Eastern Economy Edition

Computer Programming in

V. RAJARAMAN

PHI Learning

PHI

COMPUTER PROGRAMMING IN C

V. RAJARAMAN
Honorary Professor
Supercomputer Education & Research Centre
Indian Institute of Science Bangalore 560012

PHI Learning Private Limited
New Delhi-110001
2010

Contents

Preface

This book introduces computer programming to a beginner using the programming language C. The version of C used is the one standardised by the American National Standards Institute (ANSI).

When C was introduced in mid 1970 as an efficient relatively "low level" programming language it was not widely used. However, with the advent of UNIX operating system and the quest for an efficient high level programming language, C rapidly gained users. In 1983 ANSI standardised C and C rapidly gained popularity. It is now the preferred language among professionals for writing efficient programs.

This book has been written assuming that the reader will have no prior knowledge of computers or programming. The book begins with an introduction to algorithms and computers. It is followed by a chapter on the methodology of evolving algorithms. A number of algorithms are developed in this chapter from first principles. A simplified model of the computer is introduced and used to illustrate how algorithms are executed by computers. The next chapter introduces the concept of high level languages and their translation. We start discussion of C language by encouraging a student to read and understand C programs. This methodology is used to motivate the student and encourage him/her to start writing programs. Formal discussion of rules of syntax of the language follow. Rules for writing variable names, arithmetic expressions, statements and input/output functions are given. Two chapters, following this, present the branching control statements and loop control statements. A number of example programs are used to motivate and illustrate the use of these statements. The need to adopt a good *style* in writing programs is stressed and illustrated in all the example programs. A chapter on *array* data structure and its use and a chapter on logical operations on data follow this. At this juncture a student would be ready to write relatively large programs to do some significant jobs. A chapter is therefore written, at this stage, in which are illustrated three complete programs—one to simulate a small computer, another to process students' result data and a third one to analyse responses to a questionnaire. The concept of *functions* in C and their application in program development and modularisation is taken up next. The rest of book is devoted to a discussion of the use of C in solving non-numerical problems. This part of the book presents the rich data structures and their applications which are supported by C. The distinctive features of C and the power of the language to solve a variety of interesting problems is brought out in these chapters. This part of the book starts with a chapter on processing *character strings*, their representation, manipulation and applications. Following this is a chapter which illustrates the use of enumerated data types. A method of defining and using a data structure known as a *stack* is also presented in this chapter. The next chapter describes the data structure known as structures. A data type known as *pointer type* is available in C which distinguishes it from many other popular languages such as *Fortran* and *Cobol*. This data type and how it can be manipulated is described in the next chapter. The freedom allowed by C to manipulate pointers makes C very powerful. At the same time this facility can lead to subtle errors which are difficult to detect. The use of pointers in creating and manipulating

interesting data structures such as *lists* and *trees* is illustrated with examples. The next chapter is devoted primarily to a discussion of *recursion*. The concept of recursion is defined first. Problems in which it can be profitably used and where it should not be used are explained. The power and elegance of recursive formulation of algorithms is brought out with some examples. C provides instructions to manipulate bits in words stored in memory. This is a powerful facility. We illustrate its use.

This chapter is followed by a discussion on how files are created, edited, searched, updated and merged. This chapter also introduces a feature in C which allows direct accessing of structures stored in a file. All the concepts are illustrated with well chosen examples. The book concludes with a discussion of some miscellaneous features of C.

In the author's view the most important feature of this book is the illustration of all important language features through carefully selected examples. Besides illustrating the correct use of the language they also show how algorithms are formulated to solve problems. All program were executed on a workstation using ANSI C and the programs are reproduced by photo-offset to eliminate printing errors. The methodology adopted in the book will make it eminently suitable for self-study.

A book of this type naturally gained a number of ideas from previous books on the subject. The author would like to thank those authors, too numerous to acknowledge individually. The author would like to express his gratitude to Mr. S. Sundaram, Mr. T.S. Mohan, Mr. D. Sampath and Dr. S.K. Ghoshal for valuable discussions. The author thanks Mr. Karthikeyan for keying in the programs and testing them. The author would like to thank Ms. Mallika for secretarial assistance. Special thanks are due to the entire staff of Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, for their assistance and cooperation.

This manuscript was prepared using the funds provided by the Ministry of Human Resources Development, Government of India, through the Curriculum Development Cell of the Indian Institute of Science, Bangalore. The author would like to express his thanks for this support. The author thanks Prof. I.G. Sarma, Chairman, Centre for Continuing Education, Indian Institute of Science, for readily agreeing to his request for the above financial support.

Sincere thanks are due to Prof. C.N.R. Rao, Director, Indian Institute of Science, Bangalore, for evincing keen interest in the author's endeavours and for continued encouragement.

Finally the author would like to express his heartfelt appreciation to his wife Dharma, who corrected the manuscript and proofs and cheerfully devoted herself to the book-writing project.

Bangalore

V. Rajaraman

1. Computer Algorithms

Learning Objectives

In this chapter we will learn:

1. The properties of an algorithm
2. The basic structure of a computer necessary to carry out algorithms
3. Steps followed to solve a problem using a computer

1.1 ALGORITHMS

The fundamental knowledge necessary to solve problems using a computer is the notion of *an algorithm*. An algorithm is a precise specification of a sequence of instructions to be carried out in order to solve a given problem. Each instruction tells what task is to be performed. Specification of a sequence of instructions to do a job is used in many fields. One such specification taken from a popular magazine is given as Example 1.1.

Example 1.1 Recipe for Bournvita Cake

Ingredients

Bengal gram flour 2 cups, sugar 2 cups, ghee 3 cups, milk 1/2 cup, Bournvita 6 teaspoons, vanilla essence 5 drops, 1/2 cup water.

Method

Step 1: Warm 1 cup ghee. Put gram flour and fry for 2 minutes on low flame.

Step 2: Separately dissolve Bournvita in warm milk and add vanilla essence.

Step 3: Add 2 cups sugar in 1/2 cup water in a pan and warm it till sugar dissolves. Boil syrup till it become sticky.

Step 4: Pour the prepared Bournvita in syrup and stir continuously.

Step 5: Add fried gram flour to the syrup and continue stirring 15 to 20 minutes adding the remaining melted ghee until the mixture does not stick to the pan.

Step 6: Pour the mixture on a plate smeared with melted ghee shaking while pouring to ensure uniform spreading.

Step 7: After 10 to 15 minutes cut into 40 pieces.

Result

40 pieces of Bournvita cake.

The recipe given above is similar to an algorithm but it does not technically qualify as one. It is similar to an algorithm in the following respects:

1. It is a sequence of instructions to be carried out to solve the problem of making Bournvita cake.

2. It begins with a list of ingredients which is the *input.*

3. The sequence of instructions specify the actions to be carried out with the input to produce the cake which is the *output.*

4. It takes a finite time to execute the instructions and stop.

It does not technically qualify as an algorithm as the instructions given to make the cake are not precise. For example, in Step 3, the instruction to be carried out is to boil syrup until sticky. The interpretation of degrees of stickiness is subjective and would vary from person to person. In Step 5 the exact time to cook is not specified. In fact the instructions are meant for a reasonably experienced cook with "commonsense".

We will now examine another sequence of instructions (taken again from a popular magazine).

Example 1.2 Instructions to knit a sweater

Input

Needles No. 12 = 2

Wool 4 ply = 9 balls

Method

Step 1: Cast on 133 stitches

Step 2: Repeat Steps 3 and 4, 11 times

Step 3: Knit 2, *Purl 1, Knit 1, Repeat from * to last stitch, Knit 1.

Step 4: Knit, *Purl 1, Knit 1, Repeat from * to End

.....

.....

(Similar Steps)

Result

A Sweater

The above example illustrates the following points:

1. The instructions are much more precise and unambiguous when compared to the recipe for Bournvita cake. There is very little chance for misinterpretation.

2. The number of different types of actions to be carried out are very few. If we know how to knit, how to purl, cast stitches on or off needles and count then any sweater can be knit.

3. By a proper permutation and combination of this elementary set of actions a virtually infinite number of patterns may be created.

The preciseness of the instructions combined with the small variety of tasks to be performed to carry out each instruction facilitates design of a machine to automate knitting. In fact a machine similar to the modern computer was a loom designed by a French engineer Jacquard in 1801 which could be 'programmed' to create a large number of patterns. The program consisted of cards with specific patterns of holes in them which would control the loom.

We have illustrated two sequences of instructions for solving problems and stated that they are similar to algorithms but do not possess all the necessary attributes of algorithms.

What are the attributes which characterize an algorithm? What is the origin of this word?

The origin of this word has been hotly debated. It is, however, generally accepted among mathematicians that it comes from the name of a famous Arab mathematician Abu Jafar Mohammed ibn Musa al-Khowarizmi (literally meaning father of Jafar Mohammad, son of Moses, native of al-Khowarzm) who wrote the celebrated book "Kitab al jabr Walmuqabla" (Rules of restoration and reduction) around A.D.825. The last part of his name al-Khowarizmi got corrupted into algorithm.

An algorithm may be defined as a finite sequence of instructions which has the following five basic properties:

1. A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are the *inputs* which are processed by the algorithm.

2. The *processing rules* specified in the algorithm must be precise, unambiguous and lead to a specific action. In other words the instructions must not be vague. It should be possible to carry out the given instruction. For example, the instruction: "add 15 to 20" can be carried out. On the other hand it is not possible to carry out the instruction 'Find the square root of – 5'. An instruction which can be carried out is called an *effective* instruction.

3. Each *instruction* must be sufficiently *basic* such that it can, in principle, be carried out in a finite time by a person with paper and pencil.

4. The total time to carry out all the steps in the algorithm must be finite. An algorithm may contain instructions to repetitively carry out a group of instructions. This requirement implies that the number of repetitions must be finite.

5. An algorithm must have one or more *output.*

Based on the above definition we see that a recipe does not qualify as an algorithm as it is not precise. The knitting pattern, on the other hand, does qualify. Just as a machine may be built to knit sweaters using the knitting algorithm it is possible to build a machine to process data fed to it provided the data processing rules are specified as an algorithm. Further, the basic instructions used by the algorithm should be elementary such as add, subtract, read a character, write a character, compare numbers and find out which is larger, etc., so that these can be carried out by a machine. In the example on knitting we saw that even though the basic types of instructions are few it is possible to develop an enormous number of different interesting patterns by permuting and combining these. The same principle is used in building computers. Thus with a machine which can carry out as few as ten different operations it is possible to carry out a large variety of data processing tasks by designing appropriate algorithms.

1.2 CHARACTERISTICS OF COMPUTERS

The interesting features of a computer are as follows:

1. Computers are built to carry out a small variety of elementary instructions. It is not necessary to have more than fifty types of instructions even for a very versatile machine.

2. Each instruction is carried out in less than a millionth of a second.

3. Each instruction is carried out obediently with no questions asked.

4. Each instruction is carried out without making a mistake.

A computer may thus be thought of as a servant who would carry out instructions at a very high speed obediently, uncritically and without exhibiting any emotions. Giving instructions to an obedient servant who has no commonsense is difficult. Take the instance of a Colonel who sent his obedient orderly to a post office with the order "Buy ten 25 paise stamps". The orderly went with the money to the post office and did not return for a long time. The Colonel got worried and went in search of him to the post office and found the orderly standing there with the stamps in his hands. When the Colonel became angry and asked the orderly why he was standing there, pat came the reply that he was ordered to buy ten 25 paise stamps but not ordered to return with them!

A consequence of the uncritical acceptance of orders by the computer is the need to give extensive, detailed, and correct instructions for solving problems. This is quite challenging. If you wish to give correct instructions to an idiot to do a job, you better know how to do the job correctly yourself!

1.3 AN ILLUSTRATIVE COMPUTER ALGORITHM

In order to carry out a task using a computer the following steps are followed:

1. The given task is analysed.

2. Based on the analysis an algorithm to carry out the task is formulated. The algorithm has to be precise, concise and unambiguous. Based on our discussions we recognize that this task is difficult and time consuming.

3. The algorithm is expressed in a precise notation. This can be interpreted and executed by a computing machine and is called a *computer program.* The notation is called a *computer programming language.*

4. The computer program is fed to a computer.

5. The computer interprets the program, carries out the instructions given in the program and computes the results.

We will now consider an example of formulating an algorithm.

Example 1.3 A procedure to pick the largest tender from a set of tenders

We first decide the format in which tenders would be presented. We assume that each tender will have an identification number and the tender value.

Method

Having decided the data format we formulate the steps needed in a procedure to pick the highest tender.

Step 1: Read the first tender and note down its value as the maximum tender value so far encountered. Note down the tender identification number.

Step 2: As long as tenders are not exhausted do Steps 3 and 4. Go to Step 5 when tenders are exhausted.

Step 3: Read the next tender and compare the tender with the current maximum tender value.

Step 4: If this tender value is greater than that previously noted down as maximum tender then erase the previous maximum value noted and replace it by this new value. Replace the previously noted tender identification number by this identification number. Else do not do anything.

Step 5: Print the final value of maximum tender and its identification noted down in Step 4.

Observe that Example 1.3 qualifies as an algorithm as:

1. It has inputs. The inputs in this case are the set of tenders each with an identification number and value.

2. Each step in the algorithm is precise, has a unique interpretation and can be effectively carried out.

3. Each instruction is basic. If we sit with paper and pencil, and follow the instruction literally, we would be able to perform the job and do it in a finite length of time.

4. Observe that the algorithm is repetitive. In other words Steps 3 and 4 are repeated until all tenders are exhausted. The number of repetitions are finite as the number of tenders are finite. Thus the algorithm will terminate in a finite length of time.

5. The algorithm has an output, namely, the identification number of the maximum tender and the value of the maximum tender.

Other interesting features illustrated by this algorithm are:

1. The structure of the algorithm does not change when the input data change. In other words we can use as input one set of tenders and it will pick the highest tender in this set. If another set of tenders is used as input (after the completion of the previous job) then the highest tender in this new set will be picked by the same algorithm. It is thus a good idea to generalise an algorithm and write it so that it can work with a class of input data.

2. The number of instructions in an algorithm is finite and fixed. The number of *executions* of instructions, however, is not constant. It depends on the input data. For instance, the number of times Steps 3 and 4 in the algorithm of Example 1.3 are executed depends on the number of tenders input to the algorithm.

3. The output obtained by executing an algorithm depends on the input used by it. If input data has errors then the output will also be wrong. If for example, in Example 1.3, the tender identity of the maximum tender is incorrectly entered in the input the same mistake will be found in the output.

The preciseness of the instructions combined with the small variety of tasks to be performed to carry out each instruction facilitates design of a machine to automate knitting. In fact a machine similar to the modern computer was a loom designed by a French engineer Jacquard in 1801 which could be 'programmed' to create a large number of patterns. The program consisted of cards with specific patterns of holes in them which would control the loom.

We have illustrated two sequences of instructions for solving problems and stated that they are similar to algorithms but do not possess all the necessary attributes of algorithms.

What are the attributes which characterize an algorithm? What is the origin of this word?

The origin of this word has been hotly debated. It is, however, generally accepted among mathematicians that it comes from the name of a famous Arab mathematician Abu Jafar Mohammed ibn Musa al-Khowarizmi (literally meaning father of Jafar Mohammad, son of Moses, native of al-Khowarzm) who wrote the celebrated book "Kitab al jabr Walmuqabla" (Rules of restoration and reduction) around A.D.825. The last part of his name al-Khowarizmi got corrupted into algorithm.

An algorithm may be defined as a finite sequence of instructions which has the following five basic properties:

1. A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are the *inputs* which are processed by the algorithm.

2. The *processing rules* specified in the algorithm must be precise, unambiguous and lead to a specific action. In other words the instructions must not be vague. It should be possible to carry out the given instruction. For example, the instruction: "add 15 to 20" can be carried out. On the other hand it is not possible to carry out the instruction 'Find the square root of – 5'. An instruction which can be carried out is called an *effective* instruction.

3. Each *instruction* must be sufficiently *basic* such that it can, in principle, be carried out in a finite time by a person with paper and pencil.

4. The total time to carry out all the steps in the algorithm must be finite. An algorithm may contain instructions to repetitively carry out a group of instructions. This requirement implies that the number of repetitions must be finite.

5. An algorithm must have one or more *output.*

Based on the above definition we see that a recipe does not qualify as an algorithm as it is not precise. The knitting pattern, on the other hand, does qualify. Just as a machine may be built to knit sweaters using the knitting algorithm it is possible to build a machine to process data fed to it provided the data processing rules are specified as an algorithm. Further, the basic instructions used by the algorithm should be elementary such as add, subtract, read a character, write a character, compare numbers and find out which is larger, etc., so that these can be carried out by a machine. In the example on knitting we saw that even though the basic types of instructions are few it is possible to develop an enormous number of different interesting patterns by permuting and combining these. The same principle is used in building computers. Thus with a machine which can carry out as few as ten different operations it is possible to carry out a large variety of data processing tasks by designing appropriate algorithms.

1.2 CHARACTERISTICS OF COMPUTERS

The interesting features of a computer are as follows:

1. Computers are built to carry out a small variety of elementary instructions. It is not necessary to have more than fifty types of instructions even for a very versatile machine.

2. Each instruction is carried out in less than a millionth of a second.

3. Each instruction is carried out obediently with no questions asked.

4. Each instruction is carried out without making a mistake.

A computer may thus be thought of as a servant who would carry out instructions at a very high speed obediently, uncritically and without exhibiting any emotions. Giving instructions to an obedient servant who has no commonsense is difficult. Take the instance of a Colonel who sent his obedient orderly to a post office with the order "Buy ten 25 paise stamps". The orderly went with the money to the post office and did not return for a long time. The Colonel got worried and went in search of him to the post office and found the orderly standing there with the stamps in his hands. When the Colonel became angry and asked the orderly why he was standing there, pat came the reply that he was ordered to buy ten 25 paise stamps but not ordered to return with them!

A consequence of the uncritical acceptance of orders by the computer is the need to give extensive, detailed, and correct instructions for solving problems. This is quite challenging. If you wish to give correct instructions to an idiot to do a job, you better know how to do the job correctly yourself!

1.3 AN ILLUSTRATIVE COMPUTER ALGORITHM

In order to carry out a task using a computer the following steps are followed:

1. The given task is analysed.

2. Based on the analysis an algorithm to carry out the task is formulated. The algorithm has to be precise, concise and unambiguous. Based on our discussions we recognize that this task is difficult and time consuming.

3. The algorithm is expressed in a precise notation. This can be interpreted and executed by a computing machine and is called a *computer program.* The notation is called a *computer programming language.*

4. The computer program is fed to a computer.

5. The computer interprets the program, carries out the instructions given in the program and computes the results.

We will now consider an example of formulating an algorithm.

Example 1.3 A procedure to pick the largest tender from a set of tenders

We first decide the format in which tenders would be presented. We assume that each tender will have an identification number and the tender value.

Method

Having decided the data format we formulate the steps needed in a procedure to pick the highest tender.

Step 1: Read the first tender and note down its value as the maximum tender value so far encountered. Note down the tender identification number.

Step 2: As long as tenders are not exhausted do Steps 3 and 4. Go to Step 5 when tenders are exhausted.

Step 3: Read the next tender and compare the tender with the current maximum tender value.

Step 4: If this tender value is greater than that previously noted down as maximum tender then erase the previous maximum value noted and replace it by this new value. Replace the previously noted tender identification number by this identification number. Else do not do anything.

Step 5: Print the final value of maximum tender and its identification noted down in Step 4.

Observe that Example 1.3 qualifies as an algorithm as:

1. It has inputs. The inputs in this case are the set of tenders each with an identification number and value.

2. Each step in the algorithm is precise, has a unique interpretation and can be effectively carried out.

3. Each instruction is basic. If we sit with paper and pencil, and follow the instruction literally, we would be able to perform the job and do it in a finite length of time.

4. Observe that the algorithm is repetitive. In other words Steps 3 and 4 are repeated until all tenders are exhausted. The number of repetitions are finite as the number of tenders are finite. Thus the algorithm will terminate in a finite length of time.

5. The algorithm has an output, namely, the identification number of the maximum tender and the value of the maximum tender.

Other interesting features illustrated by this algorithm are:

1. The structure of the algorithm does not change when the input data change. In other words we can use as input one set of tenders and it will pick the highest tender in this set. If another set of tenders is used as input (after the completion of the previous job) then the highest tender in this new set will be picked by the same algorithm. It is thus a good idea to generalise an algorithm and write it so that it can work with a class of input data.

2. The number of instructions in an algorithm is finite and fixed. The number of *executions* of instructions, however, is not constant. It depends on the input data. For instance, the number of times Steps 3 and 4 in the algorithm of Example 1.3 are executed depends on the number of tenders input to the algorithm.

3. The output obtained by executing an algorithm depends on the input used by it. If input data has errors then the output will also be wrong. If for example, in Example 1.3, the tender identity of the maximum tender is incorrectly entered in the input the same mistake will be found in the output.

2. Developing Algorithms

Learning Objectives

In this chapter we will learn:

1. The broad strategy to solve a problem using a computer
2. Flow charts and their use
3. How to trace the execution of an algorithm by a computer

In this chapter we will explain, with a number of examples, how a broad strategy for solving a problem may be developed into a detailed sequence of elementary instructions. This process of development is aided by *flow charts*. A flow chart depicts pictorially the sequence in which instructions are carried out in an algorithm. Flow charts are used not only as aids in developing algorithms but also to document algorithms.

Another algorithm development and documentation tool is the *decision table*. A decision table depicts in a tabular form the set of conditions to be tested and the sequence of actions to be performed for solving a problem. This tool is useful in solving problems which require a large number of decisions.

In this chapter we will also introduce a simple model of a computer which will enable us to understand how an algorithm is executed by a computer. We will also see how the "correctness" of an algorithm can be checked by using test data on this simple model of a computer.

2.1 FLOW CHARTS

Example 2.1

The problem is to pick the largest of three numbers.

Broad Strategy

Read the three numbers A, B and C. Compare A with B. If A is larger compare it with C. If A is found larger than C then A is the largest number; otherwise C is the largest number. If in the first step A is smaller than or equal to B, then B is compared with C. If B is larger than C then B is the largest number else C is the largest number.

The strategy given above is not concise. It may be expressed much more clearly and concisely as a flow chart. The flow chart is given in Fig. 2.1.

The arrows in this flow chart represent the direction of flow of control in the procedure (to solve the problem), that is, having completed one task the direction of the arrow is to be followed to go to the next task.

