CHAPTER 1

Introduction to Computers and Programming

Objectives

To understand the basic concepts of the hardware and software components of a computer system, program development, program compilation and execution in C, and the C program processing environment.

General purpose computers have become an important part of everyday life, particularly with the evolution of internet technology. Computers and computer programming are essential components in modern problem solving and in the technology of information transfer. A computer can store large volumes of data with fast access, perform complex computations at high speed, and accurately and efficiently share the resulting information. Solid-state electronics and integrated circuits [particularly very large scale integrated (VLSI) circuits], have revolutionized the computer industry, making it possible to build increasingly powerful computers with storage capacity and processing power at decreasing cost. Modern machines are compact, efficient, affordable, and indispensable in every area of research and applications used in industry, universities, medicine, music, entertainment, business, and communications, among others.

Computers are used in educational institutions for teaching, research, and administrative activities; in industry they are used for research, design, and manufacturing; in medical sciences for research, diagnosis, treatment, and record keeping; in aeronautics and space exploration for the design, control, and navigation of space vehicles; in the exploration of natural resources, for weather forecasting, film production, and in other such fields. Special types of computer systems are used for engineering analysis and design, to construct intelligent decision systems, to process natural language documents, to control production, and in the development of expert diagnostic systems, in speech recognition, image processing, and robotics.

Because computers and computer programs are widely used in mission critical applications, cost effectiveness, speed, and reliability are very important factors. The degree of reliability needed depends on the application. For applications such as space exploration, nuclear power production, defense, and medical technology, failure of computerized devices is not acceptable. A high degree of reliability of the entire system is critical and essential. The concept of reliability has led to the development of fault-tolerant computer systems. Such systems are designed not to fail (hardware or software) under any circumstances. A critical part of these systems is the reliable and failure-free computer program.

Computer users have little control over the speed and accuracy of the computers they are using. However, users who write their own programs can make decisions affecting the speed of processing by how they choose to have the computer store the data, how it is accessed, and the method used to solve the problem. For example, accessing data randomly is slower than accessing it sequentially and trial and error methods of solution are slower than using formulas. Program writers can also influence the accuracy of answers by the choice of formulas, the data storage formats, and the treatment of significant digits in numeric data. They can influence the reliability of answers by building checkpoints into the program.

1.1 Concept of Computers and Computer Systems

Computers are electromechanical devices that function semi-automatically and are capable of accepting instructions and data, performing computations, and manipulating data to produce useful results. The term *hardware* applies to the collection of all of the physical components that constitute a computer. *Software* is

the collection of all of the programs that run on the hardware of the computer. Each program consists of a sequence of instructions, which directs the computer to input and manipulate data based on the solution algorithm to produce an answer to the problem and output the results. *Firmware* is the part of hardware that is permanently programmed; this type of built-in software is essential in order to activate (*boot*) the system.

A *computer system* consists of hardware, software, and firmware. Computers are categorized according to their size (memory and number of processors), functions, and areas of applications. With the revolution of internet technology, they are further classified according to design and function as general-purpose, server, workstation, web hosting, and network computers. *General-purpose computers* can run many different programming languages and solve many different types of problems. *Special-purpose computers* use a single programming language designed for a specific type of application.

General-purpose computers are broadly classified as mainframe, super, personal, and laptop based on physical size, memory size, word size, processing speed, number of peripheral devices supported, and network support. *Mainframe computers* are found in installations where there is high demand for computer power and a large amount of data to be processed. *Supercomputers* are extremely fast machines used primarily for complex and scientific computations. *Personal computers* (PCs) are found in homes, schools, libraries, businesses and other small installations, and in laboratories where there are limited amounts of data to process. Personal computers are used primarily for science and engineering applications, bookkeeping, writing, and personal use. The present generation of personal computers provides powerful computational capability. *Laptops* are small portable computers with capabilities similar to those of personal computers.

Servers and workstations are special purpose machines. *Servers* are large computer systems that store large varieties of programs and databases to provide service to the organizations and people using the internet. Servers are the backbone of the internet, which supports millions of users and many types of applications including email service and e-commerce. *Workstations* are capable of performing large computations. They can be configured based on the memory and processing needs to be stand-alone units with processing power and graphics capabilities, or they can be connected to other computers for special activities such as email and internet access.

Most current computers store and then process the instructions one at a time in a predetermined sequence and store intermediate results of computations. Advances in hardware technology have made it possible to approach the limit of

electronic speed. Further speed increase will be attained by packaging electronic elements closer together so that the signals travel shorter distances. The electronic speed limits the amount of computing the computer can complete within a given period of time.

1.1.1 Hardware Components and Functions

The primary hardware components of a computer are the input/output devices, memory units (both primary and secondary), and processing units. The *central processing unit* (CPU) consists of the arithmetic-logic unit (ALU) and the control unit (CU). The block diagram of a typical system is shown in Figure 1.1. Large systems also have support hardware, which performs addi-

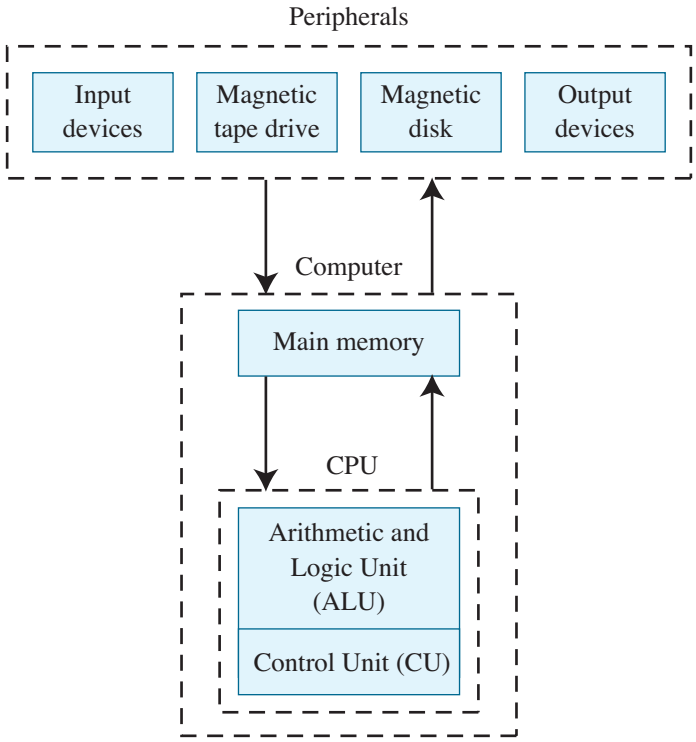


Figure 1.1 Block diagram of a computer

tional specialized functions such as graphic display, networking, image processing, or managing secondary storage. All of these are primarily electronic, consisting of digital logic gates and circuits that control and drive the mechanical processes.

Input/Output (I/O) Devices *Input devices* are used to input instructions and data into the computer, usually in character form. Some installations use optical scanners or magnetic scanner devices to read data on marked and printed paper and transmit it to the computer memory. For some purposes, punched card readers are used. Key entry devices, such as interactive terminals, are the most common form of input device used today. Instructions and data are typed at the keyboard and the characters are transmitted directly or indirectly to main memory. In recent hand-held computers, such as the palm pilot, touch pen input devices are used.

Output devices make the processed information available in the form of printouts, screen displays, graphics, sound and speech, or file output on a disk. The most common output devices are printers, displays, graphics terminals, and disk drives. A *printe*r prints on paper the output information that was stored in main memory. Some printers print characters one at a time like typewriters; others print one line at a time. Dot matrix printers are usually used for rough drafts. Higher quality output is produced by laser, chain, inkjet, and other special printers, such as those needed for desktop publishing. Large computer installations use line printers and higher speed laser printers. *Magnetic storage devices* such as tape drives and disk drives are used for both input and output. Although they are peripheral devices, they are part of the system. Magnetic storage devices provide secondary storage. They hold large volumes (gigabytes and terabytes) of data in very little space. Data values are coded in magnetic dots, representing zeros and ones. Magnetic devices are convenient, cost effective, and store information permanently, but the data stored on them can only be read by computers. These devices come in many different forms, with different access speeds and storage capacities.

Memory Devices *Memory devices* store both program instructions and data. The main memory, also called random access memory (RAM), is used for storing instructions currently being executed and data currently being processed. This is fast memory with microsecond to nanosecond speed. The *processor* communicates directly with the main memory, fetching instructions and fetching and storing data. Many of today's desktop personal computers have more RAM, which

makes them more powerful, and have greater processing speeds and capabilities than the room size computers of only a few years ago.

Central Processing Unit The central processing unit (CPU) consists of the control unit and the arithmetic-logic unit. The *control unit* (CU) contains electronic circuitry that fetches the instructions from memory and decodes them. The decoded signals for the operation code and the operands are sent to the arithmetic-logic unit (ALU), directing it to carry out the arithmetic and logic operations. The CU stores the results of the arithmetic and logic operations in main memory. It also supervises the input and output of data. The details of input and output are usually handled by independent processors called input/output (I/O) processors.

The *arithmetic-logic unit* (ALU) contains the electronic circuitry that performs standard arithmetic operations and makes logic decisions by comparing values. The arithmetic circuits can add, subtract, multiply, and divide, using two numbers at a time. The result of such operations is numeric. The logic circuits can compare either numeric or character values. The output of the logic circuits is interpreted as a logical value of true or false. The logic circuits can also perform standard logical operations: selecting, testing, and altering logical values. In large mainframe and supercomputers, the ALU often contains coprocessors so that logic decisions may be made at the same time calculations are being carried out. *Coprocessors* are also very common in microcomputers, especially when using mathematically oriented languages. All computer instructions are based on these elementary operations of fetching, arithmetic, comparison, and storing.

1.1.2 Software Components and Functions

A *program* is a sequence of instructions written in a programming language that directs a computer in problem solving. The *software* in a computer system consists of programs written to support the basic operations of the system and programs written to carry out an application. There are several levels of software active at all times. Figure 1.2 shows some of these software systems and their hierarchy.

The software that controls the execution of an application program is called *system software*. The major system software components are the operating system and the programming language systems, which include utility programs and library routines. These programs keep the computer functioning efficiently and provide a comfortable environment for the user. They allow the user to access data in a variety of forms and to set up filing systems for data.

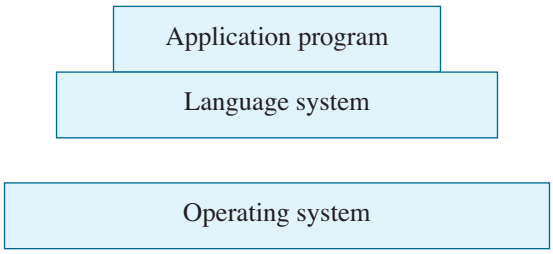


Figure 1.2 Application and system software

The *operating system* makes the system facilities available to the application programs and controls their use. It provides access to a variety of programming and debugging tools and provides an interface between the user and the computer hardware. It schedules program execution and directs traffic through the computer. The operating system manages all the system resources—allocating and deallocating memory, processor, devices, and files to a particular application program as they are needed. Operating systems have their own nonstandardized control languages. Instructions in these languages direct the computer to undertake tasks and make resources available to the tasks. Every programmer needs at least minimal knowledge of the control language of the computer being used.

System management is designed to balance processing and input/output, with the aim of providing reasonable minimum turnaround and maximum throughput. *Turnaround* is the amount of time elapsing between a request to the computer to execute a program and the availability of output. *Throughput* is measured by the number of jobs that are completed in a given time period.

With interactive systems, response time is also important. *Response time* is the time lapse between interactive input and output that indicates that the input has been received.

The operating system contains system utility programs and a system library. *System utility programs* are data management and device management system software. The data management software manages the formatting of the input/output. The device management software makes the devices available as though they are extensions of memory. The *system library* contains graphics packages, mathematics packages, statistics packages, database management routines, and data communication and networking software.

The *file management system,* which is part of the operating system, controls the storage and retrieval of records from program and data files, which are

normally stored on magnetic devices such as disks. It provides instructions and data to memory as the processor needs them.

Language systems are classified as high-level or low-level depending on their similarity to human languages or hardware languages. Languages that are reasonably machine independent and people-oriented are called *high-level languages*. *Compilers* are language systems that translate programs written in high-level languages such as FORTRAN, C, and C++ into machine code, which the hardware can interpret and execute directly. Assembly language and other machine-oriented languages are called *low-level languages*. Programs written in assembly language are also assembled (translated) to machine code. Compilers that translate source code into particularly efficient machine code are called *optimizing compilers*.

Application software consists of programs written to analyze data and solve specific problems. Application programs produce output concerning the exterior world, while system programs produce output concerning the state of the computer system. Application programs may be written by users, but more often are written by professional programmers. Programmers may write a program to compute a payroll, for example, or to implement an automatic navigation system for an aircraft. When adequate application software can be purchased, programmers often modify it to their company's specific needs.

