19	1k
 itemtst3.gpr	06-Oct-2002 11:19	10k
 itemtst4.cc	06-Oct-2002 11:19	2k
 itemtst4.gdt	06-Oct-2002 11:19	1k
 itemtst4.gpr	06-Oct-2002 11:19	10k
 itemtst5.cc	06-Oct-2002 11:19	2k
 itemtst5.gdt	06-Oct-2002 11:19	1k
 itemtst5.gpr	06-Oct-2002 11:19	11k
 itemtst6.cc	06-Oct-2002 11:19	2k

 itemtst6.gdt	06-Oct-2002 11:19	1k
 itemtst6.gpr	06-Oct-2002 11:19	11k
 mkallnru.out	06-Oct-2002 11:19	19k
 mkalltr.ouu	06-Oct-2002 11:19	4k
 morbas00.cc	06-Oct-2002 11:19	1k
 morbas00.gdt	06-Oct-2002 11:19	1k
 morbas00.gpr	06-Oct-2002 11:19	9k
 morbas01.cc	06-Oct-2002 11:19	1k
 morbas01.gdt	06-Oct-2002 11:19	1k
 morbas01.gpr	06-Oct-2002 11:19	9k
 morbas02.cc	06-Oct-2002 11:19	1k
 morbas02.gdt	06-Oct-2002 11:19	1k
 morbas02.gpr	06-Oct-2002 11:19	9k
 morbas03.cc	06-Oct-2002 11:19	1k
 morbas03.gdt	06-Oct-2002 11:19	1k
 morbas03.gpr	06-Oct-2002 11:19	9k
 morbas04.cc	06-Oct-2002 11:19	1k
 morbas04.gdt	06-Oct-2002 11:19	1k
 morbas04.gpr	06-Oct-2002 11:19	9k
 nofunc.cc	06-Oct-2002 11:19	1k
 nofunc.gdt	06-Oct-2002 11:19	1k
 nofunc.gpr	06-Oct-2002 11:19	9k
 pump1.cc	06-Oct-2002 11:19	1k
 pump1.gdt	06-Oct-2002 11:19	1k
 pump1.gpr	06-Oct-2002 11:19	9k
 pump1a.cc	06-Oct-2002 11:19	1k
 pump1a.gdt	06-Oct-2002 11:19	1k

	pump1a.gpr	06-Oct-2002	11:19	9k
	pump2.cc	06-Oct-2002	11:19	1k
	pump2.gdt	06-Oct-2002	11:19	1k
	pump2.gpr	06-Oct-2002	11:19	9k
	readme.dif	06-Oct-2002	11:19	7k
	readme.txt	06-Oct-2002	11:19	10k
	scopclas.cc	06-Oct-2002	11:19	1k
	scopclas.gdt	06-Oct-2002	11:19	1k
	scopclas.gpr	06-Oct-2002	11:19	9k
	scopclas.out	06-Oct-2002	11:19	1k
	shop2.in	06-Oct-2002	11:19	3k
	shop3.in	06-Oct-2002	11:19	3k
	strcmp.cc	06-Oct-2002	11:19	2k
	strcmp.gdt	06-Oct-2002	11:19	1k
	strcmp.gpr	06-Oct-2002	11:19	9k
	strex1.cc	06-Oct-2002	11:19	1k
	strex1.gdt	06-Oct-2002	11:19	1k
	strex1.gpr	06-Oct-2002	11:19	9k
	strex2.cc	06-Oct-2002	11:19	1k
	strex2.gdt	06-Oct-2002	11:19	1k
	strex2.gpr	06-Oct-2002	11:19	9k
	strex3.cc	06-Oct-2002	11:19	1k
	strex3.gdt	06-Oct-2002	11:19	1k
	strex3.gpr	06-Oct-2002	11:19	9k
	strex5.cc	06-Oct-2002	11:19	1k
	strex5.gdt	06-Oct-2002	11:19	1k
	strex5.gpr	06-Oct-2002	11:19	9k

 strex6.cc	06-Oct-2002 11:19	1k
 strex6.gdt	06-Oct-2002 11:19	1k
 strex6.gpr	06-Oct-2002 11:19	9k
 string1.cc	06-Oct-2002 11:19	1k
 string1.h	06-Oct-2002 11:19	1k
 string3.cc	06-Oct-2002 11:19	1k
 string3.h	06-Oct-2002 11:19	1k
 string4.cc	06-Oct-2002 11:19	1k
 string4.h	06-Oct-2002 11:19	1k
 string5.cc	06-Oct-2002 11:19	2k
 string5.h	06-Oct-2002 11:19	1k
 string5a.cc	06-Oct-2002 11:19	2k
 string5x.cc	06-Oct-2002 11:19	2k
 string5x.out	06-Oct-2002 11:19	1k
 string5y.cc	06-Oct-2002 11:19	1k
 string5y.out	06-Oct-2002 11:19	1k
 string6.cc	06-Oct-2002 11:19	3k
 string6.h	06-Oct-2002 11:19	1k
 strsort1.cc	06-Oct-2002 11:19	1k
 strtst1.cc	06-Oct-2002 11:19	1k
 strtst3.cc	06-Oct-2002 11:19	1k
 strtst3a.cc	06-Oct-2002 11:19	1k
 strtst3a.err	06-Oct-2002 11:19	1k
 strtst4.cc	06-Oct-2002 11:19	1k
 strtst5.cc	06-Oct-2002 11:19	1k
 strtst5x.cc	06-Oct-2002 11:19	1k
 test.cc	06-Oct-2002 11:19	1k

 test.gdt	06-Oct-2002 11:19	1k
 test.gpr	06-Oct-2002 11:19	9k
 testpare.cc	06-Oct-2002 11:19	1k
 testpare.gdt	06-Oct-2002 11:19	1k
 testpare.gpr	06-Oct-2002 11:19	9k
 vect1.cc	06-Oct-2002 11:19	1k
 vect1.gdt	06-Oct-2002 11:19	1k
 vect1.gpr	06-Oct-2002 11:19	9k
 vect2.cc	06-Oct-2002 11:19	1k
 vect2.gdt	06-Oct-2002 11:19	1k
 vect2.gpr	06-Oct-2002 11:19	9k
 vect2a.cc	06-Oct-2002 11:19	1k
 vect2a.gdt	06-Oct-2002 11:19	1k
 vect2a.gpr	06-Oct-2002 11:19	9k
 vect3.cc	06-Oct-2002 11:19	1k
 vect3.gdt	06-Oct-2002 11:19	1k
 vect3.gpr	06-Oct-2002 11:19	9k
 vector.h	06-Oct-2002 11:19	3k
 wassert.h	06-Oct-2002 11:19	1k

Index of /whos/code

	Name	Last modified	Size	Description
	Parent Directory	06-Oct-2002 11:18	-	
	wassert.h	06-Oct-2002 11:19	1k	
	vector.h	06-Oct-2002 11:19	3k	
	vect3.gpr	06-Oct-2002 11:19	9k	
	vect3.gdt	06-Oct-2002 11:19	1k	
	vect3.cc	06-Oct-2002 11:19	1k	
	vect2a.gpr	06-Oct-2002 11:19	9k	
	vect2a.gdt	06-Oct-2002 11:19	1k	
	vect2a.cc	06-Oct-2002 11:19	1k	
	vect2.gpr	06-Oct-2002 11:19	9k	
	vect2.gdt	06-Oct-2002 11:19	1k	
	vect2.cc	06-Oct-2002 11:19	1k	
	vect1.gpr	06-Oct-2002 11:19	9k	
	vect1.gdt	06-Oct-2002 11:19	1k	
	vect1.cc	06-Oct-2002 11:19	1k	
	testpare.gpr	06-Oct-2002 11:19	9k	
	testpare.gdt	06-Oct-2002 11:19	1k	
	testpare.cc	06-Oct-2002 11:19	1k	
	test.gpr	06-Oct-2002 11:19	9k	
	test.gdt	06-Oct-2002 11:19	1k	
	test.cc	06-Oct-2002 11:19	1k	
	strtst5x.cc	06-Oct-2002 11:19	1k	
	strtst5.cc	06-Oct-2002 11:19	1k	

 strtst4.cc	06-Oct-2002 11:19	1k
 strtst3a.err	06-Oct-2002 11:19	1k
 strtst3a.cc	06-Oct-2002 11:19	1k
 strtst3.cc	06-Oct-2002 11:19	1k
 strtst1.cc	06-Oct-2002 11:19	1k
 strsort1.cc	06-Oct-2002 11:19	1k
 string6.h	06-Oct-2002 11:19	1k
 string6.cc	06-Oct-2002 11:19	3k
 string5y.out	06-Oct-2002 11:19	1k
 string5y.cc	06-Oct-2002 11:19	1k
 string5x.out	06-Oct-2002 11:19	1k
 string5x.cc	06-Oct-2002 11:19	2k
 string5a.cc	06-Oct-2002 11:19	2k
 string5.h	06-Oct-2002 11:19	1k
 string5.cc	06-Oct-2002 11:19	2k
 string4.h	06-Oct-2002 11:19	1k
 string4.cc	06-Oct-2002 11:19	1k
 string3.h	06-Oct-2002 11:19	1k
 string3.cc	06-Oct-2002 11:19	1k
 string1.h	06-Oct-2002 11:19	1k
 string1.cc	06-Oct-2002 11:19	1k
 strex6.gpr	06-Oct-2002 11:19	9k
 strex6.gdt	06-Oct-2002 11:19	1k
 strex6.cc	06-Oct-2002 11:19	1k
 strex5.gpr	06-Oct-2002 11:19	9k
 strex5.gdt	06-Oct-2002 11:19	1k
 strex5.cc	06-Oct-2002 11:19	1k

 strex3.gpr	06-Oct-2002 11:19	9k
 strex3.gdt	06-Oct-2002 11:19	1k
 strex3.cc	06-Oct-2002 11:19	1k
 strex2.gpr	06-Oct-2002 11:19	9k
 strex2.gdt	06-Oct-2002 11:19	1k
 strex2.cc	06-Oct-2002 11:19	1k
 strex1.gpr	06-Oct-2002 11:19	9k
 strex1.gdt	06-Oct-2002 11:19	1k
 strex1.cc	06-Oct-2002 11:19	1k
 strcmp.gpr	06-Oct-2002 11:19	9k
 strcmp.gdt	06-Oct-2002 11:19	1k
 strcmp.cc	06-Oct-2002 11:19	2k
 shop3.in	06-Oct-2002 11:19	3k
 shop2.in	06-Oct-2002 11:19	3k
 scopclas.out	06-Oct-2002 11:19	1k
 scopclas.gpr	06-Oct-2002 11:19	9k
 scopclas.gdt	06-Oct-2002 11:19	1k
 scopclas.cc	06-Oct-2002 11:19	1k
 readme.txt	06-Oct-2002 11:19	10k
 readme.dif	06-Oct-2002 11:19	7k
 pump2.gpr	06-Oct-2002 11:19	9k
 pump2.gdt	06-Oct-2002 11:19	1k
 pump2.cc	06-Oct-2002 11:19	1k
 pump1a.gpr	06-Oct-2002 11:19	9k
 pump1a.gdt	06-Oct-2002 11:19	1k
 pump1a.cc	06-Oct-2002 11:19	1k
 pump1.gpr	06-Oct-2002 11:19	9k

 pump1.gdt	06-Oct-2002 11:19	1k
 pump1.cc	06-Oct-2002 11:19	1k
 nofunc.gpr	06-Oct-2002 11:19	9k
 nofunc.gdt	06-Oct-2002 11:19	1k
 nofunc.cc	06-Oct-2002 11:19	1k
 morbas04.gpr	06-Oct-2002 11:19	9k
 morbas04.gdt	06-Oct-2002 11:19	1k
 morbas04.cc	06-Oct-2002 11:19	1k
 morbas03.gpr	06-Oct-2002 11:19	9k
 morbas03.gdt	06-Oct-2002 11:19	1k
 morbas03.cc	06-Oct-2002 11:19	1k
 morbas02.gpr	06-Oct-2002 11:19	9k
 morbas02.gdt	06-Oct-2002 11:19	1k
 morbas02.cc	06-Oct-2002 11:19	1k
 morbas01.gpr	06-Oct-2002 11:19	9k
 morbas01.gdt	06-Oct-2002 11:19	1k
 morbas01.cc	06-Oct-2002 11:19	1k
 morbas00.gpr	06-Oct-2002 11:19	9k
 morbas00.gdt	06-Oct-2002 11:19	1k
 morbas00.cc	06-Oct-2002 11:19	1k
 mkalltr.ouu	06-Oct-2002 11:19	4k
 mkallnru.out	06-Oct-2002 11:19	19k
 itemtst6.gpr	06-Oct-2002 11:19	11k
 itemtst6.gdt	06-Oct-2002 11:19	1k
 itemtst6.cc	06-Oct-2002 11:19	2k
 itemtst5.gpr	06-Oct-2002 11:19	11k
 itemtst5.gdt	06-Oct-2002 11:19	1k

 itemtst5.cc	06-Oct-2002	11:19	2k
 itemtst4.gpr	06-Oct-2002	11:19	10k
 itemtst4.gdt	06-Oct-2002	11:19	1k
 itemtst4.cc	06-Oct-2002	11:19	2k
 itemtst3.gpr	06-Oct-2002	11:19	10k
 itemtst3.gdt	06-Oct-2002	11:19	1k
 itemtst3.cc	06-Oct-2002	11:19	1k
 itemtst2.out	06-Oct-2002	11:19	5k
 itemtst2.gpr	06-Oct-2002	11:19	11k
 itemtst2.gdt	06-Oct-2002	11:19	1k
 itemtst2.cc	06-Oct-2002	11:19	1k
 itemtst1.gpr	06-Oct-2002	11:19	9k
 itemtst1.gdt	06-Oct-2002	11:19	1k
 itemtst1.cc	06-Oct-2002	11:19	1k
 item6.h	06-Oct-2002	11:19	1k
 item6.cc	06-Oct-2002	11:19	2k
 item5.h	06-Oct-2002	11:19	1k
 item5.cc	06-Oct-2002	11:19	1k
 item4.h	06-Oct-2002	11:19	1k
 item4.cc	06-Oct-2002	11:19	1k
 item2.h	06-Oct-2002	11:19	1k
 item2.cc	06-Oct-2002	11:19	1k
 item1.h	06-Oct-2002	11:19	1k
 item1.cc	06-Oct-2002	11:19	1k
 invent2.h	06-Oct-2002	11:19	1k
 invent2.cc	06-Oct-2002	11:19	1k
 invent1.h	06-Oct-2002	11:19	1k

 invent1.cc	06-Oct-2002	11:19	1k
 initf.out	06-Oct-2002	11:19	1k
 initf.gpr	06-Oct-2002	11:19	9k
 initf.gdt	06-Oct-2002	11:19	1k
 initf.cc	06-Oct-2002	11:19	1k
 inite.out	06-Oct-2002	11:19	1k
 inite.gpr	06-Oct-2002	11:19	9k
 inite.gdt	06-Oct-2002	11:19	1k
 inite.cc	06-Oct-2002	11:19	1k
 initd.out	06-Oct-2002	11:19	1k
 initd.gpr	06-Oct-2002	11:19	9k
 initd.gdt	06-Oct-2002	11:19	1k
 initd.cc	06-Oct-2002	11:19	1k
 initc.out	06-Oct-2002	11:19	1k
 initc.gpr	06-Oct-2002	11:19	9k
 initc.gdt	06-Oct-2002	11:19	1k
 initc.cc	06-Oct-2002	11:19	1k
 initb.out	06-Oct-2002	11:19	1k
 initb.gpr	06-Oct-2002	11:19	9k
 initb.gdt	06-Oct-2002	11:19	1k
 initb.cc	06-Oct-2002	11:19	1k
 inita.out	06-Oct-2002	11:19	1k
 inita.gpr	06-Oct-2002	11:19	9k
 inita.gdt	06-Oct-2002	11:19	1k
 inita.cc	06-Oct-2002	11:19	1k
 func1.gpr	06-Oct-2002	11:19	9k
 func1.gdt	06-Oct-2002	11:19	1k

 func1.cc	06-Oct-2002 11:19	1k
 dangchar.out	06-Oct-2002 11:19	1k
 dangchar.gpr	06-Oct-2002 11:19	9k
 dangchar.gdt	06-Oct-2002 11:19	1k
 dangchar.cc	06-Oct-2002 11:19	1k
 cout1.gpr	06-Oct-2002 11:19	9k
 cout1.gdt	06-Oct-2002 11:19	1k
 cout1.cc	06-Oct-2002 11:19	1k
 count6.gpr	06-Oct-2002 11:19	9k
 count6.gdt	06-Oct-2002 11:19	1k
 count6.cc	06-Oct-2002 11:19	1k
 count5.gpr	06-Oct-2002 11:19	9k
 count5.gdt	06-Oct-2002 11:19	1k
 count5.cc	06-Oct-2002 11:19	1k
 count4.gpr	06-Oct-2002 11:19	9k
 count4.gdt	06-Oct-2002 11:19	1k
 count4.cc	06-Oct-2002 11:19	1k
 count3.gpr	06-Oct-2002 11:19	9k
 count3.gdt	06-Oct-2002 11:19	1k
 count3.cc	06-Oct-2002 11:19	1k
 count2.gpr	06-Oct-2002 11:19	9k
 count2.gdt	06-Oct-2002 11:19	1k
 count2.cc	06-Oct-2002 11:19	1k
 count1.gpr	06-Oct-2002 11:19	9k
 count1.gdt	06-Oct-2002 11:19	1k
 count1.cc	06-Oct-2002 11:19	1k
 common.in	06-Oct-2002 11:19	1k

 common.h	06-Oct-2002 11:19	1k
 code.old	06-Oct-2002 11:19	1k
 code.new	06-Oct-2002 11:19	1k
 calc1.out	06-Oct-2002 11:19	1k
 calc1.gpr	06-Oct-2002 11:19	9k
 calc1.gdt	06-Oct-2002 11:19	1k
 calc1.cc	06-Oct-2002 11:19	1k
 birthprt.gpr	06-Oct-2002 11:19	9k
 birthprt.gdt	06-Oct-2002 11:19	1k
 birthprt.cc	06-Oct-2002 11:19	1k
 birthday.gpr	06-Oct-2002 11:19	9k
 birthday.gdt	06-Oct-2002 11:19	1k
 birthday.cc	06-Oct-2002 11:19	1k
 basic09.gpr	06-Oct-2002 11:19	9k
 basic09.gdt	06-Oct-2002 11:19	1k
 basic09.cc	06-Oct-2002 11:19	1k
 basic08.gpr	06-Oct-2002 11:19	9k
 basic08.gdt	06-Oct-2002 11:19	1k
 basic08.cc	06-Oct-2002 11:19	1k
 basic07.gpr	06-Oct-2002 11:19	9k
 basic07.gdt	06-Oct-2002 11:19	1k
 basic07.cc	06-Oct-2002 11:19	1k
 basic06.gpr	06-Oct-2002 11:19	9k
 basic06.gdt	06-Oct-2002 11:19	1k
 basic06.cc	06-Oct-2002 11:19	1k
 basic05.gpr	06-Oct-2002 11:19	9k
 basic05.gdt	06-Oct-2002 11:19	1k

 basic05.cc	06-Oct-2002	11:18	1k
 basic04.gpr	06-Oct-2002	11:18	9k
 basic04.gdt	06-Oct-2002	11:18	1k
 basic04.cc	06-Oct-2002	11:18	1k
 basic03.gpr	06-Oct-2002	11:18	9k
 basic03.gdt	06-Oct-2002	11:18	1k
 basic03.cc	06-Oct-2002	11:18	1k
 basic02.gpr	06-Oct-2002	11:18	9k
 basic02.gdt	06-Oct-2002	11:18	1k
 basic02.cc	06-Oct-2002	11:18	1k
 basic01.gpr	06-Oct-2002	11:18	9k
 basic01.gdt	06-Oct-2002	11:18	1k
 basic01.cc	06-Oct-2002	11:18	1k
 basic00.gpr	06-Oct-2002	11:18	10k
 basic00.gdt	06-Oct-2002	11:18	1k
 basic00.cc	06-Oct-2002	11:18	1k
 ages.gpr	06-Oct-2002	11:18	9k
 ages.gdt	06-Oct-2002	11:18	1k
 ages.cc	06-Oct-2002	11:18	1k

Index of /whos/code

	Name	Last modified	Size	Description
	Parent Directory	06-Oct-2002 11:18	-	
	ages.cc	06-Oct-2002 11:18	1k	
	ages.gdt	06-Oct-2002 11:18	1k	
	ages.gpr	06-Oct-2002 11:18	9k	
	basic00.cc	06-Oct-2002 11:18	1k	
	basic00.gdt	06-Oct-2002 11:18	1k	
	basic00.gpr	06-Oct-2002 11:18	10k	
	basic01.cc	06-Oct-2002 11:18	1k	
	basic01.gdt	06-Oct-2002 11:18	1k	
	basic01.gpr	06-Oct-2002 11:18	9k	
	basic02.cc	06-Oct-2002 11:18	1k	
	basic02.gdt	06-Oct-2002 11:18	1k	
	basic02.gpr	06-Oct-2002 11:18	9k	
	basic03.cc	06-Oct-2002 11:18	1k	
	basic03.gdt	06-Oct-2002 11:18	1k	
	basic03.gpr	06-Oct-2002 11:18	9k	
	basic04.cc	06-Oct-2002 11:18	1k	
	basic04.gdt	06-Oct-2002 11:18	1k	
	basic04.gpr	06-Oct-2002 11:18	9k	
	basic05.cc	06-Oct-2002 11:18	1k	
	basic05.gdt	06-Oct-2002 11:19	1k	
	basic05.gpr	06-Oct-2002 11:19	9k	
	basic06.cc	06-Oct-2002 11:19	1k	

 basic06.gdt	06-Oct-2002 11:19	1k
 basic06.gpr	06-Oct-2002 11:19	9k
 basic07.cc	06-Oct-2002 11:19	1k
 basic07.gdt	06-Oct-2002 11:19	1k
 basic07.gpr	06-Oct-2002 11:19	9k
 basic08.cc	06-Oct-2002 11:19	1k
 basic08.gdt	06-Oct-2002 11:19	1k
 basic08.gpr	06-Oct-2002 11:19	9k
 basic09.cc	06-Oct-2002 11:19	1k
 basic09.gdt	06-Oct-2002 11:19	1k
 basic09.gpr	06-Oct-2002 11:19	9k
 birthday.cc	06-Oct-2002 11:19	1k
 birthday.gdt	06-Oct-2002 11:19	1k
 birthday.gpr	06-Oct-2002 11:19	9k
 birthprt.cc	06-Oct-2002 11:19	1k
 birthprt.gdt	06-Oct-2002 11:19	1k
 birthprt.gpr	06-Oct-2002 11:19	9k
 calc1.cc	06-Oct-2002 11:19	1k
 calc1.gdt	06-Oct-2002 11:19	1k
 calc1.gpr	06-Oct-2002 11:19	9k
 calc1.out	06-Oct-2002 11:19	1k
 code.new	06-Oct-2002 11:19	1k
 code.old	06-Oct-2002 11:19	1k
 common.h	06-Oct-2002 11:19	1k
 common.in	06-Oct-2002 11:19	1k
 count1.cc	06-Oct-2002 11:19	1k
 count1.gdt	06-Oct-2002 11:19	1k

 count1.gpr	06-Oct-2002 11:19	9k
 count2.cc	06-Oct-2002 11:19	1k
 count2.gdt	06-Oct-2002 11:19	1k
 count2.gpr	06-Oct-2002 11:19	9k
 count3.cc	06-Oct-2002 11:19	1k
 count3.gdt	06-Oct-2002 11:19	1k
 count3.gpr	06-Oct-2002 11:19	9k
 count4.cc	06-Oct-2002 11:19	1k
 count4.gdt	06-Oct-2002 11:19	1k
 count4.gpr	06-Oct-2002 11:19	9k
 count5.cc	06-Oct-2002 11:19	1k
 count5.gdt	06-Oct-2002 11:19	1k
 count5.gpr	06-Oct-2002 11:19	9k
 count6.cc	06-Oct-2002 11:19	1k
 count6.gdt	06-Oct-2002 11:19	1k
 count6.gpr	06-Oct-2002 11:19	9k
 cout1.cc	06-Oct-2002 11:19	1k
 cout1.gdt	06-Oct-2002 11:19	1k
 cout1.gpr	06-Oct-2002 11:19	9k
 dangchar.cc	06-Oct-2002 11:19	1k
 dangchar.gdt	06-Oct-2002 11:19	1k
 dangchar.gpr	06-Oct-2002 11:19	9k
 dangchar.out	06-Oct-2002 11:19	1k
 func1.cc	06-Oct-2002 11:19	1k
 func1.gdt	06-Oct-2002 11:19	1k
 func1.gpr	06-Oct-2002 11:19	9k
 inita.cc	06-Oct-2002 11:19	1k

 inita.gdt	06-Oct-2002 11:19	1k
 inita.gpr	06-Oct-2002 11:19	9k
 inita.out	06-Oct-2002 11:19	1k
 initb.cc	06-Oct-2002 11:19	1k
 initb.gdt	06-Oct-2002 11:19	1k
 initb.gpr	06-Oct-2002 11:19	9k
 initb.out	06-Oct-2002 11:19	1k
 initc.cc	06-Oct-2002 11:19	1k
 initc.gdt	06-Oct-2002 11:19	1k
 initc.gpr	06-Oct-2002 11:19	9k
 initc.out	06-Oct-2002 11:19	1k
 initd.cc	06-Oct-2002 11:19	1k
 initd.gdt	06-Oct-2002 11:19	1k
 initd.gpr	06-Oct-2002 11:19	9k
 initd.out	06-Oct-2002 11:19	1k
 inite.cc	06-Oct-2002 11:19	1k
 inite.gdt	06-Oct-2002 11:19	1k
 inite.gpr	06-Oct-2002 11:19	9k
 inite.out	06-Oct-2002 11:19	1k
 initf.cc	06-Oct-2002 11:19	1k
 initf.gdt	06-Oct-2002 11:19	1k
 initf.gpr	06-Oct-2002 11:19	9k
 initf.out	06-Oct-2002 11:19	1k
 invent1.cc	06-Oct-2002 11:19	1k
 invent1.h	06-Oct-2002 11:19	1k
 invent2.cc	06-Oct-2002 11:19	1k
 invent2.h	06-Oct-2002 11:19	1k

 item1.cc	06-Oct-2002 11:19	1k
 item1.h	06-Oct-2002 11:19	1k
 item2.cc	06-Oct-2002 11:19	1k
 item2.h	06-Oct-2002 11:19	1k
 item4.cc	06-Oct-2002 11:19	1k
 item4.h	06-Oct-2002 11:19	1k
 item5.cc	06-Oct-2002 11:19	1k
 item5.h	06-Oct-2002 11:19	1k
 item6.cc	06-Oct-2002 11:19	2k
 item6.h	06-Oct-2002 11:19	1k
 itemtst1.cc	06-Oct-2002 11:19	1k
 itemtst1.gdt	06-Oct-2002 11:19	1k
 itemtst1.gpr	06-Oct-2002 11:19	9k
 itemtst2.cc	06-Oct-2002 11:19	1k
 itemtst2.gdt	06-Oct-2002 11:19	1k
 itemtst2.gpr	06-Oct-2002 11:19	11k
 itemtst2.out	06-Oct-2002 11:19	5k
 itemtst3.cc	06-Oct-2002 11:19	1k
 itemtst3.gdt	06-Oct-2002 11:19	1k
 itemtst3.gpr	06-Oct-2002 11:19	10k
 itemtst4.cc	06-Oct-2002 11:19	2k
 itemtst4.gdt	06-Oct-2002 11:19	1k
 itemtst4.gpr	06-Oct-2002 11:19	10k
 itemtst5.cc	06-Oct-2002 11:19	2k
 itemtst5.gdt	06-Oct-2002 11:19	1k
 itemtst5.gpr	06-Oct-2002 11:19	11k
 itemtst6.cc	06-Oct-2002 11:19	2k

 itemtst6.gdt	06-Oct-2002 11:19	1k
 itemtst6.gpr	06-Oct-2002 11:19	11k
 mkallnru.out	06-Oct-2002 11:19	19k
 mkalltr.ouu	06-Oct-2002 11:19	4k
 morbas00.cc	06-Oct-2002 11:19	1k
 morbas00.gdt	06-Oct-2002 11:19	1k
 morbas00.gpr	06-Oct-2002 11:19	9k
 morbas01.cc	06-Oct-2002 11:19	1k
 morbas01.gdt	06-Oct-2002 11:19	1k
 morbas01.gpr	06-Oct-2002 11:19	9k
 morbas02.cc	06-Oct-2002 11:19	1k
 morbas02.gdt	06-Oct-2002 11:19	1k
 morbas02.gpr	06-Oct-2002 11:19	9k
 morbas03.cc	06-Oct-2002 11:19	1k
 morbas03.gdt	06-Oct-2002 11:19	1k
 morbas03.gpr	06-Oct-2002 11:19	9k
 morbas04.cc	06-Oct-2002 11:19	1k
 morbas04.gdt	06-Oct-2002 11:19	1k
 morbas04.gpr	06-Oct-2002 11:19	9k
 nofunc.cc	06-Oct-2002 11:19	1k
 nofunc.gdt	06-Oct-2002 11:19	1k
 nofunc.gpr	06-Oct-2002 11:19	9k
 pump1.cc	06-Oct-2002 11:19	1k
 pump1.gdt	06-Oct-2002 11:19	1k
 pump1.gpr	06-Oct-2002 11:19	9k
 pump1a.cc	06-Oct-2002 11:19	1k
 pump1a.gdt	06-Oct-2002 11:19	1k

	pump1a.gpr	06-Oct-2002	11:19	9k
	pump2.cc	06-Oct-2002	11:19	1k
	pump2.gdt	06-Oct-2002	11:19	1k
	pump2.gpr	06-Oct-2002	11:19	9k
	readme.dif	06-Oct-2002	11:19	7k
	readme.txt	06-Oct-2002	11:19	10k
	scopclas.cc	06-Oct-2002	11:19	1k
	scopclas.gdt	06-Oct-2002	11:19	1k
	scopclas.gpr	06-Oct-2002	11:19	9k
	scopclas.out	06-Oct-2002	11:19	1k
	shop2.in	06-Oct-2002	11:19	3k
	shop3.in	06-Oct-2002	11:19	3k
	strcmp.cc	06-Oct-2002	11:19	2k
	strcmp.gdt	06-Oct-2002	11:19	1k
	strcmp.gpr	06-Oct-2002	11:19	9k
	strex1.cc	06-Oct-2002	11:19	1k
	strex1.gdt	06-Oct-2002	11:19	1k
	strex1.gpr	06-Oct-2002	11:19	9k
	strex2.cc	06-Oct-2002	11:19	1k
	strex2.gdt	06-Oct-2002	11:19	1k
	strex2.gpr	06-Oct-2002	11:19	9k
	strex3.cc	06-Oct-2002	11:19	1k
	strex3.gdt	06-Oct-2002	11:19	1k
	strex3.gpr	06-Oct-2002	11:19	9k
	strex5.cc	06-Oct-2002	11:19	1k
	strex5.gdt	06-Oct-2002	11:19	1k
	strex5.gpr	06-Oct-2002	11:19	9k

 strex6.cc	06-Oct-2002 11:19	1k
 strex6.gdt	06-Oct-2002 11:19	1k
 strex6.gpr	06-Oct-2002 11:19	9k
 string1.cc	06-Oct-2002 11:19	1k
 string1.h	06-Oct-2002 11:19	1k
 string3.cc	06-Oct-2002 11:19	1k
 string3.h	06-Oct-2002 11:19	1k
 string4.cc	06-Oct-2002 11:19	1k
 string4.h	06-Oct-2002 11:19	1k
 string5.cc	06-Oct-2002 11:19	2k
 string5.h	06-Oct-2002 11:19	1k
 string5a.cc	06-Oct-2002 11:19	2k
 string5x.cc	06-Oct-2002 11:19	2k
 string5x.out	06-Oct-2002 11:19	1k
 string5y.cc	06-Oct-2002 11:19	1k
 string5y.out	06-Oct-2002 11:19	1k
 string6.cc	06-Oct-2002 11:19	3k
 string6.h	06-Oct-2002 11:19	1k
 strsort1.cc	06-Oct-2002 11:19	1k
 strtst1.cc	06-Oct-2002 11:19	1k
 strtst3.cc	06-Oct-2002 11:19	1k
 strtst3a.cc	06-Oct-2002 11:19	1k
 strtst3a.err	06-Oct-2002 11:19	1k
 strtst4.cc	06-Oct-2002 11:19	1k
 strtst5.cc	06-Oct-2002 11:19	1k
 strtst5x.cc	06-Oct-2002 11:19	1k
 test.cc	06-Oct-2002 11:19	1k

 test.gdt	06-Oct-2002 11:19	1k
 test.gpr	06-Oct-2002 11:19	9k
 testpare.cc	06-Oct-2002 11:19	1k
 testpare.gdt	06-Oct-2002 11:19	1k
 testpare.gpr	06-Oct-2002 11:19	9k
 vect1.cc	06-Oct-2002 11:19	1k
 vect1.gdt	06-Oct-2002 11:19	1k
 vect1.gpr	06-Oct-2002 11:19	9k
 vect2.cc	06-Oct-2002 11:19	1k
 vect2.gdt	06-Oct-2002 11:19	1k
 vect2.gpr	06-Oct-2002 11:19	9k
 vect2a.cc	06-Oct-2002 11:19	1k
 vect2a.gdt	06-Oct-2002 11:19	1k
 vect2a.gpr	06-Oct-2002 11:19	9k
 vect3.cc	06-Oct-2002 11:19	1k
 vect3.gdt	06-Oct-2002 11:19	1k
 vect3.gpr	06-Oct-2002 11:19	9k
 vector.h	06-Oct-2002 11:19	3k
 wassert.h	06-Oct-2002 11:19	1k

Index of /whos/code

	Name	Last modified	Size	Description
	Parent Directory	06-Oct-2002 11:18	-	
	inita.out	06-Oct-2002 11:19	1k	
	initb.out	06-Oct-2002 11:19	1k	
	initc.out	06-Oct-2002 11:19	1k	
	initd.out	06-Oct-2002 11:19	1k	
	inite.out	06-Oct-2002 11:19	1k	
	initf.out	06-Oct-2002 11:19	1k	
	common.in	06-Oct-2002 11:19	1k	
	dangchar.out	06-Oct-2002 11:19	1k	
	test.cc	06-Oct-2002 11:19	1k	
	testpare.cc	06-Oct-2002 11:19	1k	
	string5y.cc	06-Oct-2002 11:19	1k	
	morbas02.cc	06-Oct-2002 11:19	1k	
	strtst3a.cc	06-Oct-2002 11:19	1k	
	scopclas.out	06-Oct-2002 11:19	1k	
	strtst1.cc	06-Oct-2002 11:19	1k	
	string5x.out	06-Oct-2002 11:19	1k	
	cout1.cc	06-Oct-2002 11:19	1k	
	calc1.out	06-Oct-2002 11:19	1k	
	strtst3.cc	06-Oct-2002 11:19	1k	
	basic02.cc	06-Oct-2002 11:18	1k	
	string5y.out	06-Oct-2002 11:19	1k	
	basic00.cc	06-Oct-2002 11:18	1k	

 basic01.cc	06-Oct-2002	11:18	1k
 string1.h	06-Oct-2002	11:19	1k
 strtst4.cc	06-Oct-2002	11:19	1k
 morbas03.cc	06-Oct-2002	11:19	1k
 count6.cc	06-Oct-2002	11:19	1k
 itemtst1.cc	06-Oct-2002	11:19	1k
 string3.h	06-Oct-2002	11:19	1k
 count2.cc	06-Oct-2002	11:19	1k
 count5.cc	06-Oct-2002	11:19	1k
 count1.cc	06-Oct-2002	11:19	1k
 strex6.cc	06-Oct-2002	11:19	1k
 count4.cc	06-Oct-2002	11:19	1k
 count3.cc	06-Oct-2002	11:19	1k
 string4.h	06-Oct-2002	11:19	1k
 strex3.cc	06-Oct-2002	11:19	1k
 basic05.cc	06-Oct-2002	11:18	1k
 invent1.h	06-Oct-2002	11:19	1k
 strtst5x.cc	06-Oct-2002	11:19	1k
 birthday.cc	06-Oct-2002	11:19	1k
 strtst5.cc	06-Oct-2002	11:19	1k
 item1.h	06-Oct-2002	11:19	1k
 basic03.cc	06-Oct-2002	11:18	1k
 strex2.cc	06-Oct-2002	11:19	1k
 strtst3a.err	06-Oct-2002	11:19	1k
 item2.h	06-Oct-2002	11:19	1k
 morbas01.cc	06-Oct-2002	11:19	1k
 strex5.cc	06-Oct-2002	11:19	1k

 morbas00.cc	06-Oct-2002 11:19	1k
 basic06.cc	06-Oct-2002 11:19	1k
 initb.cc	06-Oct-2002 11:19	1k
 inita.cc	06-Oct-2002 11:19	1k
 inite.cc	06-Oct-2002 11:19	1k
 initf.cc	06-Oct-2002 11:19	1k
 invent2.h	06-Oct-2002 11:19	1k
 birthprt.cc	06-Oct-2002 11:19	1k
 basic04.cc	06-Oct-2002 11:18	1k
 initc.cc	06-Oct-2002 11:19	1k
 initd.cc	06-Oct-2002 11:19	1k
 basic07.cc	06-Oct-2002 11:19	1k
 strex1.cc	06-Oct-2002 11:19	1k
 calc1.cc	06-Oct-2002 11:19	1k
 ages.cc	06-Oct-2002 11:18	1k
 morbas04.cc	06-Oct-2002 11:19	1k
 string5.h	06-Oct-2002 11:19	1k
 item4.h	06-Oct-2002 11:19	1k
 item5.h	06-Oct-2002 11:19	1k
 itemtst2.cc	06-Oct-2002 11:19	1k
 basic08.cc	06-Oct-2002 11:19	1k
 item6.h	06-Oct-2002 11:19	1k
 common.h	06-Oct-2002 11:19	1k
 basic09.cc	06-Oct-2002 11:19	1k
 string6.h	06-Oct-2002 11:19	1k
 pump1.cc	06-Oct-2002 11:19	1k
 item1.cc	06-Oct-2002 11:19	1k

 string1.cc	06-Oct-2002 11:19	1k
 nofunc.cc	06-Oct-2002 11:19	1k
 test.gdt	06-Oct-2002 11:19	1k
 calc1.gdt	06-Oct-2002 11:19	1k
 cout1.gdt	06-Oct-2002 11:19	1k
 func1.gdt	06-Oct-2002 11:19	1k
 inita.gdt	06-Oct-2002 11:19	1k
 initb.gdt	06-Oct-2002 11:19	1k
 initc.gdt	06-Oct-2002 11:19	1k
 initd.gdt	06-Oct-2002 11:19	1k
 inite.gdt	06-Oct-2002 11:19	1k
 initf.gdt	06-Oct-2002 11:19	1k
 pump1.gdt	06-Oct-2002 11:19	1k
 count1.gdt	06-Oct-2002 11:19	1k
 count2.gdt	06-Oct-2002 11:19	1k
 count3.gdt	06-Oct-2002 11:19	1k
 count4.gdt	06-Oct-2002 11:19	1k
 count5.gdt	06-Oct-2002 11:19	1k
 count6.gdt	06-Oct-2002 11:19	1k
 nofunc.gdt	06-Oct-2002 11:19	1k
 strex1.gdt	06-Oct-2002 11:19	1k
 strex3.gdt	06-Oct-2002 11:19	1k
 basic04.gdt	06-Oct-2002 11:18	1k
 basic05.gdt	06-Oct-2002 11:19	1k
 birthday.gdt	06-Oct-2002 11:19	1k
 birthprt.gdt	06-Oct-2002 11:19	1k
 dangchar.gdt	06-Oct-2002 11:19	1k

 morbas02.gdt	06-Oct-2002 11:19	1k
 morbas03.gdt	06-Oct-2002 11:19	1k
 scopclas.gdt	06-Oct-2002 11:19	1k
 testpare.gdt	06-Oct-2002 11:19	1k
 strex2.gdt	06-Oct-2002 11:19	1k
 vect2.gdt	06-Oct-2002 11:19	1k
 vect3.gdt	06-Oct-2002 11:19	1k
 vect2a.gdt	06-Oct-2002 11:19	1k
 basic07.gdt	06-Oct-2002 11:19	1k
 basic08.gdt	06-Oct-2002 11:19	1k
 morbas00.gdt	06-Oct-2002 11:19	1k
 basic06.gdt	06-Oct-2002 11:19	1k
 morbas01.gdt	06-Oct-2002 11:19	1k
 dangchar.cc	06-Oct-2002 11:19	1k
 string3.cc	06-Oct-2002 11:19	1k
 funcl.cc	06-Oct-2002 11:19	1k
 item2.cc	06-Oct-2002 11:19	1k
 wassert.h	06-Oct-2002 11:19	1k
 string4.cc	06-Oct-2002 11:19	1k
 pumpla.cc	06-Oct-2002 11:19	1k
 strsort1.cc	06-Oct-2002 11:19	1k
 vect1.cc	06-Oct-2002 11:19	1k
 pump2.cc	06-Oct-2002 11:19	1k
 code.old	06-Oct-2002 11:19	1k
 vect2.cc	06-Oct-2002 11:19	1k
 itemtst3.cc	06-Oct-2002 11:19	1k
 vect2a.cc	06-Oct-2002 11:19	1k

 strex5.gdt	06-Oct-2002 11:19	1k
 item4.cc	06-Oct-2002 11:19	1k
 pump2.gdt	06-Oct-2002 11:19	1k
 pump1a.gdt	06-Oct-2002 11:19	1k
 strex6.gdt	06-Oct-2002 11:19	1k
 ages.gdt	06-Oct-2002 11:18	1k
 basic09.gdt	06-Oct-2002 11:19	1k
 itemtst3.gdt	06-Oct-2002 11:19	1k
 morbas04.gdt	06-Oct-2002 11:19	1k
 code.new	06-Oct-2002 11:19	1k
 scopclas.cc	06-Oct-2002 11:19	1k
 vect1.gdt	06-Oct-2002 11:19	1k
 vect3.cc	06-Oct-2002 11:19	1k
 strcmp.gdt	06-Oct-2002 11:19	1k
 basic03.gdt	06-Oct-2002 11:18	1k
 basic00.gdt	06-Oct-2002 11:18	1k
 invent1.cc	06-Oct-2002 11:19	1k
 itemtst1.gdt	06-Oct-2002 11:19	1k
 basic01.gdt	06-Oct-2002 11:18	1k
 basic02.gdt	06-Oct-2002 11:18	1k
 itemtst2.gdt	06-Oct-2002 11:19	1k
 itemtst4.gdt	06-Oct-2002 11:19	1k
 itemtst5.gdt	06-Oct-2002 11:19	1k
 itemtst6.gdt	06-Oct-2002 11:19	1k
 item5.cc	06-Oct-2002 11:19	1k
 invent2.cc	06-Oct-2002 11:19	1k
 itemtst4.cc	06-Oct-2002 11:19	2k

 string5x.cc	06-Oct-2002	11:19	2k
 item6.cc	06-Oct-2002	11:19	2k
 itemtst5.cc	06-Oct-2002	11:19	2k
 string5.cc	06-Oct-2002	11:19	2k
 itemtst6.cc	06-Oct-2002	11:19	2k
 strcmp.cc	06-Oct-2002	11:19	2k
 string5a.cc	06-Oct-2002	11:19	2k
 shop2.in	06-Oct-2002	11:19	3k
 vector.h	06-Oct-2002	11:19	3k
 shop3.in	06-Oct-2002	11:19	3k
 string6.cc	06-Oct-2002	11:19	3k
 mkalltr.ouu	06-Oct-2002	11:19	4k
 itemtst2.out	06-Oct-2002	11:19	5k
 readme.dif	06-Oct-2002	11:19	7k
 strex1.gpr	06-Oct-2002	11:19	9k
 strex2.gpr	06-Oct-2002	11:19	9k
 strex3.gpr	06-Oct-2002	11:19	9k
 strex5.gpr	06-Oct-2002	11:19	9k
 strex6.gpr	06-Oct-2002	11:19	9k
 testpare.gpr	06-Oct-2002	11:19	9k
 ages.gpr	06-Oct-2002	11:18	9k
 test.gpr	06-Oct-2002	11:19	9k
 calc1.gpr	06-Oct-2002	11:19	9k
 cout1.gpr	06-Oct-2002	11:19	9k
 func1.gpr	06-Oct-2002	11:19	9k
 inita.gpr	06-Oct-2002	11:19	9k
 initb.gpr	06-Oct-2002	11:19	9k

 initc.gpr	06-Oct-2002	11:19	9k
 initd.gpr	06-Oct-2002	11:19	9k
 inite.gpr	06-Oct-2002	11:19	9k
 initf.gpr	06-Oct-2002	11:19	9k
 pump1.gpr	06-Oct-2002	11:19	9k
 pump2.gpr	06-Oct-2002	11:19	9k
 count1.gpr	06-Oct-2002	11:19	9k
 count2.gpr	06-Oct-2002	11:19	9k
 count3.gpr	06-Oct-2002	11:19	9k
 count4.gpr	06-Oct-2002	11:19	9k
 count5.gpr	06-Oct-2002	11:19	9k
 count6.gpr	06-Oct-2002	11:19	9k
 nofunc.gpr	06-Oct-2002	11:19	9k
 pumpla.gpr	06-Oct-2002	11:19	9k
 basic03.gpr	06-Oct-2002	11:18	9k
 basic04.gpr	06-Oct-2002	11:18	9k
 basic05.gpr	06-Oct-2002	11:19	9k
 basic06.gpr	06-Oct-2002	11:19	9k
 birthday.gpr	06-Oct-2002	11:19	9k
 dangchar.gpr	06-Oct-2002	11:19	9k
 morbas02.gpr	06-Oct-2002	11:19	9k
 morbas03.gpr	06-Oct-2002	11:19	9k
 morbas04.gpr	06-Oct-2002	11:19	9k
 scopclas.gpr	06-Oct-2002	11:19	9k
 basic09.gpr	06-Oct-2002	11:19	9k
 birthprt.gpr	06-Oct-2002	11:19	9k
 strcmp.gpr	06-Oct-2002	11:19	9k

 basic01.gpr	06-Oct-2002	11:18	9k
 basic02.gpr	06-Oct-2002	11:18	9k
 basic07.gpr	06-Oct-2002	11:19	9k
 basic08.gpr	06-Oct-2002	11:19	9k
 itemtst1.gpr	06-Oct-2002	11:19	9k
 vect1.gpr	06-Oct-2002	11:19	9k
 vect2.gpr	06-Oct-2002	11:19	9k
 vect3.gpr	06-Oct-2002	11:19	9k
 vect2a.gpr	06-Oct-2002	11:19	9k
 morbas00.gpr	06-Oct-2002	11:19	9k
 morbas01.gpr	06-Oct-2002	11:19	9k
 basic00.gpr	06-Oct-2002	11:18	10k
 itemtst3.gpr	06-Oct-2002	11:19	10k
 itemtst4.gpr	06-Oct-2002	11:19	10k
 readme.txt	06-Oct-2002	11:19	10k
 itemtst5.gpr	06-Oct-2002	11:19	11k
 itemtst2.gpr	06-Oct-2002	11:19	11k
 itemtst6.gpr	06-Oct-2002	11:19	11k
 mkallnru.out	06-Oct-2002	11:19	19k

Index of /whos/code

	Name	Last modified	Size	Description
	Parent Directory	06-Oct-2002 11:18	-	
	ages.cc	06-Oct-2002 11:18	1k	
	ages.gdt	06-Oct-2002 11:18	1k	
	ages.gpr	06-Oct-2002 11:18	9k	
	basic00.cc	06-Oct-2002 11:18	1k	
	basic00.gdt	06-Oct-2002 11:18	1k	
	basic00.gpr	06-Oct-2002 11:18	10k	
	basic01.cc	06-Oct-2002 11:18	1k	
	basic01.gdt	06-Oct-2002 11:18	1k	
	basic01.gpr	06-Oct-2002 11:18	9k	
	basic02.cc	06-Oct-2002 11:18	1k	
	basic02.gdt	06-Oct-2002 11:18	1k	
	basic02.gpr	06-Oct-2002 11:18	9k	
	basic03.cc	06-Oct-2002 11:18	1k	
	basic03.gdt	06-Oct-2002 11:18	1k	
	basic03.gpr	06-Oct-2002 11:18	9k	
	basic04.cc	06-Oct-2002 11:18	1k	
	basic04.gdt	06-Oct-2002 11:18	1k	
	basic04.gpr	06-Oct-2002 11:18	9k	
	basic05.cc	06-Oct-2002 11:18	1k	
	basic05.gdt	06-Oct-2002 11:19	1k	
	basic05.gpr	06-Oct-2002 11:19	9k	
	basic06.cc	06-Oct-2002 11:19	1k	

 basic06.gdt	06-Oct-2002 11:19	1k
 basic06.gpr	06-Oct-2002 11:19	9k
 basic07.cc	06-Oct-2002 11:19	1k
 basic07.gdt	06-Oct-2002 11:19	1k
 basic07.gpr	06-Oct-2002 11:19	9k
 basic08.cc	06-Oct-2002 11:19	1k
 basic08.gdt	06-Oct-2002 11:19	1k
 basic08.gpr	06-Oct-2002 11:19	9k
 basic09.cc	06-Oct-2002 11:19	1k
 basic09.gdt	06-Oct-2002 11:19	1k
 basic09.gpr	06-Oct-2002 11:19	9k
 birthday.cc	06-Oct-2002 11:19	1k
 birthday.gdt	06-Oct-2002 11:19	1k
 birthday.gpr	06-Oct-2002 11:19	9k
 birthprt.cc	06-Oct-2002 11:19	1k
 birthprt.gdt	06-Oct-2002 11:19	1k
 birthprt.gpr	06-Oct-2002 11:19	9k
 calc1.cc	06-Oct-2002 11:19	1k
 calc1.gdt	06-Oct-2002 11:19	1k
 calc1.gpr	06-Oct-2002 11:19	9k
 calc1.out	06-Oct-2002 11:19	1k
 code.new	06-Oct-2002 11:19	1k
 code.old	06-Oct-2002 11:19	1k
 common.h	06-Oct-2002 11:19	1k
 common.in	06-Oct-2002 11:19	1k
 count1.cc	06-Oct-2002 11:19	1k
 count1.gdt	06-Oct-2002 11:19	1k

 count1.gpr	06-Oct-2002 11:19	9k
 count2.cc	06-Oct-2002 11:19	1k
 count2.gdt	06-Oct-2002 11:19	1k
 count2.gpr	06-Oct-2002 11:19	9k
 count3.cc	06-Oct-2002 11:19	1k
 count3.gdt	06-Oct-2002 11:19	1k
 count3.gpr	06-Oct-2002 11:19	9k
 count4.cc	06-Oct-2002 11:19	1k
 count4.gdt	06-Oct-2002 11:19	1k
 count4.gpr	06-Oct-2002 11:19	9k
 count5.cc	06-Oct-2002 11:19	1k
 count5.gdt	06-Oct-2002 11:19	1k
 count5.gpr	06-Oct-2002 11:19	9k
 count6.cc	06-Oct-2002 11:19	1k
 count6.gdt	06-Oct-2002 11:19	1k
 count6.gpr	06-Oct-2002 11:19	9k
 cout1.cc	06-Oct-2002 11:19	1k
 cout1.gdt	06-Oct-2002 11:19	1k
 cout1.gpr	06-Oct-2002 11:19	9k
 dangchar.cc	06-Oct-2002 11:19	1k
 dangchar.gdt	06-Oct-2002 11:19	1k
 dangchar.gpr	06-Oct-2002 11:19	9k
 dangchar.out	06-Oct-2002 11:19	1k
 func1.cc	06-Oct-2002 11:19	1k
 func1.gdt	06-Oct-2002 11:19	1k
 func1.gpr	06-Oct-2002 11:19	9k
 inita.cc	06-Oct-2002 11:19	1k

 inita.gdt	06-Oct-2002 11:19	1k
 inita.gpr	06-Oct-2002 11:19	9k
 inita.out	06-Oct-2002 11:19	1k
 initb.cc	06-Oct-2002 11:19	1k
 initb.gdt	06-Oct-2002 11:19	1k
 initb.gpr	06-Oct-2002 11:19	9k
 initb.out	06-Oct-2002 11:19	1k
 initc.cc	06-Oct-2002 11:19	1k
 initc.gdt	06-Oct-2002 11:19	1k
 initc.gpr	06-Oct-2002 11:19	9k
 initc.out	06-Oct-2002 11:19	1k
 initd.cc	06-Oct-2002 11:19	1k
 initd.gdt	06-Oct-2002 11:19	1k
 initd.gpr	06-Oct-2002 11:19	9k
 initd.out	06-Oct-2002 11:19	1k
 inite.cc	06-Oct-2002 11:19	1k
 inite.gdt	06-Oct-2002 11:19	1k
 inite.gpr	06-Oct-2002 11:19	9k
 inite.out	06-Oct-2002 11:19	1k
 initf.cc	06-Oct-2002 11:19	1k
 initf.gdt	06-Oct-2002 11:19	1k
 initf.gpr	06-Oct-2002 11:19	9k
 initf.out	06-Oct-2002 11:19	1k
 invent1.cc	06-Oct-2002 11:19	1k
 invent1.h	06-Oct-2002 11:19	1k
 invent2.cc	06-Oct-2002 11:19	1k
 invent2.h	06-Oct-2002 11:19	1k

 item1.cc	06-Oct-2002 11:19	1k
 item1.h	06-Oct-2002 11:19	1k
 item2.cc	06-Oct-2002 11:19	1k
 item2.h	06-Oct-2002 11:19	1k
 item4.cc	06-Oct-2002 11:19	1k
 item4.h	06-Oct-2002 11:19	1k
 item5.cc	06-Oct-2002 11:19	1k
 item5.h	06-Oct-2002 11:19	1k
 item6.cc	06-Oct-2002 11:19	2k
 item6.h	06-Oct-2002 11:19	1k
 itemtst1.cc	06-Oct-2002 11:19	1k
 itemtst1.gdt	06-Oct-2002 11:19	1k
 itemtst1.gpr	06-Oct-2002 11:19	9k
 itemtst2.cc	06-Oct-2002 11:19	1k
 itemtst2.gdt	06-Oct-2002 11:19	1k
 itemtst2.gpr	06-Oct-2002 11:19	11k
 itemtst2.out	06-Oct-2002 11:19	5k
 itemtst3.cc	06-Oct-2002 11:19	1k
 itemtst3.gdt	06-Oct-2002 11:19	1k
 itemtst3.gpr	06-Oct-2002 11:19	10k
 itemtst4.cc	06-Oct-2002 11:19	2k
 itemtst4.gdt	06-Oct-2002 11:19	1k
 itemtst4.gpr	06-Oct-2002 11:19	10k
 itemtst5.cc	06-Oct-2002 11:19	2k
 itemtst5.gdt	06-Oct-2002 11:19	1k
 itemtst5.gpr	06-Oct-2002 11:19	11k
 itemtst6.cc	06-Oct-2002 11:19	2k

 itemtst6.gdt	06-Oct-2002 11:19	1k
 itemtst6.gpr	06-Oct-2002 11:19	11k
 mkallnru.out	06-Oct-2002 11:19	19k
 mkalltr.ouu	06-Oct-2002 11:19	4k
 morbas00.cc	06-Oct-2002 11:19	1k
 morbas00.gdt	06-Oct-2002 11:19	1k
 morbas00.gpr	06-Oct-2002 11:19	9k
 morbas01.cc	06-Oct-2002 11:19	1k
 morbas01.gdt	06-Oct-2002 11:19	1k
 morbas01.gpr	06-Oct-2002 11:19	9k
 morbas02.cc	06-Oct-2002 11:19	1k
 morbas02.gdt	06-Oct-2002 11:19	1k
 morbas02.gpr	06-Oct-2002 11:19	9k
 morbas03.cc	06-Oct-2002 11:19	1k
 morbas03.gdt	06-Oct-2002 11:19	1k
 morbas03.gpr	06-Oct-2002 11:19	9k
 morbas04.cc	06-Oct-2002 11:19	1k
 morbas04.gdt	06-Oct-2002 11:19	1k
 morbas04.gpr	06-Oct-2002 11:19	9k
 nofunc.cc	06-Oct-2002 11:19	1k
 nofunc.gdt	06-Oct-2002 11:19	1k
 nofunc.gpr	06-Oct-2002 11:19	9k
 pump1.cc	06-Oct-2002 11:19	1k
 pump1.gdt	06-Oct-2002 11:19	1k
 pump1.gpr	06-Oct-2002 11:19	9k
 pump1a.cc	06-Oct-2002 11:19	1k
 pump1a.gdt	06-Oct-2002 11:19	1k

	pump1a.gpr	06-Oct-2002	11:19	9k
	pump2.cc	06-Oct-2002	11:19	1k
	pump2.gdt	06-Oct-2002	11:19	1k
	pump2.gpr	06-Oct-2002	11:19	9k
	readme.dif	06-Oct-2002	11:19	7k
	readme.txt	06-Oct-2002	11:19	10k
	scopclas.cc	06-Oct-2002	11:19	1k
	scopclas.gdt	06-Oct-2002	11:19	1k
	scopclas.gpr	06-Oct-2002	11:19	9k
	scopclas.out	06-Oct-2002	11:19	1k
	shop2.in	06-Oct-2002	11:19	3k
	shop3.in	06-Oct-2002	11:19	3k
	strcmp.cc	06-Oct-2002	11:19	2k
	strcmp.gdt	06-Oct-2002	11:19	1k
	strcmp.gpr	06-Oct-2002	11:19	9k
	strex1.cc	06-Oct-2002	11:19	1k
	strex1.gdt	06-Oct-2002	11:19	1k
	strex1.gpr	06-Oct-2002	11:19	9k
	strex2.cc	06-Oct-2002	11:19	1k
	strex2.gdt	06-Oct-2002	11:19	1k
	strex2.gpr	06-Oct-2002	11:19	9k
	strex3.cc	06-Oct-2002	11:19	1k
	strex3.gdt	06-Oct-2002	11:19	1k
	strex3.gpr	06-Oct-2002	11:19	9k
	strex5.cc	06-Oct-2002	11:19	1k
	strex5.gdt	06-Oct-2002	11:19	1k
	strex5.gpr	06-Oct-2002	11:19	9k

 strex6.cc	06-Oct-2002 11:19	1k
 strex6.gdt	06-Oct-2002 11:19	1k
 strex6.gpr	06-Oct-2002 11:19	9k
 string1.cc	06-Oct-2002 11:19	1k
 string1.h	06-Oct-2002 11:19	1k
 string3.cc	06-Oct-2002 11:19	1k
 string3.h	06-Oct-2002 11:19	1k
 string4.cc	06-Oct-2002 11:19	1k
 string4.h	06-Oct-2002 11:19	1k
 string5.cc	06-Oct-2002 11:19	2k
 string5.h	06-Oct-2002 11:19	1k
 string5a.cc	06-Oct-2002 11:19	2k
 string5x.cc	06-Oct-2002 11:19	2k
 string5x.out	06-Oct-2002 11:19	1k
 string5y.cc	06-Oct-2002 11:19	1k
 string5y.out	06-Oct-2002 11:19	1k
 string6.cc	06-Oct-2002 11:19	3k
 string6.h	06-Oct-2002 11:19	1k
 strsort1.cc	06-Oct-2002 11:19	1k
 strtst1.cc	06-Oct-2002 11:19	1k
 strtst3.cc	06-Oct-2002 11:19	1k
 strtst3a.cc	06-Oct-2002 11:19	1k
 strtst3a.err	06-Oct-2002 11:19	1k
 strtst4.cc	06-Oct-2002 11:19	1k
 strtst5.cc	06-Oct-2002 11:19	1k
 strtst5x.cc	06-Oct-2002 11:19	1k
 test.cc	06-Oct-2002 11:19	1k

 test.gdt	06-Oct-2002 11:19	1k
 test.gpr	06-Oct-2002 11:19	9k
 testpare.cc	06-Oct-2002 11:19	1k
 testpare.gdt	06-Oct-2002 11:19	1k
 testpare.gpr	06-Oct-2002 11:19	9k
 vect1.cc	06-Oct-2002 11:19	1k
 vect1.gdt	06-Oct-2002 11:19	1k
 vect1.gpr	06-Oct-2002 11:19	9k
 vect2.cc	06-Oct-2002 11:19	1k
 vect2.gdt	06-Oct-2002 11:19	1k
 vect2.gpr	06-Oct-2002 11:19	9k
 vect2a.cc	06-Oct-2002 11:19	1k
 vect2a.gdt	06-Oct-2002 11:19	1k
 vect2a.gpr	06-Oct-2002 11:19	9k
 vect3.cc	06-Oct-2002 11:19	1k
 vect3.gdt	06-Oct-2002 11:19	1k
 vect3.gpr	06-Oct-2002 11:19	9k
 vector.h	06-Oct-2002 11:19	3k
 wassert.h	06-Oct-2002 11:19	1k

```
/*TITLE "who's afraid" assert header file*/

****keyword-flag*** "%v %f %n" */
/* "1 7-May-96,0:15:00 WASSERT.H" */

****revision-history****/
/*1 WASSERT.H 7-May-96,0:15:00 As of MAINPT. */
****revision-history****/

#ifndef WASSERT_H
#define WASSERT_H

#ifndef NODEBUG
#define wassert(p) ((p) ? (void)0 : (void) WAssertFail( \
    "Assertion failed: %s, file %s, line %d\n", \
    #p, __FILE__, __LINE__ ) )
#define vectorassert(i,c) ((i<c) ? (void)0 : (void) VectorAssertFail(\
    "You have tried to use element %d of a vector which has only %d elements.\n",\
    i,c) )
#else
#define wassert(p)
#define vectorassert(i,c)
#endif

int WAssertFail(char* p_FormatString, char* p_Message,
    char* p_FileName, unsigned p_LineNumber);

int VectorAssertFail(char* p_FormatString, unsigned p_Index, unsigned p_Count);
#endif
```

```
#ifndef VECTOR_H
#define VECTOR_H

#include "common.h"
#include "wassert.h"
typedef unsigned short ArrayIndex;

template <class T>
class vector
{
protected:
    T *m_Data;
    ArrayIndex m_Count;
public:
    vector();
    vector(ArrayIndex p_Count);
    vector(ArrayIndex p_Count, T *p_Data);
    vector(const vector& p_Vector);
short operator == (const vector& p_Vector);
void *GetDataAddress(){return (void *)m_Data;}
vector& operator = (const vector& p_Vector);

T& operator [] (ArrayIndex p_ArrayIndex) const
{
    vectorassert (p_ArrayIndex, m_Count);
    return m_Data[p_ArrayIndex];
}
ArrayIndex size() const {return m_Count;}
void    resize(ArrayIndex p_NewCount);
    ~vector() {delete [] m_Data;}
};

template <class T>
vector<T>::vector()
{
    m_Data = 0;
    m_Count = 0;
}

template <class T>
vector<T>::vector(ArrayIndex p_Count)
{
    m_Data = 0;
    m_Count = p_Count;

    if (m_Count > 0)
    {
        m_Data = new T[m_Count];
        for (ArrayIndex i = 0; i < m_Count; i ++)
            m_Data[i] = T();
    }
}

template <class T>
vector<T>::vector(ArrayIndex p_Count, T *p_Data)
{
    m_Count = p_Count;
    m_Data = 0;

    if (m_Count > 0)
    {
        m_Data = new T[m_Count];
        for (ArrayIndex i = 0; i < m_Count; i ++)
            m_Data[i] = p_Data[i];
    }
}
}
```

```
template <class T>
vector<T>::vector(const vector<T>& p_Vector)
{
    if (this == &p_Vector)
        return;

    m_Count = p_Vector.m_Count;
    m_Data = 0;

    if (m_Count > 0)
    {
        m_Data = new T[m_Count];
        for (ArrayIndex i = 0; i < m_Count; i++)
            m_Data[i] = p_Vector.m_Data[i];
    }
}

template <class T>
vector<T>& vector<T>::operator = (const vector<T>& p_Vector)
{
    if (this == &p_Vector)
        return *this;

    T* OldPtr = m_Data;

    m_Count = p_Vector.m_Count;
    m_Data = 0;

    if (m_Count > 0)
    {
        m_Data = new T[m_Count];
        for (ArrayIndex i = 0; i < m_Count; i++)
            m_Data[i] = p_Vector.m_Data[i];
    }

    delete [] OldPtr;

    return *this;
}

template <class T>
short vector<T>::operator == (const vector<T>& p_Vector)
{
    if (this == &p_Vector)
        return 1;
    if (m_Count != p_Vector.m_Count)
        return 0;
    if (m_Count == 0)
        return 1;
    if (memcmp(m_Data, p_Vector.m_Data, m_Count*sizeof(T)) == 0)
        return 1;
    return 0;
}

template <class T>
void vector<T>::resize(ArrayIndex p_NewCount)
{
    T* TempPtr;
    T* OldPtr = m_Data;
    ArrayIndex i;
    ArrayIndex Smaller;

    Smaller = min(p_NewCount, m_Count);

    m_Count = p_NewCount;
```

```
m_Data = 0;
```

```
if (p_NewCount > 0)
```

```
{
```

```
    TempPtr = new T[p_NewCount];
```