For easy visual recognition a standard convention is used in drawing flow charts. This

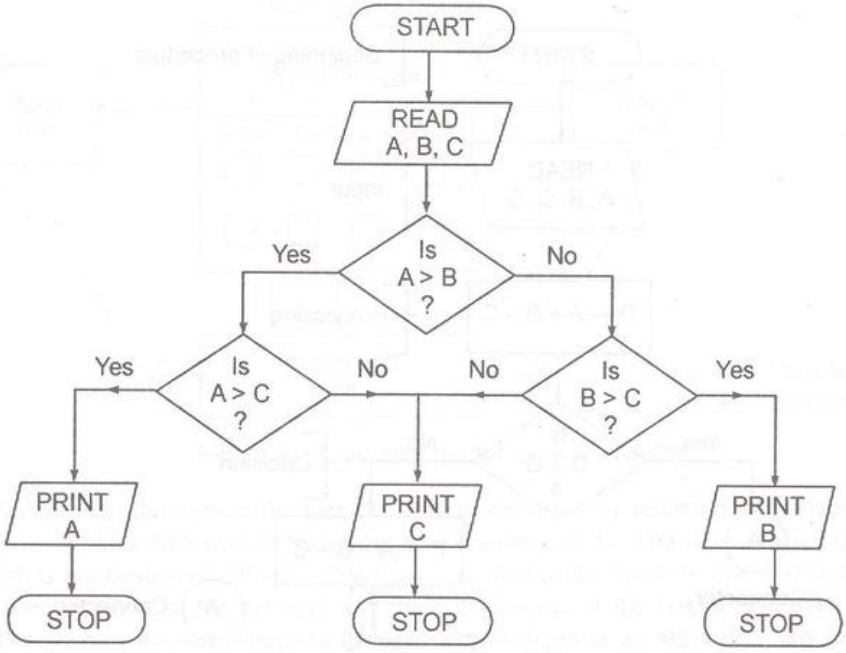


Fig. 2.1 A flow chart to pick the largest of three numbers.

convention has been standardised by the International Standards Organization and we will use this standard.

In this standard convention

(i) Parallelograms are used to represent input and output operations.

(ii) Rectangles are used to indicate any processing operation such as storage and arithmetic.

(iii) Diamond shaped boxes are used to indicate questions asked or conditions tested based on whose answers appropriate exits are taken by a procedure. The exits from the diamond shaped box are labelled with the answers to the questions.

(iv) Rectangles with rounded ends are used to indicate the beginning or the end points of a procedure.

(v) A circle is used to join different parts of a flow chart. This is called *a connector.* The use of connectors gives a neat appearance to a flow chart. Further, they are necessary if a flow chart extends over more than one page and the different parts are to be joined together. The fact that two points are to be joined is indicated by placing connectors at these points and by writing the same identifying letter or digit inside both the connectors.

(vi) Arrows indicate the direction to be followed in a flow chart.

Every line in a flow chart must have an arrow on it.

All the flow chart symbols and conventions are illustrated in Fig. 2.2. The use of connectors may also be seen in Fig. 2.2. The circles with 'A' written inside are to be joined together.

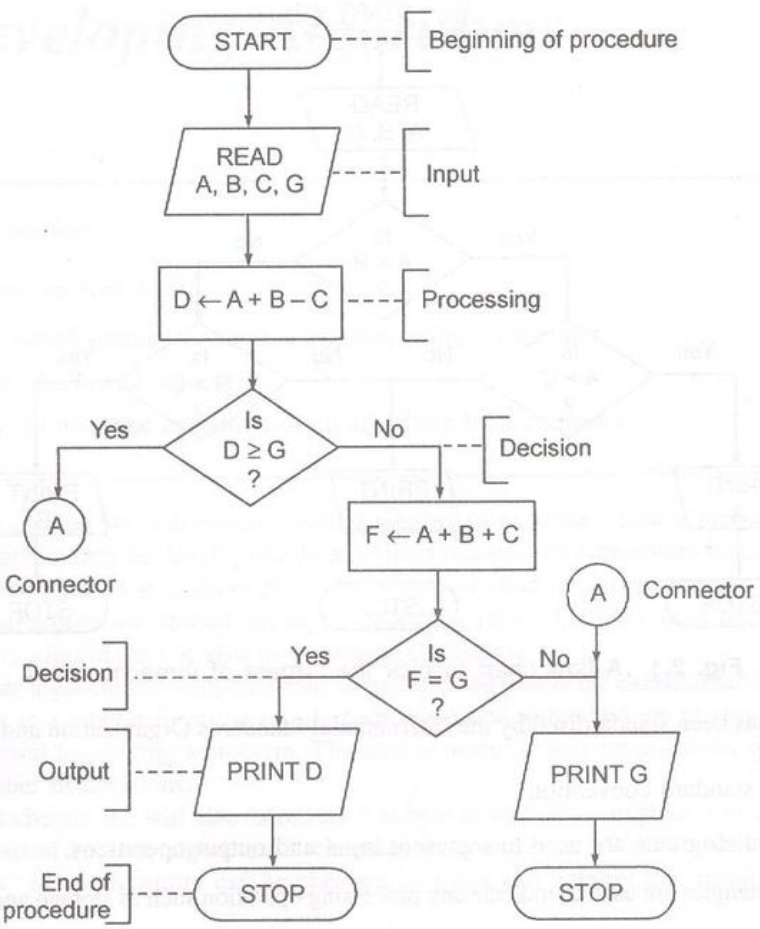


Fig. 2.2 Illustrating flow chart symbols and conventions.

2.2 A SIMPLE MODEL OF A COMPUTER

In this section we will use a simplified model of a computer to understand how algorithms are executed by a computer. This model consists of four blocks and is shown in Fig. 2.3. The four blocks are:

1. Input unit
2. Memory unit
3. Processing unit
4. Output unit

The *input unit* is used to read the data to be processed and the program which processes the data.

The *memory unit* is a store used to store the data, intermediate results and the program. We can imagine the memory to be partitioned into two parts. One part consists of a large number of labelled boxes—one box per data item. The other part stores the algorithm.

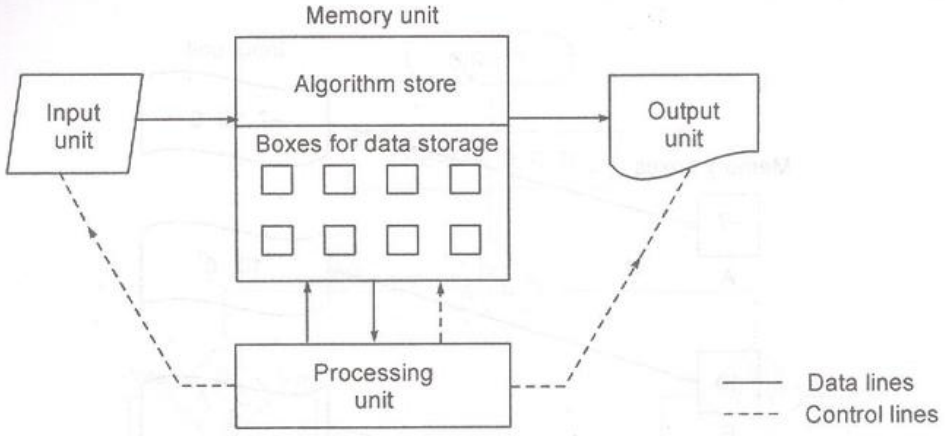


Fig. 2.3 A simple model of a computer.

A data item in a box in the memory may be retrieved by referring to the label or name of the box. When a data item is read from a box, a copy of the data item is used; the original data item is not destroyed. When a data item is written in the memory this data item is stored in the specified box in the memory and the old contents of the box is destroyed.

The *processing unit* interprets the instructions given in an algorithm and carries them out obediently and accurately. It has subunits to do arithmetic and logic.

The *output unit* is used to print the results of computation.

We will now use this model to explain how the algorithm given by the flow chart of Fig. 2.1 is executed. First an appropriately coded version of the flow chart is "stored" in the memory. The data required to solve the problem are kept ready at the input unit and arranged in the order in which they will be read by the input instruction in the flow chart.

In the example being considered if the values of A, B and C are – 7, 10, 6 then these three numbers are kept queued up and ready at the input unit.

The processing unit now starts from the START block in the flow chart. It gets all the units of the computer ready to follow the instructions stored in the memory. The next block is an instruction to READ A, B, C. This command is interpreted as follows:

> "Name three boxes in memory as A, B and C. Read the first number queued up at the input unit and place it in the box named A. Advance the queue at the input unit by one place. Take the number at the top of the queue and place it in the box named B. Repeat this procedure and place the number presently at the top of the queue in the box named C". The procedure is illustrated in Fig. 2.4.

The next block in the flow chart is the question "Is A > B?". This is interpreted as:

Read the numbers stored in boxes named A and B in memory; compare these two numbers and return an answer "yes" if the number in A 'is greater' than that in B and "no" otherwise. If the answer is "yes" then the processing unit follows the path labelled "yes" in the flow chart and if it is "no" then the path labelled "no" is followed.

With – 7, 10 and 6 stored in boxes named A, B and C respectively the answer to the question "Is A > B?" is "no" as – 7 is not greater than 10.

The processing unit now follows the "no" path in the flow chart and reaches the

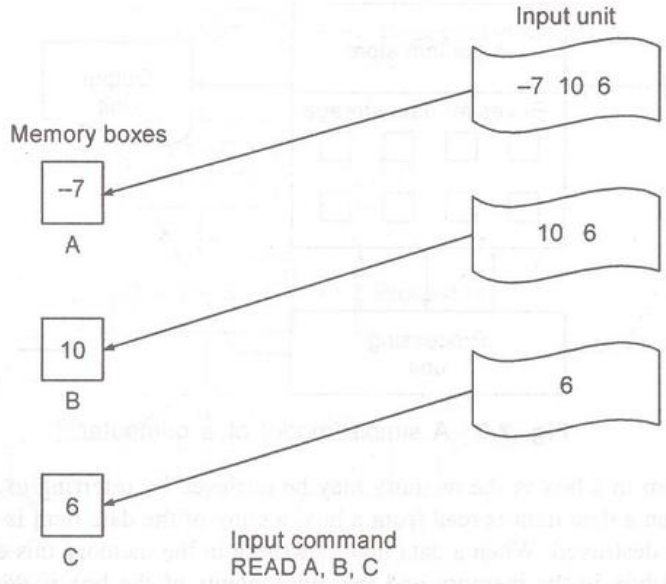


Fig. 2.4 Reading numbers from an input unit and storing in memory.

question "Is B > C ?". The processor compares the numbers stored in memory in boxes named B and C. Comparing 10 and 6 the processor answers "yes", follows the "yes" path and reaches the command PRINT B. This is interpreted as "ask the output unit to print the number stored in the memory box named B". Thus 10 is printed. After doing this the processor reaches the next command STOP and stops.

2.3 MORE FLOW CHARTING EXAMPLES

Example 2.2

In Example 2.1 we developed a flow chart to find the largest of three numbers. Suppose we want to pick the largest of four numbers instead of three numbers. We can, of course, use the same strategy and obtain the flow chart, shown in Fig. 2.5. This flow chart is bigger than the one of Fig. 2.1 but it is still not too large. If we, however, try to find the largest of five numbers using the same strategy the flow chart will become too large. The strategy used in Example 2.1 is thus not good enough to be generalised. We need an alternate strategy to pick the largest of a set of numbers assuming that we do not know how many numbers are there in the set.

Broad Strategy

The strategy is to read the first number and call it the largest number encountered so far. (If there is only one number in the list then that number would be the largest.) The second number (if present) is compared with the current largest number. If it is larger, then it replaces the current largest. These steps are repeated till all the numbers are exhausted. This strategy is expressed more precisely by the following algorithm written in English.

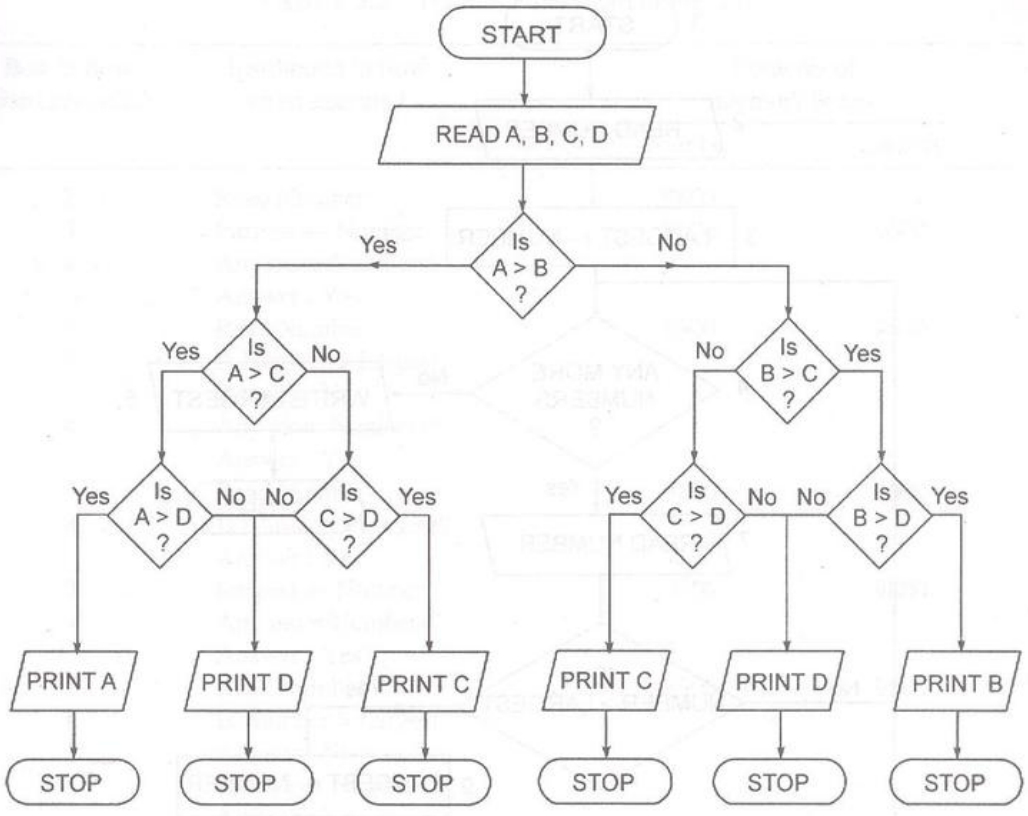


Fig. 2.5 A flow chart to pick the largest of four numbers.

Procedure 2.1: Procedure to pick the largest of a set of numbers

Step 1: Read Number

Step 2: Largest ← Number

Step 3: As long as numbers are there in the input repeat Steps 4 and 5

Step 4: Read Number

Step 5: If Number > Largest then Largest ← Number

Step 6: Write Largest

Step 7: Stop

This procedure is expressed pictorially by the flow chart of Fig. 2.6.

We will now carry out the instructions given in the flow chart as would be done by the processing unit. This methodology is adopted to make sure that the logic of solving the problem given by the flow chart is correct and that the correct results which we expect would indeed be obtained when the flow chart is faithfully followed. The method of carrying out the instructions given by the flow chart with paper and pencil is called *tracing a flow chart*. Tracing is carried out with sample data. "Memory boxes" are identified and named and their contents changed as instructed by the flow chart instruction. The flow chart of Fig. 2.6 is traced in

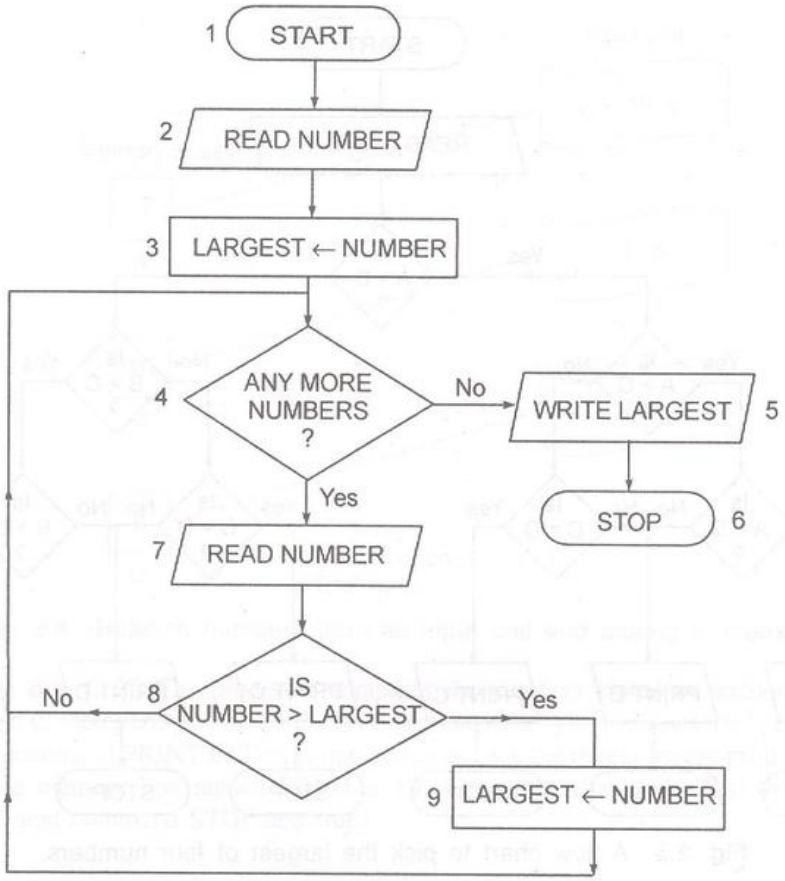


Fig. 2.6 A flow chart to pick the largest of a set of numbers.

Table 2.2 with the data of Table 2.1. The table depicts the contents of each memory box at each step during the execution of the flow chart.

TABLE 2.1 List of Numbers Used as Input

Item 1	9600 → Top of input queue
Item 2	8900
Item 3	9800
Item 4	8900
End of input list	

Referring to Table 2.2 the first row corresponds to executing instruction 2 of the flow chart. The first data item in the input queue is read and stored in memory box named Number. The next instruction commands that the contents of the memory box named Number should replace the contents of the memory box named Largest. (The symbol ← is to be interpreted as "is replaced by"). The old contents of the memory box to the right of ← is not changed. This is shown in row 2 of Table 2.2. The fourth instruction in the flow chart asks the

TABLE 2.2 Trace of Flow Chart of Fig. 2.6

Box in flow chart executed	Instruction in flow chart executed	Contents of memory boxes	
		Number	Largest
2	Read Number	9600	
3	Largest ← Number	9600	9600
4	Any more Numbers? Answer : Yes		
7	Read Number	8900	9600
8	Is Number > Largest? Answer : No		
4	Any more Numbers? Answer : Yes		
7	Read Number	9800	9600
8	Is Number > Largest? Answer : Yes		
9	Largest ← Number	9800	9800
4	Any more Numbers? Answer : Yes		
7	Read Number	8900	9800
8	Is Number > Largest? Answer : No		
4	Any more Numbers? Answer : No		
5	Write Largest	(Largest written = 9800)	
6	STOP		

question "any more numbers in the input?" It is necessary to ask this question at this stage because if the input has only one number it should be declared as the largest number. For the data of Table 2.1 the answer to this question is "yes". The processor thus proceeds to the instruction in box 7 of Fig. 2.6 which commands that the data item presently waiting at the top of the input data queue is to be read. This is item 2 as item 1 has already been read into the memory in Step 1. Thus the number 8900 is read and stored in the memory box named Number. The old contents of this box is overwritten by this new value. After this, a question is asked in box 8 of the flow chart. The answer to this question is "no". Following the arrow the flow chart returns to box 4 which again commands that the input queue is to be checked to see if more data are there. The answer is "yes". Thus item 3 which is on top of the queue is read into the memory box named Number replacing the old value. Executing box 8 of the flow chart we get a "yes" answer. Executing the command in box 9 of the flow chart the previous contents of Largest is replaced by the current contents of Number. This is shown in Table 2.2. Following the flow chart obediently the rest of the rows in Table 2.2 are obtained. Execution is finally stopped when no more numbers are left in the input. As seen from Table 2.2 execution stops after writing the largest number in the given list.

This example illustrates an important idea in procedure formulation intended for execution

by a computer. It is the idea of repetitively performing a set of instructions. In this procedure the basic idea is to pick up a number and check if it is higher than the current largest number. If it is higher replace the old largest by this and continue. This action is repetitively performed till the end of the input list is reached. The set of actions performed for each new number read is invariant. The actions are implemented in the flow chart (Fig. 2.6) by the Steps 4, 7, 8, (9); 4, 7, 8, (9);... till no numbers are left. The Steps 4, 7, 8(9) constitute a *loop* in the flow chart.

Example 2.3

It is required to develop a procedure to find the average height of boys and that of girls in a class.

Broad Strategy

Add the heights of all boys and count the number of boys. Divide the sum of heights by the number of boys. Do the same for girls.

Detailed Procedure Development

Assume that for each student in the class one slip is filled up which has the Roll No. of the student, a code identifying the sex of the student (1 is used to indicate boy and 2 for girl), and the height of the student in cms. Table 2.3 illustrates the data organization.

The procedure is to pick up a slip and find the sex code. If it is 1 then the height is added to total boys height. The number of boys is also incremented by 1. If the sex code is 2 then the height is added to total girls height. The girl count is incremented by 1. This

TABLE 2.3 Student Height Data Organization

Slip No.	Roll No.	Sex Code	Height
1	4234	1	95
2	4682	2	105
3	4762	2	100
4	4784	1	100

procedure is repeated till the end of slips is reached. When this is reached then boy and girl height totals and total numbers of boys and girls are separately available and the separate averages can be computed. The detailed procedure needs accumulators to be set up which have to be cleared to zero before actual accumulation of totals begins. The procedure is given as Procedure 2.2.

Procedure 2.2: Procedure to find average heights of boys and girls

Step 1: Initialize counters to accumulate totals and store zeros in them.

Step 2: Repeat Steps 3 and 4 until end of slips is reached.

Step 3: Read a slip.

Step 4: If sex code = 1

Total boy height ← Total boy height + height

Total boys ← Total boys + 1

Else

Total girl height ← Total girl height + height
Total girls ← Total girls + 1

Step 5: Av. boy height ← Total boy height/Total boys
Av. girl height ← Total girl height/Total girls

Step 6: Print Total boys, Av. boy height,
Total girls, Av. girl height

Step 7: Stop.

This procedure is depicted as a flow chart in Fig. 2.7 and traced in Table 2.4 with the data in Table 2.3.

