1 Programming Methodologies

Programming methodologies deal with different methods of designing programs. This will teach you how to program efficiently. This book restricts itself to the basics of programming in C and C++, by assuming that you are familiar with the syntax of C and C++ and can write, debug and run programs in C and C++. Discussions in this chapter outline the importance of structuring the programs, not only the data pertaining to the solution of a problem but also the programs that operates on the data.

Data is the basic entity or fact that is used in calculation or manipulation process. There are two types of data such as numerical and alphanumerical data. Integer and floating-point numbers are of numerical data type and strings are of alphanumeric data type. Data may be single or a set of values, and it is to be organized in a particular fashion. This organization or structuring of data will have profound impact on the efficiency of the program.

1.1. AN INTRODUCTION TO DATA STRUCTURE

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.

Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program.

Algorithm + Data Structure = Program

Data structures are the building blocks of a program; here the selection of a particular data structure will help the programmer to design more efficient programs as the complexity and volume of the problems solved by the computer is steadily increasing day by day. The programmers have to strive hard to solve these problems. If the problem is analyzed and divided into sub problems, the task will be much easier *i.e.,* divide, conquer and combine.

A complex problem usually cannot be divided and programmed by set of modules unless its solution is structured or organized. This is because when we divide the big problems into sub problems, these sub problems will be programmed by different programmers or group of programmers. But all the programmers should follow a standard structural method so as to make easy and efficient integration of these modules. Such type of hierarchical structuring of program modules and sub modules should not only reduce the complexity and control the flow of program statements but also promote the proper structuring of information. By choosing a particular structure (or data structure) for the data items, certain data items become friends while others loses its relations.

The representation of a particular data structure in the memory of a computer is called a storage structure. That is, a data structure should be represented in such a way that it utilizes maximum efficiency. The data structure can be represented in both main and auxiliary memory of the computer. A storage structure representation in auxiliary memory is often called a file structure.

It is clear from the above discussion that the data structure and the operations on organized data items can integrally solve the problem using a computer

Data structure = Organized data + Operations

1.2. ALGORITHM

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called *stepwise refinement method*. There are two approaches for algorithm design; they are *top-down* and *bottom-up* algorithm design.

1.3. STEPWISE REFINEMENT TECHNIQUES

We can write an informal algorithm, if we have an appropriate mathematical model for a problem. The initial version of the algorithm will contain general statements, *i.e.,*; informal instructions. Then we convert this informal algorithm to formal algorithm, that is, more definite instructions by applying any programming language syntax and semantics partially. Finally a program can be developed by converting the formal algorithm by a programming language manual.

From the above discussion we have understood that there are several steps to reach a program from a mathematical model. In every step there is a refinement (or conversion). That is to convert an informal algorithm to a program, we must go through several stages of formalization until we arrive at a program — whose meaning is formally defined by a programming language manual — is called stepwise refinement techniques.

There are three steps in refinement process, which is illustrated in Fig. 1.1.

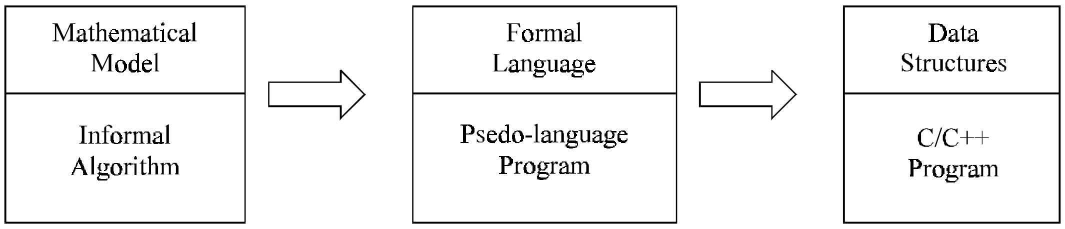

Fig. 1.1

1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.
2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.
3. In the final stage we choose an implementation for each abstract data type and write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

Following sections will discuss different programming methodologies to design a program.

