

Programming in C++

Rex Jaeschke

Programming in C++

© 1998, 2001, 2005, 2007–2008, 2009 Rex Jaeschke.

Edition: 3.0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means whatsoever, except in the case of brief quotations embodied in critical reviews and articles.

The information in this book is subject to change without notice. It should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and the publisher assume no responsibility for errors or omissions.

Java is a trademark of Sun Microsystems.

Visual C++ is a trademark of Microsoft.

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Much of the information in this book has been published previously, as follows:

1. A series of monthly columns called "Let's C Now" in *DEC Professional*, Professional Press, running from 1990–1993.
2. The book "C++: An Introduction for Experienced C Programmers", Rex Jaeschke, CBM Books 1993, ISBN 1-878956-27-2.

For the C language and library subset of C++, the reader may wish to refer to my book, "The Dictionary of Standard C", Prentice Hall 2001, ISBN 0-13-090620-4. This dictionary can be viewed on-line, free of charge, at www.prenhall.com/jaeschke. An earlier version of this dictionary, now out of print, was published by Professional Press Books 1991, ISBN 0-07-707657-5.

Preface	xi
Introduction	xi
Reader Assumptions	xii
Limitations.....	xiii
Presentation Style	xiii
Exercises and Solutions	xiii
Program Behavior	xiv
The Status of C++	xiv
Acknowledgments.....	xv
1. The Basics	1
1.1 Basic Program Structure	1
1.2 Identifiers.....	4
1.3 Introduction to Formatted I/O	5
1.4 The Data Types	9
1.4.1 Arithmetic Types.....	9
1.4.2 The Boolean Type	13
1.4.3 Enumerated Types.....	14
1.4.4 User-Defined Types	16
1.4.5 The string Type	16
1.5 Literals.....	17
1.5.1 Integer Literals.....	17
1.5.2 Character Literals.....	19
1.5.3 Floating-Point Literals.....	19
1.5.4 Boolean Literals	20
1.5.5 String Literals	20
1.6 Type Qualifiers.....	20
1.6.1 The const Qualifier	20
1.6.2 The volatile Qualifier.....	21
1.7 Type Synonyms	21
1.8 References	22
1.9 Automatic Variables.....	24
1.10 Operator Precedence.....	26
1.11 Type Conversion	28
2. Looping, Testing, and Branching	31
2.1 The while Statement	31
2.2 The for Statement.....	38
2.3 The do/while Statement.....	41
2.4 The if/else Statement.....	41
2.5 The break Statement	44
2.6 The continue Statement.....	45
2.7 The switch Statement	46
2.8 Selection and Loop Conditions	48
2.9 The goto Statement.....	49
2.10 The Null Statement.....	50
3. Arrays and Strings	53
3.1 Introduction	53
3.2 Initialization	55
3.3 C-Style Strings	56

Programming in C++

3.4	Array Manipulation.....	60
3.5	String Manipulation	62
3.6	The vector Class	67
3.7	The sizeof Operator	68
3.8	Character Testing and Conversion.....	70
4.	Functions	75
4.1	Introduction	75
4.2	Type-Safe Linkage	78
4.3	Argument Passing	79
4.3.1	Passing by Value	79
4.3.2	Passing by Reference	81
4.3.3	Passing by Address	83
4.3.4	Default Argument Values	85
4.3.5	Passing No Arguments.....	85
4.3.6	Variable-Length Argument Lists	86
4.3.7	Argument Checking and Conversion	87
4.3.8	Naming Parameters.....	89
4.3.9	sizeof and Array Parameters.....	89
4.4	Function Return	91
4.4.1	Returning by Value	91
4.4.2	Returning by Reference	91
4.4.3	Returning by Address	93
4.4.4	No Return Value	93
4.5	Recursion	93
4.6	Logical and Bit Operations.....	96
4.7	Function Overloading	100
4.8	Inline Functions.....	102
4.9	Template Functions	104
5.	Storage Classes	111
5.1	Introduction	111
5.2	Storage Duration.....	111
5.3	Scope.....	113
5.4	Linkage.....	114
5.5	The auto Storage Class	114
5.6	The register Storage Class	116
5.7	The static Storage Class.....	117
5.7.1	Statics Having No Linkage.....	117
5.7.2	Statics Having Internal Linkage.....	119
5.7.3	Statics Having External Linkage	121
5.8	Global Variables and the extern Storage Class	121
5.9	Public and Private Functions.....	123
5.10	Storage Class Summary.....	125
6.	Pointers and Addresses	127
6.1	Introduction	127
6.2	A Conceptual Model of Memory	127
6.3	Pointers as an Abstraction Tool	131
6.3.1	Context-Dependent Programs.....	132
6.3.2	Sorting and Searching.....	134