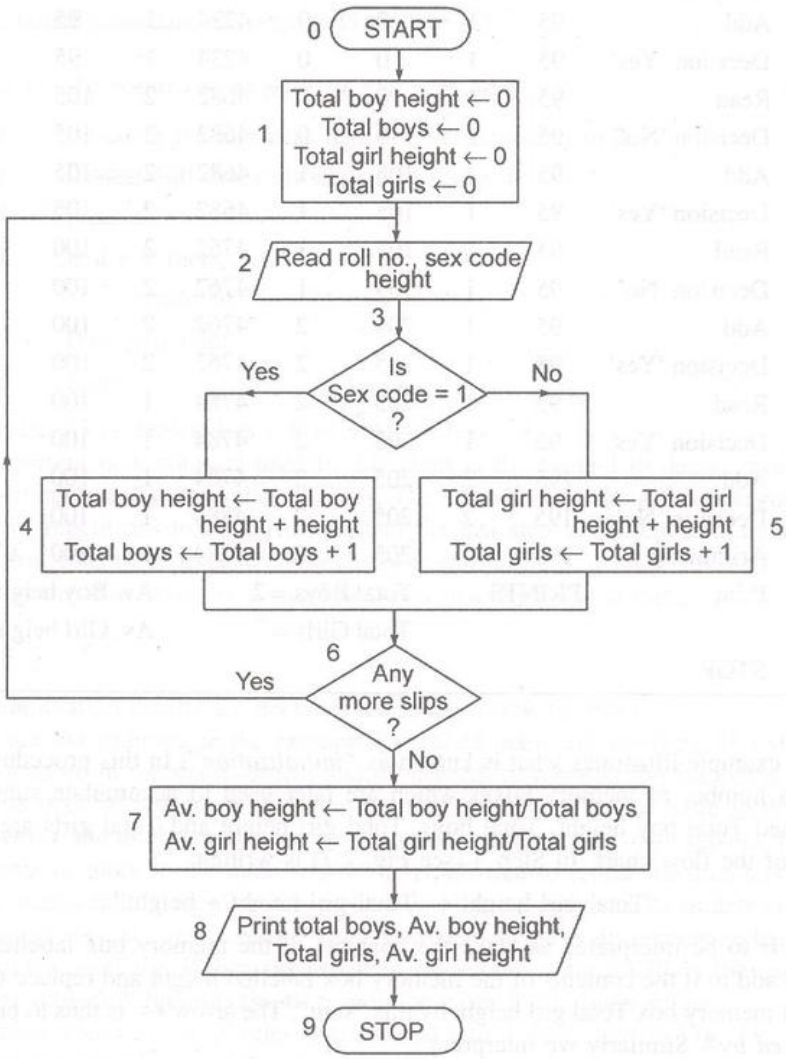


Fig. 2.7 Flow chart to find the average height of boys and girls.

TABLE 2.4 A Trace of Flow Chart of Fig. 2.7

Box in flow chart executed	Nature of box	Contents of memory boxes								
		Total boy height	Total boys	Total girl height	Total girls	Roll No.	Sex Code	Height	Av. boy height	Av. girl height
1	Process	0	0	0	0					
2	Read	0	0	0	0	4234	1	95		
3	Decision 'Yes'	0	0	0	0	4234	1	95		
4	Add	95	1	0	0	4234	1	95		
6	Decision 'Yes'	95	1	0	0	4234	1	95		
2	Read	95	1	0	0	4682	2	105		
3	Decision 'No'	95	1	0	0	4682	2	105		
5	Add	95	1	105	1	4682	2	105		
6	Decision 'Yes'	95	1	105	1	4682	2	105		
2	Read	95	1	105	1	4762	2	100		
3	Decision 'No'	95	1	105	1	4762	2	100		
5	Add	95	1	205	2	4762	2	100		
6	Decision 'Yes'	95	1	205	2	4762	2	100		
2	Read	95	1	205	2	4784	1	100		
3	Decision 'Yes'	95	1	205	2	4784	1	100		
4	Add	195	2	205	2	4784	1	100		
6	Decision 'No'	195	2	205	2	4784	1	100		
7	Arithmetic	195	2	205	2	4784	1	100	97.5	102.5
8	Print	PRINTS		Total Boys = 2,			Av. Boy height = 97.5			
				Total Girls = 2,			Av. Girl height = 102.5			
9	STOP									

This example illustrates what is known as *"initialization"*. In this procedure zeros are stored in a number of memory boxes which are later used to accumulate sums. Memory boxes named Total boy height, Total boys, Total girl height and Total girls are set to zero in Step 1 of the flow chart. In Step 5 (see Fig. 2.7) is written:

"Total girl height ← Total girl height + height"

This is to be interpreted as "take the contents of the memory box labelled Total girl height and add to it the contents of the memory box labelled height and replace the previous contents of memory box Total girl height by this "sum". The arrow ← is thus to be interpreted as *"replaced by"*. Similarly we interpret:

Total girls ← Total girls + 1

as "contents of memory box Total girls is *replaced by* (the previous contents of) Total girls + 1".

This example also uses a loop.

Example 2.4

It is required to develop a procedure to count the number of non-zero data in a list of 100 data.

Broad Strategy

The strategy is to read a data value. Check if it is non-zero and increment a counter. The reading is continued till 100 data values are read.

Detailed Procedure

The detailed procedure is given as Procedure 2.3.

Procedure 2.3: To count the number of non-zero data

Step 1: Initialize the non-zero data counter nzdcount to 0

Step 2: Repeat 100 times all steps up to Step 4

Step 3: Read d

Step 4: If d ≠ 0 then

nzdcount ← nzdcount + 1

Step 5: Print nzdcount

Step 6: Stop

Procedure 2.3 is depicted as a flow chart in Fig. 2.8.

An important new notation used in this chart is the symbol to depict repetition of a group of instructions a fixed number of times. The symbol is an elongated hexagon in which the number of repetitions to be performed and the last step to be repeated are given. The group of instructions to be executed repeatedly is said to be in *a loop*. The end of the loop is indicated in the flow chart by a small hexagon inscribed with a digit.

Example 2.5

Students' examination results are declared using the following rules.

There are two subjects in the examination called main and ancillary. If a student gets 50 percent or more in the main subject and 40 percent or more in the ancillary, he passes. If he gets less than 50 percent in the main he must get 50 percent or more in the ancillary to pass. However, the minimum passing marks are 40 percent in the main subject. If a student gets 60 percent or more in the main subject he is allowed to repeat the ancillary subject if the ancillary marks fall below 40 percent. However, there are a group of students in the class who are granted special consideration. Their pass percentage is 40 percent in the main and 40 percent in the ancillary. If they get less than 40 percent in the ancillary they are allowed to repeat that subject if they obtain 40 percent or more in the main subject.

The above complex set of rules is translated into a flow chart in Fig. 2.9.

Observe the complexity of the chart and the appearance of the same test in more than one place in the chart.

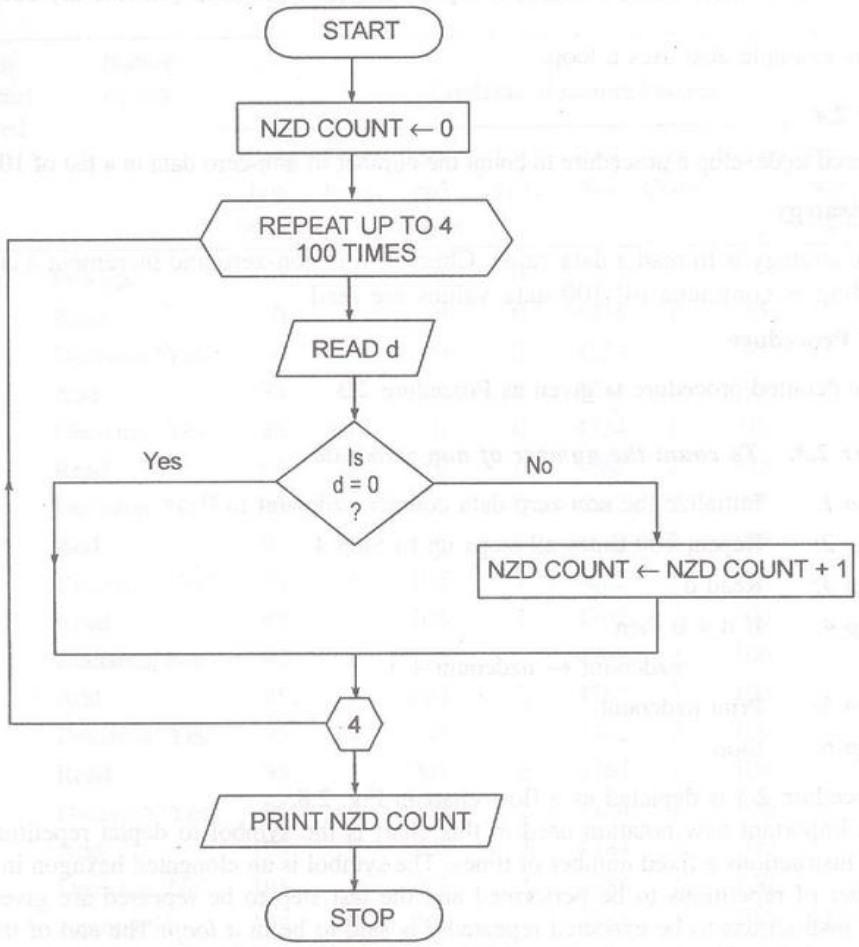


Fig. 2.8 Flow chart to count non-zero digits in a set of 100 digits.

Flow charts are not suitable for problems with complex decision logic as they become clumsy when the number of conditions to be tested is large. Another technique of representing complex decision logic is by using *decision tables*.

The procedure for Example 2.5 which was given as the flow chart of Fig. 2.9 can be equivalently represented by the decision table of Table 2.5. In this table a 'x' against an action means perform the action and ' – ' against an action means do not perform the action. The table is self-explanatory. The reader is urged to compare the flow chart of Fig. 2.9 with the table (Table 2.5).

We will discuss the use of decision tables at greater length later in this book.

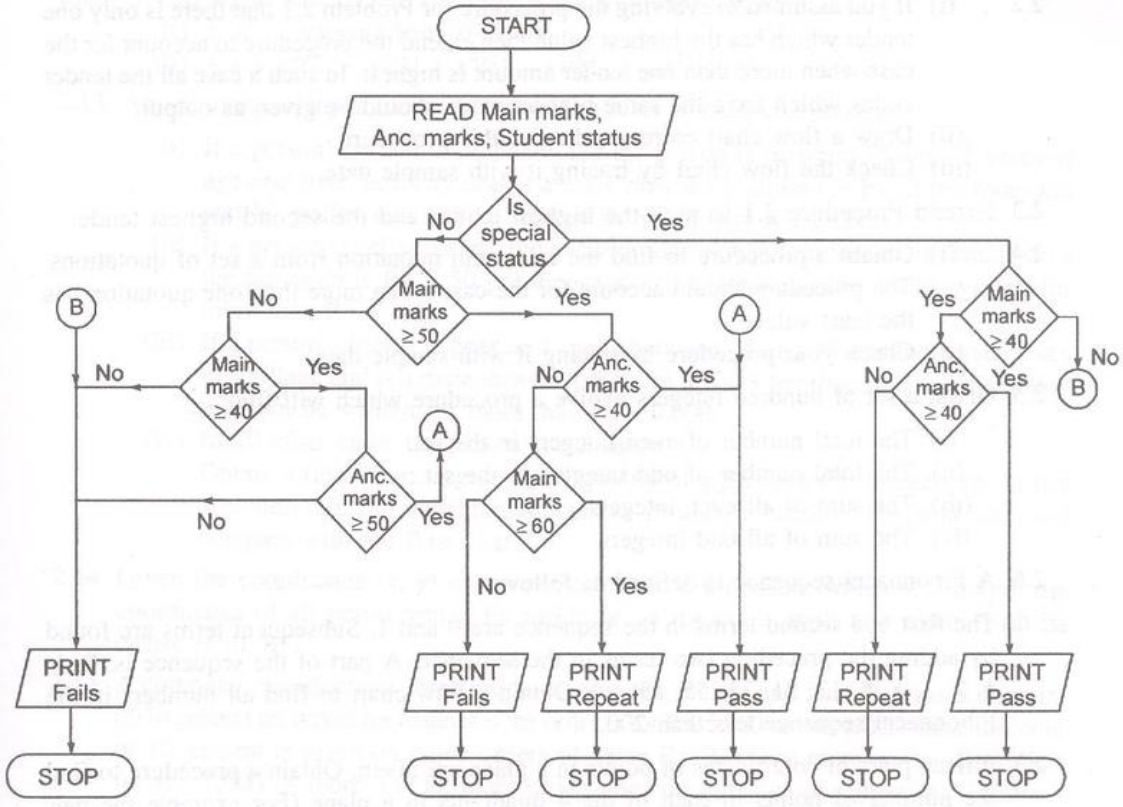


Fig. 2.9 Flow chart depicting the processing of examination results.

TABLE 2.5 A Decision Table Corresponding to Flow Chart of Fig. 2.9

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	E
Main marks percentage	≥ 50	≥ 40	≥ 60	≥ 40	≥ 40	L
Anc. marks percentage	≥ 40	≥ 50	≥ 40	≥ 40	< 40	S
Special status	No	No	No	Yes	Yes	E
Actions						
Pass	x	x	–	x	–	–
Repeat ancillary	–	–	x	–	x	–
Fail	–	–	–	–	–	x

EXERCISES

2.1 (i) Extend Procedure 2.1 to pick the largest of a set of tenders. Assume that each tender is represented by a tender identification number and the amount of the tender.

(ii) Draw a flow chart using standard symbols to represent this procedure.

2.2 (i) If you assumed in evolving the procedure for Problem 2.1 that there is only one tender which has the highest value then extend the procedure to account for the case when more than one tender amount is highest. In such a case all the tender codes which have the same highest value should be given as output.
(ii) Draw a flow chart corresponding to this procedure.
(iii) Check the flow chart by tracing it with sample data.

2.3 Extend Procedure 2.1 to pick the highest tender and the second highest tender.

2.4 (i) Obtain a procedure to find the minimum quotation from a set of quotations. The procedure should account for the case when more than one quotation has the least value.
(ii) Check your procedure by tracing it with sample data.

2.5 Given a set of hundred integers evolve a procedure which will find:
(i) The total number of even integers in the set.
(ii) The total number of odd integers in the set.
(iii) The sum of all even integers.
(iv) The sum of all odd integers.

2.6 A Fibonnacci sequence is defined as follows:

The first and second terms in the sequence are 0 and 1. Subsequent terms are found by adding the preceding two terms in the sequence. A part of the sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Obtain a flow chart to find all numbers in the Fibonnacci sequence less than 200.

2.7 Fifteen pairs of coordinates of points in a plane are given. Obtain a procedure to find the number of points in each of the 4 quadrants in a plane (For example the pair (– 3, 6) is in the second quadrant).

2.8 (i) Given hundred pairs of length and breadth of rectangles, obtain a procedure to find all the rectangles whose area is greater than their perimeters. (For example the area of the rectangle with length = 5 and breadth = 4 is greater than its perimeter.)
(ii) Extend the procedure to find the average area of the rectangles found in part (i).

2.9 (i) 100 student records are given each with the roll number of the student and his or her marks in three subjects. Obtain a procedure which will read the 100 records one after another and find the total marks, average marks and division of each student. A student with average marks of 60 percent or above is placed in first division, those with average marks less than 60 but 50 or above are placed in second division, those with less than 50 are declared fail.
(ii) Extend the procedure of part (i) to find the average marks of all 100 students and the number of students passing in first division and second division.

2.10 Given ten sequences each consisting of six digits evolve a procedure to print those sequences which are in strict ascending order. For example the sequence (5, 6, 7, 9, 11, 14) is in strict ascending order whereas the sequence (5, 5, 6, 7, 9, 11) is not in strict ascending order.

2.11 Given three points (x_1, y_1), (x_2, y_2) and (x_3, y_3) obtain a flow chart to check if they are collinear.

2.12 Obtain algorithms to find the sum of the following series:

(i) $\sum x^r$ r varying from 0 to 100

(ii) $-x + x^2/2! - x^3/3! + x^4/4! - x^5/5! \ldots - x^{13}/13!$

2.13 An insurance company uses the following rules to calculate premium:

(i) If a person's health is excellent *and* the person is between 25 and 35 years of age *and* lives in a city *and* is a male *then* the premium is Re. 2 per thousand and his policy may not be written for more than Rs. 2 lakhs.

(ii) If a person satisfies all the above conditions except that the sex is female *then* the premium is Rs. 1.50 per thousand and her policy may not be written for more than Rs. 1 lakh.

(iii) If a person's health is poor *and* age is between 25 and 35 *and* the person lives in a village and is a male *then* the premium is Rs. 9 per thousand and his policy may not be written for more than Rs. 10,000.

(iv) In all other cases the person is not insured.

Obtain a flow chart to give the eligibility of a person to be insured, his or her premium rate and maximum amount of insurance. Obtain a decision table and compare with the flow chart.

2.14 Given the coordinates (x, y) of ten points obtain a procedure which will output the coordinates of all points which lie inside or on the circle with unit radius with its centre at (0, 0).

2.15 A company manufactures three products: engines, pumps and fans. It gives a discount of 10 percent on orders for engines if the order is for Rs. 5,000 or more. The same discount of 10 percent is given on pump orders of value Rs. 2,000 or more and on fan orders for Rs. 1,000 or more. On all other orders they do not give any discount. Obtain a decision table corresponding to this word statement.

3. Programming Preliminaries

Learning Objectives

In this chapter we will learn:

1. The need for a high level programming language and a compiler
2. The difference between source and object programs
3. Difference between syntax rules and semantics of a programming language
4. History of development of C language

3.1 HIGHER LEVEL PROGRAMMING LANGUAGES FOR COMPUTERS

We saw in Chapter 2 how a flow chart is evolved to solve a problem. The flow chart is an aid to the programmer to plan his strategy to solve a problem. It cannot be directly interpreted by a computer. A computer can interpret and execute a set of coded instructions called *machine language instructions*. For instance a set of three instructions to add two numbers and store the answer in a (hypothetical computer) is given below:

Operation Code	Memory Location
0110	10001110
0111	10001111
1000	01110001

In the first instruction 0110 is an operation code to load into a storage register in CPU an operand from location 10001110 in memory. The operation code 0111 (of the second instruction) instructs the computer to add the contents of memory location 10001111 to the contents of the storage register (where the first operand has been stored by the previous instruction). The third operation code 1000 instructs that the result which is in the storage register is to be copied into location 01110001 of memory. It is evident that we should memorise all the binary operation codes and keep track of the contents of each and every location in memory to be able to write a machine language program. This is difficult as there would be more than 100 different machine instruction codes and hundreds of thousands of locations in memory. Errors will be made so often that it will be difficult to get correct answers. Secondly, the operation codes will differ from one machine model to another. Thus we have to rewrite a program for an application again and again as new computers are introduced.

Fortunately, it is not necessary to write programs in machine language any more. Computer programs may be written in a host of *high level programming languages*. The instructions in

a high level programming language are not coded numbers. They resemble ordinary English statements. Further, it is not necessary to refer to numerical memory addresses. Symbolic names may be used to label memory locations. Thus high level programming languages are easy to learn and use.

Associated with each high level programming language is an elaborate computer program which translates it into machine language. This translating program is called a *compiler* or a *language processor.* The resulting machine language program is called the *object program* and the original program written in the high level programming language is called the *source program.* These terms are explained pictorially in Fig. 3.1.

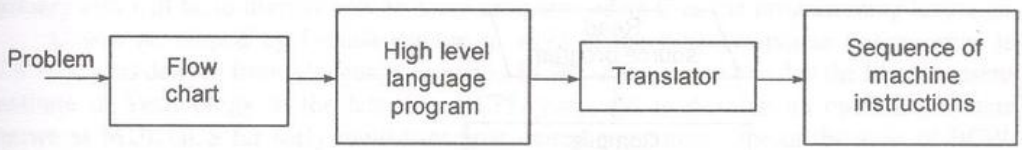


Fig. 3.1 Terms used in high level language translation.

Compilers are written by professional programmers. A high level programming language may be used in different computers. The translators are different leading to different machine language equivalents of the source program. This is illustrated in Fig. 3.2.

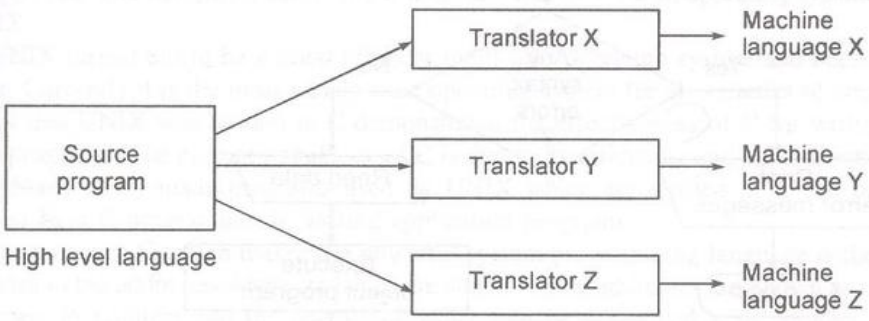


Fig. 3.2 Illustrating machine independence of high level language.

A high level programming language can be designed so that it can be used on any computer regardless of who manufactured it or what model it is. Such a language is also known as a *machine independent language.*

As a high level machine independent programming language has to be translated by a computer program it is essential to define precisely the rules for writing instructions in that language. These are the *syntax rules* of the language. When a source program in a high level language is fed to a computer, the compiler analyses each instruction and determines if any syntax rules are violated. If any syntax rules are violated by the programmer in writing his source program, these errors are printed out as messages to the programmer to enable him to correct his program. These error messages are called *compile time diagnostic error messages.* A source program which has no syntax errors is translated to an equivalent machine language object program. The object program is now executed by the computer. During execution also

errors can occur. These mostly relate to improper data or improper sequence of instructions. Such errors, if they occur, may stop a computation abruptly. Appropriate *execution time error messages* are printed for the information of the programmer. Programs which have neither syntax errors nor execution errors are successfully executed and answers are printed out.

A flow chart showing the steps in compilation of a high level language program is shown in Fig. 3.3.

