Chapter 1
Introduction to Computers

Welcome to computer science! You are about to explore a wonderful and exciting world—a world that offers many challenging and exciting careers.

In this chapter, we introduce you to the concepts of computer science, especially as they pertain to computer programming. You will study the concept of a computer system and how it relates to computer hardware and software. We will also present a short history of computer programming languages so that you understand how they have evolved and how the C language fits into the picture.

We will then describe how to write a program, first with a review of the tools and steps involved, and then with a review of a system development methodology.

Objectives

❑ To review basic computer systems concepts

❑ To be able to understand the different computing environments and their components

❑ To review the history of computer languages

❑ To be able to list and describe the classifications of computer languages

❑ To understand the steps in the development of a computer program

❑ To review the system development life cycle

1.1 Computer Systems

Today computer systems are found everywhere. Computers have become almost as common as televisions. But what is a computer? A computer is a system made of two major components: hardware and software. The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do its job. Figure 1-1 represents a **computer system**.

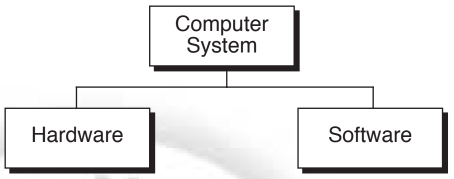

FIGURE 1-1 A Computer System

Computer Hardware

The **hardware** component of the computer system consists of five parts: input devices, central processing unit (CPU), primary storage, output devices, and auxiliary storage devices (Figure 1-2).

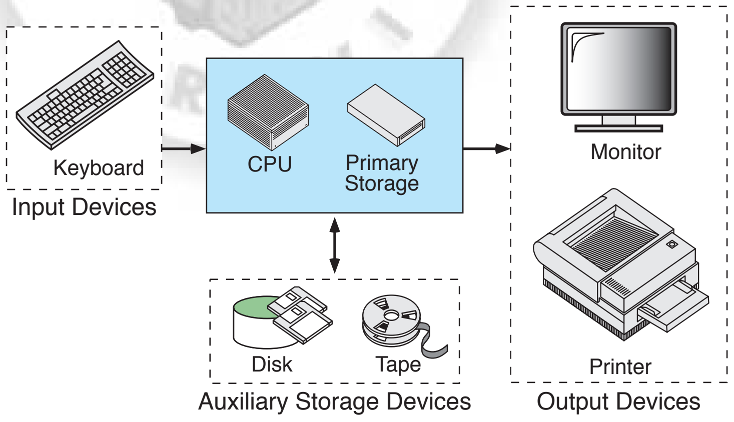

FIGURE 1-2 Basic Hardware Components

The **input device** is usually a keyboard where programs and data are entered into the computer. Examples of other input devices include a mouse, a pen or stylus, a touch screen, or an audio input unit.

The **central processing unit** (**CPU**) is responsible for executing instructions such as arithmetic calculations, comparisons among data, and movement of data inside the system. Today's computers may have one, two, or more CPUs. **Primary storage**, also known as **main memory**, is a place where the programs and data are stored temporarily during processing. The data in primary storage are erased when we turn off a personal computer or when we log off from a time-sharing computer.

The **output device** is usually a monitor or a printer to show output. If the output is shown on the monitor, we say we have a **soft copy**. If it is printed on the printer, we say we have a **hard copy**.

Auxiliary storage, also known as **secondary storage**, is used for both input and output. It is the place where the programs and data are stored permanently. When we turn off the computer, our programs and data remain in the secondary storage, ready for the next time we need them.

Computer Software

Computer **software** is divided into two broad categories: system software and application software. This is true regardless of the hardware system architecture. System software manages the computer resources. It provides the interface between the hardware and the users but does nothing to directly serve the users' needs. Application software, on the other hand, is directly responsible for helping users solve their problems. Figure 1-3 shows this breakdown of computer software.

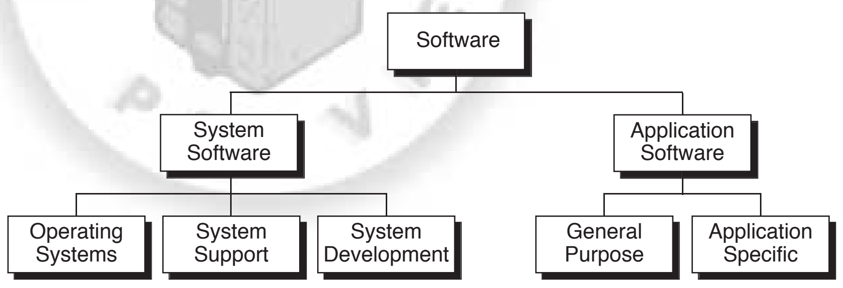

FIGURE 1-3 Types of Software

System Software

System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consist of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category, **system development software**, includes the language translators that convert programs into machine language for execution, debugging tools to ensure that the programs are error-free, and computer-assisted software engineering (CASE) systems that are beyond the scope of this book.

Application Software

Application software is broken into two classes: general-purpose software and application-specific software. **General-purpose software** is purchased from a software developer and can be used for more than one application. Examples of general-purpose software include word processors, database management systems, and computer-aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

Application-specific software can be used only for its intended purpose. A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed; they cannot be used for other generalized tasks.

The relationship between system and application software is seen in Figure 1-4. In this figure, each circle represents an interface point. The inner core is the hardware. The user is represented by the outer layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hardware. Note the opening at the bottom of the figure. This is the path followed by the user who interacts directly with the operating system when necessary.

