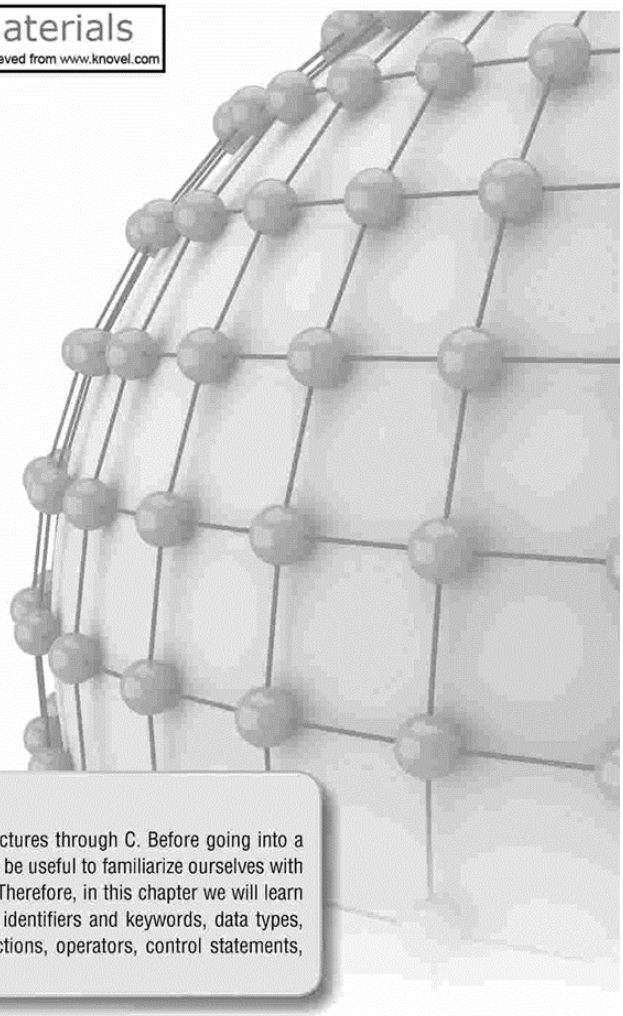
CHAPTER 1

Introduction to C



LEARNING OBJECTIVE

This book deals with the study of data structures through C. Before going into a detailed analysis of data structures, it would be useful to familiarize ourselves with the basic knowledge of programming in C. Therefore, in this chapter we will learn about the various constructs of C such as identifiers and keywords, data types, constants, variables, input and output functions, operators, control statements, functions, and pointers.

1.1 INTRODUCTION

The programming language 'C' was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Although C was initially developed for writing system software, today it has become such a popular language that a variety of software programs are written using this language. The greatest advantage of using C for programming is that it can be easily used on different types of computers. Many other programming languages such as C++ and Java are also based on C which means that you will be able to learn them easily in the future. Today, C is widely used with the UNIX operating system.

Structure of a C Program

A C program contains one or more functions, where a function is defined as a group of statements that perform a well-defined task. Figure 1.1 shows the structure of a C program. The statements in a function are written in a logical sequence to perform a specific task. The `main()` function is the most important function and is a part of every C program. Rather, the execution of a C program begins with this function.

From the structure given in Fig. 1.1, we can conclude that a C program can have any number of functions depending on the tasks that have to be performed, and each function can have any number

```
main()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
Function1()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
Function2()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
..................
..................
FunctionN()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
```

Figure 1.1 Structure of a C program

of statements arranged according to specific meaningful sequence. Note that programmers can choose any name for functions. It is not mandatory to write Function1, Function2, etc., with an exception that every program must contain one function that has its name as `main()`.

1.2 IDENTIFIERS AND KEYWORDS

Every word in a C program is either an identifier or a keyword.

Identifiers

Identifiers are basically names given to program elements such as variables, arrays, and functions. They are formed by using a sequence of letters (both uppercase and lowercase), numerals, and underscores.
Following are the rules for forming identifier names:

- Identifiers cannot include any special characters or punctuation marks (like #, $, ^, ?, ., etc.) except the underscore "_".
- There cannot be two successive underscores.
- Keywords cannot be used as identifiers.
- The case of alphabetic characters that form the identifier name is significant. For example, 'FIRST' is different from 'first' and 'First'.
- Identifiers must begin with a letter or an underscore. However, use of underscore as the first character must be avoided because several complier-defined identifiers in the standard C library have underscore as their first character. So, inadvertently duplicated names may cause definition conflicts.
- Identifiers can be of any reasonable length. They should not contain more than 31 characters. (They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.)

Keywords

Like every computer language, C has a set of reserved words often known as keywords that cannot be used as an identifier. All keywords are basically a sequence of characters that have a fixed meaning. By convention, all keywords must be written in lower case letters. Table 1.1 contains the list of keywords in C.