When an engineer uses a computer to process data, the engineer uses an input device to tell the operating system what application software to run. The application program requests data, calls on the system library for routines that decode the data, and converts it into a computer-usable format. The application program then processes the data, calling on the system library for standard mathematical routines and for routines that convert the results to a form the engineer can understand. It then calls on the file management system to store the results until the engineer needs them.

When an engineer writes his or her own application program, the program must be thoroughly tested on the computer before it can be used to process data. To do this, the engineer uses an input device to tell the operating system which compiler to run. The engineer has prepared a sequence of instructions in the programming language that are input as data to the computer and stored on a disk. The compiler checks the instructions one by one for spelling and syntax, translates them into a machine-usable format, and calls on the file management system to store the results until they are needed. Before finishing and returning control to the operating system, the computer sends error messages and program statistics to the engineer.

Before an engineer writes an application program and submits it to the computer it is necessary to design the input layout of the data, the output layout of the computational results, and to thoroughly think through the steps involved in processing the data.

1.1.3 Integration of Hardware and Software

Hardware cannot function by itself without software; software cannot function unless it has hardware to run on. Both are necessary to make a computer system work properly. Programs written in high-level languages go through several levels of translation and interpretation before the machine can execute them. Application programs written in C are translated by the compiler into a machine language that is then interpreted for execution by the digital logic gates and circuits.

1.1.4 Review Questions

1. A computer system consists of both _____ and _____.
2. Three major hardware components of a computer are _____, _____, and _____.
3. A compiler translates a high-level language program to _____.
4. The arithmetic-logic unit of a computer performs _____ and _____ operations.
5. Machine language instructions are fetched and decoded by the _____ unit.
6. Input/output operations are controlled by the _____ unit.
7. The input devices handle instructions and data in _____ form.
8. Common secondary storage devices are _____ and _____.
9. Programs written to manipulate data and solve problems are called _____ software.
10. The central processing unit consists of _____, _____, and _____.
11. The program in a computer system that manages the computer resources is called an _____ system.

12. Software programs permanently installed to control equipment are called _____.

1.2 Modular Programming

Modularization is the process of breaking down a complex problem into easily solvable, manageable, and functional units. Modularization is based on the concept of "divide and conquer." For example, some problem solutions have the form:

1. Get data
2. Process data
3. Output results

Other problem solutions may have the form:

1. Set up a table
2. Put data in the table
3. Analyze the table
4. Output the results

The process of modularization would then call for more detailed descriptions as to where to get the data, how to validate it, how to process it, how to arrange it in a table, how to analyze it, and so forth.

To understand and analyze a complex problem may be difficult, but once the problem is subdivided into smaller understandable pieces called subproblems, it is easier to understand the original problem and design a process to solve it.

1.2.1 Modular Design

The first step in modular design is to divide the complex problem into major subproblems. Then the major subproblems are divided into further sub-problems until they are simple to solve. The problems and subproblems are called *modules*. Modules should be designed in such a way that they are *loosely coupled*. Loose coupling simply means that if a change is made in any of the modules it does not force change in other modules. There are two types of modules: functional modules and spatial modules. A module with a specific function or task to perform is called a *functional module*. If there is a repeated process in several places in the

overall design, such a process can be made into a *module* and such a module can be used to replace the repeated process. It is called a *spatial module*. The C programming language provides the facility to represent these functional and spatial modules so that they are loosely coupled.

During the design process the user requirements are spelled out. A prototype of the design is developed and tested to see whether it meets the requirements. During the design process the designer must document the design and provide the capability for future updates. The design steps may be repeated several times before writing the application program, to eliminate any design flaws. If there are any design flaws in the solution algorithm, it may be too late to correct them when the final product is found to be functionally incorrect. Therefore, we cannot overstress the importance of the solution algorithm design process. This is the most crucial phase in the development of any software.

1.2.2 Structure Charts

A structure chart is a design tool that represents the functional and spatial modules of the algorithm and their relationship to one another. In a structure chart, the statement of the problem to be solved is represented in the form of a rectangular box at the top level. For example, in Figure 1.3 "Calculate the square root of a number" is the statement of the problem. The functions in the next level are represented in rectangular boxes horizontally from left to right in the order in which they are activated. If there is any function in the second level that needs to be further divided into subfunctions, they are represented in the next level. The functions at each level are represented from left to right and the further subdivision of the functions at each level is represented vertically.

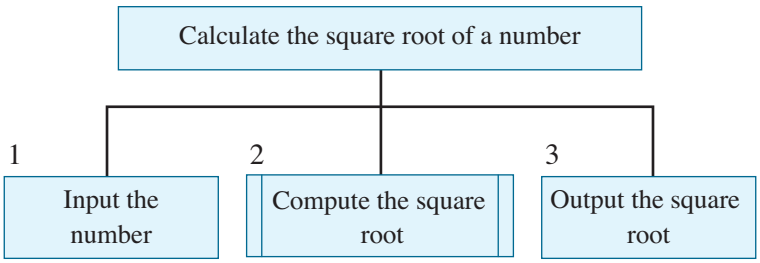


Figure 1.3 Structure chart

The structure chart represents the analysis of the problem of calculating the square root of a number. It also represents program modules. The module "Calculate the square root of a number" invokes the modules beneath it, passing data back and forth to them.

1.2.3 Functional Modules

Modules communicate with one another by passing information back and forth. In C, these modules receive numeric values and manipulate them, and the result is conveyed back to the invoking module. An invoking module is a function that calls another function. C provides the facility to write these modules as independent units of a program. These units can be compiled, debugged, and tested independently, then linked together into an executable program. Modules can be input modules, computational modules, or output modules. There are a number of functional modules available in C's rich library of functions such as `sqrt()` to compute a square root of a number or `max()` to compute a maximum of a set of values. There may also be functional modules written by other people that are available for use. Programmers write their own functional modules for specific problems and, in addition, they can use standard library functions and library functions written by other people, as they fit into their application. C is rich in library functions, which will be presented throughout the text. The list of all library functions is provided in Appendix B.

1.2.4 Review Questions

1. In modular design the large problem is divided into _____.
2. What is the relationship between lower-level and higher-level modules in a hierarchy chart?
3. Modules must be tightly coupled. True or False? Explain.

1.3 Algorithms and Program Development

Solving a problem on a computer requires a thorough understanding and analysis of the problem and of the potential data. Once the problem has been analyzed, the detailed design of a solution can be developed. One of the steps in designing and

developing a computerized solution to a problem is to develop an algorithm to solve the problem. At the same time, the algorithm must filter out inappropriate data.

1.3.1 Concept of an Algorithm

An *algorithm* is a procedure consisting of a finite number of precisely defined steps for solving a problem. Each step of an algorithm must be an unambiguous instruction which, when written in a computer language, can be executed by a computer. The order of the steps is critical, because most computers execute steps in the order in which they are presented. An algorithm must terminate whether or not the task is completely successfully executed. That is, it must recognize the end of the input data and when an answer is sufficiently accurate. Algorithms that monitor the weather, automatic teller machines, and nuclear reactors are continually active. These types of algorithms are apparent exceptions to the termination requirement. However, they idle as they continuously poll input devices and are normally reactivated as needed.

Programming Warning: Every step in the algorithm must be clear and precise and the algorithm must terminate.

Programmers should choose an algorithm on the basis of efficiency, accuracy, reliability, and clarity. Algorithms should be efficient with respect to computational time, storage requirements, and response time. The degree of accuracy required is specified by the user. An algorithm is reliable when it consistently produces correct answers from valid data and rejects invalid data. Clarity means that an understandable programming style is used. Clarity should not be compromised. However, sometimes a compromise between the other factors is necessary. For example, efficiency may be sacrificed for the sake of a high degree of reliability.

The design of algorithms to solve simple problems can be straightforward, but design of algorithms for large, complex problems can be difficult and time consuming. Although this text can only deal with simple problems, the techniques we use are important in the design of algorithms for complex problems.

One common approach to large and complex problems is to use top-down design. Top-down design starts at the top of the structure chart with a general statement of the problem written in a precise, formal way to provide a high-level specification of the algorithm. This is divided into separate logical parts and a

general specification for the solution of these parts is given. These parts correspond to major modules in the final algorithm. The parts are then further subdivided and specifications are drawn up. Finally the algorithm reaches a stage where the specification consists of computations, comparisons, and data access that can be programmed without further explanation.

An Example of a High-Level Problem Specification Calculate, in pounds, the total amount of steel required to build a pipeline to carry water.

The next step would be to divide the problem into logical parts as follows:

1. Input the pipeline dimensions
2. Check their validity
3. Calculate the cross-sectional area
4. Calculate the volume of steel
5. Calculate the weight
6. Output the weight

Each stage of the algorithm should be carefully checked before the algorithm is written in a computer programming language and again before it is tested on a computer. If the design is not proved correct at each stage, it may contain errors that make it necessary to start over. Careful validation during the design process leads to a more nearly correct solution.

1.3.2 Concept of Programs and Data

A *program* is a sequence of executable, unambiguous instructions written in a computer language. The computer can understand instructions of various types: *input/output* instructions to input data into the computer and output answers from it; *move* instructions to rearrange data; *arithmetic* instructions to perform calculations; *control* instructions to control selection and repetition of actions; *logic* instructions to help the computer make choices. These types of instructions are available in most programming languages. In C there are also instructions to control where data is stored in the computer memory.

Input refers to data sent to the computer processor from a file on an automatic storage device or from an outside source. Input may come from a keyboard or an automatic device such as a weather station.

Output refers to data sent out from the computer processor to automatic storage devices or to an outside device or a person. Output may be printed, displayed on a CRT screen, written on a tape, disk or diskette, plotted (rendered) on a graphic device, or used to control an automatic device.

The internal rearrangement of data, during which values are copied from one memory location to another, is achieved primarily through assignment instructions, which look like simple equations. For Example, the C instruction

```
a = b;
```

does not mean that `a` and `b` are the same things, but rather that `a` is being assigned (given) the same value as `b`. In effect the value contained in the memory location named `b` is copied into the memory location named `a`. After the statement `a = b` is carried out by the computer the memory location `a` will contain the same value as the memory location `b`.

Arithmetic instructions consist of the basic arithmetic operations of addition, subtraction, multiplication, division, remainder, and exponentiation (raising to a power), as well as assigning the answer to a variable. Thus,

```
a = 2.5 + 6.5 / 3.25;
```

means that 4.5 is calculated as the value assigned to `a`. Arithmetic expressions are evaluated by the computer with the same operator precedence as they are evaluated in mathematics. In this example, the division is performed before the addition.

Control instructions are used to switch from one set of instructions to another depending on the logical comparison of data values. For example, the computer might choose between addition or subtraction depending on whether a number is positive or not.

```
if(x > 0)
   z = x + 5;
else
   z = x – 5;
```

Control instructions always involve the comparison of one value with another for equality or relative size. All comparisons result in an answer of true or false, which becomes the basis for a control decision. The logical comparisons are presented in Chapter 4.

Concept of Data The collection of related information to be processed by an application program such as temperatures to be averaged over a period of time is known as *data*. Data can be numeric, character, or logical. Numeric data can be written as decimal numbers, as integers, or in scientific notation. Numeric data values are treated as numeric values even though they are written as strings of digits and special characters. In C, character data values are written as characters inside a single quote as `'a'` or strings of characters inside double quotes as `"axbcd"`. While C does not have a data type specifically for logical data, integers are used to represent true and false. Zero is interpreted as false, and all other integers are interpreted as true.

NUMERIC DATA	CHARACTER STRING DATA
254	"JAMES MILLER"
−52.75	"LMNOP3476"
0.125	"LEEMAN BROTHERS"
+17.357	"+17.357"
0.32987653E02	"+=23765"

Data values are usually stored in a file on the disk. A *data file* is a repository to store data and contains a structured collection of data. Input data must be structured in a systematic way. Every program includes a description of the data structure so that the computer knows how to interpret it. If a program is to find its input data on a disk, the data must be keyed into a data file. Input data can also be keyed directly into an executing program. When data values are keyed directly, it is particularly important that the program validate the data before using it. In either case, the end of the input data must be recognizable. When data values are keyed directly, a special control character is used as an end symbol.

Programming Hint: Check that input data is valid. Validation of input data is critical.

Data generated by the computer for output can be of many types: solutions to mathematical equations, numbers, pictures, graphs, textual material, or special symbols. Output data must be formatted properly for the output device by having appropriate vertical and horizontal spacing. Data must be in a usable form. Numbers printed or displayed should be arranged and labeled with headings, subheadings, and whatever other identification is helpful.

Output should be understandable, easy to use, and attractive. Good output is worth the time expended in designing and formatting it.

Programming Hint: Make output readable and visually attractive. Use titles and column headings. Align data. Output date and time in reports.

