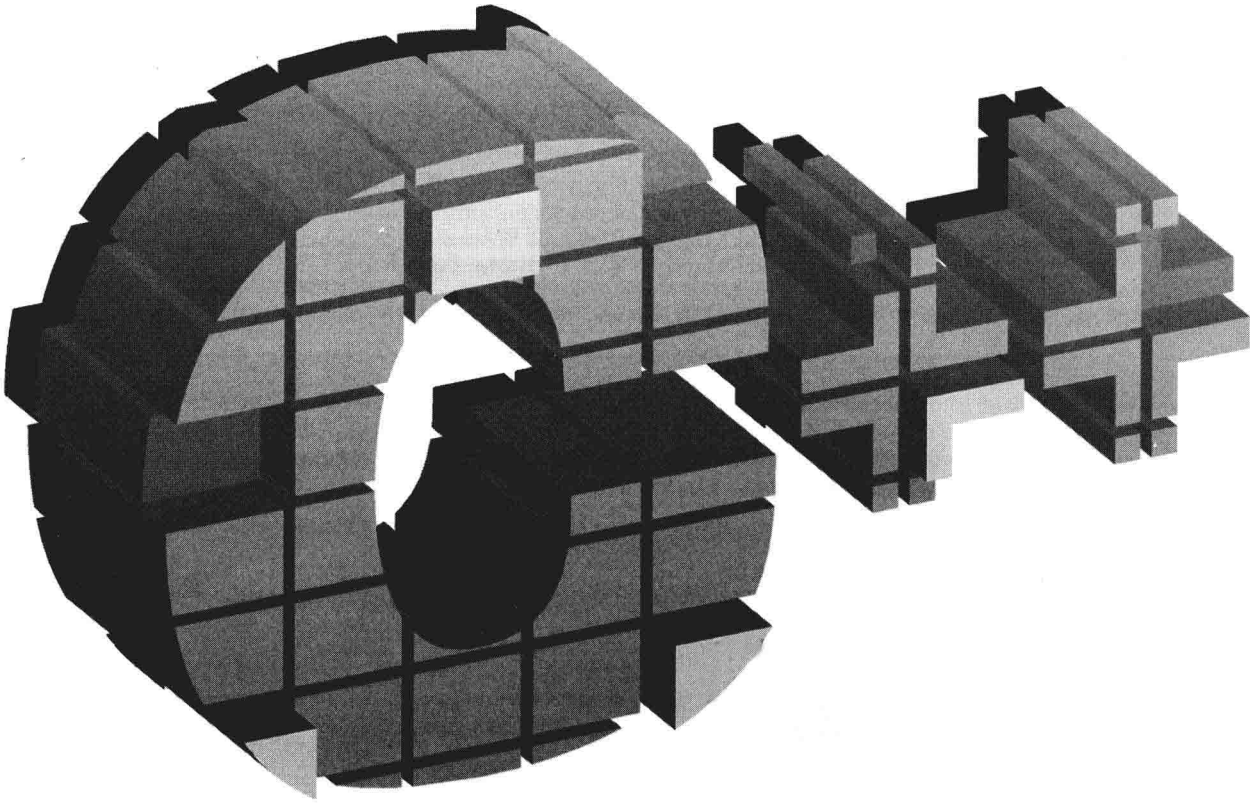
FUNDAMENTALS OF PROGRAMMING

AN INTRODUCTION TO COMPUTER PROGRAMMING USING C++

RICHARD HALTERMAN

FUNDAMENTALS OF PROGRAMMING

AN INTRODUCTION TO COMPUTER PROGRAMMING USING C++



RICHARD HALTERMAN

B&E TECH
Business and Educational Technologies
A Division of Wm. C. Brown Communications, Inc.

Business and Educational Technologies
A Division of Wm. C. Brown Communications, Inc.

Vice President and Publisher *Susan A. Simon*
Acquisitions Editor *Linda Meehan Avenarius*
Sales Manager *Paul Ducham*
Advertising/Marketing Coordinator *Jennifer Wherry Finders*
Product Development Assistant *Carrie Langas*

Wm. C. Brown Communications, Inc.