Table 1.1 Keywords in C language

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

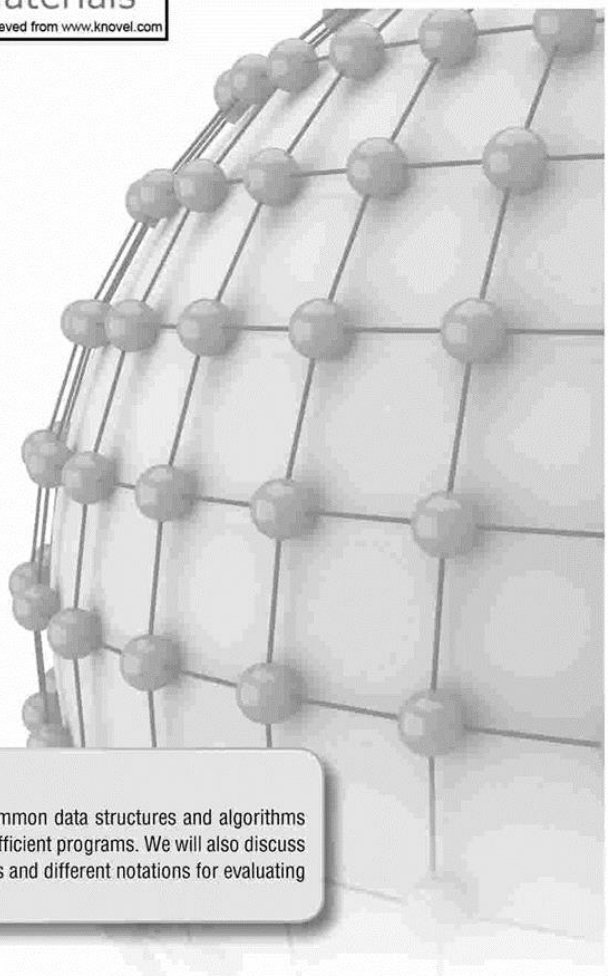
1.3 BASIC DATA TYPES

Data type determines the set of values that a data item can take and the operations that can be performed on the item. C language provides four basic data types. Table 1.2 lists the data types, their size, range, and usage for a C programmer.

The `char` data type is of one byte and is used to store single characters. Note that C does not provide any data type for storing text. This is because text is made up of individual characters. You

CHAPTER 2

Introduction to Data Structures and Algorithms



LEARNING OBJECTIVE

In this chapter, we are going to discuss common data structures and algorithms which serve as building blocks for creating efficient programs. We will also discuss different approaches to designing algorithms and different notations for evaluating the performance of algorithms.

2.1 BASIC TERMINOLOGY

We have already learnt the basics of programming in C in the previous chapter and know how to write, debug, and run simple programs in C language. Our aim has been to design good programs, where a good program is defined as a program that

- runs correctly
- is easy to read and understand
- is easy to debug *and*
- is easy to modify.

A program should undoubtedly give correct results, but along with that it should also run efficiently. A program is said to be efficient when it executes in minimum time and with minimum memory space. In order to write efficient programs we need to apply certain data management concepts.

The concept of data management is a complex task that includes activities like data collection, organization of data into appropriate structures, and developing and maintaining routines for quality assurance.

Data structure is a crucial part of data management and in this book it will be our prime concern. A *data structure* is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Data structures are widely applied in the following areas:

- Compiler design
- Statistical analysis package
- Numerical analysis
- Artificial intelligence
- Operating system
- DBMS
- Simulation
- Graphics

When you will study DBMS as a subject, you will realize that the major data structures used in the Network data model is graphs, Hierarchical data model is trees, and RDBMS is arrays.

Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently. Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design. This is because representing information is fundamental to computer science. The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

Be it any problem at hand, the application of an appropriate data structure provides the most efficient solution. A solution is said to be efficient if it solves the problem within the required resource constraints like the total space available to store the data and the time allowed to perform each subtask. And the best solution is the one that requires fewer resources than known alternatives. Moreover, the cost of a solution is the amount of resources it consumes. The cost of a solution is basically measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

Today computer programmers do not write programs just to solve a problem but to write an efficient program. For this, they first analyse the problem to determine the performance goals that must be achieved and then think of the most appropriate data structure for that job. However, program designers with a poor understanding of data structure concepts ignore this analysis step and apply a data structure with which they can work comfortably. The applied data structure may not be appropriate for the problem at hand and therefore may result in poor performance (like slow speed of operations).

Conversely, if a program meets its performance goals with a data structure that is simple to use, then it makes no sense to apply another complex data structure just to exhibit the programmer's skill. When selecting a data structure to solve a problem, the following steps must be performed.

1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to select an appropriate data structure for the problem at hand supports a data-centred view of the design process. In the approach, the first concern is the data and the operations that are to be performed on them. The second concern is the representation of the data, and the final concern is the implementation of that representation.

There are different types of data structures that the C language supports. While one type of data structure may permit adding of new data items only at the beginning, the other may allow it to be added at any position. While one data structure may allow accessing data items sequentially, the other may allow random access of data. So, selection of an appropriate data structure for the problem is a crucial decision and may have a major impact on the performance of the program.

2.1.1 Elementary Data Structure Organization

Data structures are building blocks of a program. A program built using improper data structures may not work as expected. So as a programmer it is mandatory to choose most appropriate data structures for a program.

The term *data* means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of *pi*, etc.).