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

```
        TempPtr[i] = OldPtr[i];
```

```
    m_Data = TempPtr;
```

```
}
```

```
delete [] OldPtr;
```

```
}
```

```
#endif
```

```
#include <iostream.h>
#include "vector.h"

int main()
{
    vector<short> Weight(5);
    vector<short> SortedWeight(3);
    short HighestWeight;
    short HighestIndex;
    short i;
    short k;

    cout << "I'm going to ask you to type in five weights, in pounds." << endl;

    for (i = 0; i < 5; )
    {
        cout << "Please type in weight #" << i+1 << ": ";
        cin >> Weight[i];
        if (Weight[i] <= 0)
        {
            cout << "I'm sorry, " << Weight[i] << " is not a valid weight.";
            cout << endl;
        }
        else
            i ++;
    }

    for (i = 0; i < 3; i ++ )
    {
        HighestIndex = 0;
        HighestWeight = 0;
        for (k = 0; k < 5; k ++ )
        {
            if (Weight[k] > HighestWeight)
            {
                HighestWeight = Weight[k];
                HighestIndex = k;
            }
        }
        SortedWeight[i] = HighestWeight;
        Weight[HighestIndex] = 0;
    }

    cout << "The highest weight was: " << SortedWeight[0] << endl;
    cout << "The second highest weight was: " << SortedWeight[1] << endl;
    cout << "The third highest weight was: " << SortedWeight[2] << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "vector.h"

int main()
{
    vector<short> Weight(5);
    vector<short> SortedWeight(3);
    short HighestWeight;
    short HighestIndex;
    short i;
    short k;

    cout << "I'm going to ask you to type in five weights, in pounds." << endl;

    for (i = 0; i < 5; i++)
    {
        cout << "Please type in weight #" << i+1 << ": ";
        cin >> Weight[i];
        if (Weight[i] <= 0)
        {
            cout << "I'm sorry, " << Weight[i] << " is not a valid weight.";
            cout << endl;
        }
    }

    for (i = 0; i < 3; i++)
    {
        HighestWeight = 0;
        HighestIndex = 0;
        for (k = 0; k < 5; k++)
        {
            if (Weight[k] > HighestWeight)
            {
                HighestWeight = Weight[k];
                HighestIndex = k;
            }
        }
        SortedWeight[i] = HighestWeight;
        Weight[HighestIndex] = 0;
    }

    cout << "The highest weight was: " << SortedWeight[0] << endl;
    cout << "The second highest weight was: " << SortedWeight[1] << endl;
    cout << "The third highest weight was: " << SortedWeight[2] << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "vector.h"

int main()
{
    vector<short> Weight(5);
    vector<short> SortedWeight(3);
    short HighestWeight;
    short HighestIndex;
    short i;
    short k;

    cout << "I'm going to ask you to type in five weights, in pounds." << endl;

    for (i = 0; i < 5; i++)
    {
        cout << "Please type in weight #" << i+1 << ": ";
        cin >> Weight[i];
    }

    for (i = 0; i < 3; i++)
    {
        HighestWeight = 0;
        HighestIndex = 0;
        for (k = 0; k < 5; k++)
        {
            if (Weight[k] > HighestWeight)
            {
                HighestWeight = Weight[k];
                HighestIndex = k;
            }
        }
        SortedWeight[i] = HighestWeight;
        Weight[HighestIndex] = 0;
    }

    cout << "The highest weight was: " << SortedWeight[0] << endl;
    cout << "The second highest weight was: " << SortedWeight[1] << endl;
    cout << "The third highest weight was: " << SortedWeight[2] << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "vector.h"

int main()
{
    vector<short> Weight(5);
    vector<short> SortedWeight(3);
    short HighestWeight;
    short HighestIndex;
    short i;
    short k;

    cout << "I'm going to ask you to type in five weights, in pounds." << endl;

    for (i = 0; i < 5; i++)
    {
        cout << "Please type in weight #" << i+1 << ": ";
        cin >> Weight[i];
    }

    for (i = 0; i < 3; i++)
    {
        HighestWeight = 0;
        for (k = 0; k < 5; k++)
        {
            if (Weight[k] > HighestWeight)
            {
                HighestWeight = Weight[k];
                HighestIndex = k;
            }
        }
        SortedWeight[i] = HighestWeight;
        Weight[HighestIndex] = 0;
    }

    cout << "The highest weight was: " << SortedWeight[0] << endl;
    cout << "The second highest weight was: " << SortedWeight[1] << endl;
    cout << "The third highest weight was: " << SortedWeight[2] << endl;

    return 0;
}
```

```
main()
{
    int a;
    int b;

    a = 5;

    if (b = a)
        a = 3;
}
```

```
#include <iostream.h>

int main()
{
cout << "Hello World!" << endl;
}
```

```
#include <iostream.h>
#include "string5.h"

int main()
{
    string x;
    string y;

    x = "ape";
    y = "axes";

    if (x < y)
        cout << x << " comes before " << y << endl;
    else
        cout << x << " doesn't come before " << y << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "string5.h"

int main()
{
    string x;
    string y;

    x = "post";
    y = "poster";

    if (x < y)
        cout << x << " comes before " << y << endl;
    else
        cout << x << " doesn't come before " << y << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "string4.h"

int main()
{
    short len;
    string n("Test");

    len = n.GetLength();

    cout << "The string has " << len << " characters." << endl;

    return 0;
}
```

```
E:\WHOS\CODE> gcc -o strtst3a.o -c -I. -g strtst3a.cc
strtst3a.cc: In function `int main()':
strtst3a.cc:8: member `_m_Length' is a private member of class `string'
```

```
E:\WHOS\CODE>
```

```
E:\WHOS\CODE>goto strtst3a
```

```
E:\WHOS\CODE>gcc -o string3.o -c -I. -g string3.cc
```

```
E:\WHOS\CODE>goto end
```

```
E:\WHOS\CODE>
```

```
#include <iostream.h>
#include "string3.h"

int main()
{
    string n("Test");

    n.m_Length = 12;

    n.Display();

    return 0;
}
```

```
#include <iostream.h>
#include "string3.h"

int main()
{
    string s;
    string n("Test");
    string x;

    s = n;
    n = "My name is: ";

    n.Display();

    return 0;
}
```

```
#include "string1.h"

int main()
{
    string s;
    string n("Test");
    string x;

    s = n;
    n = "My name is: ";

    x = n;
    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"
#include "vector.h"

int main()
{
    vector<string> Name(5);
    vector<string> SortedName(5);
    string FirstName;
    short FirstIndex;
    short i;
    short k;
    string HighestName = "zzzzzzzz";

    cout << "I'm going to ask you to type in five last names." << endl;

    for (i = 0; i < 5; i ++ )
    {
        cout << "Please type in name #" << i+1 << ": ";
        cin >> Name[i];
    }

    for (i = 0; i < 5; i ++ )
    {
        FirstName = HighestName;
        FirstIndex = 0;
        for (k = 0; k < 5; k ++ )
        {
            if (Name[k] < FirstName)
            {
                FirstName = Name[k];
                FirstIndex = k;
            }
        }
        SortedName[i] = FirstName;
        Name[FirstIndex] = HighestName;
    }

    cout << "Here are the names, in alphabetical order: " << endl;
    for (i = 0; i < 5; i ++ )
        cout << SortedName[i] << endl;

    return 0;
}
```

```
#ifndef STRING_H
#define STRING_H
#include <iostream.h>

class string
{
friend ostream& operator << (ostream& s, const string& Str);
friend istream& operator >> (istream& s, string& Str);

public:
    string();
    string(const string& Str);
    string& operator = (const string& Str);
    ~string();

    string(char* p);
    short GetLength();
    bool operator < (const string& Str);
    bool operator == (const string& Str);
    bool operator > (const string& Str);
    bool operator >= (const string& Str);
    bool operator <= (const string& Str);
    bool operator != (const string& Str);

private:
    short m_Length;
    char* m_Data;
};
#endif
```

```
#include <iostream.h>
#include "string.h"
#include "string6.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [ ] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [ ] m_Data;
}

short string::GetLength()
{
    return m_Length-1;
}

bool string::operator < (const string& Str)
{
    short Result;
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    Result = memcmp(m_Data, Str.m_Data, CompareLength);

    if (Result < 0)
        return true;

    if (Result > 0)
        return false;

    if (m_Length < Str.m_Length)
        return true;
}
```

```
    return false;
}

bool string::operator == (const string& Str)
{
    short Result;

    if (m_Length != Str.m_Length)
        return false;

    Result = memcmp(m_Data, Str.m_Data, m_Length);

    if (Result == 0)
        return true;

    return false;
}

ostream& operator << (ostream& s, const string& Str)
{
    short i;
    for (i=0; i < Str.m_Length-1; i++)
        s << Str.m_Data[i];

    return s;
}

istream& operator >> (istream& s, string& Str)
{
    const short BUFLLEN = 256;

    char Buf[BUFLLEN];
    memset(Buf, 0, BUFLLEN);

    if (s.peek() == '\n')
        s.ignore();
    s.getline(Buf, BUFLLEN, '\n');
    Str = Buf;

    return s;
}

bool string::operator > (const string& Str)
{
    short Result;
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    Result = memcmp(m_Data, Str.m_Data, CompareLength);

    if (Result > 0)
        return true;

    if (Result < 0)
        return false;

    if (m_Length > Str.m_Length)
        return true;

    return false;
}

bool string::operator >= (const string& Str)
{
    short Result;
```

```
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    Result = memcmp(m_Data, Str.m_Data, CompareLength);

    if (Result > 0)
        return true;

    if (Result < 0)
        return false;

    if (m_Length >= Str.m_Length)
        return true;

    return false;
}

bool string::operator <= (const string& Str)
{
    short Result;
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    Result = memcmp(m_Data, Str.m_Data, CompareLength);

    if (Result < 0)
        return true;

    if (Result > 0)
        return false;

    if (m_Length <= Str.m_Length)
        return true;

    return false;
}

bool string::operator != (const string& Str)
{
    short Result;

    if (m_Length != Str.m_Length)
        return true;

    Result = memcmp(m_Data, Str.m_Data, m_Length);

    if (Result == 0)
        return false;

    return true;
}
```

```
gcc -o string5y.o -c -I. -g string5y.cc -pedantic-errors
string5y.cc: In function `class istream & operator >>(class istream &, class string &)':
string5y.cc:9: ANSI C++ forbids variable-size array `Buf'
```

```
int main()
{
    short BUFLen = 256;
    char ch;

    char Buf[BUFLen];

    ch = Buf[0];
}
```

```
gcc -c -I. -g string5x.cc
```

```
string5x.cc: In function `class istream & operator >>(class istream &, class string &)':  
string5x.cc:84: uninitialized const `short int BUFLLEN'
```

```
#include <iostream.h>
#include "string.h"
#include "string5.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [] m_Data;
}

short string::GetLength()
{
    return m_Length-1;
}

bool string::operator < (const string& Str)
{
    short Result;
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    Result = memcmp(m_Data, Str.m_Data, CompareLength);

    if (Result < 0)
        return true;

    if (Result > 0)
        return false;

    if (m_Length < Str.m_Length)
        return true;
}
```

```
    return false;
}

ostream& operator << (ostream& s, const string& Str)
{
    for (short i=0; i < Str.m_Length; i++)
        s << Str.m_Data[i];

    return s;
}

istream& operator >> (istream& s, string& Str)
{
    const short BUFLLEN;

    char Buf[BUFLLEN];
    char ch;

    memset(Buf, 0, BUFLLEN);
    cin.get(Buf, BUFLLEN, '\n');
    cin.get(ch);
    Str = Buf;

    return s;
}
```

```
#include <iostream.h>
#include "string.h"
#include "string5.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [ ] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [ ] m_Data;
}

short string::GetLength()
{
    return m_Length-1;
}

bool string::operator < (const string& Str)
{
    short i;
    bool Result;
    bool ResultFound;
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    ResultFound = false;
    for (i = 0; (i < CompareLength) && (ResultFound == false); i++)
    {
        if (m_Data[i] < Str.m_Data[i])
        {
            Result = true;
            ResultFound = true;
        }
    }
}
```

```
        else
            {
                if (m_Data[i] > Str.m_Data[i])
                    {
                        Result = false;
                        ResultFound = true;
                    }
            }
    }

    if (ResultFound == false)
        {
            if (m_Length < Str.m_Length)
                Result = true;
            else
                Result = false;
        }

    return Result;
}

bool string::operator == (const string& Str)
{
    short Result;
    if (m_Length != Str.m_Length)
        return false;

    Result = memcmp(m_Data, Str.m_Data, m_Length);

    if (Result == 0)
        return true;

    return false;
}

ostream& operator << (ostream& s, const string& Str)
{
    short i;
    for (i=0; i < Str.m_Length-1; i++)
        s << Str.m_Data[i];

    return s;
}

istream& operator >> (istream& s, string& Str)
{
    const short BUFLLEN = 256;

    char Buf[BUFLLEN];
    memset(Buf, 0, BUFLLEN);

    if (s.peek() == '\n')
        s.ignore();
    s.getline(Buf, BUFLLEN, '\n');
    Str = Buf;

    return s;
}
```

```
class string
{
friend ostream& operator << (ostream& s, const string& Str);
friend istream& operator >> (istream& s, string& Str);

public:
    string();
    string(const string& Str);
    string& operator = (const string& Str);
    ~string();

    string(char* p);
    short GetLength();
    bool operator < (const string& Str);
    bool operator == (const string& Str);

private:
    short m_Length;
    char* m_Data;
};
```

```
#include <iostream.h>
#include "string.h"
#include "string5.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [ ] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [ ] m_Data;
}

short string::GetLength()
{
    return m_Length-1;
}

bool string::operator < (const string& Str)
{
    short Result;
    short CompareLength;

    if (Str.m_Length < m_Length)
        CompareLength = Str.m_Length;
    else
        CompareLength = m_Length;

    Result = memcmp(m_Data, Str.m_Data, CompareLength);

    if (Result < 0)
        return true;

    if (Result > 0)
        return false;

    if (m_Length < Str.m_Length)
        return true;
}
```

```
    return false;
}

bool string::operator == (const string& Str)
{
    short Result;

    if (m_Length != Str.m_Length)
        return false;

    Result = memcmp(m_Data, Str.m_Data, m_Length);

    if (Result == 0)
        return true;

    return false;
}

ostream& operator << (ostream& s, const string& Str)
{
    short i;

    for (i=0; i < Str.m_Length-1; i++)
        s << Str.m_Data[i];

    return s;
}

istream& operator >> (istream& s, string& Str)
{
    const short BUFLLEN = 256;

    char Buf[BUFLLEN];
    memset(Buf, 0, BUFLLEN);

    if (s.peek() == '\n')
        s.ignore();
    s.getline(Buf, BUFLLEN, '\n');
    Str = Buf;

    return s;
}
```

```
class string
{
public:
    string();
    string(const string& Str);
    string& operator = (const string& Str);
    ~string();

    string(char* p);
    void Display();
    short GetLength();

private:
    short m_Length;
    char* m_Data;
};
```

```
#include <iostream.h>
#include "string.h"
#include "string4.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [ ] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [ ] m_Data;
}

void string::Display()
{
    short i;

    for (i = 0; i < m_Length; i++)
        cout << m_Data[i];
}

short string::GetLength()
{
    return m_Length-1;
}
```

```
class string
{
public:
    string();
    string(const string& Str);
    string& operator = (const string& Str);
    ~string();

    string(char* p);
    void Display();

private:
    short m_Length;
    char* m_Data;
};
```

```
#include <iostream.h>
#include "string.h"
#include "string3.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [ ] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [ ] m_Data;
}

void string::Display()
{
    short i;

    for (i = 0; i < m_Length-1; i++)
        cout << m_Data[i];
}
```

```
class string
{
public:
    string();
    string(const string& Str);
    string& operator = (const string& Str);
    ~string();

    string(char* p);

private:
    short m_Length;
    char* m_Data;
};
```

```
#include <string.h>
#include "string1.h"

string::string()
: m_Length(1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, "", m_Length);
}

string::string(const string& Str)
: m_Length(Str.m_Length),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, Str.m_Data, m_Length);
}

string::string(char* p)
: m_Length(strlen(p) + 1),
  m_Data(new char [m_Length])
{
    memcpy(m_Data, p, m_Length);
}

string& string::operator = (const string& Str)
{
    if (&Str != this)
    {
        delete [ ] m_Data;
        m_Length = Str.m_Length;
        m_Data = new char [m_Length];
        memcpy(m_Data, Str.m_Data, m_Length);
    }
    return *this;
}

string::~~string()
{
    delete [ ] m_Data;
}
```

```
class string
{
public:
    string();
    string& operator = (const string& Str);
private:
    string(char* p);
    short m_Length;
    char* m_Data;
};

int main()
{
    string n;

    n = "My name is: ";

    return 0;
}
```

```
class string
{
public:
    string(const string& Str) {};
    string(char* p) {};
    string& operator = (const string& Str) {};
    ~string() {};
private:
    string();
    short m_Length;
    char* m_Data;
};

int main()
{
    string n("Test");
    string x = n;

    n = "My name is: ";

    return 0;
}
```

```
class string
{
public:
    string();
    string(const string& Str);
    string(char* p);
    string& operator=(const string& Str);
private:
    ~string();
    short m_Length;
    char* m_Data;
};

int main()
{
    string s("Test");

    return 0;
}
```

```
class string
{
public:
    string(const string& Str);
    string(char* p);
    string& operator=(const string& Str);
    ~string();
private:
    string();
    short m_Length;
    char* m_Data;
};

int main()
{
    string s("Test");
    string n;

    n = s;

    return 0;
}
```

```
class string
{
public:
    string();
    string(const string& Str);
    string(char* p);
    string& operator = (const string& Str);
    ~string();
private:
    short m_Length;
    char* m_Data;
};

int main()
{
    string s;
    string n("Test");
    string x;
    short Length;

    Length = n.m_Length;

    s = n;
    n = "My name is: ";

    x = n;
    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"

int main()
{
    string x = "x";
    string xx = "xx";
    string y = "y";
    string yy = "yy";

// testing <
    if (x < x)
        cout << "ERROR: x < x" << endl;
    else
        cout << "OKAY: x NOT < x" << endl;
    if (x < xx)
        cout << "OKAY: x < xx" << endl;
    else
        cout << "ERROR: x NOT < xx" << endl;
    if (x < y)
        cout << "OKAY: x < y" << endl;
    else
        cout << "ERROR: x NOT < y" << endl;

// testing <=
    if (x <= x)
        cout << "OKAY: x <= x" << endl;
    else
        cout << "ERROR: x NOT <= x" << endl;
    if (x <= xx)
        cout << "OKAY: x <= xx" << endl;
    else
        cout << "ERROR: x NOT <= xx" << endl;
    if (x <= y)
        cout << "OKAY: x <= y" << endl;
    else
        cout << "ERROR: x NOT <= y" << endl;

// testing >
    if (y > y)
        cout << "ERROR: y > y" << endl;
    else
        cout << "OKAY: y NOT > y" << endl;
    if (yy > y)
        cout << "OKAY: yy > y" << endl;
    else
        cout << "ERROR: yy NOT > y" << endl;
    if (y > x)
        cout << "OKAY: y > x" << endl;
    else
        cout << "ERROR: y NOT > x" << endl;

// testing >=
    if (y >= y)
        cout << "OKAY: y >= y" << endl;
    else
        cout << "ERROR: y NOT >= y" << endl;
    if (yy >= y)
        cout << "OKAY: yy >= y" << endl;
    else
        cout << "ERROR: yy NOT >= y" << endl;
    if (y >= x)
        cout << "OKAY: y >= x" << endl;
    else
        cout << "ERROR: y NOT >= x" << endl;

// testing ==
```

```
    if (x == x)
        cout << "OKAY: x == x" << endl;
    else
        cout << "ERROR: x NOT == x" << endl;
    if (x == xx)
        cout << "ERROR: x == xx" << endl;
    else
        cout << "OKAY: x NOT == xx" << endl;
    if (x == y)
        cout << "ERROR: x == y" << endl;
    else
        cout << "OKAY: x NOT == y" << endl;

// testing !=
    if (x != x)
        cout << "ERROR: x != x" << endl;
    else
        cout << "OKAY: x NOT != x" << endl;
    if (x != xx)
        cout << "OKAY: x != xx" << endl;
    else
        cout << "ERROR: x NOT != xx" << endl;
    if (x != y)
        cout << "OKAY: x != y" << endl;
    else
        cout << "ERROR: x NOT != y" << endl;

    return 0;
}
```

3-ounce cups
71
259
78
2
Bob's Distribution
2895657951
Ajax-substitute
76
344
87
80
Wholesale Plus
453570903
antihistamines
37
388
37
53
Bob's Distribution
562387144
Arturo sauce
77
116
63
66
Bob's Distribution
4687059245
Brannola
26
184
100
92
Wholesale Plus
8246082980
bread, challah
23
308
12
101
Wholesale Plus
2439398000
bread, rye
68
105
81
29
ABC Dist.
1000557518
bread, white
5
188
99
41
Bob's Distribution
3009738201
breadcrumbs, Italian
28
148
42
72
Wholesale Plus
6465816282
Bufferin
33
289

9
47
Wholesale Plus
1860120756
Carpet Fresh
91
178
93
64
Bob's Distribution
3789078521
chew sticks
63
228
92
84
ABC Dist.
5610409797
Clorox 2
2
262
51
52
ABC Dist.
4302691616
cough drops
46
206
25
99
ABC Dist.
2697340483
dish detergent, auto
6
217
48
27
Bob's Distribution
4898936499
dish detergent, liquid
63
262
52
40
ABC Dist.
9385415630
dog food (with Alpo)
11
334
84
3
Wholesale Plus
7536845964
Dove soap
21
122
1
55
Bob's Distribution
3316910545
envelopes, large
66
263
69
46
Bob's Distribution

818982741
Fantastic refill
36
145
10
77
Wholesale Plus
9287870439
flypaper
40
238
11
60
Bob's Distribution
2076249216
furniture polish, lemon
17
377
88
76
Bob's Distribution
4438609792
Gas-X
27
301
33
80
ABC Dist.
898925662
Glass-Plus
30
170
5
49
Bob's Distribution
2546048047
Kleenex, brown foil
21
358
34
55
Bob's Distribution
7549058862
Kleenex, white foil
8
290
93
63
ABC Dist.
9604241003
Kosher soap
35
145
14
4
Bob's Distribution
2194047997
laundry detergent
35
264
86
84
Wholesale Plus
5382492295
lemon ammonia
67

316
42
71
Wholesale Plus
3398099677
Metamucil (Nutrasweet)
18
226
44
52
Wholesale Plus
8146654317
mixed nuts
23
285
38
31
Bob's Distribution
6808148983
Murphy's oil soap
29
145
37
89
Wholesale Plus
2232652982
napkins
48
157
79
17
Wholesale Plus
7474168406
no-fat Entenmann's
81
161
80
39
ABC Dist.
658595676
paper towels
46
136
72
54
ABC Dist.
1737711547
Pepto-Bismol
56
165
41
91
ABC Dist.
7463546800
plastic forks
75
126
44
41
ABC Dist.
7130263457
plastic spoons
28
395
74
29

Wholesale Plus
6961080258
popcorn cakes (Quaker)
36
230
36
11
ABC Dist.
1215494497
sodas, chocolate
19
123
50
98
Wholesale Plus
9592743275
sodas, cream
22
213
15
53
Wholesale Plus
2815039584
sodas, NoCaf Diet Coke
97
267
70
7
Wholesale Plus
6572590920
sodas, orange
76
309
34
79
Bob's Distribution
1554349707
sodas, root beer
5
255
98
81
Bob's Distribution
8006675702
SOS
67
371
96
81
ABC Dist.
4166487577
sponges
70
220
52
42
Bob's Distribution
1677601629
toilet paper, unscented
11
183
20
91
Wholesale Plus
8490964304
Top Job

37

197

25

89

Wholesale Plus

2180077083

walnuts

61

212

94

53

Wholesale Plus

8605838950

water

33

360

36

1

Bob's Distribution

2595425906

3-ounce cups
71
259
Bob's Distribution
2895657951
Ajax-substitute
77
104
ABC Dist.
8144976072
antihistamines
5
224
Bob's Distribution
7904886261
Arturo sauce
96
361
Bob's Distribution
9495505623
Brannola
52
329
Wholesale Plus
5924505350
bread, challah
30
286
Bob's Distribution
2637964782
bread, rye
83
347
Bob's Distribution
9860958916
bread, white
23
308
Wholesale Plus
2439398000
breadcrumbs, Italian
11
399
Wholesale Plus
157067617
Bufferin
10
131
ABC Dist.
2844879888
Carpet Fresh
30
214
Bob's Distribution
9485730097
chew sticks
40
183
ABC Dist.
1628216044
Clorox 2
41
223
ABC Dist.
3262071273
cough drops

21
156
Wholesale Plus
807158335
dish detergent, auto
91
178
Bob's Distribution
3789078521
dish detergent, liquid
92
289
ABC Dist.
4284562764
dog food (with Alpo)
56
308
ABC Dist.
8348191371
Dove soap
54
374
Wholesale Plus
6779443026
envelopes, large
51
238
Bob's Distribution
4048335347
Fantastic refill
6
173
Wholesale Plus
609197907
flypaper
36
246
Bob's Distribution
4744515566
furniture polish, lemon
63
262
ABC Dist.
9385415630
Gas-X
51
217
Wholesale Plus
7839910737
Glass-Plus
75
278
Bob's Distribution
187583273
Kleenex, brown foil
7
132
ABC Dist.
1282433169
Kleenex, white foil
54
296
ABC Dist.
8274154401
Kosher soap
19

303
Bob's Distribution
3570245420
laundry detergent
70
378
ABC Dist.
896453021
lemon ammonia
40
238
Bob's Distribution
2076249216
Metamucil (NutraSweet)
10
276
ABC Dist.
9276116987
mixed nuts
44
182
Bob's Distribution
7506887254
Murphy's oil soap
67
177
Bob's Distribution
309508989
napkins
79
189
Bob's Distribution
4804723528
no-fat Entenmann's
34
113
ABC Dist.
2060148242
paper towels
59
326
Wholesale Plus
3310192788
Pepto-Bismol
8
290
ABC Dist.
9604241003
plastic forks
92
285
Wholesale Plus
1492434772
plastic spoons
22
397
Bob's Distribution
288813042
popcorn cakes (Quaker)
55
376
ABC Dist.
4064253824
sodas, chocolate
83
301

Bob's Distribution
9967772189
sodas, cream
50
223
Wholesale Plus
1790869528
sodas, NoCaf Diet Coke
54
344
Wholesale Plus
4275354091
sodas, orange
23
285
Bob's Distribution
6808148983
sodas, root beer
37
190
Wholesale Plus
1503129286
SOS
22
275
Wholesale Plus
8759736345
sponges
19
305
ABC Dist.
6139374741
toilet paper, unscented
16
342
ABC Dist.
9567620261
Top Job
6
337
ABC Dist.
4635837960
walnuts
12
152
Wholesale Plus
7148104811
water
56
165
ABC Dist.
7463546800

```
count1 = 1  
count2 = 6  
count3 = -32768  
count4 = 23  
count5 = 1  
count6 = 10
```

```
count1 = 2  
count2 = 7  
count3 = -32767  
count4 = 23  
count5 = 2  
count6 = 11
```

```
#include <iostream.h>

short count1; // A global variable, not explicitly initialized
short count2 = 5; // A global variable, explicitly initialized

short func1()
{
    short count3; // A local auto variable, not explicitly initialized
    short count4 = 22; // A local auto variable, explicitly initialized
    static short count5; // A local static variable, not explicitly initialized
    static short count6 = 9; // A local static variable, explicitly initialized

    count1++; // Incrementing the global variable count1 .
    count2++; // Incrementing the global variable count2 .
    count3++; // Incrementing the local uninitialized auto variable count3 .
    count4++; // Incrementing the local auto variable count4 .
    count5++; // Incrementing the local static variable count5 .
    count6++; // Incrementing the local static variable count6 .

    cout << "count1 = " << count1 << endl;
    cout << "count2 = " << count2 << endl;
    cout << "count3 = " << count3 << endl;
    cout << "count4 = " << count4 << endl;
    cout << "count5 = " << count5 << endl;
    cout << "count6 = " << count6 << endl;
    cout << endl;

    return 0;
}

int main()
{
    func1();
    func1();

    return 0;
}
```

Compiler setup instructions

=====

Here are the instructions on setting up the DJGPP compiler, copyright by DJ Delorie, from the CD-ROM in the back of this book. The CD-ROM includes the ZIP files containing the source code for this compiler and its associated programs, as required by the terms of the license under which it is reproduced for this book; you can find those files in the \djgppzip directory. You can get more recent files, or the complete set of files for the DJGPP project, by visiting www.delorie.com.

These instructions will work for DOS, as well as for DOS sessions under Windows 3.1(tm) and Windows 95(tm). I have also heard that they have been used successfully with some modifications for Windows NT(tm); you can visit my WWW site, listed at the end of this document, for details on those modifications, but I can't answer any questions about them because I'm not using Windows NT. They haven't been tested with other operating systems such as OS/2(tm). If you don't have a printed copy of these instructions, you should print them out first.

However, before starting, I have a few comments that I will share with you in an attempt to prevent some problems that readers of my previous books have run into.

1. Please follow the instructions exactly. I have found that when readers of my previous books have reported having difficulty in setting up the compiler, in almost all cases these problems were caused by their not following the instructions as they were written.
2. I have made these instructions as clear as possible, but they still assume that you have a reasonable knowledge of DOS. if you don't know enough about DOS to follow these instructions, I suggest that you get a book like "DOS for Dummies" or find someone nearby to help you. I'll be happy to try to help you if you have technical difficulties getting the compiler to work, but *please* don't write to me to ask for instructions on how to use DOS.

Now let's get to the instructions.

1. To copy the compiler from the CD-ROM to your hard disk:
(Warning: the compiler requires approximately 20 MB of disk space!)
 - a. Make a directory on your hard disk, say c:\djgpp
 - b. Use XCOPY to copy all the files in the CD-ROM directory \djgpp and below to your hard disk. If your CD-ROM drive is drive d: and you want to install to c:\djgpp, then you can type:

```
xcopy d:\djgpp c:\djgpp /s
```

Make sure that the letter of your CD-ROM is the same as the first drive letter in the xcopy line (d: in the example), and the letter of the drive where you want to install the files is the same as the second drive letter in the xcopy line (c: in the example).

Note: Do *not* use "drag and drop" to copy these files from the CD-ROM, as that will leave them in an unmodifiable state on your hard disk. This will interfere with the workings of the compiler.

- c. Add the following lines to the end of your "autoexec.bat" file, (but before the "win" line if you are running Windows 3.1). These lines assume that you want to use c: to hold temporary files, and that you have installed the compiler on drive C. Make sure that the drive letter in the line "setdjgpprun=c:" matches the drive letter where you have installed the compiler in step 1b.

(To save typing, you might want to cut and paste them from this file, which is \readme.txt on the CD-ROM).

```
set djgpptmp=c:
set djgpprun=c:
set DJGPP=%djgpprun%\DJGPP\DJGPP.ENV
set PATH=%djgpprun%\DJGPP\BIN;%PATH%
call setdjgpp %djgpprun%\djgpp %djgpprun%/djgpp
```

If you are running Windows 95 and don't have an "autoexec.bat", you can create one and put the above lines in it. Alternatively, you can create a batch file that you can call "setdos.bat" (for example), containing the entries from the "readme.txt" file that would go into "autoexec.bat" if you were running DOS and put it in whatever directory you like, like "c:\util". Then right-click on your "MS-DOS Prompt" icon, and select "Properties" and then the "Program" tab. Type the full name of that batch file ("c:\util\setdos.bat") into the "Batch file" entry in that dialog box, and it will be executed whenever you start an MS-DOS session through that icon.

- d. Under Windows 95, skip to step 3. Otherwise, make sure that your "config.sys" file contains lines that look like the following:

```
DEVICE=C:\DOS\HIMEM.SYS
DOS=HIGH
FILES=30
```

If "config.sys" already has lines in it containing "himem.sys", "dos=high", and "files=nn" (when nn is at least 30), don't add the lines above. Also note that "himem.sys" may be in a different directory than "c:\dos"; if so, make sure that line refers to the directory where "himem.sys" actually is.

- e. Go to step 3

2. If you prefer to run the compiler from the CD-ROM instead of copying it to your hard disk, assuming that your CD-ROM is drive d: and that you want to use c: to hold temporary files:

- a. Add the following lines to the end of your "autoexec.bat" file (but before the "win" line if you are running Windows 3.1). Make sure to set the drive letter in the line "set djgpprun=d:" to the letter of your CD-ROM drive.
(To save typing, you might want to cut and paste them from this file, which is \readme.txt on the CD-ROM).

```
set djgpptmp=c:
set djgpprun=d:
set DJGPP=%djgpprun%\DJGPP\DJGPP.ENV
set PATH=%djgpprun%\DJGPP\BIN;%PATH%
call setdjgpp %djgpprun%\djgpp %djgpprun%/djgpp
```

If you are running Windows 95 and don't have an "autoexec.bat", you can create one and put the above lines in it. Alternatively, you can create a batch file that you can call "setdos.bat" (for example), containing the entries from the "readme.txt" file that would go into "autoexec.bat" if you were running DOS and put it in whatever directory you like, like "c:\util". Then right-click on your "MS-DOS Prompt" icon, and select "Properties" and then the "Program" tab. Type the full name of that batch file ("c:\util\setdos.bat") into the "Batch file" entry in that dialog box, and it will be executed whenever you start an MS-DOS session through that icon.

- b. Under Windows 95, skip to step 3. Otherwise, make sure that your "config.sys" file contains lines that look like the following:

```
DEVICE=C:\DOS\HIMEM.SYS
```

```
DOS=HIGH
FILES=30
```

If "config.sys" already has lines in it containing "himem.sys", "dos=high", and "files=nn" (when nn is at least 30), don't add the lines above. Also note that "himem.sys" may be in a different directory than "c:\dos"; if so, make sure that line refers to the directory where "himem.sys" actually is.

3. After making either of the above sets of changes, reboot so that they will take effect.

4. To check whether the compiler has been set up correctly, run the go32-v2.exe program by typing the following command at a DOS prompt:

```
go32-v2
```

The last two lines of its output should report how much DPMI memory and swap space DJGPP can use on your system, like this:

```
DPMI memory available: 8020 Kb
DPMI swap space available: 240 Kb
```

If you don't get output that looks like this, with the exception of different numbers, check that you've followed the instructions exactly.

Copying and compiling the sample programs

=====

After you have set up the compiler as shown above, you should copy the sample programs to your hard disk. They are in the directory d:\whos\code (assuming that your CD-ROM is drive d:).

1. To copy the sample files to your hard disk, change to the root directory on the CD-ROM and run the batch file "copysamp", supplying a parameter to indicate where you want the sample files to go. For example, if you want the sample files to be placed under the directory "c:\whos", type:

```
d:
cd\
copysamp c:\whos
```

2. Now you can compile any of the sample programs by changing to the directory "c:\whos\normal" and running the batch file "mk.bat", giving the name of the sample program as a parameter. For example, to compile "itemtst1", type

```
mk itemtst1
```

The compiled version will be placed in the "\whos\normal" directory. You can execute it by typing its name at the DOS prompt. For example, to run "itemtst1", type

```
itemtst1
```

The results of running itemtst1 should look like this:

```
Name: Chunky Chicken
Number in stock: 32
Price: 129
Distributor: Bob's Distribution
UPC: 123456789
```

Writing and compiling your own programs

=====

Change to the "\\whos\\code" directory on the drive where you installed the compiler.

Use EDIT or Notepad to create a text file containing the source code for your program, giving it the extension ".cc". In other words, if you want your program to be called "party", then name this file "party.cc".

To compile your program, switch to the "\\whos\\normal" directory and type "mk party", substituting the name of your file for "party". Note: do **not** add the ".cc" to the end of the file name.

To run your program normally, make sure you are in the "\\whos\\normal" directory, and then type the name of the program, without the extension. In this case, you would just type "party".

To run your program under the debugger, make sure you are in the "\\whos\\normal" directory, and then type "trace party" (substituting the name of your program for "party"). Again, do **not** add the ".cc" to the end of the file name. Instructions for using the debugger can be found in the text.

Further assistance

=====

If you have any problems setting up the compiler or compiling the sample code, or have any other questions, you might want to check my web page for updates to the instructions or sample code. At the moment, that address is:

<http://www.koyote.com/users/stheller>

If you can't reach that page, or you have questions that aren't answered by it, you can email me at:

steve_heller@compuserve.com

Comparing files README.TXT and \introcpp\code\readme.txt

***** README.TXT

Here are the instructions on setting up the DJGPP compiler, copyright by DJ Delorie, from the CD-ROM in the back of this book. The CD-ROM includes the ZIP files containing the source code for this compiler and its associated programs, as required by the terms of the license under which it is reproduced for this book; you can find those files in the \djgppzip directory. You can get more recent files, or the complete set of files for the DJGPP project, by visiting www.delorie.com.

These instructions will work for DOS, as well as for DOS sessions under Windows 3.1(tm) and Windows 95(tm). I have also heard that they have been used successfully with some modifications for Windows NT(tm); you can visit my WWW site, listed at the end of this document, for details on those modifications, but I can't answer any questions about them because I'm not using Windows NT. They haven't been tested with other operating systems such as OS/2(tm). If you don't have a printed copy of these instructions, you should print them out first.

***** \introcpp\code\readme.txt

Here are the instructions on setting up the DJGPP compiler from the CD-ROM in the back of this book. They will work for DOS, as well as for DOS sessions under Windows 3.1 and Windows 95, but haven't been tested with other operating systems such as OS/2. If you don't have a printed copy of these instructions, you should print them out first.

***** README.TXT

However, before starting, I have a few comments that I will share with you in an attempt to prevent some problems that readers of my previous books have run into.

1. Please follow the instructions exactly. I have found that when readers of my previous books have reported having difficulty in setting up the compiler, in almost all cases these problems were caused by their not following the instructions as they were written.
2. I have made these instructions as clear as possible, but they still assume that you have a reasonable knowledge of DOS. if you don't know enough about DOS to follow these instructions, I suggest that you get a book like "DOS for Dummies" or find someone nearby to help you. I'll be happy to try to help you if you have technical difficulties getting the compiler to work, but *please* don't write to me to ask for instructions on how to use DOS.

***** \introcpp\code\readme.txt

If you want to copy the compiler from the CD-ROM to your hard disk (recommended, if you have at least 20 MB free), start with step 1. You can also install the compiler so that it will run from the CD-ROM without copying it to your hard disk. However, if you set up the compiler to run from the CD-ROM, the CD-ROM drive will be in your path. That means that you'll get "invalid path" messages any time that drive does not have a CD-ROM in it. If you still want to do this, start with step 2.

***** README.TXT

Now let's get to the instructions.

1. To copy the compiler from the CD-ROM to your hard disk:

***** \introcpp\code\readme.txt

1. To copy the compiler from the CD-ROM to your hard disk:

***** README.TXT

If you are running Windows 95 and don't have an "autoexec.bat", you can create one and put the above lines in it. Alternatively, you can create a batch file that you can call "setdos.bat" (for example), containing the entries from the "readme.txt" file that would go into "autoexec.bat" if you were running DOS and put it in whatever directory you like, like "c:\util". Then right-click on your "MS-DOS Prompt" icon, and select "Properties" and then the "Program" tab. Type the full name of that batch file ("c:\util\setdos.bat") into the "Batch file" entry in that dialog box, and it will be executed whenever you start an MS-DOS session through that icon.

***** \introcpp\code\readme.txt

If you are running Windows 95 and don't have an "autoexec.bat", you can create one and put the above lines in it, or you can follow the procedure under 1c above to create a startup batch file for use when running a DOS session. (To save typing, you might want to cut and paste them from this file, which is \readme.txt on the CD-ROM).

***** README.TXT

sample programs to your hard disk. They are in the directory d:\whos\code (assuming that your CD-ROM is drive d:).

***** \introcpp\code\readme.txt

sample programs to your hard disk. They are in the directory d:\introcpp\code (assuming that your CD-ROM is drive d:).

***** README.TXT

indicate where you want the sample files to go. For example, if you want the sample files to be placed under the directory "c:\whos", type:

***** \introcpp\code\readme.txt

indicate where you want the sample files to go. For example, if you want the sample files to be placed under the directory "c:\introcpp", type:

***** README.TXT

```
cd\  
copysamp c:\whos
```

***** \introcpp\code\readme.txt

```
cd\  
copysamp c:\introcpp
```

***** README.TXT

2. Now you can compile any of the sample programs by changing to the directory "c:\whos\normal" and running the batch file "mk.bat", giving the name of the sample program as a parameter. For example, to

***** \introcpp\code\readme.txt

2. Now you can compile any of the sample programs by changing to the directory "c:\introcpp\normal" and running the batch file "mk.bat", giving the name of the sample program as a parameter. For example, to

***** README.TXT

The compiled version will be placed in the "\whos\normal" directory. You can execute it by typing its name at the DOS prompt.
***** \introcpp\code\readme.txt

The compiled version will be placed in the "\introcpp\normal" directory. You can execute it by typing its name at the DOS prompt.

***** README.TXT

Change to the "\whos\code" directory on the drive where you installed the compiler.
***** \introcpp\code\readme.txt

Change to the "\introcpp\code" directory on the drive where you installed the compiler.

***** README.TXT

To compile your program, switch to the "\whos\normal" directory and type "mk party", substituting the name of your file for "party". Note:
***** \introcpp\code\readme.txt

To compile your program, switch to the "\introcpp\normal" directory and type "mk party", substituting the name of your file for "party". Note:

***** README.TXT

To run your program normally, make sure you are in the "\whos\normal" directory, and then type the name of the program, without the
***** \introcpp\code\readme.txt

To run your program normally, make sure you are in the "\introcpp\normal" directory, and then type the name of the program, without the

***** README.TXT

To run your program under the debugger, make sure you are in the "\whos\normal" directory, and then type "trace party" (substituting the name of your program for "party"). Again, do *not* add the ".cc" to
***** \introcpp\code\readme.txt

To run your program under the debugger, make sure you are in the "\introcpp\normal" directory, and then type "trace party" (substituting the name of your program for "party"). Again, do *not* add the ".cc" to

***** README.TXT

***** \introcpp\code\readme.txt

```
#include <iostream.h>

int main()
{
    short CurrentWeight;
    short HighestWeight;
    short SecondHighestWeight;

    cout << "Please enter the first weight: ";
    cin >> CurrentWeight;
    HighestWeight = CurrentWeight;
    SecondHighestWeight = 0;
    cout << "Current weight " << CurrentWeight << endl;
    cout << "Highest weight " << HighestWeight << endl;

    while (CurrentWeight > 0)
    {
        cout << "Please enter the next weight: ";
        cin >> CurrentWeight;
        if (CurrentWeight > HighestWeight)
        {
            SecondHighestWeight = HighestWeight;
            HighestWeight = CurrentWeight;
        }
        else
        {
            if (CurrentWeight > SecondHighestWeight)
                SecondHighestWeight = CurrentWeight;
        }
        cout << "Current weight " << CurrentWeight << endl;
        cout << "Highest weight " << HighestWeight << endl;
        cout << "Second highest weight " << SecondHighestWeight << endl;
    }

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short CurrentWeight;
    short HighestWeight;
    short SecondHighestWeight;

    cout << "Please enter the first weight: ";
    cin >> CurrentWeight;
    HighestWeight = CurrentWeight;
    SecondHighestWeight = 0;
    cout << "Current weight " << CurrentWeight << endl;
    cout << "Highest weight " << HighestWeight << endl;

    while (CurrentWeight > 0)
    {
        cout << "Please enter the next weight: ";
        cin >> CurrentWeight;
        if (CurrentWeight > HighestWeight)
        {
            SecondHighestWeight = HighestWeight;
            HighestWeight = CurrentWeight;
        }
        cout << "Current weight " << CurrentWeight << endl;
        cout << "Highest weight " << HighestWeight << endl;
        cout << "Second highest weight " << SecondHighestWeight << endl;
    }

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short CurrentWeight;
    short HighestWeight;

    cout << "Please enter the first weight: ";
    cin >> CurrentWeight;
    HighestWeight = CurrentWeight;
    cout << "Current weight " << CurrentWeight << endl;
    cout << "Highest weight " << HighestWeight << endl;

    while (CurrentWeight > 0)
    {
        cout << "Please enter the next weight: ";
        cin >> CurrentWeight;
        if (CurrentWeight > HighestWeight)
            HighestWeight = CurrentWeight;
        cout << "Current weight " << CurrentWeight << endl;
        cout << "Highest weight " << HighestWeight << endl;
    }

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short FirstWeight;
    short SecondWeight;
    short FirstAge;
    short SecondAge;
    short AverageWeight;
    short AverageAge;

    cout << "Please type in the first weight: ";
    cin >> FirstWeight;

    cout << "Please type in the second weight: ";
    cin >> SecondWeight;

    AverageWeight = (FirstWeight + SecondWeight) / 2;

    cout << "Please type in the first age: ";
    cin >> FirstAge;

    cout << "Please type in the second age: ";
    cin >> SecondAge;

    AverageAge = (FirstAge + SecondAge) / 2;

    cout << "The average weight was: " << AverageWeight << endl;
    cout << "The average age was: " << AverageAge << endl;

    return 0;
}
```

```
#include <iostream.h>
int main()
{
    short weight;
    short total;

    cout << "Please type in your weight, typing 0 to end:";
    cin >> weight;

    total = weight;

    while (weight > 0)
    {
        cout << "Please type in your weight, typing 0 to end:";
        cin >> weight;
        total = total + weight;
    }

    cout << "The total is: " << total << endl;
    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short weight;

    cout << "Please write your weight here: ";

    cin >> weight;

    cout << "I wish I only weighed " << weight << " pounds.";

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short weight;

    cout << "Please write your weight here. "\n";

    cin >> weight

    return 0;
}
```

```
#include <iostream.h>
#include "vector.h"

int main()
{
    vector<short> x(4);
    short Result;
    short i;

    x[0] = 3;
    for (i = 1; i < 4; i ++ )
        x[i] = x[i-1] * 2;

    Result = 0;
    for (i = 0; i < 4; i ++ )
        Result = Result + x[i];

    cout << Result << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "vector.h"

int main()
{
    vector<short> x(5);
    short Result;
    short i;

    for (i = 0; i < 5; i ++ )
        {
            x[i] = 2 * i;
        }

    for (i = 0; i < 5; i ++ )
        {
            Result = Result + x[i];
        }

    cout << Result << endl;

    return 0;
}
```

```
C:\WHOS\CODE>attrib +r trans.exe
C:\WHOS\CODE>del *.exe
Access denied
C:\WHOS\CODE>del *.
File not found
C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace pump1
C:\WHOS\CODE>trans pump1
C:\WHOS\CODE>gcc -o pump1.o -c -I. -g $$$$$$$$.cc
C:\WHOS\CODE>gxx -o pump1 pump1.o z:/djgpp/lib/stharch.a
C:\WHOS\CODE>del $$$$$$$$.cc
C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace pump1a
C:\WHOS\CODE>trans pump1a
C:\WHOS\CODE>gcc -o pump1a.o -c -I. -g $$$$$$$$.cc
C:\WHOS\CODE>gxx -o pump1a pump1a.o z:/djgpp/lib/stharch.a
C:\WHOS\CODE>del $$$$$$$$.cc
C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace pump2
C:\WHOS\CODE>trans pump2
C:\WHOS\CODE>gcc -o pump2.o -c -I. -g $$$$$$$$.cc
C:\WHOS\CODE>gxx -o pump2 pump2.o z:/djgpp/lib/stharch.a
C:\WHOS\CODE>del $$$$$$$$.cc
C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace vect1
C:\WHOS\CODE>trans vect1
C:\WHOS\CODE>gcc -o vect1.o -c -I. -g $$$$$$$$.cc
C:\WHOS\CODE>gxx -o vect1 vect1.o z:/djgpp/lib/stharch.a
C:\WHOS\CODE>del $$$$$$$$.cc
C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace vect2
C:\WHOS\CODE>trans vect2
C:\WHOS\CODE>gcc -o vect2.o -c -I. -g $$$$$$$$.cc
C:\WHOS\CODE>gxx -o vect2 vect2.o z:/djgpp/lib/stharch.a
C:\WHOS\CODE>del $$$$$$$$.cc
C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace vect2a
C:\WHOS\CODE>trans vect2a
C:\WHOS\CODE>gcc -o vect2a.o -c -I. -g $$$$$$$$.cc
C:\WHOS\CODE>gxx -o vect2a vect2a.o z:/djgpp/lib/stharch.a
```

```
C:\WHOS\CODE>del $$$$$$.cc

C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace vect3

C:\WHOS\CODE>trans vect3

C:\WHOS\CODE>gcc -o vect3.o -c -I. -g $$$$$$.cc

C:\WHOS\CODE>gxx -o vect3 vect3.o z:/djgpp/lib/stharch.a

C:\WHOS\CODE>del $$$$$$.cc

C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace morbas00

C:\WHOS\CODE>trans morbas00

C:\WHOS\CODE>gcc -o morbas00.o -c -I. -g $$$$$$.cc

C:\WHOS\CODE>gxx -o morbas00 morbas00.o z:/djgpp/lib/stharch.a

C:\WHOS\CODE>del $$$$$$.cc

C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace morbas01

C:\WHOS\CODE>trans morbas01

C:\WHOS\CODE>gcc -o morbas01.o -c -I. -g $$$$$$.cc

C:\WHOS\CODE>gxx -o morbas01 morbas01.o z:/djgpp/lib/stharch.a

C:\WHOS\CODE>del $$$$$$.cc

C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace func1

C:\WHOS\CODE>trans func1

C:\WHOS\CODE>gcc -o func1.o -c -I. -g $$$$$$.cc

C:\WHOS\CODE>gxx -o func1 func1.o z:/djgpp/lib/stharch.a

C:\WHOS\CODE>del $$$$$$.cc

C:\WHOS\CODE>
C:\WHOS\CODE>call mktrace calc1

C:\WHOS\CODE>trans calc1

C:\WHOS\CODE>gcc -o calc1.o -c -I. -g $$$$$$.cc

C:\WHOS\CODE>gxx -o calc1 calc1.o z:/djgpp/lib/stharch.a

C:\WHOS\CODE>del $$$$$$.cc

C:\WHOS\CODE>
C:\WHOS\CODE>
C:\WHOS\CODE>move *.exe ..\tracing
c:\whos\code\trans.exe => c:\whos\tracing\trans.exe [ok]
c:\whos\code\pump1.exe => c:\whos\tracing\pump1.exe [ok]
c:\whos\code\pump1a.exe => c:\whos\tracing\pump1a.exe [ok]
c:\whos\code\pump2.exe => c:\whos\tracing\pump2.exe [ok]
c:\whos\code\vect1.exe => c:\whos\tracing\vect1.exe [ok]
c:\whos\code\vect2.exe => c:\whos\tracing\vect2.exe [ok]
c:\whos\code\vect2a.exe => c:\whos\tracing\vect2a.exe [ok]
c:\whos\code\vect3.exe => c:\whos\tracing\vect3.exe [ok]
c:\whos\code\morbas00.exe => c:\whos\tracing\morbas00.exe [ok]
```

```
c:\whos\code\morbas01.exe => c:\whos\tracing\morbas01.exe [ok]
c:\whos\code\func1.exe => c:\whos\tracing\func1.exe [ok]
c:\whos\code\calc1.exe => c:\whos\tracing\calc1.exe [ok]

C:\WHOS\CODE>move *. ..\tracing
c:\whos\code\pump1 => c:\whos\tracing\pump1 [ok]
c:\whos\code\pump1a => c:\whos\tracing\pump1a [ok]
c:\whos\code\pump2 => c:\whos\tracing\pump2 [ok]
c:\whos\code\vect1 => c:\whos\tracing\vect1 [ok]
c:\whos\code\vect2 => c:\whos\tracing\vect2 [ok]
c:\whos\code\vect2a => c:\whos\tracing\vect2a [ok]
c:\whos\code\vect3 => c:\whos\tracing\vect3 [ok]
c:\whos\code\morbas00 => c:\whos\tracing\morbas00 [ok]
c:\whos\code\morbas01 => c:\whos\tracing\morbas01 [ok]
c:\whos\code\func1 => c:\whos\tracing\func1 [ok]
c:\whos\code\calc1 => c:\whos\tracing\calc1 [ok]

C:\WHOS\CODE>move ..\tracing\trans.exe .
c:\whos\tracing\trans.exe => c:\whos\code\trans.exe [ok]

C:\WHOS\CODE>del *.o

C:\WHOS\CODE>
```

E:\WHOS\CODE>call mknormu basic00

E:\WHOS\CODE>gcc -o basic00.o -c -I. -g basic00.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC00 => E:\WHOS\NORMAL\BASIC00 [ok]

E:\WHOS\CODE\BASIC00.EXE => E:\WHOS\NORMAL\BASIC00.EXE [ok]

E:\WHOS\CODE>call mknormu basic01

E:\WHOS\CODE>gcc -o basic01.o -c -I. -g basic01.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC01 => E:\WHOS\NORMAL\BASIC01 [ok]

E:\WHOS\CODE\BASIC01.EXE => E:\WHOS\NORMAL\BASIC01.EXE [ok]

E:\WHOS\CODE>call mknormu basic02

E:\WHOS\CODE>gcc -o basic02.o -c -I. -g basic02.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC02 => E:\WHOS\NORMAL\BASIC02 [ok]

E:\WHOS\CODE\BASIC02.EXE => E:\WHOS\NORMAL\BASIC02.EXE [ok]

E:\WHOS\CODE>call mknormu basic03

E:\WHOS\CODE>gcc -o basic03.o -c -I. -g basic03.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC03 => E:\WHOS\NORMAL\BASIC03 [ok]

E:\WHOS\CODE\BASIC03.EXE => E:\WHOS\NORMAL\BASIC03.EXE [ok]

E:\WHOS\CODE>call mknormu basic04

E:\WHOS\CODE>gcc -o basic04.o -c -I. -g basic04.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC04 => E:\WHOS\NORMAL\BASIC04 [ok]