1.4. MODULAR PROGRAMMING

Modular Programming is heavily procedural. The focus is entirely on writing code (functions). Data is passive in Modular Programming. Any code may access the contents of any data structure passed to it. (There is no concept of encapsulation.) Modular Programming is the act of designing and writing programs as functions, that each one performs a single well-defined function, and which have minimal interaction between them. That is, the content of each function is cohesive, and there is low coupling between functions.

Modular Programming discourages the use of control variables and flags in parameters; their presence tends to indicate that the caller needs to know too much about how the function is implemented. It encourages splitting of functionality into two types: "Master" functions controls the program flow and primarily contain calls to "Slave" functions that handle low-level details, like moving data between structures.

Two methods may be used for modular programming. They are known as top-down and bottom-up, which we have discussed in the above section. Regardless of whether the top-down or bottom-up method is used, the end result is a modular program. This end result is important, because not all errors may be detected at the time of the initial testing. It is possible that there are still bugs in the program. If an error is discovered after the program supposedly has been fully tested, then the modules concerned can be isolated and retested by them.

Regardless of the design method used, if a program has been written in modular form, it is easier to detect the source of the error and to test it in isolation, than if the program were written as one function.

1.5. TOP-DOWN ALGORITHM DESIGN

The principles of top-down design dictates that a program should be divided into a main module and its related modules. Each module should also be divided into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are intrinsically understood and cannot be further subdivided.

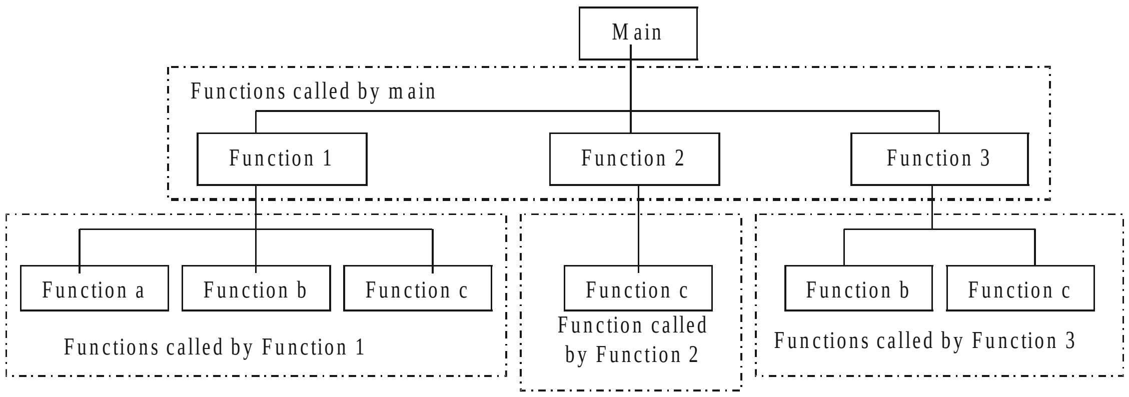

Fig. 1.2

Top-down algorithm design is a technique for organizing and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

In C, the idea of top-down design is done using functions. A C program is made of one or more functions, one and only one of which must be named *main*. The execution of the program always starts and ends with *main*, but it can call other functions to do special tasks.

1.6. BOTTOM-UP ALGORITHM DESIGN

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and build them into more complex structures, ending at the top.

The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the *main* function is tested.

1.7. STRUCTURED PROGRAMMING

It is a programming style; and this style of programming is known by several names: Procedural decomposition, Structured programming, etc. Structured programming is not programming with structures but by using following types of code structures to write programs:

1. Sequence of sequentially executed statements
2. Conditional execution of statements (*i.e.*, "if" statements)
3. Looping or iteration (*i.e.*, "for, do...while, and while" statements)
4. Structured subroutine calls (*i.e.*, functions)

In particular, the following language usage is forbidden:

- "GoTo" statements
- "Break" or "continue" out of the middle of loops
- Multiple exit points to a function/procedure/subroutine (*i.e.*, multiple "return" statements)
- Multiple entry points to a function/procedure/subroutine/method

In this style of programming there is a great risk that implementation details of many data structures have to be shared between functions, and thus globally exposed. This in turn tempts other functions to use these implementation details; thereby creating unwanted dependencies between different parts of the program.