While a data item that does not have subordinate data items is categorized as an elementary item, the one that is composed of one or more subordinate data items is called a group item. For example, a student's name may be divided into three sub-items—first name, middle name, and last name—but his roll number would normally be treated as a single item.

A *record* is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.

A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students. All these related records are stored in a file. Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so forth.

Moreover, each record in a file may consist of multiple data items but the value of a certain data item uniquely identifies the record in the file. Such a data item K is called a *primary key*, and the values K_1, K_2 ... in such field are called keys or key values. For example, in a student's record that contains roll number, name, address, course, and marks obtained, the field roll number is a primary key. Rest of the fields (name, address, course, and marks) cannot serve as primary keys, since two or more students may have the same name, or may have the same address (as they might be staying at the same place), or may be enrolled in the same course, or have obtained same marks.

This organization and hierarchy of data is taken further to form more complex types of data structures, which is discussed in Section 2.2.

2.2 CLASSIFICATION OF DATA STRUCTURES

Data structures are generally categorized into two classes: *primitive* and *non-primitive* data structures.

Primitive and Non-primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.
Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

Linear and Non-linear Structures

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

C supports a variety of data structures. We will now introduce all these data structures and they would be discussed in detail in subsequent chapters.

Arrays

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).

In C, arrays are declared using the following syntax:

```
type name[size];
```

For example,

```
int marks[10];
```

The above statement declares an array `marks` that contains 10 elements. In C, the array index starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in `marks[0]`, second element in `marks[1]`, so on and so forth. Therefore, the last element, that is the 10th element, will be stored in `marks[9]`. In the memory, the array will be stored as shown in Fig. 2.1.

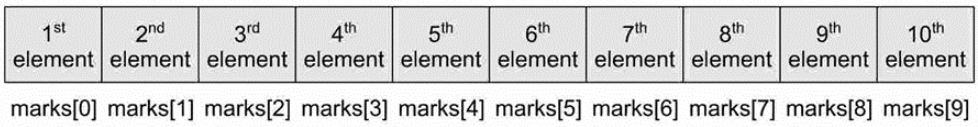


Figure 2.1 Memory representation of an array of 10 elements

Arrays are generally used when we want to store large amount of similar type of data. But they have the following limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

However, these limitations can be solved by using linked lists. We will discuss more about arrays in Chapter 3.

Linked Lists

A linked list is a very flexible, dynamic data structure in which elements (called *nodes*) form a sequential list. In contrast to static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require less maintenance.

In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list

The last node in the list contains a `NULL` pointer to indicate that it is the end or *tail* of the list. Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available. Figure 2.2 shows a linked list of seven nodes.

Figure 2.2 Simple linked list

Note *Advantage*: Easier to insert or delete data elements
Disadvantage: Slow search operation and requires more memory space

Stacks

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the `top` of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

In the computer's memory, stacks can be implemented using arrays or linked lists. Figure 2.3 shows the array implementation of a stack. Every stack has a variable `top` associated with it. `top` is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted. There is another variable `MAX`, which is used to store the maximum number of elements that the stack can store.

If `top = NULL`, then it indicates that the stack is empty and if `top = MAX-1`, then the stack is full.

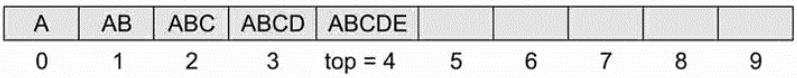


Figure 2.3 Array representation of a stack

In Fig. 2.3, `top = 4`, so insertions and deletions will be done at this position. Here, the stack can store a maximum of 10 elements where the indices range from 0–9. In the above stack, five more elements can still be stored.

A stack supports three basic operations: `push`, `pop`, and `peep`. The `push` operation adds an element to the top of the stack. The `pop` operation removes the element from the top of the stack. And the `peep` operation returns the value of the topmost element of the stack (without deleting it).

However, before inserting an element in the stack, we must check for overflow conditions. An overflow occurs when we try to insert an element into a stack that is already full.

Similarly, before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a stack that is already empty.

Queues

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the `rear` and removed from the other end called the `front`. Like stacks, queues can be implemented by using either arrays or linked lists.

Every queue has `front` and `rear` variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in Fig. 2.4.

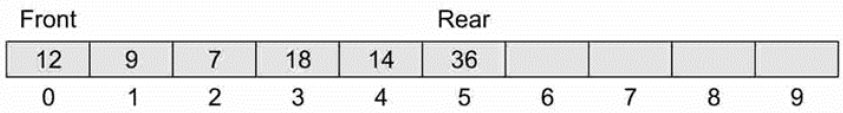


Figure 2.4 Array representation of a queue

Here, `front = 0` and `rear = 5`. If we want to add one more value to the list, say, if we want to add another element with the value 45, then the `rear` would be incremented by 1 and the value would be stored at the position pointed by the `rear`. The queue, after the addition, would be as shown in Fig. 2.5.

Here, `front = 0` and `rear = 6`. Every time a new element is to be added, we will repeat the same procedure.