Chief Executive Officer *G. Franklin Lewis*
Corporate Senior Vice President and Chief Financial Officer *Robert Chesterman*
Corporate Senior Vice President and President of Manufacturing *Roger Meyer*
Executive Vice President/General Manager, Brown & Benchmark Publishers *Tom Doran*
Executive Vice President/General Manager, Wm. C. Brown Publishers *Beverly Kolz*

A Times Mirror Company

Library of Congress Catalog Card Number: 94-78011

ISBN 0-697-25110-1

Printed in the United States of America by Wm. C. Brown Communications, Inc.,
2460 Kerper Boulevard, Dubuque, IA 52001

10 9 8 7 6 5 4 3 2 1

PREFACE

C++ is becoming a dominant programming language for developing today's computer software systems. It offers the runtime efficiency and economy of expression of C and adds advanced features for data abstraction and object-oriented programming. Knowledge of C++ is a valuable asset for any individual seeking employment in computer programming or systems development. Application software for modern operating environments and graphical user interfaces is best developed in an object-oriented fashion; indeed, most of these environments and graphical user interfaces work best when developed in an object-oriented fashion. Many of these environments have C++ based "application frameworks" that insulate the programmer from many of the complicated details that must be managed by programs designed to run on these systems.

Knowledge of the structure and syntax of the C++ language is useless without well-honed fundamental programming skills. The features that have been a part of higher-level procedural programming languages for nearly forty years—conditional execution (`if` statements), loops, array manipulation, function calls, etc.—must still be mastered to do competent object-oriented programming in C++. The principles presented in the first seven chapters of this book are not that much different from the principles presented in an introductory Pascal programming textbook from ten years ago. The language is different and admittedly offers some additional flexibility, but the underlying principles are the same. Pascal does provide a distinct pedagogical advantage over C++; namely, it was designed to teach the principles of solid structured programming. It imposes certain restrictions that C++ does not. It has a more limited way of doing almost everything that C++ can do. These limitations give a novice programmer fewer opportunities for misusing language features that lead to errors in programs that are difficult to resolve without knowledgeable assistance. While used quite successfully in some niches, Pascal is not a popular programming language for commercial software development, largely due to its label as an educational language. C++, the *programmer-friendly* language, can be intimidating for beginning programmers. It offers fewer safety nets than Pascal. It was designed to get the job done efficiently and not get in the way of a competent programmer.

Is it possible to teach students with no programming experience how to program using C++? Why not introduce the students to programming with Pascal (or another similar language) and then use C++ when advanced concepts are to be addressed? This approach has its merits, but consider the advantages of using C++ from start to finish:

- Students generally feel most comfortable using the language that they first learned. This applies to programming languages as well as conversational natural languages. This is not to say that the second programming language cannot be readily embraced and displace the original language as "most favored." (Indeed, it is almost assured that the strongest advocates for C++ did not learn how to program using C++, and most likely did not even get started with its predecessor, C.) Students who must eventually develop more advanced programs appreciate not having to take the time to learn a language that will be quickly discarded when the "real work" begins.
- The process of "unlearning" a programming language when confronted with a new language is nontrivial, especially when the structure of the two languages is similar. Students learning C++ trained in Pascal or Modula-2 (or BASIC for that matter) can become frustrated with minor differences that consume a lot of time to get right (assignment vs. equality operator, zero-based array subscripting, use of semicolons). Even though the student may totally understand the programming principle involved, these minor syntactical differences can sometimes require hours in front of the computer to correct.

- C++ can be introduced in a way that promotes good programming style and sound design principles as promoted in Pascal or Modula-2. It takes a bit of work with attention to pointing out possible pitfalls that the compiler will not detect, but the process is not as difficult as it might appear. For example, multiple initializations might be possible within a C++ `for` statement, but the single counter variable/single initialization approach (required by Pascal) can be presented as the best way to go.
- When advanced principles such as data abstraction, elementary data structures, and object-oriented programming are introduced, students are not burdened with simultaneously learning a new language and, possibly, a new development environment. These concepts are difficult enough without subjecting students to the first point mentioned above.

Nearly every other C++ book assumes that the reader has some prior programming experience. Most assume a knowledge of C. This book starts at the beginning. The basics of computer hardware are briefly introduced so that some basic terms, when used in subsequent chapters, will be understood well enough to support the discussion of programming concepts in which they are involved. After a brief history of C++ and a discussion of some general software development issues, the actual task of computer programming begins.

The main objectives of this text are outlined below:

- To develop the ability to correctly analyze a variety of problems and generate appropriate algorithmic solutions
- To instill the principles of top-down, structured design when using the procedural programming paradigm
- To introduce the concepts of object-oriented programming
- To explore the syntax and usage of the C++ programming language as a means of accomplishing all of the above objectives.