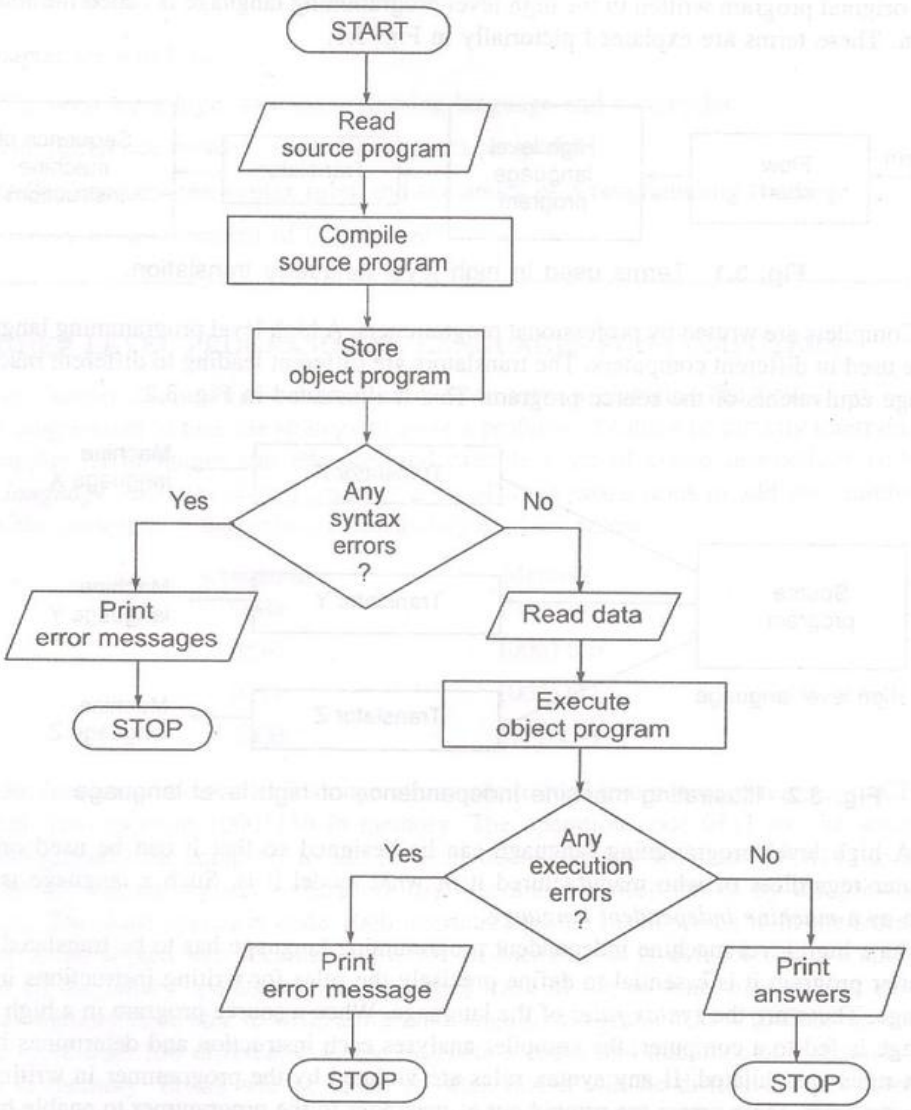


Fig. 3.3 A flow chart showing the steps in compilation of a high level language.

It is essential to emphasize that compilers *cannot* diagnose errors in logic. They can diagnose only grammatical errors in writing the source program. For example, if a programmer

(by mistake) writes an instruction to divide, instead of to multiply, this error cannot be detected by a compiler. Programs containing such errors will be successfully executed and give incorrect answers. To avoid such errors it is essential to be careful and precise in writing programs and attend to the smallest detail. It is also necessary to choose test cases which are manually computed and compared with computer solutions. The test cases must be chosen so that they test thoroughly all aspects of the program.

3.2 C LANGUAGE

In this book we will discuss in detail the high level programming language known as C. Our primary aim will be to discuss how to solve problems using C as the programming language.

C was developed by Dennis Ritchie in 1972 at the Bell Telephone Laboratories in U.S.A. C was derived from a language known as BCPL which was evolved at the Massachusetts Institute of Technology in the late 60s. BCPL was used to develop an operating system known as MULTICS for early multi-user time shared computers. One of the aims of BCPL was to achieve efficiency in compiled code. Thus BCPL was defined such that a translator could produce efficient machine language code. C being a successor of BCPL has a similar philosophy. C language has been defined so that it has the advantages of a high level language, namely, machine independence. At the same time it is concise, providing only the bare essentials required in a language so that a translator can translate it into an efficient machine language code. The first major use of the C language was to write an operating system known as UNIX.

UNIX turned out to be a good efficient multi-user operating system and became very popular. Currently it is the most widely used operating system for all varieties of computers. The fact that UNIX was written in C demonstrated the effectiveness of C for writing large system programs. The current popularity of C is due to its efficiency and its connection with UNIX. Many ready-made programs used by UNIX which are written in C can be easily borrowed by a C programmer in writing application programs.

An aspect of C which makes it a powerful system programming language is the access it provides to the addresses where variables are stored. These addresses are known as pointers. The access to pointers and the operations which can be performed with pointers is what distinguishes C from other high level languages such as FORTRAN and COBOL. Pascal does provide pointers but restricts their use. C has minimal restrictions on the use of pointers.

C has recently been standardised by the American National Standards Institute (ANSI). In this book we will use this standard. We will discuss all the important features available in C. Over a hundred programs will be written in C to illustrate how to solve problems using this language.

The main disadvantage of C, particularly for a beginner, is its conciseness and extensive use of pointers. Brief statements may be written to carry out complex computation. This is a disadvantage as it is difficult to read C programs. In other words it often becomes difficult to understand C programs written by professional programmers as they tend to use tricks with pointers and other C features to make the program small.

A program which cannot be read and understood cannot be maintained (i.e. cannot be modified or improved). As program maintenance is a very important activity of programmers it is a good programming practice to write programs which are easy to read and understand. Tricks available in C to write concise programs should not be used. In this book our attempt

will be to write programs which are easy to read and understand and thereby teach good programming style.

3.3 ON THE DESCRIPTION OF A PROGRAMMING LANGUAGE

The classical method of learning a language is to first learn the alphabets or characters used in the language. The next step is to learn how to combine these alphabets to form words, words to form sentences and sentences to express a thought. Modern language teaching is somewhat different. It lets a student read sentences and paragraphs and understand their meanings. By illustrating many such examples a student finds it easier to learn formal rules of grammar. It is more important to express a thought precisely in a language rather than be an expert in grammar. We follow a similar route in this book. We illustrate the need for certain structures to solve problems and then define rules of grammar.

Any language has two distinct parts. These are the *syntax* and *semantics*. By syntax we mean rules which are to be followed to form structures which are grammatically correct. Semantics assigns meanings to syntactic structures. A sentence which is syntactically correct need not be semantically correct. For instance, the sentence "Sita plays the violin" is both syntactically and semantically correct whereas the sentence "Violin plays Sita" is syntactically correct as it does not violate any rules of grammar. It is, however, semantically incorrect. In a similar manner we have to learn both syntax and semantics of programming languages. Syntax is easily checked mechanically and is mostly done by the language translator. Checking semantic correctness is the responsibility of the programmer.

We re-emphasize that it is not sufficient to design programs which are syntactically and semantically correct. Programs should also be simple to understand and easy to modify and improve. Thus it is essential to develop a good *style* in programming which would lead to not only correct programs but understandable and maintainable programs.

4. Simple Computer Programs

Learning Objectives

In this chapter we will learn:

1. The structure of a C program
2. How to develop small programs in C
3. How to input data and print the results obtained from a C program

---

In this chapter we will write some programs in C and as we proceed learn the syntax and semantic rules of the language.

4.1 WRITING A PROGRAM

We will consider as the first example a program to find the area and perimeter of a rectangle whose sides are p and q. The area is given by:

$$\text{area} = pq$$

and the perimeter by

$$\text{perimeter} = 2(p + q)$$

We have written the formulae above as we would normally do in algebra. In algebra, when we write pq where p and q are variables it is implied that p is multiplied by q. When we write an instruction for a computer, however, we have to explicitly specify that multiplication is to be performed. The formula is thus written in the C language as:

```
area = p * q
```

which is called a *statement*. p, q and area are called *variable names*. The symbol * is the *multiplication operator*. p * q is an *arithmetic expression*. The symbol = is the *assignment operator*. The semantics or meaning of the above statement in C is:

> "Read the contents of a memory box whose label or name is p and multiply it by the contents of another memory box named q. Store the result in a memory box named area".

Figure 4.1 illustrates this. Observe that the contents of p and q are *copied* and used. They are not destroyed. The formula for the perimeter is written in C as:

```
perimeter = 2 * (p + q)
```

In the above C statement, 2 is an *integer constant*. As before p, q and perimeter are variable names. 2 * (p + q) is an arithmetic expression. The semantics of the statement is:

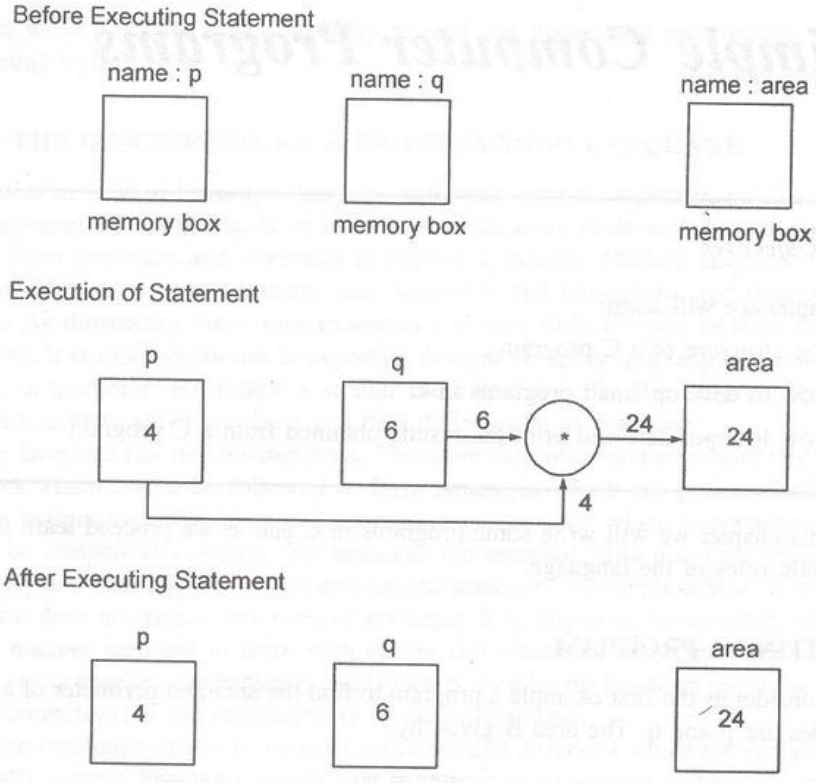


Fig. 4.1 Execution of statement: area = p * q.

"Add the contents of memory box named p with that of the memory box named q. Store the result temporarily in the CPU. Multiply this result by 2. Store the answer in memory box named perimeter".

Figure 4.2 illustrates this. We have seen in this simple example the need for constants, variable names and operators. Example Program 4.1 is a C program to solve this problem.

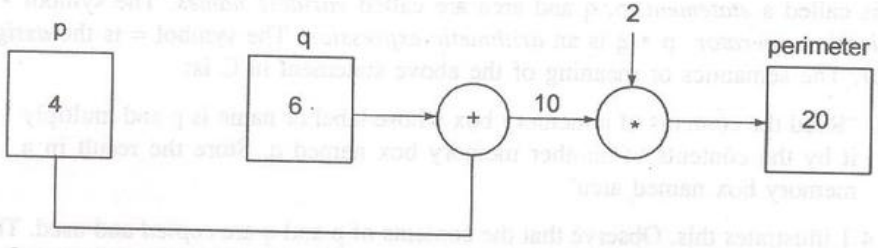


Fig. 4.2 Calculation of perimeter. Observe that the constant 2 is supplied by the compiler.

We will now explain this program. The first line in the program starts with /* and ends with */. Anything written between /* and */ is called a *comment*. In the C language comments are an aid to the programmer to read and understand a program. It is not a statement of the

```
/* Example Program 4.1 */
/* This program finds the area and perimeter of a
    rectangle */
#include <stdio.h>

main()
{
  int p, q, area, perimeter;
  p = 4;
  q = 6;
  area = p*q;
  perimeter = 2*(p+q);
  printf("area = %d \n", area);
  printf("perimeter = %d \n", perimeter);
}  /* End of main */
```

Program 4.1 Area and perimeter of a rectangle.

language. The compiler ignores comments. It is a good practice to include comments in a program which will help in understanding a program. Observe the line

#include < stdio.h >

after the comments.

This is called a *preprocessor directive*. It is written at the beginning of the program. It commands that the contents of the file stdio.h should be included in the compiled machine code at the place where # include appears. The file stdio.h contains the standard input/output routines. All preprocessor directives begin with the pound sign # which must be entered in the first column. The # include line *must not* end with a semicolon. Only one preprocessor directive can appear in one line.

The next line is main ( ). It defines what is known as a *function* in C. A C program is made up of many functions. The function main ( ) is required in *all* C programs. It indicates the start of a C program. We will use main ( ) at the beginning of all programs. Observe that main ( ) is *not* followed by a comma or semicolon.

Braces { and } *enclose* the computations carried out by main ( ). Each line in the program is a *statement*. Every statement is terminated by a semicolon;. The statement itself can be written anywhere in a line. More than one statement can be on a line as a semicolon separates them. However, it is a good practice to write one statement per line.

In this program the first statement is:

int p, q, area, perimeter;

This statement is called a *declaration*. It informs the compiler that p, q, area and perimeter are variable names and that individual boxes must be reserved for them in the memory of the computer. Further it tells that the data to be stored in the memory boxes named p, q, area and perimeter are integers, namely, whole numbers without a fractional part (e.g., 0, 1, 2, ...). These can be either positive or negative. The statement is said to *declare* p, q, area and perimeter as of type *integer*. The statement ends with a semicolon. The effect of this statement is shown in Fig. 4.3. Observe that 4 locations are reserved and named as: p, q, area and perimeter respectively.

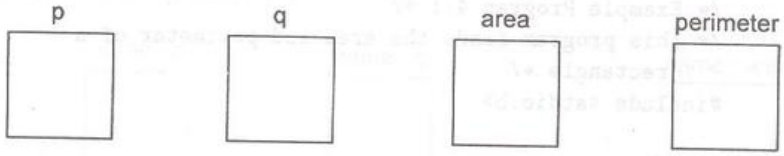


Fig. 4.3 Effect of defining p, q, area and perimeter as integer variable names.

The statement p = 4; is an *assignment statement*. It commands that the integer 4 be stored in the memory box named p. When the statement is executed the integer, 4 will be stored in the memory box named p as shown in Fig. 4.4.

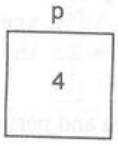


Fig. 4.4 Effect of statement p = 4.

More briefly we can state that this statement *assigns* a value 4 to the variable name p. The next statement q = 6; assigns 6 to q. Following this is the statement:

```
area = p * q;
```

This is an *arithmetic statement*. It commands that the numbers stored in memory boxes p and q should be *copied* into the CPU. The original contents of p and q remain in their respective boxes. These numbers are multiplied by the CPU and the product is stored in the box named area. After executing this statement the box named area will contain 24 as shown in Fig. 4.1.

```
area = p * q;
```

The next statement perimeter = 2 * (p + q); is also an arithmetic statement. We have already explained how this statement is interpreted. After this statement is executed the number stored in memory box named perimeter should be 20.

The next two statements in the program are commands to display the contents of memory box named area and perimeter respectively. C language does not provide any statements for reading data into memory or displaying the contents of memory. Instead a set of standard *library functions* provided by the UNIX operating system for input and output are borrowed and used. The library functions for input and output are contained in the file stdio.h which we included using a preprocessor directive. Unless otherwise specified C assumes that the input of data is from the keyboard of a video terminal and the output is displayed on the screen of a video terminal. The library function used for display is printf( ). The general form of this function is:

```
printf(format-string, variable 1, variable 2, ..., variable n)
```

The format-string is enclosed in quotes. It specifies any message to be printed, and the manner in which the contents of variables are to be displayed for each of the variables in the list of variables.

In the statement:

printf("area = %d\n", area)

the format-string is:

"area = %d\n"

and the variable is area. This format-string specifies that area is to be displayed as it is. The special symbol %d says: "interpret the variable area occurring after the comma in the printf statement as an integer and display its value". The symbol \n causes the display to advance to the next line. Thus when this statement is carried out we will see on the screen

area = 24

After the statement:

printf("perimeter = %d\ n", perimeter)

is carried out we will have on the screen

area = 24

perimeter = 20

Observe that perimeter = 20 is displayed on the next line as /n in the previous printf statement advanced the display by one line.

4.2 INPUT STATEMENT

In Example Program 4.1 the variables p and q are assigned values. If we want to find the area and perimeter of another rectangle with sides 8 and 10, for instance, we have to replace the statements p = 4 and q = 6 by p = 8 and q = 10 respectively. We have to then recompile the program and run it again. This is not a good idea. We should actually have a facility to assign any desired values to p and q by feeding the data through an input unit. (Usually the keyboard of a Video Display Unit.) This is possible if we can use an *input statement.* The input statement in C uses the library function scanf. To read an integer from the keyboard and assign it to variable' p the statement is:

scanf("%d", &p);

The format string is %d. This says that the value to be stored in the specified variable name is an integer. Following this string we write &p which states that the variable name to which the integer is assigned is p. The symbol & in front of p is essential. This symbol gives the address of the variable name p so that data can be stored in it. We can similarly write another statement to read a value into a variable name q. It is:

scanf("%d", &q);

There should be no blank space between the quote sign "and%, % and d and d and". No blank should separate & and q. We can combine the two statements into one and write:

scanf("%d %d", &p, &q);

In this case the values of p and q input on a terminal should be separated by a blank space. If 8 and 10 are to be stored in p and q then the values input through the keyboard are:

8 10

The integer 8 is followed by one or more blanks after which the integer 10 is typed on the keyboard.

We can now rewrite Example Program 4.1. Using this input statement by replacing the statements:

p = 4; q = 6; with

scanf("%d %d", &p, &q);

At the end of the program we type a line containing the data. This program is written as Example Program 4.2. To compile and execute the program we have to use the commands appropriate to the operating system of the computer being used. The student should get this information from the appropriate manuals. In Appendix I we give the method which is used when a Unix operating system is used. We will write some more simple programs in the next section.

```
/* Example Program 4.2 */
/* Use of Input/Output statements */
#include <stdio.h>

main()
{
  int p, q, area, perimeter;
  scanf("%d %d", &p, &q);
  /* Reads values of p and q from terminal
  & before p and q is essential */
  printf("p = %d, q = %d\n", p, q);
  /* The printf statement above is used to check if scanf
     has read the values of p and q correctly */
  area = p*q;
  perimeter = 2*(p+q);
  printf("area = %d\n", area);
  printf("perimeter = %d\n", perimeter);
} /* End of main */
```

Program 4.2 Area and perimeter—Use of input/output statements.

4.3 SOME C PROGRAM EXAMPLES

In this section we will write a few simple programs in C. This is intended to let a student start programming quickly. Detailed rules of syntax of the language will be given later.

Example 4.1

As the first example we will write a program to convert a temperature given in Celsius to Fahrenheit. The formula for conversion is

$$F = 1.8C + 32 \tag{4.1}$$

where C is the temperature in Celsius and F that in Fahrenheit. The program is given as Example Program 4.3. Observe that we have used more descriptive names for variable names rather than just F and C. Using such variable names makes the program more readable. Also

```
/* Example Program 4.3 */
/* This program converts a Celsius temperature to
   Fahrenheit */
#include <stdio.h>

main()
{
  float fahrenheit, celsius;
  scanf("%f", &celsius);
  printf("Celsius = %f\n", celsius);
  fahrenheit = 1.8 * celsius + 32.0;
  printf("Fahrenheit = %f\n", fahrenheit);
} /* End of main */
```

Program 4.3 Celsius to Fahrenheit conversion.

observe that when we write the term 1.8C in Equation 4.1 the multiplication operator is implied. In a computer program such operators should be explicitly specified.

After # include line we define the main function. Observe the statement:

float fahrenheit, celsius;

This is a declaration which informs the compiler that fahrenheit and celsius are variable names in which *floating point* numbers will be stored. By floating point number we mean a number with a fractional part, for example, 14.456. (We will describe later in detail the representation of floating point numbers in a computer.) The following statement

scanf("%f", &celsius)

is a command to read a floating point number and store it in variable name celsius. The format-string %f indicates that a floating point number is to be read. We follow this with a statement to print the value read. The next statement is an arithmetic statement. Contents of celsius is multiplied by 1.8 and added to 32.0 and stored in fahrenheit. Observe that the number 32 is written with a decimal point to indicate that it is a floating point number. The last statement is to print the answer.

We will now consider an example of summing a small series. The use of intermediate variables simplifies the program and reduces the number of arithmetic operations performed by a computer.

Example 4.2

Required to calculate

$$\text{sum} = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - x^{10}/10!$$

A program to calculate the sum is given as Example Program 4.4. We will present a more concise program later in the book. Observe that after calculating x^2 and storing it in x2 it is used to calculate x^4, x^6 etc. See also that we have written the statement

printf("x = %f \n", x)

immediately after scanf. The printf statement displays on the screen the value of x read by scanf. This enables the programmer to make sure what has been read into x. Such a check is important, particularly for a beginner. This is called echo check, as we are printing the value read immediately after reading it.

```
/* Example Program 4.4 */
/*  Program to sum a series */
#include <stdio.h>

main()
{
  float x, x2, x3, x4, x6, x8, x10, sum;
  scanf("%f", &x);
  printf("x = %f\n", x);
  x2 = x*x;
  x4 = x2*x2 / 24.0;
  x6 = x4*x2 / 30.0;
  x8 = x6*x2 / 56.0;
  x10 = x8*x2 / 90.0;
  sum = 1.0 - 0.5*x2 + x4 - x6 + x8 - x10;
  printf("value of x = %f, sum = %f\n", x, sum);
} /* End of main */
```

Program 4.4 Summing a series.

Example 4.3

Mangoes cost Rs. 52.80 per dozen. It is required to find the cost of 28 mangoes to the nearest paisa and to print the answer as Rupees and Paise. A program to do this is given as Example Program 4.5.

In Example Program 4.5 in the main ( ) function we first define the variables. The scanf

```
/* Example Program 4.5 */
/* This program calculates the price of 28 mangoes
   given the price of a dozen mangoes  */
#include <stdio.h>

main()
{
  int rupees, paise, cost_of_28;
  float cost_dozen, rcost_28;
  /* read cost of 12 mangoes */
  scanf("%f", &cost_dozen);
  printf("Cost of dozen = %f\n", cost_dozen);
  rcost_28 = ((cost_dozen * 100.0 /12.0) * 28.0
      + 0.5);
  cost_of_28 = rcost_28;
  rupees = cost_of_28 / 100;
  paise = cost_of_28 % 100;
  printf("Cost of 28 mangoes\n");
  printf("Rs.%d ps.%d\n", rupees, paise);
}  /* End of main */
```

Program 4.5 Price of 28 mangoes.

statement reads the price of a dozen mangoes. The next statement prints the values read by scanf. This printf statement is used to assure oneself that the correct data has been read. The next statement calculates the cost of 28 mangoes to the nearest integral paisa. Adding 0.5 ensures that the integral part is rounded correctly. cost_of_28 is declared an integer whereas rcost_28 is a floating point number. When we assign rcost_28 to cost_of_28 the fractional part of rcost_28 is removed and the integral part is stored in cost_of_28. In the next statement the operator / is a division operator. Thus cost_of_28 is divided by 100. As both numerator and denominator are integers the result obtained when / operator is used as an integer. For example, 438/100 = 4 and not 4.38. The quotient obtained when cost_of_28 is divided by 100 is stored in rupees.