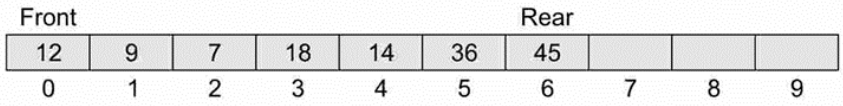


Figure 2.5 Queue after insertion of a new element

Now, if we want to delete an element from the queue, then the value of `front` will be incremented. Deletions are done only from this end of the queue. The queue after the deletion will be as shown in Fig. 2.6.

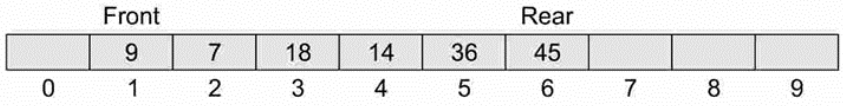


Figure 2.6 Queue after deletion of an element

However, before inserting an element in the queue, we must check for overflow conditions. An overflow occurs when we try to insert an element into a queue that is already full. A queue is full when `rear = MAX-1`, where `MAX` is the size of the queue, that is `MAX` specifies the maximum number of elements in the queue. Note that we have written `MAX-1` because the index starts from 0.

Similarly, before deleting an element from the queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If `front = NULL` and `rear = NULL`, then there is no element in the queue.

Trees

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree. The root element is the topmost node which is pointed by a 'root' pointer. If `root = NULL` then the tree is empty.

Figure 2.7 shows a binary tree, where `R` is the root node and T_1 and T_2 are the left and right sub-trees of `R`. If T_1 is non-empty, then T_1 is said to be the left successor of `R`. Likewise, if T_2 is non-empty, then it is called the right successor of `R`.

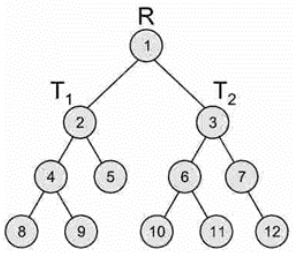


Figure 2.7 Binary tree

In Fig. 2.7, node 2 is the left child and node 3 is the right child of the root node 1. Note that the left sub-tree of the root node consists of the nodes 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12.

Note *Advantage*: Provides quick search, insert, and delete operations
Disadvantage: Complicated deletion algorithm

Graphs

A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.

In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions. Figure 2.8 shows a graph with five nodes.

A node in the graph may represent a city and the edges connecting the nodes can represent roads. A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections. Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph.

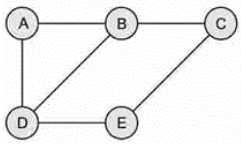


Figure 2.8 Graph

Note that unlike trees, graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph. When two nodes are connected via an edge, the two nodes are known as *neighbours*. For example, in Fig. 2.8, node A has two neighbours: B and D.

Note *Advantage*: Best models real-world situations
Disadvantage: Some algorithms are slow and very complex

2.3 OPERATIONS ON DATA STRUCTURES

This section discusses the different operations that can be performed on the various data structures previously mentioned.

Traversing It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

Deleting It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

Sorting Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging Lists of two sorted data items can be combined to form a single list of sorted data items.

$0 \leq c < 50/100$

This is a contradictory value as for any value of c as it cannot be assured to be less than 50/100 or 1/2.

An imprecise analogy between the asymptotic comparison of functions $f(n)$ and $g(n)$ and the relation between their values can be given as:

$f(n) = \Omega(g(n)) \approx f(n) \geq g(n)$ $\quad\quad$ $f(n) = \omega(g(n)) \approx f(n) > g(n)$

Points to Remember

- A data structure is a particular way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently.
- There are two types of data structures: primitive and non-primitive data structures. Primitive data structures are the fundamental data types which are supported by a programming language. Non-primitive data structures are those data structures which are created using primitive data structures.
- Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.
- If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. However, if the elements of a data structure are not stored in sequential order, then it is a non-linear data structure.
- An array is a collection of similar data elements which are stored in consecutive memory locations.
- A linked list is a linear data structure consisting of a group of elements (called nodes) which together represent a sequence.
- A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
- A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front.
- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical tree structure.
- The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees.
- A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships can exist between the nodes.
- An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
- An algorithm is basically a set of instructions that solve a problem.
- The time complexity of an algorithm is basically the running time of the program as a function of the input size.
- The space complexity of an algorithm is the amount of computer memory required during the program execution as a function of the input size.
- The worst-case running time of an algorithm is an upper bound on the running time for any input.
- The average-case running time specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.
- Amortized analysis guarantees the average performance of each operation in the worst case.
- The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed and the type of the loop that is being used.