History shows that it is unrealistic to master a particular programming language with the intention of using it alone for the rest of one's programming career (assuming that one's career exceeds ten years). It was mentioned that few of today's C++ programmers learned to program using C++. (C++ is still a relatively young language, after all.) These programmers had to adapt their existing skills to a new language. These *generic* programming skills are much more valuable in the long run than is complete knowledge of the syntax of C++ with no idea of how to use it productively. Programming languages come and go; in ten years, C++ will likely be eclipsed by some super post-object-oriented visual code generator. Despite the differences in the programming paradigms that evolve over time, all software development requires well-developed logical and sequential reasoning skills. The problem to be solved must be understood completely, and a plan of attack must be correctly formulated given the tools at hand. Forty years ago, the tools consisted of computer machine language, teletypewriter terminals, and little else. Today, graphical user interfaces, higher-level languages like C++, source code debuggers, visual interface builders, comprehensive class libraries, and code generators are available to expedite the development process. All of the fancy tools available to the modern programmer are useless without the underlying problem analysis, organizational, and logical reasoning skills. As the tools get better, the problems get harder. Relevant software development today is just as difficult, or more so, than it was forty years ago.

While a large portion of the book is devoted to explaining C++ syntax and usage, the examples provide exposure to a variety of problem types that are typically encountered by programmers. Their programming solutions provide an insight into the problem-solving process. The best way to learn how to solve problems is to solve problems that are similar to problems whose solutions are known. Many of the programming assignments found at the end of each chapter are modifications of problems solved in the chapter or extensions of less demanding assignments from previous chapters.

This text can be used in several ways. It can be used in a two course sequence in which chapters 1–15 are covered in sequence. The instructor may spend at least part of the first class period introducing the students to the C++ development environment available. The text assumes no particular system; the procedure for editing, compiling, copying files, and so forth may be different on different systems (PC, Macintosh, Windows, Unix, etc.). On a semester schedule, chapters 1–8 could be covered in the first semester; chapters 9–15 would naturally follow in the second semester. This leaves the discussion of pointers and dynamic memory to the second course. Chapter 15 (Lower-level Programming) is optional, depending on the nature of the course. The table below contains a course outline based on 16 week semesters.

This book can be used for a second programming course that emphasizes elementary data structures and introduces abstract data types and object-oriented programming. If the students have had exposure to C++ or C, the first seven chapters may be reviewed in the first class period or two. Chapters 8–14 should be covered in detail. Chapter 15 (Lower-level Programming) is optional, depending on the nature of the class. If the students have no prior knowledge of C++ or C but have experience with another structured language (like Pascal or Modula-2), then the first seven chapters should be covered quickly but completely, emphasizing the syntactical deviations that might confuse experienced programmers unfamiliar with C++ or C. (The students should already know, for example, what loops are, how they are entered and exited, and how conditions for termination are checked; the task is translating that knowledge into the C++ form.)

Semester 1

Week	Chapter	Topic
1	1	Introduction
2	2	Variables, I/O
3	2,3	Expressions, data types
4	3	Binary representations
5	4	Relational ops, `if`
6	4	More `if`s, `switch`
7	5	`while`, `do . . .while`
8	5	`for`, loop termination
9	6	Functions
10	6	Storage classes
11	7	Arrays
12	7	Strings
13	7	Multidim. arrays
14	8	Structs
15	8	Classes, OOP
16	8	OOP
17	—	Final Examination

Semester 2

Week	Chapter	Topic
1	8	Review, OOP
2	9	Pointers
3	9	Dynamic memory
4	10	Self-referential structs
5	10	Linked lists
6	10	Recursion
7	11	Class operators
8	11	Copy semantics
9	11	Templates
10	12	Stacks, queues
11	13	Trees, BSTs
12	13	N-ary trees
13	14	Inheritance
14	14	Polymorphism
15	14,15	Class design
16	15	Lower-level prog.
17	—	Final Examination

The C++ code presented conforms to the American National Standards Institute (ANSI) base document as described in *The Annotated C++ Reference Manual* (commonly referred to as the *ARM*) by Margaret Ellis and Bjarne Stroustrup. Presently, a joint ANSI and International Standards Organization (ISO) committee is working on a C++ language standard. The *ARM* is the document that is currently identified as the "standard." The code herein conforms to AT&T release 3.0 of C++. (This corresponds to a level of C++ functionality, not to any particular compiler vendor's C++ version number). Templates are used heavily, beginning in chapter 11. Exception handling (AT&T release 4.0) is not presented since this feature was not widely available on many popular compilers at the time of this writing. If a site's development system supports exception handling, this topic could be introduced at any time after chapter 6, but it would naturally fit into the topics discussed in chapters 9 or 11.