E:\WHOS\CODE\BASIC04.EXE => E:\WHOS\NORMAL\BASIC04.EXE [ok]

E:\WHOS\CODE>call mknormu basic05

E:\WHOS\CODE>gcc -o basic05.o -c -I. -g basic05.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC05 => E:\WHOS\NORMAL\BASIC05 [ok]

E:\WHOS\CODE\BASIC05.EXE => E:\WHOS\NORMAL\BASIC05.EXE [ok]

E:\WHOS\CODE>call mknormu basic06

E:\WHOS\CODE>gcc -o basic06.o -c -I. -g basic06.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC06 => E:\WHOS\NORMAL\BASIC06 [ok]

E:\WHOS\CODE\BASIC06.EXE => E:\WHOS\NORMAL\BASIC06.EXE [ok]

E:\WHOS\CODE>call mknormu basic07

E:\WHOS\CODE>gcc -o basic07.o -c -I. -g basic07.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC07 => E:\WHOS\NORMAL\BASIC07 [ok]

E:\WHOS\CODE\BASIC07.EXE => E:\WHOS\NORMAL\BASIC07.EXE [ok]

E:\WHOS\CODE>call mknormu basic08

E:\WHOS\CODE>gcc -o basic08.o -c -I. -g basic08.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC08 => E:\WHOS\NORMAL\BASIC08 [ok]

E:\WHOS\CODE\BASIC08.EXE => E:\WHOS\NORMAL\BASIC08.EXE [ok]

E:\WHOS\CODE>call mknormu basic09

E:\WHOS\CODE>gcc -o basic09.o -c -I. -g basic09.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BASIC09 => E:\WHOS\NORMAL\BASIC09 [ok]

E:\WHOS\CODE\BASIC09.EXE => E:\WHOS\NORMAL\BASIC09.EXE [ok]

E:\WHOS\CODE>call mknormu birthday

```
E:\WHOS\CODE>gcc -o birthday.o -c -I. -g birthday.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\BIRTHDAY => E:\WHOS\NORMAL\BIRTHDAY [ok]

E:\WHOS\CODE\BIRTHDAY.EXE => E:\WHOS\NORMAL\BIRTHDAY.EXE [ok]

E:\WHOS\CODE>call mknormu calc1

E:\WHOS\CODE>gcc -o calc1.o -c -I. -g calc1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\CALC1 => E:\WHOS\NORMAL\CALC1 [ok]

E:\WHOS\CODE\CALC1.EXE => E:\WHOS\NORMAL\CALC1.EXE [ok]

E:\WHOS\CODE>call mknormu count1

E:\WHOS\CODE>gcc -o count1.o -c -I. -g count1.cc -pedantic-errors -Wparentheses -O -Wall
count1.cc: In function `short int counter()':
count1.cc:10: control reaches end of non-void function `counter()'
E:\WHOS\CODE>call mknormu count2

E:\WHOS\CODE>gcc -o count2.o -c -I. -g count2.cc -pedantic-errors -Wparentheses -O -Wall
count2.cc: In function `short int counter()':
count2.cc:5: warning: `short int count' might be used uninitialized in this function
count2.cc:9: warning: `short int n' might be used uninitialized in this function
count2.cc:10: control reaches end of non-void function `counter()'
E:\WHOS\CODE>call mknormu count3

E:\WHOS\CODE>gcc -o count3.o -c -I. -g count3.cc -pedantic-errors -Wparentheses -O -Wall
count3.cc: In function `short int counter()':
count3.cc:10: control reaches end of non-void function `counter()'
E:\WHOS\CODE>call mknormu count4

E:\WHOS\CODE>gcc -o count4.o -c -I. -g count4.cc -pedantic-errors -Wparentheses -O -Wall
count4.cc: In function `short int counter()':
count4.cc:10: control reaches end of non-void function `counter()'
E:\WHOS\CODE>call mknormu count5

E:\WHOS\CODE>gcc -o count5.o -c -I. -g count5.cc -pedantic-errors -Wparentheses -O -Wall
count5.cc: In function `short int counter()':
count5.cc:10: control reaches end of non-void function `counter()'
E:\WHOS\CODE>call mknormu count6

E:\WHOS\CODE>gcc -o count6.o -c -I. -g count6.cc -pedantic-errors -Wparentheses -O -Wall
count6.cc: In function `short int counter()':
count6.cc:10: control reaches end of non-void function `counter()'
E:\WHOS\CODE>call mknormu cout1

E:\WHOS\CODE>gcc -o cout1.o -c -I. -g cout1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\COUT1 => E:\WHOS\NORMAL\COUT1 [ok]

E:\WHOS\CODE\COUT1.EXE => E:\WHOS\NORMAL\COUT1.EXE [ok]

E:\WHOS\CODE>call mknormu dangchar

E:\WHOS\CODE>gcc -o dangchar.o -c -I. -g dangchar.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\DANGCHAR => E:\WHOS\NORMAL\DANGCHAR [ok]

E:\WHOS\CODE\DANGCHAR.EXE => E:\WHOS\NORMAL\DANGCHAR.EXE [ok]

E:\WHOS\CODE>call mknormu func1

E:\WHOS\CODE>gcc -o func1.o -c -I. -g func1.cc -pedantic-errors -Wparentheses -O -Wall
func1.cc: In function `int main()':
func1.cc:20: warning: unused variable `short int Result'
E:\WHOS\CODE\FUNC1 => E:\WHOS\NORMAL\FUNC1 [ok]

E:\WHOS\CODE\FUNC1.EXE => E:\WHOS\NORMAL\FUNC1.EXE [ok]

E:\WHOS\CODE>call mknormu inita
```

```
E:\WHOS\CODE>gcc -o inita.o -c -I. -g inita.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\INITA => E:\WHOS\NORMAL\INITA [ok]

E:\WHOS\CODE\INITA.EXE => E:\WHOS\NORMAL\INITA.EXE [ok]

E:\WHOS\CODE>call mknormu initb

E:\WHOS\CODE>gcc -o initb.o -c -I. -g initb.cc -pedantic-errors -Wparentheses -O -Wall
initb.cc: In function `short int counter()':
initb.cc:16: warning: `short int count' might be used uninitialized in this function
initb.cc:20: warning: `short int n' might be used uninitialized in this function
E:\WHOS\CODE\INITB => E:\WHOS\NORMAL\INITB [ok]

E:\WHOS\CODE\INITB.EXE => E:\WHOS\NORMAL\INITB.EXE [ok]

E:\WHOS\CODE>call mknormu initc

E:\WHOS\CODE>gcc -o initc.o -c -I. -g initc.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\INITC => E:\WHOS\NORMAL\INITC [ok]

E:\WHOS\CODE\INITC.EXE => E:\WHOS\NORMAL\INITC.EXE [ok]

E:\WHOS\CODE>call mknormu initd

E:\WHOS\CODE>gcc -o initd.o -c -I. -g initd.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\INITD => E:\WHOS\NORMAL\INITD [ok]

E:\WHOS\CODE\INITD.EXE => E:\WHOS\NORMAL\INITD.EXE [ok]

E:\WHOS\CODE>call mknormu inite

E:\WHOS\CODE>gcc -o inite.o -c -I. -g inite.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\INITE => E:\WHOS\NORMAL\INITE [ok]

E:\WHOS\CODE\INITE.EXE => E:\WHOS\NORMAL\INITE.EXE [ok]

E:\WHOS\CODE>call mknormu initf

E:\WHOS\CODE>gcc -o initf.o -c -I. -g initf.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\INITF => E:\WHOS\NORMAL\INITF [ok]

E:\WHOS\CODE\INITF.EXE => E:\WHOS\NORMAL\INITF.EXE [ok]

E:\WHOS\CODE>call mknormu invent1

E:\WHOS\CODE>gcc -o invent1.o -c -I. -g invent1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu invent2

E:\WHOS\CODE>gcc -o invent2.o -c -I. -g invent2.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu item1

E:\WHOS\CODE>gcc -o item1.o -c -I. -g item1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu item2

E:\WHOS\CODE>gcc -o item2.o -c -I. -g item2.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu item4

E:\WHOS\CODE>gcc -o item4.o -c -I. -g item4.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu item5

E:\WHOS\CODE>gcc -o item5.o -c -I. -g item5.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu item6

E:\WHOS\CODE>gcc -o item6.o -c -I. -g item6.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu itemtst1

E:\WHOS\CODE>gcc -o itemtst1.o -c -I. -g itemtst1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\ITEMTST1 => E:\WHOS\NORMAL\ITEMTST1 [ok]
```

```
E:\WHOS\CODE\ITEMTST1.EXE => E:\WHOS\NORMAL\ITEMTST1.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu itemtst2
```

```
E:\WHOS\CODE>gcc -o itemtst2.o -c -I. -g itemtst2.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\ITEMTST2 => E:\WHOS\NORMAL\ITEMTST2 [ok]
```

```
E:\WHOS\CODE\ITEMTST2.EXE => E:\WHOS\NORMAL\ITEMTST2.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu itemtst3
```

```
E:\WHOS\CODE>gcc -o itemtst3.o -c -I. -g itemtst3.cc -pedantic-errors -Wparentheses -O -Wall  
itemtst3.cc: In function `int main()':  
itemtst3.cc:34: member `m_UPC' is a private member of class `StockItem'  
itemtst3.cc:43: member `m_InStock' is a private member of class `StockItem'
```

```
E:\WHOS\CODE>call mknormu itemtst4
```

```
E:\WHOS\CODE>gcc -o itemtst4.o -c -I. -g itemtst4.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\ITEMTST4 => E:\WHOS\NORMAL\ITEMTST4 [ok]
```

```
E:\WHOS\CODE\ITEMTST4.EXE => E:\WHOS\NORMAL\ITEMTST4.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu itemtst5
```

```
E:\WHOS\CODE>gcc -o itemtst5.o -c -I. -g itemtst5.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\ITEMTST5 => E:\WHOS\NORMAL\ITEMTST5 [ok]
```

```
E:\WHOS\CODE\ITEMTST5.EXE => E:\WHOS\NORMAL\ITEMTST5.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu itemtst6
```

```
E:\WHOS\CODE>gcc -o itemtst6.o -c -I. -g itemtst6.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\ITEMTST6 => E:\WHOS\NORMAL\ITEMTST6 [ok]
```

```
E:\WHOS\CODE\ITEMTST6.EXE => E:\WHOS\NORMAL\ITEMTST6.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu itmtst2a
```

```
E:\WHOS\CODE>gcc -o itmtst2a.o -c -I. -g itmtst2a.cc -pedantic-errors -Wparentheses -O -Wall  
gcc.exe: itmtst2a.cc: No such file or directory (ENOENT)  
gcc.exe: No input files
```

```
E:\WHOS\CODE>call mknormu morbas00
```

```
E:\WHOS\CODE>gcc -o morbas00.o -c -I. -g morbas00.cc -pedantic-errors -Wparentheses -O -Wall  
morbas00.cc: In function `int main()':  
morbas00.cc:7: warning: `short int Result' might be used uninitialized in this function  
morbas00.cc:20: warning: `short int n' might be used uninitialized in this function
```

```
E:\WHOS\CODE\MORBAS00 => E:\WHOS\NORMAL\MORBAS00 [ok]
```

```
E:\WHOS\CODE\MORBAS00.EXE => E:\WHOS\NORMAL\MORBAS00.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu morbas01
```

```
E:\WHOS\CODE>gcc -o morbas01.o -c -I. -g morbas01.cc -pedantic-errors -Wparentheses -O -Wall  
morbas01.cc: In function `int main()':  
morbas01.cc:9: warning: unused variable `short int j'
```

```
E:\WHOS\CODE\MORBAS01 => E:\WHOS\NORMAL\MORBAS01 [ok]
```

```
E:\WHOS\CODE\MORBAS01.EXE => E:\WHOS\NORMAL\MORBAS01.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu morbas02
```

```
E:\WHOS\CODE>gcc -o morbas02.o -c -I. -g morbas02.cc -pedantic-errors -Wparentheses -O -Wall  
morbas02.cc: In function `int main()':  
morbas02.cc:7: stray '\\' in program  
morbas02.cc:11: parse error before `return'
```

```
E:\WHOS\CODE>call mknormu morbas03
```

```
E:\WHOS\CODE>gcc -o morbas03.o -c -I. -g morbas03.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\MORBAS03 => E:\WHOS\NORMAL\MORBAS03 [ok]
```

E:\WHOS\CODE\MORBAS03.EXE => E:\WHOS\NORMAL\MORBAS03.EXE [ok]

E:\WHOS\CODE>call mknormu morbas04

E:\WHOS\CODE>gcc -o morbas04.o -c -I. -g morbas04.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\MORBAS04 => E:\WHOS\NORMAL\MORBAS04 [ok]

E:\WHOS\CODE\MORBAS04.EXE => E:\WHOS\NORMAL\MORBAS04.EXE [ok]

E:\WHOS\CODE>call mknormu nofunc

E:\WHOS\CODE>gcc -o nofunc.o -c -I. -g nofunc.cc -pedantic-errors -Wparentheses -O -Wall
nofunc.cc: In function `int main()':
nofunc.cc:13: warning: unused variable `short int Length'
nofunc.cc:12: warning: unused variable `short int Result'
nofunc.cc:9: warning: unused variable `short int i'
E:\WHOS\CODE\NOFUNC => E:\WHOS\NORMAL\NOFUNC [ok]

E:\WHOS\CODE\NOFUNC.EXE => E:\WHOS\NORMAL\NOFUNC.EXE [ok]

E:\WHOS\CODE>call mknormu pump1

E:\WHOS\CODE>gcc -o pump1.o -c -I. -g pump1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\PUMP1 => E:\WHOS\NORMAL\PUMP1 [ok]

E:\WHOS\CODE\PUMP1.EXE => E:\WHOS\NORMAL\PUMP1.EXE [ok]

E:\WHOS\CODE>call mknormu pump1a

E:\WHOS\CODE>gcc -o pump1a.o -c -I. -g pump1a.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\PUMP1A => E:\WHOS\NORMAL\PUMP1A [ok]

E:\WHOS\CODE\PUMP1A.EXE => E:\WHOS\NORMAL\PUMP1A.EXE [ok]

E:\WHOS\CODE>call mknormu pump2

E:\WHOS\CODE>gcc -o pump2.o -c -I. -g pump2.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\PUMP2 => E:\WHOS\NORMAL\PUMP2 [ok]

E:\WHOS\CODE\PUMP2.EXE => E:\WHOS\NORMAL\PUMP2.EXE [ok]

E:\WHOS\CODE>call mknormu scopclas

E:\WHOS\CODE>gcc -o scopclas.o -c -I. -g scopclas.cc -pedantic-errors -Wparentheses -O -Wall
scopclas.cc: In function `short int func1()':
scopclas.cc:8: warning: `short int count3' might be used uninitialized in this function
scopclas.cc:22: warning: `short int n' might be used uninitialized in this function
E:\WHOS\CODE\SCOPCLAS => E:\WHOS\NORMAL\SCOPCLAS [ok]

E:\WHOS\CODE\SCOPCLAS.EXE => E:\WHOS\NORMAL\SCOPCLAS.EXE [ok]

E:\WHOS\CODE>call mknormu strcmp

E:\WHOS\CODE>gcc -o strcmp.o -c -I. -g strcmp.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\STRCMP => E:\WHOS\NORMAL\STRCMP [ok]

E:\WHOS\CODE\STRCMP.EXE => E:\WHOS\NORMAL\STRCMP.EXE [ok]

E:\WHOS\CODE>call mknormu strex1

E:\WHOS\CODE>gcc -o strex1.o -c -I. -g strex1.cc -pedantic-errors -Wparentheses -O -Wall
strex1.cc: In function `int main()':
strex1.cc:21: member `m_Length' is a private member of class `string'
E:\WHOS\CODE>call mknormu strex2

E:\WHOS\CODE>gcc -o strex2.o -c -I. -g strex2.cc -pedantic-errors -Wparentheses -O -Wall
strex2.cc: In function `int main()':
strex2.cc:9: constructor `string::string()' is private

```
strex2.cc:17: within this context
strex2.cc:17: in base initialization for class `string'
E:\WHOS\CODE>call mknormu strex3
```

```
E:\WHOS\CODE>gcc -o strex3.o -c -I. -g strex3.cc -pedantic-errors -Wparentheses -O -Wall
strex3.cc:12: warning: `class string' only defines a private destructor and has no friends
strex3.cc: In function `int main()':
strex3.cc:16: destructor for type `string' is private in this scope
E:\WHOS\CODE>call mknormu strex5
```

```
E:\WHOS\CODE>gcc -o strex5.o -c -I. -g strex5.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\STREX5 => E:\WHOS\NORMAL\STREX5 [ok]
```

```
E:\WHOS\CODE\STREX5.EXE => E:\WHOS\NORMAL\STREX5.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu strex6
```

```
E:\WHOS\CODE>gcc -o strex6.o -c -I. -g strex6.cc -pedantic-errors -Wparentheses -O -Wall
strex6.cc: In function `int main()':
strex6.cc:9: constructor `string::string(char *)' is private
strex6.cc:21: within this context
strex6.cc:6: in passing argument 1 of `string::operator =(const string &)'
E:\WHOS\CODE>call mknormu string1
```

```
E:\WHOS\CODE>gcc -o string1.o -c -I. -g string1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu string3
```

```
E:\WHOS\CODE>gcc -o string3.o -c -I. -g string3.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu string4
```

```
E:\WHOS\CODE>gcc -o string4.o -c -I. -g string4.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu string5
```

```
E:\WHOS\CODE>gcc -o string5.o -c -I. -g string5.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu string5a
```

```
E:\WHOS\CODE>gcc -o string5a.o -c -I. -g string5a.cc -pedantic-errors -Wparentheses -O -Wall
string5a.cc: In method `bool string::operator <(const class string &)':
string5a.cc:51: warning: `bool Result' might be used uninitialized in this function
E:\WHOS\CODE>call mknormu string5x
```

```
E:\WHOS\CODE>gcc -o string5x.o -c -I. -g string5x.cc -pedantic-errors -Wparentheses -O -Wall
string5x.cc: In function `class istream & operator >>(class istream &, class string &)':
string5x.cc:79: uninitialized const `short int const BUFLen'
string5x.cc:81: ANSI C++ forbids variable-size array `Buf'
string5x.cc:79: warning: `short int const BUFLen' might be used uninitialized in this function
E:\WHOS\CODE>call mknormu string5y
```

```
E:\WHOS\CODE>gcc -o string5y.o -c -I. -g string5y.cc -pedantic-errors -Wparentheses -O -Wall
string5y.cc: In function `class istream & operator >>(class istream &, class string &)':
string5y.cc:9: ANSI C++ forbids variable-size array `Buf'
E:\WHOS\CODE>call mknormu string6
```

```
E:\WHOS\CODE>gcc -o string6.o -c -I. -g string6.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE>call mknormu strsort1
```

```
E:\WHOS\CODE>gcc -o strsort1.o -c -I. -g strsort1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\STRSORT1 => E:\WHOS\NORMAL\STRSORT1 [ok]
```

```
E:\WHOS\CODE\STRSORT1.EXE => E:\WHOS\NORMAL\STRSORT1.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu strtst1
```

```
E:\WHOS\CODE>gcc -o strtst1.o -c -I. -g strtst1.cc -pedantic-errors -Wparentheses -O -Wall
E:\WHOS\CODE\STRTST1 => E:\WHOS\NORMAL\STRTST1 [ok]
```

```
E:\WHOS\CODE\STRTST1.EXE => E:\WHOS\NORMAL\STRTST1.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu strtst3
```

```
E:\WHOS\CODE>gcc -o strtst3.o -c -I. -g strtst3.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\STRTST3 => E:\WHOS\NORMAL\STRTST3 [ok]
```

```
E:\WHOS\CODE\STRTST3.EXE => E:\WHOS\NORMAL\STRTST3.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu strtst3a
```

```
E:\WHOS\CODE>gcc -o strtst3a.o -c -I. -g strtst3a.cc -pedantic-errors -Wparentheses -O -Wall  
strtst3a.cc: In function `int main()':  
strtst3a.cc:8: member `m_Length' is a private member of class `string'  
E:\WHOS\CODE>call mknormu strtst4
```

```
E:\WHOS\CODE>gcc -o strtst4.o -c -I. -g strtst4.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\STRTST4 => E:\WHOS\NORMAL\STRTST4 [ok]
```

```
E:\WHOS\CODE\STRTST4.EXE => E:\WHOS\NORMAL\STRTST4.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu strtst5
```

```
E:\WHOS\CODE>gcc -o strtst5.o -c -I. -g strtst5.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\STRTST5 => E:\WHOS\NORMAL\STRTST5 [ok]
```

```
E:\WHOS\CODE\STRTST5.EXE => E:\WHOS\NORMAL\STRTST5.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu strtst5x
```

```
E:\WHOS\CODE>gcc -o strtst5x.o -c -I. -g strtst5x.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\STRTST5X => E:\WHOS\NORMAL\STRTST5X [ok]
```

```
E:\WHOS\CODE\STRTST5X.EXE => E:\WHOS\NORMAL\STRTST5X.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu vect1
```

```
E:\WHOS\CODE>gcc -o vect1.o -c -I. -g vect1.cc -pedantic-errors -Wparentheses -O -Wall  
vect1.cc: In function `int main()':  
vect1.cc:9: warning: `short int HighestIndex' might be used uninitialized in this function  
E:\WHOS\CODE\VECT1 => E:\WHOS\NORMAL\VECT1 [ok]
```

```
E:\WHOS\CODE\VECT1.EXE => E:\WHOS\NORMAL\VECT1.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu vect2
```

```
E:\WHOS\CODE>gcc -o vect2.o -c -I. -g vect2.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\VECT2 => E:\WHOS\NORMAL\VECT2 [ok]
```

```
E:\WHOS\CODE\VECT2.EXE => E:\WHOS\NORMAL\VECT2.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu vect2a
```

```
E:\WHOS\CODE>gcc -o vect2a.o -c -I. -g vect2a.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\VECT2A => E:\WHOS\NORMAL\VECT2A [ok]
```

```
E:\WHOS\CODE\VECT2A.EXE => E:\WHOS\NORMAL\VECT2A.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu vect3
```

```
E:\WHOS\CODE>gcc -o vect3.o -c -I. -g vect3.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE\VECT3 => E:\WHOS\NORMAL\VECT3 [ok]
```

```
E:\WHOS\CODE\VECT3.EXE => E:\WHOS\NORMAL\VECT3.EXE [ok]
```

```
E:\WHOS\CODE>call mknormu wassert
```

```
E:\WHOS\CODE>gcc -o wassert.o -c -I. -g wassert.cc -pedantic-errors -Wparentheses -O -Wall  
E:\WHOS\CODE>del *.o
```

```
E:\WHOS\CODE>
```

```
#include <iostream.h>
#include <fstream.h>
#include "string6.h"
#include "item6.h"
#include "invent2.h"

int main()
{
    ifstream InputStream("shop2.in");
    string PurchaseUPC;
    short PurchaseCount;
    string ItemName;
    short OldInventory;
    short NewInventory;
    Inventory MyInventory;
    StockItem FoundItem;
    string TransactionCode;

    MyInventory.LoadInventory(InputStream);

    cout << "What is the UPC of the item? ";
    cin >> PurchaseUPC;

    FoundItem = MyInventory.FindItem(PurchaseUPC);
    if (FoundItem.IsNull() == true)
    {
        cout << "Can't find that item. Please check UPC." << endl;
        return 0;
    }

    OldInventory = FoundItem.GetInventory();
    ItemName = FoundItem.GetName();

    cout << "There are currently " << OldInventory << " units of "
    << ItemName << " in stock." << endl;

    cout << "Please enter transaction code as follows:" << endl;
    cout << "S (sale)" << endl;
    cout << "C (price check)" << endl;
    cin >> TransactionCode;

    if (TransactionCode == "C" || TransactionCode == "c")
    {
        cout << "The name of that item is: " << ItemName << endl;
        cout << "Its price is: " << FoundItem.GetPrice();
    }
    else if (TransactionCode == "S" || TransactionCode == "s")
    {
        cout << "How many items were sold? ";
        cin >> PurchaseCount;

        FoundItem.DeductSaleFromInventory(PurchaseCount);
        MyInventory.UpdateItem(FoundItem);

        cout << "The inventory has been updated." << endl;

        FoundItem = MyInventory.FindItem(PurchaseUPC);
        NewInventory = FoundItem.GetInventory();

        ofstream OutputStream("shop2.out");
        MyInventory.StoreInventory(OutputStream);

        cout << "There are now " << NewInventory << " units of "
        << ItemName << " in stock." << endl;
    }

    return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include "string6.h"
#include "item5.h"
#include "invent1.h"

int main()
{
    ifstream InputStream("shop2.in");
    string PurchaseUPC;
    short PurchaseCount;
    string ItemName;
    short OldInventory;
    short NewInventory;
    Inventory MyInventory;
    StockItem FoundItem;
    string TransactionCode;

    MyInventory.LoadInventory(InputStream);

    cout << "What is the UPC of the item? ";
    cin >> PurchaseUPC;

    FoundItem = MyInventory.FindItem(PurchaseUPC);
    if (FoundItem.IsNull() == true)
    {
        cout << "Can't find that item. Please check UPC." << endl;
        return 0;
    }

    OldInventory = FoundItem.GetInventory();
    ItemName = FoundItem.GetName();

    cout << "There are currently " << OldInventory << " units of "
    << ItemName << " in stock." << endl;

    cout << "Please enter transaction code as follows:\n";
    cout << "S (sale), C (price check): ";
    cin >> TransactionCode;

    if (TransactionCode == "C" || TransactionCode == "c")
    {
        cout << "The name of that item is: " << ItemName << endl;
        cout << "Its price is: " << FoundItem.GetPrice();
    }
    else if (TransactionCode == "S" || TransactionCode == "s")
    {
        cout << "How many items were sold? ";
        cin >> PurchaseCount;

        FoundItem.DeductSaleFromInventory(PurchaseCount);
        MyInventory.UpdateItem(FoundItem);

        cout << "The inventory has been updated." << endl;

        FoundItem = MyInventory.FindItem(PurchaseUPC);
        NewInventory = FoundItem.GetInventory();

        cout << "There are now " << NewInventory << " units of "
        << ItemName << " in stock." << endl;
    }

    return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include "vector.h"
#include "string6.h"
#include "item4.h"

int main()
{
    ifstream ShopInfo("shop2.in");
    vector<StockItem> AllItems(100);
    short i;
    short InventoryCount;
    short OldInventory;
    short NewInventory;
    string PurchaseUPC;
    string ItemName;
    short PurchaseCount;
    bool Found;

    for (i = 0; i < 100; i ++)
    {
        AllItems[i].Read(ShopInfo);
        if (ShopInfo.fail() != 0)
            break;
    }

    InventoryCount = i;
    cout << "What is the UPC of the item? ";
    cin >> PurchaseUPC;
    Found = false;

    for (i = 0; i < InventoryCount; i ++)
    {
        if (AllItems[i].CheckUPC(PurchaseUPC) == true)
        {
            Found = true;
            break;
        }
    }

    if (Found == true)
    {
        OldInventory = AllItems[i].GetInventory();
        ItemName = AllItems[i].GetName();

        cout << "There are currently " << OldInventory << " units of "
        << ItemName << " in stock." << endl;
        cout << "How many items were sold? ";
        cin >> PurchaseCount;

        AllItems[i].DeductSaleFromInventory(PurchaseCount);
        cout << "The inventory has been updated." << endl;

        NewInventory = AllItems[i].GetInventory();
        cout << "There are now " << NewInventory << " units of "
        << ItemName << " in stock." << endl;
    }
    else
        cout << "Can't find that item. Please check UPC" << endl;

    return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>
#include "vector.h"
#include "string6.h"
#include "item2.h"

int main()
{
    ifstream ShopInfo("shop2.in");
    vector<StockItem> AllItems(100);
    short i;
    short InventoryCount;
    string PurchaseUPC;
    short PurchaseCount;
    bool Found;

    for (i = 0; i < 100; i ++)
    {
        AllItems[i].Read(ShopInfo);
        if (ShopInfo.fail() != 0)
            break;
    }

    InventoryCount = i;

    cout << "What is the UPC of the item?" << endl;
    cin >> PurchaseUPC;
    cout << "How many items were sold?" << endl;
    cin >> PurchaseCount;

    Found = false;
    for (i = 0; i < InventoryCount; i ++)
    {
        if (PurchaseUPC == AllItems[i].m_UPC)
        {
            Found = true;
            break;
        }
    }

    if (Found == true)
    {
        AllItems[i].m_InStock -= PurchaseCount;
        cout << "The inventory has been updated." << endl;
    }
    else
        cout << "Can't find that item. Please check UPC" << endl;

    return 0;
}
```

Name: 3-ounce cups
Number in stock: 71
Price: 259
Distributor: Bob's Distribution
UPC: 2895657951

Name: Ajax-substitute
Number in stock: 77
Price: 104
Distributor: ABC Dist.
UPC: 8144976072

Name: antihistamines
Number in stock: 5
Price: 224
Distributor: Bob's Distribution
UPC: 7904886261

Name: Arturo sauce
Number in stock: 96
Price: 361
Distributor: Bob's Distribution
UPC: 9495505623

Name: Brannola
Number in stock: 52
Price: 329
Distributor: Wholesale Plus
UPC: 5924505350

Name: bread, challah
Number in stock: 30
Price: 286
Distributor: Bob's Distribution
UPC: 2637964782

Name: bread, rye
Number in stock: 83
Price: 347
Distributor: Bob's Distribution
UPC: 9860958916

Name: bread, white
Number in stock: 23
Price: 308
Distributor: Wholesale Plus
UPC: 2439398000

Name: breadcrumbs, Italian
Number in stock: 11
Price: 399
Distributor: Wholesale Plus
UPC: 157067617

Name: Bufferin
Number in stock: 10
Price: 131
Distributor: ABC Dist.
UPC: 2844879888

Name: Carpet Fresh
Number in stock: 30
Price: 214
Distributor: Bob's Distribution
UPC: 9485730097

Name: chew sticks
Number in stock: 40

Price: 183
Distributor: ABC Dist.
UPC: 1628216044

Name: Clorox 2
Number in stock: 41
Price: 223
Distributor: ABC Dist.
UPC: 3262071273

Name: cough drops
Number in stock: 21
Price: 156
Distributor: Wholesale Plus
UPC: 807158335

Name: dish detergent, auto
Number in stock: 91
Price: 178
Distributor: Bob's Distribution
UPC: 3789078521

Name: dish detergent, liquid
Number in stock: 92
Price: 289
Distributor: ABC Dist.
UPC: 4284562764

Name: dog food (with Alpo)
Number in stock: 56
Price: 308
Distributor: ABC Dist.
UPC: 8348191371

Name: Dove soap
Number in stock: 54
Price: 374
Distributor: Wholesale Plus
UPC: 6779443026

Name: envelopes, large
Number in stock: 51
Price: 238
Distributor: Bob's Distribution
UPC: 4048335347

Name: Fantastic refill
Number in stock: 6
Price: 173
Distributor: Wholesale Plus
UPC: 609197907

Name: flypaper
Number in stock: 36
Price: 246
Distributor: Bob's Distribution
UPC: 4744515566

Name: furniture polish, lemon
Number in stock: 63
Price: 262
Distributor: ABC Dist.
UPC: 9385415630

Name: Gas-X
Number in stock: 51
Price: 217
Distributor: Wholesale Plus

UPC: 7839910737

Name: Glass-Plus
Number in stock: 75
Price: 278
Distributor: Bob's Distribution
UPC: 187583273

Name: Kleenex, brown foil
Number in stock: 7
Price: 132
Distributor: ABC Dist.
UPC: 1282433169

Name: Kleenex, white foil
Number in stock: 54
Price: 296
Distributor: ABC Dist.
UPC: 8274154401

Name: Kosher soap
Number in stock: 19
Price: 303
Distributor: Bob's Distribution
UPC: 3570245420

Name: laundry detergent
Number in stock: 70
Price: 378
Distributor: ABC Dist.
UPC: 896453021

Name: lemon ammonia
Number in stock: 40
Price: 238
Distributor: Bob's Distribution
UPC: 2076249216

Name: Metamucil (NutraSweet)
Number in stock: 10
Price: 276
Distributor: ABC Dist.
UPC: 9276116987

Name: mixed nuts
Number in stock: 44
Price: 182
Distributor: Bob's Distribution
UPC: 7506887254

Name: Murphy's oil soap
Number in stock: 67
Price: 177
Distributor: Bob's Distribution
UPC: 309508989

Name: napkins
Number in stock: 79
Price: 189
Distributor: Bob's Distribution
UPC: 4804723528

Name: no-fat Entenmann's
Number in stock: 34
Price: 113
Distributor: ABC Dist.
UPC: 2060148242

Name: paper towels
Number in stock: 59
Price: 326
Distributor: Wholesale Plus
UPC: 3310192788

Name: Pepto-Bismol
Number in stock: 8
Price: 290
Distributor: ABC Dist.
UPC: 9604241003

Name: plastic forks
Number in stock: 92
Price: 285
Distributor: Wholesale Plus
UPC: 1492434772

Name: plastic spoons
Number in stock: 22
Price: 397
Distributor: Bob's Distribution
UPC: 288813042

Name: popcorn cakes (Quaker)
Number in stock: 55
Price: 376
Distributor: ABC Dist.
UPC: 4064253824

Name: sodas, chocolate
Number in stock: 83
Price: 301
Distributor: Bob's Distribution
UPC: 9967772189

Name: sodas, cream
Number in stock: 50
Price: 223
Distributor: Wholesale Plus
UPC: 1790869528

Name: sodas, NoCaf Diet Coke
Number in stock: 54
Price: 344
Distributor: Wholesale Plus
UPC: 4275354091

Name: sodas, orange
Number in stock: 23
Price: 285
Distributor: Bob's Distribution
UPC: 6808148983

Name: sodas, root beer
Number in stock: 37
Price: 190
Distributor: Wholesale Plus
UPC: 1503129286

Name: SOS
Number in stock: 22
Price: 275
Distributor: Wholesale Plus
UPC: 8759736345

Name: sponges
Number in stock: 19

Price: 305
Distributor: ABC Dist.
UPC: 6139374741

Name: toilet paper, unscented
Number in stock: 16
Price: 342
Distributor: ABC Dist.
UPC: 9567620261

Name: Top Job
Number in stock: 6
Price: 337
Distributor: ABC Dist.
UPC: 4635837960

Name: walnuts
Number in stock: 12
Price: 152
Distributor: Wholesale Plus
UPC: 7148104811

Name: water
Number in stock: 56
Price: 165
Distributor: ABC Dist.
UPC: 7463546800

```
#include <iostream.h>
#include <fstream.h>
#include "vector.h"
#include "string6.h"
#include "item2.h"

int main()
{
    ifstream ShopInfo("shop2.in");
    vector<StockItem> AllItems(100);
    short i;
    short InventoryCount;

    for (i = 0; i < 100; i ++ )
    {
        AllItems[i].Read(ShopInfo);
        if (ShopInfo.fail() != 0)
            break;
    }

    InventoryCount = i;

    for (i = 0; i < InventoryCount; i ++ )
    {
        AllItems[i].Display();
    }

    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"
#include "item1.h"

int main()
{
    StockItem soup;

    soup = StockItem("Chunky Chicken",32,129,
        "Bob's Distribution","123456789");

    soup.Display();

    return 0;
}
```

```
class StockItem
{
public:
    StockItem();

    StockItem(string Name, short InStock, short Price,
string Distributor, string UPC);

    void Display();
    void Read(ifstream& s);

// the following function was added for inventory update
    void Write(ofstream& s);

    bool CheckUPC(string ItemUPC);
    void DeductSaleFromInventory(short QuantitySold);
    short GetInventory();
    string GetName();
    string GetUPC();
    bool IsNull();
    short GetPrice();

private:
    short m_InStock;
    short m_Price;
    string m_Name;
    string m_Distributor;
    string m_UPC;
};
```

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include "string6.h"
#include "item6.h"

StockItem::StockItem()
: m_InStock(0), m_Price(0), m_Name(),
  m_Distributor(), m_UPC()
{
}

StockItem::StockItem(string Name, short InStock,
short Price, string Distributor, string UPC)
: m_InStock(InStock), m_Price(Price), m_Name(Name),
  m_Distributor(Distributor), m_UPC(UPC)
{
}

void StockItem::Display()
{
    cout << "Name: ";
    cout << m_Name << endl;
    cout << "Number in stock: ";
    cout << m_InStock << endl;
    cout << "Price: ";
    cout << m_Price << endl;
    cout << "Distributor: ";
    cout << m_Distributor << endl;
    cout << "UPC: ";
    cout << m_UPC << endl;
    cout << endl;
}

void StockItem::Read(ifstream& s)
{
    s >> m_Name;
    s >> m_InStock;
    s >> m_Price;
    s >> m_Distributor;
    s >> m_UPC;
}

bool StockItem::CheckUPC(string ItemUPC)
{
    if (m_UPC == ItemUPC)
        return 1;

    return 0;
}

void StockItem::DeductSaleFromInventory(short QuantitySold)
{
    m_InStock -= QuantitySold;
}

short StockItem::GetInventory()
{
    return m_InStock;
}

string StockItem::GetName()
{
    return m_Name;
}

string StockItem::GetUPC()
{
```

```
        return m_UPC;
    }

bool StockItem::IsNull()
{
    if (m_UPC == "")
        return 1;

    return 0;
}

short StockItem::GetPrice()
{
    return m_Price;
}

//function added for inventory update
void StockItem::Write(ofstream& s)
{
    s << m_Name << endl;
    s << m_InStock << endl;
    s << m_Price << endl;
    s << m_Distributor << endl;
    s << m_UPC << endl;
    return;
}
```

```
class StockItem
{
public:
    StockItem();

    StockItem(string Name, short InStock, short Price,
string Distributor, string UPC);

    void Display();
    void Read(ifstream& s);

    bool CheckUPC(string ItemUPC);
    void DeductSaleFromInventory(short QuantitySold);
    short GetInventory();
    string GetName();
    bool IsNull();
    short GetPrice();
    string GetUPC();

private:
    short m_InStock;
    short m_Price;
    string m_Name;
    string m_Distributor;
    string m_UPC;
};
```

```
#include <iostream.h>
#include <fstream.h>
#include "string6.h"
#include "item5.h"

StockItem::StockItem()
: m_InStock(0), m_Price(0), m_Name(),
  m_Distributor(), m_UPC()
{
}

StockItem::StockItem(string Name, short InStock,
short Price, string Distributor, string UPC)
: m_InStock(InStock), m_Price(Price), m_Name(Name),
  m_Distributor(Distributor), m_UPC(UPC)
{
}

void StockItem::Display()
{
    cout << "Name: ";
    cout << m_Name << endl;
    cout << "Number in stock: ";
    cout << m_InStock << endl;
    cout << "Price: ";
    cout << m_Price << endl;
    cout << "Distributor: ";
    cout << m_Distributor << endl;
    cout << "UPC: ";
    cout << m_UPC << endl;
    cout << endl;
}

void StockItem::Read(ifstream& s)
{
    s >> m_Name;
    s >> m_InStock;
    s >> m_Price;
    s >> m_Distributor;
    s >> m_UPC;
}

bool StockItem::CheckUPC(string ItemUPC)
{
    if (m_UPC == ItemUPC)
        return true;

    return false;
}

void StockItem::DeductSaleFromInventory(short QuantitySold)
{
    m_InStock -= QuantitySold;
}

short StockItem::GetInventory()
{
    return m_InStock;
}

string StockItem::GetName()
{
    return m_Name;
}

bool StockItem::IsNull()
{

```

```
    if (m_UPC == "")  
        return true;
```

```
    return false;
```

```
}
```

```
short StockItem::GetPrice()
```

```
{
```

```
    return m_Price;
```

```
}
```

```
string StockItem::GetUPC()
```

```
{
```

```
    return m_UPC;
```

```
}
```

```
class StockItem
{
public:
    StockItem();

    StockItem(string Name, short InStock, short Price,
string Distributor, string UPC);

    void Display();
    void Read(ifstream& s);

    bool CheckUPC(string ItemUPC);
    void DeductSaleFromInventory(short QuantitySold);
    short GetInventory();
    string GetName();

private:
    short m_InStock;
    short m_Price;
    string m_Name;
    string m_Distributor;
    string m_UPC;
};
```

```
#include <iostream.h>
#include <fstream.h>
#include "string6.h"
#include "item4.h"

StockItem::StockItem()
: m_InStock(0), m_Price(0), m_Name(),
  m_Distributor(), m_UPC()
{
}

StockItem::StockItem(string Name, short InStock,
short Price, string Distributor, string UPC)
: m_InStock(InStock), m_Price(Price), m_Name(Name),
  m_Distributor(Distributor), m_UPC(UPC)
{
}

void StockItem::Display()
{
    cout << "Name: ";
    cout << m_Name << endl;
    cout << "Number in stock: ";
    cout << m_InStock << endl;
    cout << "Price: ";
    cout << m_Price << endl;
    cout << "Distributor: ";
    cout << m_Distributor << endl;
    cout << "UPC: ";
    cout << m_UPC << endl;
    cout << endl;
}

void StockItem::Read(ifstream& s)
{
    s >> m_Name;
    s >> m_InStock;
    s >> m_Price;
    s >> m_Distributor;
    s >> m_UPC;
}

bool StockItem::CheckUPC(string ItemUPC)
{
    if (m_UPC == ItemUPC)
        return true;
    else
        return false;
}

void StockItem::DeductSaleFromInventory(short QuantitySold)
{
    m_InStock -= QuantitySold;
}

short StockItem::GetInventory()
{
    return m_InStock;
}

string StockItem::GetName()
{
    return m_Name;
}
```

```
class StockItem
{
public:
    StockItem();

    StockItem(string Name, short InStock, short Price,
string Distributor, string UPC);

    void Display();
    void Read(ifstream& s);

private:
    short m_InStock;
    short m_Price;
    string m_Name;
    string m_Distributor;
    string m_UPC;
};
```

```
#include <iostream.h>
#include <fstream.h>
#include "string6.h"
#include "item2.h"

StockItem::StockItem()
: m_InStock(0), m_Price(0), m_Name(),
  m_Distributor(), m_UPC()
{
}

StockItem::StockItem(string Name, short InStock,
short Price, string Distributor, string UPC)
: m_InStock(InStock), m_Price(Price), m_Name(Name),
  m_Distributor(Distributor), m_UPC(UPC)
{
}

void StockItem::Display()
{
    cout << "Name: ";
    cout << m_Name << endl;
    cout << "Number in stock: ";
    cout << m_InStock << endl;
    cout << "Price: ";
    cout << m_Price << endl;
    cout << "Distributor: ";
    cout << m_Distributor << endl;
    cout << "UPC: ";
    cout << m_UPC << endl;
    cout << endl;
}

void StockItem::Read(ifstream& s)
{
    s >> m_Name;
    s >> m_InStock;
    s >> m_Price;
    s >> m_Distributor;
    s >> m_UPC;
}
```

```
class StockItem
{
public:
    StockItem();

    StockItem(string Name, short InStock, short Price,
string Distributor, string UPC);

    void Display();

private:
    short m_InStock;
    short m_Price;
    string m_Name;
    string m_Distributor;
    string m_UPC;
};
```

```
#include <iostream.h>
#include "string6.h"
#include "item1.h"

StockItem::StockItem()
: m_Name(), m_InStock(0), m_Price(0), m_Distributor(), m_UPC()
{
}

StockItem::StockItem(string Name, short InStock,
short Price, string Distributor, string UPC)
: m_Name(Name), m_InStock(InStock), m_Price(Price),
  m_Distributor(Distributor), m_UPC(UPC)
{
}

void StockItem::Display()
{
    cout << "Name: ";
    cout << m_Name << endl;
    cout << "Number in stock: ";
    cout << m_InStock << endl;
    cout << "Price: ";
    cout << m_Price << endl;
    cout << "Distributor: ";
    cout << m_Distributor << endl;
    cout << "UPC: ";
    cout << m_UPC << endl;
}
```

```
#include "vector.h"

class Inventory
{
public:
    Inventory();

    short LoadInventory(ifstream& InputStream);

// the following function was added for inventory update
    void StoreInventory(ofstream& OutputStream);

    StockItem FindItem(string UPC);
    bool UpdateItem(StockItem Item);

private:
    vector<StockItem> m_Stock;
    short m_StockCount;
};
```

```
#include <iostream.h>
#include <fstream.h>
#include "vector.h"
#include "string6.h"
#include "item6.h"
#include "invent2.h"

Inventory::Inventory()
: m_Stock (vector<StockItem>(100)),
  m_StockCount(0)
{
}

short Inventory::LoadInventory(ifstream& InputStream)
{
    short i;

    for (i = 0; i < 100; i ++ )
    {
        m_Stock[i].Read(InputStream);
        if (InputStream.fail() != 0)
            break;
    }

    m_StockCount = i;
    return m_StockCount;
}

StockItem Inventory::FindItem(string UPC)
{
    short i;
    bool Found = false;

    for (i = 0; i < m_StockCount; i ++ )
    {
        if (m_Stock[i].GetUPC() == UPC)
        {
            Found = true;
            break;
        }
    }

    if (Found == true)
        return m_Stock[i];

    return StockItem();
}

bool Inventory::UpdateItem(StockItem Item)
{
    string UPC = Item.GetUPC();

    short i;
    bool Found = true;

    for (i = 0; i < m_StockCount; i ++ )
    {
        if (m_Stock[i].GetUPC() == UPC)
        {
            Found = true;
            break;
        }
    }

    if (Found == true)
        m_Stock[i] = Item;

    return Found;
}
```

```
}  
  
//function added for inventory update  
void Inventory::StoreInventory(ofstream& OutputStream)  
{  
    short i;  
  
    for (i = 0; i < m_StockCount; i ++)  
        m_Stock[i].Write(OutputStream);  
}
```

```
#include "vector.h"

class Inventory
{
public:
    Inventory();

    short LoadInventory(ifstream& InputStream);
    StockItem FindItem(string UPC);
    bool UpdateItem(StockItem Item);

private:
    vector<StockItem> m_Stock;
    short m_StockCount;
};
```

```
#include <iostream.h>
#include <fstream.h>
#include "vector.h"
#include "string6.h"
#include "item5.h"
#include "invent1.h"

Inventory::Inventory()
: m_Stock (vector<StockItem>(100)),
  m_StockCount(0)
{
}

short Inventory::LoadInventory(ifstream& InputStream)
{
    short i;

    for (i = 0; i < 100; i ++ )
    {
        m_Stock[i].Read(InputStream);
        if (InputStream.fail() != 0)
            break;
    }

    m_StockCount = i;
    return m_StockCount;
}

StockItem Inventory::FindItem(string UPC)
{
    short i;
    bool Found = false;

    for (i = 0; i < m_StockCount; i ++ )
    {
        if (m_Stock[i].CheckUPC(UPC) == true)
        {
            Found = true;
            break;
        }
    }

    if (Found == true)
        return m_Stock[i];

    return StockItem();
}

bool Inventory::UpdateItem(StockItem Item)
{
    string UPC = Item.GetUPC();

    short i;
    bool Found = false;

    for (i = 0; i < m_StockCount; i ++ )
    {
        if (m_Stock[i].CheckUPC(UPC) == true)
        {
            Found = true;
            break;
        }
    }

    if (Found == true)
        m_Stock[i] = Item;

    return Found;
}
```

}

1 2 3 4 5 6 7 8 9 10

```
//global variable, not explicitly initialized
```

```
short count;
```

```
#include <iostream.h>
```

```
short mess()
```

```
{  
    short xyz;
```

```
    xyz = 5;
```

```
    return 0;
```

```
}
```

```
short counter()
```

```
{  
    count ++;
```

```
    cout << count << " ";
```

```
    return 0;
```

```
}
```

```
int main()
```

```
{  
    short i;
```

```
    for (i = 0; i < 10; i ++)
```

```
    {  
        mess();
```

```
        counter();
```

```
    }
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

```
//global variable, explicitly initialized
```

```
short count = 0;
```

```
#include <iostream.h>
```

```
short mess()
```

```
{  
    short xyz;
```

```
    xyz = 5;
```

```
    return 0;
```

```
}
```

```
short counter()
```

```
{  
    count ++;
```

```
    cout << count << " ";
```

```
    return 0;
```

```
}
```

```
int main()
```

```
{  
    short i;
```

```
    for (i = 0; i < 10; i ++)
```

```
    {  
        mess();
```

```
        counter();
```

```
    }
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

```
//static local variable, not explicitly initialized
```

```
#include <iostream.h>
```

```
short mess()  
{  
    short xyz;  
  
    xyz = 5;  
  
    return 0;  
}
```

```
short counter()  
{  
    static short count;  
  
    count ++;  
  
    cout << count << " ";  
  
    return 0;  
}
```

```
int main()  
{  
    short i;  
  
    for (i = 0; i < 10; i ++)  
        {  
            mess();  
            counter();  
        }  
  
    cout << endl;  
  
    return 0;  
}
```

```
//static local variable, explicitly initialized
```

```
#include <iostream.h>
```

```
short mess()
```

```
{  
    short xyz;  
  
    xyz = 5;  
  
    return 0;  
}
```

```
short counter()
```

```
{  
    static short count = 0;  
  
    count ++;  
  
    cout << count << " ";  
  
    return 0;  
}
```

```
int main()
```

```
{  
    short i;  
  
    for (i = 0; i < 10; i ++)  
    {  
        mess();  
        counter();  
    }  
  
    cout << endl;  
  
    return 0;  
}
```

6 6 6 6 6 6 6 6 6 6

```
//auto local variable, uninitialized
```

```
#include <iostream.h>
```

```
short mess()  
{  
    short xyz;  
  
    xyz = 5;  
  
    return 0;  
}
```

```
short counter()  
{  
    short count;  
  
    count ++;  
  
    cout << count << " ";  
  
    return 0;  
}
```

```
int main()  
{  
    short i;  
  
    for (i = 0; i < 10; i ++)  
        {  
            mess();  
            counter();  
        }  
  
    cout << endl;  
  
    return 0;  
}
```

1 1 1 1 1 1 1 1 1 1

```
//auto local variable, initialized
```

```
#include <iostream.h>
```

```
short mess()  
{  
    short xyz;  
  
    xyz = 5;  
  
    return 0;  
}
```

```
short counter()  
{  
    short count = 0;  
  
    count ++;  
  
    cout << count << " ";  
  
    return 0;  
}
```

```
int main()  
{  
    short i;  
  
    for (i = 0; i < 10; i ++)  
        {  
            mess();  
            counter();  
        }  
  
    cout << endl;  
  
    return 0;  
}
```

```
#include <iostream.h>

short Average(short First, short Second)
{
    short Result;

    Result = (First + Second) / 2;

    return Result;
}

int main()
{
    short FirstWeight;
    short SecondWeight;
    short FirstAge;
    short SecondAge;
    short AverageWeight;
    short AverageAge;

    cout << "Please type in the first weight: ";
    cin >> FirstWeight;

    cout << "Please type in the second weight: ";
    cin >> SecondWeight;

    AverageWeight = Average(FirstWeight, SecondWeight);

    cout << "Please type in the first age: ";
    cin >> FirstAge;

    cout << "Please type in the second age: ";
    cin >> SecondAge;

    AverageAge = Average(FirstAge, SecondAge);

    cout << "The average weight was: " << AverageWeight << endl;
    cout << "The average age was: " << AverageAge << endl;

    return 0;
}
```

Low: 1111111111
Middle: ABCDEFGHIJ
Alias: ABCDEFGHIJ
High: mnopqrst00

```
#include <iostream.h>

int main()
{
    char High[10];
    char Middle[10];
    char Low[10];
    char* Alias;
    short i;

    for (i = 0; i < 10; i ++ )
        {
            Middle[i] = 'A' + i;
            High[i] = '0';
            Low[i] = '1';
        }

    Alias = Middle;

    for (i = 10; i < 20; i ++ )
        {
            Alias[i] = 'a' + i;
        }

    cout << "Low: ";
    for (i = 0; i < 10; i ++ )
        cout << Low[i];

    cout << endl;

    cout << "Middle: ";
    for (i = 0; i < 10; i ++ )
        cout << Middle[i];

    cout << endl;

    cout << "Alias: ";
    for (i = 0; i < 10; i ++ )
        cout << Alias[i];

    cout << endl;

    cout << "High: ";
    for (i = 0; i < 10; i ++ )
        cout << High[i];

    cout << endl;
}
```

```
#include <iostream.h>

int main()
{
    short x;
    char y;

    x = 1;
    y = 'A';

    cout << "On test #" << x << ", your mark is: " << y << endl;

    return 0;
}
```

```
#include <iostream.h>

short count;

short counter()
{
    count ++;

    cout << count << " ";

    return 0;
}

int main()
{
    short i;

    for (i = 0; i < 10; i ++ )
        counter();

    return 0;
}
```

```
#include <iostream.h>

short count = 0;

short counter()
{
    count ++;

    cout << count << " ";

    return 0;
}

int main()
{
    short i;

    for (i = 0; i < 10; i ++ )
        counter();

    return 0;
}
```

```
#include <iostream.h>

short counter()
{
    static short count;

    count ++;

    cout << count << " ";

    return 0;
}

int main()
{
    short i;

    for (i = 0; i < 10; i ++ )
        counter();

    return 0;
}
```

```
#include <iostream.h>

short counter()
{
    static short count = 0;

    count ++;

    cout << count << " ";

    return 0;
}

int main()
{
    short i;

    for (i = 0; i < 10; i ++ )
        counter();

    return 0;
}
```

```
#include <iostream.h>

short counter()
{
    short count;

    count ++;

    cout << count << " ";

    return 0;
}

int main()
{
    short i;

    for (i = 0; i < 10; i ++ )
        counter();

    return 0;
}
```

```
#include <iostream.h>

short counter()
{
    short count = 0;

    count ++;

    cout << count << " ";

    return 0;
}

int main()
{
    short i;

    for (i = 0; i < 10; i ++ )
        counter();

    return 0;
}
```

As in third printing.