The main disadvantage is that all decisions made from the start of the project depends directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over a time. When that happens, there is a great risk that large parts of the application need to be rewritten.

1.8. ANALYSIS OF ALGORITHM

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements :

1. Space complexity
2. Time complexity

1.8.1. SPACE COMPLEXITY

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion.

Some of the reasons for studying space complexity are:

1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.
2. We may be interested to know in advance that whether sufficient memory is available to run the program.
3. There may be several possible solutions with different space requirements.
4. Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

- *Instruction space* : Space needed to store the executable version of the program and it is fixed.
- *Data space* : Space needed to store all constants, variable values and has further two components :
  - (*a*) Space needed by constants and simple variables. This space is fixed.
  - (*b*) Space needed by fixed sized structural variables, such as arrays and structures.
  - (*c*) Dynamically allocated space. This space usually varies.
- *Environment stack space:* This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :
  - (*a*) Return address : *i.e.*, from where it has to resume after completion of the called function.
  - (*b*) Values of all lead variables and the values of formal parameters in the function being invoked .

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

1.8.2. TIME COMPLEXITY

The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used. Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in Fig. 1.3.

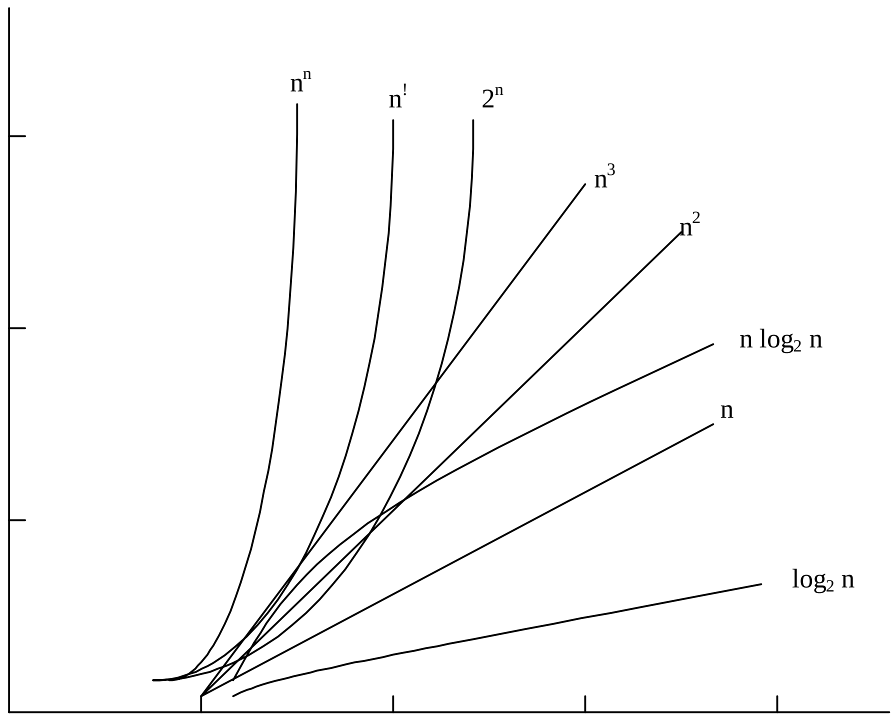

Fig. 1.3

The function that involves '*n*' as an exponent, *i.e.*, 2^n, n^n, n ! are called exponential functions, which is too slow except for small size input function where growth is less than or equal to n^c,(where '*c*' is a constant) *i.e.*; n^3, n^2, $n \log_2 n$, n, $\log_2 n$ are said to be polynomial. Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

1. Best case
2. Average case
3. Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

1.8.3. AMSTRONG COMPLEXITY

In many situations, data structures are subjected to a sequence of instructions rather than one set of instruction. In this sequence, one instruction may perform certain modifications that have an impact on other instructions in the sequence at the run time itself. For example in a *for* loop there are 100 instructions in an *if* statement. If *if* condition is false then these 100 instructions will not be executed. If we apply the time complexity analysis in worst case, entire sequence is considered to compute the efficiency, which is an excessively large and unrealistic analysis of efficiency. But when we apply amortized complexity, the complexity is calculated when the instructions are executed (*i.e.*, when *if* condition is true)

Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

1.9. TIME-SPACE TRADE OFF

There may be more than one approach (or algorithm) to solve a problem. The best algorithm (or program) to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice, it is not always possible to achieve both of these objectives. One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution. Thus, we may have to sacrifice one at the cost of the other. If the space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint such as in real time system, we have to choose a program that takes less time to complete its execution at the cost of more space.

1.10. BIG "OH" NOTATION

Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function $f(n)$ of an algorithm increases as 'n' increases.

Let us find out the algorithm complexity by analyzing the sequential searching algorithm. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains 'n' elements, the maximum possible number of comparisons with the target value will be 'n' *i.e.*, the worst case. That is the target value will be found at the nth position of the array.

$$f(n) = n$$

i.e., the worst case is when an algorithm requires a maximum number of iterations or steps to search and find out the target value in the array.

The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (*i.e.*, we need to compare the target value with only one element from the array)—we have found the element by executing only one iteration (or by least possible statements)

$$f(n) = 1$$

Average case falls between these two extremes (*i.e.*, best and worst). If the target value is found at the $n/2$nd position, on an average we need to compare the target value with only half of the elements in the array, so

$$f(n) = n/2$$

The complexity function f(n) of an algorithm increases as 'n' increases. The function $f(n)= O(n)$ can be read as "f of n is big Oh of n" or as "$f(n)$ is of the order of n". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

The generalized form of the theorem is

$$f(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \ldots\ldots + c_2 n^2 + c_1 n^1 + c_0 n^0$$

Where the constant $c_k > 0$

Then, $f(n) = O(n^k)$

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as :

1. Constant time O(1)
2. Logarithmic time Olog(n)
3. Linear time O(n)
4. Polynomial time O(n^c)
5. Exponential time O(c^n)

$\Big\}$ Where $c > 1$

1.11. LIMITATION OF BIG "OH" NOTATION

Big Oh Notation has following two basic limitations :

1. It contains no effort to improve the programming methodology. Big Oh Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyze and calculate the efficiency (by finding time complexity) of the program.
2. It does not exhibit the potential of the constants. For example, one algorithm is taking $1000n^2$ time to execute and the other n^3 time. The first algorithm is O(n^2), which implies that it will take less time than the other algorithm which is O(n^3). However in actual execution the second algorithm will be faster for $n < 1000$.

We will analyze and design the problems in data structure. As we have discussed to develop a program of an algorithm, we should select an appropriate data structure for that algorithm.

1.12. CLASSIFICATION OF DATA STRUCTURE

Data structures are broadly divided into two :

1. *Primitive data structures* : These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc. These primitive data structures are the basis for the discussion of more sophisticated (non-primitive) data structures in this book.

2. *Non-primitive data structures* : It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Array, list, files, linked list, trees and graphs fall in this category.