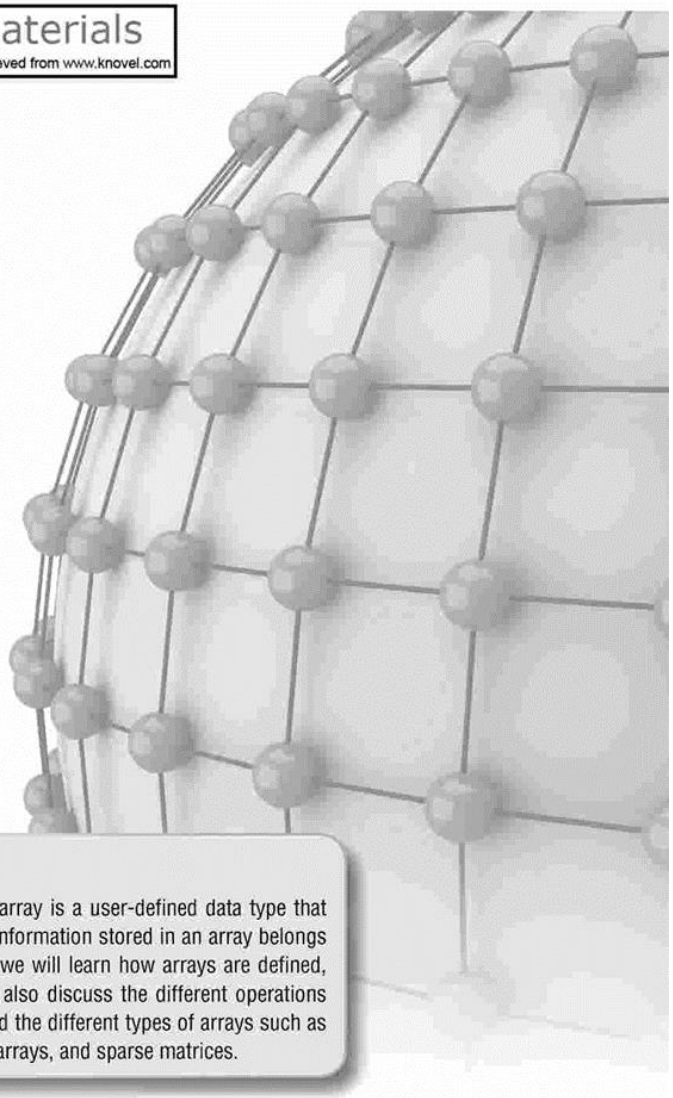
Exercises

Review Questions

1. Explain the features of a good program.
2. Define the terms: data, file, record, and primary key.

CHAPTER 3

Arrays



LEARNING OBJECTIVE

In this chapter, we will discuss arrays. An array is a user-defined data type that stores related information together. All the information stored in an array belongs to the same data type. So, in this chapter, we will learn how arrays are defined, declared, initialized, and accessed. We will also discuss the different operations that can be performed on array elements and the different types of arrays such as two-dimensional arrays, multi-dimensional arrays, and sparse matrices.

3.1 INTRODUCTION

We will explain the concept of arrays using an analogy. Consider a situation in which we have 20 students in a class and we have been asked to write a program that reads and prints the marks of all the 20 students. In this program, we will need 20 integer variables with different names, as shown in Fig. 3.1.

Now to read the values of these 20 variables, we must have 20 read statements. Similarly, to print the value of these variables, we need 20 write statements. If it is just a matter of 20 variables, then it might be acceptable for the user to follow this approach. But would it be possible to follow this approach if we have to read and print the marks of students,

- in the entire course (say 100 students)
- in the entire college (say 500 students)
- in the entire university (say 10,000 students)

The answer is no, definitely not! To process a large amount of data, we need a data structure known as *array*.

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an `index` (also known as the *subscript*). The subscript is an ordinal number which is used to identify an element of the array.

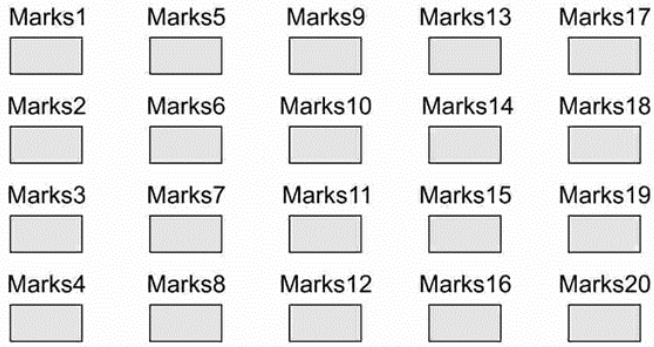


Figure 3.1 Twenty variables for 20 students

3.2 DECLARATION OF ARRAYS

We have already seen that every variable must be declared before it is used. The same concept holds true for array variables. An array must be declared before being used. Declaring an array means specifying the following:

- *Data type*—the kind of values it can store, for example, `int`, `char`, `float`, `double`.
- *Name*—to identify the array.
- *Size*—the maximum number of values that the array can hold.

Arrays are declared using the following syntax:

```
type name[size];
```

The type can be either `int`, `float`, `double`, `char`, or any other valid data type. The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array. For example, if we write,

```
int marks[10];
```

then the statement declares `marks` to be an array containing 10 elements. In C, the array index starts from zero. The first element will be stored in `marks[0]`, second element in `marks[1]`, and so on. Therefore, the last element, that is the 10th element, will be stored in `marks[9]`. Note that `0, 1, 2, 3` written within square brackets are the subscripts. In the memory, the array will be stored as shown in Fig. 3.2.