Observe that in the next statement:

```
paise = cost_of_28 % 100;
```

We have introduced an operator %. This operator is the *mod* operator which gives the remainder obtained after dividing. For example 438% 100 = 38. The *mod* operator is specifically used when the operands are integers. The rest of the program is self-explanatory.

Example Program 4.5 is correct and will work for the given problem. It is, however, not general. If we want to calculate the price of 29 mangoes instead of 28 the program should be re-written. When a program is written it should be made as general as possible. We can easily generalize the program by keeping the quantity to be bought and cost per dozen as variables. This will enable us to use the same program for different quantities of mangoes and varying costs by changing only the data entered on a terminal. This generalized program is given as Example Program 4.6.

```
/* Example Program 4.6 */
/* This program calculates the price of x mangoes
   given the price of a dozen mangoes */
#include <stdio.h>

main()
{
  int rupees, paise, cost, quantity;
  float cost_dozen;

  scanf("%f %d", &cost_dozen, &quantity);
  printf("Cost of dozen = %f, quantity = %d\n",
         cost_dozen, quantity);
  cost = ((cost_dozen * 100.0 / 12.0) *
          quantity + 0.5);
  rupees = cost/100;
  paise = cost % 100;
  printf("Number of mangoes = %d\n", quantity);
  printf("cost = Rs.%d ps.%d\n", rupees, paise);
} /* end of main */
```

Program 4.6 Price of x mangoes.

Observe the statement:

```
cost = ((cost_of_dozen * 100.0/12.0) * quantity + 0.5)
```

In this statement cost_of_dozen on the right hand side is a floating point numbers. The calculation will thus give a floating point number as result. The left hand side, however, is an integer variable name in which only an integer can be stored. Thus the fractional part of the floating point number obtained as result on the right hand side is removed and the integral part is stored in cost.

Example 4.4

A five digit positive integer is given. It is required to find the sum of individual digits. For example, if the given number is 96785 the required sum is 9 + 6 + 7 + 8 + 5 = 35. Finding the sum of digits of a number is used in data processing to validate numbers read in as data. A program to find the sum of digits of an integer is given as Example Program 4.7.

```
/* Example Program 4.7 */
/* This program reads a positive integer which is
   5 digits long and sums the digits in it  */
#include <stdio.h>

main()
{
 int digit_1, digit_2, digit_3, digit_4,
 digit_5, sum, number, n;
 scanf("%d", &number);
 printf("number = %d\n", number);
 n = number;
 digit_1 = n % 10;
 n = n / 10;
 digit_2 = n % 10;
 n = n / 10;
 digit_3 = n % 10;
 n = n / 10;
 digit_4 = n % 10;
 n = n /10;
 digit_5 = n;
 sum = digit_1 + digit_2 + digit_3 + digit_4
   + digit_5;
 printf("sum of digits = %d\n", sum);
} /* End of main */
```

Program 4.7 Adding digits of an integer.

One method of checking whether the program is correct is to act as though we are a computer and execute each instruction as it is given in the program. This is known as tracing the program. Assume that the number read in is 96785. We now follow the instructions in the program.

After reading the number the program prints it. The values stored in the variables in each step is shown below:

n = 96785

digit_1 = n % 10 = 96785 % 10 = 5

n = n/10 = 96785/10 = 9678

digit_2 = n % 10 = 9678 % 10 = 8

n = n/10 = 9678/10 = 967

digit_3 = n % 10 = 967 % 10 = 7

n = n/10 = 967/10 = 96

digit_4 = n % 10 = 96 % 10 = 6

n = n/10 = 96/10 = 9

digit_5 = 9

sum = digit_1 + digit_2 + digit_3 + digit_4 + digit_5

= 5 + 8 + 7 + 6 + 9 = 35

At the end of the program the data read into the variable name number is preserved.

Observe that there is a simple repetitive pattern in this program. A concise program may be written which employs this pattern. In order to do this we need some more instructions which we will learn in future chapters.

EXERCISES

4.1 Write a program to convert fahrenheit temperature to celsius.

4.2 Read the following program and explain what it does. Trace it with yard = 20

```
# include < stdio.h >
main ( )
{ int yard, feet, inch;
    scanf("%d", &yard);
    feet = 3 * yard;
    inch = 12 * feet;
    printf("feet=%d inch =%d", feet, inch);
}
```

4.3 Write a program to convert pounds to kilograms.

4.4 Given a 5 digit integer write a program to print it in reverse order (for example given 92674 the result should be 47629).

4.5 Write a program to read the radius of a circle and compute its area and circumference.

4.6 Given a number = $d_5\ d_4\ d_3\ d_2\ d_1$ find $\sum i\ d_i \bmod 11$ where i varies from 1 to 5.

4.7 Write a program to find the rupee equivalent of x U.S. dollars using the exchange rate to nearest ten paise. For example if x = 42.25 and exchange rate is Rs. 31.16 per dollar the answer should be Rupees 1317 paise 90. (You should allow any arbitrary exchange rate and read it in as data in your program.)

4.8 Write a program to express a length x given in millimeters in meters, centimeters and millimeters.

5. Numeric Constants and Variables

Learning Objectives

In this chapter we will learn:

1. Different types of numeric constants which may be used in C
2. Types of numeric variables in C and how they are declared and initialized
3. How constants may be given symbolic names
4. Specification of syntax rules for numeric constants and variables

In the last chapter we saw how small programs are written in C. Before we write larger programs it is necessary to learn the rules of syntax of the language in some detail. We will give the rules for specifying constants and variable names in this chapter.

5.1 CONSTANTS

The term constant means that it does not change during the execution of a program. Constants may be classified as:

(i) integer constants

(ii) floating point constants

Integer Constants

Integer constants are whole numbers without any fractional part. Floating point constants (also known as real constants), on the other hand, may have fractional parts. There are three types of integers which are allowed in C. They are:

(i) Decimal constants (base 10)

(ii) Octal constants (base 8)

(iii) Hexadecimal constants (base 16)

The allowed digits in a *decimal* constant are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The digits of an *Octal* constant can be 0, 1, 2, 3, 4, 5, 6, 7 and that in hexadecimal constant are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Rule

A decimal integer constant must have atleast one digit and must be written without a decimal point. The first digit in the constant should not be 0. It may have either sign + or – If either sign does not precede the constant it is assumed to be positive.

Examples

The following are valid decimal integer constants:

(i) 12345

(ii) 3468

(iii) – 9746

The following are invalid decimal integer constants:

(i) 11. (Decimal point not allowed)

(ii) 45,256 (Comma not allowed)

(iii) $ 125 (Currency symbol not allowed)

(iv) 025 (First digit should not be 0 for a decimal integer)

(v) x248 (First symbol not a digit)

C also provides *short* and *long* integers. Integer size depends on the word size of a machine. In a 32 bit machine it has a range + $(2^{31} - 1)$ to $- 2^{31}$; short integer declaration means it is a 16 bit integer in such a machine with a range $(2^{15} - 1)$ to $- 2^{16}$. The size of long integer depends on the machine and the C language implementation.

The qualifier *signed* or unsigned may be applied to any integer. Unsigned integers are 0 or positive. Arithmetic with unsigned integers are modulo n where n is the number of bits used to store the integer. Suppose we have two short unsigned integers a and b.

Let a = 1024 and b = 65035

then d = a + b = 523 (66059 *mod* 65536 = 523)

(The maximum number that can be stored in a 16 bit unsigned short integer = $2^{16} - 1$ = 65535).

A long integer constant is written with a L or lower case *l* at the end of the number, for example, 67894676L. An unsigned integer is written with the letter U at the end of the integer. For example, 4578U is an unsigned integer. 14678946UL is an unsigned long integer. A declaration int implies that it is a signed integer.

Examples

– 246876 (an integer)

– 267896789L (a long integer)

– 28 (A short integer)

26789U (An unsigned integer)

36794267UL (unsigned long integer)

The rule to write octal constant is:

Rule

An octal constant must have atleast one digit and start with the digit 0. It must be written without a decimal point. It may have either sign + or –. A constant which has no sign is taken as positive.

Examples

The following are valid octal constants:

(i) 0245

(ii) – 0467

(iii) + 04013

The following are invalid octal constants:

(i) 25 (Does not begin with 0)

(ii) 0387 (8 is not an octal digit)

(iii) 04.32 (Decimal point not allowed)

The rule to write a hexademical constant is:

Rule

A hexadecimal constant must have atleast one hexadecimal digit and start with 0x or 0X. It may have either sign.

Examples

The following are valid hexadecimal constants:

(i) 0x14AF

(ii) 0X34680

(iii) – 0x2673E

The following are invalid hexadecimal constants:

(i) 0345 (Must start of 0x)

(ii) 0x45H3 (H not a hexadecimal digit)

(iii) Hex2345 (0x defines a hexadecimal number, not Hex)

Floating point constants

A floating point constant may be written in one or two forms called *fractional form* or the *exponent form*, the rules for writing a floating point constant in these two forms is given below:

Rule

A floating point constant in a fractional form must have atleast one digit to the right of the decimal point. It may have either the + or the – sign preceding it. If a sign does not precede it then it is assumed to be positive.

Examples

The following are valid floating point constants:

(i) 1.0

(ii) – 0.5678

(iii) 5800000.0

(iv) – 0.0000156

The following are invalid:

(i) 1 (Decimal point missing)

(ii) 1. (No digit following the decimal point)

(iii) – 1/2. (Symbol / illegal)

(iv) .5 (No digit to the left of the decimal point)

(v) 58,678.94 (comma not allowed)

In the examples above even though 5800000.0 and – 0.0000156 are valid floating point constants it is more convenient to write them as 0.58×10^7 and 0.156×10^{-4} respectively. ($\times$ is multiplication symbol). The exponent notation for writing floating point constants provides this facility. In this notation these two numbers may be written as:

(i) 0.58E7 or 0.58e7

(ii) – 0.156E – 4 or – 0.156e – 4

In the above examples E7 and E – 4 are used to represent 10^7 and 10^{-4} respectively (E or e may be used).

In this notation a floating point constant is represented in two parts: a mantissa part (the part appearing before E) and an exponent part (the part following E). Thus 0.58 and – 0.156 are the respective mantissas and 7 and – 4 the exponents. We will now give the formal rule for writing floating point constants in the exponent form.

Rule

A floating point constant in the exponent form consists of a mantissa and an exponent. The mantissa must have at least one digit. It may have a sign. The mantissa is followed by the letter E or e and the exponent. The exponent must be an integer (without a decimal point) and must have at least one digit. A sign for the exponent is optional.

The actual number of digits in the mantissa and the exponent depends on the computer being used, for example, the mantissa may have up to seven digits and the exponent may be between – 38 and + 38 in IBM PC. For details of other machines one may refer to the appropriate manufacturer's manual.

Examples

The following are valid floating point constants in the exponent form:

(i) (a) 152E08 (b) 152.0E8 (c) 152e + 8
(d) 152E + 08 (e) 15.2e9 (f) 1520E7

(ii) – 0.148E – 5

(iii) 152.859E25

(iv) 0.01540e05

Observe that in (i) above six equivalent ways of writing the same constant are given.

The following are invalid:

(i) 152.AE8	(Mantissa must have a digit following.)
(ii) 125 * e9	(* not allowed)
(iii) + 145.8E	(No digit specified for exponent)
(iv) – 125.9E5.5	(Exponent cannot be a fraction)
(v) 0.158E + 954	(Exponent too large)
(vi) 125,458.25 e – 5	(Comma not allowed in mantissa)
(vii) .2E8	(A digit must precede . in mantissa)

5.2 SCALAR VARIABLES

A quantity which may vary during program execution is called a variable. Each variable has a specific storage location in memory where its numerical value is stored. The variable is given a name and the variable name is the "name tag" for the storage location. The value of the variable at any instant during the execution of a program is equal to the number stored in the storage location identified by the name of the variable. Figure 5.1 illustrates this. The variable name in Fig. 5.1 is total and its contents is 2.8.

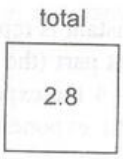


Fig. 5.1 Illustrating a variable name and its content.

When a variable name can contain only one number it is called a *scalar variable*. In C the word *identifier* is used as the general terminology for names given to variables, functions, constants, structures, etc.

As variable names are name tags of memory locations where numbers are stored, and numbers are of two types, integer and floating point, one has to declare a variable name as being of type floating point or type integer. A number stored in memory location with variable name of type integer is an integer (with the restrictions given while discussing integer constants) and one stored in a location with a variable name of type float may have a fractional part (with restrictions discussed in the last section). We will now give the rules for forming identifiers used as variable names.

Rule

An identifier is any combination of one or more letters (upper or lower case) or digits. The first character in the identifier must be a letter. It must not contain any character other than letters or digits.

The underscore _ is taken as a letter. Identifier may have any number of letters and digits. Even though the C syntax rule allows identifiers of any length, in practice, the number of characters in an identifier which are significant is dependent on the specific translator.

ANSI standard specifies at least 31 characters in an identifier as significant. Both upper and lower case letters (i.e. capital and small letters) may be used for writing an identifier.

Upper and lower case letters are treated as distinct. In other words temp and Temp are two different identifiers. The convention in C is to use only lower case letters for variable names.

Examples

The following are valid identifiers which may be used to name variables:

(i) nA
(ii) theta
(iii) amin
(iv) min50
(v) temp_X
(vi) absolute

The following are invalid identifiers:

(i)	$count	(First character $ invalid)
(ii)	roll#	(Character # not allowed)
(iii)	no.	(Character . invalid)
(iv)	2nd	(First character digit not allowed)
(v)	ROLL NO	(No blanks allowed in a name)
(vi)	case	(It is a reserved word)

We stated above that case is an illegal variable name as it is a *reserved word*. A set of keywords are used in C for special purposes and may not be used for any other purpose. They are known as reserved words. A list of reserved words in C are given in Appendix II.

5.3 DECLARING VARIABLE NAMES

Identifiers used as variable names are explicitly typed as float or integer by the following *declaration* which should appear at the beginning of a program before the variable names are used.

type-name variable name, ..., variable name;

We have used italics for type-name to emphasize that it is a reserved word. The *type_name* available for variable names storing numbers are *int* for integers and *float* for floating point numbers. An example of valid declaration is given below:

int n, height, count, digit;

float average, x_coordinate, p;

When a variable name is declared then a memory location is identified and given this name. The following declarations of variables are invalid:

float, a, b, c; (comma after float not valid)

int:x; (: after int invalid)

real x, y; (real is not the correct type-name)

Note that an enormous number of variable names may be defined using the rules to form identifiers. It is good practice to exploit this enormous choice in naming variables in programs by using meaningful identifiers. Thus for example if one is calculating a temperature, pressure and the life of a catalyst in a chemical process the corresponding variables in programs may be named temperature, pressure and catalyst_life. Contrast this with the names t, p and cl which one may have given. These latter names are not meaningful and thus provide no help in understanding the role of a variable name.

Double precision floating point numbers

When a variable name is declared float the number of mantissa digits stored and exponent size depends on the computer on which the C program runs. For machines with 32 bit word such as IBM PCs it is 7 digit mantissa and an exponent range of ± 38. In some calculations the mantissa length may not be sufficient. Thus C provides a type-name called *double*. The use of double in declaring a floating point variable provides 16 mantissa digits storage for the variable name. Arithmetic with variable names declared double carries out all arithmetic operations using the 16-digit mantissa. (The exponent size is normally not affected. It is, however, implementation dependent.)

If we want variable names theta, lambda and iota to store 16 digit mantissa floating point numbers we use the declaration:

```
double theta, lambda, iota;
```

All floating point constants are normally taken as double precision. If we want to specify a constant as requiring only 7 digit precision then we use the suffix f or F.

For example,

```
4.35678F, 3.4678e – 15F, – 4.96789e – 16f
```

are all taken as constants needing 7 digit mantissa 436.87492L is a long double precision constant.

5.4 DEFINING CONSTANTS

In C a value can be assigned to a variable name when it is declared. If no value is assigned it is undefined. For example we can write:

```
int x = 2, p = – 2842, val = 3492;
float y, z = 2.5678e – 5, q = – 3.8469e3;
```

In the above declaration y is undefined. The other variables are assigned initial values specified. (Some compilers assign 0 to a variable when declared but it is good practice to explicitly initialize variable values). Unless a variable is defined it cannot be used in an arithmetic expression.

A constant appearing as it is in a program does not convey any meaning regarding its purpose or role to a programmer. Frequently it is attractive to associate an identifier or a name with the constant and use this name instead of the actual number everywhere the number occurs in a program. C allows defining an identifier as having constant value using #define directive. This is called a *pre-processor directive* as it is not part of a C program. This directive is placed at the beginning of a C program. # occurs in the first column. No semicolon follows #define.

For example the following pre-processor directive gives names to three different constants:

```
#define PI 3.1415927
#define MINIMUMBAL 500.0
#define MAXSPEED 200
```

The name in a #define line is formed using the syntax rules given for identifier. In addition to giving a meaningful name for a constant #define makes it easier to modify programs. Suppose a constant occurs a dozen times in a program and its value is to be changed. If the constant is declared using #define, the change is done in only one place, namely, where it is defined and not in all twelve places.

We use #define line to specify a constant when:

(i) the constant is used at many places in a program

(ii) the constant is subject to frequent change

(iii) a meaningful name for a constant would aid in understanding a program.

Even though an identifier of a constant can be composed of both upper case and lower case letters the conventions used in C is to use capital letters to specify constants to distinguish them from variables which are formed using lower case letters only.

Table 5.1 gives examples of various types of variables and constants described in this chapter.

Table 5.1 Summary of Variable Declarations and Constants

Data types	Example
int	int i, j ;
unsigned int	unsigned int k = 5 ;
long int	long int p, q ;
float	float value = 32.5602, x ;
double	double y, temp ;
decimal integer	2468
octal integer	– 02357
Hexadecimal integer	0xAFF2
long integer	267894346
integer	37842
short integer	– 46
unsigned integer	3625
floating point number	– 34.46e – 5
double precision number	0.6734679423e + 4

EXERCISES

5.1 Pick the incorrect decimal integer constants from the following list. Explain why they are incorrect.

(i) – 4689
(ii) + – 785
(iii) 4.25,325
(iv) 1/4
(v) 62 – 34 – 86
(vi) 0234
(vii) 2A45
(viii) Ox234

5.2 Pick the incorrect type declarations from the following list. Explain why they are incorrect.

(i) float, servo, mass, iota;
(ii) int servo, digit, count;
(iii) int rs.ps, unsigned;
(iv) float real, root, big;

5.3 Pick the incorrect floating point constants from the following list. Explain why they are incorrect.

(i) 40,943.65
(ii) 428.58
(iii) 46 + E2
(iv) 46E2
(v) 485. + 6
(vi) 462XE – 2
(vii) 425E2.5
(viii) .0045E + 6
(ix) 1/2.2
(x) 465.
(xi) 43

5.4 Pick the incorrect identifiers from the following list. Explain why they are incorrect.

(i) constant
(ii) variable
(iii) double
(iv) Rs-ps
(v) roll.no
(vi) lambda
(vii) lab man
(viii) int result

5.5 Classify the following constants as decimal, octal or hexadecimal:

(i) 0234
(ii) – 0456
(iii) 0xAB56
(iv) – 468734689
(v) – 0x38
(vi) 22

6. Arithmetic Expressions

Learning Objectives

In this chapter we will learn:

1. How to form arithmetic expressions using integer, float and/or double variables and constants
2. Various arithmetic operators, their precedence when they operate on variables and rules of associativity among operators of equal precedence
3. The assignment operator, assignment expressions and their use.

An arithmetic expression is a series of variable names and constants connected by arithmetic operation symbols, namely, addition, subtraction, multiplication and division. During the execution of the object program the actual numerical values stored in variable names are used, together with the operation symbols, to calculate the value of the expression. In this chapter we will discuss the syntax rules for writing arithmetic expressions and rules followed by C to evaluate them.

6.1 ARITHMETIC OPERATORS AND MODES OF EXPRESSIONS

The arithmetic operation symbols used in C and their meanings are given in Table 6.1. In Table 6.1 the symbol – is used for both *unary minus* and *subtraction*. In the expression – A + B the – symbol is a unary minus, that is, it indicates that the negative of A is to be taken. In the expression A – B, however, the – symbol indicates a binary – or subtraction. Thus in this case B is to be subtracted from A.

Table 6.1 Arithmetic Operator Symbols

Operation	Arithmetic Operator	
	For Float	For Integers
Unary minus	–	–
Division	/	/
Remainder obtained in integer division		%
Multiplication	*	*
Addition	+	+
Subtraction	–	–

The / operation for integer gives the quotient after division. Thus if we write 7/3 the answer would be 2. The % operation gives the remainder obtained in integer division. Thus 7%3 is 1. The symbol used for division of floating point numbers is /. Thus 7.0/3.0

would give 2.3333333. Observe that C does not have an operator to raise a variable to a power. Integer powers may be obtained by repeated multiplication. Thus x^4 may be written as x * x * x * x. Fractional powers have to be obtained by using a mathematical library function pow(x, y) which gives x^y. We will describe the mathematical functions available in C later in the book.

6.2 INTEGER EXPRESSIONS

Integer expressions are formed by connecting (using integer arithmetic operators) constants or variable names declared as integer or integer constant identifiers with similar quantities. For example, the following are integer expressions:

```
#define P 20
int a, b, c, d, k, p, m;
```

(i) – a + b

(ii) p/m + P

(iii) – a + b * m

(iv) (a/b – p) * (k% m * 8)

(v) – a * c + 4 – p/m * a % c * 6 + P

(Remember that p and P are different quantities)

The following are incorrect integer expressions:

```
int a, b, c, d, k, p, m;
```

(i) a – b + j (j not declared integer)

(ii) a – 4.0 (4.0 not an integer)

(iii) a + + b + c (+ + occur together)

(iv) a/b * % c (Two operator symbols, namely, * and % occur together)

(v) – a (% b + c) (Incorrect left parenthesis)

(vi) a ** b + m (Two operators symbols namely * and * occur together)

(vii) (p * (m + k) (One right parenthesis missing)

(viii) k /*m (Two operators / and * occur next to one another)

6.3 FLOATING POINT EXPRESSIONS

Floating point expressions are formed by connecting (using arithmetic operators) variable names declared as float or double with floating point (or double precision) constants or floating point variables or floating point constant identifiers. The following are a list of real expressions:

```
#define A 3.145
float x, y, z, n;
double p, j;
```

(i) – x + y

(ii) n/z + A

(iii) – z + p + n * z

(iv) (x/y – j) * (p/j * 6.0)

(v) x * p – 8.5 + p/0.8435e – 2/j + A

The following are incorrect real expressions:

```
float x, y, z, n, p, j;
```

(i) x – z + d	(d not declared float)
(ii) (x + y)(p + z)	(Missing operator. Should be written as (x + y) * (p + z) if that is intended)
(iii) z ** n	(* follows another operator *)
(iv) n . j	(. not an arithmetic operator)
(v) x/ + p	(Two operators / and + occur together)
(vi) (x + (j – p)	(Right parenthesis missing)
(vii) z p * n	(Missing operator between variables z and p)

6.4 OPERATOR PRECEDENCE IN EXPRESSIONS

In a program the value of any expression is calculated by executing one arithmetic operation at a time. The order in which the arithmetic operations are executed in an expression is based on the rules of *precedence* of operators. The precedence of operators is: unary – FIRST, multiplication (*) Division(/) and %, SECOND, Addition(+) and Subtraction(–) LAST.