6.3.3	Lists.....	136
6.4	Using Addresses.....	136
6.5	Using Pointers.....	138
6.5.1	Pointers and <code>const</code>	147
6.6	Pointer Arithmetic.....	149
6.7	Functions That Return Pointers.....	156
6.8	Subscripting and Pointer Operations.....	159
6.9	Arrays of Pointers.....	162
6.10	Accessing Command-Line Arguments.....	166
6.11	Dynamic Memory Allocation.....	168
6.12	Generic Pointers.....	171
6.13	Pointers to Functions.....	174
6.13.1	Introduction.....	174
6.13.2	Overloaded Functions.....	179
6.13.3	Inline Functions.....	181
6.13.4	Template Functions.....	181
6.14	Common Pointer Problems.....	182
7.	Structures, Bit-Fields, and Unions.....	185
7.1	Introduction.....	185
7.2	Structure I/O.....	188
7.3	Nested Structures.....	190
7.4	Structure Initialization.....	192
7.5	Manipulating Structures as a Whole.....	193
7.6	Layouts without Tags.....	194
7.7	Arrays of Structures.....	195
7.8	Pointers to Structures.....	198
7.9	Linked Lists.....	203
7.9.1	Singly Linked Lists.....	204
7.9.2	Doubly Linked Lists.....	205
7.9.3	Circular Lists.....	206
7.9.4	<i>n</i> -link Lists.....	208
7.9.5	Dynamically Managed Lists.....	208
7.10	Mutually Referential Lists.....	211
7.11	Structure Member Alignment.....	211
7.12	Bit-Fields.....	215
7.13	Unions.....	218
8.	Introducing the Standard Class Libraries.....	225
8.1	Introduction to Classes.....	225
8.2	Parent Headers.....	226
8.3	Constructors and Destructors.....	226
8.4	Operator Functions.....	228
8.5	Member Functions.....	229
8.6	Other Members.....	230
9.	The <code>string</code> Class.....	233
9.1	Introduction.....	233
9.2	Constructing Strings.....	233
9.3	Accessing String Elements.....	235
9.4	Assignment Operations.....	236

Programming in C++

9.5	Converting to C-Style Strings	237
9.6	Comparison Operations	238
9.7	Insertion Operations	239
9.8	Concatenation Operations.....	239
9.9	Search Operations	240
9.10	Replacement Operations	240
9.11	Extracting Substrings	241
9.12	Size and Capacity	241
9.13	I/O Operations	241
9.14	Internationalization and Wide Strings	242
9.15	Getting Along with Other String Classes.....	242
9.16	Implementation Details	242
9.17	A Summary of Class string	243
10.	Input and Output	249
10.1	Format Control.....	249
10.2	I/O Member Functions	256
10.3	File I/O	257
10.4	String Encoding and Decoding	261
10.5	User-Defined Manipulators	263
11.	Classes and Objects	269
11.1	Introduction	269
11.2	Data Hiding	270
11.3	Encapsulation.....	272
11.4	More on Member Functions.....	279
11.4.1	Argument Passing Machinery.....	279
11.4.2	Inline Member Functions	281
11.4.3	Overloading Member Functions.....	285
11.4.4	Default Argument Values	287
11.5	Class Members versus Instance Members	288
11.6	Relaxing Member Access Restrictions	293
11.7	Classes versus Structures and Unions.....	298
12.	Object Creation and Destruction.....	299
12.1	Introduction	299
12.2	Temporary Objects	302
12.3	Arrays and Nested Objects	303
12.4	The Effects of Changing Control Flow.....	305
12.5	Static Data and Execution Order.....	306
12.6	Dynamically Allocated Objects	309
12.7	Copy Constructors	310
12.8	Conversion by Constructor	312
12.9	Private Constructors	314
12.10	ctor Initializers	317
13.	Operator Overloading	323
13.1	Introduction	323
13.2	A Simple Example	324
13.3	Overloading I/O	326
13.4	Class IntVector Revisited.....	331
13.5	Overloading Assignment.....	333

13.6	Overloading Subscripting.....	334
13.7	Overloading Binary Operators	336
13.8	Overloading Unary Operators.....	338
13.9	Class-Specific Versions of <code>new</code> and <code>delete</code>	340
13.10	Conversion Functions	342
13.11	Some Rules of Thumb	343
14.	Inheritance	345
14.1	Introduction	345
14.2	The Is-A versus Has-A Test.....	347
14.3	A Simple Class Hierarchy.....	350
14.4	Abstract Classes	360
14.5	Interfaces	364
14.6	Protected Members.....	365
14.7	Public, Protected, and Private Inheritance	367
14.8	A Universal Base Class	369
14.9	Multiple Inheritance	369
14.10	Virtual Base Classes	370
14.11	Inheritance after the Fact.....	371
14.12	Miscellaneous Issues	372
15.	Exception Handling	373
15.1	Introduction	373
15.2	The Basic Machinery.....	374
15.3	Throwing an Exception	376
15.4	Derived Exception Types.....	381
15.5	Unwinding the Stack.....	382
15.6	Ensuring Resource Deallocation	384
15.7	Exception Specifications	390
15.8	Final Note.....	393
16.	Class Templates	395
16.1	Introduction	395
16.2	Explicit Template Instantiation	398
16.3	Non-Parameterized Template Arguments.....	400
16.4	Templates and Friends.....	403
16.5	Miscellaneous Topics	404
17.	Name Spaces	407
17.1	Introduction	407
17.2	Name Space Pollution.....	407
17.3	The Namespace Machinery	407
17.4	Nested Namespaces	410
17.5	Long Names and Namespace Aliases.....	411
17.6	Adding to a Namespace	413
17.7	The Standard Headers and <code>using</code> Directives.....	414
17.8	The <code>using</code> Declaration.....	415
17.9	File Scope Statics.....	417
18.	The Preprocessor	419
18.1	Introduction	419
18.2	Macros	419