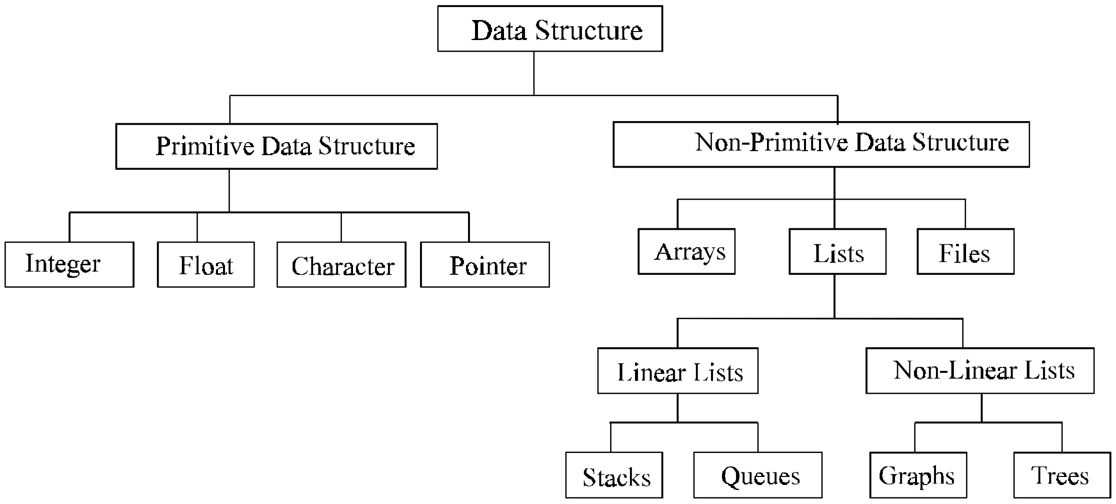

Fig. 1.4. Classifications of data structures

The Fig. 1.4 will briefly explain other classifications of data structures. Basic operations on data structure are to create a (non-primitive) data structure; which is considered to be the first step of writing a program. For example, in Pascal, C and C++, variables are created by using declaration statements.

```
int Int_Variable;
```

In C/C++, memory space is allocated for the variable "*Int_Variable*" when the above declaration statement executes. That is a data structure is created. Discussions on primitive data structures are beyond the scope of this book. Let us consider non-primitive data structures.

1.13. ARRAYS

Arrays are most frequently used in programming. Mathematical problems like matrix, algebra and etc can be easily handled by arrays. An array is a collection of homogeneous data elements described by a single name. Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis. If an element of an array is referenced by single subscript, then the array is known as one dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two dimensional array and so on. Analogously the arrays whose elements are referenced by two or more subscripts are called multi dimensional arrays.

1.13.1. ONE DIMENSIONAL ARRAY

One-dimensional array (or linear array) is a set of '*n*' finite numbers of homogenous data elements such as :

1. The elements of the array are referenced respectively by an index set consisting of '*n*' consecutive numbers.
2. The elements of the array are stored respectively in successive memory locations.

'*n*' number of elements is called the length or size of an array. The elements of an array 'A' may be denoted in C as

A[0], A[1], A[2], ... A[*n* –1].

The number '*n*' in A[*n*] is called a subscript or an index and A[n] is called a subscripted variable. If '*n*' is 10, then the array elements A[0], A[1]......A[9] are stored in sequential memory locations as follows :

A[0]	A[1]	A[2]	......	A[9]

In C, array can always be read or written through loop. To read a one-dimensional array, it requires one loop for reading and writing the array, for example:

For reading an array of '*n*' elements

```
for (i = 0; i < n; i ++)
scanf ("%d",&a[i]);
```

For writing an array

```
for (i = 0; i < n; i ++)
printf ("%d", &a[i]);
```

1.13.2. MULTI DIMENSIONAL ARRAY

If we are reading or writing two-dimensional array, two loops are required. Similarly the array of '*n*' dimensions would require '*n*' loops. The structure of the two dimensional array is illustrated in the following figure :

int A[10][10];

A_{00}	A_{01}	A_{02}						A_{08}	A_{09}
A_{10}	A_{11}								A_{19}
A_{20}									
A_{30}									
									A_{69}
A_{70}								A_{78}	A_{79}
A_{80}	A_{81}						A_{87}	A_{88}	A_{89}
A_{90}	A_{91}	A_{92}				A_{96}	A_{97}	A_{98}	A_{99}

1.13.3. SPARSE ARRAYS

Sparse array is an important application of arrays. A sparse array is an array where nearly all of the elements have the same value (usually zero) and this value is a constant. One-dimensional sparse array is called sparse vectors and two-dimensional sparse arrays are called sparse matrix.

The main objective of using arrays is to minimize the memory space requirement and to improve the execution speed of a program. This can be achieved by allocating memory space for only non-zero elements.