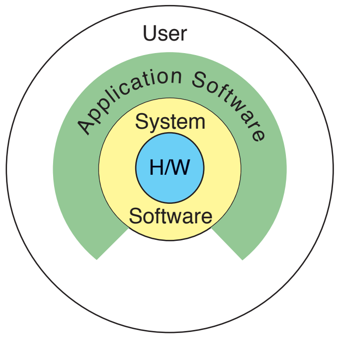

FIGURE 1-4 Relationship Between System and Application Software

If users cannot buy software that supports their needs, then a custom-developed application must be built. In today's computing environment, one of the tools used to develop software is the C language that we will be studying in this text.

1.2 Computing Environments

In the early days of computers, there was only one environment: the mainframe computer hidden in a central computing department. With the advent of minicomputers and personal computers, the environment changed, resulting in computers on virtually every desktop. In this section we describe several different environments.

Personal Computing Environment

In 1971, Marcian E. Hoff, working for Intel, combined the basic elements of the central processing unit into the microprocessor. This first computer on a chip was the Intel 4004 and was the grandparent many times removed of Intel's current system.

If we are using a personal computer, all of the computer hardware components are tied together in our **personal computer** (or **PC**[1] for short). In this situation, we have the whole computer for ourself; we can do whatever we want. A typical personal computer is shown in Figure 1-5.

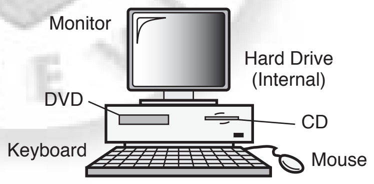

FIGURE 1-5 Personal Computing Environment

Time-Sharing Environment

Employees in large companies often work in what is known as a **time-sharing environment**. In the time-sharing environment, many users are connected to one or more computers. These computers may be minicomputers or central mainframes. The terminals they use are often nonprogrammable, although today we see more and more microcomputers being used to simulate terminals. Also, in the time-sharing environment, the output devices (such as printers) and auxiliary storage devices (such as disks) are shared by all of the

1. PC is now generally accepted to mean any hardware using one of Microsoft's Windows operating systems, as opposed to Apple's Macintosh. We use it in the original, generic sense meaning any personal computer.

users. A typical college lab in which a minicomputer is shared by many students is shown in Figure 1-6.

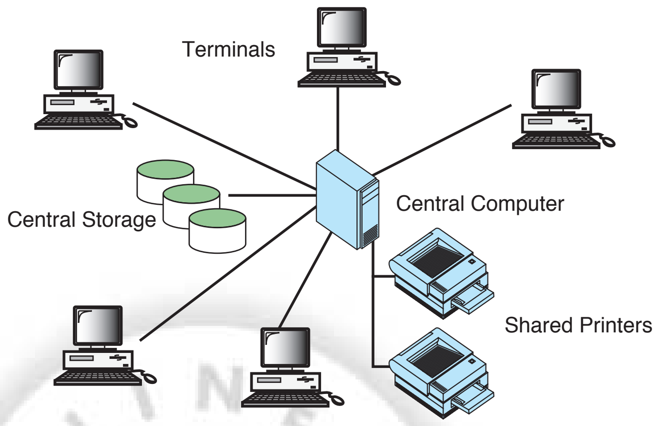

FIGURE 1-6 Time-sharing Environment

In the time-sharing environment, all computing must be done by the central computer. In other words, the central computer has many duties: It must control the shared resources; it must manage the shared data and printing; and it must do the computing. All of this work tends to keep the computer busy. In fact, it is sometimes so busy that the user becomes frustrated by the computer's slow responses.

Client/Server Environment

A **client/server** computing environment splits the computing function between a central computer and users' computers. The users are given personal computers or workstations so that some of the computation responsibility can be moved from the central computer and assigned to the workstations. In the client/server environment, the users' microcomputers or workstations are called the **client**. The central computer, which may be a powerful microcomputer, minicomputer, or central mainframe system, is known as the **server**. Because the work is now shared between the users' computers and the central computer, response time and monitor display are faster and the users are more productive. Figure 1-7 shows a typical client/server environment.

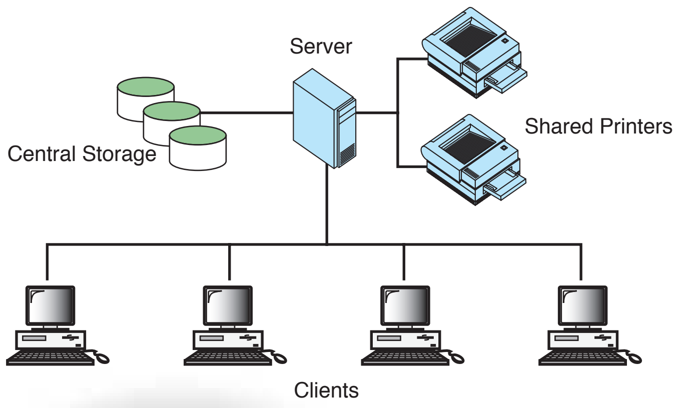

FIGURE 1-7 The Client/Server Environment

Distributed Computing

A **distributed computing** environment provides a seamless integration of computing functions between different servers and clients. The Internet provides connectivity to different servers throughout the world. For example, eBay uses several computers to provide its auction service. This environment provides a reliable, scalable, and highly available network. Figure 1-8 shows a distributed computing system.

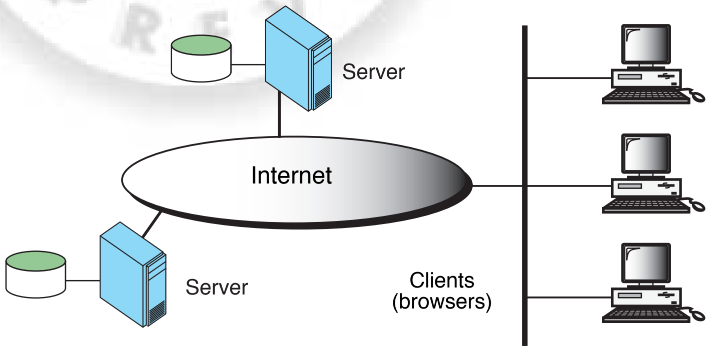

FIGURE 1-8 Distributed Computing

1.3 Computer Languages

To write a program for a computer, we must use a **computer language**. Over the years computer languages have evolved from machine languages to natural languages. A summary of computer languages is seen in Figure 1-9.

