C++ Primer Plus

Fifth Edition

SAMS

Stephen Prata

C++

Primer Plus

Fifth Edition

Stephen Prata

SAMS

800 East 96th St., Indianapolis, Indiana, 46240 USA

C++ Primer Plus

International Standard Book Number: 0-672-32697-3

Library of Congress Catalog Card Number: 2004095067

Printed in the United States of America

First Printing: November, 2004

07 06 05 04 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

1-317-428-3341

international@pearsontechgroup.com

ASSOCIATE PUBLISHER
Michael Stephens

ACQUISITIONS EDITOR
Loretta Yates

DEVELOPMENT EDITOR
Songlin Qiu

MANAGING EDITOR
Charlotte Clapp

PROJECT EDITOR
George E. Nedeff

COPY EDITOR
Kitty Jarrett

INDEXER
Erika Millen

PROOFREADER
Suzanne Thomas

TECHNICAL EDITOR
David Horvath

PUBLISHING COORDINATOR
Cindy Teeters

MULTIMEDIA DEVELOPER
Dan Scherf

BOOK DESIGNER
Gary Adair

CONTENTS AT A GLANCE

TABLE OF CONTENTS

ABOUT THE AUTHOR

Stephen Prata teaches astronomy, physics, and computer science at the College of Marin in Kentfield, California. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. Stephen has authored or coauthored more than a dozen books for The Waite Group. He wrote The Waite Group's *New C Primer Plus*, which received the Computer Press Association's 1990 Best How-to Computer Book Award, and The Waite Group's *C++ Primer Plus*, nominated for the Computer Press Association's Best How-to Computer Book Award in 1991.

DEDICATION

To my colleagues and students at the College of Marin, with whom it is a pleasure to work.

—Stephen Prata

ACKNOWLEDGMENTS

Acknowledgments for the Fifth Edition

I'd like to thank Loretta Yates and Songlin Qiu of Sams Publishing for guiding and managing this project. Thanks to my colleague Fred Schmitt for several useful suggestions. Once again, I'd like to thank Ron Liechty of Metrowerks for his helpfulness.

Acknowledgments for the Fourth Edition

Several editors from Pearson and from Sams helped originate and maintain this project; thanks to Linda Sharp, Karen Wachs, and Laurie McGuire. Thanks, too, to Michael Maddox, Bill Craun, Chris Maunder, and Phillipe Bruno for providing technical review and editing. And thanks again to Michael Maddox and Bill Craun for supplying the material for the Real World Notes. Finally, I'd like to thank Ron Liechty of Metrowerks and Greg Comeau of Comeau Computing for their aid with C++ compilers.

Acknowledgments for the Third Edition

I'd like to thank the editors from Macmillan and The Waite Group for the roles they played in putting this book together: Tracy Dunkelberger, Susan Walton, and Andrea Rosenberg. Thanks, too, to Russ Jacobs for his content and technical editing. From Metrowerks, I'd like to thank Dave Mark, Alex Harper, and especially Ron Liechty, for their help and cooperation.

Acknowledgments for the Second Edition

I'd like to thank Mitchell Waite and Scott Calamar for supporting a second edition and Joel Fugazzotto and Joanne Miller for guiding the project to completion. Thanks to Michael Marcotty of Metrowerks for dealing with my questions about their beta version CodeWarrior compiler. I'd also like to thank the following instructors for taking the time to give us feedback on the first edition: Jeff Buckwalter, Earl Brynner, Mike Holland, Andy Yao, Larry Sanders,

Shahin Momtazi, and Don Stephens. Finally, I wish to thank Heidi Brumbaugh for her helpful content editing of new and revised material.

Acknowledgments for the First Edition

Many people have contributed to this book. In particular, I wish to thank Mitch Waite for his work in developing, shaping, and reshaping this book, and for reviewing the manuscript. I appreciate Harry Henderson's work in reviewing the last few chapters and in testing programs with the Zortech C++ compiler. Thanks to David Gerrold for reviewing the entire manuscript and for championing the needs of less-experienced readers. Also thanks to Hank Shiffman for testing programs using Sun C++ and to Kent Williams for testing programs with AT&T cfront and with G++. Thanks to Nan Borreson of Borland International for her responsive and cheerful assistance with Turbo C++ and Borland C++. Thank you, Ruth Myers and Christine Bush, for handling the relentless paper flow involved with this kind of project. Finally, thanks to Scott Calamar for keeping everything on track.

WE WANT TO HEAR FROM YOU!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

For more information about this book or another Sams Publishing title, visit our web site at `www.samspublishing.com`. Type the ISBN (0672326973) or the title of a book in the Search field to find the page you're looking for.

INTRODUCTION

Preface to the Fifth Edition

Learning C++ is an adventure of discovery, particularly because the language accommodates several programming paradigms, including object-oriented programming, generic programming, and the traditional procedural programming. C++ was a moving target as the language added new features, but now, with the ISO/ANSI C++ Standard, Second Edition (2003), in place, the language has stabilized. Contemporary compilers support most or all of the features mandated by the standard, and programmers have had time to get used to applying these features. The fifth edition of this book, ***C++ Primer Plus,*** reflects the ISO/ANSI standard and describes this matured version of C++.

C++ Primer Plus discusses the basic C language and presents C++ features, making this book self-contained. It presents C++ fundamentals and illustrates them with short, to-the-point programs that are easy to copy and experiment with. You'll learn about input/output (I/O), how to make programs perform repetitive tasks and make choices, the many ways to handle data, and how to use functions. You'll learn about the many features C++ has added to C, including the following:

- Classes and objects
- Inheritance
- Polymorphism, virtual functions, and runtime type identification (RTTI)
- Function overloading
- Reference variables
- Generic, or type-independent, programming, as provided by templates and the Standard Template Library (STL)
- The exception mechanism for handling error conditions
- Namespaces for managing names of functions, classes, and variables

The Primer Approach

C++ Primer Plus brings several virtues to the task of presenting all this material. It builds on the primer tradition begun by ***C Primer Plus*** nearly two decades ago and embraces its successful philosophy:

- A primer should be an easy-to-use, friendly guide.
- A primer doesn't assume that you are already familiar with all relevant programming concepts.
- A primer emphasizes hands-on learning with brief, easily typed examples that develop your understanding, a concept or two at a time.
- A primer clarifies concepts with illustrations.
- A primer provides questions and exercises to let you test your understanding, making the book suitable for self-learning or for the classroom.

Following these principles, the book helps you understand this rich language and how to use it. For example:

- It provides conceptual guidance about when to use particular features, such as using public inheritance to model what are known as *is-a* relationships.
- It illustrates common C++ programming idioms and techniques.
- It provides a variety of sidebars, including tips, cautions, things to remember, compatibility notes, and real-world notes.

The author and editors of this book do our best to keep the presentation to-the-point, simple, and fun. Our goal is that by the end of the book, you'll be able to write solid, effective programs and enjoy yourself doing so.

Sample Code Used in This Book

This book provides an abundance of sample code, most of it in the form of complete programs. Like the previous editions, this book practices generic C++ so that it is not tied to any particular kind of computer, operating system, or compiler. Thus, the examples were tested on a Windows XP system, a Macintosh OS X system, and a Linux system. Only a few programs were affected by compiler non-conformance issues. Compiler compliance with the C++ standard has improved since the previous edition of this book first appeared.

