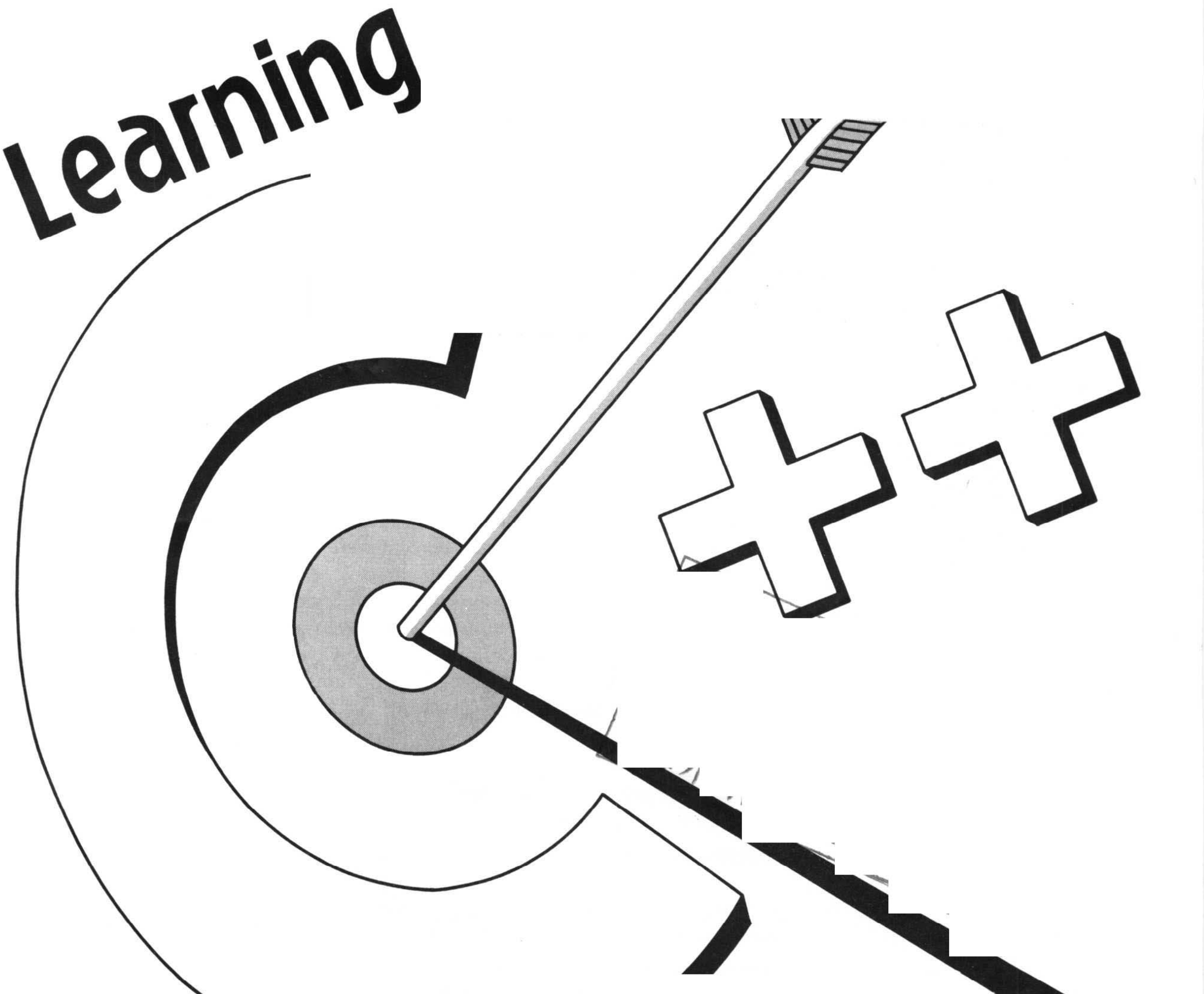
Learni

C++

Neill Graham

Learning C++

Neill Graham



Irwin McGraw-Hill

Boston, Massachusetts Burr Ridge, Illinois Dubuque, Iowa
Madison, Wisconsin New York, New York San Francisco, California St. Louis, Missouri

Irwin/McGraw-Hill

A Division of The ***McGraw-Hill*** *Companies*

This book was set in Bembo type by TBH Typecast, Inc.
Development and production management were provided by Cole and Associates.
The editors were David M. Shapiro and Elliot Simon;
production coordination was done by Simon & Assocs.
The cover and interior were designed by Seventeenth Street Studios.
Illustrations were done by Advanced Presentations.
R. R. Donnelley & Sons Company was printer and binder.