Example: Tabular output (columns labeled)

PARTS INVENTORY REPORT (11/17)

PART NAME	UNITS	UNIT COST	TOTAL COST OF UNITS
Bolts	50	0.50	25.00
Nails	500	0.01	5.00
Struts	200	0.10	20.00
…..	…….	……..	
Screws	400	0.05	20.00

Example: Interactive output (values labeled)

```
MATERIAL VOLUME
DIAMETER = 5     LENGTH = 10     VOLUME = 195.35
……               ……              ……
DIAMETER = 8     LENGTH = 14     VOLUME = 427.85
```

Not all values calculated by a program are part of the output data. Such values as counts, intermediate computational results, and logical values that control processing are *internal* data. Internal data also includes status flags for various pieces of equipment and for functions of the operating system.

To a computer, data consists of more than just numbers or character strings. A set of data values is identified as being of a particular data type, being written in a particular notation, being suitable for certain operations, and being given a name by which it can be identified. There also may be criteria for validity and attached units of measurement. For example, data representing the speed of a car would be numeric, be written with a decimal point, be called "speed" or "velocity," be in miles/hour or kilometers/hour, be used in arithmetic, or be used as output, and would not be a negative number. Data representing an identification number would be characters, written within double quotes, might be either alphabetic characters or digits, and could be used only for comparison and for output.

1.3.3 Procedure for Problem Analysis

To analyze a problem, besides writing a precise specification of the algorithm, the user should write precise specifications for the input and output data. These specifications should include initial conditions and boundary conditions, such as those associated with mathematical models and computational simulations, in addition to the equations or formulas to be used. Some problems do not have exact solutions. Such problems should be analyzed for ways of obtaining approximate solutions, determining the quality of these solutions and the degree of accuracy needed. Side effects, options, and possible disastrous cases must be considered. Once the problem is clearly understood and the associated side effects are resolved, programming can begin.

Problem Types and Solution Methods There are engineering and scientific type problems requiring solutions for which empirical equations have been derived with known and unknown variables. There are also problems for which closed-form solutions are known using appropriate formulas with known and unknown variables. There are other problems for which mathematical models have been developed in the form of a process to be followed using dependent and independent variables. All of these equations are solved by some appropriate numerical scheme.

There are three basic types of solutions. First, when a closed-form solution to a problem is known, the appropriate equations can be used in a program. Second, when no closed-form solutions are known, or when the mathematical solution involves trial and error, numerical methods can be used to approximate the solution. Third, some types of problems have so many independent variables that rather than a single answer, a variety of possible solutions is desirable. In addition, there are problems having more than one solution where it may be satisfactory to find any one solution rather than all of them.

Data Types The procedure for problem analysis includes analyzing categories of possible input data. In some cases the input values are external to the program, in other cases they are generated internally. When external data values are used, the input must be validated by the computer according to clearly defined specifications as the computer cannot guess the intent of the person entering the data. Data specification includes such requirements as the type of data expected (numeric or character), the form of the data (integer, real numbers, complex num-

bers, logical data, etc.), the size of the data (number of digits or characters), the arrangement of the data (lists, tables, records, etc.), and any restrictions on the range of the data values. For example, a set of measurements of the depth of water in a lake might be assumed to contain at least one measurement; be positive, real numbers rather than integers; be significant to two decimal places; and have a unit of measurement such as meters. Program documentation should state this and the algorithm should check that the data meets these conditions.

Design step: Describe the input and output formats.

1.3.4 Solution Design Methodology

Once the problem has been clearly stated and analyzed, and the details of the data specified, the solution can be developed. This involves the actual design of the input, output, internal data, the development of an algorithm, and the development of test data for the algorithm. For example, the computer must know whether the data values consist of integers or real numbers and whether they are in normal decimal notation or scientific notation. It must know how many data values are expected and how to identify the end of the list of data values. If the algorithm includes a repeated calculation, assumed preconditions and postconditions must be recognized.

At the design stage, the algorithm can be represented by structure charts, pseudocode and/or flowcharts, or by other logic representation techniques. Consider the simple problem of calculating and printing the average depth of water in a lake with a set of sample depth measurements. The basic steps in the calculation are:

1. Input, validate, and add the data values one by one, counting the number of data values
2. Divide the sum of the data values by the number of data values
3. Print the average depth

The first of these steps includes repeated input, validation, and addition. The program instructions must include information about the variables to be added and incremented in the repetition loop, and how to detect the end of the data. The instruction to calculate the average and print the average follows the repetition loop. Step 1 will only work if there is an internal variable to use to accumulate the sum of the data values, and an internal variable to use as a counter. Both of these

must be cleared before any data is processed. The precondition for Step 1 is that total = 0 and count = 0 where total and count are names for the internal variables. The precondition for Step 2 is that count > 0. Therefore there must be at least one positive data value.

The simplest pseudocode description of this algorithm is as follows:

```
initialize total and count
process positive data values in a set of data values
calculate average
print average
```

This could also be drawn as a flowchart as shown in Figure 1.4.

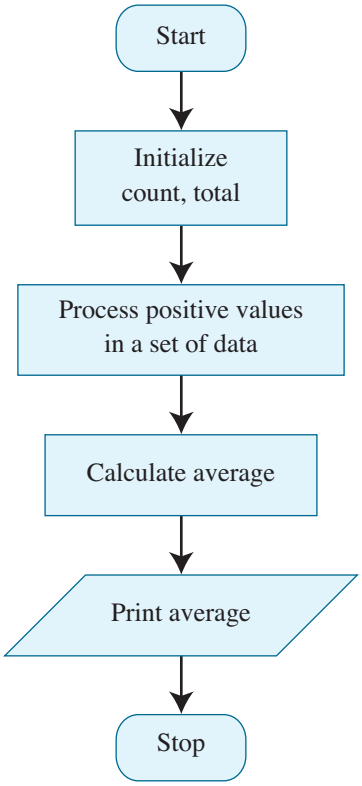


Figure 1.4 Flowchart to compute an average

In the flowchart in Figure 1.4, arrows indicate the flow of control while different shaped boxes are used for different types of operations. Unless there are many data values to be processed the same way, the flow of control is strictly from the top down.

The following is a pseudocode description of this algorithm that assumes there is at least one valid data value.

Algorithm

```
Initialize the total and the count to zero (This is necessary to meet the first
      precondition)
For each input data value greater than 0      (precondition: assume total ≥ 0, count ≥ 0
                                               every time a new data value is obtained)
   Add the data value to the total
   Add 1 to the count
End for
Divide the total by the count to get the average depth ( precondition; assume count > 0)
Print the average depth
Stop
```

Programming Hint: Note that totals and count must always be initialized to zero to reflect the possibility that there may not be any data to process.

This can be drawn as a flowchart as shown in Figure 1.5.

The steps in the flowchart in Figure 1.5 can be described either in language or in mathematical notation. Note that we have not yet considered how to identify the end of the data or how to control the repetition.

The part of the pseudocode that is bracketed by `For` and `Endfor` processes the entire set of data values and can be drawn in flowchart form as shown in Figure 1.6.

Note that each data value is processed separately; therefore, there is a loop back to obtain the next data value. This construct enters at the top and exits at the bottom; but it exits only when there are no more data values to be processed.

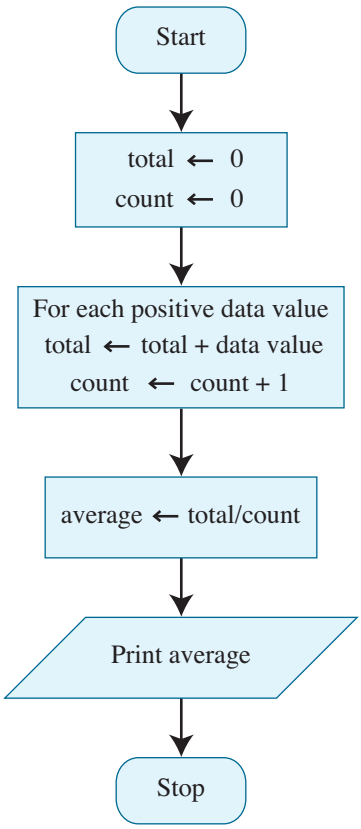


Figure 1.5 Flowchart to compute average

For each input data value > 0 the part of the pseudocode that is bracketed by `If` and `Endif` processes a single data value and can be drawn as shown in Figure 1.7.

Note that a positive value will be added to the total and the count will be incremented. However, nothing will be done if a data value is not positive. Put these `For...Endfor` and `If...Endif` constructs together with the preceding sequential flowchart and we obtain the flowchart of Figure 1.8.

Pseudocode *Pseudocode* is a semiformal description of the steps to be carried out by the computer, including steps that are to be repeated and decisions that are to be made. There is more than one way of writing an algorithm in pseudocode.

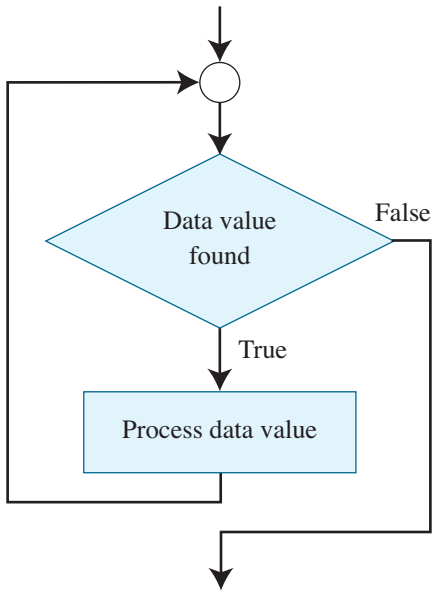


Figure 1.6 Flowchart for repetition

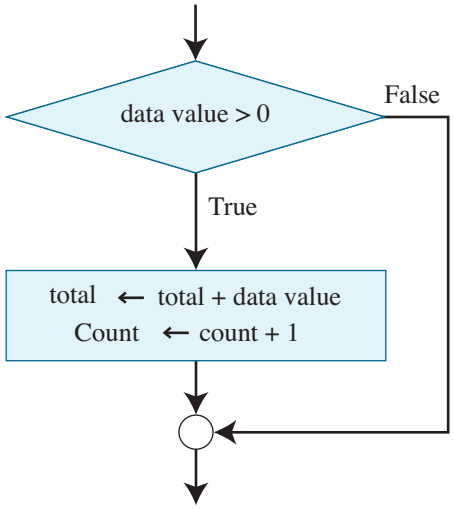


Figure 1.7 Flowchart for looping with counter

Another more detailed and mathematical way of describing this algorithm in pseudocode would be:

```
total ← 0.0
count ← 0
For each input data value found
   If data value > 0      (precondition: a data value was found)
      total ← total + data value
      count ← count +1
   Endlf
Endfor
average ← total / count   (precondition: positive data value was found)
print average
Stop
```

Note that several assumptions have been made: that there is at least one positive measurement, that the measurement is a real number rather than an integer (total is initialized to 0.0 rather than 0), and that a measurement of 0 doesn't count.

The processing of a data value depends on the value being positive, which in turn depends on an input value being found. This is shown by the two levels of indentation in the pseudocode.

Convention for Writing Pseudocode

1. Summations and counters must be initialized to zero, and other variables that require initial values must also be initialized.
2. Assignment of values is shown by an arrow pointing to the left.
3. The beginning and the end of any selected calculation is indicated as well as the basis for including or excluding it.
4. The beginning and the end of any repetition is indicated, as well as the basis for continuing and/or stopping the repetition.
5. Conditional steps and repetition steps are indented.
6. Either words or arithmetic operations may be used for arithmetic.

7. The algorithm termination must be clearly indicated.

These conventions are demonstrated in the previous examples of pseudocode.

Flowchart A *flowchart* uses standard symbols to show different operations and the order of execution of the steps of the algorithm. Lines and arrows are used to show the flow of control. Figure 1.8 shows a flowchart for the problem of finding

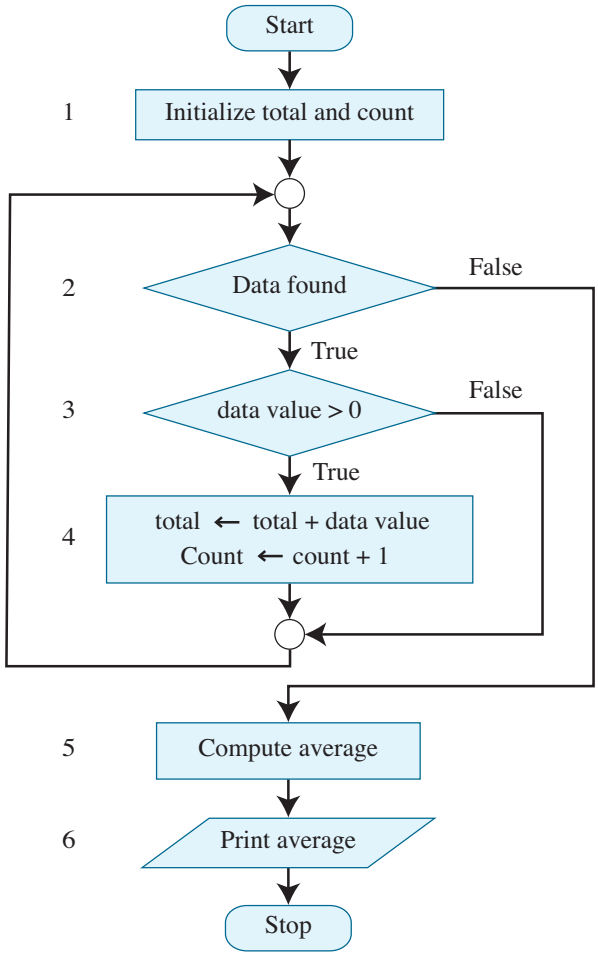
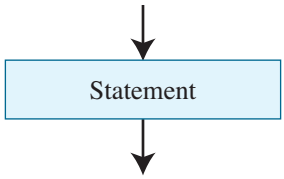


Figure 1.8 Flowchart to compute average depth

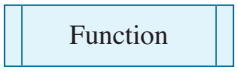
the average depth of water in a lake. Flowcharts help the programmer understand the flow of control in complex algorithms.