Even though a great deal of time is spent examining the syntax and usage of C++, this book is not meant to be a comprehensive reference for the language. Its primary objective is teaching programming, and C++ is merely the medium. There are other books (the *ARM*, among others listed in the bibliography) that should be consulted when the limits of the language are to be explored. Unlike its predecessor C, C++ is a complex language. Indeed, C's simplicity was a virtue that even its critics had to concede. The syntactical and semantic structures of C++ are much more elaborate. This added complexity is mostly due to the advanced object-oriented programming features that were added, but some of it results from the additional baggage that C++ compilers must bear to insure compatibility with older C code. The result is that it is much more difficult for an individual to comprehend the full scope of the language. In this book, the most practical aspects of, for example, inheritance are covered in chapter 14. Public inheritance is the most popular and is presented with several examples. One example of private inheritance is also provided, but there are many other options available through combinations of public, protected, or private inheritance in conjunction with the public, protected, and private visibility specifications of members of the base class(es). Similarly, pointers to members, overloading `new` and `delete`, function pointers, and other tidbits that can be quite useful in certain situations are omitted to keep the text to a reasonable size.

This book would not exist in its present form without the assistance and understanding of many individuals. All of the students in my *Fundamentals of Programming I and II* classes at Southern College from the Fall of 1987 to present provided inspiration and ideas for this text through all of their questions, compiler errors, program design woes, innovative and insightful observations, and brilliant programming achievements. The reviewers of the manuscript provided welcome comments and criticisms that allowed me to make some sections clearer and fix some technical errors. They also provided guidance that prompted me to rearrange some of the chapters to produce a better organized text. Paul Ducham, Linda Meehan Avenarius, and Carrie Langas, editors at B&E Tech, provided much needed direction. Paul had enough faith in the original manuscript to start the publishing process. Linda oversaw the review process and helped me shore up its weaknesses to produce a viable book.

I wish to thank the following reviewers: Barbara Boucher Owens, St. Edward's University; Suzanne Sever, Wayne State College; Edward S. Miller, Lewis-Clark State College; Robert A. McDonald, East Stroudsburg University; William J. Moon, Palm Beach Community College; and Dr. Wm. C. Muellner, Elmhurst College.

Linden deCarmo, of IBM Corporation, contributed the PowerPC assembly code in chapter 1.

Janet, my lovely wife, served as my immediate proofreader, saving me from countless opportunities for grammatical embarrassment. She, as well as my daughters Jessica and Rachel, endured the whole process and provided much needed intangible support.