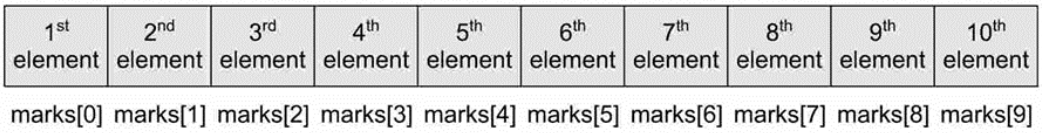


Figure 3.2 Memory representation of an array of 10 elements

Figure 3.3 shows how different types of arrays are declared.

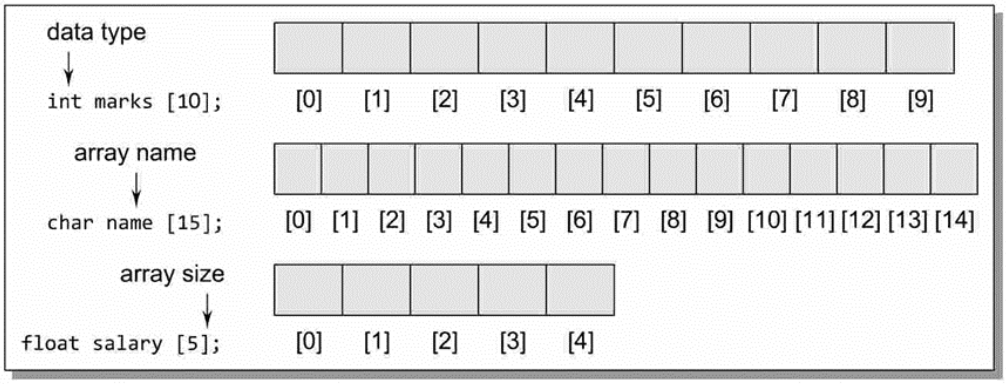


Figure 3.3 Declaring arrays of different data types and sizes

3.3 ACCESSING THE ELEMENTS OF AN ARRAY

Storing related data items in a single array enables the programmers to develop concise and efficient programs. But there is no single function that can operate on all the elements of an array. To access all the elements, we must use a loop. That is, we can access all the elements of an array by varying the value of the subscript into the array. But note that the subscript must be an integral value or an expression that evaluates to an integral value. As shown in Fig. 3.2, the first element of the array `marks[10]` can be accessed by writing `marks[0]`. Now to process all the elements of the array, we use a loop as shown in Fig. 3.4.

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
       marks[i] = -1;
```

Figure 3.4 Code to initialize each element of the array to –1

Figure 3.5 shows the result of the code shown in Fig. 3.4. The code accesses every individual element of the array and sets its value to –1. In the `for` loop, first the value of `marks[0]` is set to –1, then the value of the index `(i)` is incremented and the next value, that is, `marks[1]` is set to –1. The procedure continues until all the 10 elements of the array are set to –1.

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 3.5 Array `marks` after executing the code given in Fig. 3.4

Note There is no single statement that can read, access, or print all the elements of an array. To do this, we have to use a loop to execute the same statement with different index values.

3.3.1 Calculating the Address of Array Elements

You must be wondering how C gets to know where an individual element of an array is located in the memory. The answer is that the array name is a symbolic reference to the address of the first byte of the array. When we use the array name, we are actually referring to the first byte of the array.

The subscript or the index represents the offset from the beginning of the array to the element being referenced. That is, with just the array name and the index, C can calculate the address of any element in the array.

Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient. The address of

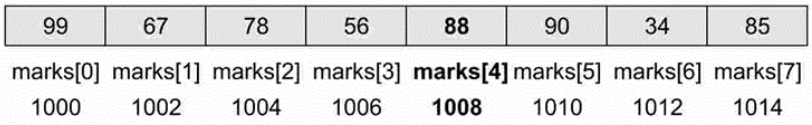
other data elements can simply be calculated using the base address. The formula to perform this calculation is,

```
Address of data element, A[k] = BA(A) + w(k - lower_bound)
```

Here, `A` is the array, `k` is the index of the element of which we have to calculate the address, `BA` is the base address of the array `A`, and `w` is the size of one element in memory, for example, size of `int` is 2.

Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution



We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

```
marks[4] = 1000 + 2(4 - 0)
         = 1000 + 2(4) = 1008
```

3.3.2 Calculating the Length of an Array

The length of an array is given by the number of elements stored in it. The general formula to calculate the length of an array is

```
Length = upper_bound - lower_bound + 1
```

where `upper_bound` is the index of the last element and `lower_bound` is the index of the first element in the array.

Example 3.2 Let `Age[5]` be an array of integers such that

```
Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7
```

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.