The sample code for the complete programs described in this book is available on the Sams website, at `www.samspublishing.com`. Enter this book's ISBN (without the hyphens) in the Search box and click Search. When the book's title is displayed, click the title to go to a page where you can download the code. You also can find solutions to selected programming exercises at this site.

How This Book Is Organized

This book is divided into 17 chapters and 10 appendixes, summarized here.

Chapter 1: Getting Started

Chapter 1 relates how Bjarne Stroustrup created the C++ programming language by adding object-oriented programming support to the C language. You'll learn the distinctions between procedural languages, such as C, and object-oriented languages, such as C++. You'll read about the joint ANSI/ISO work to develop a C++ standard. This chapter discusses the mechanics of creating a C++ program, outlining the approach for several current C++ compilers. Finally, it describes the conventions used in this book.

Chapter 2: Setting Out to C++

Chapter 2 guides you through the process of creating simple C++ programs. You'll learn about the role of the `main()` function and about some of the kinds of statements that C++ programs use. You'll use the predefined `cout` and `cin` objects for program output and input, and you'll learn about creating and using variables. Finally, you'll be introduced to functions, C++'s programming modules.

Chapter 3: Dealing with Data

C++ provides built-in types for storing two kinds of data: integers (numbers with no fractional parts) and floating-point numbers (numbers with fractional parts). To meet the diverse requirements of programmers, C++ offers several types in each category. Chapter 3 discusses those types, including creating variables and writing constants of various types. You'll also learn how C++ handles implicit and explicit conversions from one type to another.

Chapter 4: Compound Types

C++ lets you construct more elaborate types from the basic built-in types. The most advanced form is the class, discussed in Chapters 9 through 13. Chapter 4 discusses other forms, including arrays, which hold several values of a single type; structures, which hold several values of unlike types; and pointers, which identify locations in memory. You'll also learn how to create and store text strings and to handle text I/O by using C-style character arrays and the C++ `string` class. Finally, you'll learn some of the ways C++ handles memory allocation, including using the `new` and `delete` operators for managing memory explicitly.

Chapter 5: Loops and Relational Expressions

Programs often must perform repetitive actions, and C++ provides three looping structures for that purpose: the `for` loop, the `while` loop, and the `do while` loop. Such loops must know when they should terminate, and the C++ relational operators enable you to create tests to guide such loops. In Chapter 5 you learn how to create loops that read and process input character-by-character. Finally, you'll learn how to create two-dimensional arrays and how to use nested loops to process them.

Chapter 6: Branching Statements and Logical Operators

Programs can behave intelligently if they can tailor their behavior to circumstances. In Chapter 6 you'll learn how to control program flow by using the `if`, `if else`, and `switch` statements and the conditional operator. You'll learn how to use logical operators to help express decision-making tests. Also, you'll meet the `cctype` library of functions for evaluating character relations, such as testing whether a character is a digit or a nonprinting character. Finally, you'll get an introductory view of file I/O.

Chapter 7: Functions: C++'s Programming Modules

Functions are the basic building blocks of C++ programming. Chapter 7 concentrates on features that C++ functions share with C functions. In particular, you'll review the general format of a function definition and examine how function prototypes increase the reliability of programs. Also, you'll investigate how to write functions to process arrays, character strings, and structures. Next, you'll learn about recursion, which is when a function calls itself, and see how it can be used to implement a divide-and-conquer strategy. Finally, you'll meet pointers to functions, which enable you to use a function argument to tell one function to use a second function.

Chapter 8: Adventures in Functions

Chapter 8 explores the new features C++ adds to functions. You'll learn about inline functions, which can speed program execution at the cost of additional program size. You'll work with reference variables, which provide an alternative way to pass information to functions. Default arguments let a function automatically supply values for function arguments that you omit from a function call. Function overloading lets you create functions having the same name but taking different argument lists. All these features have frequent use in class design. Also, you'll learn about function templates, which allow you to specify the design of a family of related functions.

Chapter 9: Memory Models and Namespaces

Chapter 9 discusses putting together multifile programs. It examines the choices in allocating memory, looking at different methods of managing memory and at scope, linkage, and namespaces, which determine what parts of a program know about a variable.

Chapter 10: Objects and Classes

A class is a user-defined type, and an object (such as a variable) is an instance of a class. Chapter 10 introduces you to object-oriented programming and to class design. A class declaration describes the information stored in a class object and also the operations (class methods) allowed for class objects. Some parts of an object are visible to the outside world (the public portion), and some are hidden (the private portion). Special class methods (constructors and destructors) come into play when objects are created and destroyed. You will learn

about all this and other class details in this chapter, and you'll see how classes can be used to implement ADTs, such as a stack.

Chapter 11: Working with Classes

In Chapter 11 you'll further your understanding of classes. First, you'll learn about operator overloading, which lets you define how operators such as + will work with class objects. You'll learn about friend functions, which can access class data that's inaccessible to the world at large. You'll see how certain constructors and overloaded operator member functions can be used to manage conversion to and from class types.

Chapter 12: Classes and Dynamic Memory Allocation

Often it's useful to have a class member point to dynamically allocated memory. If you use `new` in a class constructor to allocate dynamic memory, you incur the responsibilities of providing an appropriate destructor, of defining an explicit copy constructor, and of defining an explicit assignment operator. Chapter 12 shows you how and discusses the behavior of the member functions generated implicitly if you fail to provide explicit definitions. You'll also expand your experience with classes by using pointers to objects and studying a queue simulation problem.

Chapter 13: Class Inheritance

One of the most powerful features of object-oriented programming is inheritance, by which a derived class inherits the features of a base class, enabling you to reuse the base class code. Chapter 13 discusses public inheritance, which models *is-a* relationships, meaning that a derived object is a special case of a base object. For example, a physicist is a special case of a scientist. Some inheritance relationships are polymorphic, meaning you can write code using a mixture of related classes for which the same method name may invoke behavior that depends on the object type. Implementing this kind of behavior necessitates using a new kind of member function called a virtual function. Sometimes using abstract base classes is the best approach to inheritance relationships. This chapter discusses these matters, pointing out when public inheritance is appropriate and when it is not.

Chapter 14: Reusing Code in C++

Public inheritance is just one way to reuse code. Chapter 14 looks at several other ways. Containment is when one class contains members that are objects of another class. It can be used to model *has-a* relationships, in which one class has components of another class. For example, an automobile has a motor. You also can use private and protected inheritance to model such relationships. This chapter shows you how and points out the differences among the different approaches. Also, you'll learn about class templates, which let you define a class in terms of some unspecified generic type, and then use the template to create specific classes in terms of specific types. For example, a stack template enables you to create a stack of integers or a stack of strings. Finally, you'll learn about multiple public inheritance, whereby a class can derive from more than one class.

Chapter 15: Friends, Exceptions, and More

Chapter 15 extends the discussion of friends to include friend classes and friend member functions. Then it presents several new developments in C++, beginning with exceptions, which provide a mechanism for dealing with unusual program occurrences, such an inappropriate function argument values and running out of memory. Then you'll learn about RTTI, a mechanism for identifying object types. Finally, you'll learn about the safer alternatives to unrestricted typecasting.