RLH

CONTENTS

CHAPTER 1

INTRODUCTION TO COMPUTER SYSTEMS AND THE ROLE OF C++ IN SOFTWARE DEVELOPMENT

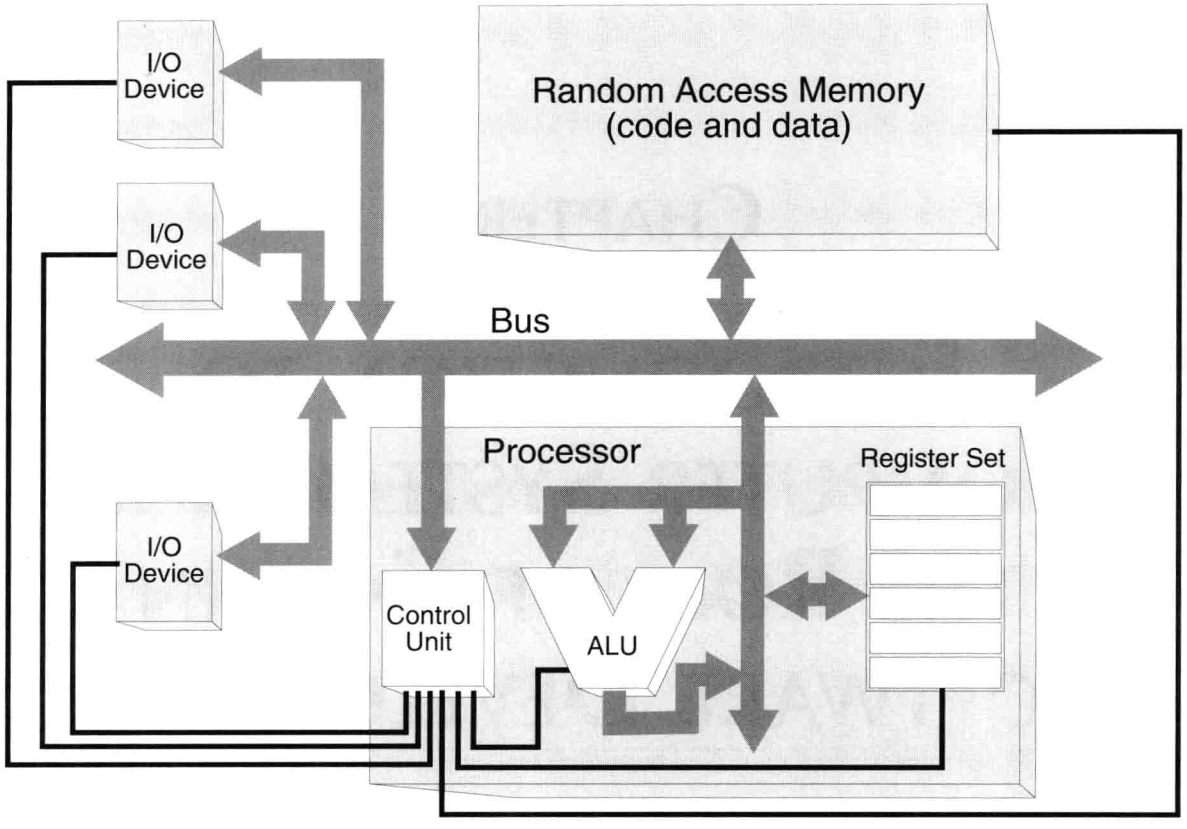
1.1 INTRODUCTION

A computer is a complex system. Like other complex systems, such as the human body or the U.S. government, it is composed of smaller systems and components. Computer systems are made up of two major systems—*hardware* and *software*. Each component is useless without the other. This book is concerned primarily with a particular kind of software development—*applications programming*. It is, however, impractical to discuss software development without knowledge of the hardware involved. In fact, since we will be writing programs and running these programs on actual machines, some knowledge of the hardware system is essential. The next section is an overview of computer hardware. It is far from comprehensive; in fact, it is merely enough to get us started. More information on the hardware will be provided later as the need arises.

1.2 COMPUTER HARDWARE

Hardware comprises the parts of the computer that can actually be seen and touched. The integrated circuits (chips), circuit boards, cables, keyboards, monitors, disk drives, etc. are all part of the hardware system. At the heart of the hardware is the *processor* or *central processing unit* (CPU). (See figure 1.1.)

The processor, or *microprocessor* in a microcomputer, is a chip (integrated circuit) that is the "brain" of the computer. The processor controls most of the other hardware. When a microcomputer is described as a 486 or Pentium machine, the name comes from the particular Intel



A Simplified View of Hardware
Figure 1.1

microprocessor around which the system was built. The Motorola 680x0 and the PowerPC chip are other examples of microprocessors used within today's microcomputers. The processor is responsible for moving data around from one part of the system to another, performing calculations on the data, and otherwise comparing and modifying the data. (Consider data here to mean some "chunks" of information.) The processor contains the *arithmetic-logic unit* (ALU) that performs calculations and comparisons and the *control unit* (CU) that sends signals to other parts of the hardware system controlling their function. The processor also contains a small number of *registers* that are used to store data temporarily for calculations. The software that we create will be translated into instructions for the processor. The programs we write control the computer by controlling the processor.

The processor is connected to the other primary hardware components through a link known as the *bus*. At the simplest level, the bus is merely a bunch of parallel wires running between the processor, memory, input devices, and output devices. Data is passed from an input device to the processor through the bus. You can think of the bus as an interstate highway system. Devices are connected to the highway by on and off ramps or "exits." Data can get from one component to another by getting on and off at the correct exits.

Memory, sometimes called *main memory* or *RAM (Random Access Memory)*, is where data are stored. The programs that we write (translated into processor instructions) are also stored in memory. Memory is simply a large storage area for data and machine instructions. A computer does all of its magic by performing some ridiculously simple steps over and over. The processor fetches an instruction from memory. Based on the particular instruction, it is likely to grab a piece of data from memory, do something to that datum element, and place it back somewhere (possibly the same

place) in memory. This is called the "fetch-execute cycle" because the processor fetches an instruction from memory and then executes that instruction. The instructions that the processor fetches constitute the computer's *software.*

Input and output (I/O) devices are responsible for putting data into memory or retrieving data from memory. The keyboard is one example of an input device. When you use a wordprocessor or text editor, the letters you type not only appear on the screen, but also are stored in a particular place in the computer's memory. The screen is a type of output device. What is visible on the screen corresponds in some way to data stored in memory. Disk drives allow information to be both stored and retrieved from memory. The *diskette* and *fixed disk* (hard disk) are also known as secondary memory or secondary storage. Data can be stored on a diskette as in the computer's main memory; however, the data in main memory are lost when the power is removed to the computer. Data stored on diskette remain until erased or overwritten. Other I/O devices include mice, printers, trackballs, joysticks, and so forth.