For example a sparse array can be viewed as

$$\begin{array}{ccccccc}
0 & 0 & 8 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 9 & 0 \\
0 & 0 & 0 & 3 & 0 & 0 & 0 \\
0 & 31 & 0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 0 & 7 & 0 & 0
\end{array}$$

Fig. 1.5. Sparse array

We will store only non-zero elements in the above sparse matrix because storing all the elements of the sparse array will be consisting of memory sparse. The non-zero elements are stored in an array of the form.

A[0……*n*][1……3]

Where ‘*n*’ is the number of non-zero elements in the array. In the above Fig. 1.4 ‘*n* = 7’. The space array given in Fig. 1.4 may be represented in the array A[0……7][1…..3].

A[0][1] A[0][2]

	1	**2**	**3**
0	5	7	7
1	1	3	8
2	2	2	1
3	2	6	9
4	3	4	3
5	4	2	31
6	4	6	4
7	5	5	7

Fig. 1.6. Sparse array representation

The element A[0][1] and A[0][2] contain the number of rows and columns of the sparse array. A[0][3] contains the total number of nonzero elements in the sparse array.

A[1][1] contains the number of the row where the first nonzero element is present in the sparse array. A[1][2] contains the number of the column of the corresponding nonzero element. A[1][3] contains the value of the nonzero element. In the Fig. 1.4, the first nonzero element can be found at 1st row in 3rd column.

1.14. VECTORS

A vector is a one-dimensional ordered collection of numbers. Normally, a number of contiguous memory locations are sequentially allocated to the vector. A vector size is fixed and, therefore, requires a fixed number of memory locations. A vector can be a column vector which represents a '*n*' by 1 ordered collections, or a row vector which represents a 1 by '*n*' ordered collections.

The column vector appears symbolically as follows :

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ \vdots \\ A_n \end{pmatrix}$$

A row vector appears symbolically as follows :

$$A = (A_1, A_2, A_3, \ldots\ldots A_n)$$

Vectors can contain either real or complex numbers. When they contain real numbers, they are sometime called real vectors. When they contain complex numbers, they are called complex vectors.

1.15. LISTS

As we have discussed, an array is an ordered set, which consist of a fixed number of elements. No deletion or insertion operations are performed on arrays. Another main disadvantage is its fixed length; we cannot add elements to the array. Lists overcome all the above limitations. A list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made. A list represented by displaying the relationship between the adjacent elements is said to be a linear list. Any other list is said to be non linear. List can be implemented by using pointers. Each element is referred to as nodes; therefore a list can be defined as a collection of nodes as shown below :

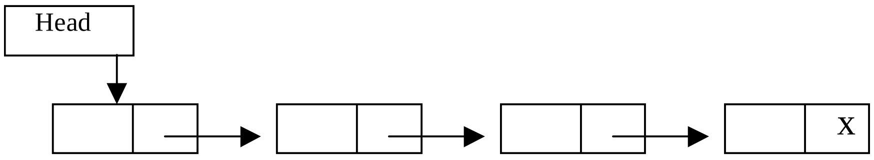

Fig. 1.7

1.16. FILES AND RECORDS

A file is typically a large list that is stored in the external memory (*e.g.*, a magnetic disk) of a computer.

A record is a collection of information (or data items) about a particular entity. More specifically, a record is a collection of related data items, each of which is called a filed or attribute and a file is a collection of similar records.

Although a record is a collection of data items, it differs from a linear array in the following ways:

(*a*) A record may be a collection of non-homogeneous data; *i.e.*, the data items in a record may have different data types.

(*b*) The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

1.17. CHARACTERISTICS OF STRINGS

In computer terminology the term 'string' refers to a sequence of characters. A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string. The number of characters in a string is called the length of the string. If the length of the string is zero then it is called the empty string or null string.

1.17.1. STRING REPRESENTATION

Strings are stored or represented in memory by using following three types of structures :

- Fixed length structures
- Variable length structures with fixed maximum
- Linear structures

FIXED LENGTH REPRESENTATION. In fixed length storage each line is viewed as a record, where all records have the same length. That is each record accommodates maximum of same number of characters.