For example, in the integer expression – a * b /c + d the unary – is done first, the result – a is multiplied by b, the product is divided by c (integer division) and d is added to it. The answer is thus:

$$\frac{-ab}{c} + d$$

All expressions are evaluated from *left* to *right*. All the unary negations are done *first*. After completing this the expression is scanned again from left to right; now all *, / and % operations are executed in the order of their appearance. Finally all the additions and subtractions are done starting again from the left of the expression.

For example, in the expression:

– a * b/c + d % k – m/d * k + c

the computer would evaluate – a in the first left to right scan. In the second scan

$$\frac{-ab}{c} \quad d \,\%\, k \quad \text{and} \quad \frac{m}{d} k$$

would be evaluated. In the last scan the expression evaluated would be:

$$\frac{-ab}{c} + d \% k - \frac{m}{d} k + c$$

In the float expression:

$$x * y + z/n + p + j/z$$

In the first scan

xy, z/n and j/z

are evaluated. In the second scan the expression evaluated is:

$$xy + \frac{z}{n} + p + \frac{j}{z}$$

Use of Parentheses:

Parentheses are used if the order of operations governed by the precedence rules are to be overridden.

In the expression with a single pair of parentheses the expression inside the parentheses is evaluated FIRST. Within the parentheses the evaluation is governed by the precedence rules.

For example, in the expression:

$$a * b/(c + d * k/m + k) + a$$

the expression within the parentheses is evaluated first giving:

$$c + \frac{dk}{m} + k$$

After this the expression is evaluated from left to right using again the rules of precedence giving

$$\frac{ab}{c + \frac{dk}{m} + k} + a$$

If an expression has many pairs of parentheses then the expression in the innermost pair is evaluated first, the next innermost next and so on till all parentheses are removed. After this the operator precedence rules are used in evaluating the rest of the expression.

For example, in the real expression:

$$((x * y) + z/(n * p + j) + x)/y + z$$

xy, np + j will be evaluated first.

In the next scan

$$xy + \frac{z}{np + j} + x$$

will be evaluated. In the final scan the expression evaluated would be:

$$\frac{xy + \frac{z}{np + j} + x}{y} + z$$

6.5 EXAMPLES OF ARITHMETIC EXPRESSIONS

In this section we will consider some expressions which occur frequently in practice and are improperly translated into C expressions by beginners.

Example 6.1

Consider the expression:

$$\frac{1}{1+x}$$

where x is a floating point variable.

A beginner might write the expression as 1/1 + x. This expression is syntactically correct. It does not, however, mean what the programmer intended it to mean. The computer will evaluate it using the precedence rules as (1/1) + x. Observe that this mistake in writing is an error in logic rather than in syntax and the computer cannot detect this error. It will go ahead and compute the wrong answer. The correct way (logically and syntactically) of translating the expression is 1/(1 + x). The parentheses enclosing the denominator expression are essential.

In the expression (1 + x) the constant 1 is an integer and x is declared float. An expression such as this which mixes an integer with float is called a mixed mode expression. C allows this. It automatically converts the integer to float before adding. In general a "narrower type" is converted to "broader type" to avoid losing information. Beginners should avoid mixing types as the "automatic conversion" may lead to errors which are difficult to locate. We will discuss this in greater detail later in the chapter.

Example 6.2

Consider the expression:

$$\frac{a+b}{a-b}$$

where a and b are real. A translation of this which is syntactically correct, might be a + b/ a – b

This will be evaluated as:

$$a + \frac{b}{a} - b$$

which is not what the programmer intended. The expression a + b/(a – b) is also a wrong translation as the expression evaluated would be:

$$a + \frac{b}{a-b}$$

The correct expression is (a + b)/(a – b)

The above examples illustrate the need for parentheses. In practice a good rule to follow is: "if in doubt use parentheses".

Example 6.3

Consider the expression $a^2 + b^2 - 2ab$. A careless programmer might translate this into

a * a + b * b – 2a * b. This has a syntax error. The expression 2a * b has no meaning syntactically. It is to be written as 2 * a * b. An operator is required, it is not implied. After this correction the expression is syntactically correct and does what the programmer intended it to do. The expression is:

$$a * a + b * b - 2 * a * b$$

Example 6.4

Consider a polynomial expression: $5x^4 + 3x^3 + 2x^2 + x + 10$.

A correct syntactic translation of this expression is:

$$5 * x * x * x * x + 3 * x * x * x + 2 * x * x + x + 10$$

This however, is an inefficient way of writing a polynomial for computer evaluation as it involves a total of 4 addition operations and 9 multiplication operations (verify this). Another technique of writing the polynomial requires a far lesser number of arithmetic operations and is given below:

$$\begin{aligned} 5x^4 + 3x^3 + 2x^2 + x + 10 &= 10 + x + 2x^2 + 3x^3 + 5x^4 \\ &= 10 + x(1 + 2x + 3x^2 + 5x^3) \\ &= 10 + x(1 + x(2 + 3x + 5x^2)) \\ &= 10 + x(1 + x(2 + x(3 + 5x))) \end{aligned}$$

Translated into C this is:

$$10 + x * (1 + x * (2 + x * (3 + 5 * x)))$$

The above expression requires only four multiplications and four additions. Even though this expression is efficient it is less readable. Further, errors could be made in parenthesising. Whenever there are a large number of parentheses in an expression one check is to separately count the number of left and right parentheses. These counts should be equal.

Example 6.5

Consider the following expression:

$$\frac{mv^2}{2.5} + \frac{gh}{4d}$$

A careless programmer may translate this as:

$$m * v * v/2.5 + gh/4.d$$

In the second term gh is written as it appears in the formula and is wrong. 4d is translated as 4.d which is also wrong. They should be written as g*h and 4*d respectively. After making these corrections the expression is:

$$m * v * v /2.5 + g * h / 4 * d$$

which is syntactically correct. There is still one error left.

g * h / 4 * d will be evaluated as

$$\frac{gh}{4} * d$$

The correct way of writing this would be:

$$(m * v * v /2.5) + (g * h)/(4 * d)$$

Example 6.6

A note on integer division

When an integer is divided by another integer the answer may have a fractional part. All fractions are, however, discarded during calculation with the / operator. Thus 3/4 will give an answer 0 and 5/4 an answer 1. When expressions are evaluated using precedence rules, whenever division is performed, the truncated numbers are the ones which are used in the final evaluation. Thus i/10 * 10 and 10 * i/10, even though they seem equivalent, give different answers. If i = 35, the first expression gives:

$$35/10 * 10 = 3 * 10 = 30$$

The second expression gives:

$$10 * 35/10 = 350/10 = 35$$

6.6 ASSIGNMENT STATEMENTS

An assignment statement has the following form:

Variable name = expression

For example,

total_pay = gross_pay – deduction + allowance and
rate = gross_rate – discount

are assignment statements.

Total_pay + deduction = gross_pay + allowance

is *not* an assignment statement as this has expressions on both sides of the assignment operator =

When an assignment statement is executed, the expression on the right of the assignment operator is first *evaluated* and the *number* obtained is stored in the storage location named by the variable name appearing on the left of the assignment operator.

Thus when an expression is assigned to a variable the previous value of the variable is *replaced by* the value calculated using the expression. For example, in the assignment statement b = b + 3 the integer 3 is added to the number stored in variable name b and this new value replaces the old value stored in b.

The equal sign must thus be interpreted as "is to be replaced by" rather than "is equal to" (A more appropriate symbol would have been ← instead of =)

Type conversion in statements

In the examples given in the last section the variable types used in the expression on the right hand side of the = sign was the same as the variable type on the left. This need not be so.

Rule

A variable declared float or double can be set equal to an integer expression and vice-versa.

If an integer expression is assigned to a float variable name then the value of the integer expression is converted to a floating point number and is stored in the float variable name.

For example, consider the following statements:

```
int j , k ;
float a ;
a = j/k ;
```

If j = 3 and k = 2 then j/k = 1. The variable a will be assigned the value 1.0

If a floating expression is assigned to an integer variable the value of the float expression is "truncated" and stored in the integer variable name. In other words the *fractional part* of the floating point number obtained by evaluating the expression is chopped off (not rounded!) and the integer part is stored in the integer variable name.

For example, consider the following statements:

```
float a, b ;
int p ;
p = a/b ;
```

If a = 7.2 and b = 2.0 then a/b = 3.6. When 3.6 is assigned to p it will be truncated to 3 (as p is declared int) and stored in p.

6.7 DEFINING VARIABLES

A variable is said to be *defined* if a value has been assigned to it. In other words, a variable is defined if a number has been stored in the storage location corresponding to its name.

Unless a variable is defined it cannot be used in an arithmetic expression. Values may be assigned to variables when they are declared in main ( ).

They may also be defined by the statement:

Variable name = Constant

For example x = 4.5 defines the variable x by storing 4.5 in the memory location corresponding to x.

If all the variables appearing on the right of the assignment operator in an assignment statement are defined then the variable on the left is also defined. It follows from the fact that when an assignment statement is executed the numerical value of the expression on the right of the assignment operator is evaluated and stored in a memory location corresponding to the variable on the left.

For example the statement b = a * d + e – f defines the variable b if a, d, e and f are already defined.

6.8 ARITHMETIC CONVERSION

C allows mixing of float with integers in expressions. In such a case integers are converted to float before computation. However, if there is an integer sub-expression in the expression,

it is computed using integer arithmetic. This feature is often overlooked by programmers and leads to unexpected errors. Consider the expression:

$$(1 + x)/(1 - x)$$

In the above expression 1 is converted to float and the expression is correctly computed. If we want to write an expression for i $x^2/2$ where i is an integer and x is float and write (i/2)* x * x then (i/2) is computed using integer arithmetic. Thus if i = 1 the expression will be zero which is wrong.

Consider the expression:

```
int k ;
float x , y ;
y = x * k/4 with x = 8, k = 3
```

If we write y = x * (k/4) then y will equal 8. * (3/4) = 8. * 0 = 0. If we write y = (x * k)/4 then k is converted to float first giving 24.0 for the numerator. The denominator integer 4 is converted to float and 24.0 is divided by 4.0 giving 6.0.

Thus whenever integer division is seen in a sub-expression one should re-examine the expression carefully.

It has been observed in practice that mixing integers with float can lead to mistakes which are difficult to locate. The conversion is so automatic that it is easy to overlook a problem. Thus it is better not to use this facility.

C provides *explicit type conversion* functions using which a programmer can intentionally change the type of expressions in an arithmetic statement. This is done by a unary operator called a *cast*. If we write (type-name) expression the expression is converted to the *type-name*. For example, if we write:

(float) (integer expression or variable name)

then the integer expression or variable name is converted to float. If we write:

(int) (float expression or variable name)

then the float expression or variable name is converted to integer. These functions are safer to use as the programmer intentionally uses them. Using this we can write the expression

$$i/2x^2$$

as: (float) i/2 * x * x

The expression xk/4 may be written as:

x * (float) k/4

Conversion with float and double

If in an expression real variables declared as float and double appear together all variables are converted to double. For example, in the statement:

```
float x, y, s ;
double p, q, z ;
  z = y * p/q + s ;
  x = q/(y + s) ;
```

the variable names y and s are assumed to be double. The result is double as z is double. In the second statement (y + s) is calculated using single precision. As q is double (y + s) is converted to double before division. As x is single precision, the answer is made single precision and stored in x.

6.9 ASSIGNMENT EXPRESSIONS

We have already defined the assignment operator = which assigns the value calculated for the expression on the right hand side of the operator to the variable name on the left hand side of the operator. For example, in

```
a = x + y * z ;
```

the assignment operator is =

C unlike most other languages provides a new assignment operator. For example, if we write

```
float x ;
x + = y ;
```

the expression is taken as

```
x = x + y
```

In general, if we write:

<variable name><operator> = <expression>

it is interpreted as :

variable name = (variable name) (operator) (expression)

Example 6.7 illustrates the result of applying this operator.

Example 6.7

```
float x = 4.0, y = 12.0, z = 3.5 ;
int p = 6, q = 4, r = 8 ;
x + = 3.0 ;
x * = y ;
x / = y + z ;
p % = q ;
q − = p + 2 * r + 4 ;
q / = p% r + 4 ;
```

We get as results:

```
x = x + 3.0 = 7.0
x = x * y = 7.0 * 12.0 = 84.0
x = x/(y + z) = 84.0/(12.0 + 3.5) = 5.4193548
p = p% q = 6% 4 = 2
q = q − (p + 2 * r + 4) = 4 − (2 + 16 + 4) = − 18
q = q/(p% r + 4)
  = − 18/(2% 8 + 4) = − 18/(2 + 4) = − 3
```

Assignment operator is particularly useful when a long variable name appears on the left hand side of the expression. For example,

```
sum_of_height_of_students + = height;
```

is much more concise than writing

```
sum_of_height_of_students = sum_of_height_of_students + height;
```

6.10 INCREMENT AND DECREMENT OPERATORS

C has two useful operators for incrementing and decrementing the values stored in variable names.

Thus if we write + + x then 1 is added to the contents of the variable name x. This is called incrementing x. If we write

```
y = + + x ;
```

then x is incremented by 1 and the result is stored in y. It is equivalent to the statements,

```
x = x + 1 ;
y = x ;
```

If we write – – z then 1 is subtracted from the contents of the variable z. If we write:

```
p = – – z ;
```

it is equivalent to the statement:

```
z = z – 1 ;
p = z ;
```

This is mainly a short hand notation to make it easy to write shorter programs.

There is another way in which increment and decrement can be used in C. If we write

```
y = x ++ ;
```

it means first assign x to y and *then* increment x. In other words this statement is equivalent to statements:

```
y = x ;
x = x + 1 ;
```

Similarly

```
p = z – – ; is equivalent to
p = z ;
z = z – 1 ;
```

The method of incrementing and decrementing given above are confusing to a beginner and should be used carefully.

Example 6.8

The following example illustrates the use of increment and decrement operators.

```
int x = 6, y = 8, z, w, p = 10, q ;
y = x ++ ;
z = ++ x ;
```

```
w = p - - ;
q = ++ p ;
```

The values obtained by calculating the above expressions are:

```
y = 6
x = 6 + 1 = 7
x = x + 1 = 8
z = 8
p = p - 1 = 9
w = p = 9
p = p + 1 = 10
q = 10
```

In fact the unary operators ++ and - - can change the values of operands on which they operate without the need for an assignment statement. For example, if x = 8 and we write ++ x then 8 + 1 = 9 is stored in x. In other words, x gets incremented.

6.11 MULTIPLE ASSIGNMENTS

C also has the provision to do multiple assignments. For example, we can write:

```
a = b = c = 1
```

1 will be assigned to c. The value of c will be assigned to b and the value of b to a. Thus a = 1, b = 1, c = 1.

In the statement

```
a = b = c + d - e
```

C language will first calculate c + d - e. This will be assigned to b. The value stored in b will be assigned to a. Thus an assignment b = (c + d - e) can be *thought to have a value* which is the value of b. The = operator is said to be *right associative* as we scan the statement from right to left. + and - operators in c + d - e are *left associative* as we scan the expression from left to right. We can also write:

```
a = x = b = c + d - e
```

which will be interpreted as:

```
a = (x = (b = c + d - e ))
```

If we write: a + = b * = c + d - e
it is equivalent to:

```
a = a + b = a + (b * (c + d - e ))
```

Observe that when we use the operator + = or * = the right hand side is assumed to be enclosed in parentheses.

Thus

```
x * = y + 1
```

is x = x * (y + 1) and *not* x = x * y + 1

The right to left associativity of = , * = operators is clarified by the following example. Suppose we write

```
a = (b = d * = c)
```

with c = 3, d = 4. The operators = and * = have equal precedence but are right associative. As (b = d * = c) is enclosed in parentheses and parentheses has higher precedence it will be done first. Within the parantheses = and * = have equal precedence but due to right associativity of = and * =, operation * = will be done before =. Thus d will be evaluated as d = d * c = 12. The = operator will next assign 12 to b. Thus a = b = d = 12 at the end of execution. The expression:

```
a = b = d * = c
```

will be interpreted as:

```
a = (b = ( d * = c ))
```

Observe that we have put parentheses from right to left to illustrate right associativity of = and * = operators which have the same precedence. In this case d = d * c = 12. Thus a = 12 and b = 12 at the end of execution.

SUMMARY

We have introduced many concepts in this chapter. We will summarise them now.

1. Arithmetic expressions are formed by variable names and or constants operated on by arithmetic operators.
2. In Table 6.2 we have given all arithmetic operators and their meanings.

Table 6.2 Unary and Binary Arithmetic Operators

Operation	Operator	Operands		Example	Effect
		Number	Type of both		
Unary					
minus	–	1	int or float*	– x	x ← – x
increment	++	1	int or float	++ x or x ++	x ← x + 1
decrement	– –	1	int or float	– – x or x – –	x ← x – 1
Binary					
add	+	2	int or float	x + y	Add y to x
subtract	–	2	int or float	x – y	Subtract y from x
multiply	*	2	int or float	x * y	multiply x by y
divide	/	2	int	x/y	integer quotient
divide	/	2	float	x/y	Divide x by y
modulus	%	2	int	x%y	integer remainder

*Wherever *float* is mentioned *double* may also be used.

3. An expression is evaluated by scanning it from left to right. The order of application of operators is determined by hierarchy rules. In Table 6.3 we summarise these rules.

Table 6.3 Rules of Hierarchy of Operators

Rank	Symbol	Evaluation Rule
First	( )	Evaluation expressions enclosed by parentheses, innermost parentheses first.
Second	Unary operators – –, ++, and cast, namely, (int), (float) or (double)	Right to left associativity among these unary operators.
Third	/, *, %	Division, multiplication and modulus are applied in the order in which they appear during left to right scan of expression.
Fourth	– , +	Binary – and + are done next in their order of appearance during left to right scan of expression.
Last	= , + =, * = – =, / =, % =	Right to left associativity among these operators.

4. An assignment expression is: x = (expression) the symbol = is an assignment operator. The expression is calculated and the result is stored in x. The assignment expression x = (expression) has a value equal to that assigned to x.
5. Assignment operators of the type <operator>= where <operator> is a binary operator are available in C. The statement: x + = (expression), is equivalent to x = x + (expression) and x /= (expression) is equivalent to x = x/(expression). In general x <operator> = (expression) is interpreted as x = x<operator>(expression).
6. The type of variable on the left hand side of an assignment operator need not match that of the expression on the right hand side. Table 6.4 summarises the rules when the types do not match.

Table 6.4 Type Conversion in Assignment Statement x = (exp)

Type of x	Type of (exp)	Effect
float	float or double	x ← float (exp)
double	float or double	x ← double (exp)
int	float, double	x ← int (exp)
int	int	x ← int (exp)
float	int	x ← float (exp)
double	int	x ← double (exp)

7. C allows mixing of float and integers in arithmetic expressions. In such a case integers are converted to float before computation. Integer sub-expressions are computed using integer arithmetic rules. It is not a good practice to mix integers and float. Explicit type conversion using cast operator is a better solution. If float and double variables are found in an expression then all float variables are converted to double.
8. Increment and decrement operators ++ and – – respectively are convenient to use. The effect of using these operators is shown in Table 6.5.

Table 6.5 Increment and Decrement

Operator	Example	Effect	Remarks
++	++x	x ← x + 1	Increment
– –	– –x	x ← x – 1	Decrement
++	x++	x ← x + 1	Increment
– –	x– –	x ← x – 1	Decrement
++	p = ++x	x ← x + 1 p ← x	Increment and store
– –	p = – –x	x ← x – 1 p ← x	Decrement and store
++	p = x++	p ← x x ← x + 1	Store and Increment
– –	p = x– –	p ← x x ← x – 1	Store and decrement

EXERCISES

6.1 Write C expressions corresponding to the following:
Assume all quantities are of type float.

(i) $\frac{ax + b}{ax - b}$

(ii) $\frac{2x + 3y}{x - 6}$

(iii) $x^5 + 10x^4 + 8x^3 + 4x + 2$

(iv) $(4x + 3)(2y + 2z - 4)$

(v) $\frac{ax + b}{c} + \frac{dy + e}{2f} - \frac{a}{bd}$

(vi) $\frac{a}{b(b - a)}$

6.2 What is the final value of b in the following sequence of statements?

```
float b ;
int i ;
b = 2.56 ;
b = (b + 0.05) * 10 ;
```

```
i = b ;
b = i ;
b = b/10.0 ;
```

If b = 2.56 is replaced by b = 2.54 above, what is the final value of b?

6.3 What is the value of i calculated by each of the following statements?

```
int i, j = 3, k = 6 ;
```

(i)
```
i = j * 2/3 + k/4 + 6 – j * j * j/8 ;
```

(ii)
```
float a = 1.5, b = 3.0 ;
i = b/2.0 + b * 4.0/a – 8.0
```

(iii)
```
int i, j = 3 ;
i = j/2 * 4 + 3/8 + j * j * j % 4 ;
```

6.4 Write the final value of k in the following program:

```
int k = 5, i = 3, j = 252, m ;
m = i * 1000 + j * 10 ;
k = m/1000 + k ;
k = m % k ++ ;
```

6.5 Evaluate the following expressions:

```
float a = 2.5, b = 2.5 ;
```

(i) `a + 2.5/b + 4.5`
(ii) `(a + 2.5)/b + 4.5`
(iii) `(a + 2.5)/(b + 4.5)`
(iv) `a/2.5/b`
(v) `b++ / a + b– –`

6.6 Evaluate each of the following expressions (The expressions are to be evaluated independent of one another):

```
int i = 3, j = 4, k = 2 ;
```

(i) `i ++ – j – –`
(ii) `i – – % j ++`
(iii) `j ++ / i – –`
(iv) `++ k % – – j`
(v) `k ++ * – – i`
(vi) `j + 1 / i – 1`
(vii) `i – 1 %j + 1`
(viii) `i = j * = k = 4`
(ix) `i * = k = ++ j + i`
(x) `i = j /= k + 4`

6.7 Find the value of i, j k at the end of the execution of each of the following statements:

```
int k = 3, i = 4, z = 6 ;
k + = i % j ++ ;
k % = ++ i * j /2 ;
k * = k ++ * j – i – – ;
```

6.8 Find the value of a in each of the following statements:

```
int i = 5, j = 5, k = 7;
float a = 3.5, b = 5.5, c = 2.5 ;
```

(i) `a = b – i/j + c/j`
(ii) `a = (b – i)/(j + c) /j`
(iii) `a = b – ((i + j)/(k + i)) * c`
(iv) `a = b – i + j/k + i * c`
(v) `a = b + 1% 1 + c`
(vi) `a = (b + 1) % (1 + c)`

7. Input and Output in C Programs

Learning Objectives

In this chapter we will learn:

1. How to input data to C program and display the results
2. The need for format string and how to specify a format string for a given requirement
3. How to display messages using printf statement

We have seen that any program requires data to be read into variable names. Data stored in variable names need to be displayed. As a programming language, C does not provide any input-output (I/O) statements. Instead a set of standard library functions provided by the operating system UNIX for input and output are borrowed and used by C. Compilers using other operating systems such as MS-DOS have also emulated the same idea. Some minor differences may be there which a programmer should find out from the system staff.