Standard Flowchart Symbols Each flowchart symbol represents part of an algorithm or program. When flowcharts are used in a structured way, symbols are grouped together to form a single-entry, single-exit structure.

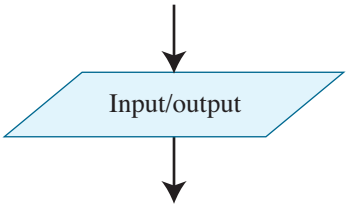
Process box: This is a rectangle with one control line leading into it and one leading out of it. At the lowest level it represents a single instruction that computes, moves data from one place to another place in memory, or carries out some other type of data manipulation. At a higher level it represents a sequence of instructions, which jointly implement a step in the program.



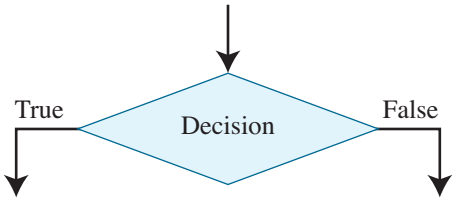
A function box: A process box that represents a complicated function that will be implemented separately is drawn as follows.



Input/output box: This is a parallelogram symbol with one control line leading into it and one leading out of it. At the lowest level it represents a single input/output instruction. At a higher level it represents a sequence of input/output instructions, which jointly implement an input/output step in the program.

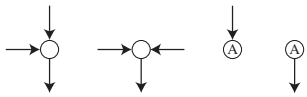


Decision box: A diamond-shaped symbol is used for the comparison of quantities for equality. The comparison generates a true or a false answer, which is the basis for a decision.



Execution follows either the path on the "true" branch or the path on the "false" branch, but not both paths.

Connector: A small circle is used as a connector symbol when two flow lines are to be coming together. Two lines are drawn into the circle and a single line is drawn out of it. A connector is also used when a flowchart is too large to fit on a single page or too complicated for all the lines to be drawn completely. In this case, the circle has only a single line in or a single line out and is labeled to show that parts of the flowchart are connected.



Flow lines: These lines are used to connect process boxes and decision symbols in the order of the logic and control flow of the program. An arrowhead is used to indicate the direction of the logic flow.

A flowchart that shows a step indicating input of data values must follow it with a check to determine whether the input attempt was successful. If the values are to be processed one at a time, they must be input one at a time. The flowchart loop indicates the processing of individual data values only when there are values. Note that the computer cannot identify the last data value. It can only detect the presence or absence of a data value.

Sample Test Data To test this analysis, we select a set of values, for example 23, 15, and 12 and work through the steps of the algorithm of Figure 1.8.

STEP	1	2	3	2	3	2	3	2	4	5
Count	0	0	1	1	2	2	3	3	3	3
Total	0	0	23	23	38	38	50	50	50	50
Data	-	23	23	15	15	12	12	-	-	-
Average	-	-	-	-	-	-	-	-	16.67	
Output	-	-	-	-	-	-	-	-	-	16.67

The algorithm is correct with this ordinary data. However, using extreme data may uncover errors—for example, if there are no positive data values.

STEP	1	2	4	5
Total	0	0	0	
Count	0	0	0	
Data	-	-	-	
Average	-	-	0/0	
Output	-	-	-	

This is a logical and computational error. This situation could be detected by checking the value of the count before computing the average, as shown in Figure 1.9.

At this stage of the design process, working with both words and a variety of diagrams provides more insight into the solution than either method would by itself. Errors found when working with the diagrams can be used to correct the specifications, which in turn leads to changes in the diagrams. This feedback and correction process should be repeated until the designer is satisfied that the solution is feasible with available computer resources. Any errors not found during the design phase can be very expensive to correct later on.

Structure Chart *Structure charts* were introduced in Section 1.2.2 as a way to show modularity. Since each step in an algorithm is a simple module, they can be used to represent an algorithm. The steps in Figure 1.10 are numbered for easy reference. The structure chart in Figure 1.10 corresponds to the levels of the

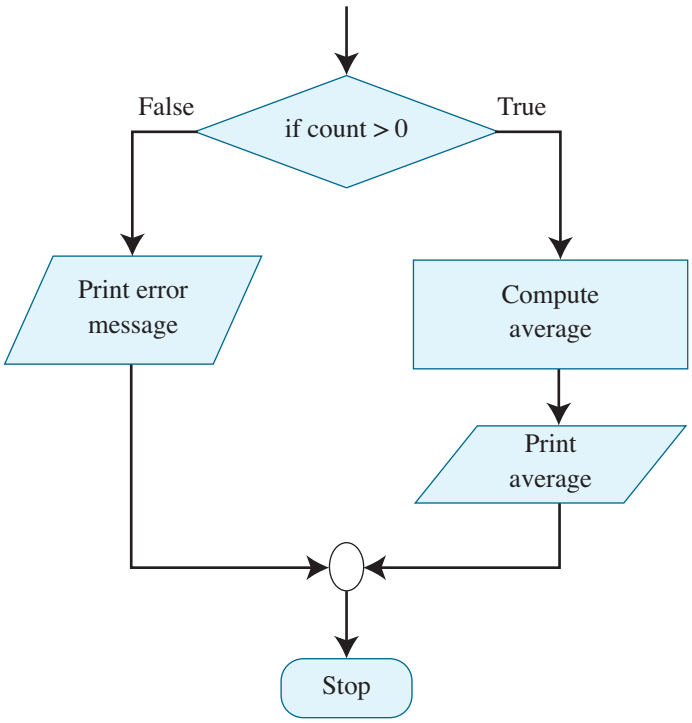


Figure 1.9 Branching structure

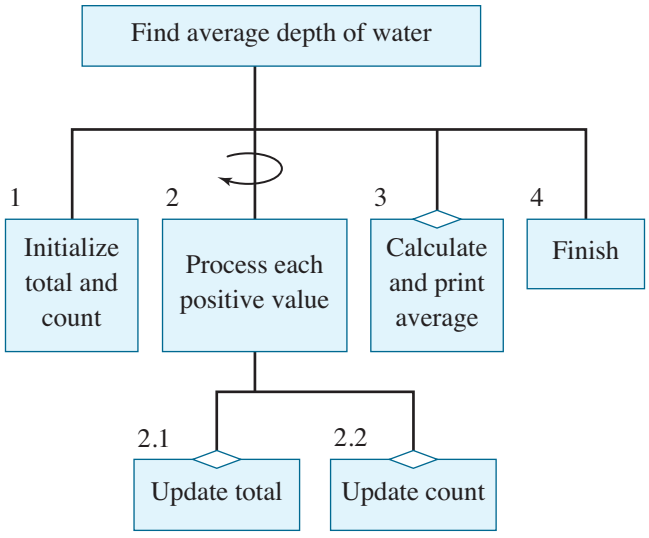


Figure 1.10 Structure chart to compute average depth

pseudocode. At the highest level of the chart is the statement of the problem solution. The next level contains the boxes that correspond to the major steps in the solution. Below these are their substeps, down to as many levels as are necessary. The bottom level of the structure chart describes the processing of a single data value. The circular arrow indicates the repetition part of the structure chart. The diamond shaped symbol indicates a conditional box.

Design Step: Draw the structure chart, draw the flowchart, and write the pseudocode.

The final part of the design process involves writing any necessary control statements in the system command language of the computer. In general, these will be statements to direct data to and from programs; to control the compilation, linkage (collecting routines), and execution of programs; and to save output from the compilation, linkage, and execution steps in appropriate files for later use.

1.3.5 Concept of Structured Programming

The development of algorithms in top-down methodology should use only the three basic control structures. A program is a sequence of steps; each step is an instruction or a group of instructions to be executed by the computer. Some of the instructions are executed only once; others are executed selectively; others are executed repeatedly. The three basic language structures are the sequence structure, selection structure, and repetition structure.

Sequence Structure A *sequence structure* is one in which execution control flows from one step to the next in sequence, without skipping any of the algorithmic steps, executing each step exactly once. This is shown in the diagram in Figure 1.11. An example of this structure written in C is:

```
stmt 1          x = 18.25;
stmt 2          y = 8.75;
stmt 3          z = x + y;
```

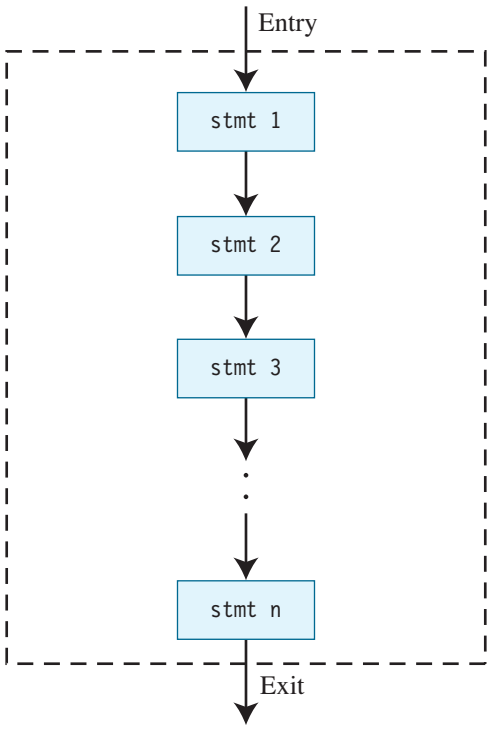


Figure 1.11 Sequence structure

Selection Structure The *selection structure* gives the computer a choice of executing one of a set of statements. Usually there are two alternative sequences, but in some cases only one sequence, and in some cases many. Exactly one alternative must be chosen, and executed only one time. C has special constructs for the selection structure. The diagrams for selection structure with one sequence and two alternative sequences are shown in Figure 1.12 and Figure 1.13 respectively. Notice that within the selection structure a sequence structure is embedded in one or more branches with the entries at the top and the exit at the bottom. The following examples of these structures show the general forms in pseudocode and a C equivalent.

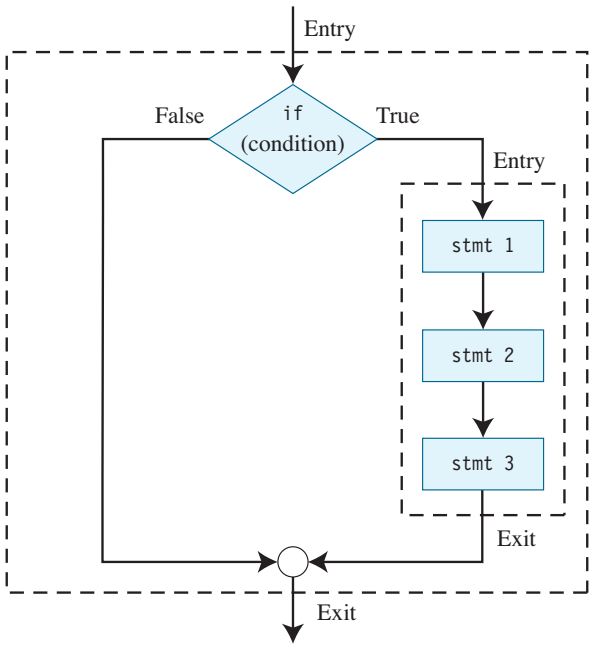


Figure 1.12 Selection structure with one alternative

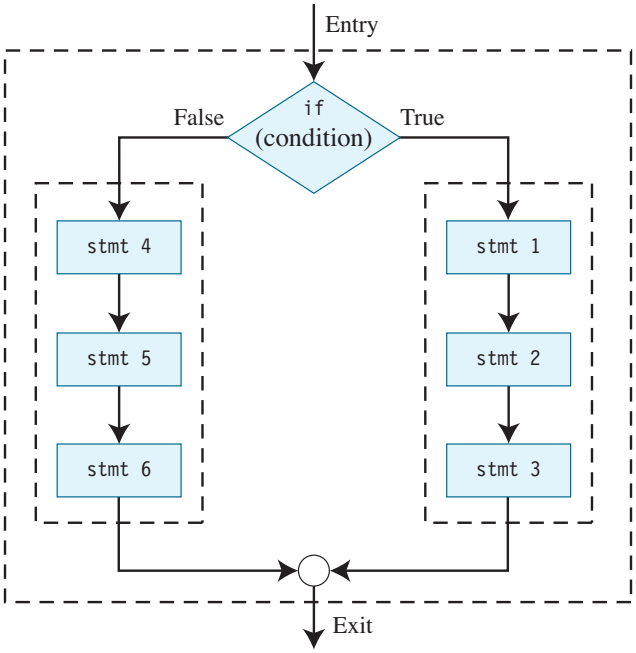


Figure 1.13 Selection structure with two alternatives

The pseudocode corresponding to each flowchart and an example of C code are as follows:

```
if condition                if ( x > y )
   stmt 1                   {
   stmt 2                         x = -x;
   stmt 3                         w = x *5 - y / 2.0;
end if                            p = 20 * x * w;
                           }

if (condition)              if ( x >= 0 )
   stmt 1                   {
   stmt 2                         z = 5 * x;
   stmt 3                         w = y + z;
otherwise                         p = n * m;
   stmt 4                   }
   stmt 5                   else
   stmt 6                   {
end if                            n = m * 5 - k * x;
                                  p = n - 5 * r - s / t;
                                  c = n + p;
                           }
```

Notice that in the second example there are two sequence structures: One embedded in the left branch and one in the right branch with an entry at the top and an exit at the bottom of each. The embedded sequences are contained between braces ("{" and "}") in the C code.