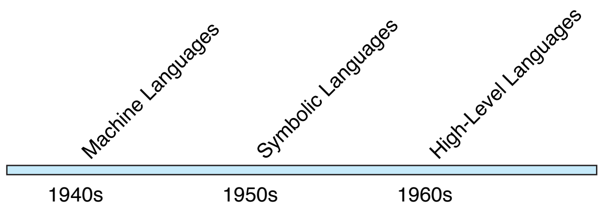

FIGURE 1-9 Computer Language Evolution

Machine Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own **machine language**, which is made of streams of 0's and 1's. Program 1-1 shows an example of a machine language. This program multiplies two numbers and prints the results.

PROGRAM 1-1 The Multiplication Program in Machine Language

```
         00000000 00000100 0000000000000000
01011110 00001100 11000010 0000000000000010
         11101111 00010110 0000000000000101
         11101111 10011110 0000000000001011
11111000 10101101 11011111 0000000000010010
         01100010 11011111 0000000000010101
11101111 00000010 11111011 0000000000010111
11110100 10101101 11011111 0000000000011110
00000011 10100010 11011111 0000000000100001
11101111 00000010 11111011 0000000000100100
01111110 11110100 10101101
11111000 10101110 11000101 0000000000101011
00000110 10100010 11111011 0000000000110001
11101111 00000010 11111011 0000000000110100
         01010000 11010100 0000000000111011
                  00000100 0000000000111101
```

The instructions in machine language must be in streams of 0's and 1's because the internal circuits of a computer are made of switches, transistors, and other electronic devices that can be in one of two states: off or on. The off state is represented by 0; the on state is represented by 1.

The only language understood by computer hardware is machine language.

Symbolic Languages

It became obvious that few programs would be written if programmers continued to work in machine language. In the early 1950s, Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols, or mnemonics, to represent the various machine language instructions. Because they used symbols, these languages were known as **symbolic languages**. Program 1-2 shows the multiplication program in a symbolic language.[2]

PROGRAM 1-2 The Multiplication Program in Symbolic Language

```
        entry   main,^m<r2>
        subl2   #12,sp
        jsb     C$MAIN_ARGS
        movab   $CHAR_STRING_CON

        pushal  -8(fp)
        pushal  (r2)
        calls   #2,SCANF
        pushal  -12(fp)
        pushal  3(r2)
        calls   #2,SCANF
        mull3   -8(fp),-12(fp),-
        pusha   6(r2)
        calls   #2,PRINTF
        clrl    r0
        ret
```

Because a computer does not understand symbolic language, it must be translated to the machine language. A special program called an assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language, they soon became known as **assembly languages**. This name is still used today for symbolic languages that closely represent the machine language of their computer.

Symbolic language uses symbols, or mnemonics, to represent the various machine language instructions.

2. The symbolic language format is label, operators, and operands. There are no identifiers in this example, so the left column is empty.

High-Level Languages

Although symbolic languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware that they were using. Working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of **high-level languages**.

High-level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer. High-level languages are designed to relieve the programmer from the details of the assembly language. High-level languages share one thing with symbolic languages, however: They must be converted to machine language. The process of converting them is known as **compilation**.

The first widely used high-level language, FORTRAN,[3] was created by John Backus and an IBM team in 1957; it is still widely used today in scientific and engineering applications. Following soon after FORTRAN was COBOL.[4] Admiral Hopper was again a key figure in the development of the COBOL business language.

C is a high-level language used for system software and new application code. Program 1-3 shows the multiplication program as it would appear in the C language.

PROGRAM 1-3 The Multiplication Program in C

```
/* This program reads two integers from the keyboard
   and prints their product.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Definitions
   int number1;
   int number2;
   int result;

// Statements
```

continued

3. FORTRAN is an acronym for FORmula TRANslation.
4. COBOL is an acronym for COmmon Business-Oriented Language.

PROGRAM 1-3 The Multiplication Program in C *(continued)*

```
      scanf  ("%d", &number1);
      scanf  ("%d", &number2);
      result = number1 * number2;
      printf ("%d", result);
      return 0;
   }  // main
```

1.4 Creating and Running Programs

As we learned in the previous section, computer hardware understands a program only if it is coded in its machine language. In this section, we explain the procedure for turning a program written in C into machine language. The process is presented in a straightforward, linear fashion, but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code.

It is the job of the programmer to write and test the program. There are four steps in this process: (1) writing and editing the program, (2) compiling the program, (3) linking the program with the required library modules, and (4) executing the program. These steps are seen in Figure 1-10.

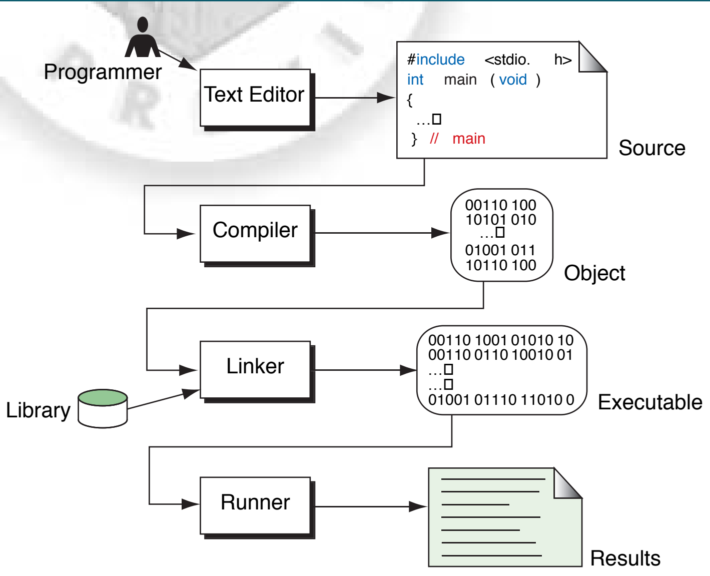

FIGURE 1-10 Building a C Program

Writing and Editing Programs

The software used to write programs is known as a **text editor**. A text editor helps us enter, change, and store character data. Depending on the editor on our system, we could use it to write letters, create reports, or write programs. The main difference between text processing and program writing is that programs are written using lines of code, while most text processing is done with characters and lines.