If no I/O device is specified C assumes that the keyboard of a video-terminal is the input device and the video screen, the output device. There are a pair of I/O functions which we will discuss in this chapter. These are: scanf( ) for input and printf ( ) for output. These are used for data to be read in a specific format and written in a specific format.

7.1 OUTPUT FUNCTION

The general form of an output function is:

printf (format-string, var_1, var_2, ... , var_n)

where format-string gives information on how many variables to expect, what type of arguments they are, how many columns are to be reserved for displaying them and any character string to be printed. The printf( ) function may sometimes display only a message and not any variable value. In the following example:

printf ("Answers are given below") ;

the format-string is:

Answers are given below

and there are no variables. This statement displays the format-string on the video display device. After displaying, the cursor on the screen will remain at the end of the string. If we want it to move to the next line to display information on the next line, we should have the format-string:

printf ("Answers are given below \n");

In this string the symbol \n commands that the cursor should advance to the beginning of the next line.

In the following example:

printf("Answer x = %d \n", x) ;

%d specifies how the value of x is to be displayed. It indicates that x is to be displayed as a decimal integer. The variable x is of type int. %d is called the *conversion specification* and d the *conversion character*. In the example:

printf(" a = %d, b = %f \n", a, b) ;

the variable a is of type int and b of type float or double. %d specifies that a is to be displayed as an integer and %f specifies that, b is to be displayed as a decimal fraction. In this example %d, and %f are conversion specifications and d, f are conversion characters.

If a = 248 and b = – 468.8643 they will be displayed as:

a = 248, b = – 468.8643

If the approximate value of b is not known it is preferable to display it in mantissa, exponent form. For this the specification is %e. When this specification is used the answer will be displayed in the form

+ m. m m m m m m e ± pp

Example Program 7.1 illustrates how printf ( ) displays answers. Observe that there is

```
/* Example Program 7.1 */
/* Program illustrating printf() */
#include <stdio.h>

main()
{
  int a = 2567, b = -467;
  float x = -123.4567, y = 12345.67;
  double z = -123.4567891234, w = 123456789.1234;
  printf("Output:\n");
  printf("12345678901234567890123456789O\n");
  printf("\n");
  printf("%d, %d, %f, %f \n", a, b, x, y);
  printf("\n");
  printf("%d, %d, %e, %e \n", a, b, x, y);
  printf("%f, %f\n", z, w);
  printf("%e, %e\n", z, w);
}

Output:
12345678901234567890123456789O

2567, -467, -123.456703, 12345.669922

2567, -467, -1.234567e+02, 1.234567e+04
-123.456789, 123456789.123400
-1.234568e+02, 1.234568e+08
```

Program 7.1 Illustrating printf statement use.

a discrepancy between the values assigned to x and y in the program and those printed by the printf statement. This is due to the fact decimal floating point numbers are stored in the machine in binary mantissa, exponent form. The number of bits used for mantissa in single precision mode in 32 bit computers is usually 22 bits. This limits the number of significant decimal digits to 6. Thus only the first 6 significant digits will be guaranteed to be correct when the number is printed. More digits will be printed depending on the format specification but they may be meaningless.

So far we specified that the variable values to be printed should be integers, decimal fractions or decimal fractions in exponent notation. We have not specified the number of columns to be reserved to display a variable. The function decides, based on the value to be displayed, the number of columns to be reserved on the display unit. We can explicitly specify the number of columns to be reserved by using the conversion specification:

% w conversion character

Here *w* is an integer which specifies the width of the field to be displayed. Example Program 7.2 illustrates this:

```
/* Example Program 7.2 */
/* Program illustrating use of width specification */
#include <stdio.h>

main()
{
  int a = 2567, b = -467;
  float x = -123.4567, y = 45645.6;
  double z = -789.4567891234, p= 3456789.1234;
  printf("Output:\n");
  printf("12345678901234567890123456789012345678901234567890\n");
  printf("a =%6d, b =%8d, x =%8f \n", a, b, x);
  printf("x =%18e, y =%18e \n", x, y);
  printf("z =%18f, p =%10e \n", z, p);
  printf("a =%-6d, b =%-8d\n", a, b);
  printf("x =%-15f, y =%-18e \n", x, y);
} /* End of main */

Output:
12345678901234567890123456789012345678901234567890
a =  2567, b =    -467, x =-123.456703
x =     -1.234567e+02, y =      4.564560e+04
z =       -789.456789, p =3.456789e+06
a =2567  , b =-467
x =-123.456703     , y =4.564560e+04

/* Comment: Observe that x = %8f gives more than
   8 column width of output. printf seems to use
   a default 6 digits after decimal point. It is
   implementation dependent */
```

Program 7.2 Use of width specification in printf.

Observe the conversion specification %-6d where sign precedes the width specifier. This specification left justifies the number printed.

A more general specification is:

%w.p conversion character

In this specification *w* is the width of the field to be reserved and *p* is the precision, that is, the number of digits to be displayed after the decimal point in a fraction. Example Program 7.3 illustrates this.

```
/* Example Program 7.3 */
#include <stdio.h>

main()
{
  float x = -123.456789, y = 45745678.9;
  double z = -789.4567801234, w = 567456789.1234;
  printf("Output:\n");
  printf("12345678901234567890123456789012345678\n");
  printf("x =%12.6f, y =%12.1f \n", x, y);
  printf("z =%19.5e, w =%23.16e\n", z, w);
  printf("x =%30.20f\n", x);
  printf("z =%30.15e\n", z);
} /* End of main */
```

```
Output:
12345678901234567890123456789012345678
x = -123.456787, y =  45745680.0
z =      -7.89457e+02, w = 5.6745678912340000e+08
x =   -123.45678710937500000000
z =       -7.894567801234000e+02
```

Program 7.3 Printing double precision numbers.

In Table 7.1 we summarise various conversion specifications and their meanings.

Table 7.1 Conversion Specifications

Type of variable	Conversion character	Quantity to be displayed	Conversion specification	Displayed quantity
integer	d	signed decimal integer	%d	– 1239
integer	u	unsigned decimal integer	%u	1238
octal	o	unsigned octal value	%o	1302
hexadecimal	x	unsigned hexadecimal	%x	A34F
floating point or double precision	f	real decimal	%f	– 345.678
floating point or double precision	e	real decimal	%e	– 3.456788e5

Table 7.2 General Conversion Specification: % – w.p conversion char

%	First character of specification (compulsory)
–	Minus sign for left justification (optional)
w	Digits specifying field width (optional)
.	Period separating width from precision (optional)
p	Digits specifying precision (optional)
d, u, o, x, f or e	Conversion character (compulsory)
l	The letter (ell) if integer is a long integer

7.2 INPUT FUNCTION

The function scanf ( ) is used to read data into variables from the standard input, namely, a keyboard attached to a video terminal. The general format of an input statement is:

scanf (format-string, var_1, var_2, ... , var_n)

where format-string gives information to the computer on the type of data to be stored in the list of variables var_1, var_2...var_n and in how many columns they will be found.

For example, in the statement:

```
scanf ( "%d%d", &p, &q) ;
```

the two variables in which numbers are to be stored are p and q. The data to be stored are integers. The integers will be separated by a blank in the data typed on the keyboard.

A sample data line may thus be:

```
456 58578
```

Observe that the symbol & (called ampersand) should precede each variable name. Ampersand is used to indicate that the address of the variable name should be found to store a value in it. The manner in which data is read by a scanf statement may be explained by assuming an arrow to be positioned above the first data value. The arrow moves to the next data value after storing the first data value in the storage location corresponding to the first variable name in the list. A blank character should separate the data values.

This is illustrated below:

```
scanf ( "%d%d%d%d", &p, &q, &s, &t)
```

Data line:

```
initial                                  final
arrow                                    arrow
position                                 position
   ↓                                        ↓
  258        454        682        743
   p          q          s          t
```

The statement could also have been written as:

```
scanf ( "%d%d", &p, &q);
scanf ("%d%d", &s, &t);
```

with the data line as shown above.

The scanf statement causes data to be read from one or more lines till numbers are stored in all the specified variable names.

Observe that no blanks should be left between % and characters such as *d* in the format-string. Writing scanf("% d %d", &p, &q) is thus wrong. Another common mistake is to forget the symbol & in front of the variable name. *This symbol & is essential.*

If some of the variables in the list of variables in scanf are of type integer and some are float, appropriate descriptions should be used in the format-string. The example

```
scanf ("%d%f%e", &a, &b, &c);
```

specifies that an integer is to be stored in a, float is to be stored in b and a float written using the exponent format in c. The appropriate sample data line is:

```
485 498.762 6.845e–12
```

In Table 7.3 we specify the character to be used in the format-string for various types of data.

Table 7.3 Characters to be Used in Format-String for Various Types of Data

Character after % in format-string	Type of data to be entered via keyboard
d	integer
u	unsigned integer
o	octal integer
x	hexadecimal integer
f	floating point with decimal point in number
e	floating point expressed in exponent notation.

We give below some examples of scanf statements and the corresponding data line.

Examples

(i) scanf ("%d%u%o%e", &x, &y, &z, &p);
Data line:
– 258 4578 0234 – 28.65e–12

(ii) scanf ("%f%x%d%e", &p, &q, &v, &s);
Data line:
–256.42 AF06 358 –456.78e14

Example Program 7.4 illustrates how scanf and printf statements work together. Example Program 7.5 illustrates the use of printf with width specifications for the data to be displayed.

In C both scanf and printf are functions. The name of the function scanf and printf have values assigned to them when they complete their job. The scanf function has an integer value equal to the number of data converted and stored by it. If it tries to read data after the end of data is reached then it has a value EOF (called end of file). EOF is defined in stdio.h (EOF is normally –1 but it may be implementation dependent).

```
/* Example Program 7.4 */
/* Program illustrating use of scanf and printf
   statements  */
#include <stdio.h>

main()
{
  int a, b, c, d;
  float x, y, z, p;
  scanf("%d %o %x %u", &a, &b, &c, &d);
  printf("The first four data displayed\n");
  printf("%d %o %x %u\n", a, b, c, d);
  scanf("%f %e %e %f", &x, &y, &z, &p);
  printf("Display of the rest of the data read\n");
  printf("%f %e %e %f\n", x, y, z, p);
  printf("End of display\n");
}

Input:
-684 0362 abf6 3856 -26.68 2.8e-3 1.256e22 6.856

Output:

The first four data displayed
-684 362 abf6 3856
Display of the rest of the data read
-26.680000 2.800000e-03 1.256000e+22 6.856000
End of display
```

Program 7.4 Use of scanf statements.

The printf function has an integer value equal to number of data item printed. If it encounters an error while printing it has EOF stored in it.

```
/* Example Program 7.5 */
#include <stdio.h>

main()
{
  float a, b;
  int c, d;

  scanf("%f %f %d %d", &a, &b, &c, &d);
  printf("Output displayed\n");
  printf("----------------\n");
  printf("a =%15f b =%16.8e\n", a, b);
  printf("c =%10d d =%d\n", c, d);
}
```

```
Input:
28.467 -62.467 8345 -248

Output displayed
----------------
a =       28.466999 b = -6.24669991e+01
c =       8345 d =-248
```

Program 7.5 Use of scanf and printf statements.

EXERCISES

7.1 Write printf statements to output the following:

(i) int a, b, c
(ii) float x, y, z
(iii) unsigned int a, b, c
(iv) float p, q, r in exponent format
(v) int a, b, c with integers left justified
(vi) int a, b, c with field widths of 8 left justified
(vii) float p, q, r with 6 decimal digits after the decimal point
(viii) int i, j, k in octal form
(ix) int p, q, r with 5 decimal digits after the decimal point, left justified
(x) int p, q, r in hexadecimal form.

7.2 Write printf statements to print the following table:

Case	x	y	z
1	25.25	38.42	61.4256
2	30.25	42.856	323.468
3	725.68	734.467	854.678

7.3 Write printf statements to print the following:

x = 468.7e25 y = 256 z = – 256.725

7.4 Write scanf statements to read:

(i) Three integers, a floating point number and an octal number.
(ii) Two floating point numbers in exponent form, a hexadecimal number and an octal number.
(iii) Two negative integers and three floating point numbers.

7.5 A table of floating point numbers with 5 numbers in each line and 5 lines is given. Write scanf statements to read them.

8. Conditional Statements

Learning Objectives

In this chapter we will learn:

1. The need for conditional statements and how they are expressed in C
2. The concept of a compound statement also known as a block in C
3. The need to be careful in nesting conditional statements.

The order in which the statements are written in a program is extremely important. Normally the statements are executed sequentially as written in the program. In other words when all the operations specified in a particular statement are executed, the statement appearing on the next line of the program is taken up for execution. This is known as the *normal flow of control*. If one were restricted to this normal flow of control it would not be possible to perform alternate actions based on the result of testing a condition.

Example 8.1

Suppose we introduce in Example 4.3 a rule that a discount of 10 percent is given on purchases of mangoes greater than five dozens. To implement this rule we need a statement which will test the quantity purchased to determine if it is more than five dozens. Such a statement is called a *conditional statement*. Using this statement Example Program 4.6 is rewritten as Example Program 8.1.

The new statement introduced is:

if (quantity > 60) discount = 0.1 ;

The statement is interpreted as:

if quantity is greater than 60 then execute the statement discount = 0.1 (i.e., set discount = 0.1), otherwise *do not* execute the statement discount = 0.1.

Thus in Example Program 8.1 after the scanf statement, discount is set to zero. The next statement sets discount to 0.1 if the quantity of the purchase is greater than 60. The next statement computes the cost (in paise) with discount = 0.1 if quantity > 60 and discount = 0 otherwise.

In the statement:

if (quantity > 60) discount = 0.1 ;

we have used a *relational operator* which compares two quantities. We will discuss next the relational operators available in C.

```
/* Example Program 8.1 */
#include <stdio.h>

main()
{
  int rupees, paise, quantity;
  float cost_dozen, discount, cost;
  scanf("%f %d", &cost_dozen, &quantity);
  printf("Cost of dozen = %f, quantity = %d\n",
          cost_dozen, quantity);
  discount = 0.0;
  if(quantity > 60)
    discount = 0.1;
  cost = ((cost_dozen*100.0/12.0) * (float)quantity
          * (1.0 - discount) + 0.5);
  rupees = (int)cost/100;
  paise = (int)cost%100;
  printf("Number of mangoes = %d\n", quantity);
  printf("Cost = Rs.%d ps.%d\n", rupees, paise);
}
```

Program 8.1 Price of mangoes with discount.

8.1 RELATIONAL OPERATORS

Table 8.1 lists all the relational operators available in C.

Table 8.1 List of Relational Operators

Operator	Meaning
==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to

Two real or integer variables, constants or expressions connected by a relational operator returns a value which can be either *true* or *false*. For example, the expression n < k can be true or false. Similarly the expression index == 4 would be true if index has a value 4, else it will be false. The symbol == is a relational operator used to compare the values of variables (or constants) whereas the symbol = is used to signify the operation of assignment. Thus a = b means replace the contents of a by the contents of b whereas a == b checks for the arithmetic equality of a and b and returns a value *true* or *false*.

An expression formed with relational operators is known as *logical expression*. Some valid logical expressions are given below:

(i) a >= b

(ii) k != b

(iii) (a + b/c) < (c + d + f)

(iv) a == x

(v) x <= 0.005

In general two integer or floating point expressions may be connected by a relational operator. Float and integers should not be mixed in a logical expression.

Some illegal logical expressions are given below:

(i) int k ; 2.5 < k (An integer cannot be compared with a real quantity)

(ii) (a + b) << (c + d) (<< not a valid relational operator)

(iii) 2.5 = A (= not legal. Should write ==)

(iv) a =< b (=< is not a valid operator ; <= is the correct operator)

The value returned by a logical expression, as we have seen, can be either *true* or *false*. In C *true* is assigned a non zero integer value and *false* a value 0. Thus we can also use in an *if* statement an arithmetic expression instead of a logical expression. If the arithmetic expression is non zero the *true* branch is taken and if it is zero the *false* branch is taken.

8.2 COMPOUND STATEMENT

The compound statement (also known as a *block*) is a group of statements enclosed by braces, namely, {and}. The individual statements enclosed by braces are executed in the order in which they appear. The general form of the compound statement is:

```
{ S1 ;
  S2 ;
  .
  .
  .
  Sn; }
```

Observe that no ; (semicolon) is used after right braces.

For example:

```
{ scanf ( " %f%f", &a, &b) ;
        c = a + b ;
  printf ( " %f, %f, %f", a, b, c);}
```

is a compound statement. Observe that a semicolon separates one statement from the next. More than one statement may be placed on the same line as a semicolon is a statement separator.

8.3 CONDITIONAL STATEMENTS

The statement: *if (quantity > 60) discount = 0.1 ;* is known as a *conditional statement.* This is a simpler version of a general conditional statement available in C. The general conditional statement is:

```
if (logical expression)
   { S1t;
     S2t;
     ....
     ....
     Snt ; }
else
   { S1e ;
     S2e;
     ...
     ....
     Sme ;} /* End of if */
```

In the above general form *if* the logical expression is *true* then the compound statement {S1t; S2t; ... Snt;} is executed and control jumps to the next statement following the *if* statement. Thus the compound statement following *else*, namely, {S1e; S2e; Sme;} is ignored. If the logical expression is *false* then the compound statement appearing first is ignored and the compound statement {S1e; S2e; Sme;} following *else* is executed. As we pointed out earlier C allows use of an arithmetic expression instead of a logical expression in an *if* statement. A non zero value of an arithmetic expression is taken as *true.*

In case only one statement is to be executed if the logical expression is *true* and another single statement if it is *false* then the statement may be written as:

if (logical expression) St ; else Se ;

Observe that in the above case braces are not needed as *St* and *Se* are single statements. Another particular form of the conditional statement is:

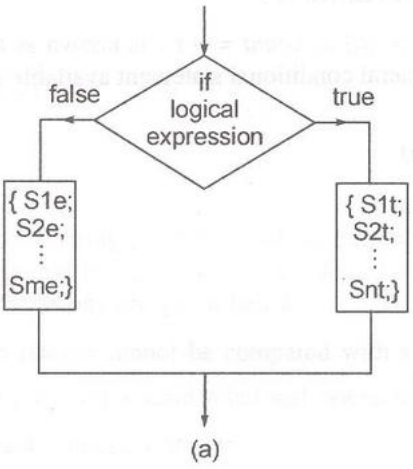
```
if (logical expression)
   { S1t ;
     S2t ;
     ....
     ....
     Snt;  }
```

In the above case if the *logical expression* is *true* then the compound statement *{S1t ; S2t ; ...; Snt;}* is executed. If the *logical expression* is *false* then control jumps to the statement following the right braces ignoring the compound statement.

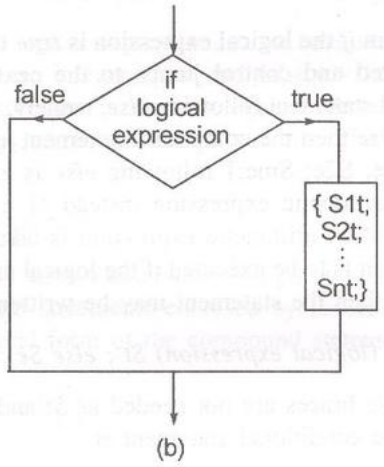
These two conditional statements are illustrated by the flow charts of Fig. 8.1(a) and Fig. 8.1(b).

A number of valid conditional statements are given below:

(i) `if (a >= b) b = c + d * x ;`



(a)



(b)

Fig. 8.1 Flow charts illustrating conditional statements.

(ii) `if (n == 50) x = x + 2 ; else x = x + 4 ;`

(iii) `if (z >= a - b + c) {x = x + 1 ; y = y + 2 ; }`

(iv)
```
if (sexcode == 1)
        {++ total_boys ;
          total_boy_height  += height ;}
else
        {++ total_girls ;
          total_girl_height += height; }
```

(v)
```
if ( a > b )
        if ( c > d )
          x = y ;
```

(vi) `if (a > b) if (c > d) x = y; else x = z ; else x = w;`

In Example 8.2(v) the first *if* clause has in turn another *if* clause. This is allowed. The way it is interpreted is shown in the flow chart of Fig. 8.2. Example 8.2(vi) is another

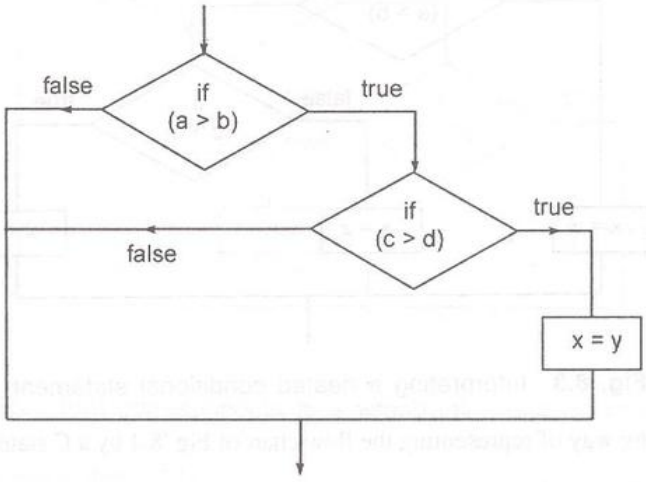


Fig. 8.2 Flow chart for the nested conditional statement of Example 11.3(v).

example of a nested *if else* clause. Such a statement is interpreted by bracketing the innermost *if* clause. Thus we would interpret Example 8.2(vi) as:

```
if (a > b)
        if (c > d)
                x = y ;
        else
                x = z ;
else
        x = w ;
```

Such an indenting is recommended as a good style and one should make it a habit to use such an indenting which makes it easy to understand the statement. A flow chart corresponding to this statement is given in Fig. 8.3.

C allows any number of nested conditional statements. One should, however, use this facility with care as understanding many levels of nested conditional statements is not easy.

We give below some illegal conditional statements.