Programming in C++

18.2.1	Object-Like Macros.....	420
18.2.2	Function-Like Macros	428
18.2.3	Defining Macros at Compile Time	436
18.3	Headers.....	436
18.4	Conditional Compilation	438
18.5	Miscellaneous Directives	442
18.5.1	The #pragma Directive	442
18.5.2	The #error Directive.....	442
18.5.3	The #line Directive.....	443
18.5.4	The # Directive.....	443
18.5.5	The #undef Directive.....	443
18.6	Predefined Macros	444
19.	Sundry Issues	447
19.1	The Comma Operator	447
19.2	Pointers to Arrays	449
19.2.1	Introduction.....	449
19.2.2	Dynamic Allocation of Multidimensional Arrays	452
19.2.3	An Abstraction Tool	452
19.3	Mastering Declarations.....	454
19.3.1	Introduction.....	454
19.3.2	Basic Types.....	454
19.3.3	Deriving From a Basic Type.....	455
19.3.4	Deriving from Derived Types	455
19.3.5	Precedence of Punctuators	456
19.3.6	Forcing Punctuator Precedence	457
19.3.7	Writing Declarations.....	458
19.3.8	Reading Declarations.....	460
19.3.9	Using Type Information	461
19.4	Sequence Points.....	462
19.4.1	Introduction.....	462
19.4.2	Full Expressions	465
19.4.3	Sequence Point Operators.....	467
19.4.4	Conclusion	469
19.5	Lvalues	469
19.5.1	Introduction.....	469
19.5.2	Two Kinds of Lvalues	470
19.5.3	Operators That Generate Lvalues.....	471
19.5.4	Operators That Need Modifiable Lvalues.....	471
19.6	Variable-Length Argument Lists	472
19.6.1	Introduction.....	472
19.6.2	Implementing a Maximum Function	474
19.6.3	The va_* Routines	477
19.7	Mutable Data Members	478
19.8	Run-Time Type Information.....	481
19.9	Pointers to Class Members	487
Annex A.	Operator Precedence	495
Annex B.	Standard Run-Time Library.....	499
B.1	Standard C++ Library Headers	499

B.2 Standard C Library Headers Supported by Standard C++	500
Index	501

Preface

Introduction

Welcome to the world of C++. Throughout this book, we look at the statements and constructs of the C++ programming language. Each statement and construct is introduced by example with corresponding explanations, and, except where errors are intentional, the examples are complete programs or subroutines that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book was written with teaching in mind, and evolved from an earlier series on C. It is intended for use both in a classroom environment as well as for self-paced learning.

C++ is a general-purpose, high-level language that supports object-oriented programming (OOP). From a grammatical viewpoint, C++ is a relatively simple language having about 75 keywords. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable functions while still maintaining portability, there is little need for language extensions especially considering the fact that C++ supports calls to procedures written in other languages. However, extensions do exist.

Depending on their language backgrounds, programmers new to C++ may initially find programs hard to read because, traditionally, most C++ code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, C++ language keywords must be written in lowercase. Keywords are also reserved words.

If you have ever wondered what all those special punctuation marks on most keyboards are for, the answer may well be "to write C++ programs!" C++ uses almost all of them for one reason or another and sometimes combines them for yet other purposes.

C++ supports a structured, modular approach to programming using callable subroutines, called *functions*. Source files can be compiled separately, with external references being resolved at runtime.

C++ lends itself to writing terse code. However, there is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. In fact, that's what you will get by default. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. However, good code doesn't happen automatically—you have to work at it. Throughout the book, I make numerous comments and suggestions regarding style. Perhaps the best advice I can give in that regard is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

Tip: Avoid initializing enumerators explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

C++ is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and BASIC, for example, will do just fine. In any event, compared to other high-level languages (including Java and C#), C++ is a relatively expensive language to learn and master.

Reader Assumptions

This is *not* a first course in programming. (Nor do I recommend C++ as a first programming language.)

I assume that you know how to use your particular text editor, C++ compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the C++ programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker
- Number system theory
- Bit operations such as AND, inclusive-OR, exclusive-OR, complement, and left- and right-shift
- Data representation
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables
- Use of single- and multi-dimensional arrays
- Creation and use of sequential files and how to do formatted and unformatted I/O
- Basic data structures such as linked lists

Many of C++'s more powerful capabilities, particularly exception handling and inheritance, require advanced programming experience before they can be understood and exploited fully.

Although C++ is essentially a superset of C, you do *not* need to know C to use this book. If you do know C, simply read quickly over those sections that seem familiar to you. I say, "Read quickly" rather than "skip" because you may well find that C++ defines things more fully or a little differently than does C. (There are also a few incompatibilities.)

If you know C, you have all the OO stuff to learn. If you know some OO language other than C++, you'll already be familiar with the OO concepts, but not the syntax that is largely common to C, C++, C#, and Java. And if your programming background is in some procedural language such as PL/I, Pascal, FORTRAN, or BASIC, you'll need to

read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of functions and their associated argument passing and value returning.

Limitations

This book covers almost all of the C++ language. It also introduces the core class library. However, a very small percentage of library facilities are mentioned or covered in any detail. The Standard C++ library contains so many classes and functions that whole books have been written about that subject alone.

This book is directed at teaching the C++ language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced solutions.

GUI, calling non-C++ routines, threading, inter-process communications, operating system-specific, and a number of other advanced features are outside the scope of this text.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming-language training courses for some 14 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and well focused. Specifically, I introduce the basic elements and constructs of the language using procedural programming examples. Once those fundamentals have been mastered, I move on to object-oriented concepts and syntax. Then come the more advanced language topics and library usage. Many books on object-oriented languages use objects, inheritance, exception handling, GUI, and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other is; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named `Source`, where each chapter has its own subdirectory, within which each program has its own subdirectory. For example, the source code for the program called `ba04` can be found in the following fully qualified directory path: `Source, Basics, ba04`. By convention, the names of C++ source files end in `.cpp`.