```
/*TITLE common header file*/

****keyword-flag*** "%v %f %n" */
/* "1 7-May-96,0:15:02 COMMON.H" */

****revision-history****/
/*1 COMMON.H 7-May-96,0:15:02 As of MAINPT. */
****revision-history****/

#ifndef COMMON_H
#define COMMON_H

#include <limits.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>

#define min(a,b)    (((a) < (b)) ? (a) : (b))

#define TRUE 1
#define FALSE 0

typedef unsigned short ArrayIndex;
typedef unsigned long Ulong;
typedef Ulong AccountNumber;

#include "wassert.h"
#include "vector.h"

#endif
```

basic00.cc
basic01.cc
basic02.cc
basic03.cc
basic04.cc
basic05.cc
basic06.cc
basic07.cc
basic08.cc
basic09.cc
birthday.cc
calc1.cc
common.h
count1.cc
count2.cc
count3.cc
count4.cc
count5.cc
count6.cc
cout1.cc
dangchar.cc
func1.cc
inita.cc
initb.cc
initc.cc
initd.cc
inite.cc
initf.cc
invent1.cc
invent1.h
invent2.cc
invent2.h
item1.cc
item1.h
item2.cc
item2.h
item4.cc
item4.h
item5.cc
item5.h
item6.cc
item6.h
itemtst1.cc
itemtst2.cc
itemtst3.cc
itemtst4.cc
itemtst5.cc
itemtst6.cc
mknorm.bat
morbas00.cc
morbas01.cc
morbas02.cc
morbas03.cc
morbas04.cc
nofunc.cc
pump1.cc
pump1a.cc
pump2.cc
scopclas.cc
strcmp.cc
strex1.cc
strex2.cc
strex3.cc
strex5.cc
strex6.cc
string1.cc

string1.h
string3.cc
string3.h
string4.cc
string4.h
string5.cc
string5.h
string5a.cc
string5x.cc
string5y.cc
string6.cc
string6.h
strsort1.cc
strtst1.cc
strtst3.cc
strtst3a.cc
strtst4.cc
strtst5.cc
strtst5a.cc
strtst5x.cc
testpare.cc
vect1.cc
vect2.cc
vect2a.cc
vect3.cc
vector.h
wassert.cc
wassert.h

allcode.bat
basic00.cc
basic01.cc
basic02.cc
basic03.cc
basic04.cc
basic05.cc
basic06.cc
basic07.cc
basic08.cc
basic09.cc
birthday.cc
calc1.cc
common.h
copyarch.bat
copyexe.bat
copysamp.bat
count1.cc
count2.cc
count3.cc
count4.cc
count5.cc
count6.cc
cout1.cc
dangchar.cc
func1.cc
inita.cc
initb.cc
initc.cc
initd.cc
inite.cc
initf.cc
invent1.cc
invent1.h
invent2.cc
invent2.h
item1.cc
item1.h
item2.cc
item2.h
item4.cc
item4.h
item5.cc
item5.h
item6.cc
item6.h
itemtst1.cc
itemtst2.cc
itemtst3.cc
itemtst4.cc
itemtst5.cc
itemtst6.cc
mkallnr.bat
mkallnru.bat
mkmore.bat
mkmoreu.bat
mknorm.bat
mknormu.bat
mktemp.bat
mktempu.bat
morbas00.cc
morbas01.cc
morbas02.cc
morbas03.cc
morbas04.cc
nofunc.cc

pump1.cc
pump1a.cc
pump2.cc
scopclas.cc
strcmp.cc
strex1.cc
strex2.cc
strex3.cc
strex5.cc
strex6.cc
string1.cc
string1.h
string3.cc
string3.h
string4.cc
string4.h
string5.cc
string5.h
string5a.cc
string5x.cc
string5y.cc
string6.cc
string6.h
strsort1.cc
strtst1.cc
strtst3.cc
strtst3a.cc
strtst4.cc
strtst5.cc
strtst5a.cc
strtst5x.cc
temp.bat
testpare.cc
vect1.cc
vect2.cc
vect2a.cc
vect3.cc
vector.h
wassert.cc
wassert.h
whoscode.bat

```
The value of j in Calc is: 0  
The value of j in main is: 12  
The value of j in Calc is: 5  
The value of j in main is: 23  
The value of j in Calc is: 16  
The value of j in main is: 40
```

```
#include <iostream.h>

short i;

short Calc(short x, short y)
{
static short j = 0;

    cout << "The value of j in Calc is: " << j << endl;

    i ++;

    j = x + y + j;

    return j;
}

int main()
{
    short j;

    for (i = 0; i < 5; i ++ )
        {
            j = Calc(i + 5, i * 2) + 7;
            cout << "The value of j in main is: " << j << endl;
        }

    return 0;
}
```

```
#include <iostream.h>
#include <fstream.h>

short Birthday(short age)
{
    age ++;
    return age;
}

int main()
{
    short x;
    short y;

    x = 46;
    y = Birthday(x);

    ofstream prout("lpt1");

    prout << "Your age was: " << x << endl;
    prout << "Happy Birthday: your age is now " << y << endl;
    prout << '\f' << endl;

    prout.close();

    return 0;
}
```

```
#include <iostream.h>

short Birthday(short age)
{
    age ++;
    return age;
}

int main()
{
    short x;
    short y;

    x = 46;
    y = Birthday(x);

    cout << "Your age was: " << x << endl;
    cout << "Happy Birthday: your age is now " << y << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"

int main()
{
    short x;

    cout << "Elena can increase her $10 allowance each week ";
    cout << "by adding new chores." << endl;

    cout << "For every extra chore Elena does, she gets ";
    cout << "another dollar." << endl;

    cout << "How many extra chores were done? " << endl;
    cin >> x;

    if (x==0)
    {
        cout << "There is no extra allowance for Elena ";
        cout << "this week. " << endl;
    }
    else
    {
        cout << "Elena will now earn " << 10 + x;
        cout << " dollars this week." << endl;
    }

    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"

int main()
{
    string answer;

    cout << "Please respond to the following statement ";
    cout << "with either true or false\n";

    cout << "Susan is the world's most tenacious novice.\n";
    cin >> answer;

    if (answer != "true")
        if (answer != "false")
            cout << "Please answer with either true or false.";

    if (answer == "true")
        cout << "Your answer is correct\n";

    if (answer == "false")
        cout << "Your answer is erroneous\n";

    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"

int main()
{
    string name;
    short age;

    cout << "What is your name? ";
    cin >> name;

    cout << "Thank you, " << name << endl;

    cout << "What is your age? ";
    cin >> age;

    if (age < 47)
        cout << "My, what a youngster!" << endl;
    else
        cout << "Hi, Granny!" << endl;

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short n;

    cout << "Excluding yourself, please type the ";
    cout << "number of guests in your dinner party.\n";

    cin >> n;

    if (n>20)
        cout << "Sorry, your party is too large. ";
    else
        cout << "A table for " << n+1 << " is ready. ";

    return 0;
}
```

```
#include <iostream.h>

int main()
{
    short n;

    cout << "Please type in the number of guests ";
    cout << "of your dinner party. ";
    cin >> n;

    cout << "A table for " << n+1 << "is ready. ";

    return 0;
}
```

```
#include <iostream.h>
```

```
int main()
```

```
{
```

```
    short Secret;
```

```
    short Guess;
```

```
    Secret = 3;
```

```
    cout << "Try and guess my number. Hint: It's from 0 to 9" << endl;
```

```
    cin >> Guess;
```

```
    while (Guess != Secret)
```

```
    {
```

```
        cout << "Sorry, that's not correct." << endl;
```

```
        cin >> Guess;
```

```
    }
```

```
    cout << "You guessed right!" << endl;
```

```
    return 0;
```

```
}
```

```
#include <iostream.h>

int main()
{
    short balance;

    cout << "Please enter your bank balance: ";
    cin >> balance;

    if (balance < 10000)
        cout << "Please remit $20 service charge." << endl;
    else
        cout << "Have a nice day!" << endl;

    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"
```

```
int main()
{
    string s1;
    string s2;

    cin >> s1;
    cin >> s2;

    cout << s1;
    cout << " ";
    cout << s2;

    return 0;
}
```

```
#include <iostream.h>
#include "string6.h"
```

```
int main()
{
    string s1;
    string s2;

    s1 = "This is a test ";
    s2 = "and so is this.";

    cout << s1;
    cout << s2;

    return 0;
}
```

```
#include "string6.h"
```

```
int main()
```

```
{  
    char c1;  
    char c2;  
    string s1;  
    string s2;  
  
    c1 = 'A';  
    c2 = c1;  
  
    s1 = "This is a test ";  
    s2 = "and so is this.";  
  
    return 0;  
}
```

```
#include <iostream.h>
int main()
{
    short FirstAge;
    short SecondAge;
    short Result;

    cout << "Please enter the first age: ";
    cin >> FirstAge;

    cout << "Please enter the second age: ";
    cin >> SecondAge;

    if (FirstAge > SecondAge)
        Result = FirstAge - SecondAge;
    else
        Result = SecondAge - FirstAge;

    cout << "The difference in ages is: " << Result;
    return 0;
}
```

Index of /whos/code

	Name	Last modified	Size	Description
	Parent Directory	06-Oct-2002 11:18	-	
	mkallnru.out	06-Oct-2002 11:19	19k	
	itemtst6.gpr	06-Oct-2002 11:19	11k	
	itemtst2.gpr	06-Oct-2002 11:19	11k	
	itemtst5.gpr	06-Oct-2002 11:19	11k	
	readme.txt	06-Oct-2002 11:19	10k	
	itemtst4.gpr	06-Oct-2002 11:19	10k	
	itemtst3.gpr	06-Oct-2002 11:19	10k	
	basic00.gpr	06-Oct-2002 11:18	10k	
	morbas01.gpr	06-Oct-2002 11:19	9k	
	morbas00.gpr	06-Oct-2002 11:19	9k	
	vect2a.gpr	06-Oct-2002 11:19	9k	
	vect3.gpr	06-Oct-2002 11:19	9k	
	vect2.gpr	06-Oct-2002 11:19	9k	
	vect1.gpr	06-Oct-2002 11:19	9k	
	itemtst1.gpr	06-Oct-2002 11:19	9k	
	basic08.gpr	06-Oct-2002 11:19	9k	
	basic07.gpr	06-Oct-2002 11:19	9k	
	basic02.gpr	06-Oct-2002 11:18	9k	
	basic01.gpr	06-Oct-2002 11:18	9k	
	strcmp.gpr	06-Oct-2002 11:19	9k	
	birthprt.gpr	06-Oct-2002 11:19	9k	
	basic09.gpr	06-Oct-2002 11:19	9k	

 scopclas.gpr	06-Oct-2002	11:19	9k
 morbas04.gpr	06-Oct-2002	11:19	9k
 morbas03.gpr	06-Oct-2002	11:19	9k
 morbas02.gpr	06-Oct-2002	11:19	9k
 dangchar.gpr	06-Oct-2002	11:19	9k
 birthday.gpr	06-Oct-2002	11:19	9k
 basic06.gpr	06-Oct-2002	11:19	9k
 basic05.gpr	06-Oct-2002	11:19	9k
 basic04.gpr	06-Oct-2002	11:18	9k
 basic03.gpr	06-Oct-2002	11:18	9k
 pump1a.gpr	06-Oct-2002	11:19	9k
 nofunc.gpr	06-Oct-2002	11:19	9k
 count6.gpr	06-Oct-2002	11:19	9k
 count5.gpr	06-Oct-2002	11:19	9k
 count4.gpr	06-Oct-2002	11:19	9k
 count3.gpr	06-Oct-2002	11:19	9k
 count2.gpr	06-Oct-2002	11:19	9k
 count1.gpr	06-Oct-2002	11:19	9k
 pump2.gpr	06-Oct-2002	11:19	9k
 pump1.gpr	06-Oct-2002	11:19	9k
 initf.gpr	06-Oct-2002	11:19	9k
 inite.gpr	06-Oct-2002	11:19	9k
 initd.gpr	06-Oct-2002	11:19	9k
 initc.gpr	06-Oct-2002	11:19	9k
 initb.gpr	06-Oct-2002	11:19	9k
 inita.gpr	06-Oct-2002	11:19	9k
 func1.gpr	06-Oct-2002	11:19	9k

 cout1.gpr	06-Oct-2002 11:19	9k
 calc1.gpr	06-Oct-2002 11:19	9k
 test.gpr	06-Oct-2002 11:19	9k
 ages.gpr	06-Oct-2002 11:18	9k
 testpare.gpr	06-Oct-2002 11:19	9k
 strex6.gpr	06-Oct-2002 11:19	9k
 strex5.gpr	06-Oct-2002 11:19	9k
 strex3.gpr	06-Oct-2002 11:19	9k
 strex2.gpr	06-Oct-2002 11:19	9k
 strex1.gpr	06-Oct-2002 11:19	9k
 readme.dif	06-Oct-2002 11:19	7k
 itemtst2.out	06-Oct-2002 11:19	5k
 mkalltr.ouu	06-Oct-2002 11:19	4k
 string6.cc	06-Oct-2002 11:19	3k
 shop3.in	06-Oct-2002 11:19	3k
 vector.h	06-Oct-2002 11:19	3k
 shop2.in	06-Oct-2002 11:19	3k
 string5a.cc	06-Oct-2002 11:19	2k
 strcmp.cc	06-Oct-2002 11:19	2k
 itemtst6.cc	06-Oct-2002 11:19	2k
 string5.cc	06-Oct-2002 11:19	2k
 itemtst5.cc	06-Oct-2002 11:19	2k
 item6.cc	06-Oct-2002 11:19	2k
 string5x.cc	06-Oct-2002 11:19	2k
 itemtst4.cc	06-Oct-2002 11:19	2k
 invent2.cc	06-Oct-2002 11:19	1k
 item5.cc	06-Oct-2002 11:19	1k

 itemtst6.gdt	06-Oct-2002 11:19	1k
 itemtst5.gdt	06-Oct-2002 11:19	1k
 itemtst4.gdt	06-Oct-2002 11:19	1k
 itemtst2.gdt	06-Oct-2002 11:19	1k
 basic02.gdt	06-Oct-2002 11:18	1k
 basic01.gdt	06-Oct-2002 11:18	1k
 itemtst1.gdt	06-Oct-2002 11:19	1k
 invent1.cc	06-Oct-2002 11:19	1k
 basic00.gdt	06-Oct-2002 11:18	1k
 basic03.gdt	06-Oct-2002 11:18	1k
 strcmp.gdt	06-Oct-2002 11:19	1k
 vect3.cc	06-Oct-2002 11:19	1k
 vect1.gdt	06-Oct-2002 11:19	1k
 scopclas.cc	06-Oct-2002 11:19	1k
 code.new	06-Oct-2002 11:19	1k
 morbas04.gdt	06-Oct-2002 11:19	1k
 itemtst3.gdt	06-Oct-2002 11:19	1k
 basic09.gdt	06-Oct-2002 11:19	1k
 ages.gdt	06-Oct-2002 11:18	1k
 strex6.gdt	06-Oct-2002 11:19	1k
 pump1a.gdt	06-Oct-2002 11:19	1k
 pump2.gdt	06-Oct-2002 11:19	1k
 item4.cc	06-Oct-2002 11:19	1k
 strex5.gdt	06-Oct-2002 11:19	1k
 vect2a.cc	06-Oct-2002 11:19	1k
 itemtst3.cc	06-Oct-2002 11:19	1k
 vect2.cc	06-Oct-2002 11:19	1k

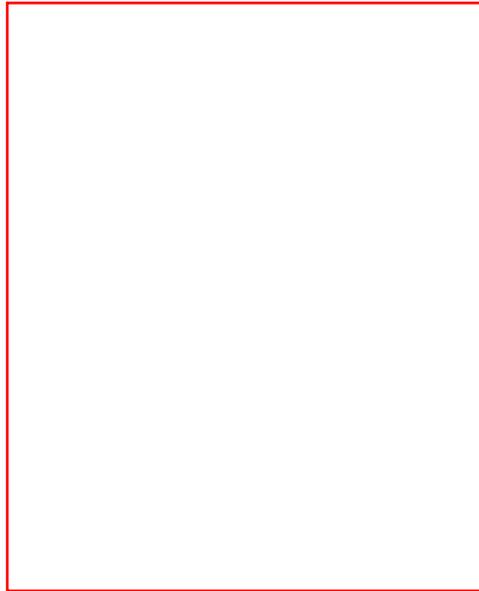
 code.old	06-Oct-2002 11:19	1k
 pump2.cc	06-Oct-2002 11:19	1k
 vect1.cc	06-Oct-2002 11:19	1k
 strsort1.cc	06-Oct-2002 11:19	1k
 pumpla.cc	06-Oct-2002 11:19	1k
 string4.cc	06-Oct-2002 11:19	1k
 wassert.h	06-Oct-2002 11:19	1k
 item2.cc	06-Oct-2002 11:19	1k
 func1.cc	06-Oct-2002 11:19	1k
 string3.cc	06-Oct-2002 11:19	1k
 dangchar.cc	06-Oct-2002 11:19	1k
 morbas01.gdt	06-Oct-2002 11:19	1k
 basic06.gdt	06-Oct-2002 11:19	1k
 morbas00.gdt	06-Oct-2002 11:19	1k
 basic08.gdt	06-Oct-2002 11:19	1k
 basic07.gdt	06-Oct-2002 11:19	1k
 vect2a.gdt	06-Oct-2002 11:19	1k
 vect3.gdt	06-Oct-2002 11:19	1k
 vect2.gdt	06-Oct-2002 11:19	1k
 strex2.gdt	06-Oct-2002 11:19	1k
 testpare.gdt	06-Oct-2002 11:19	1k
 scopclas.gdt	06-Oct-2002 11:19	1k
 morbas03.gdt	06-Oct-2002 11:19	1k
 morbas02.gdt	06-Oct-2002 11:19	1k
 dangchar.gdt	06-Oct-2002 11:19	1k
 birthprt.gdt	06-Oct-2002 11:19	1k
 birthday.gdt	06-Oct-2002 11:19	1k

 basic05.gdt	06-Oct-2002 11:19	1k
 basic04.gdt	06-Oct-2002 11:18	1k
 strex3.gdt	06-Oct-2002 11:19	1k
 strex1.gdt	06-Oct-2002 11:19	1k
 nofunc.gdt	06-Oct-2002 11:19	1k
 count6.gdt	06-Oct-2002 11:19	1k
 count5.gdt	06-Oct-2002 11:19	1k
 count4.gdt	06-Oct-2002 11:19	1k
 count3.gdt	06-Oct-2002 11:19	1k
 count2.gdt	06-Oct-2002 11:19	1k
 count1.gdt	06-Oct-2002 11:19	1k
 pump1.gdt	06-Oct-2002 11:19	1k
 initf.gdt	06-Oct-2002 11:19	1k
 inite.gdt	06-Oct-2002 11:19	1k
 initd.gdt	06-Oct-2002 11:19	1k
 initc.gdt	06-Oct-2002 11:19	1k
 initb.gdt	06-Oct-2002 11:19	1k
 inita.gdt	06-Oct-2002 11:19	1k
 func1.gdt	06-Oct-2002 11:19	1k
 cout1.gdt	06-Oct-2002 11:19	1k
 calc1.gdt	06-Oct-2002 11:19	1k
 test.gdt	06-Oct-2002 11:19	1k
 nofunc.cc	06-Oct-2002 11:19	1k
 string1.cc	06-Oct-2002 11:19	1k
 item1.cc	06-Oct-2002 11:19	1k
 pump1.cc	06-Oct-2002 11:19	1k
 string6.h	06-Oct-2002 11:19	1k

 basic09.cc	06-Oct-2002 11:19	1k
 common.h	06-Oct-2002 11:19	1k
 item6.h	06-Oct-2002 11:19	1k
 basic08.cc	06-Oct-2002 11:19	1k
 itemtst2.cc	06-Oct-2002 11:19	1k
 item5.h	06-Oct-2002 11:19	1k
 item4.h	06-Oct-2002 11:19	1k
 string5.h	06-Oct-2002 11:19	1k
 morbas04.cc	06-Oct-2002 11:19	1k
 ages.cc	06-Oct-2002 11:18	1k
 calcl.cc	06-Oct-2002 11:19	1k
 strex1.cc	06-Oct-2002 11:19	1k
 basic07.cc	06-Oct-2002 11:19	1k
 initd.cc	06-Oct-2002 11:19	1k
 initc.cc	06-Oct-2002 11:19	1k
 basic04.cc	06-Oct-2002 11:18	1k
 birthprt.cc	06-Oct-2002 11:19	1k
 invent2.h	06-Oct-2002 11:19	1k
 initf.cc	06-Oct-2002 11:19	1k
 inite.cc	06-Oct-2002 11:19	1k
 inita.cc	06-Oct-2002 11:19	1k
 initb.cc	06-Oct-2002 11:19	1k
 basic06.cc	06-Oct-2002 11:19	1k
 morbas00.cc	06-Oct-2002 11:19	1k
 strex5.cc	06-Oct-2002 11:19	1k
 morbas01.cc	06-Oct-2002 11:19	1k
 item2.h	06-Oct-2002 11:19	1k

 strtst3a.err	06-Oct-2002 11:19	1k
 strex2.cc	06-Oct-2002 11:19	1k
 basic03.cc	06-Oct-2002 11:18	1k
 item1.h	06-Oct-2002 11:19	1k
 strtst5.cc	06-Oct-2002 11:19	1k
 birthday.cc	06-Oct-2002 11:19	1k
 strtst5x.cc	06-Oct-2002 11:19	1k
 invent1.h	06-Oct-2002 11:19	1k
 basic05.cc	06-Oct-2002 11:18	1k
 strex3.cc	06-Oct-2002 11:19	1k
 string4.h	06-Oct-2002 11:19	1k
 count3.cc	06-Oct-2002 11:19	1k
 count4.cc	06-Oct-2002 11:19	1k
 strex6.cc	06-Oct-2002 11:19	1k
 count1.cc	06-Oct-2002 11:19	1k
 count5.cc	06-Oct-2002 11:19	1k
 count2.cc	06-Oct-2002 11:19	1k
 string3.h	06-Oct-2002 11:19	1k
 itemtst1.cc	06-Oct-2002 11:19	1k
 count6.cc	06-Oct-2002 11:19	1k
 morbas03.cc	06-Oct-2002 11:19	1k
 strtst4.cc	06-Oct-2002 11:19	1k
 string1.h	06-Oct-2002 11:19	1k
 basic01.cc	06-Oct-2002 11:18	1k
 basic00.cc	06-Oct-2002 11:18	1k
 string5y.out	06-Oct-2002 11:19	1k
 basic02.cc	06-Oct-2002 11:18	1k

 strtst3.cc	06-Oct-2002 11:19	1k
 calc1.out	06-Oct-2002 11:19	1k
 cout1.cc	06-Oct-2002 11:19	1k
 string5x.out	06-Oct-2002 11:19	1k
 strtst1.cc	06-Oct-2002 11:19	1k
 scopclas.out	06-Oct-2002 11:19	1k
 strtst3a.cc	06-Oct-2002 11:19	1k
 morbas02.cc	06-Oct-2002 11:19	1k
 string5y.cc	06-Oct-2002 11:19	1k
 testpare.cc	06-Oct-2002 11:19	1k
 test.cc	06-Oct-2002 11:19	1k
 dangchar.out	06-Oct-2002 11:19	1k
 common.in	06-Oct-2002 11:19	1k
 initf.out	06-Oct-2002 11:19	1k
 inite.out	06-Oct-2002 11:19	1k
 initd.out	06-Oct-2002 11:19	1k
 initc.out	06-Oct-2002 11:19	1k
 initb.out	06-Oct-2002 11:19	1k
 inita.out	06-Oct-2002 11:19	1k



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

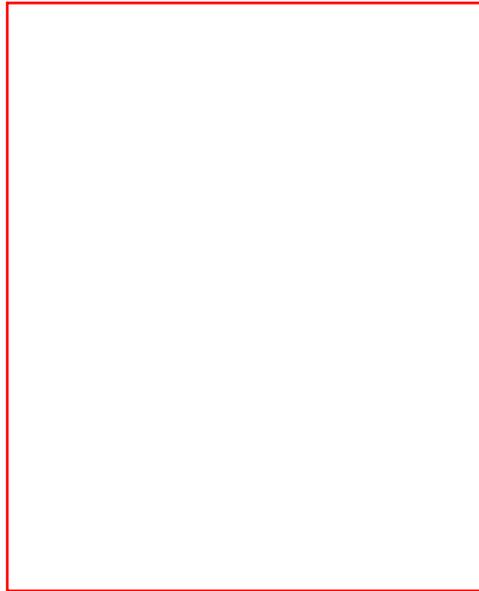
This book is dedicated to Susan Patricia Caffee Heller, the light of my life. Without her, this book would not be what it is; even more important, I would not be what I am: a happy man.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Acknowledgements

I'd like to thank all those readers who have provided feedback on the earlier printings of this book, especially those who have posted reviews on Amazon.com; their contributions have made this a better book and have helped me make the transition to full-time writing.

I'd also like to thank Sam Porter and Kurt Matti, our hosts on our honeymoon in Europe, for all the hospitality they showed us, most especially for letting us use their telephone lines for our email.

Of course, I'm indebted to Ed Yourdon for his glorious Foreword. I'm happy that the book seems to have lived up to his praise.

Finally, my editor at AP Professional, Tom Stone, has been everything that a technical author could hope for (and most don't get).

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Letter from a Novice

One day last March I found myself reading this little message on my computer monitor:

[#: 288196 S10/Have You Heard?]

[25-Mar-95 20:34:55]

[Sb: Readers for my new book]

[Fm: Steve Heller 71101,1702]

[To: all]

Hi!

I'm looking for some readers for a book I'm working on, which teaches people how to program, using C++ as the language. The ideal candidate is someone who wants to learn how to program, but has little or no knowledge or experience in programming. I can't pay anything, but participants will learn how to program, and will be acknowledged in the book, as well as getting an autographed copy. If you're interested, please reply by e-mail.

Steve

As I considered my response to this message I felt a little trepidation for what was to come. I have only known one profession: nursing. I had owned and used a computer for only a little over two years at the time and thought DOS was too difficult to understand. Not only had I no prior knowledge of programming, I had very little knowledge of computers in general. Yet, what knowledge I did have of the computer fed my curiosity for more, and as my love of the computer grew, it soon was apparent I had no choice. I replied by e-mail.

Evidently I was considered an ideal candidate as I did not wait long for acceptance as a test reader. I then embarked upon a project that I thought would last a few weeks to a few months. It was my expectation that I would read a chapter at a time, submit a few comments occasionally and nothing more. Never in my wildest dreams would I have ever expected that I would end up reading every page of this book with utmost attention to detail, leaving no word unturned, no concept overlooked, no hair on my head left unpulled for the next nine months of my life. But, if we were going to make this book as clear as possible for anyone who wanted to read it, there was no other way.

The process of writing this book was an enormous effort on both our parts. Neither Steve nor I could have ever imagined the type of dialogue that would ensue. It just happened; as I asked the questions, he answered the questions and I asked again. The exchange would continue until we were both satisfied that I "had it". When that happened, Steve knew the right wording for the book, and I could move on to the next concept. It was an experience like no other in my life. It was a time filled with confusion, frustration, anger, acceptance, understanding, and joy. The process often gave cause for others to question my motivation for doing it. Admittedly, at times my frustration was such that I

wanted to give up. But it was my mountain, and I had to climb it. I would not accept defeat.

The material was not the only source of my frustration. There was the inherent difficulty that Steve and I had in keeping communication flowing and speaking the same language. We found we had very different writing styles which added another obstacle to our undertaking. He, the ultimate professional, and I, the incorrigible misfit, finally managed a happy medium. We corresponded on a daily basis almost exclusively in e-mail. Through that medium it was our challenge to get into each other's minds. He had to learn how I thought and I had to learn how he thought, not an easy task for an "expert" and a "novice" who were total strangers.

What you are about to read is the refinement of the writings of the mind of a genius filtered through the mind of a novice. Ideally, the result of this combination has produced the best possible information written in the most understandable form. The book as you see it is considerably different from the original version, as a result of this filtering process. For that same reason, it is also different from any other book of its kind. To our knowledge, this is the first time that someone with no previous background in programming has been enlisted in the creation of such a book. Of course, it took tutoring, but that tutoring is what led to the simplification of the text so that future novices could enjoy a relatively painfree reading experience.

During the first few months of this process, I had to take it on faith that somehow all of this material would eventually make sense. But late one quiet night while first reading about object-oriented programming, (the creation of "classes") I was abruptly shaken by the most incredible feeling. I was held spellbound for a few moments, I gasped and was gripped with the sudden realization and awe of the profound beauty of the code. I had finally caught a glimpse of what programming was *really* all about. This experience later led me to write the following quotation to my sister:

Programming is SOOOO gorgeous. So fine, so delicate. It is so beautifully spun with silk threads, tiny and intricate. Yet like silk it can be strong and powerful, it all depends on how you use it.

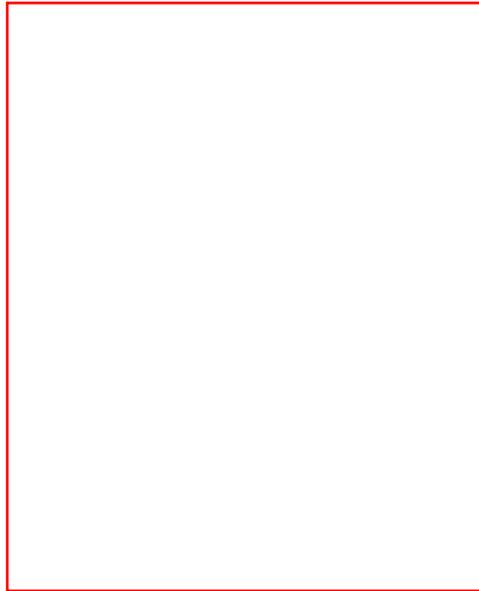
At last, I had "gotten it". Within these pages the beauty lies dormant waiting for those who embark on the task of learning, to be viewed only when the work has been done and the mountain scaled. It is an exquisite panorama and a most worthy journey.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Foreword

Argh! Another "Programming for Dummies" book! I have to admit that when Steve Heller prevailed upon me to read his manuscript and prepare a foreword for this book, I had serious misgivings. Steve is a talented, articulate fellow, and I've enjoyed corresponding with him on the Internet for the past few years -- ever since he gleefully pointed out in one of his earlier books (*Efficient C/C++ Programming*) that I had made the outrageously shortsighted comment in my 1975 textbook, *Techniques of Program Structure and Design*, that "unless you're very rich or very eccentric, you'll never have the luxury of owning your own computer."

Still, I dreaded the task of reading his entry-level programming book: almost every one I've read during my career has either been deadly boring, or childishly condescending, or both. Imagine my surprise, then, when I discovered that *Who's Afraid of C++?* was not only a *good* book, but an exciting one. Because I'm writing these words in mid-February 1996, you won't be impressed when I tell you that it's the best computer book that I've read in 1996 -- so let me put it more strongly: this is the best technical book I've read since *Zen and the Art of Motorcycle Maintenance* appeared in the mid-1970s.

Before I explain this rather bizarre analogy, let me address the central theme of *Who's Afraid of C++?*: learning a new programming language. Anyone who has gone through this process knows that it isn't easy. True, there are languages like LOGO and Basic that can be taught to novices within short periods of time; one of my most enjoyable experiences in the computer field was teaching LOGO to a group of 6- and 7- year old children during a two-week summer class on Fire Island. But languages like C++ -- the subject of this book -- are far more difficult, not only for novices but for veterans of other programming languages. Ask any COBOL programmer whether he found it difficult to absorb the intricate syntax and arcane vocabulary of C++ and you're likely to get a groan.

But for many of us, C++ has become a prerequisite to continued employment in the software industry. Groan though they may, more and more COBOL programmers are making the transition -- for the simple reason that mainframe applications are being replaced by PC and client-server apps, and a lot of them are written in C++. Even if the main part of an application is written in a simpler language like Visual Basic, it's likely that some portions (e.g., the VBX components) will have to be programmed in C or C++. I often joke that C++ is the assembly language of the 90s, but I sometimes forget to remind my friends that assembly language was the first language I learned, and I would have been very nervous trying to program in any higher-level language if I didn't have a good idea of what was going on at the level of hardware registers and memory addresses.

Does this mean that all of today's veteran programmers will be required to learn C++? Well, perhaps not: after all, vast numbers of programmers *have* managed to make a comfortable living by creating applications in high-level languages without really understanding what was going on "under the covers." Judging from the employment ads in the newspapers, you can get a job today if you speak Visual Basic, PowerBuilder, or Smalltalk; but your odds of getting (or keeping) a programming job are usually *much* better if you also know C++. And for the beleaguered COBOL programmer, that's a key point to remember: whether it's fair or not, languages like Visual Basic are often regarded as "toys", while C++ is considered a "serious" language for building industrial-strength applications.

I got an inkling of the nature of this sea-change in programming languages in late 1995, at a panel session discussing IBM's newly-released version of object-oriented COBOL. It's an exciting, powerful

new language, and I think that one could make a strong argument that OO-COBOL is a more logical migration path for today's legacy COBOL programmers and vintage-1972 application programs than any other alternative. Nevertheless, when I asked the audience of some 100 people -- all of whom were COBOL fans of one kind or another -- whether they would advise their children to learn COBOL if their children intended to pursue a programming profession, less than 5 percent raised their hands. I didn't have the opportunity to see how many would have recommended C++, but I'm virtually certain the number of raised hands would have been far higher.

In mid-1995, another reason for learning C++ appeared on the horizon without any advance warning: the introduction of Sun Microsystems' "Java" language. Unless you've been living in a cave for the past few years, you know about the frenzy surrounding all aspects of the Internet and the World Wide Web; and you may have heard about Java as the language that promises to bring "live content" to Web pages. As I write this foreword, it's too early to tell whether Java really will revolutionize the Internet to the degree promised by its supporters, but there is one thing for sure: Java is mostly a subset of C++, and if you've learned C++, learning Java will be much easier.

On the other hand, that raises an interesting question: why not learn Java first, and just forget about C++? It's similar to the argument one often hears about the relationship between C and C++. Presumably, it's a lot easier for an experienced C programmer to learn the additional syntax of C++, though the object-oriented paradigm supported by C++ typically requires a great deal of "un-learning". Similarly, one can assume that an existing knowledge of C++ will make it easier to learn Java -- but since one of the most important aspects of Java is what it *eliminates* from the C++ language (e.g., pointers), there is a certain amount of un-learning required here as well.

All of this is particularly relevant for the novice programmer, who typically has no prior programming experience, and who barely has the time and patience to learn *one* language, let alone two or three. If we knew that we were going to be developing all of our applications to operate on the World Wide Web, and if we knew for certain that Sun would be able to withstand the onslaught of Microsoft and its Internet-friendly version of Visual Basic, maybe we could recommend to the novice that she skip C and C++ and just focus on Java ... or, following the lead of Netscape, perhaps learn only the "Java-lite" language known as JavaScript.

But we need a reality check here. For the time being, it's safe to assume that there will be a lot of programs that do *not* run on the Web. As of late 1995, the number of personal computers was estimated to be in the range of 200-300 million worldwide, but the number of Internet users has been pegged at the far lower figure of 30 million. Each of the 300 million PCs is a candidate for C++ programming (not to mention all the mainframes and mid-size machines). As for the Internet: well, if you express the number of Internet users as a fraction of the human race, it still rounds to zero. In any case, if Java does become a dominant language in the next few years, I'm confident that Steve Heller will be able to produce a modified version of *Who's Afraid of C++?* with the same dramatic success. Author's note: By a tremendous coincidence, I indeed have written a book called *Who's Afraid of Java?*, which is available at finer bookstores everywhere. None of this explains why an utter novice who does *not* intend to become a professional computer programmer should necessarily learn to program in C++, or why she should learn to program in *any* language. There has been a great deal of debate about this since PCs first invaded the mainstream of society a dozen years ago; colleges, high schools, and even some elementary schools have adopted -- and then sometimes abandoned -- a policy

that students should learn computer programming for the same reason they learn biology or chemistry or geometry. Indeed, one could argue that the odds are far better that you'll need to write a small computer program (or at least have a decent understanding of the logic behind an existing program) in your day-to-day life than that you'll have to prove the Pythagorean Theorem or dissect a frog or recite the chemical composition of aspirin.

But I'm not going to pursue this argument; after 30+ years in the software industry, I'm biased, and I suspect that professional educators are equally biased. And it really doesn't matter; regardless of the opinion of advocates and opponents of compulsory programming courses, the practical reality for the adults or college students who are the likely readers of this book is that it's a personal choice. And this is a key point: quite aside from the technical intricacies of syntax and structure, learning a programming language is an intensely personal experience. It's often agonizing, it's sometimes tedious (especially if you don't have the proper tools!), and it's occasionally fun. On very rare occasions -- perhaps only once a year, and sometimes only once or twice in one's career -- it's more than fun: it's exhilarating, it's a rush, it's a shot of pure adrenaline. For some, it's almost a religious experience. And religious experiences, no matter how much we talk about them or write about them, are ultimately *personal* experiences. In the case of programming, that personal experience is also solitary in most cases, for it occurs at 3 in the morning when you're exhausted and frazzled and at your wit's end, and about to give up -- and the program you've been sweating over finally gets up on its hind legs and *runs*.

It's the personal nature of the programming experience that makes this book such an unexpected and powerful masterpiece. As you'll see from the outset, it was *not* a solitary experience, but an ongoing dialogue between mentor and student -- not a make-believe student, but a real one. Perhaps it's a bit unfair comparing *Who's Afraid of C++?* to Robert Pirsig's *Zen and the Art of Motorcycle Maintenance*, for Mr. Heller and his student appear not to have suffered the same degree of psychic trauma as the characters in Mr. Pirsig's book. But the personal drama is intense nonetheless, and it would be interesting to read even if you had no interest in the technical subject matter. Assuming that you *do* intend to learn C++, the dialogue between the teacher and his often-frustrated student will suck you into a deeper level of involvement and participation than would ever have been possible in a normal "programming for dummies" book.

For those devoid of any emotion or interest in personal relationships, it's possible to skip over this part of the book and focus entirely on the technical stuff. It's all there, and it's all accurate, and it's all well-written. But there are other computer books of that ilk (even though very few at the introductory level), and the problem is that your mind begins to wander, about halfway through each chapter; by the end of the chapter, you can't even remember what the topic was. Such a fate is not likely with this book.

One last note: everything I've written here identifies Steve Heller as the sole author of *Who's Afraid of C++?*. But his student, Susan, is definitely more than a student; by the end of the book, she has become a full-fledged collaborator and approaches the status of co-author. I congratulate Steve for having written a superb piece of technical work, but I have some personal words of admiration for Susan: I offer you my congratulations for having the energy, the intellect, the tenacity, and the passion to bring this collaboration to its fruition. You did a helluva job -- and it promises to be a long collaboration indeed, stretching far beyond the final page of the book. As my friends in New York

City often like to say about such developments; *Mazel tov!*

Ed Yourdon

New York City

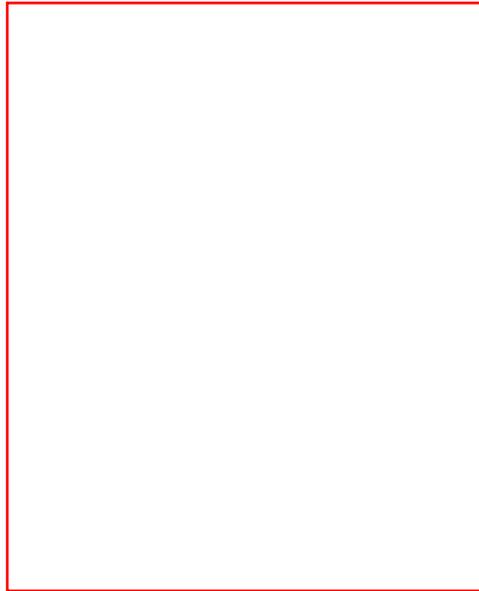
February 1996

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Prologue

Introduction to Programming

"Begin at the beginning, and go on till you come to the end: then stop." This method of telling a story is as good today as it was when the King of Hearts prescribed it to the White Rabbit. In this book, we must begin with you, the reader, since my job is to explain a technical subject to you. It might appear that I'm at a severe disadvantage; after all, I've never met you.

Nevertheless, I can make some pretty good guesses about you. You almost certainly own a computer and know how to use its most common application, word processing. If you use the computer in business, you probably also have an acquaintance with spreadsheets and perhaps some database experience as well. Now you have decided to learn how to program the computer yourself rather than relying completely on programs written by others. On the other hand, you might be a student using this book as a text in an introductory course on programming. In that case, you'll be happy to know that this book isn't written in the dry, overly academic style employed by many textbook writers. I hope that you will enjoy reading it, as my "test readers" have.

Whether you are using this book on your own or in school, there are many good reasons to learn how to program. You may have a problem that hasn't been solved by commercial software; you may want a better understanding of how commercial programs function so you can figure out how to get around their shortcomings and peculiarities; or perhaps you're just curious about how computers perform their seemingly magical feats. Whatever the initial reason, I hope you come to appreciate the great creative possibilities opened up by this most ubiquitous of modern inventions.¹ Before we begin, however, we should agree on definitions for some fundamental words in the computing field. Susan had some incisive observations about the power of words. Here is our exchange on that issue:

Susan: I will read something usually at face value, but often there is much more to it; that is why I don't get it. Then, when I go back and really think about what those words mean, it will make more sense. This book almost needs to be written in ALL CAPS to get the novice to pay closer attention to each and every word.

Steve: IMAGINE WRITING A BOOK IN ALL CAPS! THAT WOULD BE VERY DIFFICULT TO READ, DON'T YOU THINK?

Many of the technical words used in this book are in the Glossary at the end of the book; it is also very helpful to have a good technical dictionary of computer terms, as well as a good general dictionary of English.

Of course, you may not be able to remember all of these technical definitions the first time through. If you can't recall the exact meaning of one of these terms, just look up the word or phrase in the index, and it will direct you to the page where the definition is stated. You could also look in the Glossary, at the end of the book. Definitions of key technical terms are listed there in alphabetical order.

Before we continue, let's check in again with Susan. The following is from her first letter to me about the contents of this book:

Susan: I like the one-on-one feel of your text, like you are just talking to me. Now, you did make a few references to how simple some things were (which ?) I didn't catch on to, so it kinda made me feel I was not too bright for not seeing how apparently simple those things were. . . .

I think maybe it would have been helpful if you could have stated from the onset of this book just what direction you were taking, at least chapter by chapter. I would have liked to have seen a goal stated or at least a summary of objectives from the beginning. I often would have the feeling I was just suddenly thrown into something as I was reading along. Also (maybe you should call this *C++ for Dummies*, or is that taken already?)², you might even *define* what programming is! What a concept! Because it did occur to me that since I have never seen it done, or a language or anything, I really don't know what programming *is*! I just knew it was something that nerds do.

Susan's wish is my command, so I have provided a list of objectives at the beginning of each chapter after this one. I've also fulfilled her request for a definition of some programming terms, starting as follows:

An **algorithm** is a set of precisely defined steps to calculate an answer to a problem or set of problems, which is guaranteed to arrive at such an answer eventually. As this implies, a set of steps that might never end is *not* an algorithm.

Programming is the art and science of solving problems by the following procedure:³

1. Find or invent a general solution to a class of problems.
2. Express this solution as an algorithm or set of algorithms.
3. Translate the algorithm(s) into terms so simple that a stupid machine like a computer can follow them to calculate the specific answer for any specific problem in the class.

At this point, let's see what Susan had to say about the above definition and my response:

Susan: Very descriptive. How about this definition: "Programming is the process of being creative using the tools of science such as incremental problem solving to make a stupid computer do what you want it to"? That I understand!

Your definition is just fine. A definition has to be concise and descriptive and that you have done and covered all the bases. But you know what is lacking? An example of what it looks like. Maybe just a little statement that really looks bizarre to me and then say that by the end of the chapter you will actually know what this stuff really means! Sort of like a coming attraction type of thing.

Steve: I understand the idea of trying to draw the reader into the "game". However, I think that presenting a bunch of apparent gibberish with no warning could frighten readers as easily as it might intrigue them. I think it's better to delay showing examples until they have some background.

Now let's return to our list of definitions:

Hardware refers to the physical components of a computer, the ones you can touch. Examples include the keyboard, the monitor, the printer.

Software refers to the other, nonphysical components of a computer, the ones you cannot touch. If you can install it on your hard disk, it's software. Examples include a spreadsheet, a word processor, a database program.

Source code is a program in a form suitable for reading and writing by a human being.

An **executable program** (or just an *executable*, for short) is a program in a form suitable for running on a computer.

Object code is a program in a form suitable for incorporation into an executable program.

Compilation is the process of translating source code into object code. Almost all of the software on your computer was created by this process.

A **compiler** is a program that performs compilation as defined above.

How to Write a Program

Now you have a definition of programming. Unfortunately, however, this doesn't tell you how to write a program. The process of solving a problem by programming in C++ follows these steps:⁴

- **Problem:** After discussions between the user and the programmer, the programmer defines the problem precisely.
- **Algorithms:** The programmer finds or creates algorithms that will solve the problem.
- **C++:** The programmer implements these algorithms as source code in C++.
- **Executable:** The programmer runs the C++ compiler, which must already be present on the programmer's machine, to translate the source code into an executable program.
- **Hardware:** The user runs the resulting executable program on a computer.

These steps advance from the most abstract to the most concrete, which is perfectly appropriate for an experienced C++ programmer. However, if you're using this book to learn how to program in C++, obviously you're not an experienced C++ programmer, so before you can follow this path to solving a problem you're going to need a fairly thorough grounding in all of these steps. It's not really feasible to discuss each step exhaustively before going to the next one, so I've created a little "step indicator" that you'll see on each page of the text, with the currently active step shown in bold. For example, when we're discussing algorithms, the indicator will display the word **Algorithms** in bold.

The five steps of this indicator correspond to the five steps in problem solving just defined. I hope this device will make it easier for you to follow the sometimes tortuous path to programming knowledge. Let's see what Susan thinks of it:

Susan: With all the new concepts and all the new language and terms, it is so hard to know what one thing has to do with the other and where things are supposed to fit into the big picture. With the key, you can see how these things all fit as logical steps to an end. Now, I know it isn't going to be easy, but at least I know what my destination is before I board the plane. Anyway, you have to understand; for someone like me, this is an enormous amount of new material to be introduced to all at once. When you are bombarded with so many new terms and so many abstract concepts, it is a little hard to sort out what is what. Will you have guidelines for each of the steps? Since I know a little about this already, the more I look at the steps, I just know that what is coming is going to be a big deal. For example, take step 1: you have to give the ingredients for properly defining a problem. If something is left out, then everything that follows won't work.

Steve: I hope you won't find it that frustrating, because I explain all of the steps carefully as I do them. Of course, it's possible that I haven't been careful enough, but in that case you can let me know and I'll explain it further.

Unfortunately, it's not possible for me to provide a thorough guide to all of those steps, as that would be a series of books in itself. However, there's a wonderful small book called *How to Solve It* by G. Polya, that you should be able to get at your local library. It was written to help students solve geometry problems, but the techniques are applicable in areas other than geometry. I'm going to recommend that readers of my book read it if they have any trouble with general problem solving.

The steps for solving a problem via programming might sound reasonable in the abstract, but that doesn't mean that you can follow them easily without practice. Assuming that you already have a pretty good idea of what the problem is that you're trying to solve, the Algorithms step is likely to be the biggest stumbling block. Therefore, it might be very helpful to go into that step in a bit more detail.

Baby Steps

If we already understand the problem we're going to solve, the next step is to figure out a plan of attack, which we will then break down into small enough steps to be expressed in C++. This is called **stepwise refinement**, since we start out with a "coarse" solution and refine it until the steps are within the capability of the C++ language. For a complex problem, this may take several intermediate steps, but let's start out with a simple example. Say that we want to know how much older one person is than another. We might start with the following general outline:

1. Get ages from user.
2. Calculate difference of ages.
3. Print the result.

This can in turn be broken down further as follows:

1. Get ages from user.
 1. Ask user for first age.
 2. Ask user for second age.
2. Subtract second age from first age.
3. Print result.

This looks okay, except that if the first person is younger than the second one, then the result will be negative. That may be acceptable. If so, we're just about done, since these steps are simple enough for us to translate them into C++ fairly directly. Otherwise, we'll have to modify our program to do something different depending on which age is higher. For example,

1. Get ages from user.
 1. Ask user for first age.
 2. Ask user for second age.
2. Compute difference of ages.
 1. If first age is greater than second, subtract second age from first age.
 2. Otherwise, subtract first age from second age.
3. Print result.

You've probably noticed that this is a much more detailed description than would be needed to tell a human being what you want to do. That's because the computer is extremely stupid and literal: it does only what you tell it to do, not what you meant to tell it to do. Unfortunately, it's very easy to get one of the steps wrong, especially in a complex program. In that case, the computer will do something ridiculous, and you'll have to figure out what you did wrong. This "debugging", as it's called, is one of the hardest parts of programming. Actually, it shouldn't be too difficult to understand why that is the case. After all, you're looking for a mistake you've made yourself. If you knew exactly what you were doing, you wouldn't have made the mistake in the first place.

I hope that this brief discussion has made the process of programming a little less mysterious. In the final analysis, it's basically just logical thinking.⁵

On with the Show

Now that you have some idea how programming works, it's time to see exactly how the computer actually performs the steps in a program, which is the topic of Chapter [hardware.htm](#).

Footnotes

1. Of course, it's also possible that you already know how to program in another language and are using this book to learn how to do so in C++. If so, you'll have a head start; I hope that you'll learn enough to make it worth your while to wade through some material you already know.
2. As it happens, that title is indeed taken. However, I'm not sure it's been applied appropriately, since the book with that title assumes previous knowledge of C! What that says about C programmers is better left to the imagination.
3. This definition is possibly somewhat misleading since it implies that the development of a

program is straightforward and linear, with no revisions required. This is known as the "waterfall model" of programming, since water going over a waterfall follows a preordained course in one direction. However, real-life programming doesn't usually work this way; rather, most programs are written in an incremental process as assumptions are changed and errors are found and corrected.

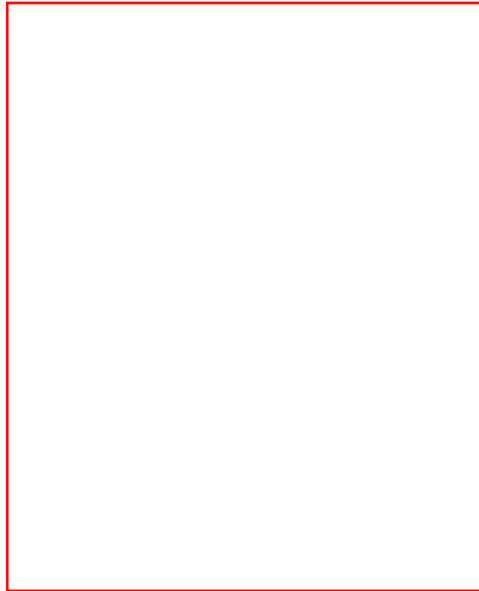
4. This description is actually a bit oversimplified, as we'll see in the discussion of *linking* in a later chapter.
5. Of course, the word *just* in this sentence is a bit misleading; taking logical thinking for granted is a sure recipe for trouble.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Hardware Fundamentals

Getting Started

Like any complex tool, the computer can be understood on several levels. For example, it's entirely possible to learn to drive an automobile without having the slightest idea of how it works. The analogy with computers is that it's relatively easy to learn how to use a word processor without having any notion of how such programs work. On the other hand, programming is much more closely analogous to designing an automobile than it is to driving one; therefore, we're going to have to go into some detail about the internal workings of a computer, not at the level of electronic components, but at the lowest level accessible to a programmer. This is a book on learning to program in C++, not on how a computer works.¹ Therefore, it might seem better to start there and eliminate this detour, and indeed many (perhaps most) books on C++ do exactly that. However, in working out in detail how I'm going to explain C++ to you, I've come to the conclusion that it would be virtually impossible to explain *why* certain features of the language exist and how they actually work, without your understanding *how* they relate to the underlying computer hardware.

I haven't come to this position by pure logical deduction, either. In fact, I've worked backward from the concepts that you will need to know to program in C++ to the specific underlying information that you will have to understand first. I'm thinking in particular of one specific concept that is supposed to be extremely difficult for a beginning programmer in C++ to grasp. With the approach we're taking, you shouldn't have much trouble understanding this concept by the time you get to it in Chapter [string.htm](#); it's noted as such in the discussion there. I'd be interested to know how you find my explanation there, given the background that you'll have by that point; don't hesitate to e-mail me about this topic (or any other, for that matter).² On the other hand, if you're an experienced programmer, a lot of this will be just review for you. Nonetheless, it can't hurt to go over the basics one more time before diving into the ideas and techniques that make C++ different from other languages.

Now let's begin with some definitions and objectives for this chapter.

Definitions

A **digit** is one of the characters used in any positional numbering system to represent all numbers starting at 0 and ending at one less than the base of the numbering system. In the decimal system, there are ten digits, 0 through 9, and in the hexadecimal system there are sixteen digits, 0 through 9 and a through f.

A **binary** number system is one that uses only two digits, 0 and 1.

A **hexadecimal** number system is one that uses 16 digits, 0-9 and a-f.

CPU is an abbreviation for Central Processing Unit. This is the "active" part of your computer, which executes all the *machine instructions* that make the computer do useful work.

A **machine instruction** is one of the fundamental operations that a *CPU* can perform. Some examples

of these operations are addition, subtraction, or other arithmetic operations; other possibilities include operations that control what instruction will be executed next. All C++ programs must be converted into machine instructions by a *compiler* before they can be executed by the *CPU*.

An **assembly language** program is the human-readable representation of a set of *machine instructions*; each assembly language statement corresponds to one machine instruction. By contrast, a C++ program consists of much higher-level operations which cannot be directly translated one-for-one into machine instructions.

Objectives of This Chapter

By the end of this chapter, you should

1. Understand the programmer's view of the most important pieces of hardware in your computer.
2. Understand the programmer's view of the most important pieces of software in your computer.
3. Be able to solve simple problems using both the binary and hexadecimal number systems.
4. Understand how whole numbers are stored in the computer.

Behind the Curtain

First we'll need to expand on the definition of *hardware*. As noted earlier, *hardware* means the physical components of a computer, the ones you can touch.³ Examples are the monitor, which displays your document while you're working on it, the keyboard, the printer, and all of the interesting electronic and electromechanical components inside the case of your computer.⁴ Right now, we're concerned with the programmer's view of the hardware. The hardware components of a computer with which you'll be primarily concerned are the disk, RAM (short for Random Access Memory), and last but certainly not least, the CPU (short for Central Processing Unit).⁵ We'll take up each of these topics in turn.

Disk

When you sit down at your computer in the morning, before you turn it on, where are the programs you're going to run? To make this more specific, suppose you're going to use a word processor to revise a letter you wrote yesterday before you turned the computer off. Where is the letter, and where is the word processing program?

You probably know the answer to this question; they are stored on a disk inside the case of your computer.⁶ Disks use magnetic recording media, much like the material used to record speech and music on cassette tapes, to store information in a way that will not be lost when the power is turned off. How exactly is this information (which may be either executable programs or data such as word processing documents) stored?

We don't have to go into excruciating detail on the storage mechanism, but it is important to understand some of its characteristics. A disk consists of one or more circular *platters*, which are

extremely flat and smooth pieces of metal or glass covered with a material that can be very rapidly and accurately magnetized in either of two directions, "north" and "south". To store large amounts of data, each platter is divided into many millions of small regions, each of which can be magnetized in either direction independent of the other regions. The magnetization is detected and modified by *recording heads*, similar in principle to those used in tape cassette decks. However, in contrast to the cassette heads, which make contact with the tape while they are recording or playing back music or speech, the disk heads "fly" a few millionths of an inch away from the platters, which rotate at very high velocity.⁷ The separately magnetizable regions used to store information are arranged in groups called *sectors*, which are in turn arranged in concentric circles called *tracks*. All tracks on one side of a given platter (a *recording surface*) can be accessed by a recording head dedicated to that recording surface; each sector is used to store some number of *bytes* of the data, generally a few hundred to a few thousand. "Byte" is a coined word meaning a group of 8 *binary digits*, or *bits* for short.⁸ You may wonder why the data aren't stored in the more familiar decimal system, which of course uses the digits from 0 through 9. This is not an arbitrary decision; on the contrary, there are a couple of very good reasons that data on a disk are stored using the binary system, in which each digit has only two possible states, 0 and 1. One of these reasons is that it's a lot easier to determine reliably whether a particular area on a disk is magnetized "north" or "south" than it is to determine 1 of 10 possible levels of magnetization. Another reason is that the binary system is also the natural system for data storage using electronic circuitry, which is used to store data in the rest of the computer.

While magnetic storage devices have been around in one form or another since the very early days of computing, the advances in technology just in the last 10 years have been staggering. To comprehend just how large these advances have been, we need to define the term used to describe storage capacities: the Megabyte. The standard engineering meaning of *Mega* is "multiply by 1 million", which would make a Megabyte equal to 1 million (1,000,000) bytes. As we have just seen, however, the natural number system in the computer field is binary. Therefore, "one Megabyte" is often used instead to specify the nearest "round" number in the binary system, which is 2^{20} (2 to the 20th power), or 1,048,576 bytes.⁹ This wasn't obvious to Susan, so I explained it some more, as you can see here:

Susan: Just how important is it to really understand that the Megabyte is 2^{20} (1,048,576) bytes? I know that a meg is not really a meg; that is, it's more than a million. But I don't understand 2^{20} , so is it enough to just take your word on this and not get bogged down as to why I didn't go any further than plane geometry in high school? You see, it makes me worry and upsets me that I don't understand how you "round" a binary number.

Steve: The ^ symbol is a common way of saying "to the power of", so 2^{20} would be 2 to the power of 20; that is, 20 2s multiplied together. This is a "round" number in binary just as $10 * 10 * 10$ (1000) is a "round" number in decimal.

1985, a Space Odyssey

With that detail out of the way, we can see just how far we've come in a short period of time. In 1985, I purchased a 20 Megabyte disk for \$900 (\$45 per Megabyte); its **access time**, which measures how long it takes to retrieve data, was approximately 100 milliseconds (milli = 1/1000, so a millisecond is

one thousandth of a second). In April 1997, a 6510 Megabyte disk cost as little as \$449, or approximately *7 cents* per Megabyte; in addition to delivering 650 times as much storage per dollar, this disk had an access time of 14 milliseconds, which is approximately 7 times as fast as the old disk. Of course, this significantly understates the amount of progress in technology in both economic and technical terms. For one thing, a 1997 dollar is worth considerably less than a 1985 dollar. In addition, the new drive is superior in every other measure as well: It is much smaller than the old one, consumes much less power, and has many times the projected reliability of the old drive.

This tremendous increase in performance and price has prevented the long-predicted demise of disk drives in favor of new technology. However, the inherent speed limitations of disks still require us to restrict their role to the storage and retrieval of data for which we can afford to wait a relatively long time.

You see, while 14 milliseconds isn't very long by human standards, it is a long time indeed to a modern computer. This will become more evident as we examine the next essential component of the computer, the *RAM*.

RAM

The working storage of the computer, where data and programs are stored while we're using them is called **RAM**, which is an acronym for Random Access Memory.¹⁰ For example, your word processor is stored in RAM while you're using it. The document you're working on is likely to be there as well unless it's too large to fit all at once, in which case parts of it will be retrieved from the disk as needed. Since we have already seen that both the word processor and the document are stored on the disk in the first place, why not leave them there and use them in place, rather than copying them into RAM?

The answer, in a word, is *speed*. RAM is physically composed of millions of microscopic switches on a small piece of silicon known as a *chip*: a 4 megabit RAM chip has approximately 4 million of them.¹¹ Each of these switches can be either on or off; we consider a switch that is "on" to be storing a 1, and a switch that is "off" to be storing a 0. Just as in storing information on a disk, where it was easier to magnetize a region in either of two directions, it's a lot easier to make a switch that can be turned on or off reliably and quickly than one that can be set to any value from 0 to 9 reliably and quickly. This is particularly important when you're manufacturing millions of them on a silicon chip the size of your fingernail.

A main difference between disk and RAM is what steps are needed to access different areas of storage. In the case of the disk, the head has to be moved to the right track (an operation known as a *seek*), and then we have to wait for the platter to spin so that the region we want to access is under the head (called *rotational delay*). On the other hand, with RAM, the entire process is electronic; we can read or write any byte immediately as long as we know which byte we want. To specify a given byte, we have to supply a unique number called its **memory address** or just **address** for short.

Return to Sender, Address Unknown

What is an address good for? Let's see how my discussion with Susan on this topic started:

Susan: About memory addresses: are you saying that each little itty bitty tiny byte of RAM is a separate address? Well, this is a little hard to imagine.

Steve: Actually, each byte of RAM *has* a separate address, which doesn't change, and a value, which does.

In case the notion of an address of a byte of memory on a piece of silicon is too abstract, it might help to think of an address as a set of directions as to how to find the byte being addressed, much like directions to someone's house. For example, "Go three streets down, then turn left. It's the second house on the right". With such directions, the house number wouldn't need to be written on the house. Similarly, the memory storage areas in RAM are addressed by position; you can think of the address as telling the hardware which street and house you want, by giving directions similar in concept to the preceding example. Therefore, it's not necessary to encode the addresses into the RAM explicitly.

Susan wanted a better picture of this somewhat abstract idea:

Susan: Where are the bytes on the RAM, and what do they look like?

Steve: Each byte corresponds to a microscopic region of the RAM chip. As to what they look like, have you ever seen a printed circuit board such as the ones inside your computer? Imagine the lines on that circuit board reduced thousands of times in size to microscopic dimensions, and you'll have an idea of what a RAM chip looks like inside.

Since it has no moving parts, storing and retrieving data in RAM is much faster than waiting for the mechanical motion of a disk platter turning.¹² As we've just seen, disk access times are measured in milliseconds, or thousandths of a second. However, RAM access times are measured in *nanoseconds* (abbreviated *ns*); *nano* means one billionth. In early 1997, a typical speed for RAM was 70 ns, which means that it is possible to read a given data item from RAM about 200,000 times as quickly as from a disk. In that case, why not use disks only for permanent storage, and read everything into RAM in the morning when we turn on the machine?

The reason is cost. In early 1997, the cost of 16 Megabytes of RAM was approximately \$80. For that same amount of money, you could have bought over 1100 Megabytes of disk space!¹³ Therefore, we must reserve RAM for tasks where speed is all-important, such as running your word processing program and holding a letter while you're working on it. Also, since RAM is an electronic storage medium (rather than a magnetic one), it does not maintain its contents when the power is turned off. This means that if you had a power failure while working with data only in RAM, you would lose everything you had been doing.¹⁴ This is not merely a theoretical problem, by the way; if you don't remember to save what you're doing in your word processor once in a while, you might lose a whole day's work from a power outage of a few seconds.¹⁵ Before we get to how a program actually works, we need to develop a better picture of how RAM is used. As I've mentioned before, you can think of RAM as consisting of a large number of bytes, each of which has a unique identifier called an *address*. This address can be used to specify which byte we mean, so the program might specify that it wants to read the value in byte 148257, or change the value in byte 66666. Susan wanted to make sure she had the correct understanding of this topic:

Susan: Are the values changed in RAM depending on what program is loaded in it?

Steve: Yes, and they also change while the program is executing. RAM is used to store both the program itself and the values it manipulates.

This is all very well, but it doesn't answer the question of how the program actually uses or changes values in RAM, or performs arithmetic and other operations; that's the job of the CPU, which we will take up next.

The CPU

The **CPU** (Central Processing Unit) is the "active" component in the computer. Like RAM, it is physically composed of millions of microscopic transistors on a chip; however, the organization of these transistors in a CPU is much more complex than on a RAM chip, as the latter's functions are limited to the storage and retrieval of data. The CPU, on the other hand, is capable of performing dozens or hundreds of different fundamental operations called *machine instructions*, or just *instructions* for short. While each instruction performs a very simple function, the tremendous power of the computer lies in the fact that the CPU can perform (or *execute*) tens or hundreds of millions of these instructions per second.¹⁶ These instructions fall into a number of categories: instructions that perform arithmetic operations such as adding, subtracting, multiplying, and dividing; instructions that move information from one place to another in RAM; instructions that compare two quantities to help make a determination as to which instructions need to be executed next and instructions that implement that decision; and other, more specialized types of instructions.

Of course, adding two numbers together (for example) requires that the numbers be available for use. Possibly the most straightforward way of making them available is to store them in and retrieve them from RAM whenever they are needed, and indeed this is done sometimes. However, as fast as RAM is compared to disk drives (not to mention human beings), it's still pretty slow compared to modern CPUs. For example, the computer on which I started to write this book had a 66 Megahertz (abbreviated MHz) CPU, which can execute up to 66 million instructions per second (abbreviated MIPS),¹⁷ or one instruction approximately every 16 ns.¹⁸ To see why RAM is a bottleneck, let's calculate how long it would take to execute an instruction if all the data had to come from and go back to RAM. A typical instruction would have to read some data from RAM, and write its result back there; first, though, the instruction itself has to be loaded (or *fetched*) into the CPU before it can be executed. Let's suppose we have an instruction in RAM, reading one data item also in RAM, and writing the result back to RAM. Then the minimum timing to do such an instruction could be calculated as in Figure [lowspeed](#).

RAM vs. CPU speeds (Figure lowspeed)

Time	Function
------	----------