This book is printed on acid-free paper.

LEARNING C++

7 8 9 0 DOC/DOC 9 0 9 8 7

ISBN 0-07-023983-5

Library of Congress Cataloging-in-Publication Data

Graham, Neill, (date).
Learning C++ / Neill Graham.
p. cm.
ISBN 0-07-023983-5
1. C++ (Computer program language) I. Title. II. Title: Learning C Plus Plus.
QA76.73.C153G73 1991
00513'3–dc20 90-44655

Preface

This book is an introduction to C++ and object-oriented programming for readers who know at least one programming language (which will probably be Pascal) but who are not necessarily familiar with C.

C++ and C

C++, an object-oriented extension of C, addresses the needs of two recent trends: the widespread use of C as a software implementation language, and the increasing interest in object-oriented programming.

C is by far the most popular language for professional software development on minicomputers and microcomputers. While offering all the known advantages of a higher-level language, C also provides the low-level access to hardware and system software that is characteristic of assembly language. Many microcomputer software packages that were originally written in assembly language have now been translated into C.

Object-Oriented Programming

Object-oriented programming allows a complex program to be built out of much simpler constructs called *objects*, which interact by exchanging messages. The following are characteristic of object-oriented programming:

- *Classes.* Objects are defined by means of classes; once a class has been defined, any number of objects of that class are easily created. Programmers are thus encouraged to reuse code by defining general purpose classes and using them in many different applications.
- *Abstraction.* To use an object, we need only know its public interface—what messages it understands and how it responds to each. We *do not* need to know anything about its internal workings, which are hidden from the object's users.
- *Inheritance.* A new class can be derived from one or more existing classes and can inherit some or all of their properties. This further encourages code reuse because classes derived from a general-purpose class can be customized as needed for each particular application.
- *Polymorphism.* Different kinds of objects can understand the same message, though they may respond to it in different ways. We can send such a message without knowing what kind of object is involved. For example, objects representing different geometrical figures might all respond to a message to draw the corresponding figure on the screen; we could send a `draw` message to any such object and be assured that the appropriate figure will be drawn.

About This Book

The background in C necessary for effective use of this book is provided in Chapter 1 (program structure, data types, expressions, and control statements) and in Chapter 3 (arrays, pointers, and strings).

The study of object-oriented programming begins in Chapter 2, which introduces the basic concepts and develops a simple bank account class as an example. Although Chapter 3 is devoted mainly to the elements of array and pointer manipulation, it does introduce the C++ `new` and `delete` operators, and it defines a class of dynamic lookup tables to illustrate all the material in the chapter.

Chapter 4 covers operator and function overloading as well as friend functions and operators. This chapter introduces a different approach to object-oriented programming, in which objects are passed as arguments to overloaded friend functions and operators rather than being sent messages via member functions.

In Chapters 5 and 6 we return to more conventional object-oriented programming. Chapter 5 develops inheritance; although multiple inheritance is introduced and illustrated, most of the discussion and examples focus on single inheritance, which beginners should master before going on to the more complicated multiple inheritance. Chapter 6 introduces polymorphism and its implementation via virtual functions; abstract base classes and pure virtual functions are also covered.

Chapter 7 develops a discrete-event simulation as a case study. Most examples in the preceding chapters have involved a conventional main program interacting with one or more objects; a major goal of Chapter 7 is to present a program whose computations proceed mainly via message passing between objects. This simulation *does not* use the C++ task library, which is not available in all implementations and which hides some details that it is instructive to work through.

Chapter 8 is devoted to further exploration of the input-output library. A complete discussion of this intricate library is beyond the scope of this book; instead, we focus on the following areas: the stream classes (which provide a good example of multiple inheritance); controlling the format of output and the expected format of input; detecting errors and end-of-file; using named files, such as disk files; using command-line parameters; and a brief introduction to direct access with `tellg()` and `seekg()`.