Selection structures may be stacked and nested depending on the logic of the algorithm. Symbols such as >=, =, *, +, /, −, etc., used in the C code will be explained later. However, their meaning should be obvious. Stacked and nested control structures will be discussed in detail in Chapter 4.

Repetition or Looping Structure The *repetition structure* contains one or more instructions that must be executed many times. A diagram for one form of this structure is shown in Figure 1.14. A pseudocode equivalent and C example follow. C has special constructs for the repetition structures.

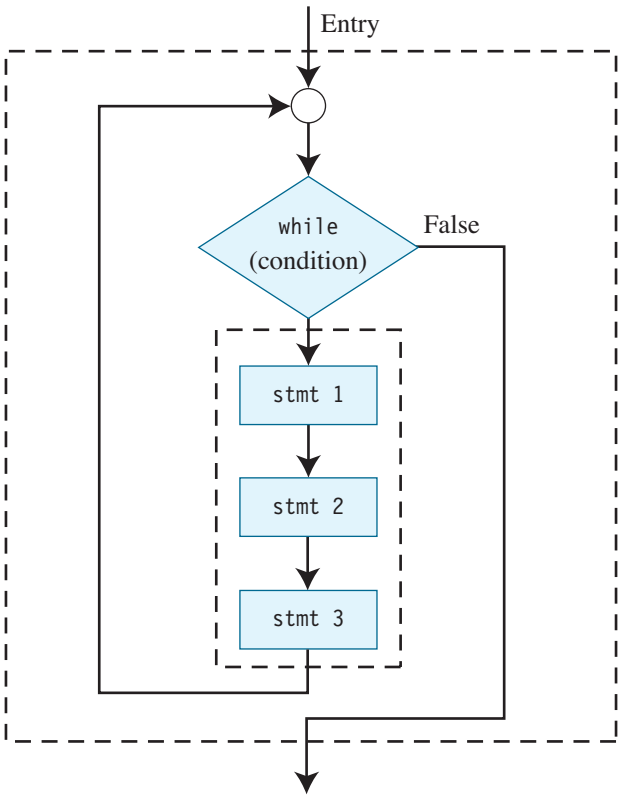


Figure 1.14 Repetition structure

The *while* and *do while* are the standard explicit repetition control structures in C. The basic difference between them is that the `while` construct begins by testing the condition as shown in Figure 1.14. The `do while` construct executes the subordinate sequence first and then tests the condition.

```
For condition          while( count < 10)
  stmt 1               {
  stmt 2                  scanf("%d", &num);
  stmt 3                  sum =  sum + num;
End for                   count  = count + 1;
                       }
```

The detailed implementation of the input statement `scanf("%d", &num)` and the control structure `while(count < 10)` with their syntax and semantics will be explained in Chapters 3 and 4.

1.3.6 Review Questions

1. What is an algorithm?
2. What is a program?
3. Explain top-down design methodology.
4. Why is it important to validate algorithm design?
5. What does an assignment instruction do?
6. Name some different forms of computer output.
7. What are the fundamental types of instructions in any programming language?
8. What kind of data can computers process besides numbers?
9. Information supplied to an application program is called _____.
10. Information produced by an application program is called _____.
11. Data generated within a program that is not part of the output is called _____ data.
12. What is data validation?
13. What is pseudocode used for?
14. What is a hierarchy chart used for?
15. What is a flowchart used for?
16. Why is the format of the output data important?
17. Why are reliability and efficiency important?
18. What are control instructions used for?

1.4 Program Processing

Once the design is complete, the computer program can be written. This consists of *source code* (data specification and instructions for the computer to interpret) and *documentation* (comments that document the code and the whole

programming process as an aid to the programmer and other people who may need to read or alter the code). When the code appears to be error-free, it is submitted to the computer for compilation. The compiler not only converts the source code to object code, it checks for errors in grammar, spelling, and punctuation. Usually it is necessary to compile code several times, making corrections, before it compiles correctly. The resulting object code is linked to modules from the system library. Again there is a possibility of error if all of the modules are not linked properly. When a program finally links correctly, it is executed using input data and providing output data that then needs to be verified and validated. When consistently correct output is obtained, using all types of input data, the program is ready to be used. At that point the programmer finishes the documentation. The details of documentation standards in the C language will be discussed in Chapter 2.

1.4.1 Program Coding

Source code should be written and tested one module at a time, corresponding to the submodules of a complex hierarchy chart or flowchart. As it is written, comments are included in the source code, documenting the purpose of the module, the input and output variables, the formula used, and anything else that clarifies what is being written. When a module is finished, it should be carefully read and tested by hand, using representative data before being tested on the computer.

Implementation Step: Desk check the computer source code before compiling and executing it.

The C language includes input and output statements, arithmetic assignment statements, `if` structures, `while` structures, a statement to mark the end of the program—everything that has been included in the hierarchy chart, pseudocode, and flowchart examples.

Consider the simple example of finding the sum of the numbers from 1 through 100 by generating them and adding them. The pseudocode and the corresponding C source code would be as follows:

```
sum ← 0
number ← 1
For number ≤ 100
   sum ← sum + number
   number ← number + 1
Endfor
print sum
```

```
#include <stdio.h>
int main( )
{
    int sum, number;
    sum = 0;
    number = 1;
    while(number < 101)
    {
        sum = sum + number;
        number = number + 1;
    }
    printf("%d\n", sum);

    return 0;
}
```

The C source code instructions outside the outer braces are instructions to the compiler. The outer braces enclose the C language equivalent of the pseudocode. The inner braces correspond to the indentation in the pseudocode; the beginning and then the end of the instructions that are to be repeated. Note that all of the instructions are necessary, if the sum were not cleared in the beginning, it might give a wrong answer. If the number was not started at 1, the code would not meet the specifications of summing numbers from 1 to 100. The summation takes place by adding each new value of a number to the accumulated sum. Then the number is incremented to the next value. The repetition is controlled by checking whether the current value of the number is < 101. If that check were missing, or if the number was not incremented, the repetition would never end. Whenever an algorithm contains repetition, the termination condition must be reachable. The details of this code will be discussed in the next chapter. For now it is enough to recognize its relationship to the examples given in discussing the design method.

1.4.2 Program Compilation

The first step in testing and debugging a program on the computer is *compilation*. During this process, the computer checks whether the program has syntax, punctuation, or spelling errors, that is, that it follows correct grammatical rules for C. If it appears to be all right, it is translated into machine code.

Note that the compilation process does not detect incorrect spelling of the names of library modules or other separately compiled modules. It also does not detect errors in the description of input and output data. And it does not detect errors in logic that lead to wrong answers, nonterminating repetitions, or impossible situations such as division by zero. Any mistakes in defining the lower-level modules of a complicated problem will not show up until later.

When the compiler detects an error, it identifies the location of the error and indicates what is wrong. It is up to the programmer to go back and fix the source code before compiling it again. Of course if any program logic is changed, it may be necessary to desk check the source code once more, using the sample data.

When the program compiles correctly, it produces object code and a list of the other modules needed. These are either stored for later use, or submitted to the linker in the program execution phase of the process. A program that compiles correctly does not need to be compiled again.

1.4.3 Program Execution

After the main program and any submodules have been written and compiled correctly, the resulting object modules are linked along with any mathematical or other library routines needed, and a single executable module is built. The process of building an executable module, or linking, is shown in Figure 1.15. The linkers and loaders of the operating system are responsible for building the executable module and executing it. If these system routines detect errors, corrections must be made to the appropriate source code module, any changed modules are recompiled, and the linking is repeated.

When the program is finally executed, the computer carries out the programmer's instructions in logical sequence, accepts the designated input, performs the required calculations, and produces output. The output includes messages indicating whether the execution contained errors and statistics about the computer resources used. When there are execution errors, the source code must be corrected and the program recompiled, re-linked, and re-executed. The programming

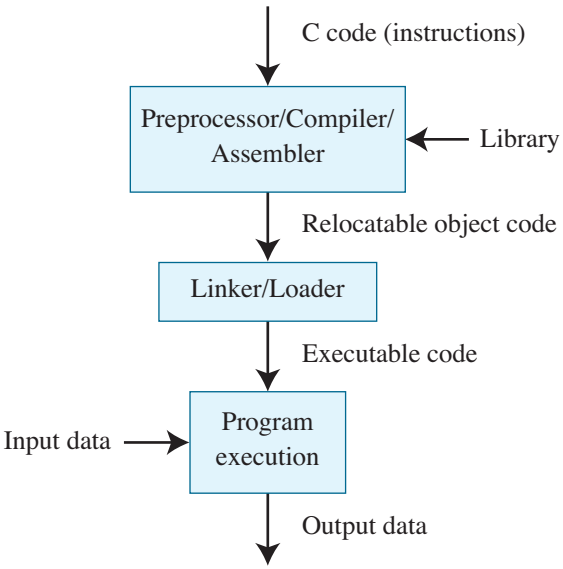


Figure 1.15 Program compilation and execution

is finished only when the program terminates normally, producing correct answers without any execution errors.

Program Warning: Computers will not allow the linking and execution of programs with compilation errors.

1.4.4 Program Testing

Program development is almost never error-free. However, it would be better to develop a correct program than attempt to find errors in an already completed program by repeated recompilation, linking, and execution. Errors may be of many types, such as data specification, problem specification, program design errors, implementation errors, typing errors, logic errors, and data input errors. Errors are detected at different stages of the programming process. The errors in a program are called *bugs*, and the process of detecting and removing them is called *debugging*. It is desirable to detect any error as early as possible in the

programming process to minimize the changes and avoid extra implementation cost and effort.

Logic errors are the hardest to detect, since programs may produce incorrect answers. To check the accuracy of the logic, it is necessary to know in advance what the answers should be for certain sets of test data. Before a program can be used to produce new answers, it must be run with typical data for which the answers are known. It must also be run with extreme data that has known answers. If the computer does not produce the correct answers, there is probably something wrong with the specifications or with the design of the program. A correct program not only produces correct answers for valid data, it diagnoses invalid data and processes it appropriately.

Programming Hint: Test the program for correctness using specially designed sets of data that test all the statements of the program.

Logic errors may be caused by careless copying of numbers, formulas, or data formats. Because it is very hard to find typographical errors, sometimes it is helpful to have another programmer look at the code. Explaining the code to someone else is also a good way of finding logic errors.

Some programming languages have built-in tools for testing, and some installations have system and software tools for testing for correctness. Use these aids whenever they are available. When they are not, build programs using extra output statements so that you can see what is happening at intermediate stages of processing.

Testing is a process intentionally designed to find errors in programs. This must be done systematically. Each module must be separately tested for errors and when the modules are put together, the results must be tested further. Any time a change is made, there is the possibility of introducing more errors. Not only must a program give correct answers to correct input, it should be able to detect incorrect input. It should also avoid giving incorrect answers for accurate but unusual input.

Testing procedures for large, complex systems are extensive and detailed. The test cases must be carefully designed to check the entire system. In general, the people testing the system should not be the same ones who design and implement it.

Programming Hint: Echo print input data until the computed results are accurate.

1.4.5 Program Documentation

Documentation, an important part of software development, should be carried out simultaneously with design. Documentation is used to keep track of the design process and to keep track of implementation and testing. It becomes part of the final system, to be used by the programmers who will maintain or modify the system. There are two types of documentation, system documentation and program documentation. *System documentation* includes functional descriptions, introductory manuals, reference manuals, installation manuals, user manuals, and so forth. *Program documentation* includes all phases of program development documents. It should include:

- Statement of the problem
- Glossary of input/output variables
- Description of each module of the program
- Error messages produced by the program and their causes
- Security measures to be incorporated in the program to protect the programs and data
- Test data to be used in program modification

Some of the program documentation is part of the source code of the program.