Our text editor could be a generalized word processor, but it is more often a special editor included with the compiler. Some of the features you should look for in your editor are search commands to locate and replace statements, copy and paste commands to copy or move statements from one part of a program to another, and formatting commands that allow you to set tabs to align statements.

After we complete a program, we save our file to disk. This file will be input to the compiler; it is known as a **source file**.

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the **compiler**. The C compiler is actually two separate programs: the **preprocessor** and the **translator**.

The preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as **preprocessor commands**. These commands tell the preprocessor to look for special code libraries, make substitutions in the code, and in other ways prepare the code for translation into machine language. The result of preprocessing is called the **translation unit**.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting **object module** to a file that can then be combined with other precompiled units to form the final program. An object module is the code in machine language. Even though the output of the compiler is machine language code, it is not yet ready to run; that is, it is not yet executable because it does not have the required C and other functions included.

Linking Programs

As we will see later, a C program is made up of many functions. We write some of these functions, and they are a part of our source program. However, there are other functions, such as input/output processes and mathematical library functions, that exist elsewhere and must be attached to our program. The **linker** assembles all of these functions, ours and the system's, into our final executable program.

Executing Programs

Once our program has been linked, it is ready for execution. To execute a program, we use an operating system command, such as `run`, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the **loader**. It locates the executable program and reads it into memory. When everything is loaded, the program takes control and it begins execution. In today's integrated development environments, these steps are combined under one mouse click or pull-down window.

In a typical program execution, the program reads data for processing, either from the user or from a file. After the program processes the data, it prepares the output. Data output can be to the user's monitor or to a file. When the program has finished its job, it tells the operating system, which then removes the program from memory.

1.5 System Development

We've now seen the steps that are necessary to build a program. In this section, we discuss *how* we go about developing a program. This critical process determines the overall quality and success of our program. If we carefully design each program using good structured development techniques, our programs will be efficient, error-free,[5] and easy to maintain.

System Development Life Cycle

Today's large-scale, modern programming projects are built using a series of interrelated phases commonly referred to as the **system development life cycle**. Although the exact number and names of the phases differ depending on the environment, there is general agreement as to the steps that must be followed. Whatever the methodology, however, today's software engineering concepts require a rigorous and systematic approach to software development.[6]

One very popular development life cycle is the **waterfall model**. Depending on the company and the type of software being developed, this model consists of between five and seven phases. Figure 1-11 is one possible variation on the model.

5. Many computer scientists believe that all programs contain at least one *bug*—an undetected error—that is just waiting to cause problems, given the right set of circumstances. Programs have run for years without problems only to fail when an unusual situation occurs. Perhaps the most famous bug was the one known as Y2K because it caused programs to fail on January 1, 2000.
6. For a discussion of various models, see *Software Engineering: A Practitioner's Approach,* 5th. ed. by Roger S. Pressman, (New York, NY: McGraw-Hill, 2001).

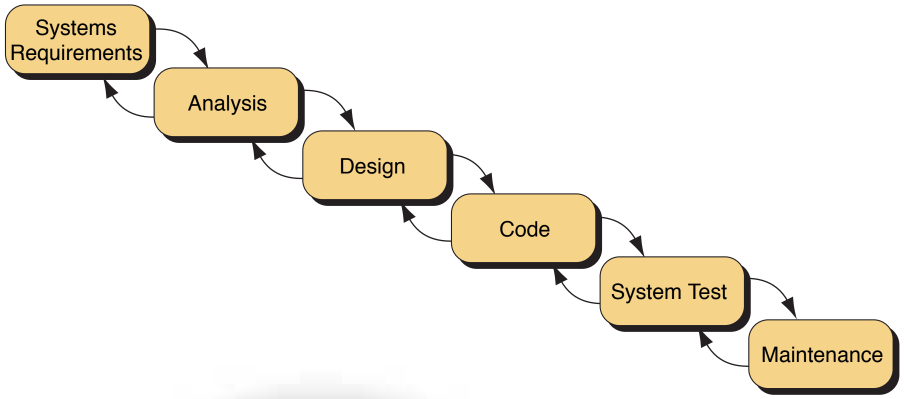

FIGURE 1-11 Waterfall Model

The waterfall model starts with *systems requirements*. In this phase, the systems analyst defines requirements that specify what the proposed system is to accomplish. The requirements are usually stated in terms that the user understands. The *analysis* phase looks at different alternatives from a system's point of view, while the *design* phase determines how the system will be built. In the design phase, the functions of the individual programs that will make up the system are determined and the design of the files and/or the databases is completed. Finally, in the fourth phase, *code*, we write the programs. This is the phase that is explained in this book. After the programs have been written and tested to the programmer's satisfaction, the project proceeds to *system test*. All of the programs are tested together to make sure the system works as a whole. The final phase, *maintenance*, keeps the system working once it has been put into production.