```
70 ns  Read instruction from RAM
```

```
70 ns  Read data from RAM
```

```
16 ns  Execute instruction
```

```
70 ns  Write result back to RAM
```

```
-----
```

```
226 ns  Total instruction execution time
```

To compute the effective MIPS of a CPU, we divide 1 second by the time it takes to execute one instruction. Given the assumptions in this example, the CPU could execute only about 4.5 million instructions per second, which is a far cry from the peak performance of 66 MIPS claimed by the manufacturer.¹⁹ If the manufacturer's claims have any relation to reality, there must be a better way.

In fact, there is. As a result of a lot of research and development, both in academia and in the semiconductor industry, it is possible to approach the rated performance of fast CPUs.²⁰ Some of these techniques have been around as long as we've had computers; others have fairly recently "trickled down" from supercomputers to microcomputer CPUs. One of the most important of these techniques is the use of a number of different kinds of storage devices having different performance characteristics; the arrangement of these devices is called the **memory hierarchy**. Figure [hierarchyfig](#) illustrates the memory hierarchy on my home machine when I started writing this book in late 1994.

Susan and I had a short discussion of the layout of this figure:

Susan: OK, just one question on Figure [hierarchyfig](#). If you are going to include the disk in this hierarchy, I don't know why you have placed it over to the side of RAM and not above it, since it is slower and you appear to be presenting this figure in ascending order of speed from the top of the figure downward. Did you do this because it is external rather than internal memory and it doesn't "deserve" to be in the same lineage as the others?

Steve: Yes; it's not the same as "real" memory, so I wanted to distinguish it.

Before we get to the diagram, I should explain that a **cache** is a small amount of fast memory where frequently used data are stored temporarily. According to this definition, RAM functions more or less as a cache for the disk; after all, we have to copy data from a slow disk into fast RAM before we can use it for anything. However, while this is a valid analogy, I should point out that the situations aren't quite parallel. Our programs usually read data from disk into RAM explicitly; that is, we're aware of

		bx	ebx
	+-----+		
	+-----+		
		cx	ecx
	+-----+		
	+-----+		
		dx	edx
	+-----+		
	+-----+		
		si	esi
	+-----+		
	+-----+		
		di	edi
	+-----+		
	+-----+		
		bp	ebp
	+-----+		
		Registers	
		Size: 28 bytes	
		Accesses: 133 M/sec.	
	+-----+		

So just as with the disk vs. RAM trade-off before, the reason that we use the external cache is to improve performance. While RAM can be accessed about 14 million times per second, the external cache is made from a faster type of memory chips, which can be accessed about 30 million times per second. While not as extreme as the speed differential between disk and RAM, this difference is still significant. However, we can't afford to use external cache exclusively instead of RAM, because it would be too expensive to do so. In 1997, the cost of 1 MB of external cache was in the neighborhood of \$180. For that same amount of money, you could have bought over 16 Megabytes of RAM. Therefore, we must reserve external cache for tasks where speed is all-important, such as supplying frequently used data or programs to the CPU.²² The same analysis applies to the trade-off between the external cache and the *internal cache*. The internal cache's characteristics are similar to those of the the external cache, but to a greater degree; it's even smaller and faster, allowing access at the rated speed of the CPU. Both characteristics have to do with its privileged position on the same chip as the CPU; this reduces the delays in communication between the internal cache and the CPU but means that chip area devoted to the cache has to compete with area for the CPU, as long as the total chip size is held constant.

Unfortunately, we can't just increase the size of the chip to accommodate more internal cache because of the expense of doing so. Larger chips are more difficult to make, which reduces their *yield*, or the percentage of good chips. In addition, fewer of them fit on one *wafer*, which is the unit of manufacturing. Both of these attributes make larger chips more expensive to make.

Caching In

To oversimplify a bit, here's how caching reduces the effects of slow RAM. Whenever a data item is requested by the CPU, there are three possibilities:

1. It is already in the internal cache. In this case, the value is sent to the CPU without referring to RAM at all.
2. It is in the external cache; in this case it will be "promoted" to the internal cache, and sent to the CPU at the same time.
3. It is not in either the internal or external cache. In this case, it has to be entered into a location in the internal cache. If there is nothing in that cache location, the new item is simply added to the cache. However, if there is a data item already in that cache location, then the old item is displaced to the external cache, and the new item is written in its place.²³ If the external cache location is empty, that ends the activity; if it is not empty, then the item previously in that location is written out to RAM and its slot is used for the one displaced from the internal cache.²⁴

Please Register Here

Another way to improve performance that has been employed for many years is to create a small number of private storage areas, called **registers**, that are on the same chip as the CPU itself.²⁵ Programs use these registers to hold data items that are actively in use; data in registers can be accessed within the time allocated to instruction execution (16 ns in our example), rather than the much longer times needed to access data in RAM. This means that the time needed to access data in registers is predictable, unlike data that may have been displaced from the internal cache by more recent arrivals and thus must be reloaded from the external cache or even from RAM. Most CPUs have some **dedicated registers**, which aren't available to application programmers (that's us), but are reserved for the operating system (e.g., DOS, Unix, OS/2) or have special functions dictated by the hardware design; however, we will be concerned primarily with the **general registers**, which are available for our use.²⁶ The general registers are used to hold working copies of data items called **variables**, which otherwise reside in RAM during the execution of the program. These variables represent specific items of data that we wish to keep track of in our programs, such as weights and numbers of items.

The notion of using registers to hold temporary copies of variables wasn't crystal clear to Susan. Here's our discussion on that topic:

Susan: Here we go, getting lost. When you said "The general registers are used to hold working copies of data items called variables, which reside in RAM", are you saying RAM stores info when not in use?

Steve: During execution of a program, when data aren't in the general registers, they are generally stored in RAM.

Susan: See, this is confusing to me, because I didn't think RAM stores anything when turned off.

Steve: You're correct; RAM doesn't retain information when the machine is turned off. However, it is used to keep the "real" copies of data that we want to process but won't fit in the registers.²⁷

You can put something in a variable, and it will stay there until you store something else there; you can also look at it to find out what's in it. As you might expect, several types of variables are used to hold different kinds of data; the first ones we will look at are variables representing whole numbers (the so-called **integer variables**), which are a subset of the category called **numeric variables**. As this suggests, there are also variables that represent numbers that can have fractional parts. We'll look at these so-called floating-point variables briefly in a later chapter.

Different types of variables require different amounts of RAM to store them, depending on the amount of data they contain; a very common type of numeric variable, known as a `short`, requires 16 bits (that is, 2 bytes) of RAM to hold any of 65536 different values, from -32768 to 32767, including 0. As we will see shortly, these odd-looking numbers are the result of using the binary system. By no coincidence at all, the early Intel CPUs such as the 8086 had general registers that contained 16 bits each; these registers were named `ax`, `bx`, `cx`, `dx`, `si`, `di`, and `bp`. Why does it matter how many bits each register holds? Because the number (and size) of instructions it takes to process a variable is much less if the variable fits in a register; therefore, most programming languages, C++ included, relate the size of a variable to the size of the registers available to hold it. A `short` is exactly the right size to fit into a 16-bit register and therefore can be processed efficiently by the early Intel machines, whereas longer variables had to be handled in pieces, causing a great decline in efficiency of the program. Progress marches on: more recent Intel CPUs, starting with the 80386, have 32-bit general registers; these registers are called `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`. You may have noticed that these names are simply the names of the old 16-bit registers with an `e` tacked onto the front. The reason for the name change is that when Intel increased the size of the registers to 32 bits with the advent of the 80386, it didn't want to change the behavior of previously existing programs that (of course) used the old names for the 16-bit registers. So the old names, as illustrated in Figure [hierarchyfig](#), now refer to the bottom halves of the "real" (that is, 32-bit) registers, as illustrated in Figure [hierarchyfig](#); instructions using these old names behave exactly as though they were accessing the 16-bit registers on earlier machines. To refer to the 32-bit registers, you use the new names `eax`, `ebx`, and so on, for "extended" `ax`, "extended" `bx`, and so forth. What does it mean to say that instructions using the 16-bit register names "behave exactly as though they were accessing the 16-bit registers on earlier machines"? Before I can explain this, you'll have to understand the binary number system, on which all modern computers are based. To make this number system more intelligible, I have written the following little fable.

Odometer Trouble

Once upon a time, the Acme company had a factory that made golf carts. One day, Bob, the president of Acme, decided to add an odometer to the carts, so that the purchaser of the cart could estimate when to recharge the battery. To save money, Bob decided to buy the little numbered wheels for the odometers and have his employees put the odometers together. The minimum order was a thousand odometer wheels, which was more than he needed for his initial run of 50 odometers. When he got the wheels, however, he noticed that they were defective: Instead of the numbers 0-9, each wheel had

only two numbers, 0 and 1. Of course, he was quite irritated by this error, and attempted to contact the company from which he had purchased the wheels, but it had closed down for a month for summer vacation. What was he to do until it reopened?

While he was fretting about this problem, the employee who had been assigned to the task of putting the odometers together from the wheels came up with a possible solution. This employee, Jim, came into Bob's office and said, "Bob, I have an idea. Since we have lots of orders for these odometer-equipped carts, maybe we can make an odometer with these funny wheels and tell the customers how to read the numbers on the odometer."

Bob was taken aback by this idea. "What do you mean, Jim? How can anyone read those screwy odometers?"

Jim had given this some thought. "Let's take a look at what one of these odometers, say with five wheels, can display. Obviously, it would start out reading 00000, just like a normal odometer. Then when one mile has elapsed, the rightmost wheel turns to 1, so the whole display is 00001; again, this is no different from a normal odometer."

"Now we come to the tricky part. The rightmost wheel goes back to 0, not having any more numbers to display, and pushes the `tens' wheel to 1; the whole number now reads 00010. Obviously, one more mile makes it 00011, which gives us the situation shown in the following diagram:

The first few numbers (Figure firstfew)

Normal odometer	Funny odometer
00000	00000
00001	00001
00002	00010
00003	00011

Jim continued, "What's next? This time, the rightmost wheel turns over again to 0, triggering the second wheel to its next position. However, this time, the second wheel is already at its highest value, 1; therefore, it also turns over to 0 and increments the third wheel. It's not hard to follow this for a few more miles, as illustrated in Figure [nextfew](#).

The next few numbers (Figure nextfew)

Normal odometer	Funny odometer
-----------------	----------------

00004	00100
00005	00101
00006	00110
00007	00111

Bob said, "I get it. It's almost as though we were counting normally, except that you skip all the numbers that have anything but 0s or 1s in them."

"That's right, Bob. So I suppose we could make up a list of the `real' numbers and give it to the customers to use until we can replace these odometers with normal ones. Perhaps they'll be willing to work with us on this problem."

"Okay, Jim, if you think they'll buy it. Let's get a few of the customers we know the best and ask them if they'll try it; we won't charge them for the odometers until we have the real ones, but maybe they'll stick with us until then. Perhaps any odometer would be better than no odometer at all."

Jim went to work, making some odometers out of the defective wheels; however, he soon figured out that he had to use more than five wheels, because that allowed only numbers from 0 to 31. How did he know this?

Each wheel has two numbers, 0 and 1. So with one wheel, we have a total of two combinations. Two wheels can have either a 0 or a 1 for the first number, and the same for the second number, for a total of four combinations. With three wheels, the same analysis holds: 2 numbers for the first wheel * 2 for the second wheel * 2 for the third wheel = 8 possibilities in all; actually, they are the same 8 possibilities we saw in Figures [firstfew](#) and [nextfew](#).

A pattern is beginning to develop: for each added wheel, we get twice as many possible combinations. To see how this continues, take a look at Figure [howmany](#), which shows the count of combinations vs. number of wheels for all wheel counts up to 16 (i.e., 16-bit quantities).

Jim decided that 14 wheels would do the job, since the lifespan of the golf cart probably wouldn't exceed 16,383 miles, and so he made up the odometers. The selected customers turned out to be agreeable and soon found that having even a weird odometer was better than none, especially since they didn't have to pay for it. However, one customer did have a complaint: The numbers on the wheels didn't seem to make sense when translated with the chart supplied by Acme. The customer estimated that he had driven the cart about 9 miles, but the odometer displayed the following number,

11111111110111

which, according to his translation chart, was 16375 miles. What could have gone wrong?

How many combinations? (Figure howmany)

Number of wheels	Number of combinations
------------------	------------------------

[28](#) 1 2

2 4

3 8

4 16

5 32

6 64

7 128

8 256

9 512

10 1024

11 2048

12 4096

13 8192

14 16384

15 32768

16 65536

Jim decided to have the cart brought in for a checkup, and what he discovered was that the odometer cable had been hooked up backwards. That is, instead of turning the wheels forward, they were going backwards. That was part of the solution, but why was the value 16375? Just like a car odometer, in which 99999 (or 999999, if you have a 6-wheel odometer) is followed by 0, going backwards from 0 reverses that progression. Similarly, the number 11111111111111 on the funny odometers would be followed by 00000000000000, since the "carry" off the leftmost digit is lost. Therefore, if you start out at 0 and go backward 1 mile, you'll get

11111111111111

The next mile will turn the last digit back to 0, producing

11111111111110

What happens next? The last wheel turns back to 1, and triggers the second wheel to switch as well:

11111111111101

The next few "backward" numbers look like this:

11111111111100

11111111111011

11111111111010

11111111111001

11111111111000

11111111110111

and so on. If you look at the right-hand end of these numbers, you'll see that the progression is just the opposite of the "forward" numbers.

As for the customer's actual mileage, the last one of these is the number the customer saw on his backward odometer. Apparently, he was right about the distance driven, since this is the ninth "backward" number. So Jim fixed the backward odometer cable and reset the value to the correct number, 0000000001001, or 9 miles.

Eventually, Acme got the right odometer wheels with 0-9 on them, replaced the peculiar ones, and everyone lived happily ever after.

THE END

Back to the Future

Of course, the wheels that made up the funny odometers contain only two digits, 0 and 1, so the odometers use the binary system for counting. Now it should be obvious why we will see numbers like 65536 and 32768 in our discussions of the number of possible different values that a variable can hold: variables are stored in RAM as collections of bytes, each of which contains 8 bits. As the list of combinations indicates, 8 bits (1 byte) provide 256 different combinations, while 16 bits (2 bytes) can

represent 65536 different possible values.

But what about the "backward" numbers with a lot of 1s on the left? As the fable suggests, they correspond to "negative" numbers. That is, if moving 2 miles forward from 0 registers as 00000000000010, and moving 2 miles backward from 0 registers as 11111111111110, then the latter number is in some sense equivalent to -2 miles. This in fact is the way that negative integers are stored in the computer; integer variables that can store either positive or negative values are called *signed variables*. If we don't specify whether we want to be able to store either positive or negative values in a given variable, the C++ language assumes that we want that ability, and provides it for us by default.

However, adding the ability to represent negative numbers has a drawback: namely, that you can't represent as many positive numbers. This should be fairly obvious, since if we interpret some of the possible patterns as negative, they can't also be used for positive values. Sometimes, of course, we don't have to worry about negative numbers, such as counting how many employees our company has; in such cases, we can specify that we want to use *unsigned variables*, which will always be interpreted as positive (or 0) values. An example is an `unsigned short` variable, which uses 16 bits (that is, 2 bytes) to hold any number from 0 to 65535, which totals 65536 different values. This capacity can be calculated as follows: since each byte is 8 bits, 2 bytes contain a total of 16 bits, and 2^{16} is 65536.

It's important to understand that the difference between a `short` (that is, a `signed short`) and an `unsigned short` is exactly which 65536 values each can hold. An **unsigned short** can hold any whole number from 0 to 65535, whereas a **short** can hold any value from -32768 to +32767.

I hope this is clear to you, but in case it isn't, let's see how Susan and I worked over this point:

Susan: I really don't think I understand what a `short` is besides being 2 bytes of RAM, and I don't really know what `signed` and `unsigned` mean.

Steve: A `short` is indeed 2 bytes (that is, 16 bits) of RAM. This means that it can hold any of 2^{16} (65536) different values. This is a very nice range of values for holding the number of pounds that a pumpkin weighs (for example). You'll see some more uses for this type of variable later.

The difference between a (`signed`) `short` and an `unsigned short` is exactly which 65536 values each can hold. An `unsigned short` can hold any whole number from 0 to 65535, whereas a (`signed`) `short` can hold any value from -32768 to +32767. The difference between these is *solely* in the interpretation that we (and the compiler) give to the values. In other words, it's not possible to tell whether a given 2 bytes of RAM represent a `short` or an `unsigned short` by looking at the contents of those bytes; you have to know how the variable was defined in the program.

Susan: Ok, let's start over. A `short` is 2 bytes of RAM. A `short` is a variable. A `short` is a numeric variable. It can be `signed` (why is that a default?), meaning its value can be -32768 to +32767, or `unsigned`, meaning its value can be 0-65535. How's that?

Steve: That's fine. Since you've asked, the reason `signed` is the default is because that's the way it was in C, and changing it in C++ would "break" C programs that depended on this default. Bjarne Stroustrup, the inventor of C++, has a rule that C++ must be as close to C as possible but no closer. In this case, there's no real reason to change the default, so it wasn't changed.

Susan: Oh, why is it that every time you say something is fairly obvious, my mind just shuts down? When you say "if we interpret some of the possible patterns as negative, they can't also be used for positive values." Huh? Then if that is the case would not the reverse also be true? I can see how this explains the values of the signed and unsigned short, but really I don't think I have grasped this concept.

Steve: What I was trying to explain is that you have to choose one of the following two possibilities:[29](#)

1. (signed) short range: -32768 to +32767
2. unsigned short range: 0 to 65535

In other words, you have to decide whether you want a given variable to represent:

1. Any of 32768 negative numbers, 0, or 32767 positive numbers, or
2. Any of 65536 nonnegative numbers from 0 to 65535

If you want a variable with a range like that in selection 1, use a `signed short`; if you prefer the range in selection 2, use an `unsigned short`. For example, for the number of lines in a text file, you could use an `unsigned short`, since the maximum number of lines could be limited to less than 65,000 lines and couldn't ever be negative. On the other hand, to represent the number of copies of a book that have been sold in the last month, including the possibility of returns exceeding sales, a `signed short` would be better, since the value could be either positive or negative.

Susan: In other words, if you are going to be using variables that might have a negative value then use a `signed short`, and if you want strictly "positive" numbers then use an `unsigned short`. Right?

Steve: Exactly!

Susan: Well, then, how do you write a `short` to indicate that it is `signed` or `unsigned`?

Steve: When you define it, you have to specify that it is `unsigned` if you want it to be `unsigned`; the default is `signed`. In other words, if we define a variable `x` as `short x;`, it will be `signed`, whereas if we want a variable called `x` that is an `unsigned short`, we have to say `unsigned short x;`.

Susan: So does it make any difference if your variable is going to overlap the `signed` and `unsigned short` ranges? For example, if you are using numbers from 10,000 to 30,000, would it matter which `short` you used? It falls under the definition of both.

Steve: You can use whichever you wish in that case.

Over-Hexed

You may have noticed that it's tedious and error prone to represent numbers in binary; a long string of 0s and 1s is hard to remember or to copy. For this reason, the pure binary system is hardly ever used to specify numbers in computing. However, we have already seen that binary is much more "natural" for computers than the more familiar decimal system. Is there a number system that we humans can use a little more easily than binary, while retaining the advantages of binary for describing internal events in the computer?

As it happens, there is. It's called **hexadecimal**, which means "base 16". As a rule, the term *hexadecimal* is abbreviated to *hex*. Since there are 16 possible combinations of 4 bits ($2*2*2*2$), hexadecimal notation allows 4 bits of a binary number to be represented by one hex digit.

Unfortunately, however, there are only 10 "normal" digits, 0-9.³⁰ To represent a number in any base, you need as many different digit values as the base, so that any number less than the base can be represented by one digit. For example, in base 2, you need only two digits, 0 and 1. In base 8 (*octal*), you need eight digits, 0-7.³¹ So far, so good. But what about base 16? To use this base, we need 16 digits. Since only 10 numeric digits are available, hex notation needs a source for the other six digits. Because letters of the alphabet are available and familiar, the first six letters, a-f, were adopted for this service.³² Although the notion of a base-16 numbering system doesn't seem strange to people who are familiar with it, it can really throw someone who learned normal decimal arithmetic solely by rote, without understanding the concepts on which it is based. This topic of hexadecimal notation occupied Susan and me for quite awhile; here's some of the discussion we had about it:

Susan: I don't get this at all! What is the deal with the letters in the hex system? I guess it would be okay if 16 wasn't represented by 10!

Steve: Well, there are only 10 "normal" digits, 0-9. To represent a number in any base, you need as many "digits" as the base, so that any number less than the base can be represented by one "digit". This is no problem with a base less than ten, such as octal, but what about base 16? To use this base we need 16 digits, 0-9 and a-f. One way to remember this is to imagine that the "hex" in "hexadecimal" stands for the six letters a through f and the "decimal" stands for the 10 digits 0-9.

Susan: OK, so a hex digit represents 16 bits? So then is hex equal to 2 bytes? According to the preceding a hex digit is 4 bits.

Steve: Yes, a hex digit represents 4 bits. Let's try a new approach. First, let me define a new term, a *hexit*. That's short for "hex digit", just like "bit" is short for "binary digit".

1. How many one-digit decimal numbers exist?
2. How many two-digit decimal numbers exist?
3. How many three-digit decimal numbers exist?
4. How many four-digit decimal numbers exist?
5. How many one-bit binary numbers exist?
6. How many two-bit binary numbers exist?
7. How many three-bit binary numbers exist?
8. How many four-bit binary numbers exist?
9. How many one-hexit hexadecimal numbers exist?
10. How many two-hexit hexadecimal numbers exist?
11. How many three-hexit hexadecimal numbers exist?
12. How many four-hexit hexadecimal numbers exist?

The answers are:

1. 10
2. 100
3. 1000
4. 10000
5. 2
6. 4
7. 8
8. 16
9. 16
10. 256
11. 4096
12. 65536

What do all these answers have in common? Let's look at the answers a little differently, in powers of 10, 2, and 16, respectively:

1. $10 = 10^1$
2. $100 = 10^2$
3. $1000 = 10^3$
4. $10000 = 10^4$
5. $2 = 2^1$
6. $4 = 2^2$
7. $8 = 2^3$
8. $16 = 2^4$
9. $16 = 16^1$
10. $256 = 16^2$
11. $4096 = 16^3$
12. $65536 = 16^4$

That is, a number that has one digit can represent "base" different values, where "base" is two, ten, or sixteen (in our examples). Every time we increase the size of the number

by one more digit, we can represent "base" times as many possible different values, or in other words, we multiply the range of values that the number can represent by the base. Thus, a two-digit number can represent any of "base*base" values, a three-digit number can represent any of "base*base*base" values, and so on. That's the way positional number systems such as decimal, binary, and hex work. If you need a bigger number, you just add more digits.

Okay, so what does this have to do with hex? If you look at the above table, you'll see that 2^4 (16) is equal to 16^1 . That means that 4 bits are exactly equivalent to one hexit in their ability to represent different numbers: exactly 16 possible numbers can be represented by four bits, and exactly 16 possible numbers can be represented by one hexit.

This means that you can write one hexit wherever you would otherwise have to use four bits, as illustrated in Figure [binhex](#).

Binary to hex conversion table (Figure binhex)

4-bit value 1-hexit value

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b

1100	c
1101	d
1110	e
1111	f

So an 8-bit number, such as:

0101 1011

can be translated directly into a hex value, like this:

5 b

For this reason, binary is almost never used. Instead, we use hex as a shortcut to eliminate the necessity of reading, writing, and remembering long strings of bits.

Susan: A hex digit or hexit is like a four-wheel odometer in binary. Since each wheel is capable of only one of two values, being either (1) or (0), then the total number of possible values is 16. Thus your $2*2*2*2 = 16$. I think I've got this down.

Steve: You certainly do!

Susan: If it has 4 bits and you have 2 of them then won't there be eight "wheels" and so forth? So 2 hex would hold XXXXXXXX places and 3 hex would hold XXXXXXXXXXXX places.

Steve: Correct. A one-hexit number is analogous to a one-digit decimal number. A one-hexit number contains 4 bits and therefore can represent any of 16 values. A two-hexit number contains 8 bits and therefore can represent any of 256 values.

Now that we've seen how each hex digit corresponds exactly to a group of four binary digits, here's an exercise you can use to improve your understanding of this topic: Invent a random string of four binary digits and see where it is in figure [binhex](#). I guarantee it'll be there somewhere! Then look at the "hex" column and see what "digit" it corresponds to. There's nothing really mysterious about hex; since we have run out of digits after 9, we have to use letters to represent the numbers `ten', `eleven', `twelve', `thirteen', `fourteen', and `fifteen'.

Now, here's a table showing the correspondence between some decimal, hex, and binary numbers, with the values of each digit position in each number base indicated, and the calculation of the total of

all of the bit values in the binary representation, as shown in Figure [diffrep](#).

Different representations of the same numbers (Figure diffrep)

Decimal	Hexadecimal	Binary	Sum of binary
Place Values	Place Values	Place Values	digit values
10 1	16 1	16 8 4 2 1	
0	0 0	0 0 0 0 0	= 0 + 0 + 0 + 0 + 0
1	0 1	0 0 0 0 1	= 0 + 0 + 0 + 0 + 1
2	0 2	0 0 0 1 0	= 0 + 0 + 0 + 2 + 0
3	0 3	0 0 0 1 1	= 0 + 0 + 0 + 2 + 1
4	0 4	0 0 1 0 0	= 0 + 0 + 4 + 0 + 0
5	0 5	0 0 1 0 1	= 0 + 0 + 4 + 0 + 1
6	0 6	0 0 1 1 0	= 0 + 0 + 4 + 2 + 0
7	0 7	0 0 1 1 1	= 0 + 0 + 4 + 2 + 1
8	0 8	0 1 0 0 0	= 0 + 8 + 0 + 0 + 0
9	0 9	0 1 0 0 1	= 0 + 8 + 0 + 0 + 1
1 0	0 a	0 1 0 1 0	= 0 + 8 + 0 + 2 + 0
1 1	0 b	0 1 0 1 1	= 0 + 8 + 0 + 2 + 1
1 2	0 c	0 1 1 0 0	= 0 + 8 + 4 + 0 + 0
1 3	0 d	0 1 1 0 1	= 0 + 8 + 4 + 0 + 1
1 4	0 e	0 1 1 1 0	= 0 + 8 + 4 + 2 + 0

1 5	0 f	0 1 1 1 1	=	0 + 8 + 4 + 2 + 1
1 6	1 0	1 0 0 0 0	=	16 + 0 + 0 + 0 + 0
1 7	1 1	1 0 0 0 1	=	16 + 0 + 0 + 0 + 1
1 8	1 2	1 0 0 1 0	=	16 + 0 + 0 + 2 + 0
1 9	1 3	1 0 0 1 1	=	16 + 0 + 0 + 2 + 1

Another reason to use hex rather than decimal is that byte values expressed as hex digits can be combined directly to produce larger values, which is not true with decimal digits. In case this isn't obvious, let's go over it in more detail. Since each hex digit (0-f) represents exactly 4 bits, two of them (00-ff) represent 8 bits, or one byte. Similarly, 4 hex digits (0000-ffff) represent 16 bits, or a short value; the first two digits represent the first byte of the 2-byte value, and the last two digits, the second byte. This can be extended to any number of bytes. On the other hand, representing 4 bits requires two decimal digits, as the values range from 00-15, whereas it takes three digits (000-255) to represent one byte. A 2-byte value requires five decimal digits, since the value can be from 00000 to 65535. As you can see, there's no simple relationship between the decimal digits representing each byte and the decimal representation of a 2-byte value.

Susan had some more thoughts on the hexadecimal number system. Let's listen in:

Susan: I think you need to spend a little more time reviewing the hex system, like an entire chapter. <G> Well, I am getting the impression that we are going to be working with hex, so I am trying to concentrate my understanding on that instead of binary. I think this all moves a little too fast for me. I don't know what your other reviewers are saying but I just feel like I get a definition of a abstract concept, and the next thing I know I am supposed to be doing something with it, like make it work. Ha! I personally need to digest new concepts, I really need to think them over a bit, to take them in and absorb them. I just can't start working with it right away.

As usual, I've complied with her request; the results are immediately ahead.

Exercises

Here are some exercises that you can use to check your understanding of the binary and hexadecimal number systems.³³ I've limited the examples to addition and subtraction, as that is all that you're ever likely to have to do in these number systems. These operations are exactly like their equivalents in the decimal system, except that as we have already seen, the hexadecimal system has six extra digits after 9: a, b, c, d, e, and f. We have to take these into account in our calculations: for example, adding 9 and 5, rather than producing 14, produces e.

1. Using the hexadecimal system, answer these problems:

a. $1a + 2e = ?$

b. $12 + 18 = ?$

c. $50 - 12 = ?$

2. In the binary system, answer these problems:

a. $101 + 110 = ?$

b. $111 + 1001 = ?$

c. $1010 - 11 = ?$

Consider the two types of numeric variables we've encountered so far, `short` and `unsigned short`. Let's suppose that `x` is a `short`, and `y` is an `unsigned short`, both of them currently holding the value 32767, or 7fff in hex.

3. What is the result of adding 1 to `y`, in both decimal and hex?

4. What is the result of adding 1 to `x`, in both decimal and hex?

Answers to exercises can be found at the end of the chapter.

Registering Relief

Before we took this detour into the binary and hexadecimal number systems, I promised to explain what it means to say that the instructions using the 16-bit register names "behave exactly as though they were accessing the 16-bit registers on earlier machines". After a bit more preparation, we'll be ready for that explanation.

First, let's take a look at some characteristics of the human-readable version of machine instructions: assembly language instructions. The **assembly language** instructions we will look at have a fairly simple format.³⁴ The name of the operation is given first, followed by one or more spaces. The next element is the "destination", which is the register or RAM location that will be affected by the instruction's execution. The last element in an instruction is the "source", which represents another register, a RAM location, or a constant value to be used in the calculation. The source and destination are separated by a comma.³⁵ Here's an example of a simple assembly language instruction:

```
add ax,1
```

In this instruction, `add` is the operation, `ax` is the destination, and the constant value `1` is the source.

Thus, `add ax, 1` means to add 1 to the contents of **ax**, replacing the old contents of `ax` with the result.

Let's see what Susan has to say about the makeup of an assembly language instruction:

Susan: So the destination can be a register, cache, or RAM?

Steve: Yes, that's right. However, I should make it clear that the cache is transparent to the programmer. That is, you don't say "write to the cache" or "read from the cache"; you just use the RAM addresses and the hardware takes care of using the cache as appropriate to speed up access to frequently used locations. On the other hand, you do have to address registers explicitly when writing an assembly language program.

Now we're finally ready to see what the statement about using the 16-bit register names on a 32-bit machine means. Suppose we have the register contents shown in Figure [beforeaddax.1](#) (indicated in hexadecimal).

32 and 16 bit registers, before `add ax, 1` (Figure [beforeaddax.1](#))

32-bit register	32-bit contents	16-bit register	16-bit contents
<code>eax</code>	<code>1235ffff</code>	<code>ax</code>	<code>ffff</code>

If we were to add 1 to register `ax`, by executing the instruction `add ax, 1`, the result would be as shown in Figure [afteraddax.1](#).

32 and 16 bit registers, after `add ax, 1` (Figure [afteraddax.1](#))

32-bit register	32-bit contents	16-bit register	16-bit contents
<code>eax</code>	<code>12350000</code>	<code>ax</code>	<code>0000</code>

In case this makes no sense, consider what happens when you add 1 to 9999 on a four digit counter such as an odometer. It "turns over" to 0000, doesn't it? The same applies here: ffff is the largest number that can be represented as four hex digits, so if you add 1 to a register that has only four (hex) digits of storage available, the result is 0000.

As you might imagine, Susan was quite intrigued with the above detail; here is her reaction.

Susan: I have a understanding retention half-life of about 30 nanoseconds, but while I was reading this I was understanding it except I am boggled as to how adding 1 to ffff makes 0000, see, I am still not clear on Hex. Question: When you show the contents of a 32-bit register as being 12350000, then is the 1235 the upper half and the 0000 the lower half? Is that what you are saying?

Steve: That's right!

As this illustrates, instructions that refer to `ax` have no effect whatever on the upper part of `eax`; they behave exactly as though the upper part of `eax` did not exist. However, if we were to execute the instruction `add eax,1` instead of `add ax,1`, the result would look like Figure [afteraddeax.1](#).

32 and 16 bit registers, after `add eax,1` (Figure [afteraddeax.1](#))

32-bit register	32-bit contents	16-bit register	16-bit contents
<code>eax</code>	12360000	<code>ax</code>	0000

In this case, `eax` is treated as a whole. Similar results apply to the other 32-bit registers and their 16-bit counterparts.

On a RAMpage

Unfortunately, it isn't possible to use only registers and avoid references to RAM entirely, if only because we'll run out of registers sooner or later. This is a good time to look back at the diagram of the "memory hierarchy" (figure [hierarchyfig](#)) and examine the relative speed and size of each different kind of memory.

The "size" attribute of the disk and RAM are specified in Megabytes, whereas the size of an external cache is generally in the range from 64 Kilobytes to 1 Megabyte. As I mentioned before, the internal cache is considerably smaller, usually in the 8 to 16 Kilobyte range. The registers, however, provide a total of 28 *bytes* of storage; this should make clear that they are the scarcest memory resource. To try to clarify why the registers are so important to the performance of programs, I've listed the "speed" attribute in number of accesses per second, rather than in milliseconds, nanoseconds, and so forth. In the case of the disk, this is about 100 accesses per second. RAM can be accessed about 14 million times per second. The clear winners, though, are the internal cache and the registers, which can be accessed 66 million times per second and 133 million times per second, respectively.

Registering Bewilderment

In a way, the latter figure (133 million accesses per second for registers) overstates the advantages of registers relative to the cache. You see, any given register can be accessed only 66 million times per second; however, many instructions refer to two registers and still execute in one CPU cycle. Therefore, the maximum number of references per second is more than the number of instructions per second.

However, this leads to another question: Why not have instructions that can refer to more than one memory address (known as *memory-to-memory* instructions) and still execute in one CPU cycle? In that case, we wouldn't have to worry about registers; since there's (relatively) a lot of cache and very few registers, it would seem to make more sense to eliminate the middleman and simply refer to data in the cache.³⁶ Of course, there is a good reason for the provision of both registers and cache. The main drawback of registers is that there are so few of them; on the other hand, one of their main advantages is also that there are so few of them. Why is this?

The main reason to use registers is that they make instructions shorter: since there are only a few registers, we don't have to use up a lot of bits specifying which register(s) to use. That is, with eight registers, we only need 3 bits to specify which register we need. In fact, there are standardized 3-bit codes that might be thought of as "register addresses", which are used to specify each register when it is used to hold a variable. Figure [registermapfig](#) is the table of these register codes.³⁷

32 and 16 bit register codes (Figure registermapfig)

Register address	16-bit register	32-bit register
000	ax	eax
001	cx	ecx
010	dx	edx

011	bx	ebx
100	sp	esp
101	bp	ebp
110	si	esi
111	di	edi

By contrast, with a "memory-to-memory" architecture, each instruction would need at least 2 bytes for the source address, and 2 bytes for the destination address.³⁸ Adding 1 byte to specify what the instruction is going to do, this would make the minimum instruction size 5 bytes, whereas some instructions that use only registers can be as short as 1 byte. This makes a big difference in performance because the caches are quite limited in size; big programs don't fit in the caches, and therefore require a large number of RAM accesses. As a result, they execute much more slowly than small programs.

Slimming the Program

This explains why we want our programs to be smaller. However, it may not be obvious why using registers reduces the size of instructions, so here's an explanation.

Most of the data in use by a program are stored in RAM. When using a 32-bit CPU, it is theoretically possible to have over 4 billion bytes of memory (2^{32} is the exact number). Therefore, that many distinct addresses for a given byte of data are possible; to specify any of these requires 32 bits. Since there are only a few registers, specifying which one you want to use takes only a few bits; therefore, programs use register addresses instead of memory addresses wherever possible, to reduce the number of bits in each instruction required to specify addresses.

I hope this is clear, but it might not be. It certainly wasn't to Susan. Here's the conversation we had on this topic:

Susan: I see that you are trying to make a point about why registers are more efficient in terms of making instructions shorter, but I just am not picturing exactly how they do this. How do you go from "make the instructions much shorter" to "we don't have to use up a lot of bits specifying which registers to use"?

Steve: Let's suppose that we want to move data from one place to another in memory. In that case, we'll have to specify two addresses: the "from" address and the "to" address. One way to do this is to store the addresses in the machine language instruction. Since each address is at least 16 bits, an instruction that contains two

addresses needs to occupy at least 32 bits just for the addresses, as well as some more bits to specify exactly what instruction we want to perform. Of course, if we're using 32-bit addresses, then a "two-address" instruction would require 64 bits just for the two addresses, in addition to whatever bits were needed to specify the type of instruction.

Susan: OK. . . think I got this. . .

Steve: On the other hand, if we use registers to hold the addresses of the data, we need only enough bits to specify each of two registers. Since there aren't that many registers, we don't need as many bits to specify which ones we're referring to. Even on a machine that has 32 general registers, we'd need only 10 bits to specify two registers; on the Intel machines, with their shortage of registers, even fewer bits are needed to specify which register we're referring to.

Susan: Are you talking about the bits that are needed to define the instruction?

Steve: Yes.

Susan: How would you know how many bits are needed to specify the two registers?

Steve: If you have 32 different possibilities to select from, you need 5 bits to specify one of them, because 32 is 2 to the fifth power. If we have 32 registers, and any of them can be selected, that takes 5 bits to select any one of them. If we have to select two registers on a CPU with 32 registers, we need 10 bits to specify both registers.

Susan: So what does that have to do with it? All we are talking about is the instruction that indicates "select register" right? So that instruction should be the same and contain the same number of bits whether you have 1 or 32 registers.

Steve: There is no "select register" instruction. Every instruction has to specify whatever register or registers it uses. It takes 5 bits to select 1 of 32 items and only 3 bits to select 1 of 8 items; therefore, a CPU design that has 32 registers needs longer instructions than one that has only 8 registers.

Susan: I don't see why the number of registers should have an effect on the number of bits one instruction should have.

Steve: If you have two possibilities, how many bits does it take to select one of them? 1 bit. If you have four possibilities, how many bits does it take to select one of them? 2 bits. Eight possibilities require 3 bits; 16 possibilities require 4 bits; and finally 32 possibilities require 5 bits.

Susan: Some machines have 32 registers?

Steve: Yes. The PowerPC, for example. Some machines have even more registers than

that.

Susan: If the instructions to specify a register are the same, then why would they differ just because one machine has more than another?

Steve: They aren't the same from one machine to another. Although every CPU that I'm familiar with has registers, each type of machine has its own way of executing instructions, including how you specify the registers.

Susan: OK, and in doing so it is selecting a register, right? An instruction should contain the same number of bits no matter how many registers it has to call on.

Steve: Let's take the example of an add instruction, which as its name implies, adds two numbers. The name of the instruction is the same length, no matter how many registers there are; that's true. However, the actual representation of the instruction in machine language has to have room for enough bits to specify which register(s) are being used in the instruction.

Susan: They are statements right? So why should they be bigger or smaller if there are more or fewer registers?

Steve: They are actually machine instructions, not C++ statements. The computer doesn't know how to execute C++ statements, so the C++ compiler is needed to convert C++ statements into machine instructions. Machine instructions need bits to specify which register(s) they are using; so, with more registers available, more bits in the instructions have to be used to specify the register(s) that the instructions are using.

Susan: Do all the statements change the values of bits they contain depending on the number of registers that are on the CPU?

Steve: Yes, they certainly do. To be more precise, the machine language instructions that execute a statement are larger or smaller depending on the number of registers in the machine, because they need more bits to specify one of a larger number of registers.

Susan: "It takes five bits to select one of 32 items. . ."

". . .and only three bits to select one of eight items." Why?

Steve: What is a bit? It is the amount of information needed to select one of two alternatives. For example, suppose you have to say whether a light is on or off. How many possibilities exist? Two. Since a single bit has two possible states, 0 or 1, we can represent "on" by 1 and "off" by 0 and thus represent the possible states of the light by one bit.

Now suppose that we have a fan that has four settings: low, medium, high, and off. Is

one bit enough to specify the current setting of the fan? No, because one bit has only two possible states, while the fan has four. However, if we use two bits, then it will work. We can represent the states by bits as follows:

bits	state
----	-----
00	off
01	low
10	medium
11	high

Note that this is an arbitrary mapping; there's no reason that it couldn't be like this instead:

bits	state
----	-----
00	medium
01	high
10	off
11	low

However, having the lowest "speed" (that is, off) represented by the lowest binary value (00) and the increasing speeds corresponding to increasing binary values makes more sense and therefore is easier to remember.

This same process can be extended to represent any number of possibilities. If we have eight registers, for example, we can represent each one by 3 bits, as noted previously in figure [registermapfig](#) on page . That is the actual representation in the Intel architecture; however, whatever representation might have been used, it would require 3 bits to select among eight possibilities. The same is true for a machine that has 32 registers, except that you need 5 bits instead of 3.

Susan: Okay, so then does that mean that more than one register can be in use at a time? Wait, where is the room that you are talking about?

Steve: Some instructions specify only one register (a "one-register" instruction), while others specify two (a "two-register" instruction); some don't specify any registers. For example, most "branch" instructions are in the last category; they specify which address to continue execution from. These are used to implement `if` statements, `for` loops, and other flow control statements.

Susan: So, when you create an instruction you have to open up enough "room" to talk to all the registers at once?

Steve: No, you have to have enough room to specify any one register, for a one-register instruction, or any two registers for a two-register instruction.

Susan: Well, this still has me confused. If you need to specify only one register at any given time, then why do you always need to have all the room available? Anyway, where is this room? Is it in RAM or is it in the registers themselves? Let's say you are going to specify an instruction that uses only 1 of 32 registers. Are you saying that even though you are going to use just one register you have to make room for all 32?

Steve: The "room" that I'm referring to is the bits in the instruction that specify which register the instruction is using. That is, if there are eight registers and you want to use one of them in an instruction, 3 bits need to be set aside in the instruction to indicate which register you're referring to.

Susan: So you need the bits to represent the address of a register?

Steve: Right. However, don't confuse the "address of a register" with a memory address. They have nothing to do with one another, except that they both specify one of a number of possible places to store information. That is, register `ax` doesn't correspond to memory address 0, and so on.

Susan: Yes, I understand the bit numbers in relation to the number of registers.

Steve: That's good.

Susan: So the "address of a register" is just where the CPU can locate the register in the CPU, not an address in RAM. Is that right?

Steve: Right. The address of a register merely specifies which of the registers you're referring to; all of them are in the CPU.

After that comedy routine, let's go back to Susan's reaction to something I said earlier about registers

and variables:

Susan: The registers hold only variables. . . Okay, I know what is bothering me! What else is there besides variables? Besides nonvariables, please don't tell me that. (Actually that would be good, now that I think of it.) But this is where I am having problems. You are talking about data, and a variable is a type of data. I need to know what else is out there so I have something else to compare it with. When you say a register can hold a variable, that is meaningless to me, unless I know what the alternatives are and where they are held.

Steve: What else is there besides variables? Well, there are constants, like the number 5 in the statement `x = 5;`. Constants can also be stored in registers. For example, let's suppose that the variable `x`, which is a `short`, is stored in location 1237. In that case, the statement `x = 5;` might generate an instruction sequence that looks like this:

```
mov ax,5
```

```
mov [1237],ax
```

where the number in the `[]` is the address of the variable `x`. The first of these instructions loads 5 into register `ax`, and the second one stores the contents of `ax` (5, in this case) into the memory location 1237.

Sometimes, however, constants aren't loaded into registers as in this case but are stored in the instructions that use them. This is the case in the following instruction:

```
add ax,3
```

This means to add 3 to whatever was formerly in register `ax`. The 3 never gets into a register but is stored as part of the instruction.³⁹

A Fetching Tale

Another way of reducing overhead is to read instructions from RAM in chunks, rather than one at a time, and feed them into the CPU as it needs them; this is called *prefetching*. This mechanism operates in parallel with instruction execution, loading instructions from RAM into special dedicated registers in the CPU before they're actually needed; these registers are known collectively as the *prefetch queue*. Since the prefetching is done by a separate unit in the CPU, the time to do the prefetching doesn't increase the time needed for instruction execution. When the CPU is ready to execute another instruction, it can get it from the prefetch queue almost instantly, rather than having to wait for the slow RAM to provide each instruction. Of course, it does take a small amount of time to retrieve the next instruction from the prefetch queue, but that amount of time is included in the normal instruction execution time.

Susan: I don't understand prefetching. What are "chunks"? I mean I understand what

you have written, but I can't visualize this. So, there is just no time used to read an instruction when something is prefetched?

Steve: A separate piece of the CPU does the prefetching at the same time as instructions are being executed, so instructions that have already been fetched are available without delay when the execution unit is ready to "do" them.

The effect of combining the use of registers and prefetching the instructions can be very significant. In our example, if we use an instruction that has already been loaded, which reads data from and writes data only to registers, the timing reduces to that shown in Figure [highspeed](#).

Instruction execution time, using registers and prefetching (Figure highspeed)

Time	Function
0 ns	Read instruction from RAM

[40](#)

0 ns	Read data from register ^{41}
------	---

16 ns	Execute instruction
-------	---------------------

0 ns	Write result back to register ^{42}
------	---

16 ns	Total instruction execution time
-------	----------------------------------

As I indicated near the beginning of this chapter, the manufacturers aren't lying to us; if we design our programs to take advantage of these (and other similar) efficiency measures taken by the manufacturer, we can often approach the maximum theoretical performance figures. You've just been subjected to a barrage of information on how a computer works. Let's go over it again before continuing.

Review

Three main components of the computer are of most significance to programmers: disk, RAM, and the CPU; the first two of these store programs and data that are used by the CPU.

Computers represent pieces of information (or data) as binary digits, universally referred to as *bits*. Each bit can have the value 0 or 1. The binary system is used instead of the more familiar decimal system because it is much easier to make devices that can store and retrieve 1 of 2 values than 1 of 10. Bits are grouped into sets of eight, called *bytes*.

The disk uses magnetic recording heads to store and retrieve groups of a few hundred bytes on rapidly spinning platters in a few milliseconds. The contents of the disk are not lost when the power is turned off, so it is suitable for more or less permanent storage of programs and data.

RAM, which is an acronym for Random Access Memory, is used to hold programs and data while they're in use. It is made of millions of microscopic transistors on a piece of silicon called a *chip*. Each bit is stored using a few of these transistors. RAM does not retain its contents when power is removed, so it is not good for permanent storage. However, any byte in a RAM chip can be accessed in about 70 nanoseconds (billionths of a second), which is hundreds of thousands of times as fast as accessing a disk. Each byte in a RAM chip can be independently stored and retrieved without affecting other bytes, by providing the unique memory address belonging to the byte you want to access.

The CPU (also called the *processor*) is the active component in the computer. It is also made of millions of microscopic transistors on a chip. The CPU executes programs consisting of instructions stored in RAM, using data also stored in RAM. However, the CPU is so fast that even the typical RAM access time of 70 nanoseconds is a bottleneck; therefore, computer manufacturers have added both *external cache* and *internal cache*, which are faster types of memory used to reduce the amount of time that the CPU has to wait. The internal cache resides on the same chip as the CPU and can be accessed without delay. The external cache sits between the CPU and the regular RAM; it's faster than the latter, but not as fast as the internal cache. Finally, a very small part of the on-chip memory is organized as *registers*, which can be accessed within the normal cycle time of the CPU, thus allowing the fastest possible processing.

Conclusion

In this chapter, we've covered a lot of material on how a computer actually works. As you'll see, this background is essential if you're going to understand what really happens inside a program. In the next chapter, we'll get to the "real thing": how to write a program to make all this hardware do something useful.

Answers to Exercises

1. Hexadecimal arithmetic

a. 48

You probably won't be surprised to hear that Susan didn't care much for this answer originally. Here's the discussion on that topic:

Susan: Problem 1a. My answer is 38. Why? My own personal way of thinking: If $a = 10$ right? and if $e = 14$ and if $1 * 10 = 10$ and if $2 * 14 = 28$ then if you add $10 + 28$ you get 38. So please inform me how you arrived at 48? I didn't bother with the rest of the problems. If I couldn't get the first one right, then what was the point?

Steve: Here's how you do this problem:

$$1(1 * 16) + a(10 * 1)$$

$$2(2 * 16) + e(14 * 1)$$

$$3(3 * 16) + 18(24 * 1 = 1 * 16 + 8 * 1)$$

Carry the 1 from the low digit to the high digit of the answer, to produce:

$$4(4 * 16) + 8(8 * 1), \text{ or } 48 \text{ hex, which is the answer.}$$

b. 2a

c. 3e

2. Binary arithmetic

a. 1011

b. 10000

c. 111

3. 32768 decimal, or 8000 in hex

4. -32768, or 8000 in hex

Why is the same hex value rendered here as -32768, while it was 32768 in question 3? The only difference between `short` and `unsigned short` variables is how their values are interpreted. In particular, `short` variables having values from 8000h to ffffh are considered negative, while `unsigned short` values in that range are positive. That's why the range of `short` values is -32768 to +32767, whereas `unsigned short` variables can range from 0 to 65535.

Footnotes

1. Some people believe that you should learn C before you learn C++. Obviously, I'm not one of those people; for that matter, neither is the inventor of C++, Bjarne Stroustrup. On page 169 of his book, *The Design and Evolution of C++*, he says "Learn C++ first. The C subset is easier to learn for C/C++ novices and easier to use than C itself."
2. The concept I'm referring to is the *pointer*, in case you want to make a note of it here.
3. Whenever I refer to a *computer*, I mean a modern microcomputer capable of running MS-

DOS; these are commonly referred to as *PCs*. Most of the fundamental concepts are the same in other kinds of computers, but the details differ.

4. Although it's entirely possible to program without ever seeing the inside of a computer, you might want to look in there anyway, just to see what the CPU, RAM chips, disk drives, etc., look like. Some familiarization with the components would give you a head start if you ever want to expand the capacity of your machine.
5. Other hardware components can be important to programmers of specialized applications; for example, game programmers need extremely fine control on how information is displayed on the monitor. However, we have enough to keep us busy learning how to write general data-handling programs; you can always learn how to write games later, if you're interested in doing so.
6. Technically, this is a hard disk, to differentiate it from a floppy disk, the removable storage medium often used to distribute software or transfer files from one computer to another. Although at one time, many small computers used floppy disks for their main storage, the tremendous decrease in hard disk prices means that today even the most inexpensive computer stores programs and data on a hard disk.
7. The heads have to be as close as possible to the platters because the influence of a magnet (called the *magnetic field*) drops off very rapidly with distance. Thus, the closer the heads are, the more powerful the magnetic field is and the smaller the region that can be used to read and write data reliably. Of course, this leaves open the question of why the heads aren't in contact with the surface; that would certainly solve the problem of being too far away. Unfortunately, this seemingly simple solution would not work at all. There is a name for the contact of heads and disk surface while the disk is spinning, *head crash*. The friction caused by such an event destroys both the heads and disk surface almost instantly.
8. In some old machines, bytes sometimes contained more or less than 8 bits, but the 8-bit byte is virtually universal today.
9. In case you're not familiar with the \wedge notation, the number on its right indicates how many copies of the number to the left have to be multiplied together to produce the final result. For example, $2^5 = 2 * 2 * 2 * 2 * 2$, whereas $4^3 = 4 * 4 * 4$. Of course, I've just introduced another symbol you might not be familiar with: the $*$ is used to indicate multiplication in programming.
10. *RAM* is sometimes called "internal storage", as opposed to "external storage", that is, the disk.
11. Each switch is made of several transistors. Unfortunately, an explanation of how a transistor works would take us too far afield. Consult any good encyclopedia, such as the Encyclopedia Britannica, for this explanation.
12. There's also another kind of electronic storage, called **ROM**, for Read-Only Memory; as its name indicates, you can read from it, but you can't write to it. This is used for storing permanent information, such as the program that allows your computer to read a small program from your *boot disk*; that program, in turn, reads in the rest of the data and programs needed to start up the computer. This process, as you probably know, is called *booting* the computer. In case you're wondering where that term came from, it's an abbreviation for *bootstrapping*, which is intended to suggest the notion of pulling yourself up by your bootstraps. Also, you may have noticed that the terms RAM and ROM aren't symmetrical; why isn't RAM called RWM, Read-Write Memory? Because that's too hard to pronounce.
13. To be sure, an 1100 Megabyte disk cost more than \$80 in April 1997, but 1100 Megabytes of space on the 6510 Megabyte drive I mentioned earlier would represent a little less than \$80 of its total price of \$449.

14. The same disaster would happen if your system were to crash, which is not that unlikely if you're using certain popular PC graphically oriented operating environments whose names start with "W".
15. Most modern word processors can automatically save your work once in a while, for this very reason. I heartily recommend using this facility; it's saved my bacon more than once.
16. Each type of CPU has a different set of instructions, so that programs compiled for one CPU cannot in general be run on a different CPU. Some CPUs, such as the very popular 80x86 ones from Intel, fall into a "family" of CPUs in which each new CPU can execute all of the instructions of the previous family members. This allows upgrading to a new CPU without having to throw out all of your old programs, but correspondingly limits the ways in which the new CPU can be improved without affecting this "family compatibility".
17. You shouldn't get the idea from the coincidence of the Megahertz and MIPS numbers that 1 MIPS means the same as 1 MHz. It so happens that, for the 486, the fastest instructions take one clock cycle, and only one instruction can finish executing in each clock cycle. Therefore, if most of the instructions your program executes are one-cycle instructions, you can approach 66 MIPS on a 66 MHz 486. This relationship doesn't hold in general; for example, the Pentium™ machines can execute two instructions simultaneously in some cases, and therefore a 90 MHz Pentium can run at up to 180 MIPS in the ideal case.
18. In the case of MHz, *Mega* really means "million" (that is, 1,000,000), in contrast to its use in describing storage capacities. I'm sorry if this is confusing, but it can't be helped.
19. $1 \text{ second} / 226 \text{ ns per instruction} = 4,424,788 \text{ instructions per second}$.
20. As will be illustrated in Figure [highspeed](#).
21. These complementary roles played by RAM and the disk explain why the speed of the disk is also illustrated in the memory hierarchy.
22. There are other reasons to limit the size of an external cache. For one thing, it uses a lot of power and thus produces a lot of heat; this isn't good for electronic components.
23. Here, I'm assuming that this is a *direct-mapped cache*, which means that each cache "location" can hold exactly one item. It's also possible to have a cache that stores more than one item in a "location", in which case one of the other items already there will be displaced to make room for the new one. The one selected is usually the one that hasn't been accessed for the longest time, on the theory that it's probably not going to be accessed again soon; this is called the *least recently used* (abbreviated LRU) replacement algorithm.
24. This is fairly close to the actual way caches are used to reduce the time it takes to get frequently used data from RAM (known as *caching reads*); reducing the time needed to write changed values back to RAM (*caching writes*) is more complicated.
25. In case you're wondering how a small number of registers can help the speed of a large program, I should point out that no matter how large a program is, the vast majority of instructions and data items in the program are inactive at any given moment. In fact, less than a dozen instructions are in various stages of execution at any given time even in the most advanced CPU available in 1997. The computer's apparent ability to run several distinct programs simultaneously is an illusion produced by the extremely high rate of execution of instructions.
26. All of the registers are physically similar, being just a collection of circuits in the CPU used to hold a value. As indicated here, some registers are dedicated to certain uses by the design of the CPU, whereas others are generally usable. In the case of the general registers, which are all functionally similar or identical, a compiler often uses them in a conventional way; this stylized usage simplifies the compiler writer's job.

27. Since RAM doesn't maintain its contents when power is turned off, anything that a program needs to keep around for a long time, such as inventory data to be used later, should be saved on the disk. We'll see how that is accomplished in a future chapter.
28. If you think that last number looks familiar, you're right: it's the number of different values that I said could be stored in a type of numeric variable called a `short`. This is no coincidence; read on for the detailed explanation.
29. If neither of these does what you want, don't despair. Other types of numeric variables have different ranges; we'll go over them quickly in Appendix .
30. Paging Dr. Seuss. . .
31. In the early days of computing, base 8 was sometimes used instead of base 16, especially on machines that used 12-bit and 36-bit registers; however, it has fallen into disuse because almost all modern machines have 32-bit registers.
32. Either upper or lower case letters are acceptable to most programs (and programmers). I'll use lower case because such letters are easier to distinguish than upper case ones; besides, I find them less irritating to look at.
33. Please note that the ability to do binary or hexadecimal arithmetic is **not** essential to further reading in this book.
34. I'm simplifying here. There are instructions that follow other formats, but we'll stick with the simple ones for the time being.
35. Of course, the actual machine instructions being executed in the CPU don't have commas, register names, or any other human-readable form; they consist of fixed-format sequences of bits stored in RAM. The CPU actually executes machine language instructions rather than assembly language ones; a program called an **assembler** takes care of translating the assembly language instructions into machine instructions. However, we can usually ignore this step, because each assembly language instruction corresponds to one machine instruction. This correspondence is quite unlike the relationship between C++ statements and machine instructions, which is far more complex.
36. Perhaps I should remind you that the programmer doesn't explicitly refer to the cache; you can just use normal RAM addresses and let the hardware take care of making sure that the most frequently referenced data ends up in the cache.
37. Don't blame me for the seemingly scrambled order of the codes; that's the way Intel's CPU architects assigned them to registers when they designed the 8086 and it's much too late to change them now. Luckily, we almost never have to worry about their values, because the assembler takes care of the translation of register names to register addresses.
38. If we want to be able to access more than 64 Kilobytes worth of data, which is necessary in most modern programs, we'll need even more room to store addresses.
39. We'll go into this whole notion of using registers to represent and manipulate variables in grotesque detail in Chapter [basics.htm](#).
40. Since the instruction is already in the prefetch queue, this step doesn't count against the execution time. Hence the 0 in the time column.
41. This time is included under "Execute instruction".
42. This time is included under "Execute instruction".

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Basics of Programming

Creative Programming?

After that necessary detour into the workings of the hardware, we can now resume our regularly scheduled explanation of the creative possibilities of computers. It may sound odd to describe computers as providing grand scope for creative activities: Aren't they monotonous, dull, unintelligent, and extremely limited? Yes, they are. However, they have two redeeming virtues that make them ideal as the canvas of invention: They are extraordinarily fast and spectacularly reliable. These characteristics allow the creator of a program to weave intricate chains of thought and have a fantastic number of steps carried out without fail. We'll begin to explore how this is possible after we go over some definitions and objectives for this chapter.

Definitions

An **identifier** is a user defined name; variable names are identifiers. Identifiers must not conflict with keywords such as `if` and `while`; for example, you cannot create a variable with the name `while`.

A **keyword** is a word defined in the C++ language, such as `if` and `while`. It is illegal to define an identifier such as a variable name that conflicts with a keyword.

Objectives of This Chapter

By the end of this chapter, you should

1. Understand what a program is and have some idea how a program works.
2. Understand how to get information into and out of a program.
3. Understand how to use `if` and `while` to control the execution of a program.¹
4. Understand how a portion of a program can be marked off so that it will be treated as one unit.
5. Be able to read and understand a simple program I've written in C++.

Speed Demon

The most impressive attribute of modern computers, of course, is their speed; as we have already seen, this is measured in MIPS (millions of instructions per second).

Of course, raw speed is not very valuable if we can't rely on the results we get. ENIAC, one of the first electronic computers, had a failure every few hours, on the average; since the problems it was used to solve took about that much time to run, the likelihood that the results were correct wasn't very high. Particularly critical calculations were often run several times, and if the users got the same answer twice, they figured it was probably correct. By contrast, modern computers are almost incomprehensibly reliable. With almost any other machine, a failure rate of one in every million operations would be considered phenomenally low, but a computer with such a failure rate would make dozens of errors per second.²

Blaming It on the Computer

On the other hand, if computers are so reliable, why are they blamed for so much that goes wrong with modern life? Who among us has not been the victim of an erroneous credit report, or a bill sent to the wrong address, or been put on hold for a long time because "the computer is down"? The answer is fairly simple: It's almost certainly not the computer. More precisely, it's very unlikely that the CPU was at fault; it may be the software, other equipment such as telephone lines, tape or disk drives, or any of the myriad "peripheral devices" that the computer uses to store and retrieve information and interact with the outside world. Usually, it's the software; when customer service representatives tell you that they can't do something obviously reasonable, you can count on its being the software. For example, I once belonged to a 401K plan whose administrators provided statements only every three months, about three months after the end of the quarter; in other words, in July I found out how much my account had been worth at the end of March. The only way to estimate how much I had in the meantime was to look up the share values in the newspaper and multiply by the number of shares. Of course, the mutual fund that issued the shares could tell its shareholders their account balances at any time of the day or night; however, the company that administered the 401K plan didn't bother to provide such a service, as it would have required doing some work.³ Needless to say, whenever I hear that "the computer can't do that" as an excuse for such poor service, I reply "Then you need some different programmers."

That Does Not Compute

All of this emphasis on computation, however, should not blind us to the fact that computers are not solely arithmetic engines. The most common application for which PCs are used is word processing, which is hardly a hotbed of arithmetical calculation. While we have so far considered only numeric data, this is a good illustration of the fact that computers also deal with another kind of information, which is commonly referred to by the imaginative term **nonnumeric variables**. Numeric variables are those suited for use in calculations, such as in totalling a set of weights. On the other hand, nonnumeric data are items that are not used in calculations like adding, multiplying, or subtracting: Examples are names, addresses, telephone numbers, Social Security numbers, bank account numbers, or drivers license numbers. Note that just because something is called a *number*, or even is composed entirely of the digits 0-9, does not make it numeric data by our standards. The question is how the item is used. No one adds, multiplies, or subtracts drivers license numbers, for example; they serve solely as identifiers and could just as easily have letters in them, as indeed some do.

For the present, though, let's stick with numeric variables. Now that we have defined a couple of types of these variables, `short` and `unsigned short`, what can we do with them? To do anything with them, we have to write a C++ program, which consists primarily of a list of operations to be performed by the computer, along with directions that influence how these operations are to be translated into machine instructions. This raises an interesting point: Why does our C++ program have to be translated into machine instructions? Isn't the computer's job to execute (or *run*) our program?

Lost in Translation

Yes, but it can't run a C++ program. The only kind of program any computer can run is one made of machine instructions; this is called a **machine language** program, for obvious reasons. Therefore, to get our C++ program to run, we have to translate it into a machine language program. Don't worry, you won't have to do it yourself; that's why we have a program called a *compiler*.⁴ The most basic tasks that the compiler performs are the following:

1. Assigning memory addresses to variables. This allows us to use names for variables, rather than having to keep track of the address of each variable ourselves.
2. Translating arithmetic and other operations (such as +, -, etc.) into the equivalent machine instructions, including the addresses of variables assigned in the previous step.⁵

This is probably a bit too abstract to be easily grasped, so let's look at an example as soon as we have defined some terms. Each complete operation understood by the compiler is called a *statement*, and ends with a semicolon (;).⁶ Figure [littlenumeric](#) shows some sample statements that do arithmetic calculations.⁷

A little numeric calculation (Figure littlenumeric)