Each chapter contains exercises, some of which have the character `*` following their number. For each exercise so marked, a solution is provided electronically in a directory tree named `Labs`, where each chapter has its own

Programming in C++

subdirectory, within which each program has its own subdirectory.¹ For example, lab solution lbcl01.cpp has the following fully qualified name: Labs, Classes, lbcl01, lbcl01.cpp.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See lab directory xx.)" This indicates the corresponding solution in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Program Behavior

According to the C++ Standard, for correct, well-formed programs, almost all behaviors are well defined and predictable. However, in certain cases, an implementation can choose the behavior it deems best, and this behavior need not be the same as that exhibited by other implementations. Such behavior is called *implementation-defined*, and must be documented by each implementation. For example:

Implementation-Defined Behavior: The range of values that can be stored in an object of a given arithmetic type.

In other cases, an implementation can choose whatever behavior it wants at that time *without* needing to reproduce or document that choice. Such behavior is called *unspecified*. For example:

Unspecified Behavior: The order of evaluation of the function designator and its argument list, and the order of evaluation of arguments within the list.

A third category is *undefined behavior*, which can result from situations for which the standard imposes no requirements or is otherwise silent. For example:

Undefined Behavior: Subscripting a string with an out-of-bounds index.

Clearly, one must be aware of implementation-defined and unspecified behaviors when writing code that is to be ported across different platforms. And one should always avoid relying on undefined behavior.

The Status of C++

The first ANSI/ISO standard for C++ was adopted in September 1998. Its official designation is ISO/IEC 14882. A slightly improved version was issued in 2003. Electronic copies of this standard can be purchased for \$18. (For more information, see www.ansi.org or www.iso.ch.)

¹ The solutions are only available to licensees of these materials when they are used in formal training scenarios.

The C++ standards committee, ISO/IEC JTC 1/SC 22/WG 21, is now working on various maintenance issues, Technical Reports (TRs), as well as a revision.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C++ seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, September 2009

1. The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements.

1.1 Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. C++ has five different kinds of tokens: keywords, identifiers, literals, operators, and punctuators.

For the most part, C++ is a free-format language, with space between tokens being optional. However, in some cases, some kind of separator is needed between tokens so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive characters from the set: space, horizontal tab, vertical tab, form-feed, and *new-line* (entered by pressing the RETURN or ENTER key). In a source file, an arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, and after the last token.

Let's look at the basic structure of a C++ program that writes a welcome message to the console (see directory ba01):

```
/*-
This program is a simple example of C++.
-*/

/*1*/  #include <iostream>

int main()          /* start of the program */
{                  // beginning of function body

/*2*/  std::cout << "Welcome to C++\n";

/*3*/  return 0;    // terminate with success
}                  // end of a function body
```

A *delimited comment* is one surrounded by `/*` and `*/`, and can be used on part of a source line, as a whole source line, or it can span any number of source lines. A *line-oriented comment* begins with `//` and continues until the end of its source line. A comment of either kind is treated as a single space. A delimited comment can occur anywhere white space can be used. A line-oriented comment can only appear at the end of a source line. Although comments of the same kind do not nest, each kind of comment can be used to disable a source line containing the other.¹

¹ Refer to §18.4 to see a novel way to "comment out" a source line containing a comment.

Programming in C++

A C++ program consists of one or more *functions* that can be defined in any order in one or more source files (or *translation units*, as Standard C++ calls them). A program must contain at least one function, called `main`, and this function's name must be spelled using lowercase letters.¹ This specially named function indicates where the program is to begin execution.²

The parentheses following the function name `main` surround the function's *parameter list*.³ The parentheses are required, even if no arguments are expected.

The body of a function is enclosed within a matching pair of braces. All executable code must reside within the body of some function or other. Statements are executed in sequential order unless branching or looping statements dictate otherwise. A program terminates when it returns from `main`, either by dropping into the closing brace of that function—which acts as an implicit `return` statement—or via an explicit `return` statement. According to the C++ standard, the explicit use of `return 0` in case 3⁴ above is no longer necessary; however, at the time of writing, many compilers continue to issue a warning if this statement is omitted.

We'll learn about the `return` statement in §1. However, why return a value from `main`? Many programs either are invoked at the operating system's command-line level or are spawned by another program. In either case, when a program terminates, it has the ability to return one piece of information to the program that invoked it. Perhaps it returns a status code or a count of the number of transactions processed. We refer to this returned value as the program's *exit status code*. The value used and its meaning are the business of the programmer. Unless otherwise stated, all `main` functions in this book contain the statement `return 0;`. The value zero has no special significance, although on some systems it is interpreted as some form of success code.

Case 1 provides access to the I/O library, and case 2 outputs the welcome text to the console. These cases are discussed in detail in §1.3.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

We can write any given correct program in an infinite number of ways simply by using different amounts and kinds of white space (including comments). Obviously, there are very good, overt styles and very bad, cryptic styles. Then there is the large and subjective gray area in between.

¹ While mixed- or uppercase spellings of `main` might be recognized by some systems, this is undefined behavior.

² Some environments require a different entry-point name. For example, certain Microsoft Windows programs begin execution at `winmain` instead.

³ A function uses parameters to declare what it is expecting to be passed, via an *argument list*, at run time.

⁴ Throughout this book, many code examples contain source lines with leading delimited comments of the form `/*n*/`, where *n* is a number. Such lines are referred to as *cases*, with `/*1*/` being case 1, `/*2*/` being case 2, and so on.