Programming Rule: Program code must be documented.

The total documentation package should include everything anyone who uses the system needs to know about the system. It should be organized according to the needs of the various people who deal with the system.

Once a programming project has been completed and proven useful, it takes on a life of its own, outliving the immediate need, the programming team, and the hardware. Therefore, modifications become inevitable.

Documentation may need to be changed any time changes are made to the program. Just as there are various versions of the source code on the computer, so there will be various versions of the documentation.

Programming Objective: Write easy-to-maintain source code and easy-to-understand documentation.

During the life of a production program, further modifications will be needed to meet changing situations and to correct previously undetected errors. All useful programs can be expected to evolve to meet changing circumstances.

1.4.6 Review Questions

1. What does a compiler do?
2. What does a linker do?
3. The language in which the program is written is called _____ code.
4. The compilation process produces _____ code.
5. What does a loader do?
6. List the types of errors according to developmental stages.
7. Give two reasons why documentation is important.
8. The process of locating and correcting errors is called _____.
9. Why is it important to test a program with data that intentionally contains errors?

1.5 Program Processing Environment

There are two ways to look at a processing environment: as a computer system environment or as a programming environment. The system environment in which the program runs is characterized as single-user or single-job, time-sharing, multiprogramming, or multiprocessing. The programming environment is batch, interactive, or real-time.

1.5.1 Computer System Environment

A *single-job environment* is one in which only one program at a time can be loaded into the computer for execution. All the system resources in such an environment, such as disk, memory, and processor, are allocated to a single job. Once the job is completed, all the system resources are released.

Most mainframe computers and minicomputers and some microcomputers support a time-sharing environment. A *time-sharing environment* is one in which

several users can have access to the computer at the same time, running different programs. Multiprogramming, multiprocessing, and parallel processing are all different forms of time-sharing environments.

In a *multiprogramming environment*, several executable programs can exist in memory at the same time, but only one program is active at any given instant. The programs that are loaded for execution will take turns using the time and resources available within the time limit allocated to each. In a *multiprocessing environment* there is more than one CPU, making it possible to execute several programs simultaneously. Alternately, several processors may be working on the same program at the same time. In a *parallel processing* environment, the program is partitioned into blocks that can be executed in parallel by different processors.

1.5.2 Programming Environment

Programming environments are designed for different user needs. In *batch processing*, programs are executed when it is convenient for the computer installation. Usually large amounts of data are involved and the actual time of execution is not critical. In *interactive processing*, programs are executed while the user waits online for the output. *Real-time processing* is used to directly control equipment.

Batch Processing In multiprogramming and multiprocessing environments, the batch processing mode is commonly used for program testing and for numeric applications. In *batch processing*, the operating system takes control of the program. The computer schedules and controls program execution. Several jobs may be entered through terminals, or loaded from disk files, and left to be executed when sufficient time is available.

Jobs submitted in a batch environment are stored on the disk (*spooled*) and scheduled by the operating system according to the priority of the job and the resources it needs. The output is spooled to the disk print file so that it can be printed once the program has terminated and the printer is available.

Batch processing is used when there are large amounts of data to process, or when time is not critical. The output usually consists of reports. Batch processing cannot be used when the user must interact with the program, processing transactions, correcting drawings, or directing choices. Batch processing is used, however, for updating an online transaction system when the transactions can be collected and processed at a single time, for example, at the end of the day.

Interactive Processing In *interactive processing*, the user is in communication with the computer system. Data values are entered through terminals and the user expects an immediate response from the system. Output design may differ from that sent to a printer. Interactive processing is primarily used for transaction processing, changing permanently stored data, and retrieving information.

Several interactive terminals can be connected to a large computer system, each one having access to the computer hardware and software resources. If there are several users at the same time, the CPU will share its time with all of them. There are various strategies for allocating time to a user. The operating system attempts to keep all the hardware operating as near capacity as possible without causing any user to wait very long. C is designed for both batch processing and interactive processing applications.

Real-time Processing In *real-time processing*, when a computer is used to directly control equipment, computer response must be as fast as data collection. Embedded computer systems are real-time systems. Examples include a computer onboard an aircraft that controls the autopilot, or computers that control nuclear reactor cooling systems, space shuttles, pacemakers, power fluctuations, and automobile ignition systems. Real-time systems are online systems that must respond immediately to changing needs. They have critical time constraints. Such systems are dedicated to single applications. C may be used for real-time applications in special hardware and software environments.

1.5.3 Review Questions

1. What is meant by time-sharing?
2. What is a batch-processing environment?
3. What is multiprogramming?
4. A multiprocessing system will have more than one what?
5. Why is real-time processing important?
6. Is interactive processing the same as real-time processing? Explain.
7. Why is programming for interactive processing different from programming for batch processing?

1.6 Samples of Algorithms

The following examples show the design and development of algorithmic procedures. These examples are chosen from engineering, science, and mathematics. Each step of the algorithm must perform a single function. Together, the steps must arrive at the desired solution. The algorithm must end. Each algorithm must have a single entry point and a single exit.

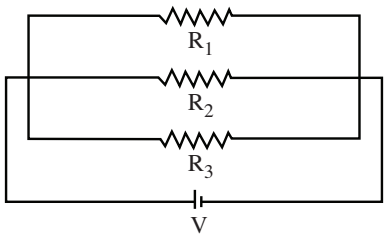
The most common complications in learning to design an algorithm are the need to detect data which would cause problems in a formula, the need to anticipate that data may be missing, the need to count data, and the accumulation of totals. All of these except totals and counting of totals are shown in the following examples. The use of totals and counting have already been introduced in this chapter.

1.6.1 Resistance and Voltage of a Parallel Circuit

This example includes data validation.

Problem Compute the effective resistance and voltage of an electrical circuit containing three resistances connected in parallel, with the current and resistance as input data.

Data Three positive values for resistances, and a positive value for current (ohms).



Method The formula for the resistance R of a parallel circuit with resistances R_1, R_2, and R_3 and the voltage V assuming a current of I is as follows:

$$1/R = 1/R_1 + 1/R_2 + 1/R_3$$

Solved for the effective resistance R, this is

$$R = 1/(1/R_1 + 1/R_2 + 1/R_3)$$

The voltage is computed from the following formula:

$$V = I \times R$$

Algorithm in Pseudocode

Input R_1, R_2, R_3, and I the resistances and current respectively
Check that they are positive
If so, compute the effective resistance:
  $R \leftarrow 1/(1/R_1 + 1/R_2 + 1/R_3)$
Compute the voltage using
  $V \leftarrow I \times R$
Output R and V, labeled
Stop

The algorithm is shown as a structure chart in Figure 1.16. The algorithm is also shown as a flowchart in Figure 1.17. The structure chart and the flowchart show the input of the resistances and current, and the output of computed effective resistance and the computed voltage.

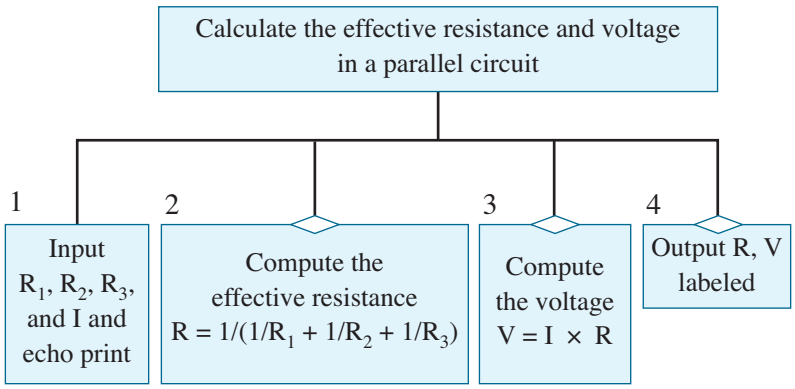


Figure 1.16 Structure chart to compute the voltage of a parallel circuit

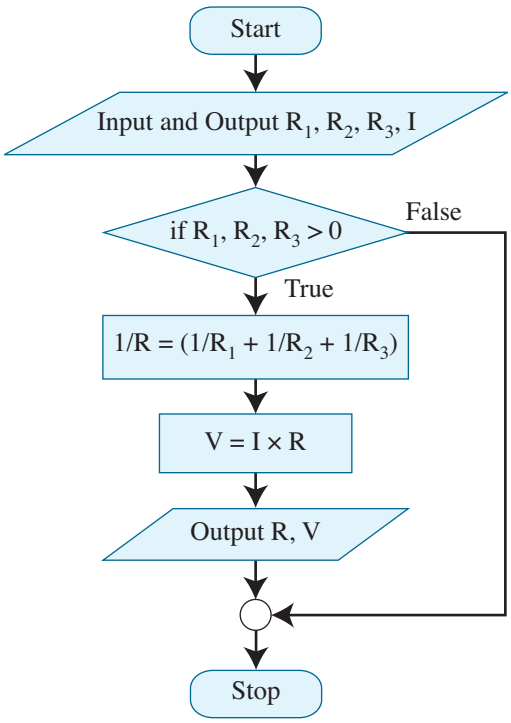


Figure 1.17 Flowchart to compute the voltage of a parallel circuit

1.6.2 Volume of a Sphere

The second part of this example introduces the use of a counter to control data input.

Problem Write an algorithm to compute the volume of a sphere of diameter *d*. Output the diameter and the volume.

Method The volume of a sphere of diameter *d* is given by the formula:

$$r = d / 2$$
$$\text{volume} = 4/3\ \pi\ r^3$$

Data A positive real number is representing the diameter in inches.

The algorithm is shown as a structure chart in Figure 1.18. The algorithm is shown as a flowchart in Figure 1.19. The input is the diameter of the sphere and the output is the computed volume. The algorithm shows the calculation of volume for one sphere.

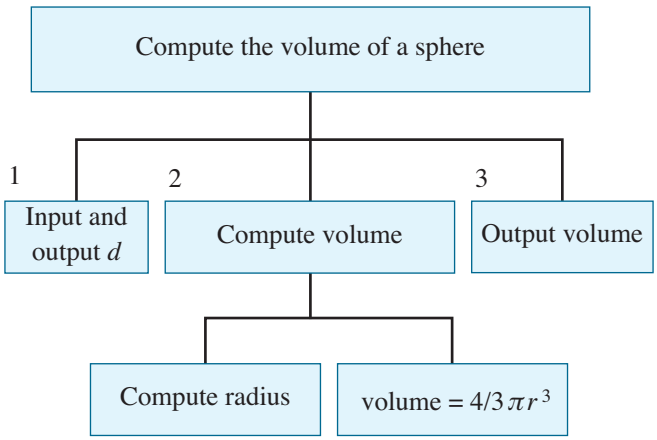


Figure 1.18 Structure chart to compute the volume of a sphere

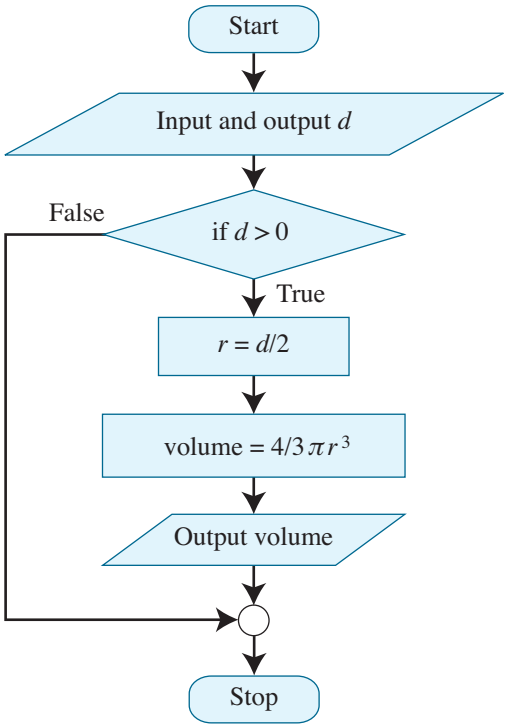


Figure 1.19 Flowchart to compute the volume of a sphere

Algorithm in Pseudocode

```
Input and output diameter d
If d > 0
    Compute the radius
      r ← d / 2
    Compute the volume
      Volume ← 4/3 π r^3
    Output the volume
End if
Stop
```

The preceding algorithm computes the volume for one sphere of diameter d. The following algorithm shows the computation of the volumes of several spheres with different diameters.

If the volume is to be calculated for 10 spheres, the algorithm is as shown in the structure chart of Figure 1.20 and the flowchart of Figure 1.21.