Chapter 16: The `string` Class and the Standard Template Library

Chapter 16 discusses some useful class libraries recently added to the language. The `string` class is a convenient and powerful alternative to traditional C-style strings. The `auto_ptr` class helps manage dynamically allocated memory. The STL provides several generic containers, including template representations of arrays, queues, lists, sets, and maps. It also provides an efficient library of generic algorithms that can be used with STL containers and also with ordinary arrays. The `valarray` template class provides support for numeric arrays.

Chapter 17: Input, Output, and Files

Chapter 17 reviews C++ I/O and discusses how to format output. You'll learn how to use class methods to determine the state of an input or output stream and to see, for example, whether there has been a type mismatch on input or whether the end-of-file has been detected. C++ uses inheritance to derive classes for managing file input and output. You'll learn how to open files for input and output, how to append data to a file, how to use binary files, and how to get random access to a file. Finally, you'll learn how to apply standard I/O methods to read from and write to strings.

Appendix A: Number Bases

Appendix A discusses octal, hexadecimal, and binary numbers.

Appendix B: C++ Reserved Words

Appendix B lists C++ keywords.

Appendix C: The ASCII Character Set

Appendix C lists the ASCII character set, along with decimal, octal, hexadecimal, and binary representations.

Appendix D: Operator Precedence

Appendix D lists the C++ operators in order of decreasing precedence.

Appendix E: Other Operators

Appendix E summarizes the C++ operators, such as the bitwise operators, not covered in the main body of the text.

Appendix F: The `string` Template Class

Appendix F summarizes `string` class methods and functions.

Appendix G: The STL Methods and Functions

Appendix G summarizes the STL container methods and the general STL algorithm functions.

Appendix H: Selected Readings and Internet Resources

Appendix H lists some books that can further your understanding of C++.

Appendix I: Converting to ANSI/ISO Standard C++

Appendix I provides guidelines for moving from C and older C++ implementations to ANSI/ISO C++.

Appendix J: Answers to Review Questions

Appendix J contains the answers to the review questions posed at the end of each chapter.

Note to Instructors

One of the goals of this edition of *C++ Primer Plus* is to provide a book that can be used as either a teach-yourself book or as a textbook. Here are some of the features that support using *C++ Primer Plus,* Fifth Edition, as a textbook:

- This book describes generic C++, so it isn't dependent on a particular implementation.
- The contents track the ISO/ANSI C++ standards committee's work and include discussions of templates, the STL, the `string` class, exceptions, RTTI, and namespaces.
- It doesn't assume prior knowledge of C, so it can be used without a C prerequisite. (Some programming background is desirable, however.)
- Topics are arranged so that the early chapters can be covered rapidly as review chapters for courses that do have a C prerequisite.
- Chapters include review questions and programming exercises. Appendix J provides the answers to the review questions. Solutions to selected programming exercises can be found at the Sams website (www.samspublishing.com).

- The book introduces several topics that are appropriate for computer science courses, including abstract data types (ADTs), stacks, queues, simple lists, simulations, generic programming, and using recursion to implement a divide-and-conquer strategy.
- Most chapters are short enough to cover in a week or less.
- The book discusses ***when*** to use certain features as well as ***how*** to use them. For example, it links public inheritance to ***is-a*** relationships and composition and private inheritance to ***has-a*** relationships, and it discusses when to use virtual functions and when not to.

Conventions Used in This Book

This book uses several typographic conventions to distinguish among various kinds of text:

- Code lines, commands, statements, variables, filenames, and program output appear in a `computer typeface`:

```
#include <iostream>
int main()
{
    using namespace std;
    cout << "What's up, Doc!\n";
    return 0;
}
```

- Program input that you should type appears in **`bold computer typeface`**:

```
Please enter your name:
Plato
```

- Placeholders in syntax descriptions appear in an ***`italic computer typeface`***. You should replace a placeholder with the actual filename, parameter, or whatever element it represents.
- *Italic type* is used for new terms.

This book includes several elements intended to illuminate specific points:

Compatibility Note

Most compilers are not yet 100% compliant with the ISO/ANSI Standard, and these notes warn you of discrepancies you may encounter.

Remember

These notes highlight points that are important to remember.

Real-World Note

Several professional programmers offer observations based on their experiences.

Sidebar

A sidebar provides a deeper discussion or additional background to help illuminate a topic.

Tip

Tips present short, helpful guides to particular programming situations.

Caution

A caution alerts you to potential pitfalls.

Note

The notes provide a catch-all category for comments that don't fall into one of the other categories.

Systems Used to Develop This Book's Programming Examples

For the record, the examples in this book were developed using Microsoft Visual C++ 7.1 (the version that comes with Microsoft Visual Studio .NET 2003) and Metrowerks CodeWarrior Development Studio 9 on a Pentium PC with a hard disk and running under Windows XP Professional. Most programs were checked using the Borland C++ 5.5 command-line compiler and GNU `gpp` 3.3.3 on the same system, using Comeau 4.3.3 and GNU g++ 3.3.1 on an IBM-compatible Pentium running SuSE 9.0 Linux, and using Metrowerks Development Studio 9 on a Macintosh G4 under OS 10.3. This book reports discrepancies stemming from lagging behind the standard generically, as in "older implementations use `ios::fixed` instead of `ios_base::fixed`." This book reports some bugs and idiosyncrasies in older compilers that would prove troublesome or confusing; most of these have been fixed in current releases.

C++ offers a lot to the programmer; learn and enjoy!

CHAPTER 1

GETTING STARTED

In this chapter you'll learn about the following:

- The history and philosophy of C and of C++
- Procedural versus object-oriented programming
- How C++ adds object-oriented concepts to the C language
- How C++ adds generic programming concepts to the C language
- Programming language standards
- The mechanics of creating a program

Welcome to C++! This exciting language, which blends the C language with support for object-oriented programming, became one of the most important programming languages of the 1990s and continues strongly into the 2000s. Its C ancestry brings to C++ the tradition of an efficient, compact, fast, and portable language. Its object-oriented heritage brings C++ a fresh programming methodology, designed to cope with the escalating complexity of modern programming tasks. Its template features bring yet another new programming methodology: generic programming. This triple heritage is both a blessing and a bane. It makes the language very powerful, but it also means there's a lot to learn.

This chapter explores C++'s background further and then goes over some of the ground rules for creating C++ programs. The rest of the book teaches you to use the C++ language, going from the modest basics of the language to the glory of object-oriented programming (OOP) and its supporting cast of new jargon—objects, classes, encapsulation, data hiding, polymorphism, and inheritance—and then on to its support of generic programming. (Of course, as you learn C++, these terms will be transformed from buzzwords to the necessary vocabulary of cultivated discourse.)