Style Tip: Be overt; pick a style that isn't too far outside the mainstream, and stick with it. Above all, be consistent. And remember, the style you develop when learning a language is the one with which you will likely stay, so give some thought to programming style from the outset.

The style used in this book is very common. If you choose to adopt a different style, note that no matter how great you think your particular style is, every minute you spend arguing about whose style is superior and why—or worse yet, changing other people's code to your style—is generally time wasted. If no corporate-wide or group style guide exists, and the people on a multi-person project cannot agree on a common style, the project manager should dictate one.

Exercise 1-1*: Compile and link the empty program shown above, and look at the size of the resulting executable file on disk; it may be surprisingly large. Look at the listing produced by the linker and see what goes into a seemingly "empty" program. (See lab directory lbba03.)

Exercise 1-2*: Try compiling, linking, and executing a main program whose name is something like Main or MAIN. If either of these works, use xyz instead and try again. (See lab directory lbba04.)

Exercise 1-3*: Using `/*` and `*/`, comment out a block of code that already contains this kind of comment, and see how your compiler reacts. (Some compilers actually do support nested comments as an extension.) (See lab file lbba05.)

Exercise 1-4*: What happens if you leave off the closing `*/` from a comment? (See lab file lbba06.)

Let's move on to the more common case of a program having multiple functions (see directory ba02):

```
/* C++ program with two functions */

void compute()
{
    // ...
    /*1*/ return;      // redundant statement
}

int main()
{
    /*2*/ compute();   // call function 'compute'

    return 0;
}
```

C++ supports modularization via functions. A source file can contain one or more functions, defined in any order. (That said, it is no accident that function `compute` is defined before function `main` in this example. In §1, we'll learn how to define functions in any order as well as how to put them in separate source files.) Unlike some languages, in C++, function definitions cannot be nested. That is, each function's definition must be outside the

6. Pointers and Addresses

So far, we have learned how to do things in C++ that we already know how to do in some other language. In this chapter, we will be exposed to those capabilities of C++ that make it different from older higher-level languages.

6.1 Introduction

Without a doubt, a difficult part of C and C++ to master is pointers. For most high-level language programmers, the idea of using addressing directly and defining variables—called *pointers*—that contain addresses is quite foreign. And even after these programmers come to grips with the idea, they often have considerable difficulty in dealing with the syntax.

To many programmers, pointers seem unnecessary. After all, they have implemented thousands of lines of production code over many years and they have never had the need to even know about, let alone use, pointers. Why then must they master them to use C++? That is a reasonable question and one that we will address.

6.2 A Conceptual Model of Memory

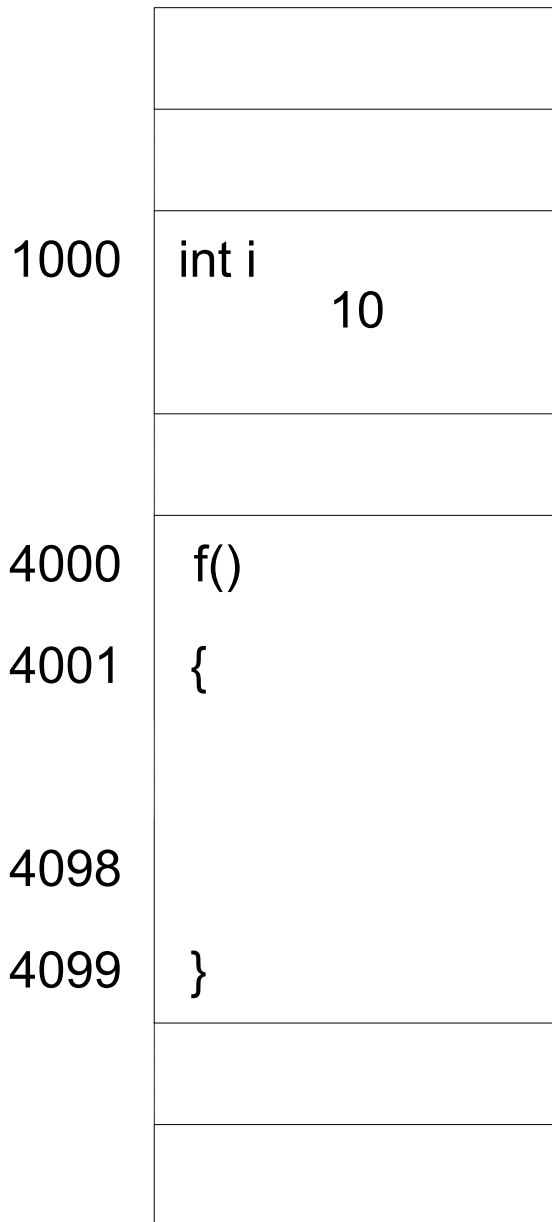
Before we can talk much about addressing and pointers, we need a conceptual model of how memory is organized when a program is loaded for execution. Consider a program that contains the following fragment:

```
void f()
{
    static int i = 10;

    statement(s)
}
```

When this program is loaded into memory for execution, Figure 6–1 indicates how things might be organized, with regard to the variable *i* and the function *f*. (The exact way in which a system represents this may vary. However, the concepts suggested by this model still apply.)

Figure 6–1: Code and Data Mapping



Memory is organized into an array of cells, with each cell having a unique location, which we shall call an *address*. On machines whose memory is *byte-addressable*, each byte has its own address. On machines that are *word-addressable*, each word has its own address. In the latter case, bytes do not physically exist; characters must be packed into words, and the address of a packed character is the address of its parent word and the character offset within that word.