Appendix 1 provides a list of reserved words, and Appendix 2 provides a chart showing the precedence and associativity of operators. A comprehensive glossary defines important terms in C++ and object-oriented programming, and the "For Further Study" section suggests additional readings.

AT&T Releases and Turbo C++ Appendix

C++ was developed at AT&T, and for now the AT&T compiler and reference manual set the standard for the language. Most current software and books are based on AT&T Release 2.0, which was expected to be the final revision of the language prior to the development of an ANSI standard for C++.

To correct some problems with Release 2.0, however, a small number of changes were made. These resulted in Release 2.1, which appeared while this book was in the final stages of preparation. This book is based on Release 2.0; however, the most important changes in Release 2.1 are discussed briefly, mainly in footnotes.

The example programs in this book have also been tested with Turbo C++, which is compatible with AT&T Release 2.0. As the first C++ implementation from a major supplier of microcomputer software, Turbo C++ is expected to enjoy a substantial share of the market for MS-DOS C++ implementations. Appendix 3 is an introduction to Turbo C++ and its integrated development environment. Also discussed are a few problems in Version 1.00 of Turbo C++ that must be worked around in order to run some of the example programs.

Acknowledgments

For their helpful comments on various drafts of the manuscript, the author expresses his appreciation to J. Eugene Ball, University of Delaware; Krys J. Kochut, University of Georgia; Robert McCoard, California State University, Northridge; Tomasz Müldner, Acadia University; Roy Ogawa, University of Nevada, Las Vegas; and Steven Stepanek, California State University, Northridge.

Contents

PREFACE, VII

1 Elements of C++

HELLO, WORLD!, 1
Comments, 1
Source Files, 1
Header Files and `#include` *Directives,* 3
Functions and `main()`, 4
Input and Output, 4
FUNCTION DECLARATIONS, DEFINITIONS, AND CALLS, 7
IDENTIFIERS, 9
VALUES, TYPES, AND CONSTANTS, 10
Integer Types, 10
Floating Types, 14
DATA OBJECTS, 15
Declaration, Initialization, and Assignment, 16
Constant Data Objects, 18
Scopes of Identifiers, 19
Lifetimes of Data Objects, 20
OPERATORS AND EXPRESSIONS, 21
Arithmetic Operators, 22
Assignment Operators, 23
Increment and Decrement Operators, 26
Equality, Relational, and Logical Operators, 27
Conditional Expressions, 29
The Comma Operator, 30
Functions That Take Arguments and Return Values, 31
Side Effects and Sequence Points, 33
TYPE CONVERSIONS, 35
Implicit Conversions, 36
Explicit Conversions, 37
EXAMPLE: COMPUTING DISCOUNTS, 39
CONTROL STATEMENTS: REPETITION, 42
The `while` *Statement,* 42
The `do` *Statement,* 43
Example: The Inverse Fibonacci Rabbit Problem, 44
The `for` *Statement,* 46
Example: Computing Interest, 48
CONTROL STATEMENTS: SELECTION, 52
The `if` *and* `if-else` *Statements,* 52
The `switch` *Statement,* 56
EXERCISES, 59

2 Classes and Objects

OBJECT-ORIENTED PROGRAMMING, 61
Objects and Classes, 61
Methods, Messages, and Data Hiding, 63

CLASSES, 63
Classes Declared with `struct`, 64
Member Functions, 66
Constructors and Destructors, 70
Classes Declared with `class`, 73
Compiling and Linking, 74
Example: A Program Using Class `account`, 80
MORE ABOUT MEMBER FUNCTIONS, 81
Function Overloading, 81
Default Arguments, 87
Inline Expansion, 88
Constant Objects and Functions, 94
LINKAGE, 96
EXERCISES, 98