Learning C++: What Lies Before You

C++ joins three separate programming traditions: the procedural language tradition, represented by C; the object-oriented language tradition, represented by the class enhancements C++ adds to C; and generic programming, supported by C++ templates. This chapter looks

into those traditions. But first, let's consider what this heritage implies about learning C++. One reason to use C++ is to avail yourself of its object-oriented features. To do so, you need a sound background in standard C, for that language provides the basic types, operators, control structures, and syntax rules. So if you already know C, you're poised to learn C++. But it's not just a matter of learning a few more keywords and constructs. Going from C to C++ involves about as much work as learning C in the first place. Also, if you know C, you must unlearn some programming habits as you make the transition to C++. If you don't know C, you have to master the C components, the OOP components, and the generic components to learn C++, but at least you may not have to unlearn programming habits. If you are beginning to think that learning C++ may involve some mind-stretching effort on your part, you're right. This book will guide you through the process in a clear, helpful manner, one step at a time, so the mind-stretching will be sufficiently gentle to leave your brain resilient.

C++ Primer Plus approaches C++ by teaching both its C basis and its new components, so it assumes that you have no prior knowledge of C. You'll start by learning the features C++ shares with C. Even if you know C, you may find this part of the book a good review. Also, it points out concepts that will become important later, and it indicates where C++ differs from C. After you have a good grounding in the basics of C, you'll learn about the C++ superstructure. At that point, you'll learn about objects and classes and how C++ implements them. And you will learn about templates.

This book is not intended to be a complete C++ reference; it doesn't explore every nook and cranny of the language. But you will learn all the major features of the language, including some, such as templates, exceptions, and namespaces, that are more recent additions.

Now let's take a brief look at some of C++'s background.

The Origins of C++: A Little History

Computer technology has evolved at an amazing rate over the past few decades. Today a notebook computer can compute faster and store more information than the mainframe computers of the 1960s. (Quite a few programmers can recall bearing offerings of decks of punched cards to be submitted to a mighty, room-filling computer system with a majestic 100KB of memory—not enough memory to run a good personal computer game today.) Computer languages have evolved, too. The changes may not be as dramatic, but they are important. Bigger, more powerful computers spawn bigger, more complex programs, which, in turn, raise new problems in program management and maintenance.

In the 1970s, languages such as C and Pascal helped usher in an era of structured programming, a philosophy that brought some order and discipline to a field badly in need of these qualities. Besides providing the tools for structured programming, C also produced compact, fast-running programs, along with the ability to address hardware matters, such as managing communication ports and disk drives. These gifts helped make C the dominant programming language in the 1980s. Meanwhile, the 1980s witnessed the growth of a new programming paradigm: object-oriented programming, or OOP, as embodied in languages such as SmallTalk and C++. Let's examine these C and OOP a bit more closely.

The C Language

In the early 1970s, Dennis Ritchie of Bell Laboratories was working on a project to develop the Unix operating system. (An ***operating system*** is a set of programs that manages a computer's resources and handles its interactions with users. For example, it's the operating system that puts the system prompt onscreen and that runs programs for you.) For this work Ritchie needed a language that was concise, that produced compact, fast programs, and that could control hardware efficiently. Traditionally, programmers met these needs by using assembly language, which is closely tied to a computer's internal machine language. However, assembly language is a ***low-level*** language—that is, it is specific to a particular computer processor. So if you want to move an assembly program to a different kind of computer, you may have to completely rewrite the program, using a different assembly language. It was a bit as if each time you bought a new car, you found that the designers decided to change where the controls went and what they did, forcing you to relearn how to drive. But Unix was intended to work on a variety of computer types (or platforms). That suggested using a high-level language. A ***high-level*** language is oriented toward problem solving instead of toward specific hardware. Special programs called ***compilers*** translate a high-level language to the internal language of a particular computer. Thus, you can use the same high-level language program on different platforms by using a separate compiler for each platform. Ritchie wanted a language that combined low-level efficiency and hardware access with high-level generality and portability. So, building from older languages, he created C.

C Programming Philosophy

Because C++ grafts a new programming philosophy onto C, we should first take a look at the older philosophy that C follows. In general, computer languages deal with two concepts—data and algorithms. The ***data*** constitutes the information a program uses and processes. The ***algorithms*** are the methods the program uses (see Figure 1.1). Like most mainstream languages when C was created, C is a ***procedural*** language. That means it emphasizes the algorithm side of programming. Conceptually, procedural programming consists of figuring out the actions a computer should take and then using the programming language to implement those actions. A program prescribes a set of procedures for the computer to follow to produce a particular outcome, much as a recipe prescribes a set of procedures for a cook to follow to produce a cake.

Earlier procedural languages, such as FORTRAN and BASIC, ran into organizational problems as programs grew larger. For example, programs often use branching statements, which route execution to one or another set of instructions, depending on the result of some sort of test. Many older programs had such tangled routing (called "spaghetti programming") that it was virtually impossible to understand a program by reading it, and modifying such a program was an invitation to disaster. In response, computer scientists developed a more disciplined style of programming called ***structured programming***. C includes features to facilitate this approach. For example, structured programming limits branching (choosing which instruction to do next) to a small set of well-behaved constructions. C incorporates these constructions (the `for` loop, the `while` loop, the `do while` loop, and the `if else` statement) into its vocabulary.

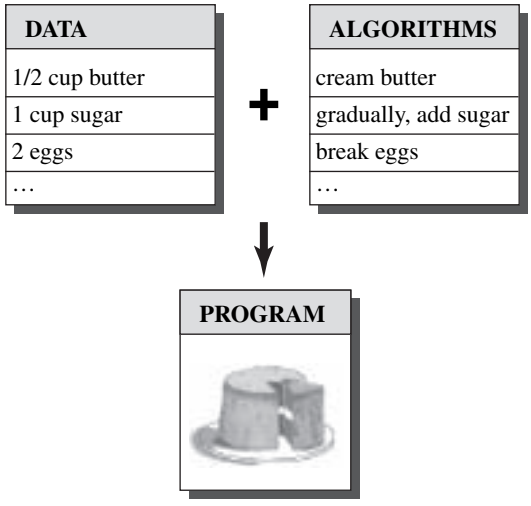


FIGURE 1.1
Data + algorithms = program.

Top-down design was another of the new principles. With C, the idea is to break a large program into smaller, more manageable tasks. If one of these tasks is still too broad, you divide it into yet smaller tasks. You continue with this process until the program is compartmentalized into small, easily programmed modules. (Organize your study. Aargh! Well, organize your desk, your table top, your filing cabinet, and your bookshelves. Aargh! Well, start with the desk and organize each drawer, starting with the middle one. Hmmm, perhaps I can manage that task.) C's design facilitates this approach, encouraging you to develop program units called *functions* to represent individual task modules. As you may have noticed, the structured programming techniques reflect a procedural mind-set, thinking of a program in terms of the actions it performs.

The C++ Shift: Object-Oriented Programming

Although the principles of structured programming improved the clarity, reliability, and ease of maintenance of programs, large-scale programming still remains a challenge. OOP brings a new approach to that challenge. Unlike procedural programming, which emphasizes algorithms, OOP emphasizes the data. Rather than try to fit a problem to the procedural approach of a language, OOP attempts to fit the language to the problem. The idea is to design data forms that correspond to the essential features of a problem.