Many machines use addresses that really are unsigned integers. Others use addresses that are signed integers. Still others use a segmented address that is made up of two parts: a base address and an offset. For the purposes of our discussion, we will make no assumptions about how physical memory really is addressed on any particular machine; that is unimportant in our model.

Function `f` contains a `static int` object, `i`. Being static, this variable typically is allocated memory by the compiler and/or linker. In any event, memory is allocated for it before `main` begins execution. `i` occupies one or more consecutive locations in memory, starting at address 1000. We say then that the address of `i` is 1000. (The addresses used in this discussion have been picked arbitrarily. For actual memory placement details, use your debugger or consult the listing file produced by your linker.)

Once the compiler and linker have done their job, all variable and function names have disappeared, and all references to them have been reduced to references to their corresponding addresses. And yes, even functions have addresses. In Figure 6–1, the machine instructions for the executable code in function `f` were allocated memory locations 4000–4099.

The reason for all this is that a CPU is a very simple-minded machine. It doesn't know anything about abstractions such as variables and functions. What it does know is how to go to a specific address in memory to get or put a single- or multibyte object. It also knows how to branch or jump to an address and begin executing instructions at that location. Therefore, the principal job of a compiler and linker, in combination, is to reduce source code to an ordered set of machine instructions, many of which deal with addresses.

Ultimately, the CPU deals with addresses. However, that doesn't necessarily mean we as programmers need to deal with them. In fact, most languages provide no way to deal with addresses directly. However, C++ does, and while we can try to avoid using them directly, addresses are produced and used all over the place in things as fundamental as uses of `cin` and `cout`. There is no escaping the concept of addresses in C++; for better or worse, addresses simply are an integral part of C++ programming.

As discussed above, objects and functions are referenced via their addresses. And if we change our code in certain ways, when next we compile and link, these objects and functions may be located at different addresses. Therefore, we are rarely, if ever, interested in the particular address at which something is located. In fact, the addresses used during any particular execution of a program are relative to that execution only. Consider the case of an automatic variable. It is allocated memory at run time, typically on some stack. If we call its parent function from several different places, it is almost certain that for each of its lifetimes, the variable will live at different locations. (This would certainly be true for automatic variables defined in functions that are called recursively.)

If addresses can easily change during execution or over the life of an application, why are we interested in them? If we know where something lives, we can remember that information and, later on, access it indirectly through that saved address. Accessing objects and functions in this manner provides a very powerful abstraction tool, and one that we will explore in the next section.

In Figure 6–2, our memory model has been extended to where we can save an object's address. The address of `i` is 1000. If we can find out `i`'s address, we can use it to access `i` indirectly. We might also want to save that address in memory for future reference. We do this by creating a variable capable of storing the address of `i`. Let us call it `pi`, for "pointer to `int`". Let us assume that the pointer variable is located at address 1100.

11. Classes and Objects

Classes are the main concept underlying the object-oriented nature of C++. Simply stated, a *class* is a special kind of structure.

In this chapter, we will learn about data hiding and encapsulation. *Data hiding* is the process whereby implementation details of a class can be hidden from general access. *Encapsulation* involves the syntactic association of data and the functions that have permission to operate on it.

11.1 Introduction

Given what we learned about structures in §1, one problem is that any code that can access a structure object as a whole can also access all its members. The problem then, becomes one of a lack of discipline. If we can get at any member of a structure in scope, we do so, generally in a rather haphazard way. That is, the way in which we interface with such objects is not controlled, making debugging and maintenance more difficult. If we could limit the ways in which objects could be accessed, it would be much easier to debug cases in which these objects are inadvertently overwritten. It would also be easier to maintain the code, as we would need to look at only those functions having access in order to learn about the underlying object.

A class permits us to partition the members in an object of that class into two main groups:¹ private and public. We make members public by using the *public access specifier* keyword, as shown in the class definition for `CircleB` in the following example (see directory `cl01`):

```
// CircleA.h

struct CircleA
{
    int xorigin;
    int yorigin;
    float radius;
};

// CircleB.h

class CircleB
{
public:
    int xorigin;
    int yorigin;
    float radius;
};
```

¹ A third category, protected, is discussed in §1.

Programming in C++

All members in class `CircleB` are below the `public` access specifier, so they are public, and can be accessed in the same way as the members in the structure `CircleA`. Without the explicit access specifier, the members would be private; however, one of the main points of object-oriented design is to make class data members private, so access to them is restricted.

In the example above, and in almost all other examples throughout this book, the class definition is placed in its own header having the same name. For example, class `Point` would be defined inside the user-defined header `Point.h`.

For completeness, here is an example (see directory `cl01`) that uses these layouts:

```
#include "CircleA.h"
#include "CircleB.h"

void test()
{
    CircleA ca;

    ca.xorigin = 10;
    ca.yorigin = 20;
    ca.radius = 1.5f;

    CircleB cb;

    cb.xorigin = 200;
    cb.yorigin = 50;
    cb.radius = 3.67f;
}
```

11.2 Data Hiding

Using the following example (see directory `cl02`), let's see the implications of making a member private:

```
// Circle.h

class Circle
{
    int xorigin;
public:
    int yorigin;
private:
    float radius;
};
```

```

#include "Circle.h"

void f(Circle *p);

int main()
{
    Circle c;

    /*1*/  c.xorigin = 4;           // error, member is private
    /*2*/  c.yorigin = 5;           // okay, member is public
    /*3*/  c.radius = 6;           // error, member is private

    f(&c);

    return 0;
}

void f(Circle *p)
{
    /*4*/  p->xorigin = 4;           // error, member is private
    /*5*/  p->yorigin = 5;           // okay, member is public
    /*6*/  p->radius = 6;           // error, member is private
}

```

By default, all members of a class are private; it is as if there is an occurrence of the `private` access specifier keyword before the first member. In this case, we have one implicitly private member, one explicitly public member, and one explicitly private member, in that order. The simplest approach is to declare all the private members first, followed by the `public` access specifier and then the public members.¹

C++ programmers often say that `c` is an *instance* of the object type `Circle`, or that `c` is an *instantiation* of that type.