Although the implication of the waterfall approach is that the phases flow in a continuous stream from the first to the last, this is not really the case. Note the iteration as indicated by the backward-flowing arrows in Figure 1-11. As each phase is developed, errors and omissions will often be found in the previous work. When this happens, it is necessary to go back to the previous phase to rework it for consistency and to analyze the impact caused by the changes. Hopefully, this is a short rework. We are aware of at least three major projects, however, that were in the code and test phases when it was determined that they could not be implemented and had to be canceled. When this happens, millions of dollars and years of development are lost.

Program Development

Program Development is a multistep process that requires that we understand the problem, develop a solution, write the program, and then test it.

When we are given the assignment to develop a program, we will be given a program requirements statement and the design of any program interfaces. We should also receive an overview of the complete project so that we will understand how our part fits in to the whole. Our job is to determine how to take the inputs we are given and convert them into the outputs that have been specified. This is known as *program design*. To give us an idea of how this process works, let's look at a simple problem: Calculate the square footage of a house. How do we go about doing this?

Understand the Problem

The first step in solving any problem is to understand it. We begin by reading the requirements statement carefully. When we think that we fully understand it, we review our understanding with the user and the systems analyst. Often this involves asking questions to confirm our understanding.

For example, after reading our simple requirements statement, we should ask several clarifying questions.

- ❏ What is the definition of square footage?
- ❏ How is the square footage going to be used?
  - ■ for insurance purposes?
  - ■ to paint the inside or outside of the house?
  - ■ to carpet the whole house?
- ❏ Is the garage included?
- ❏ Are closets and hallways included?

Each of these potential uses requires a different measure. If we don't clarify the exact purpose—that is, if we make assumptions about how the output is going to be used—we could supply the wrong answer.

As this example shows, even the simplest problem statements need clarification. Imagine how many questions must be asked for a programmer to write a program that will contain hundreds or thousands of detailed statements.

Develop the Solution

Once we fully understand the problem and have clarified any questions we may have, we need to develop our solution. Three tools will help us in this task: (1) structure charts, (2) pseudocode, and (3) flowcharts. Generally, we will use only two of them—a structure chart and either pseudocode or a flowchart.

The structure chart is used to design the whole program. Pseudocode and flowcharts, on the other hand, are used to design the individual parts of the program. These parts are known as modules in pseudocode or functions in the C language.

Structure Chart

A **structure chart**, also known as a hierarchy chart, shows the functional flow through our program. Large programs are complex structures consisting of many interrelated parts; thus, they must be carefully laid out. This task is similar to that of a design engineer who is responsible for the operational design of any complex item. The major difference between the design built by a programmer and the design built by an engineer is that the programmer's product is software that exists only inside the computer, whereas the engineer's product is something that can be seen and touched.

The structure chart shows how we are going to break our program into logical steps; each step will be a separate module. The structure chart shows the interaction between all the parts (modules) of our program.

It is important to realize that the design, as represented by the structure chart, is done before we write our program. In this respect, it is like the architect's blueprint. We would not start to build a house without a detailed set of plans. Yet one of the most common errors of both experienced and new programmers alike is to start coding a program before the design is complete and fully documented.

This rush to start is due in part to programmers' thinking they fully understand the problem and also due to their excitement about solving a new problem. In the first case, what they find is that they did not fully understand the problem. By taking the time to design the program, they will raise more questions that must be answered and therefore will gain a better understanding of the problem.

An old programming proverb: Resist the temptation to code.

The second reason programmers code before completing the design is just human nature. Programming is a tremendously exciting task. To see our design begin to take shape, to see our program creation working for the first time, brings a form of personal satisfaction that is a natural high.

In the business world, when we complete a structure chart design, we will convene a review panel for a *structured walk-through* of our program. Such a panel usually consists of a representative from the user community, one or two peer programmers, the systems analyst, and possibly a representative from the testing organization. In the review, we will walk our review team through our structure chart to show how we plan to solve the objectives of our program. The team will then offer constructive suggestions as to how to improve our design.

The primary intent of the review is to increase quality and save time. The earlier a mistake is detected, the easier it is to fix it. If we can eliminate only one or two problems with the structured walk-through, the time will be well spent. Naturally, in a programming class, you will not be able to convene a full panel and conduct a formal walk-through. What you can do, however, is review your design with some of your classmates and with your professor.

Looking again at our problem to calculate the square footage of a house, let's assume the following answers to the questions raised in the previous section.

1. The purpose of calculating the square footage is to install new floor covering.
2. Only the living space will be carpeted. The garage and closets will not be considered.
3. The kitchen and bathrooms will be covered with linoleum; the rest of the house is to be carpeted.

With this understanding, we decide to write separate modules for the kitchen, bathroom(s), bedrooms, family room, and living room. We use separate modules because the various rooms may require a different quality of linoleum and carpeting. The structure chart for our design is shown in Figure 1-12.

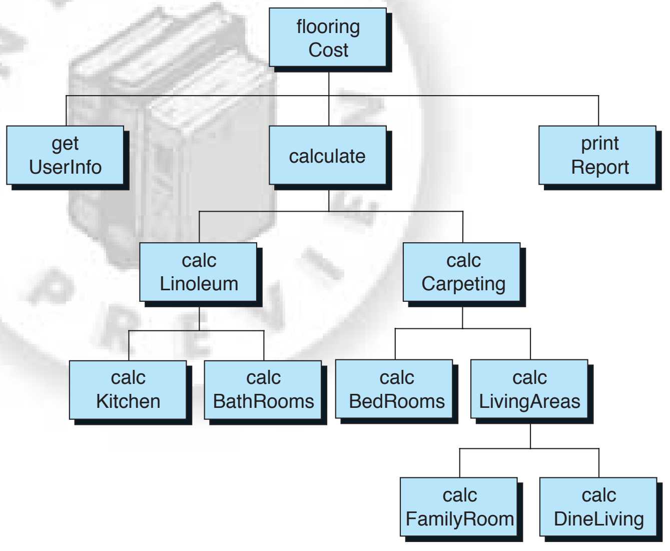

FIGURE 1-12 Structure Chart for Calculating Square Footage

Whether you use a flowchart or pseudocode to complete the design of your program will depend on your experience, the difficulty of the process you are designing, and the culture and standards of the organization where you are working. We believe that new programmers should first learn program design by flowcharting because a flowchart is a visual tool that is easier to create than pseudocode. On the other hand, pseudocode is more common among professional programmers.