In C++, a *class* is a specification describing such a new data form, and an *object* is a particular data structure constructed according to that plan. For example, a class could describe the general properties of a corporation executive (name, title, salary, unusual abilities, for example), while an object would represent a specific executive (Guilford Sheepblat, vice president, $325,000, knows how to restore the Windows registry). In general, a class defines what data is used to represent an object *and* the operations that can be performed on that data. For example, suppose you were developing a computer drawing program capable of drawing a rectangle. You could define a class to describe a rectangle. The data part of the specification could

include such things as the location of the corners, the height and width, the color and style of the boundary line, and the color and pattern used to fill the rectangle. The operations part of the specification could include methods for moving the rectangle, resizing it, rotating it, changing colors and patterns, and copying the rectangle to another location. If you then used your program to draw a rectangle, it would create an object according to the class specification. That object would hold all the data values describing the rectangle, and you could use the class methods to modify that rectangle. If you drew two rectangles, the program would create two objects, one for each rectangle.

The OOP approach to program design is to first design classes that accurately represent those things with which the program deals. For example, a drawing program might define classes to represent rectangles, lines, circles, brushes, pens, and the like. The class definitions, recall, include a description of permissible operations for each class, such as moving a circle or rotating a line. Then you would proceed to design a program, using objects of those classes. The process of going from a lower level of organization, such as classes, to a higher level, such as program design, is called ***bottom-up*** programming.

There's more to OOP than the binding of data and methods into a class definition. For example, OOP facilitates creating reusable code, and that can eventually save a lot of work. Information hiding safeguards data from improper access. Polymorphism lets you create multiple definitions for operators and functions, with the programming context determining which definition is used. Inheritance lets you derive new classes from old ones. As you can see, OOP introduces many new ideas and involves a different approach to programming than does procedural programming. Instead of concentrating on tasks, you concentrate on representing concepts. Instead of taking a top-down programming approach, you sometimes take a bottom-up approach. This book will guide you through all these points, with plenty of easily grasped examples.

Designing a useful, reliable class can be a difficult task. Fortunately, OOP languages make it simple to incorporate existing classes into your own programming. Vendors provide a variety of useful class libraries, including libraries of classes designed to simplify creating programs for environments such as Windows or the Macintosh. One of the real benefits of C++ is that it lets you easily reuse and adapt existing, well-tested code.

C++ and Generic Programming

Generic programming is yet another programming paradigm supported by C++. It shares with OOP the aim of making it simpler to reuse code and the technique of abstracting general concepts. But whereas OOP emphasizes the data aspect of programming, generic programming emphasizes the algorithmic aspect. And its focus is different. OOP is a tool for managing large projects, whereas generic programming provides tools for performing common tasks, such as sorting data or merging lists. The term ***generic*** refers to create code that is type independent. C++ data representations come in many types—integers, numbers with fractional parts, characters, strings of characters, and user-defined compound structures of several types. If, for example, you wanted to sort data of these various types, you would normally have to create a separate sorting function for each type. Generic programming involves extending the language

so that you can write a function for a generic (that is, not specified) type once and use it for a variety of actual types. C++ templates provide a mechanism for doing that.

The Genesis of C++

Like C, C++ began its life at Bell Labs, where Bjarne Stroustrup developed the language in the early 1980s. In Stroustrup's own words, "C++ was designed primarily so that my friends and I would not have to program in assembler, C, or various modern high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer" (Bjarne Stroustrup, ***The C++ Programming Language***, Third Edition. Reading, MA: Addison-Wesley, 1997).

Real-World Note: Bjarne Stroustrup's Home Page

Bjarne Stroustrup designed and implemented the C++ programming language and is the author of the definitive reference manuals *The C++ Programming Language* and *The Design and Evolution of C++*. His personal website at AT&T Labs Research should be the first C++ bookmark, or favorite, you create:

`www.research.att.com/~bs`

This site includes an interesting historical perspective of the hows and whys of the C++ language, Stroustrup's biographical material, and C++ FAQs. Surprisingly, Stroustrup's most frequently asked question is how to pronounce *Bjarne Stroustrup*. Download the `.WAV` file to hear for yourself!

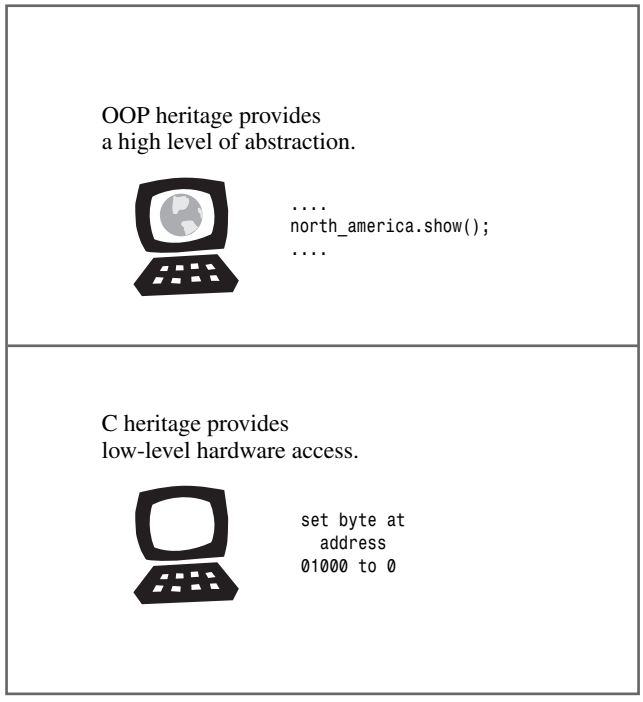
Stroustrup was more concerned with making C++ useful than with enforcing particular programming philosophies or styles. Real programming needs are more important than theoretical purity in determining C++ language features. Stroustrup based C++ on C because of C's brevity, its suitability to system programming, its widespread availability, and its close ties to the Unix operating system. C++'s OOP aspect was inspired by a computer simulation language called Simula67. Stroustrup added OOP features and generic programming support to C without significantly changing the C component. Thus C++ is a superset of C, meaning that any valid C program is a valid C++ program, too. There are some minor discrepancies, but nothing crucial. C++ programs can use existing C software libraries. ***Libraries*** are collections of programming modules that you can call up from a program. They provide proven solutions to many common programming problems, thus saving you much time and effort. This has helped the spread of C++.

The name *C++* comes from the C increment operator ++, which adds one to the value of a variable. Therefore, the name C++ correctly suggests an augmented version of C.

A computer program translates a real-life problem into a series of actions to be taken by a computer. While the OOP aspect of C++ gives the language the ability to relate to concepts involved in the problem, the C part of C++ gives the language the ability to get close to the hardware (see Figure 1.2). This combination of abilities has enabled the spread of C++. It may also involve a mental shift of gears as you turn from one aspect of a program to another. (Indeed, some OOP purists regard adding OOP features to C as being akin to adding wings to

a pig, albeit a lean, efficient pig.) Also, because C++ grafts OOP onto C, you can ignore C++'s object-oriented features. But you'll miss a lot if that's all you do.

FIGURE 1.2
C++ duality.



Only after C++ achieved some success did Stroustrup add templates, enabling generic programming. And only after the template feature had been used and enhanced did it become apparent that templates were perhaps as significant an addition as OOP—or even more significant, some would argue. The fact that C++ incorporates both OOP and generic programming, as well as the more traditional procedural approach, demonstrates that C++ emphasizes the utilitarian over the ideological approach, and that is one of the reasons for the language's success.