In `main`, even though object `c` is in scope, we cannot access the private members `xorigin` and `radius` directly in cases 1 and 3. And when we pass the address of `c` to function `f`, we are likewise prohibited from accessing those members in cases 4 and 6. On the other hand, the public member `yorigin` is directly and indirectly accessible, as shown in cases 2 and 5.

As we learned in §7.1, a structure can contain more than data, and since a class is a structure, a class can do likewise; for example (see directory `cl03`), it can contain enumerations and type synonyms:

¹ Some programmers prefer to do the exact opposite; public first then private.

13. Operator Overloading

The addition of classes is a step toward integrating user-defined types into the language. Once a class has been defined, we can create objects of that type by using the same notation as C++'s built-in types. However, to make these classes really fit in we need to be able to operate on class objects using the same easy notation allowed by the built-in operators. In this chapter, we will see how most of the built-in operators can be given meaning in the context of class objects.

13.1 Introduction

Consider the case in which we have two objects (called `emp1` and `emp2`) of type `Employee`, and we'd like to see if they are identical. The intuitive thing to want to write is

```
Employee emp1, emp2;

// the variables get initialized somehow

if (emp1 == emp2)
{
    // ...
}
```

Similarly, if we had a `Matrix` type we might want to add `Matrix` objects together, or to add one to an integer. Again, the intuitive approach would be to use something like

```
Matrix m1, m2, m3, m4;

m3 = m1 + m2;
m4 = m3 + 50;
```

C++ allows this. We can assign meaning to almost all of the built-in operators when they are used in the context of a user-defined type. For example, we can define what `+` means when its operands have type `Matrix`. Assigning an alternative meaning to existing operators is called *operator overloading*. For mathematical classes such as those that implement complex arithmetic, vectors, and matrices, most of the built-in arithmetic operators immediately become candidates for overloading.

While we can define `m1 - m2` to produce a sum, that would be counterintuitive; that is, we must take care that the operator symbol we choose for a particular meaning implies something reasonable. Which operator, for example, would we choose to invert a matrix? None of the built-in operators implies such an operation, so we would be forced to choose one that is obscure. Also, we should try to be consistent. If we make `%` mean completely different things for different classes, it would be very hard to understand the resulting expressions that use that operator; just when does it mean the usual remainder and when is it one of the (possibly obscure) overloaded uses?

Programming in C++

All operators maintain the precedence assigned to them in the language definition, whether or not they are overloaded. For example, the binary `*` and `/` operators always have higher precedence than the binary `+` and `-` operators. Similarly, the associativity of operators with the same precedence is preserved. Another reasonable restriction is that overloaded operators must have the same number of operands as their built-in counterparts. And except for the function call operator, they cannot have default arguments. Four operators cannot be overloaded: `.`, `.*`, `::`, and `?:`.

When using C++'s built-in types, we have come to learn certain rules. For example:

- Certain expressions are commutative (for example, `a + b` is equivalent to `b + a`).
- `i++`, `++i`, `i += 1`, and `i = i + 1` are related.

However, once we start to overload operators, none of these guarantees is automatically preserved; yet we should make them so.

A few operators are predefined for all types, including classes. These are simple assignment (`=`), address-of (`&`), indirect member selection (`->`), and comma (`,`). In any event, if we can come up with a good reason, these can also be overloaded on a per-class basis.

Another aspect to consider is the type of the result of each operator we overload. We should make an overloaded operator behave in the way the programmer expects based on the built-in equivalent. For example, the built-in relational and equality operators produce a result of type `bool`, so it makes a great deal of sense for us to mimic this behavior.

Similarly, many built-in operators expect operands of specific types, and if their operands are not the expected type but are compatible types, they are promoted before being operated on. This is only true for overloaded versions if we make it so.

13.2 A Simple Example

The following example (see `ov01.cpp`) overloads the equality operators `==` and `!=` when both operands are objects of type `Point`:

```
// Point.h

class Point
{
    // ...
public:
    // ...

    bool operator==(const Point& p) const    // p1 == p2
    {
        return (getX() == p.getX()) && (getY() == p.getY());
    }
}
```

```

    bool operator!=(const Point& p) const    // p1 != p2
    {
        return !(*this == p);
    }
};

```

To overload an operator, we must use the keyword `operator`, as shown above. In the first case, we specify that we wish to overload the `==` operator. Since this is a binary operator, we must specify the types of both its operands; however, we have shown only one argument, of type `Point`. This is the type of the right operand. The type of the left operand is automatically assumed to be an object of the class being defined, in this case, `Point`, which is made available via `this`. The type of the result produced is indicated by the function's return type, in this case `bool`.