Example 8.3

(i) if a >= b x = y ; else x = z; (parentheses missing)

(ii) if (x <= z) ; { a = b + c – d ; w = b + c + d ;} (; after if (x < = z) is wrong)

(iii) if (a != b) {x = y + z ; r = s + t ; } ;
else {x = y – z ; y = s – t ; } (; before else incorrect)

(iv) if (x = y) i = 1 else if (x != y) i = 2; (= not a relational operator

(v) if (x >= y) if (x <= z) else i = 4 ; (inner if has statement missing)

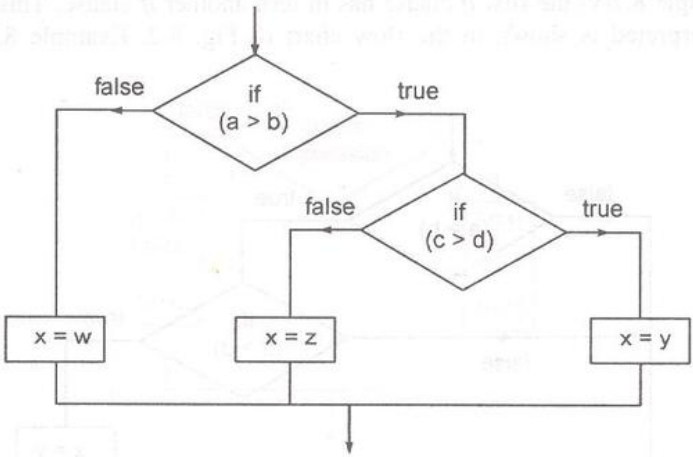


Fig. 8.3 Interpreting a nested conditional statement.

The following way of representing the flow chart of Fig. 8.4 by a C statement is incorrect:

```
if ( a > b )
        if ( c > d )
                x = y ;
else
                x = z ;
```

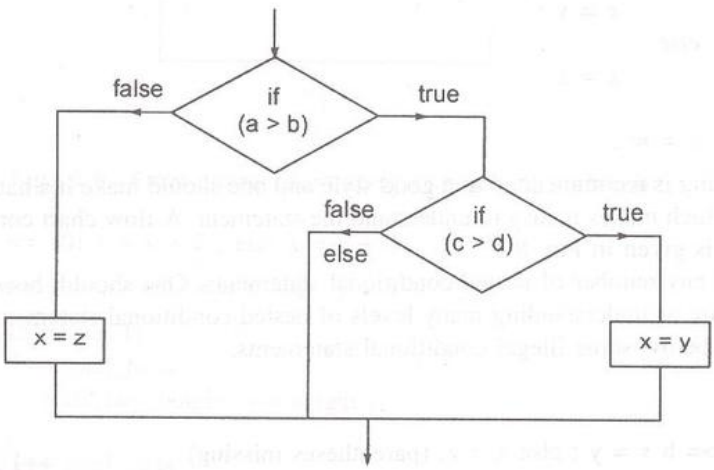


Fig. 8.4 A flow chart to be converted to a C statement.

The indenting in the above statement suggests that the *else* corresponds to *if (a > b)*. This interpretation is wrong. An *else* is always paired with its logically nearest *if*. The indenting is misleading. The flow chart corresponding to this statement is given as Fig. 8.5. A better

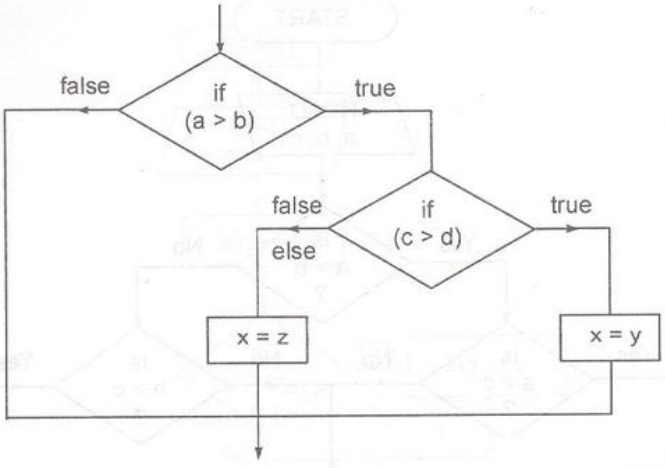


Fig. 8.5 Flow chart corresponding to a wrong interpretation of a C statement.

style of writing this statement is:

```
if (a > b)
    {if (c > d)
        x = y ;
    else
        x = z ; }
```

If the flow chart of Fig. 8.4 is to be written in C the correct statement is:

```
if ( a > b )
    {if ( c > d )
        x = y ; }
else
    x = z ;
```

8.4 EXAMPLE PROGRAMS USING CONDITIONAL STATEMENTS

Example 8.2

Consider the algorithm for picking the largest of three numbers developed in Chapter 2. The flow chart corresponding to this algorithm is reproduced as Fig. 8.6. A program corresponding to this flow chart is given as Example Program 8.2. Observe the indentation and the use of { } appropriately which makes the program easy to understand.

Example 8.3

An alternate strategy for solving the same problem which can be generalized to pick the largest of an arbitrary set of numbers (not merely 3) is given in the flow chart of Fig. 8.7. A program corresponding to this is given as Example Program 8.3. Observe that this program

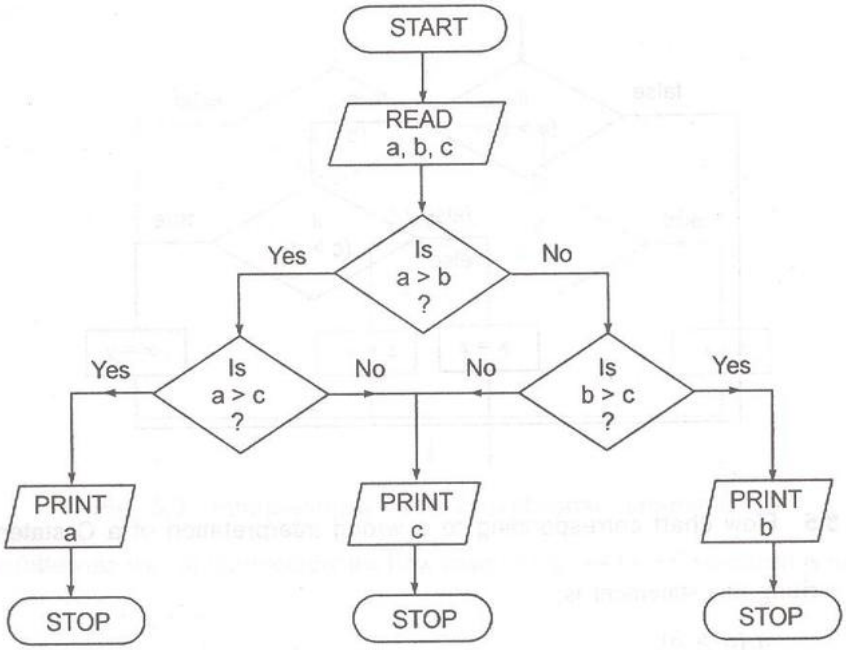


Fig. 8.6 Flow chart to pick the largest of three numbers.

```
/* Example Program 8.2 */
/* Picking largest of 3 numbers */
#include <stdio.h>

main()
{
  int a, b, c;
  scanf("%d %d %d", &a, &b, &c);
  printf("a = %4d, b = %4d, c = %4d\n", a, b, c);
  if (a > b)
    {
      if (a > c)
        printf("a = %4d\n", a);
      else
        printf("c = %4d\n", c);
    }
  else
    {
      if (b > c)
        printf("b = %4d\n", b);
      else
        printf("c = %4d\n", c);
    }
}
```

Program 8.2 Picking largest of three numbers—method 1.

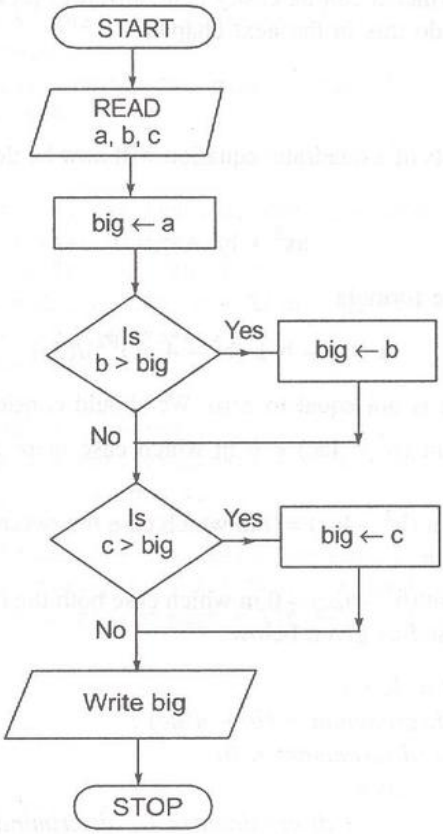


Fig. 8.7 Flow chart illustrating an alternate strategy to pick the largest of three numbers.

```
/* Example Program 8.3 */
/* Alternate method to pick the largest of three
   numbers */
#include <stdio.h>

main()
{
  int a, b, c, big;
  scanf("%d %d %d", &a, &b, &c);
  printf("a = %4d, b = %4d, c = %4d\n", a, b, c);
  big = a;
  if (b > big)
    big = b;
  if (c > big)
    big = c;
  printf("The largest number is %4d\n", big);
}
```

Program 8.3 Picking largest of three numbers—method 2.

is easy to understand. Further it can be easily generalized to pick the largest of an arbitrary set of numbers. We will do this in the next chapter.

Example 8.4

A program to find the roots of a quadratic equation will now be developed. Given a quadratic equation

$$ax^2 + bx + c = 0$$

the roots are given by the formula

$$x = \{ - b \pm (b^2 - 4\ ac)^{1/2}\}/(2a)$$

if we assume that *a* is not equal to zero. We should consider three cases:

(i) The discriminant $(b^2 - 4ac) < 0$ in which case there are two complex conjugate roots.

(ii) The discriminant $(b^2 - 4ac) = 0$ in which case the two roots are real and equal. The roots are $- b/2a$.

(iii) The discriminant $(b^2 - 4ac) > 0$ in which case both the roots are real. The algorithm may be expressed as given below:

```
Read a, b, c ;
    discriminant = (b^2 - 4 ac) ;
    if (discriminant < 0)
        then
            { discriminant = ( - discriminant);
              Imaginary part of root = (discriminant)^1/2/2a
              Real part of root = - b/2a;
              Write out the complex conjugate roots
    else
              If (discriminant == 0)
                  then
                      { root = - b/2a ;
                        Write out repeated roots}
              else
                      { root1 = - b + (discriminant)^1/2/2a ;
                        root2 = - b - (discriminant)^1/2/2a ;
                        Write out the two real roots}
              end of algorithm.
```

The above algorithm is translated to C in Example Program 8.4. Observe the ease of translating the algorithm. Note the indentation of the *if* statement to aid program understanding. Also note the placement of semicolons and curly brackets in the *if* statement. A pre-processor line #include <mathlib.h> has been placed in this program. This library is necessary to calculate the root of a number.

```
/* Example Program 8.4 */
/* Solution of quadratic equation */
#include <stdio.h>
/* math library needed for square root */
#include <math.h>
main()
{
  float a, b , c, discrmnt, x_imag_1, x_imag_2,
  x_real_1, x_real_2, temp;
  scanf("%f %f %f", &a, &b, &c);
  printf("a = %f, b = %f, c = %f\n", a, b, c);
  discrmnt = b*b - 4.0*a*c;
  if (discrmnt < 0)
    {
      discrmnt = -discrmnt;
      x_imag_1 = sqrt(discrmnt) / (2.0 * a);
      x_imag_2 = -x_imag_1;
      x_real_1 = -b / (2.0 * a);
      printf("Complex conjugate roots\n");
      printf("real part = %16.8e\n", x_real_1);
      printf("imaginary part = %16.8e\n", x_imag_1);
    }
  else
    {
      if (discrmnt == 0)
        {
          x_real_1 = -b / (2.0 * a);
          printf("Repeated roots\n");
          printf("Real roots = %16.8e\n", x_real_1);
        }
      else
        {
          temp = sqrt(discrmnt);
          x_real_1 = (-b + temp) / (2.0 * a) ;
          x_real_2 = (-b - temp) / (2.0 * a);
          printf("Real roots\n");
          printf("real root_1 = %16.8e\n", x_real_1);
          printf("real root_2 = %16.8e\n", x_real_2);
        }
    }
} /* End of main */
```

Program 8.4 Solution of quadratic equation.

8.5 STYLE NOTES

The most important style rule in writing conditional statements is indentation. We have illustrated the recommended indentation in all the examples in this chapter.

It is preferable to use an "open style" leaving blank spaces between the different parts of the statement to aid readability.

Even though C allows any number of levels of nested conditional statements we do not recommend more than three levels of nesting as it becomes difficult to understand.

EXERCISES

8.1 Given a point (x, y) write a program to find out if it lies on the x-axis, y-axis or at the origin, namely, (0, 0).

8.2 Extend the program of Exercise 8.1 to find whether it lies in the first, second, third or fourth quadrant in x – y plane.

8.3 Given a point (x, y) write a program to find out whether it lies inside, outside or on a circle with unit radius and centre at (0, 0).

8.4 Given a 4-digit number representing a year write a program to find out whether it is a leap year.

8.5 Given the four sides of a rectangle write a program to find out whether its area is greater than its perimeter.

8.6 Given a triangle with sides a, b, c write a program to find whether it is an isosceles triangle.

8.7 Given three points (x_1, y_1), (x_2, y_2) and (x_3, y_3) write a program to find out whether they are collinear.

8.8 Given three numbers A, B and C write a program to print their values in an ascending order. For example if A = 10, B = 11 and C = 6 your program should print out:

Smallest number = 6
Next higher number = 10
Highest number = 11

8.9 Write a program to round a positive number greater than 1 but less than 2 with four significant digits to one with three significant digits. For example:

1.452 rounded would give 1.45
1.458 rounded would yield 1.46
1.455 rounded would be 1.46
1.445 rounded would be 1.44

8.10 A bank accepts deposits for one year or more and the policy it adopts on interest rate is as follows:

(i) If a deposit is less than Rs. 1,000 and for 2 or more years the interest rate is 5 percent compounded annually.
(ii) If a deposit is Rs. 1,000 or more but less than Rs. 5,000 and for 2 or more years the interest rate is 7 percent compounded annually.
(iii) If a deposit is more than Rs. 5,000 and is for 1 year or more the interest is 8 percent compounded annually.

(iv) On all deposits for 5 years or more interest is 10 percent compounded annually.
(v) On all other deposits not covered by the above conditions the interest is 3 percent compounded annually.

At the time of withdrawal a customer data is given with the amount deposited and the number of years the money has been with the bank. Write a program to obtain the money in the customer's account and the interest credited at the time of withdrawal. (Hint: Use of a decision table is recommended).

9. Implementing Loops in Programs

Learning Objectives

In this chapter we will learn:

1. How to set up loops in programs
2. How to use *while* loop, *for* loop and *do while* loop in C programming.

Consider Example Program 4.7 which is reproduced here as Example Program 9.1 for ready reference. We see a repetitive structure in this program. The statements:

```
/* Example Program 9.1 */
#include <stdio.h>

main()
{
  int digit_1, digit_2, digit_3, digit_4, digit_5;
  unsigned int sum, number, n;

  scanf("%d", &number);
  printf("number = %d\n", number);
  n = number;
  digit_1 = n % 10;
  n = n / 10;
  digit_2 = n % 10;
  n = n / 10;
  digit_3 = n % 10;
  n = n / 10;
  digit_4 = n % 10;
  n = n /10;
  digit_5 = n;
  sum = digit_1 + digit_2 + digit_3 + digit_4 + digit_5;
  printf("sum of digits = %d\n", sum);
} /* End of main */
```

Program 9.1 Adding digits of a number

```
digit_1 = n% 10 ;
n = n/10 ;
```

are executed again and again. The first statement finds the least significant digit of *n* and assigns it to digit_1. The second statement reduces the length of *n* by one digit by discarding

its least significant digit. The same two operations are repeated four times till all digits in *n* are exhausted. It is thus clear that if we repeatedly execute these two statements while n! = 0 (i.e. as long as n! = 0) and add the digit obtained at each step to the sum of digits we would get the answer. This is shown in the flow chart of Fig. 9.1. Observe that the steps digit =

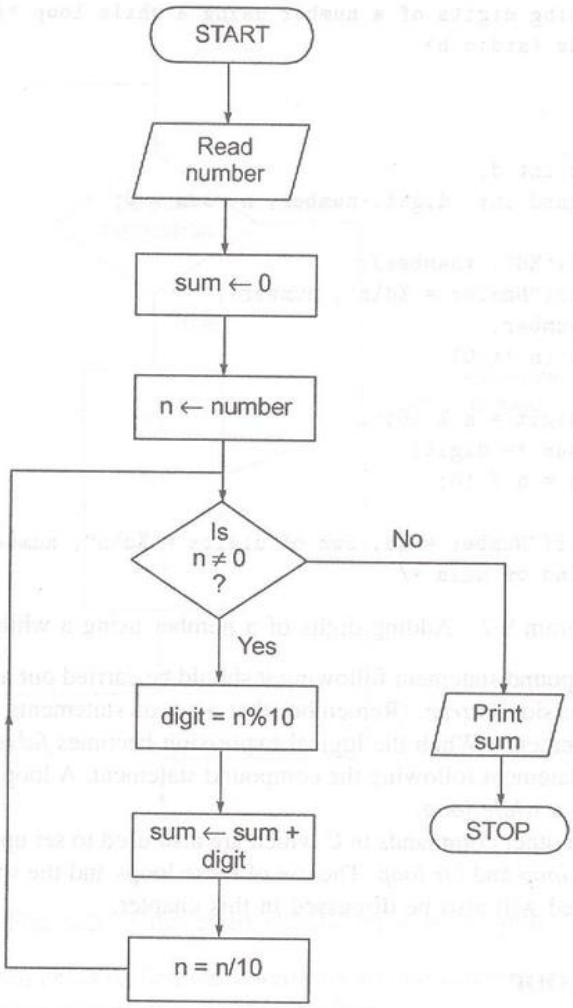


Fig. 9.1 Illustrating use of a loop in a program.

n%10, sum = sum + digit and n = n/10 are carried out again and again using a program loop as long as n! = 0 . When *n* becomes 0 control leaves the loop and executes the steps *Print* sum and *STOP*. Reformulating the strategy of finding the sum of digits using a program loop has made the program concise and at the same time it has been *generalized*. This method is general as it will add the digits of a number regardless of the number of digits in it. Contrast this with the method used in Example Program 9.1 which is valid only for a five digit number.

As such program loops are found very useful in evolving concise and generalized programs, C language provides very powerful commands for repeatedly performing a set of

statements. Example Program 9.1 is rewritten as Example Program 9.2 using a C command called *while*. Example Program 9.2 is directly derived from the flow chart of Fig. 9.1. The command:

while (logical expression)

```
/* Example Program 9.2 */
/* Summing digits of a number using a while loop */
#include <stdio.h>

main()
{
  short int d;
  unsigned int  digit, number, n, sum = 0;

  scanf("%d", &number);
  printf("Number = %d\n", number);
  n = number;
  while (n != 0)
    {
      digit = n % 10;
      sum += digit;
      n = n / 10;
    }
  printf("Number = %d, Sum of digits = %d\n", number, sum);
}  /* End of main */
```

Program 9.2 Adding digits of a number using a while loop.

orders that the compound statement following it should be carried out again and again as long as the logical expression is *true*. (Remember that a set of statements enclosed in braces {} is a compound statement.) When the logical expression becomes *false* then control gets out of the loop to the statement following the compound statement. A loop set up using the *while* command is called a *while loop*.

There are two other commands in C which are also used to set up program loops. These are called *do while loop* and *for loop*. The use of these loops and the specific occasions when these should be used will also be discussed in this chapter.

9.1 THE *while* LOOP

The general form of the *while* command is:

while (logical expression)
{ s_1 ;
s_2 ;
s_3 ;
.
.
s_n ; }
s_f;

where s_1, s_2, s_3 .... s_n are C statements. The *while* command orders that statements s_1, s_2 ... s_n enclosed by braces { } are to be executed again and again as long as the *logical expression* is *true*. When the *logical expression* becomes *false* the program jumps to the statement s_f. The flow chart of Fig. 9.2 illustrates the functioning of the *while loop*. Observe that if the logical

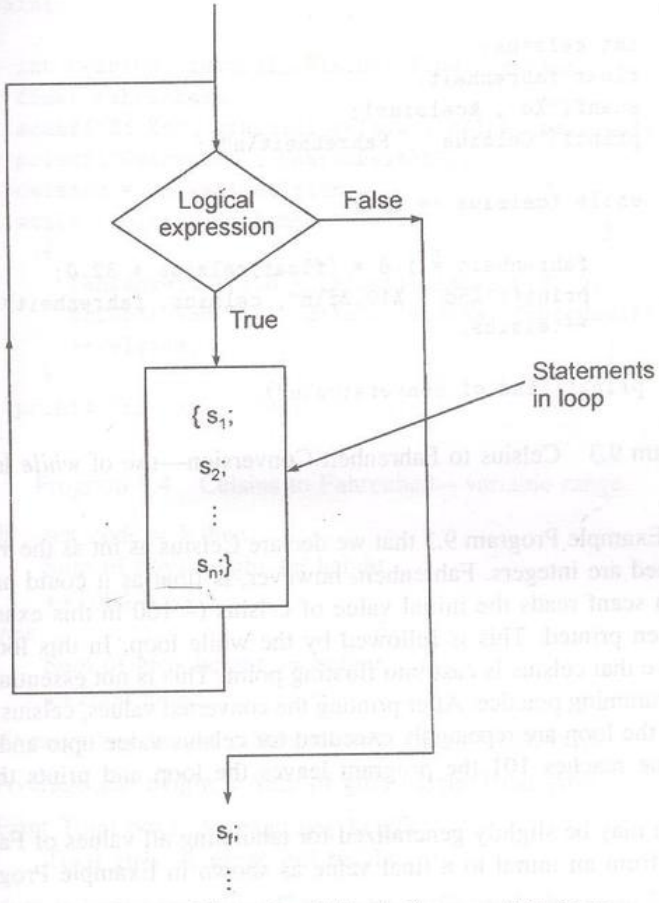


Fig. 9.2 Flow chart illustrating a *while* loop.

expression is false when checked first the statements are not carried out at all even once and the program jumps to the statement s_f outside the loop.

We will now give a few examples of using a *while* loop.

Example 9.1

Consider Example Program 4.3 which converts Celsius to Fahrenheit temperature. In that program only one Celsius temperature is read and is converted to Fahrenheit. Assume we want to convert Celsius to Fahrenheit for Celsius = –100, –99, –98, ... 0, 1, 2, 3, ... 100. In other words we want to tabulate Fahrenheit equivalent of Celsius temperatures of (– 100 to + 100 in steps of 1 degree Celsius). Example Program 9.3 illustrates how a *while* loop may be used to do this.