1.3 Computer Software

As mentioned, software controls the processor, which in turn controls the rest of the computer system. Without hardware, obviously, you could not have a computer; however, without software, you would have a computer that could do nothing. The processor would be unable to communicate with memory or I/O devices.

Software can be categorized as *systems software* and *applications software.* An *operating system* (OS) is a sophisticated piece of systems software that oversees the whole computer system. On a personal computer, DOS (Disk Operating System) is an OS that is often used. Other choices include IBM's OS/2 and Microsoft's Windows NT. Unix and the Macintosh System 7 are other popular OSs. The OS directs the processor's communication with the other hardware components. It is the platform upon which the other type of software, applications software (programs), may be executed.

To better understand how systems software and the OS work, consider what happens when the computer is turned on. Recall that the hardware can do nothing without software; all the processor can do is "fetch and execute." Upon power-up, the processor is in a reset condition; it immediately looks to a particular place in memory for an instruction to execute. It is built to look in a part of memory known as *ROM (Read-Only Memory).* ROM is a small part of memory with two special characteristics—it cannot be modified (unlike the memory mentioned above, RAM, where data and programs are stored), and it is not erased when the power is off (otherwise the computer could not start itself up the next time). The ROM contains the instructions for the processor to check the other hardware (memory, I/O devices, etc.) and load the OS into memory from disk. Once the OS is in memory, the processor follows the instructions dictated by the OS. On a PC under DOS, this same start-up sequence occurs but may look different on the screens of computers from different manufacturers since different ROMs may be used (Phoenix, Award, AMI, and others all make ROM-BIOS memories for PC compatibles). A DOS program called `command.com` (among others) is loaded into memory. The job of `command.com` is to interpret commands typed in from the keyboard.

Applications software consists of the programs that take advantage of the hardware and OS facilities. Applications software include wordprocessors, spreadsheets, databases, games, drawing programs, and just about every program you can think of. The C++ programs that are examined in this text are examples of applications software. Applications programs are typically easier to write than systems programs. Usually, the programmer does not have to worry about the details of how to access the I/O devices or memory. Programs are written that correspond to some real world problem, such as balancing a checkbook or simulating a game of tennis. The computer is simply a medium for solving these problems.

An *algorithm* is a finite sequence of well-understood steps that are followed to solve a particular problem or produce a particular result. For example, in a crude sense, a cake recipe is an algorithm for the production of a cake. There is a starting point—step 1 in the recipe. Following the recipe consists of performing a series of individual, simple, sequential (do this, then that—order is important) instructions that culminate at the stopping point—the finished cake. The technical definition of an algorithm is a bit more refined, but the essential principles are illustrated in the recipe example. An algorithm is a finite set of operations that must be performed in a particular sequence. Each operation must be well defined (that is, unambiguous), and each operation also must be able to be performed in a finite length of time. An algorithm must have a definite stopping place.

An algorithm is used in algebra to find the equation of the line (in the useful slope-intercept form) that passes between two points (x_1,y_1) and (x_2,y_2).

1. Find the slope, *m,* of the line passing through the points

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

2. Use the slope and one of the points in the point-slope form for the equation of a line

$$y - y_1 = m(x - x_1)$$

3. Solve for y (add y_1 to both sides)

$$y = m(x - x_1) + y_1$$

4. Distribute m to find b (the slope intercept)

$$y = mx - mx_1 + y_1,\ b = y_1 - mx_1$$

A program is an algorithm that has been implemented in a particular programming language. Programs can be written using many different computer languages. *Assembly language* offers absolute control over the processor because its instructions are converted directly into *machine language*—the instructions that the processor executes. It is difficult to develop programs in assembly language; the programmer must take care of all the details; namely, where values are to be placed in memory, how those values are to be manipulated with only very rudimentary operations possible, and so forth. Initially, assembly language was the only language available. It is called a *low-level* language. To solve problems in assembly language, not only must the programmer completely understand the problem at hand, but he or she must also have a good understanding of the inner workings of the microprocessor.