Pseudocode

Pseudocode is part English, part program logic. Its purpose is to describe, in precise algorithmic detail, what the program being designed is to do. This requires defining the steps to accomplish the task in sufficient detail so that they can be converted into a computer program. Pseudocode excels at this type of precise logic. The pseudocode for determining the linoleum for the bathroom is shown in Algorithm 1-1.

Pseudocode

English-like statements that follow a loosely defined syntax and are used to convey the design of an algorithm.

Most of the statements in the pseudocode are easy to understand. A prompt is simply a displayed message telling the user what data are to be entered. The *while* is a loop that repeats the three statements that follow it and uses the number of bathrooms read in statement 2 to tell when to stop. Looping is a programming concept that allows us to repeat a block of code. We will study it in Chapter 6. In this case, it allows us to process the information for one or more bathrooms.

ALGORITHM 1-1 Pseudocode for Calculate Bathrooms

```
Algorithm Calculate BathRooms
 1 prompt user and read linoleum price
 2 prompt user and read number of bathrooms
 3 set total bath area and baths processed to zero
 4 while ( baths processed < number of bathrooms )
   1  prompt user and read bath length and width
   2  total bath area =
   3           total bath area + bath length * bath width
   4  add 1 to baths processed
 5 bath cost = total bath area * linoleum price
 6 return bath cost
end Algorithm Calculate BathRooms
```

Flowchart

A **flowchart** is a program design tool in which standard graphical symbols are used to represent the logical flow of data through a function. Appendix C contains complete instructions for creating flowcharts. If you are not familiar with flowcharts, we suggest you read Appendix C now.

The flowchart in Figure 1-13 shows the design for calculating the area and cost for the bathrooms. A few points merit comment here. This flowchart is basically the same as the pseudocode. We begin with prompts for the price of the linoleum and the number of bathrooms and read these two pieces of data. (As a general rule, flowcharts do not explicitly show standard concepts such as prompts.) The loop reads the dimensions for each bathroom. Finally, when we know the total area, we calculate the price and return to `calcLinoleum`.

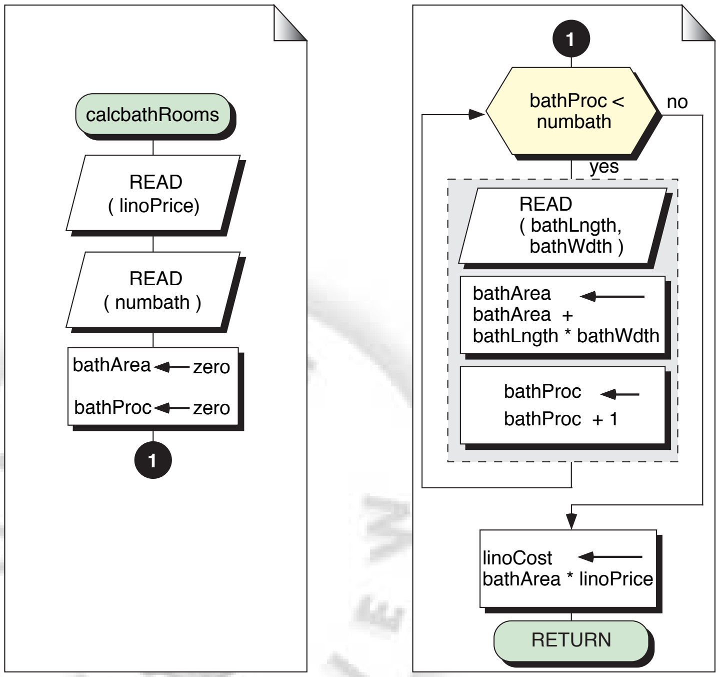

FIGURE 1-13 Flowchart for Calculate Bathrooms

Write the Program

Now it's time to write the program! But first, let's review the steps that we've used.

1. Understand the problem.
2. Develop a solution.
    a. Design the program—create the structure chart.
    b. Design the algorithms for the program using either flowcharting or pseudocode or both.
3. Write the program.

When we write a program, we start with the top box on the structure chart and work our way to the bottom. This is known as top–down implementation. You will find that it is a very easy and natural way to write programs, especially if you have done a solid job on your design.

For our first few programs, there will be only one module, representing the top box of the structure chart. The first programs are quite simple and do not require subdivision. When we get to Chapter 4, however, we will begin to write functions and our structure charts will get larger. At that time we will point out some more techniques for writing structured programs. In the meantime, concentrate on writing good pseudocode or flowcharts for the main part of our programs.

Test the Program

After we've written our program, we must test it. Program testing can be a very tedious and time-consuming part of program development. As the programmer, we are responsible for completely testing our program. In large development projects, there are often specialists known as test engineers who are responsible for testing the system as a whole—that is, for testing to make sure all the programs work together.

There are two types of testing: blackbox and whitebox. Blackbox testing is done by the system test engineer and the user. Whitebox testing is the responsibility of the programmer.

Blackbox Testing

Blackbox testing gets its name from the concept of testing the program without knowing what is inside it—without knowing how it works. In other words, the program is like a black box that we can't see into.