```
short i;

short j;

short k;

short m;

i = 5;

j = i * 3;           // j is now 15

k = j - i;          // k is now 10

m = (k + j) / 5;    // m is now 5

i = i + 1;          // i is now 6
```

To enter such statements in the first place, you can use any text editor that generates "plain" text files, such as the EDIT program that comes with DOS or Windows' Notepad. Whichever text editor you use, make sure that it produces files that contain only what you type; stay away from programs like Windows Write™ or Word for Windows™, as they add some of their own information to indicate fonts, type sizes, and the like to your file, which will foul up the compiler.

Once we have entered the statements for our program, we use the compiler, as indicated, to translate the programs we write into a form that the computer can perform; as defined in Chapter [prologue.htm](#), the form we create is called *source code*, since it is the source of the program logic, while the form of our program that the computer can execute is called an *executable program*, or just an *executable* for short.

As I've mentioned before, there are several types of variables, the `short` being only one of these types. Therefore, the compiler needs some explanatory material so that it can tell what types of variables you're using; that's what the first four lines of our little sample program fragment are for. Each line tells the compiler that the type of the variable `i`, `j`, `k`, or `m` is `short`; that is, it can contain values from -32768 to +32767.⁸ After this introductory material, we move into the list of operations to be performed. This is called the *executable* portion of the program, as it actually causes the computer to do something when the program is executed; the operations to be performed, as mentioned above, are called **statements**. The first one, `i = 5;`, sets the variable `i` to the value 5. A value such as 5, which doesn't have a name, but represents itself in a literal manner, is called (appropriately enough) a **literal** value.

This is as good a time as any for me to mention something that experienced C programmers take for granted but has a tendency to confuse novices. This is the choice of the `=` sign to indicate the operation of setting a variable to a value, which is known technically as **assignment**. As far as I'm concerned, an assignment operation would be more properly indicated by some symbol suggesting movement of data, such as `5 => i;`, meaning "store the value 5 into variable `i`". Unfortunately, it's too late to change the notation for the **assignment statement**, as such a statement is called, so you'll just have to get used to it. The `=` means "set the variable on the left to the value on the right".⁹ Now that I've warned you about that possible confusion, let's continue looking at the operations in the program. The next one, `j = i * 3;`, specifies that the variable `j` is to be set to the result of multiplying the current value of `i` by the literal value 3. The one after that, `k = j - i;`, tells the computer to set `k` to the amount by which `j` is greater than `i`; that is, `j - i`. The most complicated line in our little program fragment, `m = (k + j) / 5;`, calculates `m` as the sum of adding `k` and `j` and dividing the result by the literal value 5. Finally, the line `i = i + 1;` sets `i` to the value of `i` plus the literal value 1.

This last may be somewhat puzzling; how can `i` be equal to `i + 1`? The answer is that an assignment statement is *not* an algebraic equality, no matter how much it may resemble one. It is a command telling the computer to assign a value to a variable. Therefore, what `i = i + 1;` actually means is "Take the current value of `i`, add 1 to it, and store the result back into `i`." In other words, a C++ variable is a place to store a value; the variable `i` can take on any number of values, but only one at a time; any former value is lost when a new one is assigned.

This notion of assignment was the topic of quite a few messages with Susan. Let's go to the first round:

Susan: I am confused with the statement `i = i + 1;` when you have stated previously that `i = 5;`. So, which one is it? How can there be two values for `i`?

Steve: There can't; that is, not at one time. However, `i`, like any other variable, can take on any number of values, one after another. First, we set it to 5; then we set it to 1 more than it was before (`i + 1`), so it ends up as 6.

Susan: Well, the example made it look as if the two values of `i` were available to be used by the computer at the same time. They were both lumped together as executable material.

Steve: After the statement `i = 5;`, and before the statement `i = i + 1;`, the value of `i` is 5. After the statement `i = i + 1;`, the value of `i` is 6. The key here is that a variable such as `i` is just our name for some area of memory that can hold only one value at one time. Does that clear it up?

Susan: So, it is not like algebra? Then `i` is equal to an address of memory and does not really equate with a numerical value? Well, I guess it does when you assign a numerical value to it. Is that it?

Steve: Very close. A variable in C++ isn't really like an algebraic variable, which has a value that has to be figured out and doesn't change in a given problem. A programming language variable is just a name for a storage location that can contain a value.

With any luck, that point has been pounded into the ground, so you won't have the same trouble that Susan did. Now let's look at exactly what an assignment statement does. If the value of `i` before the statement `i = i + 1;` is 5 (for example), then that statement will cause the CPU to perform the following steps:¹⁰

1. Take the current value of `i` (5).
2. Add one to that value (6).
3. Store the result back into `i`.

After the execution of this statement, `i` will have the value 6.

What's Going on Underneath?

In a moment we're going to dive a little deeper into how the CPU accomplishes its task of manipulating data, such as we are doing here with our arithmetic program. First, though, it's time for a little pep talk for those of you who might be wondering exactly why this apparent digression is necessary. It's because if you don't understand what is going on under the surface, you won't be able to get past the "Sunday driver" stage of programming in C++. In some languages it's neither necessary or perhaps even possible to find out what the computer actually does to execute your program, but C++ isn't one of them. A good C++ programmer needs an intimate acquaintance with the internal workings of the language, for reasons which will become very apparent when we get to Chapter [string.htm](#). For the moment, you'll just have to take my word that working through these intricacies is essential; the payoff for a thorough grounding in these fundamental concepts of computing will be worth the struggle.

Now let's get to the task of exploring how the CPU actually stores and manipulates data in memory. As we saw previously, each memory location in RAM has a unique *memory address*; *machine instructions* that refer to RAM use this address to specify which *byte* or bytes of memory they wish to retrieve or modify. This is fairly straightforward in the case of a 1-byte variable, where the instruction merely specifies the byte that corresponds to the variable. On the other hand, the situation isn't quite as simple in the case of a variable that occupies more than 1 byte. Of course, no law of nature says that an instruction couldn't contain a number of addresses, one for each byte of the variable. However, this solution is never adopted in practice, as it would make instructions much longer than they need to be. Instead, the address in such an instruction specifies the first byte of RAM occupied by the variable, and the other bytes are assumed to follow immediately after the first one. For example, in the case of a `short` variable, which as we have seen occupies 2 bytes of RAM, the instruction would specify the address of the first byte of the area of RAM in which the variable is stored. However, there's one point that I haven't brought up yet: how the data for a given variable are actually arranged in memory. For example, suppose that the contents of a small section of RAM (specified as two hex digits per byte) look like Figure [ram.value](#).

A small section of RAM (Figure [ram.value](#))

Address	Hex byte value
1000	41
1001	42
1002	43
1003	44
1004	00

Also suppose that a `short` variable `i` is stored starting at address 1000. To do much with a variable, we're going to have to load it into a *general register*, one of the small number of named data storage locations in the CPU intended for general use by the programmer; this proximity allows the CPU to operate on data in the registers at maximum speed. You may recall that there are seven general registers in the 386 CPU (and its successors); they're named `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`.¹¹ Unfortunately, there's another complication here; these registers are designed to operate on 4-byte quantities, while our variable `i`, being of type `short`, is only two bytes long. Are we out of luck? No, but we do have to specify how long the variable is that we want to load. This problem is not unique to Intel CPUs, since any CPU has to have the ability to load different-sized variables into registers. Different CPUs use different methods of specifying this important piece of information; in the Intel CPUs, one way to do this is to alter the register name.¹² As we saw in the discussion of the development of Intel machines, we can remove the leading `e` from the register name to specify that we're dealing with 2-byte values; the resulting name refers to the lower two bytes of the 4-byte register. Therefore, if we wanted to load the value of `i` into register `ax` (that is, the lower half of

register `eax`), the instruction could be written as follows:¹³ `mov ax, [1000]`¹⁴ As usual, our resident novice Susan had some questions on this topic. Here is our conversation:

Susan: If you put something into 1000 that is "too big" for it, then it spills over to the next address?

Steve: Sort of. When you "put something into 1000", you have to specify exactly what it is you're "putting in". That is, it must be either a `short`, a `char`, or some other type of variable that has a defined size.

Susan: Is that how it works? Why then is it not necessary to specify that it is going to have to go into 1000 and 1001? So what you put in is not really in 1000 anymore, it is in 1000 *and* 1001? How do you refer to its REAL address? What if there is no room in 1001? Would it go to 2003 if that is the next available space?

Steve: Because the rule is that you always specify the starting address of any item (variable or constant) that is too big to fit in 1 byte. The other bytes of the item are always stored immediately following the address you specify. No bytes will be skipped when storing (or loading) one item; if the item needs 4 bytes and is to be stored starting at 1000, it will be stored in 1000-1003.

Susan: I see. In other words, the compiler will always use the next bytes of RAM, however many need to be used to store the item?

Steve: Right.

Who's on First?

Now I have a question for you. After we execute the assembly language statement `mov ax, [1000]` to load the value of `i` into `ax`, what's in register `ax`? That may seem like a silly question; the answer is obviously the value of `i`. Yes, but what is that value exactly? The first byte of `i`, at location 1000, has the value 41 hexadecimal (abbreviated 41h), and the second byte, at location 1001, has the value 42h. But the value of `i` is 2 bytes long; is it 4142h or 4241h? These are clearly not the same!

That was a trick question; there's no way for you to deduce the answer with only the information I've given you so far. The answer happens to be 4241h, because that's the way Intel decided to do it; that is, the low part of the value is stored in the byte of RAM where the variable starts. Some other CPUs do it the opposite way, where the high part of the value is stored in the byte of RAM where the variable starts; this is called *big-endian*, since the big end of the value is first, while the Intel way is correspondingly called *little-endian*. And some machines, such as the Power PC, can use either of these methods according to how they are started up. This makes it easier for them to run software written for either memory orientation.

As you might have surmised, the same system applies to 4-byte values; therefore, if we wrote the instruction `mov eax, [1000]`, since we're on a little-endian machine, it would load the `eax` register with the value 44434241h; that is, the four bytes 41, 42, 43, and 44 (hex) would be loaded into the `eax` register, with the byte having the lowest address loaded into the low end of the register.

Here's another example. A little-endian system would represent the number 1234 (hex) stored at address 5000 as in Figure [littleendian](#).

One little endian (Figure littleendian)

Address	Value
5000	34
5001	12

whereas a big-endian system would represent the same value 1234 (hex) as illustrated in Figure [bigendian](#).

A big endian example (Figure bigendian)

Address	Value
5000	12
5001	34

This really isn't much of a problem as long as we don't try to move data from one type of machine to another; however, when such data transportation is necessary, dealing with mixed endianness can be a real nuisance. Before going on, let's practice a bit with this notion of how data are stored in memory.

Exercises, First Set

1. Assume that a `short` variable named `z` starts at location 1001 in a little-endian machine. Using Figure [basicsex1](#) for the contents of memory, what is the value of `z`, in hex?

Exercise 1 (Figure basicsex1)

Address	Hex byte value
1000	3a
1001	43
1002	3c
1003	99
1004	00

Underware?

I can almost hear the wailing and tooth gnashing out there. Do I expect you to deal with all of these instructions and addresses by yourself? You'll undoubtedly be happy to learn that this isn't necessary, as the compiler takes care of these details. However, if you don't have some idea of how a compiler works, you'll be at a disadvantage when you're trying to figure out how to make it do what you want. Therefore, we're going to spend the next few pages "playing compiler"; that is, I'll examine each statement and indicate what action the compiler might take as a result. I'll simplify the statements a bit to make the explanation simpler; you should still get the idea. Figure [reallylittle](#) illustrates the set of statements that I'll compile:¹⁵

A really little numeric calculation (Figure reallylittle)

```
short i;

short j;
```

```
i = 5;
j = i + 3;
```

Compiler's Eye View

Here are the rules of this game:

1. All numbers in the C++ program are decimal; all addresses and numbers in the machine instructions are hexadecimal.[16](#)
2. All addresses are 2 bytes long.[17](#)
3. Variables are stored at addresses starting at 1000.
4. Machine instructions are stored at addresses starting at 2000.[18](#)
5. A number *not* enclosed in [] is a literal value, which represents itself. For example, the instruction `mov ax, 1000` means to move the value 1000 into the `ax` register.
6. A number enclosed in [] is an address, which specifies where data are to be stored or retrieved. For example, the instruction `mov ax, [1000]` means to move 2 bytes of data starting at location 1000, *not* the value 1000 itself, into the `ax` register.

Now, let's start compiling. The first statement, `short i;` tells me to allocate storage for a 2-byte variable called `i` that will be treated as `signed` (because that's the default). Since no value has been assigned to this variable yet, the resulting "memory map" looks like Figure [compile1](#).

Compiling, part 1 (Figure compile1)

Address	Source code variable name
1000	<code>i</code>

As you might have guessed, this exercise was the topic of a considerable amount of discussion with Susan. Here's how it started:

Susan: So the first thing we do with a variable is to tell the address that its name is `i`, but no one is home, right? It has to get ready to accept a value. Could you put a value in it without naming it, just saying address 1000 has a value of 5? Why does it have to be called `i` first?

Steve: The reason that we use names instead of addresses is because it's much easier for people to keep track of names than it is to keep track of addresses. Thus, one of the main functions of a compiler is to allow us to use names that are translated into addresses for the computer's use.

The second statement, `short j;` tells me to allocate storage for a 2-byte variable called `j` that will be treated as `signed` (because that's the default). Since no value has been assigned to this variable yet, the resulting "memory map" looks like Figure [compile2](#).

Compiling, part 2 (Figure compile2)

Address	Source code variable name
1000	<code>i</code>
1002	<code>j</code>

Here's the exchange about this step:

Susan: Why isn't the address for `j` 1001?

Steve: Because a `short` is 2 bytes, not 1. Therefore, if `i` is at address 1000, `j` can't start before 1002; otherwise, the second byte of `i` would have the same address as the first byte of `j`, which would cause chaos in the program. Imagine changing `i` and having `j` change by itself.

Susan: Okay. I just thought that each address represented 2 bytes for some reason. Then in reality each address always has just 1 byte?

Steve: Every byte of RAM has a distinct address, and there is one address for each byte of RAM. However, it is often necessary to read or write more than one byte at a time, as in the case of a `short`, which is 2 bytes in length. The machine instructions that read or write more than 1 byte specify only the address of the first byte of the item to be read or written; the other byte or bytes of that item follow the first byte immediately in memory.

Susan: Okay, this is why I was confused. I thought when you specified that the RAM address 1000 was a `short` (2 bytes), it just made room for 2 bytes. So when you specify address 1000 as a `short`, you know that 1001 will also be occupied with what you put in 1000.

Steve: Or to be more precise, location 1001 will contain the second byte of the `short` value that starts in byte 1000.

The next line is blank, so we skip it. This brings us to the statement `i = 5;` which is an executable statement, so we need to generate one or more machine instructions to execute it. We have already assigned address 1000 to `i`, so we have to generate instructions that will set the 2 bytes at address 1000 to the value that represents 5. One way to do this is to start by setting `ax` to 5, by the instruction `mov ax, 5`, then storing the contents of `ax` (5, of course) into the location where the value of `i` is kept, namely 1000, via the instruction `mov [1000], ax`.

Figure [compile3](#) shows what our "memory map" looks like so far.

Compiling, part 3 (Figure compile3)

Address	Variable Name		
1000	<code>i</code>		
1002	<code>j</code>		
Address	Machine Instruction	Assembly Language Equivalent	
2000	+-----+ b8 05 00	<code>mov ax, 5</code>	
2003	+-----+ a9 00 10	<code>mov [1000], ax</code>	
	+-----+		

Here's the next installment of my discussion with Susan on this topic:

Susan: When you use `ax` in an instruction, that is a register, not RAM?

Steve: Yes.

Susan: How do you know you want that register and not another one? What are the differences in the registers? Is `ax` the first register that data will go into?

Steve: For our current purposes, all of the 16-bit general registers (`ax`, `bx`, `cx`, `dx`, `si`, `di`, `bp`) are the same. Some of them have other uses, but all of them can be used for simple arithmetic such as we're doing here.

Susan: How do you know that you are not overwriting something more important than what you are presently writing?

Steve: In assembly language, the programmer has to keep track of that; in the case of a compiled language, the compiler takes care of it instead, which is another reason to use a compiler rather than writing assembly language programs yourself.

Susan: If it overwrites, you said important data will go somewhere else. How will you know where it went? How does it know whether what is being overwritten is important? Wait. If something is overwritten, it isn't gone, is it? It is just moved, right?

Steve: The automatic movement of data that you're referring to applies only to cached data being transferred to RAM. That is, if a slot in the cache is needed, the data that it previously held is written out to RAM without the programmer's intervention. However, the content of registers is explicitly controlled by the programmer (or the compiler, in the case of a compiled language). If you write something into a register, whatever was there before is gone. So don't do that if you need the previous contents!

Susan: How do you know that 5 will require 2 bytes?

Steve: In C++, because it's a `short`. In assembly language, because I'm loading it into `ax`, which is a 2-byte register.

Susan: Why do the the variable addresses start at 1000 and the machine addresses start at 2000?

Steve: It's arbitrary; I picked those numbers out of the air. In a real program, the compiler decides where to put things.

Susan: What do you mean by machine address? What is the machine? Where are the machine addresses?

Steve: A machine address is a RAM address. The machine is the CPU. Machine addresses are stored in the instructions so the CPU knows which RAM location we're referring to.

Susan: We talked about storing instructions before; is this what we are doing here? Are those instructions the "machine instructions"?

Steve: Yes.

Susan: Now, this may sound like a very dumb question, but please tell me where 5 comes from? I mean if you are going to move the value of 5 into the register `ax`, where is 5 hiding to take it from and to put it in `ax`? Is it stored somewhere in memory that has to be moved, or is it simply a function of the user just typing in that value?

Steve: It is stored in the instruction as a literal value. If you look at the assembly language illustration on page , you will see that the `mov ax, 5` instruction translates into the three bytes `b8 05 00`; the `05 00` is the 5 in "little-endian" notation.

Susan: Now, what is so magical about `ax` (or any register for that matter) that will transform the address 1000 to hold the value of 5?

Steve: The register doesn't do it; the execution of the instruction `mov [1000], ax` is what sets the memory starting at address 1000 to the value 5.

Susan: What are those numbers supposed to be in the machine instruction box? Those are bytes? Bytes of what? Why are they there? What do they do?

Steve: They represent the actual machine language program as it is executed by the CPU. This is where "the rubber meets the road". All of our C++ or even assembly language programs have to be translated into machine language before they can be executed by the CPU.

Susan: So this is where 5 comes from? I can't believe that there seems to be more code. What is b8 supposed to be? Is it some other type of machine language?

Steve: Machine language is exactly what it is. The first byte of each instruction is the "operation code", or "op code" for short. That tells the CPU what kind of instruction to execute; in this case, b8 specifies a "load register ax with a literal value" instruction. The literal value is the next 2 bytes, which represent the value 5 in "little-endian" notation; therefore, the full translation of the instruction is "load ax with the literal value 5".

Susan: So that is the "op code"? Okay, this makes sense. I don't like it, but it makes sense. Will the machine instructions always start with an op code?

Steve: Yes, there's always an op code first; that's what tells the CPU what the rest of the bytes in the instruction mean.

Susan: Then I noticed that the remaining bytes seem to hold either a literal value or a variable address. Are those the only possibilities?

Steve: Those are the ones that we will need to concern ourselves with.

Susan: I don't understand why machine addresses aren't in 2-byte increments like variable addresses.

Steve: Variable addresses aren't always in 2-byte increments either; it just happens that `short` variables take up 2 bytes. Other kinds of variables can and often do have other lengths.

Susan: So even though variable addresses are the same as instruction addresses they really aren't because they can't share the same actual address. That is why you distinguish the two by starting the instruction addresses at 2000 in the example and variable addresses at 1000, right?

Steve: Right. A particular memory location can hold only one data item at a time. As far as RAM is concerned, machine instructions are just another kind of data. Therefore, if a particular location is used to store one data item, you can't store anything else there at the same time, whether it's instructions or data.

The last statement, `j = i + 3;` is the most complicated statement in our program, and it's not that complicated. As with the previous statement, it's executable, which means we need to generate machine instructions to execute it. Because we haven't changed `ax` since we used it to initialize the variable `i` with the value 5, it still has that value. Therefore, to calculate the value of `j`, we can just add 3 to the value in `ax` by executing the instruction `add ax, 3`. After the execution of this instruction, `ax` will contain `i + 3`. Now all we have to do is to store that value in `j`. As indicated in the translation of the statement `short j;` the address used to hold the value of `j` is 1002. Therefore, we can set `j` to the value in `ax` by executing the instruction `mov [1002], ax`.

Figure [compile4](#) shows what the "memory map" looks like now.

Compiling, part 4 (Figure compile4)

Address	Variable Name
---------	---------------

1000	i
1002	j

Address	Machine Instruction	Assembly Language Equivalent
---------	---------------------	------------------------------

2000	+-----+ b8 05 00	mov ax,5
2003	+-----+ a9 00 10	mov [1000],ax

```

+-----+
2006 | 05 03 00 | add ax,3
+-----+
2009 | a9 02 10 | mov [1002],ax
+-----+

```

By the way, don't be misled by this example into thinking that all machine language instructions are 3 bytes in length. It's just a coincidence that all of the ones I've used here are of that length. The actual size of an instruction on the Intel CPUs can vary considerably, from 1 byte to a theoretical maximum of 12 bytes. Most instructions in common use, however, range from 1 to 5 bytes.

Here's the rest of the discussion that we had about this little exercise:

Susan: In this case `mov` means add, right?

Steve: No, `mov` means "move" and `add` means "add". When we write `mov ax, 5`, it means "move the value 5 into the `ax` register". The instruction `add ax, 3` means "add 3 to the current contents of `ax`, replacing the old contents with this new value".

Susan: So you're moving 5 but adding 3? How do you know when to use `mov` and when to use `add` if they both kind of mean the same thing?

Steve: It depends on whether you want to replace the contents of a register without reference to whatever the contents were before (`mov`) or add something to the contents of the register (`add`).

Susan: OK, here is what gets me: how do you get from address 1000 and `i=5` to `ax`? No, that's not it; I want you to tell me what is the relationship between `ax` and address 1000. I see `ax` as a register and that should contain the addresses, but here you are adding `ax` to the address. This doesn't make sense to me. Where are these places? Is address 1000 in RAM?

Steve: The `ax` register doesn't contain an address. It contains data. After the instruction `mov ax, 5`, `ax` contains the number 5. After the instruction `mov [1000], ax`, memory location 1000 contains a copy of the 2-byte value in register `ax`; in this case, that is the value of the `short` variable `i`.

Susan: So do the machine addresses represent actual bytes?

Steve: The machine addresses specify the RAM locations where data (and programs) are stored.

Execution Is Everything

Having examined what the compiler does at **compile time** with the preceding little program fragment, the next question is what happens when the compiled program is executed at **run time**. When we start out, the sections of RAM we're concerned with will look like Figure [ram1](#).

Before execution (Figure ram1)

Register	Contents	
<code>ax</code>	??	

Address	Contents	Variable Name
1000	?? ??	<code>i</code>
1002	?? ??	<code>j</code>

Address	Machine Instruction	Assembly Language Equivalent
* 2000	b8 05 00	mov ax,5
2003	a9 00 10	mov [1000],ax
2006	05 03 00	add ax,3
2009	a9 02 10	mov [1002],ax

First, a couple of rules for this part of the "game":

1. The asterisk on the left of the lower block indicates the next instruction to be executed.
2. We put ?? in the variable and register contents to start out with, to indicate that we haven't stored anything in them yet, and so we don't know what they contain.

Now let's start executing the program. The first instruction, `mov ax,5`, as we saw earlier, means "set the contents of `ax` to the value 5".

Here's the situation after `mov ax,5` is executed:

Register	Contents
ax	5

Address	Contents	Variable Name
1000	?? ??	i
1002	?? ??	j

Address	Machine Instruction	Assembly Language Equivalent
2000	b8 05 00	mov ax,5
* 2003	a9 00 10	mov [1000],ax
2006	05 03 00	add ax,3
2009	a9 02 10	mov [1002],ax

As you can see, executing `mov ax,5` has updated the contents of `ax`, and we've advanced to the next instruction.

When we have executed the next instruction, `mov [1000],ax`, the situation looks like this:

Register	Contents
ax	5

Address	Contents	Variable Name
1000	05 00	i

```

+-----+
1002 |  ?? ??          |   j
+-----+

```

Address	Machine Instruction	Assembly Language Equivalent
2000	b8 05 00	mov ax,5
2003	a9 00 10	mov [1000],ax
* 2006	05 03 00	add ax,3
2009	a9 02 10	mov [1002],ax

The situation after executing `mov [1000],ax` is just like the previous situation, except that the contents of location 1000 are now known, and of course we have moved to the next instruction.

Here's the result after the next instruction, `add ax,3`, is executed:

Register	Contents
----------	----------

ax	8
----	---

Address	Contents	Variable Name
1000	05 00	i
1002	?? ??	j

Address	Machine Instruction	Assembly Language Equivalent
2000	b8 05 00	mov ax,5
2003	a9 00 10	mov [1000],ax
2006	05 03 00	add ax,3
* 2009	a9 02 10	mov [1002],ax

As expected, `add ax,3` has increased the contents of `ax` by the value 3, leaving the result of 8. Now we're ready for the final instruction.

Here's the situation after the final instruction, `mov [1002],ax`, has been executed:

Register	Contents
----------	----------

ax	8
----	---

Address	Contents	Variable Name
1000	05 00	i

```

+-----+
1002 | 08 00 | j
+-----+

```

Address	Machine Instruction	Assembly Language Equivalent
2000	b8 05 00	mov ax,5
2003	a9 00 10	mov [1000],ax
2006	05 03 00	add ax,3
2009	a9 02 10	mov [1002],ax

After executing the final instruction, `mov [1002],ax`, the variable `i` has the value 5 and the variable `j` has the value 8.

A Cast of Characters

This should give you some idea of how numeric variables and values work. But what about nonnumeric ones?

This brings us to the subject of two new variable types and the values they can contain. These are the **char** (short for "character") and its relative, the `string`. What are these good for, and how do they work?¹⁹ A variable of type `char` corresponds to 1 byte of storage. Since a byte has 8 bits, it can hold any of 256 (2^8) values; the exact values depend on whether it is signed or unsigned, as with the `short` variables we have seen before. Going strictly according to this description, you might get the idea that a `char` is just a "really short" numeric variable. A `char` indeed can be used for this purpose in cases where no more than 256 different numeric values are to be represented. In fact, this explains why you might want a signed `char`. Such a variable can be used to hold numbers from -128 to +127; an unsigned `char`, on the other hand, has a range from 0 to 255. This facility isn't used very much any more, but in the early days of C, memory was very expensive and scarce, so it was sometimes worth the effort to use 1-byte variables to hold small values.

However, the main purpose of a `char` is to represent an individual letter, digit, punctuation mark, "special character" (e.g., \$, @, #, %, and so on) or one of the other "printable" and displayable units from which words, sentences, and other textual data such as this paragraph are composed.²⁰ The 256 different possibilities are plenty to represent any character in English, as well as a number of other European languages; in fact, this is one of the main reasons that there are 8 bits in a byte, rather than some other number.

Of course, the written forms of "ideographic" languages such as Chinese and Korean consist of far more than 256 characters, so 1 byte isn't going to do the trick for these languages. While they have been supported to some extent by schemes that switch among a number of sets of 256 characters each, such clumsy approaches to the problem made programs much more complicated and error prone. As the international market for software is increasing rapidly, it has become more important to have a convenient method of handling large *character sets*; as a result, a standard method of representing the characters of such languages by using 2 bytes per character has been developed. It's called the "Unicode standard". There's even a proposed solution that uses 32 bits per character, for the day when Unicode doesn't have sufficient capacity; that should take care of any languages that alien civilizations might introduce to our planet.

Since one `char` isn't good for much by itself, we often use groups of them, called `strings`, to make them easier to handle. Just as with numeric values, these variables can be set to literal values, which represent themselves. Figure [characters](#) is an example of how to specify and use each of these types we've just encountered. This is the first complete program we've seen, so there are a couple of new constructs that I'll have to explain to you.

By the way, in case the program in Figure [characters](#) doesn't seem very useful, that's because it isn't; it's just an example of the syntax of defining and using variables and literal values. However, we'll use these constructs to do useful work later, so going over them now isn't a waste of time.

Some real characters and strings (code/basic00.cc) (Figure characters)

<code/basic00.cc>

Why do we need the line `#include "string6.h"`? Because we have to tell the compiler how to manipulate strings. They aren't built in to its knowledge base. For the moment, it's enough to know that the contents of the file `string6.h` are needed to tell the compiler how to use strings; we'll get into the details of this mechanism later, starting in Chapter <string.htm>.

However, since we're already on the subject of files, this would be a good time to point out that the two main types of files in C++ are implementation files (also known as source files), which in our case have the extension `.cc`, and header files, which by convention have the extension `.h`.²¹ Implementation files contain statements that result in executable code, while each header file contains information that allows us to access a set of language features. We'll get into this in more detail in Chapter <function.htm>. For now, you'll just have to take my word that this is necessary; I promise I'll explain what it really means when you have enough background to understand the explanation.

You may also be puzzled by the function of the other statements in this program. If so, you're not alone. Let's see the discussion that Susan and I had about that topic.

Susan: Okay, in the example *why* did you have to write `c2 = c1`? Why not `B`? Why make one thing the same thing as the other? Make it different. Why would you even want `c2=c1`; and not just say `c1` twice, if that is what you want?

Steve: It's very hard to think up examples that are both simple enough to explain and realistic enough to make sense. You're right that this example doesn't do anything useful; I'm just trying to introduce what both the `char` type and the `string` type look like.

Susan: Come to think of it, what does `c1 = 'A'` have to do with the statement `s1 = "This is a test"`? I don't see any relationship between one thing and the other.

Steve: This is the same problem as the last one. They have nothing to do with one another; I'm using an admittedly contrived example to show how these variables are used.

Susan: I am glad now that your example of chars and strings (put together) didn't make sense to me. That is progress; it wasn't supposed to.

What does this useless but hopefully instructive program do? As is always the case, we have to tell the compiler what the types of our variables are before we can use them. In this case, `c1` and `c2` are of type `char`, whereas `s1` and `s2` are `strings`. After taking care of these formalities, we can start to use the variables. In the first executable statement, `c1 = 'A'`; we set the `char` variable `c1` to a literal value, in this case a capital `A`; we need to surround this with single quotation marks (`'`) to tell the compiler that we mean the letter `A` rather than a variable named `A`. In the next line, `c2 = c1`; we set `c2` to the same value as `c1` holds, which of course is `'A'` in this case. The next executable statement `s1 = "This is a test"`; as you might expect, sets the `string` variable `s1` to the value `"This is a test"`,²² which is a literal of a type called a **C string**. Don't confuse a C string literal with a `string`. A C string literal is a type of literal that we use to assign values to variables of type `string`. In the statement `s1 = "This is a test"`; we use a quotation mark, in this case the double quote (`"`), to tell the compiler where the literal value starts and ends.

You may be wondering why we need two different kinds of quotes in these two cases. The reason is that there are actually two types of nonnumeric data, *fixed-length data* and *variable-length data*. Fixed-length data are relatively easy to handle in a program, as the compiler can set aside the correct amount of space in advance. Variables of type `char` are 1 byte long and can thus contain exactly one character; as a result, when we set a `char` to a literal value, as we do in the line `c1 = 'A'`; the code that executes that statement has the simple task of copying exactly 1 byte representing the literal `'A'` to the address reserved for variable `c1`.²³

However, C string literals such as `"This is a test"` are variable-length data, and dealing with such data isn't so easy. Since there could be any number of characters in a C string, the code that does the assignment of a literal value like `"This is a test"` to a `string` variable has to have some way to tell where the literal value ends. One possible way to provide this

needed information would be for the compiler to store the length of the C string literal in the memory location immediately before the first character in the literal. I would prefer this method; unfortunately, it is not the method used in the C language (and its descendant the C++ language). To be fair, the inventors of C didn't make an arbitrary choice; they had reasons for their decision on how to indicate the length of a string. You see, if we were to reserve only 1 byte to store the actual length in bytes of the character data in the string, then the maximum length of a string would be limited to 255 bytes. This is because the maximum value that could be stored in the length byte, as in any other byte, is 255. Thus, if we had a string longer than 255 bytes, we would not be able to store the length of the string in the 1 byte reserved for that purpose. On the other hand, if we were to reserve 2 bytes for the length of each string, then programs that contain many strings would take more memory than they should.

While the extra memory consumption that would be caused by using a 2-byte length code may not seem significant today, the situation was considerably different when C was invented. At that time, conserving memory was very important; the inventors of C therefore chose to mark the end of a C string by a byte containing the value 0, which is called a **null byte**.²⁴ This solution has the advantage that only one extra byte is needed to indicate the end of a C string of any length. However, it also has some serious drawbacks. First, this solution makes it impossible to have a byte containing the value 0 in the middle of a C string, as all of the C string manipulation routines would treat that null byte as being the end of the C string. Second, it is a nontrivial operation to determine the length of a C string; the only way to do it is to scan through the C string until you find a null byte. As you can probably tell, I'm not particularly impressed with this mechanism; nevertheless, as it has been adopted into C++ for compatibility with C, we're stuck with it for literal strings in our programs.²⁵ Therefore, the literal string "ABCD" would occupy 5 bytes, 1 for each character, and 1 for the null byte that the compiler adds automatically at the end of the literal. But we've skipped one step: How do we represent characters in memory? There's no intuitively obvious way to convert the character 'A' into a value that can be stored in 1 byte of memory.

The answer, at least for our purposes in English, is called the **ASCII code** standard. This stands for American Standard Code for Information Interchange, which as the name suggests was invented precisely to allow the interchange of data between different programs and makes of computers. Before the invention of ASCII, such interchange was difficult or impossible, since every manufacturer made up its own code or codes. Here are the specific character codes that we have to be concerned with for the purposes of this book:

1. The codes for the capital letters start with hex 41 for 'A', and run consecutively to hex 5a for 'Z'
2. The codes for the lower case letters start with hex 61 for 'a', and run consecutively to hex 7a for 'z'..²⁶
3. The codes for the numeric digits start with hex 30 for '0', and run consecutively to hex 39 for '9'.

Given these rules, the memory representation of the string "ABCD" might look something like Figure [abcd](#).

Yet another small section of RAM (Figure abcd)

Address	Hex value
1000	41
1001	42
1002	43
1003	44
1004	00 (null byte; that is, end of C string)

Now that we see how strings are represented in memory, I can explain why we need two kinds of quotes. The double quotes tell the compiler to add the null byte at the end of the string literal, so that when the assignment statement `s1 = "This is a test " ;` is executed, the program knows when to stop copying the value to the string variable.

A Byte by Any Other Name. . .

Have you noticed that I've played a little trick here? The illustration of the string "ABCD" should look a bit familiar; its memory contents are exactly the same as in Figure [ram.value](#), where we were discussing numeric variables. I did this to illustrate an important point: the contents of memory actually consists of uninterpreted bytes, which have meaning only when used in a particular way by a program. That is, the same bytes can represent numeric data or characters, depending on how they are referred to.

This is one of the main reasons why we need to tell the C++ compiler what types our variables have. Some languages allow variables to be used in different ways at different times, but in C++ any given variable always has the same type; for example, a `char` variable can't change into a `short`. At first glance, it seems that it would be much easier for programmers to be able to use variables any way they like; why is C++ so restrictive?

The C++ **type system**, as this feature of a language is called, is specifically designed to minimize the risk of misinterpreting or otherwise misusing a variable. It's entirely too easy in some languages to change the type of a variable without meaning to; the resulting bugs can be very difficult to find, especially in a large program. In C++, the usage of a variable can be checked by the compiler. This **static type checking** allows the compiler to tell you about many errors that otherwise would not be detected until the program is running (**dynamic type checking**). This is particularly important in systems that need to run continuously for long periods of time. While you can reboot your machine if your word processor crashes due to a run-time error, this is not acceptable as a solution for errors in the telephone network, for example.

Of course, you probably won't be writing programs demanding that degree of reliability any time soon, but strict static type checking is still worthwhile in helping eliminate errors at the earliest possible stage in the development of our programs.

Some Strings Attached

After that infomercial for the advantages of static type checking, we can resume our examination of strings. You may have noticed that there's a **space** character at the end of the string "This is a test ". That's another reason why we have to use a special character like " (the double quote) to mark the beginning and end of a string; how else would the compiler know whether that space is supposed to be part of the string or not? The space character is one of the **nonprinting characters** (or **nondisplay characters**) that controls the format of our displayed or printed information; imagine how hard it would be to read this book without space characters! While we're on the subject, I should also tell you about some other characters that have special meaning to the compiler. They are listed in Figure [specialchar](#).

Special characters for program text (Figure specialchar)

Name	Graphic	Use
Single quote	'	surrounds a single character value
Double quote	"	surrounds a multi-character value
Semicolon	;	ends a statement
Curly braces	{ }	groups statements together
Parentheses	()	surrounds part of a statement

[27](#)

Backslash \ Tells the compiler that the next character should be treated

differently from the way that

it would normally be treated.²⁸

I compiled Figure [specialchar](#) at the instigation of guess who:

Susan: How about you line up all your cute little " ' \ ; things and just list their meanings? I forget what they are by the time I get to the next one. Your explanations of them are fine, but they are scattered all over the place; I just want one place that has all the explanations.

Steve: That's a good idea. As usual, you're doing a good job representing the novices; keep up the good work!

Our next task, after a little bit of practice with the memory representation of a C string, will be to see how we get the values of our strings to show up on the screen.

Exercises, Second Set

2. Assume that a C string literal starts at memory location 1001. If the contents of memory are as illustrated in Figure [basex2](#), what is the value of the C string?

A small section of RAM (Figure basex2)

Address	Hex value
1000	44
1001	48
1002	45
1003	4c
1004	4c
1005	4f
1006	00

In and Out

Most programs need to interact with their users, both to ask them what they want and to present the results when they are available. The computer term for this topic is **I/O** (short for "input/output"). We'll start by getting information from the keyboard and displaying it on the screen; later, we'll go over the more complex I/O functions that allow us to read and write data on the disk.

Figure [simple.output](#) shows how to display the text "This is a test and so is this." as promised. The meaning of << is suggested by its arrowlike shape. The information on its right is sent to the "output target" on its left. In this case, we're sending the information to one of the predefined destinations, **cout**, which stands for "character output".²⁹ Characters sent to **cout** are displayed on the screen.³⁰

Some simple output (code\basic01.cc) (Figure simple.output)

[code/basic01.cc](#)

This program will send the following output to the screen:

```
This is a test and so is this.
```

So much for (simple) output. Input from the keyboard is just as simple. Let's modify our little sample to use it, as shown in Figure [simple.input](#).

Some simple input and output (code\basic02.cc) (Figure simple.input)

[code/basic02.cc](#)

As you might have guessed, `cin` (shorthand for "character input") is the counterpart to `cout`, as `>>` is the counterpart to `<<`; `cin` supplies characters from the keyboard to the program via the `>>` operator.³¹ This program will wait for you to type in the first `string`, ended by hitting the ENTER key, then do the same for the second `string`. When you hit ENTER the second time, the program will display the first `string`, then a blank, and then the second `string`.

Susan had some questions about these little programs, beginning with the question of case sensitivity:

Susan: Are the words such as `cout` and `cin` case sensitive? I had capitalized a few of them just out of habit because they begin the sentence and I am not sure if that was the reason the compiler gave me so many error messages. I think after I changed them I reduced a few messages.

Steve: **Everything** in C++ is case sensitive. That includes keywords like `if`, `for`, `do`, and so on, as well as your own variables. That is, if you have a variable called `Name` and another one called `name`, those are completely different and unrelated to one another. You have to write `cin` and `cout` just as they appear here, or the compiler won't understand you.

If Only You Knew

In our examples so far, the program always executes the same statements in the same order. However, any real program is going to need to alter its behavior according to the data it is processing. For example, in a banking application, it might be necessary to send out a notice to a depositor whenever the balance in a particular account drops below a certain level; or perhaps the depositor would just be charged some exorbitant fee in that case. Either way, the program has to do something different depending on the balance. In particular, let's suppose that the "Absconders and Defaulters National Bank" has a minimum balance of \$10,000. Furthermore, let's assume that if you have less than that amount on deposit, you are charged a \$20 "service charge". However, if you are foolish enough to leave that ridiculous amount of money on deposit, then they will graciously allow you to get away with not paying them while they're using your money (without paying you interest, of course). To determine whether or not you should be charged for your checking account, the bank can use an **if statement**, as shown in Figure [if.statement](#).

Using an if statement (code\basic03.cc) (Figure if.statement)

[code/basic03.cc](#)

This program starts by displaying the line

```
Please enter your bank balance:
```

on the screen. Then it waits for you to type in your balance, followed by the ENTER key (so it knows when you're done). The conditional statement checks whether you're a "good customer". If your balance is less than \$10,000, the next statement is executed, which displays the line

Please remit \$20 service charge.³²

The phrase `<< endl` is new here. It means "we're done with this line of output; send it out to the screen". You could also use the special character `'\n'`, which means much the same thing; its official name is "newline".

Now let's get back to our regularly scheduled program. If the condition is **false** (that is, you have at least \$10,000 in the bank), the computer skips the statement that asks you to remit \$20; instead, it executes the one after the `else`, which tells you to have a nice day. That's what **else** is for; it specifies what to do if the condition specified in the `if` statement is `false` (that is, not `true`). If you typed in a number 10,000 or higher, the program would display the line

```
Have a nice day!
```

You don't have to specify an `else` if you don't want to. In that case, if the `if` condition isn't `true`, the program just goes to the next statement as though the `if` had never been executed.

While We're on the Subject

The `while` statement is another way of affecting the order of program execution. This conditional statement executes the statement under its control as long as a certain condition is `true`. Such potentially repeated execution is called a **loop**; a loop controlled by a **while** statement is called, logically enough, a *while loop*. Figure [while](#) is a program that uses a `while` loop to challenge the user to guess a secret number from 0 to 9, and keeps asking for guesses until the correct answer is entered.

Using a `while` statement (`code/basic04.cc`) (Figure [while](#))

[code/basic04.cc](#)

There are a few wrinkles here that we haven't seen before. Although the `while` statement itself is fairly straightforward, the meaning of its condition `!=` isn't intuitively obvious. However, if you consider the problem we're trying to solve, you'll probably come to the (correct) conclusion that `!=` means "not equal", since we want to keep asking for more guesses *while* the `Guess` is not equal to our `Secret` number.³³ Since there is a comparison operator that tests for "not equal", you might wonder how to test for "equal" as well; as is explained in some detail in the next chapter, in C++ we have to use `==` rather than `=` to compare whether two values are equal.

You might also be wondering whether an `if` statement with an `else` clause would serve as well as the `while`; after all, `if` is used to select one of two alternatives, and the `else` could select the other one. The answer is that this would allow the user to take only one guess before the program ends; the `while` loop lets the user try again as many times as needed to get the right answer.

Now you should have enough information to be able to write a simple program of your own. Susan asked for an assignment to do just that:

Susan: Based on what you have presented in the book so far, send me a setup, an exercise for me to try to figure out how to program, and I will give it a try. I guess that is the only way to do it. I can't even figure out a programmable situation on my own. So if you do that, I will do my best with it, and that will help teach me to think. (Can that be?) Now, if you do this, make it simple, and no tricks.

Of course, I did give her the exercise she asked for (exercise 3), but also of course, that didn't end the matter. She decided to add her own flourish, which resulted in exercise 4. These exercises follow below, right after some instructions on the mechanics of creating a program. The instructions assume that you've installed the software from the CD-ROM in the back of this book. Otherwise, follow the instructions in the back of the book to install the software, and then come back to these instructions.

1. Change to the `"\whos\code"` directory on the drive where you installed the compiler.
2. Type `RHIDE` to start the compiler's integrated development environment (IDE). Use `RHIDE`'s editor to create a source code file containing the source code for your program. To do this, select the File menu and click New. Then type in the code for your program, which we'll assume will be called "myprog".

3. When you get done writing the source code for your program, you will have to save it. To do this, select the File menu and click on Save. This will bring up a dialog box into which you will type the name of your file, which we'll assume is "myprog.cc" (without the quotes), and hit ENTER.
4. Once you have written the source code for your program, you will need to create a "project" to keep track of the various parts of the program. To do this, select the "Project" menu, click on "Open project", and then type in the name of the project that you want to create. If you want to call your project "myprog", you'll type "myprog" (without the quotes) and hit ENTER.
5. Now you will need to add the source file you just created to the project list so that it will be compiled. To do this, select the Project menu, then click on "Add item". Type in the name of your file, which is "myprog.cc", and hit ENTER.
6. The next step is to tell RHIDE to include the files that define how strings and vectors work. You can skip this step if you aren't using any strings or vectors, but most programs will eventually need to use one or more of these data types even if they don't need them at the start, so you might as well include these files. To do this, select the Project menu, then click on "Add item". Type in "string7.cc" and hit ENTER, then type in "wassert.cc" and hit ENTER.
7. When you have finished adding files to the project, click on the CANCEL button to close the "Add item" dialog box.
8. To compile your program, select the "Compile" menu and click "Make". This will compile any source files that haven't been compiled since they were written or last changed.
9. To run your program normally from a DOS prompt, make sure you are in the "\whos\code" directory, and then type the name of the program, without the extension. In this case, you would just type "myprog".
10. To run your program under the debugger, make sure you are in the "\whos\code" directory, and then type "RHIDE myprog" (substituting the name of your program for "myprog"). Again, do *not* add the ".cc" to the end of the file name. Once RHIDE has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger. The debugger also has a lot of other features, including displaying the values of variables during program execution, which I encourage you to explore.

Now here are the programs Susan came up with, along with some others that fall in the same category.

Exercises, Third Set

3. Write a program that asks the user to type in the number of people that are expected for dinner, not counting the user. Assuming that the number typed in is `n`, display a message that says "A table for `(n+1)` is ready.". For example, if the user types 3, display "A table for 4 is ready.".
4. Modify the program from exercise 3 to display an error message if the number of guests is more than 20.
5. Write a program that asks the user to type in his or her name and age. If the age is less than 47, then indicate that the user is a youngster; otherwise, that he or she is getting on in years.
6. Write a program that asks the user whether Susan is the world's most tenacious novice. If the answer is "true", then acknowledge the user's correct answer; if the answer is "false", then indicate that the answer is erroneous. If neither "true" nor "false" is typed in, chastise the user for not following directions.
7. Write a program that calculates how much extra allowance a teenager can earn by doing extra chores. Her allowance is calculated as \$10 if she does no extra chores; she gets \$1 additional for each extra chore she does.

Just up the Block

Our most recent programming example has contributed another item to our arsenal of programming weapons: namely, the ability to group several statements into one logical section of a program. That's the function of the **curly braces**, { and }. The first one of these starts such a section, called a **block**, and the second one ends the block. Because the two statements after the `while` are part of the same block, they are treated as a unit; both are executed if the condition in the `while` is `true`, and neither is executed if it is `false`. A block can be used anywhere that a statement can be used, and is treated in exactly the same way as if it were one statement.³⁴

At the Fair

Now we're ready to write a program that vaguely resembles a solution to a real problem. We'll start with a simple, rural type of programming problem.

Imagine that you are at a county fair. The contest for the heaviest pumpkin is about to get underway, and the judges have asked for your help in operating the "pumpkin scoreboard". This device has a slot for the current pumpkin weight (the `CurrentWeight` slot), and another slot for the highest weight so far (the `HighestWeight` slot); each slot can hold three digits from 0 to 9 and therefore can indicate any weight from 0 to 999. The judges want you to maintain an up-to-date display of the current weight and of the highest weight seen so far. The weights are expressed to the nearest pound. How would you go about this task?

Probably the best way to start is by setting the number in both slots to the weight of the first pumpkin called out. Then, as each new weight is called out, you change the number in the `CurrentWeight` slot to match the current weight; if it's higher than the number in the `HighestWeight` slot, you change that one to match as well. Of course, you don't have to do anything to the `HighestWeight` slot when a weight less than the previous maximum weight is called out, because a pumpkin with a lesser weight can't be the winner. How do we know when we are done? Since a pumpkin entered in this contest has to have a weight of at least 1 pound, the weigher calls out 0 as the weight when the weighing is finished. At that point, the number in the `HighestWeight` slot is the weight of the winner.

The procedure you have just imagined performing can be expressed a bit more precisely by the following algorithm:

1. Ask for the first weight.
2. Set the number in the `CurrentWeight` slot to this value.
3. Copy the number in the `CurrentWeight` slot to the `HighestWeight` slot.
4. Display both the current weight and the highest weight so far (which are the same, at this point)
5. While the `CurrentWeight` value is greater than 0 (that is, there are more pumpkins to be weighed), do steps 5a to 5d:
 1. Ask for the next weight.
 2. Set the number in the `CurrentWeight` slot to this weight.
 3. If the number in the `CurrentWeight` slot is greater than the number in the `HighestWeight` slot, copy the number in the `CurrentWeight` slot to the `HighestWeight` slot.
 4. Display the current weight and the highest weight so far.
6. Stop. The number in the `HighestWeight` slot is the weight of the winner.

Now we're ready to look at the actual pumpkin-weighing program. You've already seen most of the constructs that the program contains, but let's examine the role of the *preprocessor directive* `#include <iostream.h>`. This tells the compiler that we want to use the standard C++ I/O library. The term *preprocessor directive* is a holdover from the days when a separate program called the *preprocessor* handled functions such as `#include` before handing the program over to the compiler; these days, these facilities are provided by the compiler, but the name has stuck.

The `#include` command has the same effect as copying all of the code from a file called `iostream.h` into our file; `iostream.h` defines the I/O functions and variables `cout`, `cin`, `<<`, and `>>`, along with others that we haven't used yet. If we left this line out, none of our I/O statements would work.

Figure [pumpkin](#) is the translation of our little problem into C++.

A C++ Program (code\pump1.cc) (Figure pumpkin)

English

C++

--

First, we have to tell the compiler what we're up to in this program

--

Define the		
standard input		
and output		
functionality		#include <iostream.h>

This is the main		
part of the program		int main()

Start of program		{
------------------	--	---

Define variables		short CurrentWeight;
		short HighestWeight;

--

Here's the start of the "working" code

--

Ask for the first		
weight		cout << "Please enter the first weight: ";

Set the number in		
the CurrentWeight		
slot to the value		
entered by the user		cin >> CurrentWeight;

Copy the number in		
--------------------	--	--

```

the CurrentWeight      |
slot to the            |
HighestWeight slot     |   HighestWeight = CurrentWeight;
                        |
Display the current    |   cout << "Current weight " << CurrentWeight << endl;
and highest weights    |   cout << "Highest weight " << HighestWeight << endl;
                        |
While the number in   |
the CurrentWeight     |
slot is greater       |
than 0 (i.e., there   |
are more pumpkins     |
to be weighed)        |   while (CurrentWeight > 0)
Start repeated         |   {
steps                 |
                        |
Ask for the next       |
weight                |   cout << "Please enter the next weight: ";
                        |
Set the number in     |
the CurrentWeight     |
slot to this value    |   cin >> CurrentWeight;
                        |
If the number in     |
the CurrentWeight     |
slot is more than     |
the number in the     |
HighestWeight slot,   |   if (CurrentWeight > HighestWeight)

```

```

|
then copy the |
number in the |
CurrentWeight slot |
to the | HighestWeight = CurrentWeight;
HighestWeight slot |
|
Display the current | cout << "Current weight " << CurrentWeight << endl;
and highest weights | cout << "Highest weight " << HighestWeight << endl;
|
End repeated steps |
in while loop | }
|

```

--

We've finished the job; now to clean up

--

```

Tell the rest of |
the system we're |
okay | return 0;
|
End of program | }
|

```

--

Susan had some questions about variable names.

Susan: Tell me again what the different shorts mean in this figure. I am confused, I just thought a short held a variable like `i`. What is going on when you declare `HighestWeight` a short? So do the "words" `HighestWeight` work in the same way as `i`?

Steve: A `short` is a variable. The name of a `short` is made up of one or more characters; the first character must be a letter or an underscore (`_`), while any character after the first must be either a letter, an underscore, or a digit from 0 to 9. To define a `short`, you write a line that gives the name of the `short`. This is an example:
`short HighestWeight;`

Susan: OK, but then how does `i` take 2 bytes of memory and how does `HighestWeight` take up 2 bytes of memory? They look so different, how do you know that `HighestWeight` will fit into a `short`?

Steve: The length of the names that you give variables has nothing to do with the amount of storage that the variables take up. After the compiler gets through with your program, there aren't any variable names; each variable that you define in your source program is represented by the address of some area of storage. If the variable is a `short`, that area of storage is 2 bytes long; if it's a `char`, the area of storage is 1 byte long.

Susan: Then where do the names go? They don't go "into" the `short`?

Steve: A variable name doesn't "go" anywhere; it tells the compiler to set aside an area of memory of a particular length that you will refer to by a given name. If you write `short xyz;` you're telling the compiler that you are going to use a `short` (that is, 2 bytes of memory) called `xyz`.

Susan: If that is the case, then why bother defining the `short` at all?

Steve: So that you (the programmer) can use a name that makes sense to you. Without this mechanism, you'd have to specify everything as an address. Isn't it easier to say

```
HighestWeight = CurrentWeight;
```

rather than

```
mov ax,[1000]
```

```
mov [1002],ax
```

or something similar?

Susan also had a question about the formatting of the output statement `cout << "Highest weight " << HighestWeight << endl;`.

Susan: Why do we need both "Highest weight" and `HighestWeight` in this line?

Steve: Because "Highest weight" is displayed on the screen to tell the user that the following number is supposed to represent the highest weight seen so far. On the other hand, `HighestWeight` is the name of the variable that holds that information, so including `HighestWeight` in the output statement will result in displaying the highest weight we've seen so far on the screen. Of course, the same analysis applies to the next line, which displays the label "Current weight" and the value of the variable `CurrentWeight`.

The topic of **#include statements** was the cause of some discussion with Susan. Here's the play by play:

Susan: Is the include command the only time you will use the # symbol?

Steve: There are other uses for #, but you won't see any of them for a long time, if ever.

Susan: So `#include` is a command.

Steve: Right; it's a command to the compiler.

Susan: Then what are the words we have been using for the most part called? Are those just called *code* or just

statements? Can you make a list of commands to review?

Steve: The words that are defined in the language, such as `if`, `while`, `for`, and the like are called keywords. User defined names such as function and variable names are called identifiers.

Susan: So `iostream.h` is a header file telling the compiler that it is using info from the `iostreams` library?

Steve: Essentially correct; to be more precise, when we include `iostream.h`, we're telling the compiler to look into `iostream.h` for definitions that we're going to use.

Susan: Then the header file contains the secondary code of machine language to transform `cin` and `cout` into something workable?

Steve: Close, but not quite right. The machine code that makes `cin` and `cout` do their thing is in the `iostreams` library; the header file gives the compiler the information it needs to compile your references to `cout`, `cin`, `<<`, and `>>` into references to the machine code in the library.

Susan: So the header file directs the compiler to that section in the library where that machine code is stored? In other words, it is like telling the compiler to look in section XXX to find the machine code?

Steve: The header file tells the compiler what a particular part of the library does, while the library contains the machine code that actually does it.

If you're a programmer in some other language than C, you may wonder why we have to tell the compiler that we want to use the standard I/O library. Why doesn't the compiler know to use that library automatically? This seeming oversight is actually the result of a decision made very early in the evolution of C: to keep the language itself (and therefore the compiler) as simple as possible, adding functionality with the aid of standard libraries. Since a large part of the libraries can be written in C, this decision reduces the amount of work needed to "port" the C language from one machine architecture or operating system to another. Once the compiler has been ported, it's not too difficult to get the libraries to work on the new machine. In fact, even the C (or C++) compiler can be written in C (or C++), which makes the whole language quite portable. This may seem impossible. How do you get started? In fact, the process is called *bootstrapping*, from the impossible task of trying to lift yourself by your own bootstraps.³⁵ The secret is to have one compiler that's already running; then you use that compiler to compile the compiler for the new machine. Once you have the new compiler running, it is common to use it to compile itself, so that you know it's working. After all, a compiler is a fairly complex program, so getting it to compile and execute properly is a pretty good indication that it's producing the right code.

Most of the rest of the program should be fairly easy to understand, except for the two lines `int main()` and `return 0;`, which have related functions. Let's start with the line `int main()`. As we've already seen, the purpose of the `main()` part of this line is to tell the compiler where to start execution; the C++ language definition specifies that execution always starts at a block called `main`. This may seem redundant, as you might expect the compiler to assume that we want to start execution at the beginning of the program. However, C++ is intended to be useful in the writing of very large programs; such programs can and usually do consist of several implementation files, each of which contains some of the functionality of the program. Without such a rule, the compiler wouldn't know which module should be executed first.

The `int` part of this same line specifies the type of the *exit code* that will be returned from the program by a `return` statement when the program is finished executing; in this case, that type is `int`. The exit code can be used by a *batch file* to determine whether our program finished executing correctly; an exit code of 0, by convention, means that it did.³⁶ The final statement in the program is `return 0;`. This is the `return` statement just mentioned, whose purpose is to return an exit code of 0 when our program stops running. The value that is returned, 0, is an acceptable value of the type we declared in the line `int main()`, namely, `int`; if it didn't match, the compiler would tell us we had made an error.

Finally, the closing curly brace, `}`, tells the compiler that it can stop compiling the current block, which in this case is the one called `main`. Without this marker, the compiler would tell us that we have a missing `}`, which of course would be true.

Novice Alert

Susan decided a little later in our collaboration that she wanted to try to reproduce this program just by considering the English

description, without looking at my solution. She didn't quite make it without peeking, but the results are illuminating nevertheless.

Susan: What I did was to cover your code with a sheet of paper and just tried to get the next line without looking, and then if I was totally stumped then I would look. Anyway, when I saw that `if` statement then I knew what the next statement would be but I am still having problems with writing backwards. For example

```
if (CurrentWeight > HighestWeight)
```

```
HighestWeight = CurrentWeight;
```

That is just so confusing because we just want to say that if the current weight is higher than the highest weight, then the current weight will be the new highest weight, so I want to write `CurrentWeight = HighestWeight`. Anyway, when I really think about it I know it makes sense to do it the right way; I'm just having a hard time thinking like that. Any suggestions on how to think backward?

Steve: What that statement means is "*set HighestWeight to the current value of CurrentWeight*". The point here is that `=` does *not* mean "is equal to"; it means "set the variable to the left of the `=` to the value of the expression to the right of the `=`". It's a lousy way of saying that, but that's what it means.

Susan: With all the `{` and `}` all over the place, I was not sure where and when the `return 0;` came in. So is it always right before the last `}`? OK, now that I think about it, I guess it always would be.

Steve: You have to put the return statement at a place where the program is finished whatever it was doing. That's because whenever that statement is executed, the program is going to stop running. Usually, as in this case, you want to do that at the physical end of the program.

Susan: Anyway, then maybe I am doing something wrong, and I am tired, but after I compiled the program and ran it, I saw that the `HighestWeight` label was run in together with the highest number and the next sentence, which said "Please enter the next weight". All those things were on the same line and I thought that looked weird; I tried to fix it but the best I had the stamina for at the moment was to put a space between the `"` and the `P`, to at least make a separation.

Steve: It sounds as though you need some `endl`s in there to separate the lines.

Take It for a Spin

Assuming that you've installed the software from the CD-ROM in the back of this book, you can try out this program by following these steps:

1. Change to the "code" subdirectory of the directory where you copied the example programs. If you've been following the directions as written, this will be "c:\whos\code".
2. Start the Integrated Development Environment (IDE) for the compiler by typing its name, RHIDE, followed by the name of the program to be compiled. For example, to compile "pump1", type

```
RHIDE pump1
```

3. Then select "Make" from the "Compile" menu. The compiled version will be placed in the "\whos\code" directory.
4. Once you are done compiling, you can exit to DOS by selecting the File menu and clicking on Exit. Then you can execute the program by typing the name of the program you want to run at the DOS prompt, first making sure that you are in the directory where you copied the example programs. For example, to run "pump1", exit to DOS, make sure you are in "\whos\code" and type:

```
pump1
```

5. To run your program under the debugger, make sure you are in the "\whos\code" directory, and then type "RHIDE

pump1". Again, do *not* add the ".cc" to the end of the file name. Once RHIIDE has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger. The debugger also has a lot of other features, including displaying the values of variables during program execution, which I encourage you to explore.

By the way, if you're confused about the seemingly meaningless values that the debugger shows for variables before the first statement that sets each one to a value, let me assure you that they are indeed meaningless. I'll explain why that is in the next chapter.

We're almost done with this chapter, but first let's practice a little more with `chars` and `strings`.

Exercises, Fourth Set

8. Here are four possible versions of an output statement. Assuming that the value of the `string` variable called `name` is "Joe Smith", what does each one of them do?