In the late 1950s, *higher-level* languages began to appear. These languages (FORTRAN is the oldest still in wide use) allowed the programmer to write programs in a form closer to human language. Human languages (English, French, Spanish, Japanese, etc.) are inherently flexible and often ambiguous, but higher-level computer languages are quite rigid in their form and composition rules. Nonetheless, higher-level languages remove much of the drudgery associated with assembly language programming. Consider figure 1.2 comparing a section of C++ code to its equivalent assembly code for two different CPUs. (The word *code* used in the context of programming has both a noun and verb form. The noun refers to programs or sections of a program written in a particular programming language; thus, a C++ programmer writes in C++ code which is eventually translated into low-level machine language code. The act of programming is sometimes called *coding.*)

Both assembly language code sequences were adapted from the assembler output from C++ compilers designed specifically for each system. Whereas the C++ code is the same for both systems, the actual machine language code is very different. The 80x86 code is for a complex instruction set computer (CISC) processor. CISC processors contain many different machine language instructions in their set of usable instructions. The PowerPC is a reduced instruction set computer (RISC). RISC

```
_SUM_LIST:
      PUSH BP
      MOV  BP,SP
      PUSH SI
      PUSH DI
      XOR  SI,SI
      XOR  DI,DI
      JMP  SL2
SL1:
      MOV  AX,SI
      SHL  AX,1
      LES
BX,[BP+06]
      ADD  BX,AX
      MOV  AX,DI
      ADD
AX,ES:[BX]
      MOV  DI,AX
      MOV  AX,SI
      INC  AX
      MOV  SI,AX
SL2:
      CMP
SI,[BP+0A]
      JL   SL1
      MOV  AX,DI
      POP  DI
      POP  SI
      POP  BP
      RETF
```

Intel 80×86 Assembly Language

```
..LL33:
      .globl  sum
sum:
      ori     %r10,%r3,0
      addi    %r9,%r0,0
      addi    %r3,%r0,0
      cmp     0,%r9,%r4
      bge     ..LL34
..LL35:
      mulli   %r12,%r9,4
      lwzx
%r12,%r12,%r10
      addc    %r3,%r12,%r3
      addic   %r9,%r9,1
      cmp     0,%r9,%r4
      blt     ..LL35
..LL34:
      bclr    20,0
```

PowerPC Assembly Language

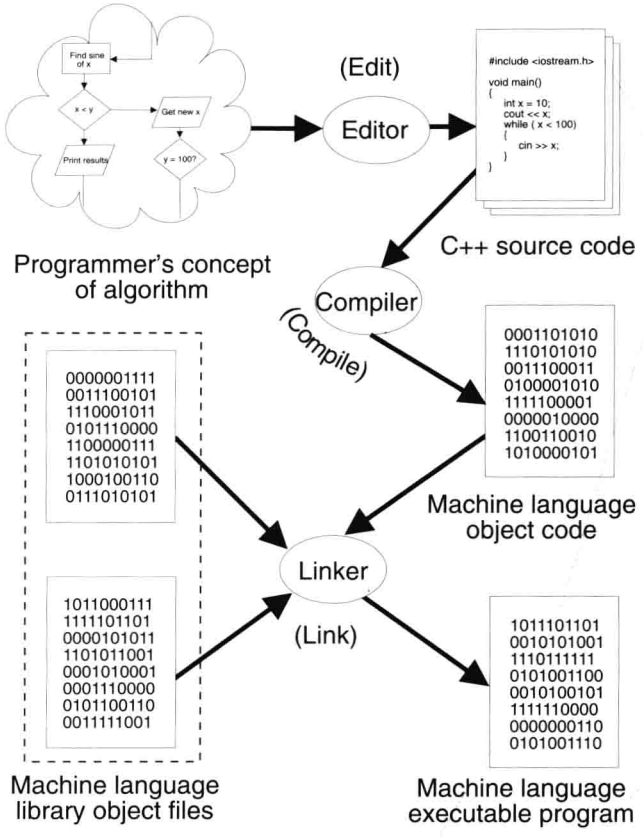
```
int sum_list(int list[], int
size)
{
     int index = 0, sum = 0;

     while ( index < size ) {
          sum = sum +
list[index];
          index = index + 1;
     }
     return sum;
}
```

C++ Language

Assembly Language vs. C++
Figure 1.2

machines employ a small number of highly optimized instructions to attempt to achieve greater processing speed. Higher-level languages, which are easier to use due to their proximity to natural languages and mathematical expression, provide another, perhaps more important, advantage. A program written in C++ can easily be adapted to execute on any computer system (PC, Macintosh, DEC VAX minicomputer, IBM mainframe, etc.); however, an assembly language program written for one kind of machine would need to be completely rewritten to work on a different kind of machine. i486 assembly language (PC) is much different from either 68040 (Macintosh) or PowerPC (Macintosh) assembly languages. (This fact should not be minimized—some programs require years of development by scores of programmers.) Higher-level languages include *C++*, *C*, *BASIC*, *Pascal*, *FORTRAN*, *COBOL*, *Ada*, *Modula-2*, *Lisp*, and *Prolog*.