Blackbox test plans are developed by looking only at the requirements statement (this is only one reason why it is so important to have a good set of requirements). The test engineer uses these requirements and his or her knowledge of systems development and the user working environment to create a test plan that will then be used when the system is tested as a whole. We should ask to see this test plan before we write our program. The test engineer's plan will help us make sure we fully understand the requirements and also help us create our own test plan.

Whitebox Testing

Whereas blackbox testing assumes that the tester knows nothing about the program, **whitebox testing** assumes that the tester knows everything about the program. In this case, the program is like a glass house in which everything is visible.

Whitebox testing is our responsibility. As the programmer, we know exactly what is going on inside the program. We must make sure that every instruction and every possible situation has been tested. That is not a simple task!

Experience will help us design good test data, but one thing we can do from the start is to get in the habit of writing test plans. We should start the test plan when we are in the design stage. As we build your structure chart, ask ourself what situations, especially unusual situations, we need to test for and make a note of them immediately because we won't remember them an hour later.

When we are writing our flowcharts or pseudocode, we need to review them with an eye toward test cases and make additional notes of the cases we may need. Finally, while we are coding, we need to keep paper handy (or a test document open in our word processor) to make notes about test cases we need.

Except for the most simple program, one set of test data will not completely validate a program.

When it is time to construct our test cases, we review our notes and organize them into logical sets. Except for very simple student programs, one set of test data will never completely validate a program. For large-scale development projects, 20, 30, or even more test cases may need to be run to validate a program.

Finally, while we are testing, we will think of more test cases. Again, write them down and incorporate them into our test plan. After our program is finished and in production, we will need the test plan again when we make modifications to the program.

How do we know when our program is completely tested? In reality, there is no way to know for sure. But there are a few things we can do to help the odds. While some of these concepts will not be clear until you have read other chapters, we include them here for completeness.

1. Verify that every line of code has been executed at least once. Fortunately, there are programming tools on the market today that will do this for us.
2. Verify that every conditional statement in our program has executed both the true and false branches, even if one of them is null. (See Chapter 5.)
3. For every condition that has a range, make sure the tests include the first and last items in the range, as well as items below the first and above the last—the most common mistakes in array range tests occur at the extremes of the range. (See Chapter 8.)
4. If error conditions are being checked, make sure all error logic is tested. This may require us to make temporary modifications to our program to force the errors. (For instance, an input/output error usually cannot be created—it must be simulated.)

1.6 Software Engineering

Software engineering is the establishment and use of sound engineering methods and principles to obtain software that is reliable and that works on real machines.[7] This definition, from the first international conference on software engineering in 1969, was proposed 30 years after the first computer was built. During that period, software was more of an art than a science. In fact, one of the most authoritative treatments of programming describes it as an art: *The Art of Computer Programming*. This three-volume series, originally written by Donald E. Knuth in the late 1960s and early 1970s,[8] is considered the most complete discussion of many computer science concepts.

Because the science and engineering base for building reliable software did not yet exist, programs written in the 1950s and 1960s were a maze of complexity known as "spaghetti code." It was not until Edsger Dijkstra wrote a letter to the editor of the *Communications of the ACM* (Association of Computing Machinery)[9] in 1968 that the concept of structured programming began to emerge.

Dijkstra was working to develop algorithms that would mathematically prove program accuracy. He proposed that any program could be written with only three constructs or types of instructions: (1) sequences, (2) the *if…else* selection statement, and (3) the *while* loop. As we will see, language developers have added constructs, such as the *for* loop and the *switch* in C. These additional statements are simply enhancements to Dijkstra's basic constructs that make programming easier. Today, virtually all programming languages offer structured programming capabilities.

Throughout this text we will be emphasizing the concepts of good software engineering. Chief among them is the concept of structured programming and a sound programming style. A section in each chapter will include a discussion of these concepts with specific emphasis on the application of the material in the chapter.

The tools of programming design have also changed over the years. In the first generation of programming, one primary tool was a block diagram. This tool provided boxes, diamonds, and other flowchart symbols to represent different instructions in a program. Each instruction was contained in a separate symbol. This concept allowed programmers to write a program on paper and check its logic flow before they entered it in the computer.

With the advance of symbolic programming, the block diagram gave way to the flowchart. Although the block diagram and flowchart look similar, the flowchart does not contain the detail of the block diagram. Many instructions are implied by the descriptive names put into the boxes; for example, the read

7. F. L Bauer, Technical University, Munich, Germany (1969).
8. Donald E. Knuth, *The Art of Computer Programming,* vols. 1 (Third Edition, 1997), 2 (Third Edition, 1997), and 3 (Second Edition, 1998). (Reading, Mass.: Addison-Wesley, 1977).
9. Edsger W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, vol. 11, no. 3 (March 1968).

statements in Figure 1-13 imply the prompt. Flowcharts have largely given way to other techniques in program design, but they are still used today by many programmers for working on a difficult logic problem.

Today's programmers are most likely to use a high-level design tool such as tight English or pseudocode. We will use pseudocode throughout the text to describe many of the algorithms we will be developing.

Finally, the last several years have seen the automation of programming through the use of computer-assisted software engineering (CASE) tools. These tools make it possible to determine requirements, design software, and develop and test software in an automated environment using programming workstations. The discussion of the CASE environment is beyond the scope of this text and is left for courses in systems engineering.

1.7 Tips and Common Errors

1. Become familiar with the text editor in your system so you will be able to create and edit your programs efficiently. The time spent learning different techniques and shortcuts in a text editor will save time in the future.
2. Also, become familiar with the compiler commands and keyboard shortcuts. On most computers, a variety of options are available to be used with the compiler. Make yourself familiar with all of these options.
3. Read the compiler's error messages. Becoming familiar with the types of error messages and their meanings will be a big help as you learn C.
4. Remember to save and compile your program each time you make changes or corrections in your source file. When your program has been saved, you won't lose your changes if a program error causes the system to fail during testing.
5. Run your program many times with different sets of data to be sure it does what you want.
6. The most common programming error is not following the old proverb to "resist the urge to code." Make sure you understand the requirements and take the time to design a solution before you start writing code.