Portability and Standards

Say you've written a handy C++ program for the elderly Pentium PC computer at work, but management decides to replace the machine with a Macintosh G5—a computer using a different processor and a different operating system. Can you run your program on the new platform? Of course, you'll have to recompile the program, using a C++ compiler designed for the new platform. But will you have to make any changes to the code you wrote? If you can recompile the program without making changes and it runs without a hitch, we say the program is *portable*.

There are a couple obstacles to portability, the first of which is hardware. A program that is hardware specific is not likely to be portable. One that takes direct control of an IBM PC video board, for example, speaks gibberish as far as, say, a Sun is concerned. (You can minimize portability problems by localizing the hardware-dependent parts in function modules; then you just have to rewrite those specific modules.) We will avoid that sort of programming in this book.

The second obstacle to portability is language divergence. Certainly, that can be a problem with spoken languages. A Yorkshireman's description of the day's events may not be portable to Brooklyn, even though English is spoken in both areas. Computer languages, too, can develop dialects. Is the Windows XP C++ implementation the same as the Red Hat Linux implementation or the Macintosh OS X implementation? Although most implementers would like to make their versions of C++ compatible with others, it's difficult to do so without a published standard describing exactly how the language works. Therefore, the American National Standards Institute (ANSI) created a committee in 1990(ANSI X3J16) to develop a standard for C++. (ANSI had already developed a standard for C.) The International Organization for Standardization (ISO) soon joined the process with its own committee (ISO-WG-21), creating a joint ANSI/ISO effort to develop the a standard for C++. These committees met jointly three times a year, and we'll simply lump them together notationally as the ANSI/ISO committee. ANSI/ISO committee's decision to create a standard emphasizes that C++ has become an important and widespread language. It also indicates that C++ has reached a certain level of maturity, for it's not productive to introduce standards while a language is developing rapidly. Nonetheless, C++ has undergone significant changes since the ANSI/ISO committee began its work.

Work on the ANSI/ISO C++ Standard began in 1990. The committee issued some interim working papers in the following years. In April 1995 it released a Committee Draft (CD) for public comment. In December 1996 it released a second version (CD2) for further public review. These documents not only refined the description of existing C++ features but also extended the language with exceptions, runtime type identification (RTTI), templates, and the Standard Template Library (STL). The final International Standard (ISO/IEC 14882:1998) was adopted in 1998 by the ISO, International Electrotechnical Commission (IEC), and ANSI. 2003 brought the publication of the second edition of the C++ standard (IOS/IEC 14882:2003); the new edition is a technical revision, meaning that it tidies up the first edition—fixing typos, reducing ambiguities, and the like—but doesn't change the language features. This book is based on that standard.

The ANSI/ISO C++ Standard additionally draws on the ANSI C Standard because C++ is supposed to be, as far as possible, a superset of C. That means that any valid C program ideally should also be a valid C++ program. There are a few differences between ANSI C and the corresponding rules for C++, but they are minor. Indeed, ANSI C incorporates some features first introduced in C++, such as function prototyping and the `const` type qualifier.

Prior to the emergence of ANSI C, the C community followed a de facto standard based on the book ***The C Programming Language***, by Kernighan and Ritchie (Addison-Wesley Publishing

Company, Reading, MA, 1978). This standard was often termed K&R C; with the emergence of ANSI C, the simpler K&R C is now sometimes called ***classic C***.

The ANSI C Standard not only defines the C language, it also defines a standard C library that ANSI C implementations must support. C++ also uses that library; this book refers to it as the ***standard C library*** or the ***standard library***. In addition, the ANSI/ISO C++ standard provides a standard library of C++ classes.

More recently, the C Standard has been revised; the new standard, often called C99, was adopted by the ISO in 1999 and ANSI in 2000. This standard adds some features to C, such as a new integer type, that some C++ compilers support. Although not part of the current C++ Standard, these features may become part of the next C++ Standard.

Before the ANSI/ISO C++ committee began its work, many people accepted the most recent Bell Labs version of C++ as a standard. For example, a compiler might describe itself as being compatible with Release 2.0 or Release 3.0 of C++.

C++ continues to evolve, and work has already begun on producing the next version of the standard. The new version is informally labeled C++0X because the expected completion date is near the end of this decade, around 2009.

This book describes the ISO/ANSI C++ Standard, second edition (ISO/IEC 14882:2003), so the examples should work with any C++ implementation that is compatible with that standard. (At least, this is the vision and hope of portability.) However, the C++ Standard is still new, and you may find a few discrepancies. For example, if your compiler is not a recent version, it may lack namespaces or the newest template features. Support for the STL, described in Chapter 16, "The `string` Class and the Standard Template Library," is spotty for older compilers. Some older Unix systems use a front-end translator that passes the translated code to a C compiler that is not fully ANSI compatible, resulting in some language features being left unimplemented and in some standard ANSI library functions and header files not being supported. Even if a compiler does conform to the Standard, some things, such as the number of bytes used to hold an integer, are implementation dependent.

Before getting to the C++ language proper, let's cover some of the groundwork related to creating programs.

The Mechanics of Creating a Program

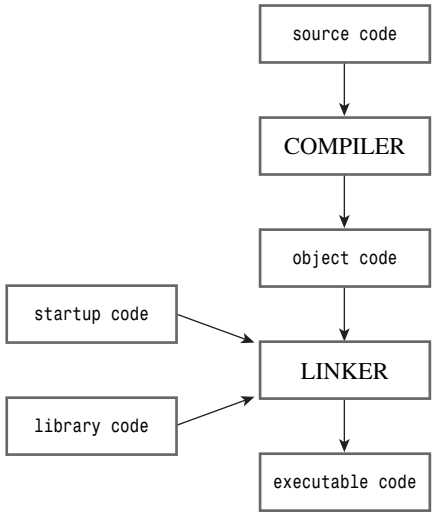
Suppose you've written a C++ program. How do you get it running? The exact steps depend on your computer environment and the particular C++ compiler you use, but they should resemble the following steps (see Figure 1.3):

1. Use a text editor of some sort to write the program and save it in a file. This file constitutes the ***source code*** for your program.
2. Compile the source code. This means running a program that translates the source code to the internal language, called ***machine language,*** used by the host computer. The file containing the translated program is the ***object code*** for your program.

3. Link the object code with additional code. For example, C++ programs normally use *libraries*. A C++ library contains object code for a collection of computer routines, called *functions*, to perform tasks such as displaying information onscreen or calculating the square root of a number. Linking combines your object code with object code for the functions you use and with some standard startup code to produce a runtime version of your program. The file containing this final product is called the *executable code*.

You will encounter the term *source code* throughout this book, so be sure to file it away in your personal random-access memory.

FIGURE 1.3
Programming steps.



The programs in this book are generic and should run in any system that supports modern C++. (However, you may need one of the latest versions to get support for namespaces and the newest template features.) The steps for putting together a program may vary. Let's look a little further at these steps.