3

Arrays, Pointers, and References

ARRAYS, 102
POINTERS, 107
The Address-of Operator, 109
Pointers and Arguments, 110
Pointers to Class Objects, 112
ARRAYS AND POINTERS, 113
Array Names as Pointers, 113
Subscripting Pointers, 115
Pointer Arithmetic, 116
Using `const` *in Pointer Declarations,* 120
STRINGS, 122
FUNCTIONS FOR STRING PROCESSING, 125
REFERENCE TYPES, 131
MEMORY MANAGEMENT WITH `new` AND `delete`, 134
TYPE-NAME DEFINITIONS, 138
EXAMPLE: CLASS `table`, 139
Class Declarations, 140
Class Implementation, 142
EXAMPLE: A PROGRAM USING CLASS `table`, 149
Function Declarations, 149
Declaring and Setting `_new_handler`, 153
Reading Characters and Strings, 155
Using a Pointer to Constant Data, 158
EXERCISES, 158

4

Operators and Friends

OPERATOR OVERLOADING, 163
FRIEND FUNCTIONS AND OPERATORS, 168
CLASS `vector`, 171
Constant Definitions, 174
The Component Array, 174
Constructors, 175
Memberwise Initialization and Assignment, 175
Addition, Subtraction, and Multiplication, 179
Equality Operators, 181
Assignment Operators, 181
Subscripting Operator, 184
Conversion to a Scalar Magnitude, 185
Input and Output, 186
Demonstration Program, 187
DEFENDING AGAINST MULTIPLE INCLUSIONS, 189
CLASS `string`, 191
String Representation, 192
Copying and Reference Counts, 192
Construction, Destruction, and Assignment, 197
Concatenation, 205
Comparisons, 208
Function `pos()`, 209
Function `length()`, 210
Function `refs()`, 210
Subscripting, 211
Substring Extraction, 212
Input and Output, 213
Demonstration Program, 214
EXERCISES, 218

5
Inheritance: Derived Classes

EXAMPLE: BANK ACCOUNTS, 221
Class `account` *and the Keyword* `protected`, 222
Class `sav_acct`, *Public Base Classes, and Constructors,* 227
Class `chk_acct`, 230
Class `time_acct` *and Redefining Inherited Functions,* 232
DEQUES, QUEUES, AND STACKS, 234
Implementation of Class `deque`, 238
LINKED LISTS, 243
Classes `list` *and* `node`, 244
Defining Derived Classes, 249
Using `ilist` *and* `inode`, 252
EXTENDING AN EXISTING CLASS, 254
MULTIPLE INHERITANCE, 257
Virtual Base Classes, 260
EXERCISES, 264

6
Polymorphism: Virtual Functions

POINTERS, REFERENCES, AND VIRTUAL FUNCTIONS, 267
Heterogeneous Lists, 269
Virtual Functions, 269
EXAMPLE: BREEDS OF DOGS, 273
ABSTRACT BASE CLASSES AND PURE VIRTUAL FUNCTIONS, 278
EXAMPLE: GRAPHICS FIGURES, 280
Using ANSI Escape Sequences, 282
Class `figure`, 283
Class `block`, 284
Class `box`, 288
Class `triangle`, 289
Class `label`, 290
Example Program: Heterogeneous Linear *List,* 292
Example Program: Heterogeneous Linked *List,* 295
EXAMPLE: EXPRESSION TREES, 299
Class Declarations: Structure of Expression Trees, 303
Class Declarations: Expression Evaluation, 304
EXAMPLE: STATE MACHINES, 305
Static Class Members: Class Variables and Functions, 309
Class `state`, 310
Classes `state_S`, `state_W`, *and* `state_P`, 315
Demonstration Program, 316
EXERCISES, 318

7
Case Study: Event-Driven Simulation

THE MACHINE-ADJUSTMENT PROBLEM, 323
Discrete, Event-Driven Simulations, 323
THE WORKINGS OF CHANCE, 326
The Geometric Distribution, 327
Class `geometric`, 328
ACTIVE OBJECTS, 334
Class `active`, 336
Class `machine`, 336
Class `adjuster`, 342
MANAGING AND COORDINATING RESOURCES, 345
Resource Coordinators, 346
Class `manager`, 347
THE SCHEDULER, 351
Class `scheduler`, 352
THE MAIN PROGRAM, 357
Sample Run, 362
EXERCISES, 363