If these operator functions are designed to behave just like their built-in operator counterparts, we can use these new versions in the same manner. For example:

```

#include <iostream>
using namespace std;
#include "Point.h"

int main()
{
    Point p1(1, 2);
    Point p2(1, 2);
    Point p3(4, -2);

    cout << boolalpha;
    cout << "p1 == p2 => " << (p1 == p2) << '\n';
    cout << "p1 != p2 => " << (p1 != p2) << '\n';

    if (p1 == p3)
    {
        cout << "p1 is equal to p3\n";
    }
    else
    {
        cout << "p1 is not equal to p3\n";
    }

    return 0;
}

```

Three Points are created, two of which have identical properties. The output produced is:

```

p1 == p2 => true
p1 != p2 => false
p1 is not equal to p3

```

14. Inheritance

In this chapter, we'll learn about single and multiple inheritance, polymorphism, abstract classes, and virtual base classes.

14.1 Introduction

If there's one thing we can say about software design, it's that the needs of a non-trivial program usually change over its lifetime. Moreover, very often, they change during its initial development phase. Since it is impossible to cater for every conceivable change or enhancement, the best we can do is to be as abstract as we can and to develop software that can be extended in an upward-compatible fashion with little or no impact on existing modules.

Consider a class `Point`, which contains `x`- and `y`-coordinates. Let's assume that this class has been used in production for quite some time and many modules in some number of applications use it. Since an individual `Point` has no color attribute, all `Points` have been displayed in the color established by the user in some way external to the `Point` object. That is, a `Point` has no control over the color in which it is displayed. However, now we wish individual `Points` to each have their own color. How should we go about this? There are three possibilities:

Approach 1: Add a color attribute to class `Point` — By adding a new member, we change the size and layout of every `Point`, requiring us to recompile all existing modules and re-link all existing applications. We'd have to add a color argument to various constructors and/or member functions but by making it the last argument and using a default value, we can probably avoid breaking most existing user code. If we haven't made any rash assumptions in the existing code, this process ought not to be too difficult but it can take time.

For all data files containing `Points`, we have to modify their record layouts so the `Points` now contain a color field; that may be simply a matter of making them all the same color or it might require some subjective changing. And when we are all done, every `Point` from this time on must have a color whether we really need one or not. That is, we only ever have one idea of a `Point`.

Here's an example of how `Point` might have been defined previously:

```
class Point
{
    int xcoord;
    int ycoord;
public:
    // ...
};
```

Here's what it might look like after the addition:

```

class Point
{
public:
    enum pointColor {red, blue, green, white};
private:
    int xcoord;
    int ycoord;
    pointColor color;
public:
    // ...
};

```

Approach 2: Create a new class `ColoredPoint` that contains a color attribute and an explicit `Point` — We might define such a class as follows:

```

class ColoredPoint
{
public:
    enum pointColor {red, blue, green, white};
private:
    Point pt;
    pointColor color;
public:
    // ...
};

```

While this approach can be made to work, a `ColoredPoint` doesn't have as close a relationship with a `Point` as we might like. For example, we cannot pass by address, or reference, a `ColoredPoint` to a function that expects a `Point`; the two types are simply not compatible. Yet a `ColoredPoint` is a `Point`, so any operation on a `Point` makes sense on a `ColoredPoint`; the language, however, simply prohibits this. As a direct result, a linked-list, stack, or queue of `Points` could not contain a `ColoredPoint`. Another limitation is that the member functions of class `Point` are not automatically available to class `ColoredPoint`. For example, assuming that the `Point` class has a function `move` that changes a `Point`'s coordinates, we cannot call this function directly for a `ColoredPoint` even though it makes sense to want to. Instead, we must call it on a `ColoredPoint`'s `pt` member, but since that member is private, we need a public wrapper `move` function in `ColoredPoint` that calls the `move` in `Point`.

Containment is the situation in which one object contains another object directly or contains a pointer or reference to another object. In this example, a `ColoredPoint` *contains* a `Point`. (Containment is also referred to as *composition*.)

Approach 3: Create a new class `ColoredPoint` that contains a color attribute and an implicit `Point`, inherited from class `Point` — We might define such a class as follows:

```

class ColoredPoint : public Point
{
public:
    enum pointColor {red, blue, green, white};
private:
    pointColor color;
public:
    // ...
};

```

In this example, we have the *derived class* ColoredPoint derived from the *base class* Point. That is, ColoredPoint *inherits* all of Point's data and function members.

This approach solves all the problems raised in the containment approach shown earlier. Specifically, the address of a ColoredPoint can be used in any situation where the address of a Point is expected. That is, a ColoredPoint *is* a Point—it just happens to have another attribute as well. As a result, a linked-list, stack, or queue of Points can contain ColoredPoints or, indeed, objects of any type derived directly or indirectly from Point. And the member functions of class Point are automatically available to class ColoredPoint. In short, inheritance provides significant notational convenience over composition.

14.2 The Is-A versus Has-A Test

A significant question that arises when defining a new class that is somehow related to an existing class is, "Should we use inheritance or composition?" A widely used, and simple, rule of thumb to apply here is, "Does an Is-A relationship exist?" That is, "Is the new class a more specialized example of the old class?" If so, use inheritance. "Does a Has-A relationship exist?" That is, "Does the new class simply contain an instance of the old class?" If so, use composition.

Let's apply these rules to a number of class families.

Family 1: Is-A manager an employee? Yes.

```

Employee
  Manager

```

Family 2: A peripheral is a general base class of which there are a number of specializations, such as disk, printer, and terminal. A fixed-disk Is-A disk, which, in turn, Is-A peripheral.

```

Peripheral
  Disk
    Fixed
    Removable
  Printer
    Character
    Line
    Page

```