Since the processor only understands its own machine language and cannot interpret any higher-level language, all higher-level languages must be translated into the machine language of the particular processor. A special program called a *compiler* converts the higher-level program text



Development Sequence
Figure 1.3

into machine language. Figure 1.3 illustrates the development process. The C++ language source code is typed in with an editor (a program that works like a simplified wordprocessor) and stored in a file. Different systems use different naming conventions. Throughout this text we will assume that the C++ source files are named with a `.cpp` extension. Some development environments prefer `.C`, `.c`, `.CXX`, `.cxx`, or perhaps some other filename extension. The compiled machine language code file, here given an `.obj` extension, may instead use `.o`. The executable program file is here assumed to have an `.exe` extension, but other systems may have other preferences (or none at all). Some systems, like Unix, distinguish between upper- and lowercase (capitalized and uncapitalized) characters in file names. Some systems, like DOS, treat both the same. All references to filenames in this book will be of the form `filename.ext` in all lowercase (uncapitalized) small font.

Exercises

1. Define the following terms: *ALU, compiler, hardware, software, bus, input, output, algorithm, higher-level language, RAM, ROM, code,* and *operating system.*

2. List the steps in the translation of a C++ program into a computer's native machine language.

3. What are the major components in any computer system?

4. How is a higher-level computer programming language different from a lower-level computer programming language?

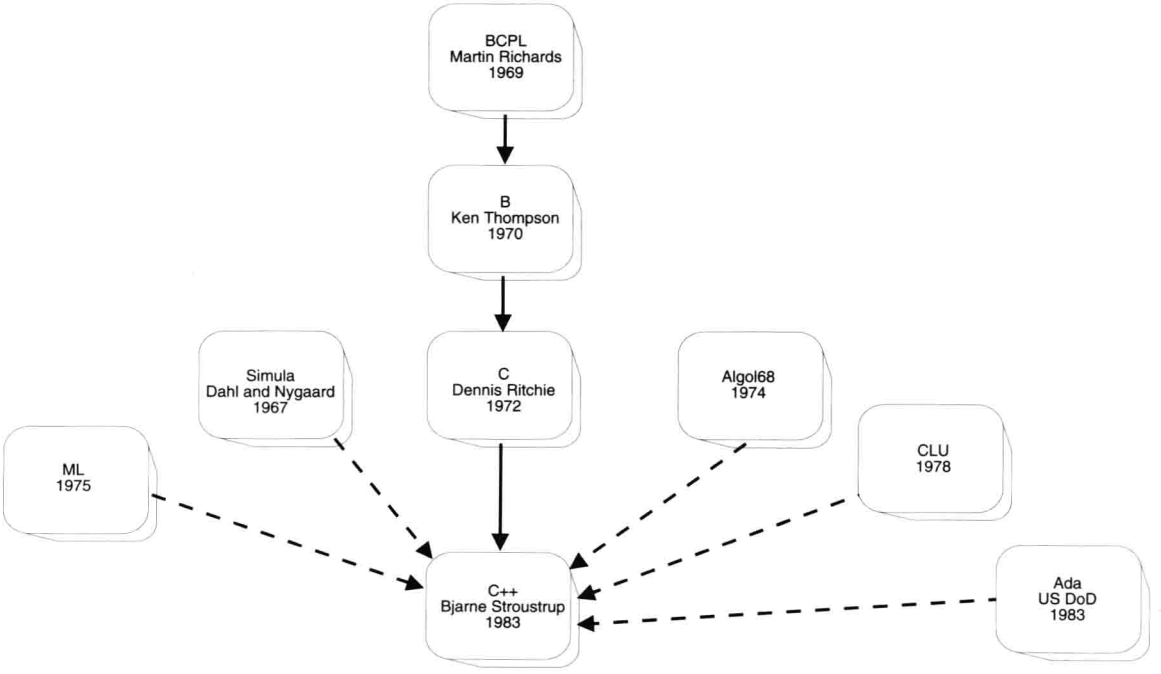
5. List some examples of *applications software.* List some examples of *systems software.*

6. What commands must you follow on your system to invoke the (1) editor, (2) compiler, and (3) the linker?

7. What filename extension for C++ source programs is preferred by your development environment?

1.4 The Story of C++

C++ is an extension of C, a higher-level computer language developed by Dennis Ritchie in the early 1970s at AT&T Bell Laboratories. C++'s genealogy is shown in figure 1.4. C was first designed to run on a PDP-11 minicomputer. The C language made possible the implementation of the Unix operating system as it is known today. More than 90 percent of Unix was written in C (the remainder was written in assembly language). By the late 1970s, C began to gain widespread popularity and support and became available for commercial use outside of AT&T Bell Labs.



C++'s Family Tree
Figure 1.4