8
More About Input and Output

THE BASIC INPUT-OUTPUT CLASSES, 367
FORMAT-STATE FLAGS, 369
FORMAT-STATE PARAMETERS, 372
SETTING FLAGS AND PARAMETERS, 372
FORMATTING EXAMPLES, 375
DETECTING ERRORS AND END-OF-FILE, 379
CLASSES FOR NAMED FILES, 381
INPUT AND OUTPUT WITH NAMED FILES, 382
USING COMMAND-LINE PARAMETERS, 387
DIRECT ACCESS, 389
EXERCISES, 392

Appendix 1
C++ KEYWORDS, 394

Appendix 2
OPERATOR PRECEDENCE AND ASSOCIATIVITY, 395

Appendix 3
TURBO C++, 397

GLOSSARY, 409
FOR FURTHER STUDY, 417
INDEX, 418

1 Elements of C++

THIS CHAPTER introduces the basic C++ constructions: variable declarations, arithmetic expressions, function calls and definitions, and input, output, and control statements. Because these constructions occur in many programming languages the concepts behind them will already be familiar, so we will be able to concentrate on how these concepts are expressed in C++.

HELLO, WORLD!

It is a tradition in C and C++ to begin any introductory text or programming course by discussing a program that prints the message "Hello, world!" Listing 1-1 shows the hello-world program; we will use an extended discussion of this program as a framework for introducing several C++ concepts.

Comments

The first two lines in Listing 1-1 are comments, although they will not seem to be such to readers familiar with C. Comments in C are enclosed between the symbols /* and */.

```
/* This is a comment */
```

Such a comment can extend over any number of lines.

```
/* Now is the time
   for all good programmers
   to put comments in their programs */
```

Listing 1-1

```
// File hello.cpp
// Program to print a greeting to the user

#include <iostream.h>

main()
{
   cout << "Hello, world!\n";
}
```

This style of comment is also allowed in C++. However, C++ also provides another style, in which a comment begins with // and extends to the end of the line. Since the comment always extends to the end of the line, no closing symbol (such as */) is needed:

```
// This is a comment

// Now is the time
// for all good programmers
// to put comments in their programs
```

The latter style is usually the easiest to write and is the style generally preferred by C++ programmers.

Source Files

The text of a program is stored on disk in one or more *source files*; each source file is printed in this book as a *listing*. At the beginning of each source file is a comment giving the name of the file; the name of the source file for the hello-world program is `hello.cpp`.

In this book we assume that the names of C++ source files end with `.cpp`; however, some C++ implementations use `.c`, `.C`, `.cp`, or `.cxx`. If the file-naming conventions for your computer system or C++ implementation differ from those assumed here, you will not be able to use the file names in this book unchanged. Nevertheless, they will still help you keep track of the source files listed here, particularly for programs having more than one source file.

Header Files and `#include` ***Directives***

Every C++ implementation comes with a *library* of predefined functions, operators, and other entities. Programmers are encouraged to use these predefined entities, but are required to declare them in each source file in which they are used. To prevent programmers from having to memorize the necessary declarations and write them out repeatedly, the implementation provides a number of *header files*, each of which contains the declarations for a certain part of the library.

The first step in compiling a C or C++ program is carried out by the *preprocessor*, which manipulates the text of the program in accordance with *preprocessor directives*, all of which begin with the symbol `#`. A header file is inserted into a program with an `#include` directive, which the preprocessor replaces with the contents of the header file. The hello-world program uses the library facilities for stream output; the necessary declarations are contained in the header file `iostream.h`,* which the program includes with the directive

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

The angle brackets enclosing the file name are made up of the less-than sign, <, and the greater-than sign, >. As with source files, the naming conventions for header files vary. In some implementations, `iostream.h` would be named `iostream.hpp` or `iostream.hxx`.

The part of the library declared in a particular header file is often referred to by the name of the header file. Thus, the part of the library declared in `iostream.h` is referred to as the *iostream library*, the part of the library declared in `math.h` is referred to as the *math library*, and so on.

* Earlier versions of C++ (those preceding AT&T Release 2.0) used the header file `stream.h` instead of `iostream.h`; thus `#include` directives for `stream.h` will be found in many existing C++ programs. Although Release 2.0 provides a header file `stream.h` for use by existing programs, `iostream.h` is recommended for new programs.