```
Length = upper_bound - lower_bound + 1
```

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length` = `4 - 0 + 1 = 5`

3.4 STORING VALUES IN ARRAYS

When we declare an array, we are just allocating space for its elements; no values are stored in the array. There are three ways to store values in an array. First, to initialize the array elements during declaration; second, to input values for individual elements from the keyboard; third, to assign values to individual elements. This is shown in Fig. 3.6.

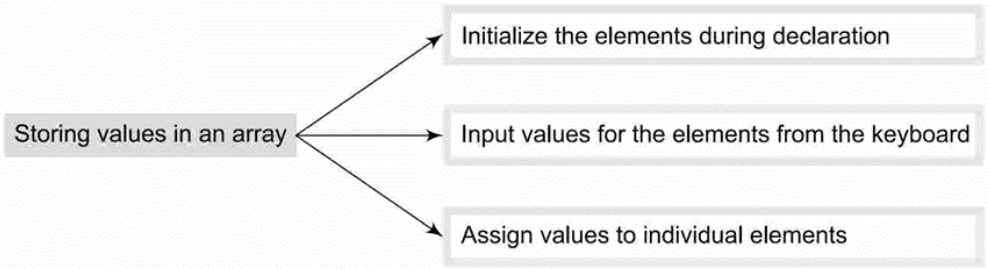


Figure 3.6 Storing values in an array

Initializing Arrays during Declaration

The elements of an array can be initialized at the time of declaration, just as any other variable. When an array is initialized, we need to provide a value for every element in the array. Arrays are initialized by writing,

```
type array_name[size]={list of values};
```

marks[0]	90
marks[1]	82
marks[2]	78
marks[3]	95
marks[4]	88

Figure 3.7 Initialization of array `marks[5]`

Note that the values are written within curly brackets and every value is separated by a comma. It is a compiler error to specify more values than there are elements in the array. When we write,

```
int marks[5]={90, 82, 78, 95, 88};
```

An array with the name `marks` is declared that has enough space to store five elements. The first element, that is, `marks[0]` is assigned value 90. Similarly, the second element of the array, that is `marks[1]`, is assigned 82, and so on. This is shown in Fig. 3.7.

While initializing the array at the time of declaration, the programmer may omit the size of the array. For example,

```
int marks[]= {98, 97, 90};
```

The above statement is absolutely legal. Here, the compiler will allocate enough space for all the initialized elements. Note that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros. Figure 3.8 shows the initialization of arrays.

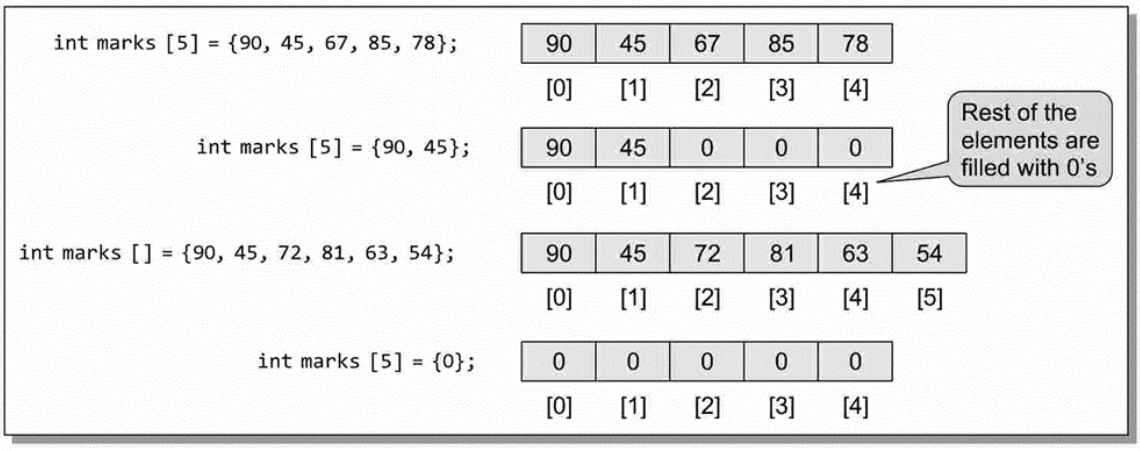


Figure 3.8 Initialization of array elements

```
int i, marks[10];
for(i=0;i<10;i++)
       scanf("%d", &marks[i]);
```

Figure 3.9 Code for inputting each element of the array

Inputting Values from the Keyboard

An array can be initialized by inputting values from the keyboard. In this method, a `while/do-while` or a `for` loop is executed to input the value for each element of the array. For example, look at the code shown in Fig. 3.9.

In the code, we start at the index `i` at 0 and input the value for the first element of the array. Since the array has 10 elements, we must input values for elements whose index varies from 0 to 9.

Assigning Values to Individual Elements

The third way is to assign values to individual elements of the array by using the assignment operator. Any value that evaluates to the data type as that of the array can be assigned to the individual array element. A simple assignment statement can be written as

```
marks[3] = 100;
```

Here, 100 is assigned to the fourth element of the array which is specified as `marks[3]`.

```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
       arr2[i] = arr1[i];
```

Figure 3.10 Code to copy an array at the individual element level

Note that we cannot assign one array to another array, even if the two arrays have the same type and size. To copy an array, you must copy the value of every element of the first array into the elements of the second array. Figure 3.10 illustrates the code to copy an array.

In Fig. 3.10, the loop accesses each element of the first array and simultaneously assigns its value to the corresponding element of the second array. The index value `i` is incremented to access the next element in succession. Therefore, when this code is executed, `arr2[0] = arr1[0]`, `arr2[1] = arr1[1]`, `arr2[2] = arr1[2]`, and so on.