```
cout << "That is very old, " << name << ". " << endl;
```

```
cout << "That is very old, " << name << '. ' << endl;
```

```
cout << "That is very old, " << name << "." << endl;
```

```
cout << "That is very old, " << name << '.' << endl;
```

Now it's time for some review on what we've covered in this chapter.

Review

We started out by discussing the tremendous reliability of computers; whenever you hear "it's the computer's fault", the overwhelming likelihood is that in fact the software is to blame rather than the hardware. Then we took a look at the fact that, although computers are calculating engines, many of the functions for which we use them don't have much to do with numeric calculations; for example, the most common use of computers is probably word processing, which doesn't use much in the way of addition or subtraction. Nevertheless, we started out our investigation of programming with numeric variables, which are easier to understand than non-numeric ones. To use variables, we need to write a C++ program, which consists primarily of a list of operations to be performed by the computer, along with directions that influence how these operations are to be translated into machine instructions.

That led us into a discussion of why and how our C++ program is translated into machine instructions by a *compiler*. We examined an example program that contained simple *source code statements*, including some that define variables and others that use those variables and constants to calculate results. We covered the symbols that are used to represent the operations of addition, subtraction, multiplication, division, and *assignment*, which are +, -, *, /, and = respectively. While the first four of these should be familiar to you, the last one is a programming notion rather than a mathematical one. This may be confusing because the operation of assignment is expressed by the = sign, but is *not* the same as mathematical equality. For example, the statement `x = 3;` does *not* mean "x is equal to 3", but rather "set the variable x to the value 3. After this discussion of the structure of statements in C++, we started an exploration of how the CPU actually stores and manipulates data in memory. The topics covered in this section included the order in which multibyte data items are stored in memory and the use of *general registers* to manipulate data efficiently.

Then we spent some time pretending to be a compiler, to see how a simple C++ program looks from that point of view, in order to improve our understanding of what the compiler does with our programs. This exercise involved keeping track of the addresses of variables and instructions and watching the effect of the instructions on the general registers and memory locations. During this exploration of the machine, we got acquainted with the *machine language* representation of instructions, which is the actual form that our executable programs take in memory. After a thorough examination of what the compiler does with our source code at *compile time*, we followed what would happen to the registers and memory locations at *run time* (that is, if the sample program were actually executed).

Then we began to look at two data types that can hold nonnumeric data, namely the `char` and the `string`. The `char` corresponds to 1 byte of storage, and therefore can hold one character of data. Examples of appropriate values for a `char` variable include letters (a-z, A-Z), digits (0-9), and special characters (e.g., , . ! @ # \$ %). A `char` can also represent a number of other "nonprintable" characters such as the "space", which causes output to move to the next character position on the screen. Actually, a `char` can also be used as a "really short" numeric variable, but that's mostly a holdover from the days when memory was a lot more expensive, and every byte counted.

One `char` isn't much information, so we often want to deal with groups of them as a single unit; an example would be a person's name. This is the province of the `string` variable type: variables of this type can handle an indefinitely long group of chars.

At the beginning of our sample program for strings and chars, we encountered a new construct, the *#include statement*. This tells the compiler where to find instructions on how to handle data types such as strings, about which it doesn't have any built-in knowledge. Then we came across the line `int main()`, which indicates where we want to start executing our program. A C++ program always starts execution at the place indicated by such a line. We also investigated the meaning of `int`, which is the *return type* of `main`. The return type tells the compiler what sort of data this program returns to the operating system when it finishes executing; the return value can be used to determine what action a batch file should take next.

As we continued looking at the sample program for strings and chars, we saw how to assign literal values to both of these types, and noticed that two different types of quotes are used to mark off the literal values: the single quote ('), which is used in pairs to surround a literal `char` value consisting of exactly one `char`, such as 'a'; and the double quote ("), which is used in pairs to surround a literal string value of the *C string* type, such as "This is a test". We also investigated the reason for these two different types of literal values, which involves the notion of a *null byte* (a byte with the value 0); this null byte is used to mark the end of a C string in memory.

This led us to the discussion of the *ASCII code*, which is used to represent characters by binary values. We also looked at the fact that the same bytes can represent either a numeric value or a C string, depending on how we use those bytes in our program. That's why it's so important to tell the compiler which of these possibilities we have in mind when we write our programs. The way in which the compiler regulates our access to variables by their type, which is defined at compile time, is called the *type system*; the fact that C++ uses this *static type checking* is one of the reasons that C++ programs can be made more robust than programs written in languages that use *dynamic type checking*, where these errors are not detected until run time.

After a short discussion of some of the special characters that have a predefined meaning to the compiler, we took an initial glance at the mechanisms that allow us to get information into and out of the computer, known as *I/O*. We looked at the `<<` function, which provides display on the screen when coupled with the built-in destination called `cout`. Immediately afterwards, we encountered the corresponding input function `>>` and its partner `cin`, which team up to give us input from the keyboard.

Next, we went over some program organization concepts, including the `if` statement, which allows the program to choose between two alternatives; the `while` statement, which causes another statement to be executed while some condition is true; and the *block*, which allows several statements to be grouped together into one logical statement. Blocks are commonly used to enable several statements to be controlled by an `if` or `while` statement.

At last we were ready to write a simple program that does something resembling useful work, and we did just that. The starting point for this program, as with all programs, was to define exactly what the program should do; in this case, the task was to keep track of the pumpkin with the highest weight at a county fair. The next step was to define a solution to this problem in precise terms. Next, we broke the solution down into steps small enough to be translated directly into C++. Of course, the next step after that was to do that translation. Finally, we went over the C++ code, line by line, to see what each line of the program did.

Now that the review is out of the way, we're about ready to continue with some more C++. First, though, let's step back a bit and

see where we are right now.

Conclusion

We've come a long way from the beginning of this chapter. Starting from basic information on how the hardware works, we've made it through our first actual, runnable program. By now, you should have a much better idea whether you're going to enjoy programming (and this book). Assuming you aren't discouraged on either of these points, let's proceed to gather some more tools, so we can undertake a bigger project.

Answers to Exercises

1. 3c43. In case you got a different result, here's a little help:
 - a. If you got the result 433a, you started at the wrong address.
 - b. If you got the result 433c, you had the bytes reversed.
 - c. Finally, if you got 3a43, you made both of these mistakes.

If you made one or more of these mistakes, don't feel too bad; even experienced programmers have trouble with hexadecimal values once in awhile. That's one reason we use compilers and assemblers rather than writing everything in hex!

2. "HELLO". If you couldn't figure out what the "D" at the beginning was for, you started at the wrong place.
3. Figure [first.dinner](#) is Susan's answer to this problem.

First dinner party program (code/basic05.cc) (Figure first.dinner)

[code/basic05.cc](#)

By the way, the reason that this program uses two lines to produce the sentence "Please type in the number of guests of your dinner party." is so that the program listing will fit on the page properly. If you prefer, you can combine those into one line that says `cout << "Please type in the number of guests of your dinner party. ";` Of course, this also applies to the next exercise.

Susan: I would have sent it sooner had I not had the last `cout` arrows going like this `>>` (details). Also, it just didn't like the use of `endl;` at the end of the last `cout` statement. It just kept saying "parse error".

Steve: If you wrote something like

```
cout << "A table for " << n+1 << "is ready. " << "endl;"
```

then it wouldn't work for two reasons. First, "endl;" is just a character string, not anything recognized by `<<`. Second, you're missing a closing `;`, because characters inside quotes are treated as just plain characters by the compiler, not as having any effect on program structure.

The correct way to use `endl` in your second output statement is as follows:

```
cout << "A table for " << n+1 << "is ready. " << endl;
```

By the way, you might want to add a " " in front of the `is` in `is ready`, so that the number doesn't run up against the `is`. That would make the line look like this:

```
cout << "A table for " << n+1 << " is ready. " << "endl;"
```

Susan: Okay.

4. Figure [second.dinner](#) is Susan's answer to this problem, followed by our discussion.

Second dinner party program (code/basic06.cc) (Figure second.dinner)

[code/basic06.cc](#)

Steve: Congratulations on getting your program to work!

Susan: Now, let me ask you this: can you ever modify `else`? That is, could I have written `else (n>20)`, or does `else` always stand alone?

Steve: You can say something like Figure [else.if](#).

else if example (Figure else.if)

```
if (x < y)
{
    cout << "x is less than y" << endl;
}
else
{
    if (x > y)
        cout << "x is greater than y" << endl;
    else
        cout << "x must be equal to y!" << endl;
}
}
```

In other words, the controlled block of an `if` statement or an `else` statement can have another `if` or `else` inside it. In fact, you can have as many "nested" `if` or `else` statements as you wish; however, it's best to avoid very deep nesting because it tends to confuse the next programmer who has to read the program.

5. The answer to this problem should look like Figure [name.age](#).

Name and age program (code/basic07.cc) (Figure name.age)

[code/basic07.cc](#)

One point that might be a bit puzzling in this program is why it's not necessary to add an `<< endl` to the end of the lines that send data to `cout` before we ask the user for input. For example, in the sequence:

```
cout << "What is your name? ";
```

```
cin >> name;
```

how do we know that the C string "What is your name? " has been displayed on the terminal before the user has to type in the answer? Obviously, it would be hard for the user to answer our request for information without a clue as to what we're asking for.

As it happens, this is a common enough situation that the designers of the `iostreams` library have anticipated it and solved it for us. When we use that library to do output to the screen and input from the keyboard, we can be sure that any screen output we have already requested will be displayed before any input is requested from the user via the keyboard.

6. Figure [novice](#) shows Susan's program, which is followed by our discussion.

Novice program (code/basic08.cc) (Figure novice)

[code/basic08.cc](#)

Susan: Steve, look at this. It even runs!

Also, I wanted to ask you one more question about this program. I wanted to put double quotes around the words `true` and `false` in the 3rd output statement because I wanted to emphasize those words, but I didn't know if the compiler could deal with that so I left it out. Would that have worked if I had?

Steve: Not if you just added quotes, because `"` is a special character that means "beginning or end of C string". Here's what you would have to do to make it work:

```
cout << "Please answer with either \"true\" or \"false\".";
```

The `\` is a way of telling the compiler to treat the next character differently from its normal usage. In this case, we are telling the compiler to treat the special character `"` as "not special"; that is, `\"` means "just the character double quote, please, and no nonsense". This is called an *escape*, because it allows you to get out of the trap of having a `"` mean something special. We also use the `\` to tell the compiler to treat a "nonspecial" character as "special"; for example, we use it to make up special characters that don't have any visual representation. You've already seen `\n`, the "newline" character, which means "start a new line on the screen".

Susan: So if we want to write some character that means something "special", then we have to use a `\` in front of it to tell the compiler to treat it like a "regular" character?

Steve: Right.

Susan: And if we want to write some character that is "regular" and make it do something "special", then we have to use a `\` in front of it to tell the compiler that it means something "special"? That's weird.

Steve: It may be weird, but that's the way it works.

Susan: I now just got it. I was going to say, why would you put the first quotation mark before the slash, but now I see. Since you are doing a endline character, you have to have quotes on both sides to surround it which you don't usually have to do because the first quotes are usually started at the beginning of the sentence, and in this case the quote was already ended. Ok, thanks for clearing that up.

Steve: You've got it.

Susan: Another thing I forgot is how you refer to the statements in `()` next to the "if" keywords; what do you call the info that is in there?

Steve: The condition.

7. Figure [allowance](#) is Susan's version of this program. Actually, it was her idea in the first place.

Allowance program (code/basic09.cc) (Figure allowance)

[code/basic09.cc](#)

8. You'll be happy (or at least unsurprised) to hear that Susan and I had quite a discussion about this problem.

Susan: Remember on my "test" program how I finally got that period in there? Then I got to thinking that maybe it should have been surrounded by single quotes ' instead of double quotes. It worked with a double quote but since it was only one character it should have been a single quote, so I went back and changed it to a single quote and the compiler *didn't like that at all*. So I put it back to the double. So what is the deal?

Steve: You should be able to use 'x' or "x" more or less interchangeably with <<, because it can handle both of those data types (char and C string, respectively). However, they are indeed different types. The first one specifies a literal char value, whereas the second specifies a literal C string value. A char value can only contain one character, but a C string can be as long as you want, from none to hundreds or thousands of characters.

Susan: Here's the line that gave me the trouble:

```
cout << "That is very old, " << name << ". " << endl;
```

Remember I wanted to put that period in at the end in that last line? It runs like this but not with the single quotes around it. That I don't understand. This should have been an error. But I did something right by mistake <G>. Anyway, is there something special about the way a period is handled?

Steve: I understand your problem now. No, it's not the period; it's the space after the period. Here are four possible versions of that line:

```
1. cout << "That is very old, " << name << ". " << endl;
```

```
2. cout << "That is very old, " << name << '. ' << endl;
```

```
3. cout << "That is very old, " << name << "." << endl;
```

```
4. cout << "That is very old, " << name << '.' << endl;
```

None of these is exactly the same as any of the others. However, 1, 3, and 4 will do what you expect, whereas 2 will produce weird looking output, with some bizarre number where the . should be. Why is this? It's not because . is handled specially, but because the space (" "), when inside quotes, either single or double, is a character like any other character. Thus, the expression '. ' in line 2 is a "multicharacter constant", which has a value dependent on the compiler; in this case, you'll get a short value equal to (256 * the ASCII value of the period) + the ASCII value of the space. This comes out to 11808, as I calculate it. So the line you see on the screen may look like this:

```
That is very old, Joe Smith11808
```

Now why do all of the other lines work? Well, 1 works because a C string can have any number of characters and be sent to cout correctly; 3 works for the same reason; and 4 works because '.' is a valid one-character constant, which is another type that << can handle.

I realize it's hard to think of the space as a character, when it doesn't look like anything; in addition, you can add spaces freely between variables, expressions, and so forth, in the program text. However, once you're dealing with C strings and literal character values, the space is just like any other character.

Susan: So it is okay to use single characters in double quotes? If so, why bother with single quotes?

Steve: Single quotes surround a literal of type `char`. This is a 1-byte value that can be thought of (and even used) as a very short number. Double quotes surround a literal of type "C string". This is a multibyte value terminated by a 0 byte, which cannot be used or treated as a number.

Susan: I am not too clear on what exactly the difference is between the `char` and "C string". I thought a `char` was like a alpha letter, and a string was just a bunch of letters.

Steve: Right. The difference is that a C string is variable length, and a `char` isn't; this makes a lot of difference in how they can be manipulated.

Susan: Am I right in thinking that a `char` could also be a small number that is not being used for calculations?

Steve: Or that is used for (very small) calculations; for instance, if you add 1 to the value 'A', you get the value for 'B'. At least that's logical.

Susan: What do you mean by "terminated by a 0 byte"? That sounds familiar; was that something from an earlier chapter which is now ancient history?

Steve: Yes, we covered that some time ago. The way the program can tell that it's at the end of a C string (which is of variable length, remember) is that it gets to a byte with the value 0. This is a crummy way to specify the size of a variable-length string, in my opinion, but it's too late to do anything about it; it's built into the compiler.

Susan: When you say a C string, do you mean the C programming language in contrast to other languages?

Steve: Yes.

Susan: All right, then the 0 byte used to terminate a C string is the same thing as a null byte?

Steve: Yes.

Susan: Then you mean that each C string must end in a 0 so that the compiler will know when to stop processing the data for the string?

Steve: Yes.

Susan: Could you also just put 0? Hey, it doesn't hurt to ask. I don't see the problem with the word *hello*; it ends with an o and not a 0. But what if you do need to end the sentence with a 0?

Steve: It's not the digit 0, which has the ASCII code 30h, but a byte with a 0 value. You can't type in a null byte directly, although you can create one with a special character sequence if you want to. However, there's no point in doing that usually, because all literal C strings such as "hello" always have an invisible 0 byte added automatically by the compiler. If for some reason you need to explicitly create a null byte, you can write it as `'\0'`, as in

```
char x = '\0';
```

which emphasizes that you really mean a null byte and not just a plain old 0 like this:

```
char x = 0;
```

The difference between these two is solely for the benefit of the next programmer to look at your code; they're exactly the same to the compiler.

Footnotes

1. Please note that capitalization counts in C++, so `IF` and `WHILE` are not the same as `if` and `while`. You have to use the latter versions.
2. However, we haven't yet eliminated the possibility of hardware errors, as the floating-point flaw in early versions of the Pentium™ processor illustrates. In rare cases, the result of the divide instruction in those processors was accurate to only about 5 decimal places rather than the normal 16 to 17 decimal places.
3. This was apparently against the plan administrator's principles.
4. How is the compiler itself translated into machine language so it can be executed? The most common method is to write the compiler in the same language it compiles and use the previous version of the compiler to compile the newest version! Of course, this looks like an infinite regress; how did the first compiler get compiled? By manual translation into *assembly language*, which was then translated by an *assembler* into machine language. To answer the obvious question, at some point an assembler was coded directly in machine language.
5. The compiler also does a lot of other work for us, which we'll get into later.
6. By the way, blank lines are ignored by the compiler; in fact, because of the trailing semicolon on each statement, you can even run all the statements together on one line if you want to, without confusing the compiler. However, that will make it much harder for someone reading your code later to understand what you're trying to do. Programs aren't written just for the compiler's benefit but to be read by other people; therefore, it is important to write them so that they can be understood by those other people. One very good reason for this is that more often than you might think, those "other people" turn out to be *you*, six months later.
7. The `//` marks the beginning of a *comment*, which is a note to you or another programmer; it is ignored by the compiler. For those of you with BASIC experience, this is just like `REM` (the "remark" keyword in that language); anything after it on a line is ignored.
8. Other kinds of variables can hold larger (and smaller) values; we'll go over them in some detail in future chapters.
9. At the risk of boring experienced C programmers, let me reiterate that `=` *does not mean* "is equal to"; it means "set the variable to the left of the `=` to the value of the expression to the right of the `=`". In fact, there is *no* equivalent in C++ to the mathematical notion of equality. We have only the assignment operator `=` and the comparison operator `==`, which we will encounter in the next chapter. The latter is used in `if` statements to determine whether two expressions have the same value. All of the valid comparison operators are listed in Figure [comparisonfig](#).
10. If you have any programming experience whatever, you may think that I'm spending too much effort on this very simple point. I can report from personal experience that it's not necessarily easy for a complete novice to grasp. Furthermore, without a solid understanding of the difference between an algebraic equality and an assignment statement, that novice will be unable to understand how to write a program.
11. Besides these general registers, a dedicated register called `esp` plays an important role in the execution of real programs. We'll see how it does this in Chapter [function.htm](#).
12. This is not the only possible solution to this problem nor necessarily the best one; for example, in many Motorola CPUs, you specify the length of the variable directly in the instruction, so loading a *word* (i.e., 2-byte) variable might be specified by the instruction `move .w`, where the `.w` means "word". Similarly, a *longword* (i.e., 4-byte) load might be specified as `move .l`, where the `.l` means "long word".
13. It's also possible to load a 2-byte value into a 32-bit (i.e., 4-byte) register such as `eax` and have the high part of that register set to 0 in one instruction, by using an instruction designed specifically for that purpose. This approach has the advantage that further processing can be done with the 32-bit registers.
14. The number inside the brackets `[]` represents a memory address.
15. As I've mentioned previously, blank lines are ignored by the compiler; you can put them in freely to improve readability.
16. However, I've cheated here by using small enough numbers in the C++ program that they are the same in hex as in decimal.
17. The real compiler on the CD-ROM actually uses 4-byte addresses, but this doesn't change any of the concepts involved.
18. These addresses are arbitrary; a real compiler will assign addresses to variables and machine instructions by its own rules.
19. In case you were wondering, the most common pronunciation of `char` has an *a* like the *a* in "married", while the *ch* sounds like "k".
20. As we will see shortly, not all characters have visible representations; some of these "nonprintable" characters are useful in controlling how our printed or displayed information looks.
21. Other compilers sometimes use other extensions for implementation files, such as `.cpp`, and for header files, such as `.hpp`.
22. Please note that there is a *space* (blank) character at the end of that C string literal, after the word "test". That space is part of the literal value.
23. Warning: Every character inside the quotes has an effect on the value of the literal, whether the quotes are single or

double; even "invisible" characters such as the *space* (' ') will change the literal's value. In other words, the line `c1 = 'A' ;` is *not* the same as the line `c1 = 'A ' ;`. The latter statement may or may not be legal, depending on the compiler you're using, but it is virtually certain not to give you what you want, which is to set the variable `c1` to the value equivalent to the character 'A'. Instead, `c1` will have some weird value resulting from combining the 'A' and the space character. In the case of a `string` value contained in double quotes, multiple characters are allowed, so "A B" and "AB" both make sense, but the space still makes a difference; namely, it keeps the 'A' and 'B' from being next to one another.

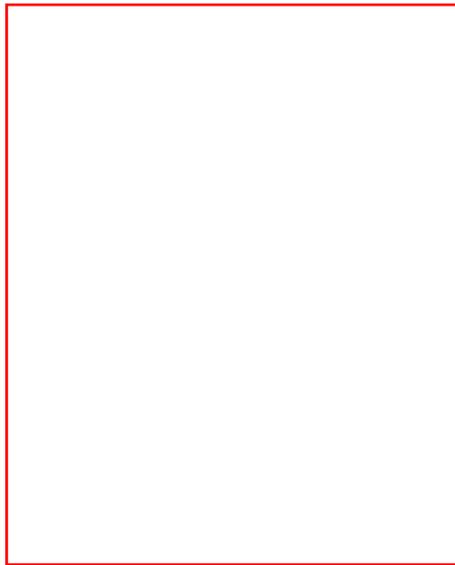
24. I don't want to mislead you about this notion of a byte having the value 0; it is *not* the same as the representation of the decimal digit "0". As we'll see, each displayable character (and a number of invisible ones) is assigned a value to represent it when it's part of a `string` or literal value (i.e., a C string literal or `char` literal). The 0 byte I'm referring to is a byte with the binary value 0.
25. Happily, we can improve on it in most other circumstances, as you'll see later.
26. You may wonder why I have to specify that the codes for each case of letters run consecutively. Believe it or not, there are a number of slightly differing codes collectively called EBCDIC (Extended Binary Coded Decimal Interchange Code), in which this is not true! Eric Raymond's amusing and interesting book, *The New Hacker's Dictionary*, has details on this and many other historical facts.
27. I'll be more specific later, when we have seen some examples.
28. For example, if you wanted to insert a " in a string, you would have to use `\"`, because just a plain " would indicate the end of the string. That is, if you were to set a `string` to the value `"This is a \"string\"."`, it would display as: `This is a "string"`.
29. The line `#include <iostream.h>` is necessary here to tell the compiler about `cout` and how it works. We'll get into this in a bit more detail later in this chapter.
30. By the way, `cout` is pronounced "see out".
31. Similarly to `cout`, `cin` is pronounced "see in" rather than "sin".
32. This explanation assumes that the "10,000" is the balance in dollars. Of course, this doesn't account for the possibility of balances that aren't a whole number of dollars, and there's also the problem of balances greater than \$32,767, which wouldn't fit into a `short`. I'll mention possible solutions to these problems in Appendix .
33. You may be wondering why we need parentheses around the expression `Guess != Secret`. The conditional expression has to be in parentheses so that the compiler can tell where it ends and the statement to be controlled by the `while` begins.
34. If you look at someone else's C++ program, you're likely to see a different style for lining up the `{ }` to indicate where a block begins and ends. As you'll notice, my style puts the `{` and `}` on separate lines rather than running them together with the code they enclose, to make them stand out, and indents them further than the conditional statement. I find this the clearest, but this is a matter where there is no consensus. The compiler doesn't care how you indent your code or whether you do so at all; it's a stylistic issue.
35. If this term sounds familiar, we've already seen it in the context of how we start up a computer when it's turned on, starting from a small *boot program* in the *ROM*, or Read-Only Memory.
36. A batch file is a text file that directs the execution of a number of programs, one after the other, without manual intervention. A similar facility is available in most operating systems.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

More Basics

A Modest Proposal

Now that we have seen how to write a simple program in C++, it's time to acquire some more tools. We'll

extend our example program from Chapter [basics.htm](#) for finding the heaviest pumpkin. Eventually, we want to provide the weights of the three heaviest pumpkins, so that first, second, and third prizes can be awarded. It might seem that this would require just a minor modification of the previous program, in which we would keep track of the heaviest so far, second heaviest so far, and third heaviest so far, rather than merely the heaviest so far. However, this modification turns out to be a bit more complicated than it seems. Since this book is intended to teach you how to program using C++, rather than just how to use the C++ language, it's worth investigating why this is so. First, though, here are the objectives for this chapter.

Objectives of This Chapter

By the end of this chapter, you should

1. Understand the likelihood of error in even a small change to a program.
2. Be aware that even seemingly small changes in a problem can result in large changes in the program that solves the problem.
3. Have some understanding of the type of thinking needed to solve problems with programming.
4. Understand the selection sorting algorithm for arranging values in order.
5. Understand how to use a `vector` to maintain a number of values under one name.
6. Be able to use the `for` statement to execute program statements a (possibly varying) number of times.
7. Be familiar with the arithmetic operators `++` and `+=`, which are used to modify the value of variables.

Algorithmic Thinking

Let's take our program modification one step at a time, starting with just the top two weights. Figure [pump1afig](#) is one possible way to handle this version of the problem.

Finding the top two weights, first try (code\pump1a.cc) (Figure satisfied)

(Figure pump1afig)

```
#include <iostream.h>

int main()
{
    short CurrentWeight;

    short HighestWeight;

    short SecondHighestWeight;
```

```
cout << "Please enter the first weight: ";

cin >> CurrentWeight;

HighestWeight = CurrentWeight;

SecondHighestWeight = 0;

cout << "Current weight " << CurrentWeight << endl;

cout << "Highest weight " << HighestWeight << endl;

while (CurrentWeight > 0)
{

    cout << "Please enter the next weight: ";

    cin >> CurrentWeight;

    if (CurrentWeight > HighestWeight)
    {

        SecondHighestWeight = HighestWeight;

        HighestWeight = CurrentWeight;

    }

    cout << "Current weight " << CurrentWeight << endl;

    cout << "Highest weight " << HighestWeight << endl;

    cout << "Second highest weight " << SecondHighestWeight << endl;

}

return 0;

}
```

The reasons behind some of the new code should be fairly obvious, but we'll go over them anyway. The new lines are **bold** so you can find them easily. First, of course, we need a new variable, `SecondHighestWeight`, to hold the current value of the second highest weight we've seen so far. Then, when the first weight is entered, the statement `SecondHighestWeight = 0;` sets the `SecondHighestWeight` to 0. After all, there isn't any second-highest weight when we've only seen one weight. The first nonobvious change is the addition of the statement `SecondHighestWeight = HighestWeight;`, which copies the old `HighestWeight` to `SecondHighestWeight`, whenever there's a new highest weight. On reflection, however, this should make sense; when a new high is detected, the old high must be the second highest value (so far). Also, we have to copy the old `HighestWeight` to `SecondHighestWeight` before we change `HighestWeight`. After we have set `HighestWeight` to a new value, it's too late to copy its old value into `SecondHighestWeight`.

First, let's see how Susan viewed this solution:

Susan: I noticed that you separate out the main program `{ }` from the other `{ }` by indenting. Is that how the compiler knows which set of `{ }` goes to which statements and doesn't confuse them with the main ones that are the body of the program?

Steve: The compiler doesn't care about indentation at all; that's just for the people reading the program. All the compiler cares about is the number of `{` it has seen so far without matching `}`. There aren't any hard rules about this; it's a "religious" issue in C++, where different programmers can't agree on the best way.

Susan: Now on this thing with setting `SecondHighestWeight` to 0. Is that initializing it? See, I know what you are doing, and yet I can't see the purpose of doing this clearly, unless it is initializing, and then it makes sense.

Steve: That's correct.

Susan: How do you know how to order your statements? For example, why did you put the `SecondHighestWeight = HighestWeight;` above the other statement? What would happen if you reversed that order?

Steve: Think about it. Let's suppose that:

`CurrentWeight` is 40

`HighestWeight` is 30

`SecondHighestWeight` is 15

and the statements were executed in the following order:

1. `HighestWeight = CurrentWeight`

2. `SecondHighestWeight = HighestWeight`

What would happen to the values? Well, statement 1 would set `HighestWeight` to `CurrentWeight`, so the values would be like this:

`CurrentWeight` is 40

`HighestWeight` is 40

`SecondHighestWeight` is 15

Then statement 2 would set `SecondHighestWeight` to `HighestWeight`, leaving the situation as follows:

`CurrentWeight` is 40

`HighestWeight` is 40

`SecondHighestWeight` is 40

This is clearly wrong. The problem is that we need the value of `HighestWeight` *before* it is set to the value of `CurrentWeight`, not afterward. After that occurs, the previous value is lost.

Susan: Yes, that is apparent; I was just wondering if the computer had to read it in the order that you wrote it, being that it was grouped together in the `{ }`. For example, you said that the compiler doesn't read the `{ }` as we write them, so I was wondering if it read those statements as we write them. Obviously it has to. So then everything descends in a progression downward and outward, as you get more detailed in the instructions.

Assuming that you've installed the software from the CD-ROM in the back of this book, you can try out this program. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE pump1a`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `pump1a` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

To run it under the debugger, make sure you are in the `code` subdirectory, and then type `"RHIDE pump1a"`. Again, do *not* add the `".cc"` to the end of the file name. Once `RHIDE` has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

A Prize Catch

This program may seem to keep track of the highest and second highest weights correctly, but in fact there's a hole in the logic. To be exact, it doesn't work correctly when the user enters a new value that's less than

the previous high value but more than the previous second-high value. In that case, the new value should be the second-high value, even though there's no new high value. For example, suppose that you enter the following weights: 5 2 11 3 7. If we were to update `SecondHighestWeight` only when we see a new high, our program would indicate that 11 was the high, and 5 the second highest; since neither 3 nor 7 is a new high, `SecondHighestWeight` would remain as it was when the 11 was entered.

Here's what ensued when Susan tried out the program and discovered this problem:

Susan: Steve, the program! I have been playing with it. Hey this is fun, but look, it took me awhile. I had to go over it and over it, and then I was having trouble getting it to put current weights that were higher than second weights into the second weight slot. For example, if I had a highest weight of 40 and the the second highest weight of 30 and then selected 35 for a current weight, it wouldn't accept 35 as the second-highest weight. It increased the highest weights just fine and it didn't change anything if I selected a lower number of the two for a current weight. Or did you mean to do that to make a point? I am supposed to find the problem? I bet that is what you are doing.

Steve: Yep, and I'm not sorry, either.<G>

Susan: You just had to do this to me, didn't you? OK, what you need to do is to put in a statement that says if the current weight is greater than the second-highest weight, then set the second-highest weight to the current weight, as illustrated in Figure [satisfied](#).

```
else
{
    if (CurrentWeight > Second HighestWeight)
        Second HighestWeight = CurrentWeight;
}
```

I hope you are satisfied.

Steve: Satisfied? Well, no, I wouldn't use that word. How about ecstatic? You have just figured out a bug in a program, and determined what the solution is. Don't tell me you don't understand how a program works.

Now I have to point out something about your code. I understood what you wrote perfectly. Unfortunately, compilers aren't very smart, and therefore have to be extremely picky. So you have to make sure to spell the variable names correctly. This would make your answer like the `if` clause shown in Figure [if.satisfied](#).

Congratulations again.

As Susan figured out, we have to add an `else` clause to our `if` statement, so that the corrected version of the statement looks like Figure [if.satisfied](#).

Using an `if` statement with an `else` clause (Figure [if.satisfied](#))

```

if (CurrentWeight > HighestWeight)
    {
        SecondHighestWeight = HighestWeight;
        HighestWeight = CurrentWeight;
    }
else
    {
        if (CurrentWeight > SecondHighestWeight)
            SecondHighestWeight = CurrentWeight;
    }

```

In this case, the condition in the first `if` is checking whether `CurrentWeight` is greater than the previous `HighestWeight`; when this is true, we have a new `HighestWeight` and we can update both `HighestWeight` and `SecondHighestWeight`. However, if `CurrentWeight` is not greater than `HighestWeight`, the `else` clause is executed. It contains another `if`; this one checks whether `CurrentWeight` is greater than the old `SecondHighestWeight`. If so, `SecondHighestWeight` is set to the value of `CurrentWeight`.

What happens if two (or more) pumpkins are tied for the highest weight? In that case, the first one of them to be encountered is going to set `HighestWeight`, as it will be the highest yet encountered. When the second pumpkin of the same weight is seen, it won't trigger a change to `HighestWeight`, since it's not higher than the current occupant of that variable. It will pass the test in the `else` clause, `if (CurrentWeight > SecondHighestWeight)`, however, which will cause `SecondHighestWeight` to be set to the same value as `HighestWeight`. This is reasonable behavior, unlikely to startle the (hypothetical) user of the program, and therefore is good enough for our purposes. In a real application program, we'd have to try to determine what the user of this program would want us to do.

Figure [pump2](#) shows the corrected program.

Finding the top two weights (code\pump2.cc) (Figure [pump2](#))

[code/pump2.cc](#)

Assuming that you've installed the software from the CD-ROM in the back of this book, you can try out this program. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE pump1a`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `pump1a` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

To run it under the debugger, make sure you are in the `code` subdirectory, and then type "RHIDE pump1a". Again, do *not* add the ".cc" to the end of the file name. Once RHIDE has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

When you are asked for a weight, type one in and hit ENTER just as when executing normally. When you enter a 0 weight, the program will stop looping and execution will take the path to the end }.

By the way, since we've just been using the `if` statement pretty heavily, this would be a good time to list all of the conditions that it can test. We've already seen some of them, but it can't hurt to have them all in one place. Figure [comparisonfig](#) lists these conditions, with translations.

What if? (Figure comparisonfig)

Condition	Controlled block will be executed if:
Symbol	

>	First item is larger than second item
<	First item is smaller than second item
>=	First item is larger than or equal to second item
<=	First item is smaller than or equal to second item
!=	First item differs from second item

`==` First item has the same value as the second item

You may wonder why we have to use `==` to test for equality rather than just `=`. That's because `=` means "assign right hand value to variable on left", rather than "compare two items for equality". This is a "feature" of C++ (and C) that allows us to accidentally write `if (a = b)` when we mean `if (a == b)`. What does `if (a = b)` mean? It means the following:

1. Assign the value of `b` to `a`.
2. If that value is 0, then the `if` is false.
3. Otherwise, the `if` is true.

Some people find this useful; I don't. Therefore, I always enable the compiler warning that tells you when you use `a =` inside an `if` statement in a way that looks like you meant to test for equality.

What a Tangled Web We Weave. . .

I hope this excursion has given you some appreciation of the subtleties that await in even the simplest change to a working program; many experienced programmers still underestimate such difficulties and the amount of time that may be needed to ensure that the changes are correct. I don't think it's necessary to continue along the same path with a program that can award three prizes. The principle is the same, although the complexity of the code grows with the number of special cases we have to handle. Obviously, a solution that could handle any number of prizes without special cases would be a big improvement, but it will require some major changes in the organization of the program. That's what we'll take up next.

You May Already Have Won

One of the primary advantages of the method we've used so far to find the heaviest pumpkin(s) is that we didn't have to save the weights of all the pumpkins as we went along. If we don't mind saving all the weights, then we can solve the "three prize" problem in a different way. Let's assume for the purpose of simplicity that there are only five weights to be saved, in which case the solution looks like this:

1. Read in all of the weights.
2. Make a list consisting of the three highest weights in descending order.
3. Award the first, second, and third prizes, in that order, to the three entries in the list of highest weights.

Now let's break those down into substeps which can be more easily translated into C++:

1. Read in all of the weights.

1. Read first number
2. Read next number
3. If we haven't read five weights yet, go back to 1b

Now we have all the numbers; proceed to calculation phase:

1. Make a list consisting of the three highest weights in descending order.
 1. Find the largest number in the original list of weights
 2. Copy it to the sorted list
 3. If we haven't found the three highest numbers, go back to 2a

Oops. That's not going to work, since we'll get the same number each time.¹

To prevent that from happening, we have to mark off each number as we select it. Here's the revised version of step 2:

1. Make a list consisting of the three highest weights in descending order.
 1. Find the largest number in the original list of weights
 2. Copy it to the sorted list
 3. Mark it off in the original list of weights, so we don't select it again
 4. If we haven't found the three highest numbers, go back to 2a

Now we're ready for output:

1. Award the first, second, and third prizes, in that order, to the three entries in the list of highest weights.
 1. Write first number
 2. Write another number
 3. If we haven't done them all, go back to 3b

Unlike our previous approach, this obviously can be generalized to handle any number of prizes. However, we have to address two problems before we can use this approach: First, how do we keep track of the weights? And, second, how do we select out the highest three weights? Both of these problems are much easier to solve if we don't have a separate variable for each weight.

Variables, by the Numbers

The solution to our first question is to use a *vector*.² This is a variable containing a number of "sub-variables" that can be addressed by position in the *vector*; each of these sub-variables is called an **element**. A *vector* has a name, just like a regular variable, but the elements do not. Instead, each element has a number, corresponding to its position in the *vector*. For example, we might want to create a *vector* of `short` values called `Weight`, with five elements. To do this, we would write this line: `vector<short> Weight(5);`³ We haven't heard from Susan for awhile, but the following exchange should make up for that.

Susan: OK, why do we need another header (`#include "vector.h"`)?⁴

Steve: Each header contains definitions for a specific purpose. For example, `iostream.h` contains definitions that allow us to get information in (I) and out (O) of the computer. On the other hand, `vector.h` contains definitions that allow us to use `vectors`.

Susan: So then using a `vector` is just another way of writing this same program, only making it a little more efficient?

Steve: In this case, the new program can do more than the old program could: The new program can easily be changed to handle virtually any number of prizes, whereas the old program couldn't.

Susan: So there is more than one way to write a program that does basically the same thing?

Steve: As many ways as there are to write a book about the same topic.

Susan: I find this to be very odd. I mean, on one hand the code seems to be so unrelentingly exact; on the other, it can be done in as many ways as there are artists to paint the same flower. That must be where the creativity comes in. Then I would expect that the programs should behave in different manners, yet accomplish the same goal.

Steve: It's possible for two programs to produce similar (or even exactly the same) results from the user's perspective and yet work very differently internally. For example, the vectorized version of the weighing program could produce exactly the same final results as the original version, even though the method of finding the top two weights was quite different.

Now we can refer to the individual elements of the `vector` called `Weight` by using their numbers, enclosed in **square brackets** (`[]`); the number in the brackets is called the **index**.⁵ Here are some examples:

```
Weight[1] = 123;
```

```
Weight[2] = 456;
```

```
Weight[3] = Weight[1] + Weight[2];
```

```
Weight[i+1] = Weight[i] + 5;
```

As these examples indicate, an element of a `vector` can be used anywhere a "regular" variable can be used.⁶ But the most valuable difference between a regular variable and an element of a `vector` is that we can vary which element we are referring to in a given statement, by varying its index. Take a look at the last sample line, in which two elements of the `vector` `Weight` are used; the first one is element `i+1` and the other is element `i`. As this indicates, we don't have to use a constant value for the element number but can calculate it while the program is executing; in this case, if `i` is 0, the two elements referred to are element 1 and element 0, while if `i` is 5, the two elements are elements 6 and 5, respectively.

The ability to refer to an element of a `vector` by number rather than by name allows us to write

statements that can refer to any element in a `vector`, depending on the value of the index variable in the statements. To see how this works in practice, let's look at Figure [vect1](#), which solves our three-prize problem.

Using a `vector` (`code\vect1.cc`) (Figure `vect1`)

[code/vect1.cc](#)

Assuming that you've installed the software from the CD-ROM in the back of this book, you can try out this program. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE vect1`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `vect1` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

To run it under the debugger, make sure you are in the `code` subdirectory, and then type `"RHIDE vect1"`. Again, do *not* add the `".cc"` to the end of the file name. Once `RHIDE` has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

When you are asked for a weight, type one in and hit ENTER just as when executing normally. After you've entered 5 weights, the program will start the sorting process. When the sorted results have been displayed (or when you're tired of tracing the program), type `q` (for *quit*) and hit ENTER to exit from the debugger.

This program uses several new features of C++ which need some explanation. First, of course, there is the line that defines the `vector` `Weight`:

```
vector<short> Weight(5);
```

As you might have guessed, this means that we want a `vector` of five elements, each of which is a `short`. As we have already seen, this means that there are five distinct index values each of which refers to one element. However, what isn't so obvious is what those five distinct index values actually are. You might expect them to be 1, 2, 3, 4 and 5; actually, they are 0, 1, 2, 3, and 4.

This method of referring to elements in a `vector` is called **zero-based indexing**.⁷ Although it might seem arbitrary to start counting at 0 rather than at 1, assembly language programmers find it perfectly natural, because the calculation of the address of an element is simpler with such indexing; the formula is "(address of first element) + (element number) * (size of element)".

This bit of history is relevant because C, the predecessor of C++, was originally intended to replace assembly language so that programs could be moved from one machine architecture to another with as little difficulty as possible. One reason for some of the eccentricities of C++ is that it has to be able to replace C as a "portable assembly language" that doesn't depend on any specific machine architecture. This explains, for example, the great concern of the inventor of C++ for run-time efficiency, as he wished to allow programmers to avoid the use of C or assembly language for efficiency.⁸ Since C++ was intended to replace C completely, it has to be as efficient as possible; otherwise, programmers might switch back from

C++ to C whenever they were concerned about the speed and size of their programs.

The last two lines in the variable definition phase define two variables, called `i` and `k`, which have been traditional names for **index variables** (i.e., variables used to hold indexes) since at least the invention of FORTRAN in the 1950s. The inventors of FORTRAN used a fairly simple method of determining the type of a variable: if it began with one of the letters I through N, it was an integer. Otherwise, it was a **floating-point variable** (i.e., one that can hold values that contain a fractional part, such as 3.876). This rule was later changed so that the user could specify what type the variable was, as we do in C++, but the default rules were the same as in the earlier versions of FORTRAN, to allow programs using the old rules to continue to compile and run correctly.

Needless to say, Susan had some questions about the names of index variables:

Susan: So whenever you see `i` or `k` you know you are dealing with a `vector`?

Steve: Not necessarily. Variables named `i` and `k` are commonly used as indexes, but they are also used for other purposes sometimes.

Susan: Anyway, if `i` and `k` are sometimes used for other purposes, then the compiler doesn't care what you use as indexes? Again, no rules, just customs?

Steve: Right. It's just for the benefit of other programmers, who will see `i` and say "oh, this is probably an index variable".

I suspect one reason for the durability of these short names is that they're easy to type, and many programmers aren't very good typists.⁹ In C++, the letters `i`, `j`, `k`, `m` and `n` are commonly used as indexes; however, `l` (the letter "ell") generally isn't, because it looks too much like a `1` (the numeral one). The compiler doesn't get confused by this resemblance, but programmers very well might.

After the variable definitions are out of the way, we can proceed to the executable portion of our program. First, we type out a note to the user, stating what to expect. Then we get to the code in Figure [using.for](#).

Using a `for` statement (from `code\vect1.cc`) (Figure [using.for](#))

```
for (i = 0; i < 5; i ++)  
  
    {  
  
        cout << "Please type in weight #" << i+1 << ": ";  
  
        cin >> Weight[i];  
  
    }
```

The first line here is called a **for statement**, which is used to control a **for loop**; this is a loop control

facility similar to the `while loop` we encountered in Chapter [basics.htm](#). The difference between these two statements is that a `for` loop allows us to specify more than just the condition under which the **controlled block** will be repetitively executed.¹⁰ A `for` statement specifies three expressions (separated by ";") that control the execution of the `for` loop: a starting expression, a continuation expression, and a modification expression. In our case, these are `i = 0`, `i < 5`, and `i ++`, respectively. Let's look at the function and meaning of each of these components.

First, the **starting expression**, `i = 0`. This is executed once before the block controlled by the `for` statement is executed. In this case, we use it to set our index variable, `i`, to 0, which will refer to the first element of our `Weight vector`.

Next, the **continuation expression**, `i < 5`. This specifies under what conditions the statement controlled by the `for` will be executed; in this case, we will continue executing the controlled statement as long as the value of `i` is less than 5. Be warned that the continuation expression is actually executed *before* every execution of the controlled block; thus, if the continuation expression is `false` when the loop is entered, the controlled block will not be executed at all.

The notion of the continuation expression is apparently confusing to some novices. Susan fell into that group.

Susan: In your definition of `for`, how come there is no ending expression? Why is it only a modification expression? Is there never a case for a conclusion?

Steve: The "continuation expression" tells the compiler when you want to continue the loop; if the continuation expression comes out false, then the loop terminates. That serves the same purpose as an "ending expression" might, but in reverse.

Finally, let's consider the **modification expression**, `i ++`.¹¹ This is exactly equivalent to `i = i + 1`, which means "set `i` to one more than its current value", an operation technically referred to as **incrementing a variable**. You may wonder why we need two ways to say the same thing; actually, there are a few reasons. One is that `++` requires less typing, which as we know isn't a strong point of many programmers; also, the `++` (pronounced "plus plus") operator doesn't allow the possibility of mistyping the statement as, for example, `i = j + 1`; when you really meant to increment `i`. Another reason why this feature was added to the C language is that, in the early days of C, compiler technology wasn't very advanced, and the `++` operator allowed the production of more efficient programs. You see, many machines can add one to a memory location by a single machine language instruction, usually called something like *increment memory*. Even a simple compiler can generate an "increment memory" instruction as a translation of `i ++`, while it takes a bit more sophistication for the compiler to recognize `i = i + 1` as an increment operation. Since incrementing a variable is a very common operation in C++, this was worth handling specially.¹² Now that we have examined all the parts of the `for` statement, we can see that its translation into English would be something like this:

1. Set the index variable `i` to 0.
2. If the value of `i` is less than 5, execute the following block (in this case, the block with the `cout` and `cin` statements). Otherwise, skip to the next statement after the end of the controlled block; that is, the one following the closing `}`.
3. Add one to the value of `i` and go back to step 2.

Susan didn't think these steps were very clear. Let's listen in on the conversation that ensued:

Susan: Where in the `for` statement does it say to skip to the next statement after the end of the controlled block when `i` is 5 or more?

Steve: It doesn't have to. Remember, the point of `{ }` is to make a group of statements act like one. A `for` statement always controls exactly one "statement", which can be a block contained in `{ }`. Therefore, when the continuation expression is no longer true, the next "statement" to be executed is whatever follows the `}` at the end of the block.

Susan: Okay, now I get it. The `{ }` curly brackets work together with the `< 5` to determine that the program should go on to the next statement.

Steve: Right.

Susan: Now, on the "controlled block" -- so other statements can be considered controlled blocks too? I mean is a controlled block basically just the same thing as a block? I reviewed your definition of *block*, and it seems to me that they are. I guess it is just a statement that in this case is being controlled by `for`.

Steve: Correct. It's called a *controlled block* because it's under the control of another statement.

Susan: So if we used `while` before the `{ }` then that would be a `while` controlled block?

Steve: Right.

Susan: Then where in step 3 or in `i++` does it say to go back to step 2?

Steve: Again, the `for` statement executes one block (the *controlled block*) repeatedly until the continuation expression is false. Since a block is equivalent to one statement, the controlled block can also be referred to as the *controlled statement*. In the current example, the block that is controlled by the `for` loop consists of the four lines starting with the opening `{` on the next line after the `for` statement itself and ending with the closing `}` after the line that says `cin >> weight[i];`.

Susan: Okay. But now I am a little confused about something else here. I thought that `cout` statements were just things that you would type in to be seen on the screen.

Steve: That's correct, except that `cout` is a variable used for I/O, not a statement.

Susan: So then why is `<< i+1 <<` put in at this point? I understand what it does now but I don't understand why it is where it is.

Steve: Because we want to produce an output line that varies depending on the value of `i`. The first time, it should say

Please enter weight #1:

The second time, it should say

Please enter weight #2:

and so on. The number of the weight we're asking for is one more than `i`; therefore we insert the expression `<< i + 1 <<` in the output statement so that it will stick the correct number into the output line at that point.

Susan: How does `<< i+1 <<` end up as #1 ?

Steve: The first time, `i` is 0; therefore, `i + 1` is 1. The # comes from the end of the preceding part of the output statement.

Now let's continue with the next step in the description of our `for` loop, the modification expression `i++`. In our example, this will be executed five times. The first time, `i` will be 0, then 1, 2, 3, and finally 4. When the loop is executed for the fifth time, `i` will be incremented to 5; therefore, step 2 will end the loop by skipping to the next statement after the controlled block.¹³ A bit of terminology is useful here: Each time through the loop is called an *iteration*.

Let's hear Susan's thoughts on this matter.

Susan: When you say that "step 2 will end the loop by skipping to the next statement after the controlled block", does that mean it is now going on to the next `for` statement? So when `i` is no longer less than 5, the completion of the loop signals the next controlled block?

Steve: In general, after all the iterations in a loop have been performed, execution proceeds to whatever statement follows the controlled block. In this case, the next statement is indeed a `for` statement, so that's the next statement that is performed after the end of the current loop.

The discussion of the `for` statement led to some more questions about loop control facilities and the use of parentheses:

Susan: How do you know when to use `()` ? Is it only with `if` and `for` and `while` and `else` and stuff like that, whatever these statements are called? I mean they appear to be modifiers of some sort; is there a special name for them?

Steve: The term **loop control** applies to statements that control loops that can execute controlled blocks a (possibly varying) number of times; these include `for` and `while`. The `if` and `else` statements are somewhat different, since their controlled blocks are executed either once or not at all. The `()` are needed in those cases to indicate where the controlling expression(s) end and the controlled block begins. You can also use `()` to control the order of evaluation of an arithmetic expression: The part of the expression inside parentheses is executed first, regardless of normal ordering rules. For example, `2*5+3` is 13, while `2*(5+3)` is 16.

Susan: So then if you just wrote `while CurrentWeight > 0` with no `()` then the compiler couldn't read it?

Steve: Correct.

Susan: Actually it is beginning to look to me as I scan over a few figures that almost everything has a caption of some sort surrounding it. Everything either has a `" "` or `()` or `{ }` or `[]` or `<>` around it. Is that how it is going to be? I am still not clear on the different uses of `()` and `{ }`; does it depend on the control loop?

Steve: The `{ }` are used to mark the controlled block, while the `()` are used to mark the conditional expression(s) for the `if`, `while`, `for`, and the like. Unfortunately, `()` also have other meanings in C++, which we'll get to eventually. The inventor of the language considers them to have been overused for too many different meanings, and I agree.

Susan: OK, I think I have it: `{ }` define blocks and `()` define expressions. How am I to know when a new block starts? I mean if I were doing the writing, it would be like a new paragraph in English, right? So are there any rules for knowing when to stop one block and start another?

Steve: It depends entirely on what you're trying to accomplish. The main purpose of a block is to make a group of statements act like one statement; therefore, for example, when you want to control a group of statements by one `if` or `for`, you group those statements into a block.

Now that we've examined the `for` statement in excruciating detail, what about the block it controls? The first statement in the block:

```
cout << "Please type in weight #" << i+1 << ": ";
```

doesn't contain anything much we haven't seen before; it just displays a request to enter a weight. The only difference from previous uses we've made of the `cout` facility is that we're inserting a numeric expression containing a variable, `i+1`, into the output. This causes the expression to be translated into a human-readable form consisting of digits. All of the expressions being sent to `cout` in one statement are strung together to make one line of output, if we don't specify otherwise. Therefore, when this statement is executed during the first iteration of the loop, the user of this program will see:

```
Please type in weight #1:
```

Then the user will type in the first weight. The same request, with a different value for the weight number, will show up each time the user hits ENTER, until five values have been accepted.

The second statement in the controlled block,

```
cin >> Weight[i];
```

is a little different. Here, we're reading the number the user has typed in at the keyboard and storing it in a

variable. But the variable we're using is different each time through the loop: it's the `i`th element of the `Weight` vector. So, on the first iteration, the value the user types in will go into `Weight[0]`; the value accepted on the second iteration will go into `Weight[1]`; and so on, until on the fifth and last iteration, the typed-in value will be stored in `Weight[4]`.

Here's Susan's take on this.

Susan: What do you mean by the `i`th element? So does `Weight[i]` mean you are directing the number that the user types in to a certain location in memory?

Steve: Yes, to the element whose number is the current value of `i`.

Susan: When you say `cin >> Weight[i]` does that mean you are telling the computer to place that variable in the index? So this serves two functions, displaying the weight the user types in and associating it to the index?

Steve: No, that statement tells the computer to place the value read in from the keyboard into element `i` of vector `Weight`.

Susan: What I am confusing is what is being seen on the screen at the time that the user types in the input. So, the user sees the number on the screen but then it isn't displayed anywhere after that number is entered? Then, the statement `cin >> weight[i]` directs it to a location somewhere in memory with a group of other numbers that the user types in?

Steve: Correct. This will be illustrated under the `contents` of `Weight` heading in Figure [weightcontents](#).

Now that we have stored all of the weights, we want to find the three highest of the weights. We'll use a sorting algorithm called a **selection sort**, which can be expressed in English as follows:

1. Repeat the following steps three times, once through for each weight that we want to select:
2. Search through the list (i.e., the `Weight` vector), keeping track of the highest weight seen so far in the list and the index of that highest weight.
3. When we get to the end of the list, copy the highest weight we've found to the next element of another list (the "output list", which in this case is the vector `SortedWeight`).
4. Finally, set the highest weight we've found in the original list to 0, so we won't select it as the highest value again on the next pass through the list.

Let's take a look at the portion of our C++ program that implements this sort, in Figure [selsort1](#).

Sorting the weights (from `code\vect1.cc`) (Figure `selsort1`)

```
for (i = 0; i < 3; i ++)  
  
    {  
  
        HighestWeight = 0;
```

```

for (k = 0; k < 5; k ++)
    {
        if (Weight[k] > HighestWeight)
            {
                HighestWeight = Weight[k];
                HighestIndex = k;
            }
    }

SortedWeight[i] = HighestWeight;

Weight[HighestIndex] = 0;

}

```

Susan had some interesting comments and questions on this algorithm. Let's take a look at our discussion of the use of the variable `i`:

Susan: Now I understand why you used the example of `i = i + 1;` in Chapter [basics.htm](#); before, it didn't make sense why you would do that silly thing. Anyway, now let me get this straight. To say that, in the context of this exercise, means you can keep adding 1 to the value of `i`? I am finding it hard to see where this works for the number 7, say, or anything above 5 for that matter. So, it just means you can have 4 +1 or + another 1, and so on? See where I am having trouble?

Steve: Remember, a `short` variable such as `i` is just a name for a 2-byte area of RAM, which can hold any value between -32768 and +32767. Therefore, the statement `i ++;` means that we want to recalculate the contents of that area of RAM by adding 1 to its former contents.

Susan: No, that is not the answer to my question. Yes, I know all that<G>. What I am saying is this: I assume that `i ++;` is the expression that handles any value over 4, right? Then let's say that you have pumpkins that weigh 1, 2, 3, 4, and 5 pounds consecutively. No problem, but what if the next pumpkin was not 6 but say 7 pounds? If at that point, the highest value for `i` was only 5 and you could only add 1 to it, how does that work? It just doesn't yet have the base of 6 to add 1 to. Now do you understand what I am saying?

Steve: I see the problem. We're using the variable `i` to indicate which weight we're talking

about, not the weight itself. In other words, the first weight is `Weight[0]`, the second is `Weight[1]`, the third is `Weight[2]`, the fourth is `Weight[3]`, and the fifth is `Weight[4]`. The actual values of the weights are whatever the user of the program types in. For example, if the user types in 3 for the first weight, 9 for the second one, 6 for the third, 12 for the fourth, and 1 for the fifth, then the `vector` will look like this:

Element Value

`Weight[0]` 3

`Weight[1]` 9

`Weight[2]` 6

`Weight[3]` 12

`Weight[4]` 1

The value of `i` has to increase by only one each time because it indicates which element of the `vector` `Weight` is to store the current value being typed in by the user. Does this clear up your confusion?

Susan: I think so. Then it can have any whole number value 0 or higher (well, up to 32767); adding the 1 means you are permitting the addition of at least 1 to any existing value, thereby allowing it to increase. Is that it?

Steve: No, I'm not permitting an addition; I'm performing it. Let's suppose `i` is 0. In that case, `Weight[i]` means `Weight[0]`, or the first element of the `Weight` vector. When I add 1 to `i`, `i` becomes 1. Therefore, `Weight[i]` now means `Weight[1]`. The next execution of `i ++;` sets `i` to 2; therefore, `Weight[i]` now means `Weight[2]`. Any time an `i` is used in an expression, for example, `Weight[i]`, `i + j`, or `i + 1` you can replace the `i` by whatever the current value of `i` is. The only place where you can't replace a variable such as `i` by its current value is when it is being modified, as in `i ++` or the `i` in `i = j + 1`. In those cases, `i` means the address where the value of the variable `i` is stored.

Susan: OK, then `i` is not the number of the value typed in by the user; it is the location of an element in the `Weight` vector, and that is why it can increase by 1, because of the `i ++`?

Steve: Correct, except that I would say "that is why it *does* increase by 1". This may just be terminology.

Susan: But in this case it can increase no more than 4 because of the `i < 5` thing?

Steve: Correct.

Susan: But it has to start with a 0 because of the `i = 0` thing?

Steve: Correct.

Susan: So then `cin >> Weight [i]` means that the number the user is typing has to go into one of those locations but the only word that says what that location could be is `Weight`; it puts no limitations on the location in that `Weight` vector other than when you defined the index variable as `short i`; . This means the index cannot be more than 32767.

Steve: Correct. The current value of `i` is what determines which element of `Weight` the user's input goes into.

Susan: I think I was not understanding this because I kept thinking that `i` was what the user typed in and we were defining its limitations. Instead we are telling it where to go.

Steve: Correct.

Having beaten that topic into the ground, let's look at the correspondence between the English description of the algorithm and the code:

1. Repeat the following steps once through for each prize:

```
for ( i = 0; i < 3; i ++)
```

(During this process the variable `i` is the index into the `SortedWeight` vector where we're going to store the weight for the current prize we're working on. While we're looking for the highest weight, `i` is 0; for the second-highest weight, `i` is 1; finally, when we're getting ready to award a third prize, `i` will be 2.)

2. Search through the input list. For each element of the list `Weight`, we check whether that element (`Weight [k]`) is greater than the highest weight seen so far in the list (`HighestWeight`). If that is the case, then we reset `HighestWeight` to the value of the current element (`Weight [k]`) and the index of the highest weight so far (`HighestIndex`) to the index of the current element (`k`).
3. When we get to the end of the input list, `HighestWeight` is the highest weight in the list, and `HighestIndex` is the index of that element of the list that had the highest weight. Therefore, we can copy the highest weight to the current element of another list (the "output list"). As mentioned earlier, `i` is the index of the current element in the output list. Its value is the number of times we have been through the outer loop before; that is, the highest weight, which we will identify first, goes in position 0 of the output list, the next highest in position 1, and so on:

```
SortedWeight [ i ] = HighestWeight ;
```

4. Finally, set the highest weight in the input list to 0, so we won't select it as the highest value again on the next pass through the list.