Functions and `main()`

The basic building blocks of C++ programs are *functions*, which correspond to functions, procedures, and subroutines in other languages. Each function carries out a well-defined operation and is *called* or *invoked* whenever that operation is needed. Functions may be defined in the program, or they may be predefined functions from the library. When library functions are used, the header files containing the necessary declarations must be included.

Parentheses are associated with functions in that they always appear in function declarations, definitions, and calls. When function names appear in running text, it is a convention of C and C++ to follow each function name with a pair of parentheses to indicate that the name refers to a function. Thus, a function named *main* is referred to as `main()` rather than `main`.

The system executes a C++ program by calling the function `main()`; therefore, every C++ program must define `main()`. Simple programs, such as the hello-world program, define only `main()`; more complex programs also define other functions, which are called directly or indirectly from `main()`.

In Listing 1-1, the definition of `main()` has the following form:

```
main()
{
    . . .
}
```

The empty parentheses following the function name `main` indicate that function `main()` takes no arguments; that is, no data is passed to it when it is called. The braces, `{` and `}`, which correspond to the keywords `BEGIN` and `END` in some other languages, enclose a *block*, which is a basic unit for grouping declarations and statements. The statements defining a function are always enclosed in a block. In Listing 1-1, the block that defines `main()` contains only a single statement.

Input and Output

The C++ library provides facilities for input and output that are far more convenient than the library functions that C programmers use. In C++, input is read from and output is written to *streams*. When `iostream.h` is included in a program, several standard streams are defined automatically. The stream `cin` is used for input, which is normally read from the user's keyboard. The stream `cout` is used for output, which is normally sent to the user's display. Some operating systems allow these streams to be redirected so that, for example, input from `cin` could be read from a disk file and output to `cout` could be written to another disk file.

The *insertion operator*, <<, inserts data into a stream. Thus the statement

```
cout << 500;
```

places the value 500 in the standard output stream `cout`. If the output stream has not been redirected, the number 500 will be printed on the user's display.

A series of << operators can be used to output several data values with a single statement. For example,

```
cout << 500 << 600 << 700;
```

outputs three numbers. However, the insertion operator does not separate printed items with spaces, so the three numbers will be run together in the printed output.

```
500600700
```

A *string* is a sequence of characters. In a C++ program, a string is represented by a *string literal*, which is a series of characters enclosed in quotation marks. When a string literal is inserted in `cout`, the characters making up the string (but not the enclosing quotation marks) are printed. Thus, the statement

```
cout << "This is a string";
```

prints

```
This is a string
```

Likewise,

```
cout << 500 << ", " << 600 << ", " << 700;
```

prints

```
500, 600, 700
```

C++ uses *escape sequences* to represent characters that are not represented by traditional symbols such as `a`, `b`, and `c`. An escape sequence consists of a backslash, \, followed by a letter or number representing the character. The entire escape sequence represents a single character. The following are some common escape sequences:

`\a`	alert—causes the computer or terminal to beep
`\n`	newline—causes the display or printer to start printing on a new line
`\t`	tab—causes the display or printer to jump to a predefined tab stop, as if the tab key had been pressed
`\"`	quotation mark inside a string literal
`\\`	backslash

Because quotation marks indicate where a string literal begins and ends, an escape sequence must be used to get a quotation mark inside a string literal. Thus the statement

```
cout << "\"Hello,\" she said.\n";
```

prints

```
"Hello," she said.
```

and (because of the final `\n`) positions the display or printer at the beginning of the next line.

Because a backslash always signals an escape sequence, the backslash character itself must be represented by an escape sequence, `\\`. This must be borne in mind particularly by users of the MS-DOS and OS/2 operating systems, which use backslashes in file names.

Now, finally, we are in a position to understand the single statement in the hello-world program. The statement

```
cout << "Hello, world!\n";
```

prints the message

```
Hello, world!
```

and positions the display or printer at the beginning of the next line.

Note that each statement ends with a semicolon. Statements constructed with operators and function calls are known as *expression statements*; an expression statement always ends with a semicolon.