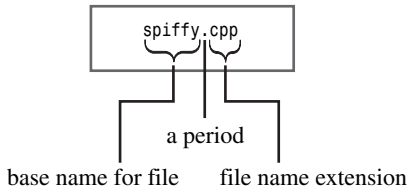
Creating the Source Code File

The rest of the book deals with what goes into a source file; this section discusses the mechanics of creating one. Some C++ implementations, such as Microsoft Visual C++, Borland C++ (various versions), Watcom C++, Digital Mars C++, and Metrowerks CodeWarrior, provide *Integrated Development Environments* (*IDE*s) that let you manage all steps of program development, including editing, from one master program. Other implementations, such as AT&T C++ or GNU C++ on Unix and Linux, and the free versions of the Borland and Digital Mars compilers, just handle the compilation and linking stages and expect you to type commands on the system command line. In such cases, you can use any available text editor to create and modify source code. On a Unix system, for example, you can use `vi` or `ed` or `ex` or `emacs`. On a DOS system, you can use `edlin` or `edit` or any of several available program editors. You can

even use a word processor, provided that you save the file as a standard DOS ASCII text file instead of in a special word processor format.

In naming a source file, you must use the proper suffix to identify the file as a C++ file. This not only tells you that the file is C++ source code, it tells the compiler that, too. (If a Unix compiler complains to you about a "bad magic number," that's just its endearingly obscure way of saying that you used the wrong suffix.) The suffix consists of a period followed by a character or group of characters called the *extension* (see Figure 1.4).

FIGURE 1.4
The parts of a source code filename.



The extension you use depends on the C++ implementation. Table 1.1 shows some common choices. For example, `spiffy.C` is a valid Unix C++ source code filename. Note that Unix is case sensitive, meaning you should use an uppercase C character. Actually, a lowercase `c` extension also works, but standard C uses that extension. So, to avoid confusion on Unix systems, you should use `c` with C programs and `C` with C++ programs. If you don't mind typing an extra character or two, you can also use the `cc` and `cxx` extensions with some Unix systems. DOS, being a bit simple-minded compared to Unix, doesn't distinguish between uppercase and lowercase, so DOS implementations use additional letters, as shown in Table 1.1, to distinguish between C and C++ programs.

TABLE 1.1 Source Code Extensions

C++ Implementation	Source Code Extension(s)
Unix	`C`, `cc`, `cxx`, `c`
GNU C++	`C`, `cc`, `cxx`, `cpp`, `c++`
Digital Mars	`cpp`, `cxx`
Borland C++	`cpp`
Watcom	`cpp`
Microsoft Visual C++	`cpp`, `cxx`, `cc`
Metrowerks CodeWarrior	`cpp`, `cp`, `cc`, `cxx`, `c++`

Compilation and Linking

Originally, Stroustrup implemented C++ with a C++-to-C compiler program instead of developing a direct C++-to-object code compiler. This program, called `cfront` (for *C front end*), translated C++ source code to C source code, which could then be compiled by a standard C compiler. This approach simplified introducing C++ to the C community. Other implementations have used this approach to bring C++ to other platforms. As C++ has developed and grown in popularity, more and more implementers have turned to creating C++ compilers that generate object code directly from C++ source code. This direct approach speeds up the compilation process and emphasizes that C++ is a separate, if similar, language.

Often the distinction between a `cfront` translator and a compiler is nearly invisible to the user. For example, on a Unix system the `CC` command may first pass your program to the `cfront` translator and then automatically pass the translator's output on to the C compiler, which is called `cc`. Henceforth, we'll use the term *compiler* to include translate-and-compile combinations. The mechanics of compiling depend on the implementation, and the following sections outline a few common forms. These sections outline the basic steps, but they are no substitute for consulting the documentation for your system.

If you have access to the `cfront` translator and know C, you may want to inspect the C translations of your C++ programs to get an inside look at how some C++ features are implemented.

Unix Compiling and Linking

The traditional C++ Unix system compiler is invoked with the `CC` command. However, these days a Unix computer instead might have no compiler, a proprietary compiler, or a third-party compiler, perhaps commercial, perhaps freeware, such as the GNU `g++` compiler. In many of these other cases (but not in the no-compiler case!), the `CC` command still works, with the actual compiler being invoked differing from system to system. For simplicity, you should assume that `CC` is available, but realize that you might have to substitute a different command for `CC` in the following discussion.

You use the `CC` command to compile your program. The name is in uppercase letters to distinguish it from the standard Unix C compiler `cc`. The `CC` compiler is a command-line compiler, meaning you type compilation commands on the Unix command line.

For example, to compile the C++ source code file `spiffy.C`, you would type this command at the Unix prompt:

```
CC spiffy.C
```

If, through skill, dedication, or luck, your program has no errors, the compiler generates an object code file with an `o` extension. In this case, the compiler produces a file named `spiffy.o`.

Next, the compiler automatically passes the object code file to the system linker, a program that combines your code with library code to produce the executable file. By default, the executable file is called `a.out`. If you used just one source file, the linker also deletes the

`spiffy.o` file because it's no longer needed. To run the program, you just type the name of the executable file:

```
a.out
```

Note that if you compile a new program, the new `a.out` executable file replaces the previous `a.out`. (That's because executable files take a lot of space, so overwriting old executable files helps reduce storage demands.) But if you develop an executable program you want to keep, you just use the Unix `mv` command to change the name of the executable file.

In C++, as in C, you can spread a program over more than one file. (Many of the programs in this book in Chapters 8, "Adventures in Functions," through 16 do this.) In such a case, you can compile a program by listing all the files on the command line, like this:

```
CC my.C precious.C
```

If there are multiple source code files, the compiler does not delete the object code files. That way, if you just change the `my.C` file, you can recompile the program with this command:

```
CC my.C precious.o
```

This recompiles the `my.C` file and links it with the previously compiled `precious.o` file.

You might have to identify some libraries explicitly. For example, to access functions defined in the math library, you may have to add the `-lm` flag to the command line:

```
CC usingmath.C -lm
```

Linux Compiling and Linking

Linux systems most commonly use `g++`, the GNU C++ compiler from the Free Software Foundation. The compiler is included in most Linux distributions, but it may not always be installed. The `g++` compiler works much like the standard Unix compiler. For example,

```
g++ spiffy.cxx
```

produces an executable file call `a.out`.

Some versions might require that you link in the C++ library:

```
g++ spiffy.cxx -lg++
```

To compile multiple source files, you just list them all in the command line:

```
g++ my.cxx precious.cxx
```

This produces an executable file called `a.out` and two object code files, `my.o` and `precious.o`. If you subsequently modify just one of the source code files, say `my.cxx`, you can recompile by using `my.cxx` and the `precious.o`:

```
g++ my.cxx precious.o
```

The Comeau C++ compiler (see www.comeaucomputing.com) is another possibility; it requires the presence of the GNU compiler. However, the Comeau compiler provides the most complete and rigorous implementation of the C++ standard.

The GNU compiler is available for many platforms, including the command-line mode for Windows-based PCs as well as for Unix systems on a variety of platforms.

Command-Line Compilers for Windows Command-Line Mode

The most inexpensive route for compiling C++ programs on a Windows PC is to download a free command-line compiler that runs in a Windows MS-DOS window. The MS-DOS version of the GNU C++ compiler is called `gpp`, and it is available at www.delorie.com/djgpp. Borland provides a free command-line compiler at www.borland.com/bcppbuilder/freecompiler. A slightly newer version of this compiler comes with the relatively inexpensive personal version of Borland C++BuilderX. Digital Mars has a free command-line compiler at www.digitalmars.com. The C++BuilderX installation is pretty automatic. For the rest, you need to read the installation directions carefully because parts of the installation processes are not automatic.