1.8 Key Terms

application software
application-specific software
assembly language
auxiliary storage
blackbox testing
central processing unit (CPU)
client
client/server
compilation
compiler
computer language
computer system
distributed environment
executable program
flowchart
general-purpose software
hard copy
hardware
high-level language
input device
linker
operating system
output device
personal computer (PC)
preprocessor
preprocessor commands
primary storage
program development
pseudocode
secondary storage
server
soft copy
software
source file
structure chart
symbolic language
system development life cycle
system development software
system software
system support software
text editor
time-sharing environment

loader	translation unit
machine language	translator
main memory	waterfall model
object module	whitebox testing

1.9 Summary

- ❑ A computer system consists of hardware and software.
- ❑ Computer hardware consists of a central processing unit (CPU), primary memory, input devices, output devices, and auxiliary storage.
- ❑ Software consists of two broad categories: system software and application software.
- ❑ The components of system software are the operating system, system support, and system development.
- ❑ Application software is divided into general-purpose applications and application-specific software.
- ❑ Over the years, programming languages have evolved from machine language, to symbolic language, and to high-level languages.
- ❑ The C language is a high-level language.
- ❑ The software used to write programs is known as a text editor.
- ❑ The file created from a text editor is known as a source file.
- ❑ The code in a source file must be translated into machine language using the C compiler, which is made of two separate programs: the preprocessor and the translator.
- ❑ The file created from the compiler is known as an object module.
- ❑ An object module is linked to the standard functions necessary for running the program by the linker.
- ❑ A linked program is run using a loader.
- ❑ The system development life cycle is a series of interrelated steps that provide a rigorous and systematic approach to software development.
- ❑ To develop a program, a programmer must complete the following steps:
  - **a.** Understand the problem.
  - **b.** Develop a solution using structure charts and either flowcharts or pseudocode.
  - **c.** Write the program.
  - **d.** Test the program.
- ❑ The development of a test plan starts with the design of the program and continues through all steps in program development.
- ❑ Blackbox testing consists primarily of testing based on user requirements.

- ❑ Whitebox testing, executed by the programmer, tests the program with full knowledge of its operational weaknesses.
- ❑ Testing is one of the most important parts of your programming task. You are responsible for whitebox testing; the systems analyst and user are responsible for blackbox testing.
- ❑ Software engineering is the application of sound engineering methods and principles to the design and development of application programs.

1.10 Practice Sets

Review Questions

1. Computer software is divided into two broad categories: system software and operational software.

   a. True
   b. False

2. The operating system provides services such as a user interface, file and database access, and interfaces to communications systems.

   a. True
   b. False

3. The first step in system development is to create a source program.

   a. True
   b. False

4. The programmer design tool used to design the whole program is the flowchart.

   a. True
   b. False

5. Blackbox testing gets its name from the concept that the program is being tested without knowing how it works.

   a. True
   b. False

6. Which of the following is a component(s) of a computer system?

   a. Hardware
   b. Software
   c. Both hardware and software
   d. Pseudocode
   e. System test

7. Which of the following is an example of application software?

   a. Database management system
   b. Language translator

- c. Operating system
- d. Sort
- e. Security monitor

8. Which of the following is not a computer language?
   - a. Assembly/symbolic language
   - b. Binary language
   - c. High-level languages
   - d. Machine language
   - e. Natural language
9. The computer language that most closely resembles machine language is
   - a. Assembly/symbolic
   - b. COBOL
   - c. FORTRAN
   - d. High-level
10. The tool used by a programmer to convert a source program to a machine language object module is a
    - a. Compiler
    - b. Language translator
    - c. Linker
    - d. Preprocessor
    - e. Text editor
11. The ________________ contains the programmer's original program code.
    - a. Application file
    - b. Executable file
    - c. Object file
    - d. Source file
    - e. Text file
12. The series of interrelated phases that is used to develop computer software is known as
    - a. Program development
    - b. Software engineering
    - c. System development life cycle
    - d. System analysis
    - e. System design
13. The ________________ is a program design tool that is a visual representation of the logic in a function within a program.
    - a. Flowchart
    - b. Program map
    - c. Pseudocode
    - d. Structure chart
    - e. Waterfall model

14. The test that validates a program by ensuring that all of its statements have been executed—that is, by knowing exactly how the program is written—is
    a. Blackbox testing
    b. Destructive testing
    c. Nondestructive testing
    d. System testing
    e. Whitebox testing

Exercises

15. Describe the two major components of a computer system.
16. Computer hardware is made up of five parts. List and describe them.
17. Describe the major differences between a time-sharing and a client/server environment.
18. Describe the two major categories of software.
19. What is the purpose of an operating system?
20. Identify at least two types of system software that you will use when you write programs.
21. Give at least one example of general-purpose and one example of application-specific software.
22. List the levels of computer languages discussed in the text.
23. What are the primary differences between symbolic and high-level languages?
24. What is the difference between a source program and an object module?
25. Describe the basic steps in the system development life cycle.
26. What documentation should a programmer receive to be able to write a program?
27. List and explain the steps that a programmer follows in writing a program.
28. Describe the three tools that a programmer may use to develop a program solution.
29. What is meant by the old programming proverb, "Resist the temptation to code"?
30. What is the difference between blackbox and whitebox testing?
31. What is software engineering?

Problems

32. Write pseudocode for *calcLivingAreas*, Figure 1-12, "Structure Chart for Calculating Square Footage."
33. Create a flowchart for a routine task, such as calling a friend, that you do on a regular basis.
34. Write pseudocode for the flowchart you created in Problem 33.