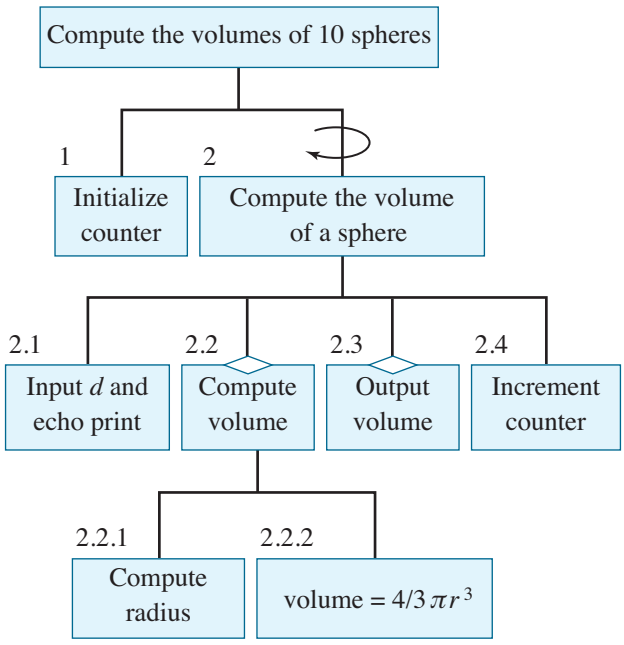


Figure 1.20 Structure chart to compute the volume of 10 spheres of different diameter

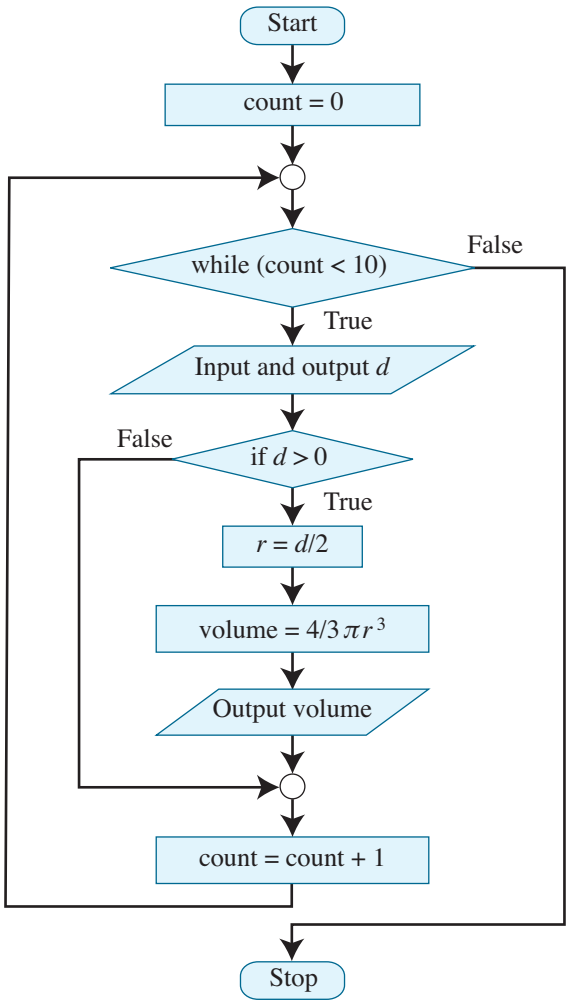


Figure 1.21 Flowchart to compute the volume of 10 spheres for each diameter

Algorithm in Pseudocode

```
Count = 0
For each of 10 spheres
   Input and output the diameter d
```

```
    If d > 0
      Compute the radius
        r ← d / 2
      Compute the volume
        Volume ← 4/3 π r^3
        Output the volume
    End if
    Count = Count + 1
End for
Stop
```

The curved arrow in the structure chart shows that the calculation of volume is repeated for different input diameters of spheres for spheres of different input diameters.

This algorithm assumes that there are 10 data values. If there are fewer than 10 data values, an error check must be incorporated in the algorithm to output an error message stating that there are fewer than 10 data values. The error message must be in the exit from the while loop. Then the algorithm will be robust, meaning that it is fail-safe.

1.6.3 Square Root Approximation

This example introduces the use of an iterative formula in a repetition structure controlled by a predetermined limit.

Problem Write an algorithm to compute the square root of x by using the Newton–Raphson formula:

$$s_{k+1} = (s_k + x/s_k)/2 \qquad k \geq 1$$
$$s_1 = x/2 \qquad x > 0$$

This formula is based on the fact that

$$\text{if } s_k = \sqrt{x}$$

then $s_k^2 = x$ and $s_{k+1} = (\sqrt{x} + x/\sqrt{x})/2 = (\sqrt{x}^2 + x)/(2\sqrt{x}) = \sqrt{x}$

Data A positive real value for x and a constant specifying the required accuracy.

Method The iterative solution of the problem assumes that given the value of x, the value s_1 is calculated. Then the formula for s_{k+1} is used repeatedly until sufficient accuracy has been obtained. For $k = 1$ $s_2 = (s_1 + x/s_1)/2$

$$k = 2 \quad s_3 = (s_2 + x/s_2)/2$$

$$\ldots\ldots\ldots$$

$$k = n \quad s_{n+1} = (s_n + x/s_n)/2$$

This process will continue until the difference between s_k and s_{k-1} is less than, or equal to, a predetermined value. When this happens, s_k is accepted as the square root of x. Note that the computer must be given explicit instructions about when to stop an iterative approximation, otherwise the calculations would continue indefinitely without ever providing an answer.

Algorithm in Pseudocode

```
Input the value of x > 0
Initialize limit
    limit ← 0.0001
Calculate s1
    s0 ← x
    s1 ← x / 2
    k ← 1
while( |sk – sk–1| > limit )
    sk+1 ← (sk + x /sk) / 2
    k ← k + 1
End while
```

Output x and s_k as the square root of x, labeled.

The algorithm is shown as a structure chart in Figure 1.22. The algorithm is shown as a flowchart in Figure 1.23. Later we will see that it is not necessary to have all the different s values available at the same time. It is possible to write this algorithm using only two names, for example s_k and s_{knext}.

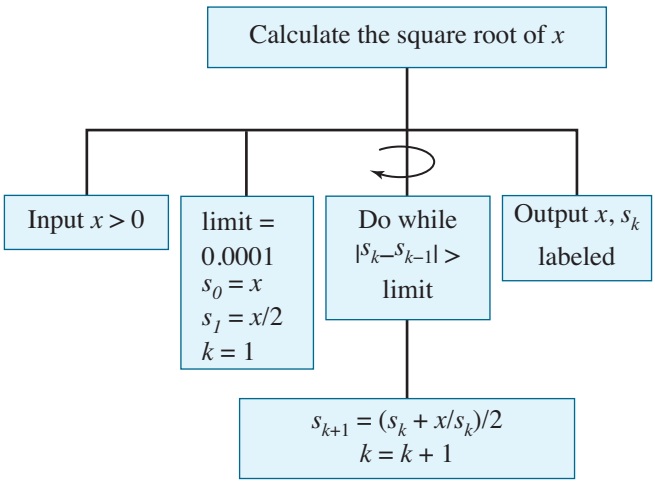


Figure 1.22 Structure chart to compute the square root of a number

Wherever possible an algorithm should be hand-checked before a computer program is written. Assume that

$$x = 25.0$$

Then

$$s_0 = 25.0$$
$$s_1 = 25.0/2 = 12.5$$
$$s_2 = (12.5 + 25.0/12.5)/2 = 7.25$$
$$s_3 = (7.25 + 25.0/7.25)/2 = 5.349\ldots$$
$$s_4 = (5.349 + 25.0/5.349)/2 = 5.0113\ldots$$

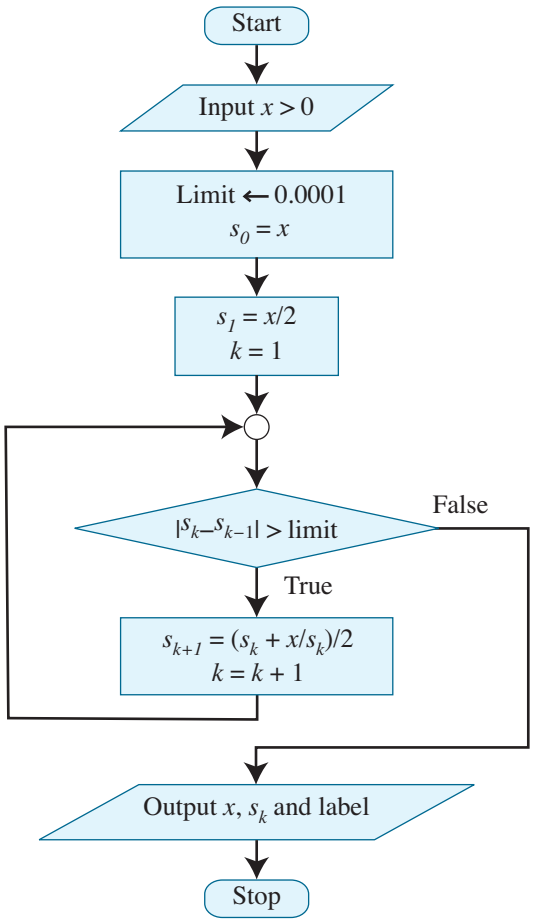


Figure 1.23 Flowchart to compute the square root of a number

The calculated value appears to be converging to 5.0, the correct square root. The computer iteration will stop when two successive values of s differ by less than a predetermined amount, in this case 0.0001.

1.6.4 Total Pressure of Gaseous Mixture

This example introduces a repetition construct controlled by an input value that gives the number of data values. The data values are validated, and if found to be invalid, an error message is printed and the process is terminated.

Problem Write an algorithm to compute the total pressure given the partial pressures of perfect gases by using Dalton's law of partial pressures of perfect gases. The law states that the pressure exerted by a mixture of perfect gases is the sum of the partial pressures of the gases.

Data Positive integers to specify the number of components of perfect gases in the mixture and positive real values to specify partial pressures of the perfect gas components in the mixture.

Method Given the partial pressures of the perfect gases A, B, C, …., N as p_A, p_B, p_C,…., p_N, the total pressure is computed as follows:

Total pressure = partial pressure of gas A + partial pressure of gas B + partial pressure of gas C + ⋯ + partial pressure of gas N

$$p = p_A + p_B + p_C + \cdots + p_N$$

The structure chart is shown in Figure 1.24 and the flowchart is shown in Figure 1.25.

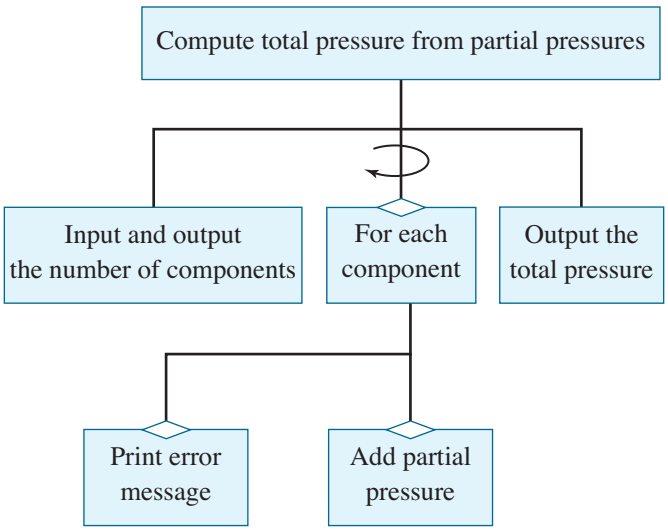


Figure 1.24 Structure chart to compute the total pressure from partial pressures

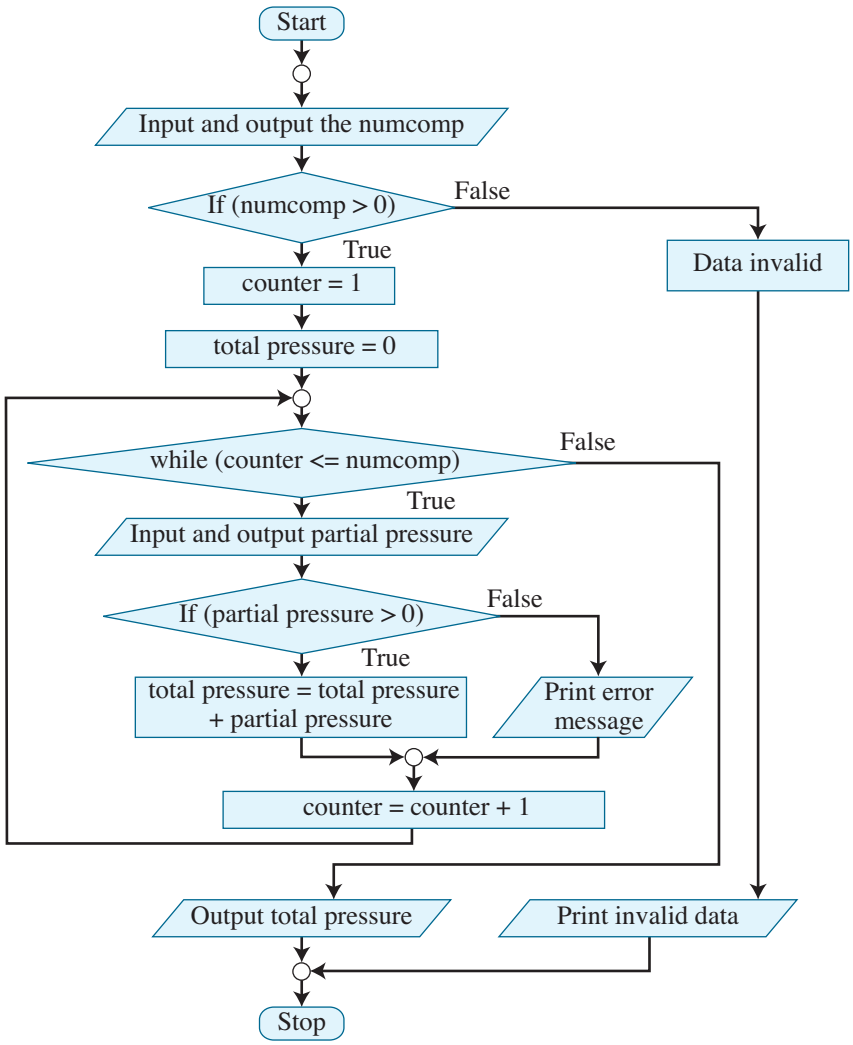


Figure 1.25 Flowchart to compute the total pressure from partial pressures

Algorithm in Pseudocode

```
Input the number of components in the gaseous mixture
Output the number of components
If number of components > 0
   counter = 1
   total pressure = 0
```

```
    For each component
        Input and output the partial pressure
        If partial pressure > 0
            total pressure = total pressure + partial pressure
        Else
            Print error message
        End if
            counter = counter + 1
    End for
        Output the total pressure
Else
        Output message "Invalid Data"
End if
Stop
```