To use the `gpp` compiler, you first open an MS-DOS window. To compile a source code file named `great.cpp`, you type the following command at the prompt:

```
gpp great.cpp
```

If the program compiles successfully, the resulting executable file is named `a.exe`.

To use the Borland online compiler, you first open an MS-DOS window. To compile a source code file named `great.cpp`, you type the following command at the prompt:

```
bcc32 great.cpp
```

If the program compiles successfully, the resulting executable file is named `great.exe`.

Windows Compilers

Windows products are too abundant and too often revised to make it reasonable to describe them all individually. Popular ones include Microsoft, Borland, Metrowerks, and Digital Mars. Despite different designs and goals, they share some common features.

Typically, you must create a project for a program and add to the project the file or files constituting the program. Each vendor supplies an IDE with menu options and, possibly, automated assistance, in creating a project. One very important matter you have to establish is the kind of program you're creating. Typically, the compiler offers many choices, such as a Windows application, an MFC Windows application, a dynamic link library, an ActiveX control, a DOS or character-mode executable, a static library, or a console application. Some of these may be available in both 16-bit and 32-bit versions.

Because the programs in this book are generic, you should avoid choices that require platform-specific code, such as Windows applications. Instead, you want to run in a character-based mode. The choice depends on the compiler. For Microsoft Visual C++, you use the Win32 Console Application option. (If you are using Visual Studio .NET, you can also check the Empty Project option I Application Settings.) Metrowerks compilers offer a Win32 Console C++ App option and a Win32 WinSIOUX C++ App option, both of which work. (The former runs the compiled program in a DOS window; the latter runs it in a standard Windows

window.) Some Borland versions feature an EasyWin choice that emulates a DOS session; other versions offer a Console option. In general, you should look to see if there is an option labeled Console, character-mode, or DOS executable, and try that.

After you have the project set up, you have to compile and link your program. The IDE typically gives you several choices, such as Compile, Build, Make, Build All, Link, Execute, and Run (but not necessarily all these choices in the same IDE!):

- *Compile* typically means compile the code in the file that is currently open.
- *Build* or *Make* typically means compile the code for all the source code files in the project. This is often an incremental process. That is, if the project has three files, and you change just one, then just that one is recompiled.
- *Build All* typically means compile all the source code files from scratch.
- As described earlier, *Link* means combine the compiled source code with the necessary library code.
- *Run* or *Execute* means run the program. Typically, if you have not yet done the earlier steps, Run does them before trying to run a program.

A compiler generates an error message when you violate a language rule and identifies the line that has the problem. Unfortunately, when you are new to a language, you may find it difficult to understand the message. Sometimes the actual error may occur before the identified line, and sometimes a single error generates a chain of error messages.

Tip

When fixing errors, fix the first error first. If you can't find it on the line identified as the line with the error, check the preceding line.

Be aware that the fact that a particular compiler accepts a program doesn't necessarily mean that the program is valid C++. And the fact that a particular compiler rejects a program doesn't necessarily mean that the program is invalid C++. Current compilers are more compliant with the Standard than their predecessors of two or three years ago. At this time, the Comeau compiler (and other users of the Edison Design Group front end) comes closest an exact image of the standard.

Tip

Occasionally, compilers get confused after incompletely building a program and respond by giving meaningless error messages that cannot be fixed. In such cases, you can clear things up by selecting Build All to restart the process from scratch. Unfortunately, it is difficult to distinguish this situation from the more common one in which the error messages merely seem to be meaningless.

Usually, the IDE lets you run the program in an auxiliary window. Some IDEs close the window as soon as the program finishes execution, and some leave it open. If your compiler closes the window, you'll have a hard time seeing the output, unless you have quick eyes and a photographic memory. To see the output, you must place some additional code at the end of the program:

```
    cin.get();  // add this statement
    cin.get();  // and maybe this, too
    return 0;
}
```

The `cin.get()` statement reads the next keystroke, so this statement causes the program to wait until you press the Enter key. (No keystrokes get sent to a program until you press Enter, so there's no point in pressing another key.) The second statement is needed if the program otherwise leaves an unprocessed keystroke after its regular input. For example, if you enter a number, you type the number and then press Enter. The program reads the number but leaves the Enter keystroke unprocessed, and it is then read by the first `cin.get()`.

The Borland C++Builder compiler departs a bit from more traditional designs. Its primary aim is Windows programming. To use older versions for generic programs, you select File, New. Then you select Console App. A window opens that includes a skeleton version of `main()`. You should retain the following two nonstandard lines if they appear in the skeleton:

```
#include <vcl\condefs.h>
#pragma hdrstop
```

For C++BuilderX, select File, New, New Console. You don't get a skeleton `main()`. Instead, you need to select File, New File and add a new `.cpp` file to the project.

C++ on the Macintosh

The primary Macintosh C++ compiler is Metrowerks CodeWarrior. It provides project-based IDEs that are similar, in basic concepts, to what you would find in a Windows compiler. You start by selecting File, New Project. You are then given a choice of project types. For CodeWarrior, choose MacOS:C/C++:ANSI C++ Console in older versions, or MacOS:C/C++:Standard Console:Std C++ Console in mid-vintage versions, or MacOS C++ Stationery:Mac OS Carbon:Standard Console:C++ Console Carbon. (You can make other valid choices; for example, you might opt for Classic instead of Carbon or C++ Console Carbon Altivec instead of plain C++ Console Carbon.)

CodeWarrior supplies a small source code file as part of the initial project. You can try compiling and running that program to see if you have your system set up properly. However, after you provide your own code, you should delete this file from the project. You do so by highlighting the file in the project window and then selecting Project, Remove.

Next, you must add your source code to the project. You can use File, New to create a new file or File, Open to open an existing file. You should use a proper suffix, such as `.cp` or `.cpp`. You use the Project menu to add this file to the project list. Some programs in this book require that you add more than one source code file. When you are ready, you select Project, Run.

Tip

To save time, you can use just one project for all the sample programs. You should delete the previous sample source code file from the project list and add the current source code. This saves disk space.

The compiler includes a debugger to help you locate the causes of runtime problems.

Summary

As computers have grown more powerful, computer programs have become larger and more complex. In response to these conditions, computer languages have evolved so that it's easier to manage the programming process. The C language incorporated features such as control structures and functions to better control the flow of a program and to enable a more structured, modular approach. To these tools C++ adds support for object-oriented programming and generic programming. This enables even more modularity and facilitates the creation of reusable code, which saves time and increases program reliability.

The popularity of C++ has resulted in a large number of implementations for many computing platform; the ISO/ANSI C++ Standard provides a basis for keeping these many implementations mutually compatible. The Standard establishes the features the language should have, the behavior the language should display, and a standard library of functions, classes, and templates. The Standard supports the goal of a portable language across different computing platforms and different implementations of the language.

To create a C++ program, you create one or more source files containing the program as expressed in the C++ language. These are text files that must be compiled and linked to produce the machine-language files that constitute executable programs. These tasks are often accomplished in an IDE that provides a text editor for creating the source files, a compiler and a linker for producing executable files, and other resources, such as project management and debugging capabilities. But the same tasks can also be performed in a command-line environment by invoking the appropriate tools individually.