The main advantage of representing the string in the above way is :

1. To access data from any given record easily.
2. It is easy to update the data in any given record.

The main disadvantages are :

1. Entire record will be read even if most of the storage consists of inessential blank space. Time is wasted in reading these blank spaces.
2. The length of certain records will be more than the fixed length. That is certain records may require more memory space than available.

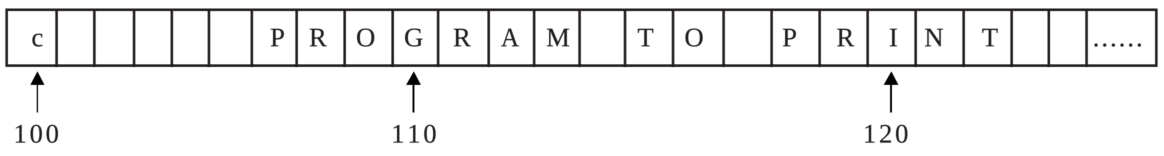

Fig. 1.8. Input data

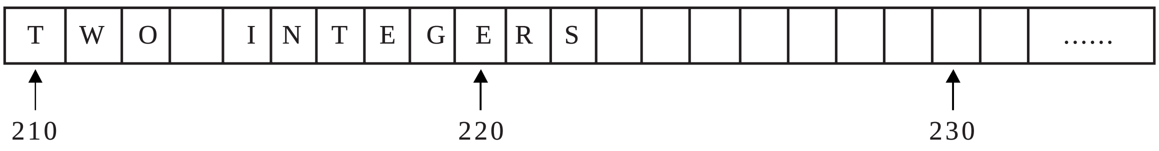

Fig. 1.9. Fixed length representation

Fig. 1.9 is a representation of input data (which is in Fig. 1.8) in a fixed length (records) storage media in a computer.

Variable Length Representation: In variable length representation, strings are stored in a fixed length storage medium. This is done in two ways.

1. One can use a marker, (any special characters) such as two-dollar sign ($$), to signal the end of the string.
2. Listing the length of the string at the first place is another way of representing strings in this method.

c		p	r	o	g	r	a	m		t	o		P	r	i	n	t		t	w	o		i	n	t	e	g	e	r	s	$$

Fig. 1.10. String representation using marker

31	c		p	r	o	g	r	a	m		t	o		p	r	i	n	t		t	w	o		i	n	t	e	g	e	r	s

Fig. 1.11. String representation by listing the length

Linked List Representations: In linked list representations each characters in a string are sequentially arranged in memory cells, called nodes, where each node contain an item and link, which points to the next node in the list (*i.e.*, link contain the address of the next node).

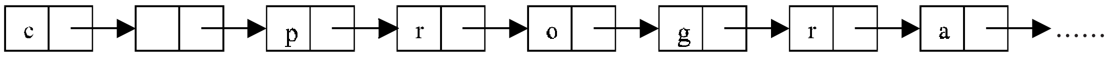

Fig. 1.12. One character per node

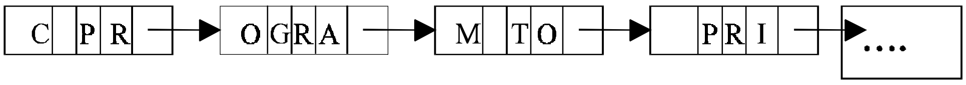

Fig. 1.13. Four character per node

We will discuss the implementation issues of linked list in chapter 5.

1.17.2. SUB STRING

Group of consecutive elements or characters in a string (or sentence) is called sub string. This group of consecutive elements may not have any special meaning. To access a sub string from the given string we need following information :

(*a*) Name of the string

(*b*) Position of the first character of the sub string in the given string

(*c*) The length of the sub string

Finding whether the sub string is available in a string by matching its characters is called pattern matching.