1.6.5 Mass Flow Rate of Air Through Pipes

This example introduces a repetition construct controlled by the use of a known final value. Two variables range through a set of values.

Problem Write an algorithm to compute the mass flow rate of air in a pipe diameter $d = 5, 10, 15, 20, \ldots, 50$ inches at pressures of $p = 50, 60, 70, \ldots, 100$ psi and a temperature of T degrees. Barometric pressure is p_b psi and velocity is V ft/sec. Compute the number of pounds of air per second flowing through the pipe.

Data Input positive real numbers to specify the universal gas constant, the velocity of the air, and the temperature.

Method Given different diameters, different pressures, universal gas constant, temperature of air, and velocity of air, compute the density of air from the following formula:

$$\gamma = \frac{p}{rt}$$

Where: γ is the density of air,

p is the absolute pressure,

r is the universal gas constant, which is 53.3ft/R, and

t is the absolute temperature in degree Kelvin.

The mass flow rate is computed from the equation

$$w = \gamma q$$

Where: q is the flow rate in cubic feet per second and q is given by the equation

$$q = av$$

Where: a is the cross sectional area of the pipe in square feet, and

v is the velocity in ft/sec

Notice that the units must be converted.

The structure chart is shown in Figure 1.26 and the flowchart is shown in Figure 1.27.

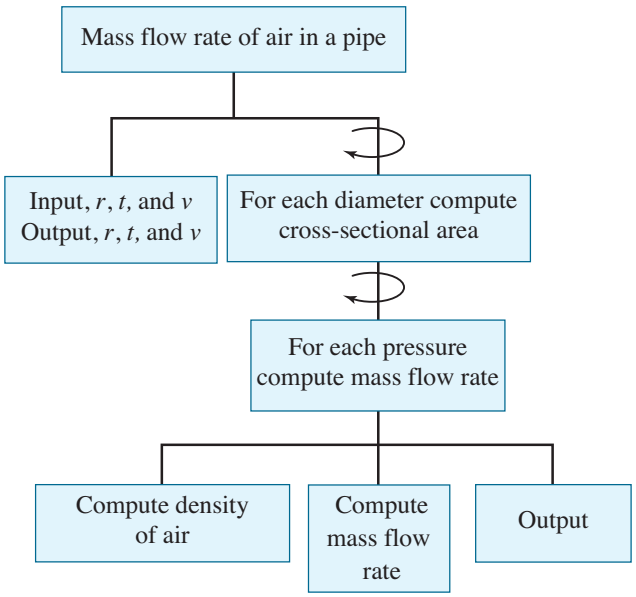


Figure 1.26 Structure chart to compute the mass flow rate of air in a pipe

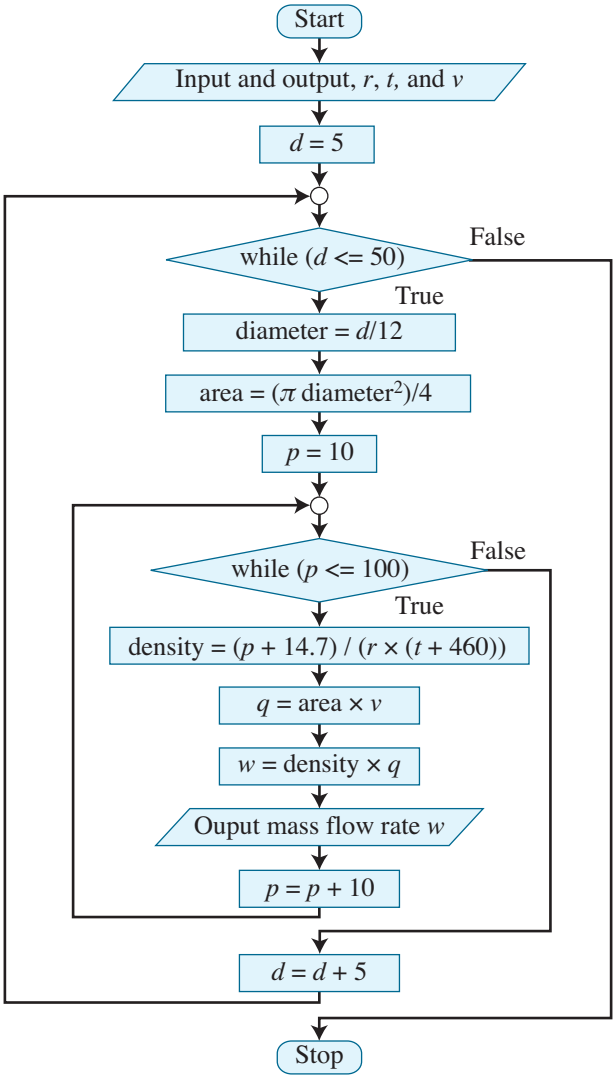


Figure 1.27 Flowchart to compute the mass flow rate of air in a pipe

Algorithm in Pseudocode

Input the universal gas constant R for air,
    temperature t and velocity v
For each diameter $5 \leq d \leq 50$ inches
  Calculate the cross sectional area of the pipe
  diameter = d / 12.0

$$\text{area} = \frac{\pi \ \text{diameter}^2}{4} \ \text{sq ft}$$

  For each pressure $50 \leq p \leq 100$ psi

$$\text{density of air} = \frac{(p + 14.7)}{r(t + 460)}$$

    volume flow rate = area × v cu.ft/sec
    mass flow rate = density of air × volume flow rate
    Output mass flow rate
  End for
End for

Chapter Summary

- Modern computers are used to solve complex problems within a reasonable amount of time. They are the backbone of modern internet technology. They handle large amounts of data of different types in various applications. Modern computers are used in e-commerce and in various telecommunication technologies for the transport of data and voice. Computers have become a part of the everyday tools of modern society.
- A computer system consists of both hardware and software. The hardware components are the memory, processor, and input/output devices (peripheral devices such as disk drives, tape drives, mouse, light-pulse magnetic character readers, optical character readers, graphic display devices, and plotters and printers). The input devices transmit the information to be processed into the computer. Output devices return the processed information from the computer system to the outside world. The processor performs the arithmetic and logic operations on the information stored in memory under the command of the control unit.

- The main software component is the operating system, which manages all the system resources and provides an environment for the user to communicate with the computer. Other software modules include the system utilities, file management software, assemblers, and compilers. The most frequently used application software modules include word processors, spreadsheets, database applications, and presentation software.
- Programs written to solve various types of problems in different areas of engineering and science are known as application software. These programs are written in high-level languages, compiled, debugged to eliminate errors, linked with library routines, and executed.
- Solving problems on a computer involves several steps. The first step is to describe the data and the problem. The next step is to divide the problem into subproblems. The subproblems are divided further into understandable and manageable units. This process, called modularization, is the major design step. It involves various design aids such as pseudocode, structure charts, and flowcharts. The other steps are implementation, testing, production use, and maintenance. The design step also includes design of data formats for input/output, an algorithm, test data, and testing procedure. Documentation is written at each step. These steps are repeated until a program is obtained that reliably produces correct results.
- An application program may run in a batch environment or an interactive environment. The environment may be single-job, time-sharing, multiprogramming, multiprocessing, or real-time.

Exercises

1. Write an algorithm to compute the volume of a cylinder with diameter d and height h and print the diameter, height, and the volume.

2. Write an algorithm to compute the volume of water in cubic feet, flowing through a pipe of diameter d in feet, with a velocity of v feet per second. The formula to compute the volume flow rate per second is given by:

   $r = d/2$
   area $= \pi \times r^2$, and
   volume = area $\times v$

The algorithm should print d, v, and volume, labeling the output.

3. Write an algorithm to compute the distance s fallen by an object in free fall. The formula is:

$$s = s_0 + v_0\, t + \tfrac{1}{2}\, a\, t^2$$

where s_0 is the initial position in feet, v_0 is the initial downward velocity in ft/sec, t is the time in seconds, and a is 32.2 ft/sec^2. The input values are s_0 and v_0. The output values are s and t where t = 0, 5, 10, 15, 20,…, 100.

4. Write an algorithm to compute the compression stress on a rectangular steel column of width w and depth d subject to a compression load of p tons for p = 10, 20, 30,…., 50. It should print the cross-sectional dimensions of the column, the cross-sectional area, the load, and the compression stress. Validate w and d as positive numbers.

area = $w \times d$, and
stress = p/area

5. Write an algorithm to compute the area of a triangle given the three sides of the triangle as a, b, and c. Use the following formula to compute the area.

$$s = \tfrac{1}{2}\,(a + b + c)$$
$$\text{area} = \sqrt{s\cdot(s-a)\cdot(s-b)\cdot(s-c)}$$

The algorithm should check that a, b, and c are each < s, and should print the sides and the area of the triangle, labeled.

6. Write an algorithm to compute the distance covered in 25 min, 50 min, 75 min, 100 min, 125 min, and 150 min, if a car is traveling at a speed of 80 miles per hour. Stop at 500 miles. It should print the time t and the distance traveled for each value of t.

7. Write an algorithm to generate the sum of the Fibonacci numbers between 1 and 100. Print the numbers and their sum. The formula for computing Fibonacci numbers is:

$\text{fib}_n = \text{fib}_{n-1} + \text{fib}_{n-2}$
$\text{fib}_1 = \text{fib}_2 = 1$

8. Write an algorithm to input a set of integer numbers, count and sum the positive numbers, and also count and sum the negative numbers. It should then print the count and sum of all positive numbers and the count and sum of all negative numbers.

9. Write an algorithm to compute the minimum diameter needed for a pipe to carry 2.22 N/s of air with minimum velocity of v m/s? The air is at t°C and under the absolute pressure of p. Given the following formulas

$$\gamma_{\text{air}} = \frac{p}{rt},$$

$w = 2.22$ N/s $= \gamma q$, therefore, $q = 2.22/\gamma$

Where w is weight flow rate
γ is specific weight for air

minimum area $a = \dfrac{q}{\text{minimum velocity}}$, and

minimum diameter $= \sqrt{(4 \times \text{area}) / \pi}$

10. Write an algorithm to compute the kinetic energy of different disks of mass $m = 10$ to 100 kg in increments of 10 and the radius $r = 10$ to 50 cm in increments of 10. The disks rotate at a speed of 500 rpm. The kinetic energy is computed from the formula:

$$k = \frac{i\omega^2}{2}$$

where i is the moment of inertia of a uniform disk given by the formula:

$$i = \frac{mr^2}{2}.$$

Where r is the radius of the disk in meters
m is the mass of the disk in kg

The angular velocity is computed as follows:

$$\omega = \frac{500 \text{ rev}}{60 \text{ sec}} \times \frac{2\pi}{1 \text{ rev}}.$$

The algorithm must output the results in the form of a table as shown.

```
mass = 10
radius  10   kinetic energy = xx.xx joules
.....
radius  50   kinetic energy = xx.xx joules

mass = 20
radius  10   kinetic energy = xx.xx joules
.....
radius  50   kinetic energy = xx.xx joules

mass = 100
radius  10   kinetic energy = xx.xx joules
.....
radius  50   kinetic energy = xx.xx joules
```