```
Weight [ HighestIndex ] = 0 ;
```

This statement is the reason that we have to keep track of the "highest index"; that is, the index of

the highest weight. Otherwise, we wouldn't know which element of the original `Weight` vector we've used and therefore wouldn't be able to set it to 0 to prevent its being used again.

Here's Susan's rendition of this algorithm:

Susan: OK, let me repeat this back to you in English. The result of this program is that after scanning the list of user input weights the weights are put in another list, which is an ordering list, named `k`. The program starts by finding the highest weight in the input list. It then takes it out, puts it in `k`, and replaces that value it took out with a 0, so it won't be picked up again. Then it comes back to find the next highest weight and does the same thing all over again until nothing is left to order. Actually this is more than that one statement. But is this what you mean? That one statement is responsible for finding the highest weight in the user input list and placing it in `k`. Is this right?

Steve: It's almost exactly right. The only error is that the list that the weights are moved to is the `SortedWeight` vector, rather than `k`. The variable `k` is used to keep track of which is the next entry to be put into the `SortedWeight` vector.

Susan: OK. There was also something else I didn't understand when tracing through the program. I did see at one point during the execution of the tracing version of this program that `i=5`. Well, first I didn't know how that could be because `i` is supposed to be `< 5`, but then I remembered that `i++` expression in the `for` loop, so I wondered if that is how this happened. I forgot where I was at that point, but I think it was after I had just completed entering 5 values and `i` was incrementing with each value. But see, it really should not have been more than 4 because if you start at 0 then that is where it should have ended up.

Steve: The reason that `i` gets to be 5 after the end of the loop is that at the end of each pass through the loop, the modification expression (`i++`) is executed before the continuation expression (`i < 5`). So, at the end of the fifth pass through the loop, `i` is incremented to 5 and then tested to see if it is still less than 5. Since it isn't, the loop terminates at that point.

Susan: I get that. But I still have a question about the statement `if Weight[k] > highest_weight`. Well, the first time through, this will definitely be true because we've initialized `HighestWeight` to 0, since any weight would be greater than 0. Is that right?

Steve: Yes. Every time through the outer (`i`) loop, as we get to the top of the inner loop, the 0 that we've just put in `HighestWeight` should be replaced by the first element of `Weight`; that is, `Weight[0]`, except of course if we've already replaced `Weight[0]` by 0 during a previous pass. It would also be possible to initialize `HighestWeight` to `Weight[0]` and then start the loop by setting `k` to 1 rather than 0. That would cause the inner (`k`) loop to be executed only four times per outer loop execution, rather than five, and therefore would be more efficient.

Susan: Then `HighestIndex=k;` is the statement that sets the placement of the highest number to its rank?

Steve: Right.

Susan: Then I thought about this. It seems that the highest weight is set first, then the sorting takes place so it makes four passes (actually five) to stop the loop.

Steve: The sorting is the whole process. Each pass through the outer loop locates one more element to be put into the `SortedWeight` vector. Is that what you're saying here?

Susan: Then the statement `Weight[HighestIndex] = 0;` comes into play, replacing the highest number selected on that pass to 0.

Steve: Correct.

Susan: Oh, when `k` is going through the sorting process why does `i` increment though each pass? It seems that `k` should be incrementing.

Steve: Actually, `k` increments on each pass through the inner loop, or 15 times in all. It's reset to 0 on each pass through the outer loop, so that we look at all of the elements again when we're trying to find the highest remaining weight. On the other hand, `i` is incremented on each pass through the outer loop or three times in all, once for each "highest" weight that gets put into the `SortedWeight` vector.

Susan: OK, I get the idea with `i`, but what is the deal with `k`? I mean I see it was defined as a `short`, but what is it supposed to represent, and how did you know in advance that you were going to need it?

Steve: It represents the position in the original list, as indicated in the description of the algorithm.

Susan: I still don't understand where `k` fits into this picture. What does it do?

Steve: It's the index in the "inner loop", which steps through the elements looking for the highest one that's still there. We get one "highest" value every time through the "outer loop", so we have to execute that outer loop three times. Each time through the outer loop, we execute the inner loop five times, once for each entry in the input list.

Susan: Too many terms again. Which is the "outer loop" and which was the "inner loop"?

Steve: The outer loop executes once for each "highest" weight we're locating. Each time we find one, we set it to 0 (at the end of the loop) so that it won't be found again the next time through.

Susan: OK, now I am confused with the statement: `if (Weight[k] > HighestWeight)`. This is what gets me: if I understand this right (and obviously I don't) how could `Weight[k]` ever be greater than `HighestWeight`, since every possible value of `k` represents one of the elements in the `Weight` vector, and `HighestWeight` is the highest weight in that vector? For this reason I am having a hard time understanding the code for step 2, but not the concept.

Steve: The value of `HighestWeight` at any time is equal to the highest weight that has

been seen *so far*. At the beginning of each execution of the outer loop, `HighestWeight` is set to 0. Then, every time that the current weight (`Weight[k]`) is higher than the current value of `HighestWeight`, we reset `HighestWeight` to the value of the current weight.

Susan: I still don't understand this statement. Help.

Steve: Remember that `HighestWeight` is reset to 0 on each pass through the outer loop. Thus, this `if` statement checks whether the `k`th element of the `Weight` vector exceeds the highest weight we've seen before in this pass. If that is true, obviously our "highest" weight isn't really the highest, so we have to reset the highest weight to the value of the `k`th element; if the `k`th element isn't the true highest weight, at least it's higher than what we had before. Since we replace the "highest" weight value with the `k`th value any time that the `k`th value is higher than the current "highest" weight, at the end of the inner loop, the number remaining in `HighestWeight` will be the true highest weight left in `Weight`. This is essentially the same algorithm as we used to find the highest weight in the original version of this program, but now we apply it several times to find successively lower "highest" weights.

Susan: OK, I understand now, `i` increments to show how many times it has looped through to find the highest number. You are doing a loop within a loop, really, it is not side by side is it?

Steve: Correct.

Susan: So, when you first enter your numbers they are placed in an index called `i`, then they are going to be cycled through again, placing them in a corresponding index named `k`, looking for the top three numbers. To start out through each pass, you first set the highest weight to the first weight since you have preset the highest weight to 0. But, to find the top three numbers you have to look at each place or element in the index. At the end of each loop you sort out the highest number and then set that removed element to 0 so it won't be selected again. You do this whole thing three times.

Steve: That's right, except for some terminology: where you say "an index called `i`", you should say "a vector called `Weight`", and where you say "an index called `k`", you should say "a vector called `SortedWeight`". The variables `i` and `k` are used to step through the vectors, but they are not the vectors themselves.

Susan: OK, then the index variables just are the working representation of what is going on in those vectors. But are not the numbers "assigned" an index? Let's see; if you lined up your five numbers you could refer to each number as to its placement in a vector. Could you then have the column of weights in the middle of the two indexes of `i` and `k` to each side?

Steve: If I understand your suggestion, it wouldn't work, because `k` and `i` vary at different speeds. During the first pass of the outer loop, `i` is 0, while `k` varies from 0 to 5; on the second pass of the outer loop, `i` is 1, while `k` varies from 0 to 5 again, and the same for the third pass of the outer loop. The value of `i` is used to refer to an individual element of the `SortedWeight` vector, the one that will receive the next "highest" weight we locate. The value of `k` is used to refer to an individual element of the `Weight` vector, the one

we're examining to see if it's higher than the current `HighestWeight`.

Susan: This is what gets me, how do you know in advance that you are going to have to set `HighestIndex` to `k`? I see it in the program as it happens and I understand it then, but how would you know that the program wouldn't run without doing that? Trial and error? Experience? Rule books? <G>

Steve: Logic. Let's look at the problem again. The sorting algorithm that we're using here is called *selection sort*, because each time through the outer loop it selects one element out of the input `vector` and moves it to the output `vector`. To prevent our selecting the same weight (i.e., the highest one in the original input) every time through the outer loop, we have to clear each weight to 0 as we select it. But, to do that, we have to keep track of which one we selected; that's why we need to save `HighestIndex`.

Being a glutton for punishment, Susan brought up the general problem of how to create an algorithm in the first place.

Susan: Do they make instruction sheets with directions of paths to follow? How do you identify problems? I mean, don't you encounter pretty much the same types of problems frequently in programming and can they not be identified some way so that if you knew a certain problem could be categorized as a Type C problem, let's say, you would approach it with a Type C methodology to the solution? Does that make sense? Probably not.

Steve: It does make sense, but for some reason such "handbooks" are rare. Actually, my previous book, *Efficient C/C++ Programming* was designed to provide something like you're suggesting, with solutions to common problems at the algorithmic level. There's also a book called *Design Patterns* that tries to provide tested solutions to common design problems, at a much higher level.

Details, Details

Let's go back and look at the steps of the algorithm more closely (on page). Step 1 should be fairly self-explanatory, once you're familiar with the syntax of the `for` statement; it causes the statements in its controlled block to be executed three times, with the index variable `i` varying from 0 to 2 in the process.

Step 2 is quite similar to the process we went through to find the highest weight in our previous two programs; however, the reason for the `HighestIndex` variable may not be obvious. We need to keep track of which element of the original `vector` (i.e., `Weight`) we have decided is the highest so far, so that this element won't be selected as the highest weight on *every* pass through the `Weight` `vector`. To prevent this error, step 4 sets each "highest" weight to a value that won't be selected on a succeeding pass. Since we know there should be no 0 weights in the `Weight` `vector`, we can set each selected element to 0 after it has been selected, to prevent its reselection. Figure [weightcontents](#) shows a picture of the situation before the first pass through the data, with ??? in `SortedWeight` to indicate that those locations contain unknown data, as they haven't been initialized yet.

Initial situation (Figure [weightcontents](#))

Index	contents	contents
-------	----------	----------

	of Weight	of SortedWeight
--	-----------	-----------------

0	5	???
1	2	???
2	11	???
3	3	
4	7	

In Figure [weightcontents](#), the highest value is 11 in `Weight[2]`. After we've located it and copied its value to `SortedWeight[0]`, we set `Weight[2]` to 0, yielding the situation in Figure [after.first](#).

After the first pass (Figure after.first)

Index	contents	contents
	of Weight	of SortedWeight
0	5	11
1	2	???
2	0	???
3	3	
4	7	

Now we're ready for the second pass. This time, the highest value is the 7 in `Weight[4]`. After we copy the 7 to `SortedWeight[1]`, we set `Weight[4]` to 0, leaving the situation in Figure [aftersecondpass](#).

After the second pass (Figure aftersecondpass)

Index	contents	contents
-------	----------	----------

	of Weight	of SortedWeight
--	-----------	-----------------

0	5	11
1	2	7
2	0	???
3	3	
4	0	

On the third and final pass, we locate the 5 in `Weight[0]`, copy it to `SortedWeight[2]`, and set `Weight[0]` to 0. As you can see in Figure [final.situation](#), `SortedWeight` now has the results we were looking for: the top three weights, in descending order.

Final situation (Figure final.situation)

Index	contents	contents
	of Weight	of SortedWeight

0	0	11
1	2	7
2	0	5
3	3	
4	0	

To Err Is Human. . .

That accounts for all of the steps in the sorting algorithm. However, our implementation of the algorithm has a weak spot that we should fix. If you want to try to find it yourself, look at the code and explanation again before going on. Ready?

The key word in the explanation is "should" in the following sentence: "Since we know there *should* be no

0 weights in the `Weight` vector, we can set each selected element to 0 after it has been selected, to prevent its reselection." How do we *know* that there are no 0 weights? We don't, unless we screen for them when we accept input. In the first pumpkin-weighing program, we stopped the input when we got a 0, but in the programs in this chapter, we ask for a set number of weights. If one of them is 0, the program will continue along happily.¹⁴ Before we change the program, though, let's try to figure out what would happen if the user types in a 0 for every weight.

You can try this scenario out yourself. To run it, just change to the `normal` subdirectory under the main directory where you installed the software, and type `vect1`. When it asks for weights, enter a 0 for each of the five weights. In case you're reading this away from your computer, here's what will happen (although the element number in the message may not be the same):

```
You have tried to use element 51082 of a vector which has only 5
elements.
```

Why doesn't the program work in this case? Because we have an **uninitialized variable**; that is, one that has never been set to a valid value. In this case, it's `HighestIndex`. Let's look at the sorting code one more time, in Figure [sortweightsagain](#).¹⁵

Sorting the weights, again (from `code\vect1.cc`) (Figure [sortweightsagain](#))

```
for (i = 0; i < 3; i ++)  
  
    {  
  
    HighestWeight = 0;  
  
    for (k = 0; k < 5; k ++)  
  
        {  
  
        if (Weight[k] > HighestWeight)  
  
            {  
  
            HighestWeight = Weight[k];  
  
            HighestIndex = k;  
  
            }  
  
        }  
  
    }  
  
    SortedWeight[i] = HighestWeight;  
  
    Weight[HighestIndex] = 0;
```

}

It's clear that `HighestWeight` is **initialized** (i.e., given a valid value) before it is ever used; the statement `HighestWeight = 0;` is the first statement in the block controlled by the outer `for` loop. However, the same is not true of `HighestIndex`. Whenever the condition in the `if` statement is `true`, both `HighestWeight` and `HighestIndex` will indeed be set to legitimate values: `HighestWeight` will be the highest weight seen so far on this pass, and `HighestIndex` will be the index of that weight in the `Weight` vector. However, what happens if the condition in the `if` statement never becomes `true`? In that case, `HighestIndex` will have whatever random value it started out with at the beginning of the program; it's very unlikely that such a value will be correct or even refer to an actual element in the `Weight` vector.

Here's the discussion that Susan and I had on this topic:

Susan: You say that `HighestIndex` isn't initialized properly. But what about when you set `k` equal to 0 and then `HighestIndex` is set equal to `k`? Is that not initialized?

Steve: The problem is that the statement `HighestIndex = k;` is executed only when `Weight[k]` is greater than `HighestWeight`. If that never occurs, then `HighestIndex` is left in some random state.

Susan: OK, then why didn't you say so in the first place? I understand that. However, I still don't understand why the program would fail if all the weights the user typed in were 0. To me it would just have a very boring outcome.

Steve: That's the case in which `HighestIndex` would never be initialized; therefore, it would contain random garbage and would cause the program to try to display an element at some random index value.

Susan: I traced through the program again briefly tonight and that reminds me to ask you why you put the highest weight value to 1596 and the second-highest weight value to 1614?

Steve: I didn't. Those just happened to be the values that those memory locations had in them before they were initialized.

Susan: I was totally confused right from the beginning when I saw that. But did you do that to show that those were just the first two weights, and that they have not been, how would you say this, "ordered" yet? I don't know the language for this in computerese, but I am sure you know what I am saying.

Steve: Not exactly; they haven't been initialized at that point, so whatever values they might contain would be garbage.

Susan: So at that point they were just the first and second weights, or did you just arbitrarily put those weights in there to get it started? Anyway, that was baffling when I saw that.

Steve: Before you set a variable to a particular value, it will have some kind of random junk in it. That's what you're seeing at the beginning of the program, before the variables have been initialized.

Susan: OK, I am glad this happened, I can see this better, but whose computer did that? Was it yours or mine? I mean did you run it first and your computer did it, or was it my computer that came up with those values?

Steve: It's your computer. The program starts out with "undefined" values for all of the uninitialized variables. What this means in practice is that their values are whatever happened to be left around in memory at those addresses. This is quite likely to be different on your machine from what it is on mine or even on yours at a different time.

Susan: So something has to be there; and if you don't tell it what it is, the old contents of memory just comes up?

Steve: Right.

Susan: If it had worked out that the higher number had been in first place, then I would have just assumed that you put that there as a starting point. I am really glad that this happened but I was not too happy about it when I was trying to figure it out.

Steve: See, it's all for your own good.

Susan: If that were the case, I would think it nearly impossible that we have the same values at any given address. How could they ever be remotely the same?

Steve: It's very unlikely that they would, unless the address were one that was used by very basic software such as DOS or Windows, which might be the same on our computers.

Susan: Anyway, then you must have known I was going to get "garbage" in those two variables, didn't you? Why didn't you advise me at least about that? Do you know how confusing it was to see that first thing?

Steve: Yes, but it's better for you to figure it out yourself. Now you really know it, whereas if I had told you about it in advance, you would have relied on my knowledge rather than developing your own.

I hope that has cleared up the confusion about the effect of an uninitialized variable in this example. But, why do we have to initialize variables ourselves? Surely they must have some value at any given time. Let's listen in on the conversation that Susan and I had about this point:

Susan: So, each bit in RAM is capable of being turned on or off by a 1 or a 0? Which one is on and which one is off? Or does that matter? How does this work electronically? I mean how does the presence of a 0 or a 1 throw the RAM into a different electronic state?

Steve: To be more exact, each "switch" is capable of existing in either the "on" or "off" state. The assignment of states to 1s and 0s is our notion, which doesn't affect the fact that there are

exactly two distinct states the switch can assume, just like a light switch (without a dimmer). We say that if the switch is off, it's storing a 0, and if it's on, it's storing a 1.

Susan: What is the "normal state" of RAM: on or off?

Steve: It's indeterminate. That's one reason why we need to explicitly set our variables to a known state before we use them.

Susan: That didn't make sense to me originally, but I woke up this morning and the first thing that came to my mind was the light switch analogy. I think I know what you meant by *indeterminate*.

If we consider the light switch as imposed with our parental and financial values, it is tempting to view the "normal state" of a light switch as off. Hey, does the light switch really care? It could sit there for 100 years in the on position as easily as in the off position. Who is to say what is normal? The only consequence is that the light bulb will have been long burned out. So it doesn't matter, it really doesn't have a normal state, unless people decide that there is one.

Steve: What you've said is correct. The switch doesn't care whether it's on or off. In that sense, the "normal" position doesn't really have a definition other than one we give it.

However, what I meant by *indeterminate* is slightly different: When power is applied to the RAM, each bit (or to be more precise, a switch that represents that bit) could just as easily start out on as off. It's actually either one or the other, but which one is pretty much random, so we have to set it to something before we know its value.

Susan: Oh, you broke my heart, when I thought I had it all figured out! Well, I guess it was OK, at least as far as the light switch was concerned, but then RAM and a light switch are not created equal. So RAM is pretty easy to please, I guess. . .

After that bit of comic relief, let's get back to the analysis of this program. It should be fairly obvious that if the user types in even one weight greater than 0, the `if` statement will be `true` when that weight is encountered, so the program will work. However, if the user typed in all 0 weights, the program would fail, as we saw before, because the condition in the `if` statement would never become `true`. To prevent this from causing program failure, all we have to do is to add one more line, the one in **bold** in Figure [selsort2](#).

Sorting the weights, with correct initialization (from `code\vect2.cc`) (Figure `selsort2`)

```
for (i = 0; i < 3; i ++)  
  
    {  
  
        HighestWeight = 0;  
  
        HighestIndex = 0;
```

```

for (k = 0; k < 5; k ++)
{
    if (Weight[k] > HighestWeight)
    {
        HighestWeight = Weight[k];
        HighestIndex = k;
    }
}

SortedWeight[i] = HighestWeight;

Weight[HighestIndex] = 0;
}

```

Now we can be sure that `HighestIndex` always has a value that corresponds to some element of the `Weight` vector, so we won't see the program fail as the previous one would.

Assuming that you've installed the software from the CD-ROM in the back of this book, you can run the corrected program to test that it works as advertised. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE vect2`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `vect2` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

To run it under the debugger, make sure you are in the `code` subdirectory, and then type `"RHIDE vect2"`. Again, do *not* add the `".cc"` to the end of the file name. Once `RHIDE` has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

After you've entered 5 weights, the program will start the sorting process. This time, entering five 0 weights will produce the expected result: The top three weights will all be 0.

By the way, it's also possible to initialize a variable at the same time as you define it. For example, the statement `short i = 12;` defines a `short` variable called `i` and sets it to the value 12 at the same time. This is generally a good practice to follow when possible; if you initialize the variable when you define it, you don't have to remember to write a separate statement to do the initialization.

To Really Foul Things up Requires a Computer

We should pay some more attention to the notion of program failure, as it's very important. The first question, of course, is what it means to say that a program "fails". The best answer is that it doesn't work correctly, but that isn't very specific.

As you can imagine, this notion was the topic of some discussion with Susan:

Susan: What do you mean by a program failing? I know it means it won't work, but what happens? Do you just get error messages, and it won't do anything? Or is it like the message that you have on page ?

Steve: In general, a program "failing" means that it does something unexpected and erroneous. Because I have put some safety features into the implementation of `vector`, you'll get an error message if you misuse a `vector` by referring to a nonexistent element.

In general, a program failure may or may not produce an error message. In the specific case that we've just seen, we'll probably get an error message while trying to access a nonexistent element of the `Weight vector`. However, it's entirely possible for a program to just "hang" (run endlessly), "crash" your system, produce an obviously ridiculous answer, or worst of all, provide a seemingly correct but actually erroneous result.

The causes of program failures are legion. A few of the possibilities are these:

1. Problems isolated to our code

1. The original problem could have been stated incorrectly.
2. The algorithm(s) we're using could have been inappropriate for the problem.
3. The algorithm(s) might have been implemented incorrectly.
4. An input value might be outside the expected range.
- 5.
6. And so on. . .

2. Problems interacting with other programs

1. We might be misusing a function supplied by the system, like the `<<` operator.
2. The documentation for a system function might be incorrect or incomplete. This is especially common in "guru"-oriented operating systems, where the users are supposed to know everything.
3. A system function might be unreliable. This is more common than it should be.
4. The compiler might be generating the wrong instructions. I've seen this on a few rare occasions.
5. Another program in the system might be interfering with our program. This is quite common in some popular operating environments that allow several programs to be executing concurrently.¹⁶
6. And so on. . .

With a simple program such as the ones we're writing here, errors such as the ones listed under problems with our code are more likely, as we have relatively little interaction with the rest of the system. As we start

to use more sophisticated mechanisms in C++, we're more likely to run into instances of interaction problems.

What, Me Worry?

After that excursion into the sources of program failure, let's get back to our question about about initializing variables. Why do we have to worry about this at all? It would seem perfectly reasonable for the compiler to make sure that our variables were always initialized to some reasonable value; in the case of numeric variables such as a `short`, 0 would be a good choice. Surely Bjarne Stroustrup, the designer of C++, didn't overlook this. No, he didn't; he made a conscious decision not to provide this facility. It's not due to cruelty or unconcern with the needs of programmers. On the contrary, he stated in the Preface to the first Edition of *The C++ Programming Language* that "C++ is a general-purpose programming language designed to make programming more enjoyable for the serious programmer".¹⁷ To allow C++ to replace C completely, he could not add features that would penalize efficiency for programs that do not use these features. Adding initialization as a built-in function of the language would make programs larger and slower if the programmer had already initialized all variables as needed. This may not be obvious, but we'll see in a later section why it is so.

Here's Susan's reaction to these points about C++:

Susan: What is run-time efficiency?

Steve: How long it takes to run the program and how much memory it uses up.

Susan: So are you saying that C++ is totally different from C? That one is not based on the other?

Steve: No, C++ is a descendant of C. However, C++ provides much more flexibility to programmers than C.

Susan: Now, about what Bjarne said back in 1986: Who enjoys this, and if C++ is intended for a serious programmer, why am I reading this book? What is a serious programmer? Would you not think a serious programmer should have at least taken Computer Programming 101?

Steve: This book should be a pretty good substitute for Computer Programming 101. You probably know considerably more than the usual graduate of such a course, although the e-mail tutoring has been a major contributor to your understanding. Anyway, if you want to learn how to program, you have to start somewhere, and it might as well be with the intention of being a serious programmer.

Garbage in, Garbage Out

In the meantime, there's something else we should do if we want the program to work as it should. As the old saying "Garbage in, garbage out" suggests, by far the best solution to handling spurious input values is to prevent them from being entered in the first place. What we want to do is to check each input value and warn the user if it's invalid. Figure [garbprev1](#) illustrates a new input routine that looks like it should do the

trick.

Garbage prevention, first attempt (from code\vect2a.cc) (Figure garbprev1)

```

for (i = 0; i < 5; i ++)
{
    cout << "Please type in weight #" << i+1 << ": ";

    cin >> Weight[i];

    if (Weight[i] <= 0)
    {
        cout << "I'm sorry, " << Weight[i] << " is not a valid weight.";

        cout << endl;
    }
}

```

Assuming that you've installed the software from the CD-ROM in the back of this book, you can try out this program. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE vect2a`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `vect2a` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

To run it under the debugger, make sure you are in the `code` subdirectory, and then type "`RHIDE vect2a`". Again, do *not* add the ".cc" to the end of the file name. Once `RHIDE` has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

When you are asked for a weight, type one in and hit ENTER just as when executing normally. After you've entered 5 weights, the program will start the sorting process. When finished, it will display the top three weights of the five that were entered.

Most of this should be familiar; the only line that has a new construct in it is the `if` statement. The condition `<=` means "less than or equal to", which is reasonably intuitive.

Unfortunately, this program won't work as we intended. The problem is what happens after the error

message is displayed; namely, the loop continues at the top with the next weight, and we never correct the erroneous input. Susan didn't have much trouble figuring out exactly what that last statement meant:

Susan: When you say that "we never correct the erroneous input", does that mean that it is added to the list and not ignored?

Steve: Right.

To fix this problem completely, we need to use the approach shown in the final version of this program (Figure [vect3](#)). Assuming that you've installed the software from the CD-ROM in the back of this book, you can try out this program. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE vect3`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `vect3` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

To run it under the debugger, make sure you are in the `code` subdirectory, and then type "`RHIDE vect3`". Again, do *not* add the ".cc" to the end of the file name. Once RHIDE has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

When you are asked for a weight, type one in and hit ENTER just as when executing normally. After you've entered 5 weights, the program will start the sorting process, and will display the results when finished.

Finding the top three weights using `vectors` (`code\vect3.cc`) (Figure [vect3](#))

[code/vect3.cc](#)

Now let's look at the changes that we've made to the program from the last revision. The first change is that the `for` loop has only two sections rather than three in its control definition (inside the `()`). As you may recall, the first section specifies the initial condition of the index variable; in this case, we're starting `i` out at 0, as is usual in C and C++. The second section indicates when we should continue executing the loop; here, it's as long as `i` is less than 5. But the third section, which usually indicates what to do to the index variable, is missing. The reason for this is that we're going to adjust the index variable manually in the loop, depending on what the user enters.

In this case, if the user enters an invalid value (i.e., less than or equal to 0), we display an error message and leave `i` as it was, so that the next time through the loop, the value will go into the same element in the `Weight vector`. When the user enters a valid value, the `else` clause increments `i` so that the next value will go into the next element in the `vector`. This fixes the error in our previous version that left incorrect entries in the `vector`. Now that we have beaten the pumpkin weighing example to a pulp, let's review the mass of information to which I've subjected you so far in this chapter.

Review

We started out by extending our pumpkin weighing program to tell us the highest two weights rather than just the highest one. During this exercise, we learned the use of the `else` clause of an `if` statement. We also saw that making even an apparently simple change to a working program can introduce an error; in this case we were copying the highest weight to the next highest weight only when a new high weight was detected. This would produce an incorrect result if a value higher than the previous second highest but lower than the current highest weight were entered.

Next we extended the program again, this time to handle any number of prizes to be given to the highest weight, second-highest weight, third-highest weights, and so on. This required a complete reorganization of the program; the new version used the *selection sort* algorithm to produce a list of as many of the highest weights as we need, in descending order. To do this, we had to use a `vector`, or set of values with a common name, to store all of the weights as they were read in. When they had all been entered, we searched through them three times, once to find each of the top three elements. A `vector`, just like a regular variable, has a name. However, unlike a regular variable, a `vector` does not have a single value, but rather consists of a number of *elements*, each of which has a separate value. An element is referred to by a number, called an *index*, rather than by a unique name; each element has a different index. The lowest index is 0, and the highest index is 1 less than than the number of elements in the `vector`; for example, with a 10 element `vector`, the legal indexes are 0 through 9. The ability to refer to an element by its index allows us to vary the element we are referring to in a statement by varying the index; we put this facility to good use in our implementation of the selection sort, which we'll review shortly.

We then added the `for` statement to our repertoire of loop control facilities. This statement provides more precise control than the `while` statement. Using `for`, we can specify a *starting expression*, a *continuation expression*, and a *modification expression*. The starting expression sets up the initial conditions for the loop. Before each possible execution of the controlled block, the continuation expression is checked, and if it is `true`, the controlled block will be executed; otherwise, the `for` loop will terminate. Finally, the modification expression is executed after each execution of the controlled block. Most commonly, the starting expression sets the initial value of a variable, the continuation expression tests whether that variable is still in the range we are interested in, and the modification expression changes the value of the variable. For example, in the `for` statement

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

the starting expression is `i = 0`, the continuation expression is `i < 5`, and the modification expression is `i ++`. Therefore, the block controlled by the `for` statement will be executed first with the variable `i` set to 0; at the end of the block, the variable `i` will be incremented by 1, and the loop will continue if `i` is still less than 5.

Then we used the `for` statement and a couple of `vectors` to implement a *selection sort*. This algorithm goes through an "input list" of `n` elements once for each desired "result element". In our case, we want the top three elements of the sorted list, so the input list has to be scanned three times. On each time through, the algorithm picks the highest value remaining in the list and adds that to the end of a new "output list". Then it removes the found value from the input list. At the end of this process, the output list has all of the desired values from the input list, in descending order of size. When going over the program, we found a weak spot in the first version: If all the weights the user typed were less than or equal to 0, the program would fail because one of the variables in the program would never be *initialized*; that is, set to a known value.

This led to a discussion of why variable initialization isn't done automatically in C++. Adding this feature to programs would make them slower and larger than C programs doing the same task, and C++ was intended to replace C completely. If C++ programs were significantly less efficient than equivalent C programs, this would not be possible, so Bjarne Stroustrup omitted this feature.

While it's important to insure that our programs work correctly even when given unreasonable input, it's even better to prevent this situation from occurring in the first place. So the next improvement we made to our pumpkin weighing program was to tell the user when an invalid value had been entered and ask for a valid value in its place. This involved a `for` loop without a modification expression, since we wanted to increment the index variable `i` to point to the next element of the `vector` only when the user typed in a valid entry; if an illegal value was typed in, we requested a legal value for the same element of the `vector`.

Exercises

So that you can test your understanding of this material, here are some exercises.

1. If the program in Figure [morebas00](#) is run, what will be displayed?

Exercise 1 (code\morbas00.cc) (Figure morebas00)

[code/morbas00.cc](#)

2. If the program in Figure [morebas01](#) is run, what will be displayed?

Exercise 2 (code\morbas01.cc) (Figure morebas01)

[code/morbas01.cc](#)

3. Write a program that asks the user to type in a weight, and display the weight on the screen.

4. Modify the program from exercise 3 to ask the user to type as many weights as desired, stopping as soon as a 0 is entered. Add up all of the weights entered and display the total on the screen at the end of the program.

Answers to exercises can be found at the end of the chapter.

Conclusion

We've covered a lot of material in this chapter in our quest for better pumpkin weighing, ranging from sorting data into order based on numeric value through the anatomy of `vectors`. Next, we'll take up some more of the language features you will need to write any significant C++ programs.

Answers to Exercises

1. The correct answer is: "Who knows?" If you said "30", you forgot that the loop variable values are from 0 through 4, rather than from 1 through 5. On the other hand, if you said "20", you had the right total of the numbers 0, 2, 4, 6, and 8, but didn't notice that the variable `Result` was never initialized. Of course, adding anything to an unknown value makes the final value unpredictable. Most current compilers, including the one on the CD-ROM in the back of this book, are capable of warning you about such problems; if you compiled the program with this warning turned on, you'd see a message something like this:

```
morbass00.cc:7: warning: `short result' may be used uninitialized in this function.
```

This is the easiest way to find such errors, especially in a large program. Unfortunately, the compiler may produce such warnings even when they are not valid, so the final decision is still up to you.

Assuming that you've installed the software from the CD-ROM in the back of this book, you can try this program out. First, you have to compile it by changing to the `code` subdirectory under the main directory where you installed the software, and typing `RHIDE morbas00`, then using the "Make" command from the "Compile" menu. Then exit back to DOS and type `morbass00` to run the program. It will ask you for weights and keep track of the highest weight and second-highest weight that you've entered. Type 0 and hit ENTER to end the program.

Running this program normally isn't likely to give you much information. To run it under the debugger, make sure you are in the `code` subdirectory, and then type "`RHIDE morbas00`". Again, do *not* add the ".cc" to the end of the file name. Once `RHIDE` has started up, you can step through the program by hitting F8, which will treat any function call as one statement, or by hitting F7, which will step into any function call. Any time you want to see the display that the user would see when running the program normally, hit Alt-F5, then ENTER to get back to the debugger.

2. The correct answer is 45. In case this isn't obvious, consider the following:

1. The value of `x[0]` is set to 3.
2. In the first `for` loop, the value of `i` starts out at 1.
3. Therefore, the first execution of the assignment statement `x[i] = x[i-1] * 2;` is equivalent to `x[1] = x[0] * 2;`. This clearly sets `x[1]` to 6.
4. The next time through the loop `i` is 2, so that same assignment statement `x[i] = x[i-1] * 2;` is equivalent to `x[2] = x[1] * 2;`. This sets `x[2]` to 12.
5. Finally, on the last pass through the loop, the value of `i` is 3, so that assignment statement `x[i] = x[i-1] * 2;` is equivalent to `x[3] = x[2] * 2;`. This sets `x[3]` to 24.
6. The second `for` loop just adds up the values of all the entries in the `x` vector; this time, we remembered to initialize the total, `Result`, to 0, so the total is calculated and displayed correctly.

Running this program normally isn't likely to give you much information, but you might want to run it under control of the `gdb` debugger. You can do this in exactly the same way as you did the previous program, except that you would type `RHIDE morbas01` rather than `RHIDE morbas00`.

3. Let's start with Susan's proposed solution to this problem, in Figure [morebas02](#), and the questions that came up during the process.

Weight requesting program, first try (code\morbas02.cc) (Figure morebas02)

<code/morbas02.cc>

Susan: Would this work? Right now by just doing this it brought up several things that I have not thought about before.

First, is the # standard for no matter what type of program you are doing?

Steve: The `iostream.h` header file is needed if you want to use `<<`, `>>`, `cin` and `cout`, which most programs do, but not all.

Susan: Ok, but I meant the actual pound sign (#), is that always a part of `iostream.h`?

Steve: It's not part of the filename, it's part of the `#include` command, which tells the compiler that you want it to pretend that you've just typed in the entire `iostream.h` file in your program at that point.

Susan: So then this header is declaring that all you are going to be doing is input and output?

Steve: Not exactly. It tells the compiler how to understand input and output via `<<` and `>>`. Each header tells the compiler how to interpret some type of library functions; `iostream.h` is the one for input and output.

Susan: Where is the word `iostream` derived from? (OK, `io`, but what about `stream`?)

Steve: A `stream` is C++ talk for "a place to get or put characters". A given `stream` is usually either an `istream` (input stream) or an `ostream` (output stream). As these names suggest, you can read from an `istream` or write to an `ostream`.

Susan: Second, is the `\n` really necessary here, or would the program work without it?

Steve: It's optional; however, if you want to use it, the `\n` should be inside the quotes, since it's used to control the appearance of the output. It can't do that if it's not sent to `cout`. Without the `\n`, the user would type the answer to the question on the same line as the question; with the `\n`, the answer would be typed on the next line, as the `\n` would cause the active screen position to move to the next line at the end of the question.

Susan: OK, that is good, since I intended for the weight to be typed on a different line. Now I understand this much better. As far as why I didn't include the `\n` inside the quotes, I can't tell you other than the time of night I was writing or it was an oversight or a typo. I was following your examples and I am not a stickler for details type person.

Now that that's settled, I have another question: Is "return 0" the same thing as an ENTER on the keyboard with nothing left to process?

Steve: Sort of. It means that you're done with whatever processing you were doing and are returning control to the operating system (the C: prompt).

Susan: How does the program handle the ENTER? I don't see where it comes into the programs you have written. It just seems that at the end of any pause that an ENTER would be appropriate. So is the ENTER something that is part of the compiler that it just knows that by the way the code is written an ENTER will necessarily come next?

Steve: The `istream` input mechanism lets you type until you hit an ENTER, then takes the result up to that point.

One more point. We never tell the user that we have received the information. I've added that to your example.

Figure [morbass02ans](#) illustrates the compiler's output for that erroneous program:

Error messages from the erroneous weight program (code\morbass02.cc) (Figure morbas02ans)

```
morbass02.cc: In function `int main()':  
morbass02.cc:7: stray '\ ' in program  
morbass02.cc:11: parse error before `return'
```

And Figure [morbass03](#) shows the corrected program.

The corrected weight program (code\morbass03.cc) (Figure morbas03)

[code/morbass03.cc](#)

4. This was an offshoot of the previous question, which occurred when Susan wondered whether the program in Figure [morbass03](#) would terminate. Let's start from that point in the conversation:

Susan: Would this only run once? If so how would you get it to repeat?

Steve: We could use a `while` loop. Let's suppose that we wanted to add up all the weights that were entered. Then the program might look like Figure [morbass04](#).

The weight totalling program (code\morbass04.cc) (Figure morbas04)

[code/morbass04.cc](#)

In case you were wondering, the reason we have to duplicate the statements to read in the weight is that we

need an initial value for the variable `weight` before we start the `while` loop so that the condition in the `while` will be calculated correctly.

By the way, there's another way to write the statement `total = total + weight;` that uses an operator analogous to `++`, the increment operator: `total += weight;`. This new operator, `+=` operator, means "add what's on the *right* to what's on the *left*". The motivation for this shortcut, as you might imagine, is the same as that for `++`: It requires less typing, is more likely to be correct, and is easier to compile to efficient code. Just like the "increment memory" instruction, many machines have an "add (something) to memory" instruction, and it's easier to figure out that such an instruction should be used for an expression like `x += y` than in the case of the equivalent `x = x + y`. Let's see what Susan has to say about this notation:

Susan: Now I did find something that was very confusing. You say that `+=` means to "add what's on the right to what's on the left" but your example shows that it is the other way around. Unless this is supposed to be mirror imaged or something, I don't get it.

Steve: No, the example is correct. `total += weight;` is the same as `total = total + weight;`, so we're adding the value on the right of the `+=` (i.e., `weight`) to the variable on the left (i.e., `total`). Is that clearer now?

Susan: OK, I think I got it now, I guess if it were more like an equation, you would have to subtract `total` from the other side when you moved it. Why is it that the math recollection that I have instead of helping me just confuses me?

Steve: Because, unfortunately, the `=` is the same symbol used to mean "is equal to" in mathematics. The `=` in C++ means something completely different: "set the thing on the left to the value on the right".

Running this program normally isn't likely to give you much information, so you might want to run it under control of the debugger. You can do this in exactly the same way as you did the previous two programs, except that you would type `RHIDE morbas04` to start.

Footnotes

1. I realize I'm breaking a cardinal rule of textbooks: Never admit that the solution to a problem is anything but obvious, so the student who doesn't see it immediately feels like an idiot. In reality, even a simple program is difficult to get right, and indicating the sort of thought processes that go into analyzing a programming problem might help demystify this difficult task.
2. In order to use `vectors`, we have to `#include` the header file `vector.h`; otherwise, the compiler won't understand that type of variable.
3. A `vector` actually contains some additional information beyond the elements themselves. Unfortunately, how a `vector` actually works is too complicated to go into in this book.
4. Note that the `#include` statement for `vector.h` in Figure [vect1](#) uses `" "` rather than `<>` around the file name. The use of `" "` tells the compiler to search for `vector.h` in the current directory first, and then the "normal" places that header files supplied with the compiler are located. This is necessary because `vector.h` in fact is in the current directory. If we had written `#include <vector.h>`, the compiler would look only in the "normal" places, and therefore would not find

`vector.h`.

5. By the way, if you're wondering how to pronounce `Weight[i]`, it's "weight sub i". "Sub" is short for **subscript**, which is an old term for "index".
6. What I'm calling a *regular variable* here is technically known as a **scalar variable**; that is, one with only one value at any given time.
7. Here's an interesting side note on a case where the inventors of a commonly used facility should have used zero-based indexing, but didn't. We're still suffering from the annoyances of this one.

Long ago, there was no standard calendar, with year numbers progressing from one to the next, when January 1st came around. Instead, years were numbered relative to the reign of the current monarch; for example, the Bible might refer to "the third year of Herod's reign". This was fine in antiquity, when most people really didn't care what year it was. There were few retirement plans or 50th wedding anniversaries to celebrate anyway. However, it was quite annoying to historians to try to calculate the age of someone who was born in the fourth year of someone's reign and died in the tenth year of someone else's. According to Grolier's Multimedia Encyclopedia:

'About AD 525, a monk named Dionysius Exiguus suggested that years be counted from the birth of Christ, which was designated AD (anno Domini, "the year of the Lord") 1. This proposal came to be adopted throughout Christendom during the next 500 years. The year before AD 1 is designated 1 BC (before Christ).'

The encyclopedia doesn't state when the use of the term BC started, but the fact that its translation in English is a suspicious sign indicating that this was considerably later. In any event, this numbering system made matters considerably easier. Now, you could tell that someone who was born in AD 1200 and died in AD 1250 was approximately 50 years old at death.

Unfortunately, however, there was still a small problem. Zero hadn't yet made it to Europe from Asia when the new plan was adopted, so the new calendar numbered the years starting with 1, rather than 0; that is, the year after 1 BC was 1 AD. While this may seem reasonable, it accounts for a number of oddities of our current calendar:

1. Date ranges spanning AD and BC are hard to calculate, since you can't just treat BC as negative. For example, if someone were born in 1 BC and died in 1 AD, how old was that person? You might think that this could be calculated as $1 - (-1)$, or 2; however, the last day of 1 BC immediately preceded the first day of 1 AD, so the person might have been only a few days old.
2. The 20th century consists of the years 1901 to 2000; the year numbers of all but the last year of that century actually start with the digits *19* rather than *20*.
3. Similarly, the third millennium starts on January 1, 2001, not 2000.

The reason for the second and third of these oddities is that since the first century started in 1 AD, the second century had to start in 101 AD; if it started in 100 AD, the first century would have consisted of only 99 years (1-99), rather than 100.

If only they had known about the zero. Then the zeroth century would have started at the beginning of 0 AD and ended on the last day of 99 AD. The first century would have started at 100 AD, and so on; coming up to current time, we would be living through the last years of the 19th century, which would be defined as all of those years whose year numbers started with *19*. The second millennium

would start on January 1, 2000, as everyone would expect.

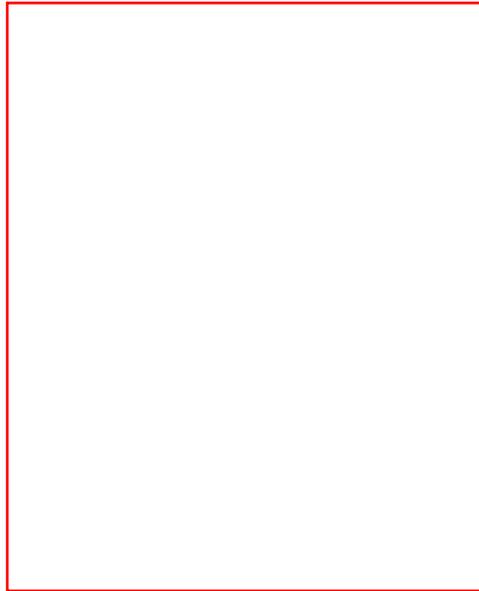
8. *Run-time efficiency* means the amount of time a program takes to run, as well as how much memory it uses. These issues are very significant when writing a program to be sold to or used by others, as an inefficient program may be unacceptable to the users.
9. I strongly recommend learning how to type (i.e., touch type). I was a professional programmer without typing skills for over 10 years before agreeing to type (someone else's) book manuscript. At that point, I decided to teach myself to touch-type, so I wrote a *Dvorak keyboard* driver for my Radio Shack Model III computer and started typing. In about a month I could type faster than with my previous two finger method and eventually got up to 80+ words per minute on English text. If you've never heard of the Dvorak keyboard, it's the one that has the letters laid out in an efficient manner; the "home row" keys are AOEUIDHTNS rather than the absurd set ASDFGHJKL;. This "new" (1930s) keyboard layout reduces effort and increases speed and accuracy compared to the old QWERTY keyboard, which was invented in the 1880s to prevent people from typing two keys in rapid succession and jamming the typebars together. This problem has been nonexistent since the invention of the Selectric typewriter (which uses a ball rather than type bars) in the 1960s, but inertia keeps the old layout in use even though it is very inefficient. In any event, since I learned to type, writing documentation has required much less effort. This applies especially to writing articles or books, which would be a painful process otherwise.
10. You may sometimes see the term *controlled statement* used in place of *controlled block*; since as we have already seen a block can be used anywhere that a single statement can be used, *controlled statement* and *controlled block* are actually just two ways of saying the same thing.
11. You don't need a space between the variable name and the ++ operator; however, I think it's easier to read this way.
12. By the way, the name C++ is sort of a pun using this notation; it's supposed to mean "the language following C". In case you're not doubled over with laughter, you're not alone. I guess you had to be there.
13. In case you're wondering why the value of `i` at the end of this loop will be 5, the reason is that at the end of each pass through the loop, the modification expression (`i ++`) is executed before the continuation expression that determines whether the next execution will take place (`i < 5`). Thus, at the end of the fifth pass through the loop, `i` is incremented to 5 and then tested to see if it is still less than 5. Since it isn't, the loop terminates at that point.
14. For that matter, what if someone types in a negative weight, such as -5? Of course, this doesn't make any sense, but it's a good idea to try to prevent errors, rather than assuming that users of a program will always act sensibly.
15. You may have noticed a slight oddity in this code. The block controlled by the `for` statement consists of exactly one statement; namely, the `if` that checks for a new `HighestWeight` value. According to the rules I've provided, that means we don't have to put curly braces (`{ }`) around it to make it a block. While this is true, long experience has indicated that it's a very good idea to make it a block anyway, as a preventive measure. It's very common to revisit old code to fix bugs or add new functions, and in so doing we might add another statement after the `if` statement at a later time, intending it to be controlled by the `for`. The results wouldn't be correct, since the added statement would be executed exactly one time after the loop was finished, rather than once each time through the loop. Such errors are very difficult to find, because the code looks all right when inspected casually; therefore, a little extra caution when writing the program in the first place often pays off handsomely.
16. Especially those whose names begin with "W".
17. *The C++ Programming Language, 2nd Edition*. v.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

[Return to the table of contents](#)

[Return to my main page](#)

What happened to this chapter?

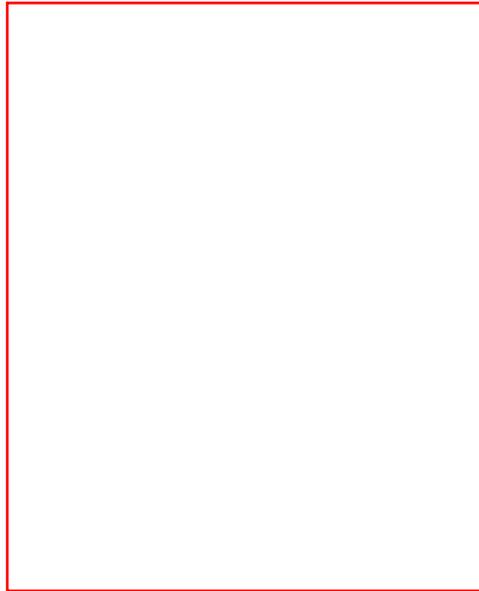
My publisher has decided to offer an electronic version of my new book, [Learning to Program in C++](#), a one volume version of my two *Who's Afraid of C++?* books, and understandably enough doesn't want me to compete with them by offering half of the new book free online.

Of course, I don't want to disappoint people who have heard about my free online books, either. So, as a compromise, I've left the first four chapters of *Who's Afraid of C++?* available on this site, accessible through the [index page](#).

I'm also offering the new book in printed form with **free** book rate shipping in the continental USA. You can order it by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

Where Am I, Anyway?

Now that you've reached the end of this book, there are some questions that have probably occurred to you. For example,

1. Am I a programmer now?
2. What am I qualified to do?
3. Where do I go from here?
4. Is that all there is to C++?

The answers to the first three of these questions, as usual with such open-ended topics, is "It all depends". Of course, I can give you some general answers; let's start with questions 1 and 2.

Yes, in the broadest sense, you are a programmer. You've read a fair amount of code and written some programs yourself. But, of course, this doesn't mean that you're a professional programmer. As I said way back at the beginning, no book can turn a novice into a professional programmer. Being a professional in any field takes a lot of hard work, and although you've undoubtedly worked hard in understanding this book, you've just begun the exploration of programming.

Questions 3 and 4 are also closely related. You now have enough background that you should be able to get some benefit from a well-written book about C++ that assumes you are already acquainted with programming; that would be a good way to continue. As for whether we've covered everything about C++, the answer is unequivocal: absolutely not. I would estimate that we have examined perhaps 5% of the very large, complicated, and powerful C++ language; however, that 5% is the foundation for the rest of your learning in this subject. Most books try to cover every aspect of the language and, as a result, cannot provide the deep coverage of fundamentals; I've worked very hard to ensure that you have the correct tools to continue your learning.

A Few Odds and Ends

I've skipped over some topics because they weren't essential to the discussion. However, since they are likely to be covered in any other book that you might read on programming in C++, I'll discuss them here briefly. This will ensure that they won't be completely foreign to you when you encounter them in your future reading.

Operator Precedence

You may recall from high school arithmetic that an expression like $5 + 3 * 9$, is calculated as though it were written $5 + (3 * 9)$, not $(5 + 3) * 9$; that is, you have to do the $*$ before the $+$, so that the correct result is 32, not 72, as it would be under the latter interpretation. The reason for performing the operations in the former order is that multiplication has a higher *precedence* than addition. Well, every operator in C++ also has a precedence that determines the order of application of each operator in an expression with more than one operator. This seems like a good idea at first glance, since after all, arithmetic does follow precedence rules like the one we just saw.

Unfortunately, C++ is just a little more complicated than arithmetic, and so its precedence rules are not as simple and easy to remember as those of arithmetic. In fact, there are 17 different levels of precedence, which no one can remember. Therefore, everyone (or at least, everyone who is sensible)

ends up using parentheses to specify what order was meant when the expression was written; of course, if we're going to have to use parentheses, then why do we need the precedence rules in the first place?

Other Native Data Types

We've confined our use of native data types to `short`, `unsigned short`, `char`, `int` (for the return type of `main` only), and `bool`. As I mentioned in Chapter [inventor.htm](#), there are other native types; you'll be seeing them in other programs and in other textbooks, so I should tell you about them now. By the way, I haven't avoided them because they're particularly difficult to use; the reason is simply that they weren't necessary to the task at hand, which was teaching you how to program, using C++. Now that we have accomplished that task, you might as well add them to your arsenal of tools. These other native types are

1. `float`
2. `double`
3. `long` (signed or unsigned)

The `float` and `double` types are used to store values that can contain fractional parts, (so-called *floating-point* numbers), rather than being restricted to whole numbers as in the case of `short` and the other integral types. Of course, this raises two questions: First, why don't we use these types all the time, if they're more flexible? Second, why are there two of these types rather than only one? These questions are related, because the main difference between `float` and `double` is that a `float` is 4 bytes long and a `double` is 8 bytes long; therefore, a `double` can store larger values and maintain higher accuracy. However, it also uses up twice the amount of memory of a `float`, which may not be important when we're dealing with a few values but is quite important if we have a `vector` or array of thousands or tens of thousands of elements.

So that explains why we'd use a `float` rather than a `double`, but not why we would use a `long` rather than a `float`; after all, they both take up four bytes. The reason that we would use a `long` is that it can store larger whole values than a `float` while retaining exact accuracy in results. Also, on a machine that doesn't have a built-in numeric processor, `longs` can be processed much more rapidly than than `floats`.

protected Species

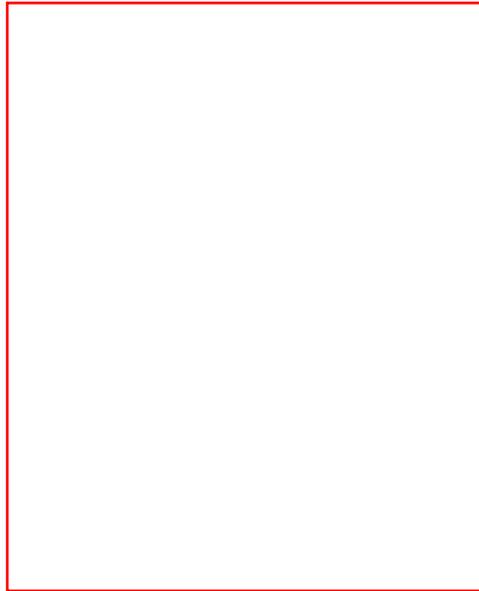
The `protected` keyword is another access specifier, like `public` and `private`. The reason that we haven't covered it in this book is that it is not used except in situations where we create a `class` that is "descended" from another class. This topic is sufficiently complex as to be the subject of several chapters in the sequel to this book, *Who's Afraid of More C++?*, and is therefore not covered in this book due to space limitations.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)



Who's Afraid of C++? - the WWW version

ISBN: 0-12-339097-4

Copyright 1997, 1996 by AP Professional

Copyright 2000, 1997, 1996 by Chrysalis Software Corporation

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)

About the Author

Steve Heller had always been fascinated by writing. In his childhood days in the 1950s and 1960s, he often stayed up far past his bedtime reading science fiction. Even in adulthood, if you came across him in his off-hours, he was more likely to be found reading a book than doing virtually anything else.

After college, Steve got into programming more or less by accident; he was working at an actuarial consulting firm and was selected to take charge of programming on their time-sharing terminal, because he was making much less than most of the other employees. Finding the programming itself to be more interesting than the actuarial calculations, he decided to become a professional programmer.

Until 1984, Steve remained on the consuming side of the writing craft. Then one day he was reading a magazine article on some programming-related topic and said to himself, "I could do better than that". With encouragement from his wife of the time, he decided to try his hand at technical writing. Steve's first article submission -- to the late lamented *Computer Language Magazine* -- was published, as were a dozen more over the next ten years.

But although writing magazine articles is an interesting pastime, writing a book is something entirely different. Steve got his chance at this new level of commitment when Harry Helms, then an editor for Academic Press, read one of his articles in *Dr. Dobb's Journal* and wrote him a letter asking whether he would be interested in writing a book for AP. He answered, "Sure, why not?", not having the faintest idea of how much work he was letting himself in for.

The resulting book, *Large Problems, Small Machines* received favorable reviews for its careful explanation of a number of facets of program optimization, and sold a total of about 20,000 copies within a year after publication of the second edition, entitled *Efficient C/C++ Programming*.

By that time, Steve was hard at work on his next book, *Who's Afraid of C++*, which is designed to make object-oriented programming intelligible to anyone from the sheerest novice to the programmer with years of experience in languages other than C++. To make sure that his exposition was clear enough for the novice, he posted a message on CompuServe requesting the help of someone new to programming. The responses included one from a woman named Susan, who ended up contributing a great deal to the book; in fact, about 100 pages of the book consist of email between Steve and Susan. Her contribution was wonderful, but not completely unexpected.

What *was* unexpected was that Steve and Susan would fall in love during the course of this project, but that's what happened. Since she lived in Texas and he lived in New York, this posed some logistic difficulties. The success of his previous book now became extremely important, as it was the key to Steve's becoming a full-time writer. Writers have been "telecommuting" since before the invention of the telephone, so his conversion from "programmer who writes" to "writer" made it possible for him to relocate to her area, which he promptly did.

Since his move to Texas, Steve has been hard at work on his writing projects, including *Introduction to C++*, a classroom text that covers more material in the same space as *Who's Afraid of C++?* at the expense of the email exchanges in the latter book, followed by *Who's Afraid of Java?*. Their latest project is *Who's Afraid of More C++?*, the sequel to this book, which came out in July of 1998. Steve's advanced algorithms book, *Optimizing C++*, was also published in that month.

Steve and Susan were married in 1997.

[Comment on this book!](#)

I'm very pleased to announce the publication of a new one-volume version of my two *Who's Afraid of C++?* books, under the title of *Learning to Program in C++*. You can order this book directly from me for only \$44.95, including free book rate shipping in the continental USA, by clicking [here](#).

[Return to the table of contents](#)

[Return to my main page](#)