SELF REVIEW QUESTIONS

1. Explain how sparse matrix can be stored using arrays?
[*Calicut - APR 1997 (BTech), MG - MAY 2002* (*BTech*)
KERALA - MAY 2002 (*BTech*)]
2. Distinguish between time and space complexity?
[*ANNA - MAY 2004 (MCA), MG - MAY 2004* (*BTech*)]
3. Discuss the performance analysis and evaluation methods of algorithm?
[*KERALA - DEC 2004* (*BTech*), *MG - MAY 2004* (*BTech*)]
4. Define and explain Big O notation?
[*MG - NOV 2004* (*BTech*), *MG - NOV 2003* (*BTech*)]
5. What are sparse matrixes? Give an example?
[*CUSAT - NOV 2002* (*BTech*), *Calicut - APR 1995* (*BTech*),
CUSAT - JUL 2002 (*MCA*), *MG - NOV 2004* (*BTech*)
KERALA - MAY 2001 (*BTech*), *KERALA - MAY 2003* (*BTech*)]
6. Explain the schemes of data representations for strings? [*MG - NOV 2004* (*BTech*)]
7. Define complexity of an algorithm. What is meant by time-space trade off ?
[*CUSAT - MAY 2000* (*BTech*), *MG - NOV 2004* (*BTech*),
KERALA - DEC 2002 (*BTech*), *MG - MAY 2000* (*BTech*)]
8. Discuss the different steps in the development of an algorithm?
[*MG - NOV 2004* (*BTech*)]
9. Discuss the advantages and disadvantages of Modular Programming.
[*Calicut - APR 1995* (*BTech*)]
10. What is an Algorithm? Explain with example the time and space analysis of an algorithm. [*Calicut - APR 1995* (*BTech*)]
11. Pattern matching in strings. [*Calicut - APR 1997* (*BTech*)]
12. Distinguish between primitive and non-primitive data structures. Explain how integer data are mapped to storage. [*CUSAT - APR 1998* (*BTech*)]
13. Explain what is meant by dynamic storage management?
[*ANNA - DEC 2004* (*BE*), *CUSAT - MAY 2000* (*BTech*)]
14. Explain in detail about *top-down* approach and *bottom-up* approach with suitable programming examples. [*ANNA - MAY 2003* (*BE*), *ANNA - DEC 2003* (*BE*)]
15. What do you mean by stepwise refinement?
[*KERALA - DEC 2004* (*BTech*), *ANNA - DEC 2003* (*BE*)
KERALA - DEC 2003 (*BTech*), *KERALA - JUN 2004* (*BTech*)
KERALA - MAY 2003 (*BTech*)]
16. What are the features of structured programming methodologies? Explain.
[*ANNA - MAY 2003* (*BE*), *ANNA - DEC 2004* (*BE*)]
17. Differentiate linear and non-linear data structures.
[*ANNA - MAY 2004* (*BE*), *ANNA - MAY 2004* (*MCA*)]
18. What are the primitive functions in a string handling system? [*ANNA - DEC 2004* (*BE*)]
19. What is meant by unstructured program? [*ANNA - MAY 2004* (*BE*)]

20. What is meant by algorithm ? What are its measures? [*ANNA - MAY 2004* (*BE*)]
21. What are primitive data types ? [*ANNA - MAY 2003* (*BE*)]
22. Explain (*i*) Array vs. record. (*ii*) Time complexity
[*KERALA - MAY 2001* (*BTech*), *KERALA - JUN 2004* (*BTech*)]
23. Explain Programming methodology. [*KERALA - DEC 2003* (*BTech*)]
24. Explain about analysis of algorithms.
[*KERALA - MAY 2003* (*BTech*), *KERALA - DEC 2003* (*BTech*)]
25. What is structured programming ? Explain. [*KERALA - MAY 2001* (*BTech*)]
26. Distinguish between a program and an algorithm. [*KERALA - MAY 2002* (*BTech*)]
27. Explain the advantages and disadvantage of list structure over array structure.
[*KERALA - MAY 2002* (*BTech*)]
28. Explain the term "data structure". [*KERALA - NOV 2001* (*BTech*)]
29. What do you understand by best, worst and average case analysis of an algorithm ?
[*KERALA - NOV 2001* (*BTech*)]
30. What are the uses of an array ? What is an ordered array ?
[*KERALA - NOV 2001* (*BTech*)]
31. How will you specify the time complexity of an algorithm ? [*CUSAT - OCT 2000* (*BTech*)]