```
// Fill an array with even numbers
int i,arr[10];
for(i=0;i<10;i++)
       arr[i] = i*2;
```

Figure 3.11 Code for filling an array with even numbers

We can also use a loop to assign a pattern of values to the array elements. For example, if we want to fill an array with even integers (starting from 0), then we will write the code as shown in Fig. 3.11.

In the code, we assign to each element a value equal to twice of its index, where the index starts from 0. So after executing this code, we will have `arr[0]=0`, `arr[1]=2`, `arr[2]=4`, and so on.

3.5 OPERATIONS ON ARRAYS

There are a number of operations that can be preformed on arrays. These operations include:

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

We will discuss all these operations in detail in this section, except searching and sorting, which will be discussed in Chapter 14.

3.5.1 Traversing an Array

Traversing an array means accessing each and every element of the array for a specific purpose.

Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements. Since, array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward. The algorithm for array traversal is given in Fig. 3.12.

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:       Apply Process to A[I]
Step 4:       SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 3.12 Algorithm for array traversal

In Step 1, we initialize the index to the lower bound of the array. In Step 2, a `while` loop is executed. Step 3 processes the individual array element as specified by the array name and index value. Step 4 increments the index value so that the next array element could be processed. The `while` loop in Step 2 is executed until all the elements in the array are processed, i.e., until I is less than or equal to the upper bound of the array.

Programming Examples

1. Write a program to read and display *n* numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, arr[20];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
               printf("\n arr[%d] = ", i);
               scanf("%d",&arr[i]);
        }
        printf("\n The array elements are ");
        for(i=0;i<n;i++)
               printf("\t %d", arr[i]);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The array elements are     1     2     3     4     5
```

2. Write a program to find the mean of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
```

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Figure 3.13 Algorithm to append a new element to an existing array

3.5.2 Inserting an Element in an Array

If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple. We just have to add 1 to the `upper_bound` and assign the value. Here, we assume that the memory space allocated for the array is still available. For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.

Figure 3.13 shows an algorithm to insert a new element to the end of an array. In Step 1, we increment the value of the `upper_bound`. In Step 2, the new value is stored at the position pointed by the `upper_bound`. For example, let us assume an array has been declared as

```
int marks[60];
```

The array is declared to store the marks of all the students in a class. Now, suppose there are 54 students and a new student comes and is asked to take the same test. The marks of this new student would be stored in `marks[55]`. Assuming that the student secured 68 marks, we will assign the value as

```
marks[55] = 68;
```

However, if we have to insert an element in the middle of the array, then this is not a trivial task. On an average, we might have to move as much as half of the elements from their positions in order to accommodate space for the new element.

For example, consider an array whose elements are arranged in ascending order. Now, if a new element has to be added, it will have to be added probably somewhere in the middle of the array. To do this, we must first find the location where the new element will be inserted and then move all the elements (that have a value greater than that of the new element) one position to the right so that space can be created to store the new value.

Example 3.3 `Data[]` is an array that is declared as `int Data[20];` and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) Calculate the length of the array.
(b) Find the `upper_bound` and `lower_bound`.
(c) Show the memory representation of the array.
(d) If a new data element with the value 75 has to be inserted, find its position.
(e) Insert a new data element 75 and show the memory representation after the insertion.

Solution

(a) Length of the array = number of elements
Therefore, `length of the array = 10`

(b) By default, `lower_bound = 0` and `upper_bound = 9`

(c)

12	23	34	45	56	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]

(d) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6th position will be moved one position towards the right to accommodate the new value

(e)

12	23	34	45	56	67	75	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]

Algorithm to Insert an Element in the Middle of an Array

The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are

(a) A, the array in which the element has to be inserted
(b) N, the number of elements in the array
(c) POS, the position at which the element has to be inserted
(d) VAL, the value that has to be inserted

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:          SET A[I + 1] = A[I]
Step 4:          SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Figure 3.14 Algorithm to insert an element in the middle of an array.

In the algorithm given in Fig. 3.14, in Step 1, we first initialize I with the total number of elements in the array. In Step 2, a `while` loop is executed which will move all the elements having an index greater than POS one position towards right to create space for the new element. In Step 5, we increment the total number of elements in the array by 1 and finally in Step 6, the new value is inserted at the desired position.

Now, let us visualize this algorithm by taking an example.

Initial Data[] is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling INSERT(Data, 6, 3, 100) will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

PROGRAMMING EXAMPLES

7. Write a program to insert a number at a given location in an array.

```
#include <stdio.h>
```

```
            scanf("%d", &num);
            for(i=0;i<n;i++)
            {
                    if(arr[i] > num)
                    {
                            for(j = n-1; j>=i; j--)
                                    arr[j+1] = arr[j];
                            arr[i] = num;
                            break;
                    }
            }
            n = n+1;
            printf("\n The array after insertion of %d is : ", num);
            for(i=0;i<n;i++)
                    printf("\n arr[%d] = %d", i, arr[i]);
            getch();
            return 0;
    }
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 4
arr[3] = 5
arr[4] = 6
Enter the number to be inserted : 3
The array after insertion of 3 is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
```

3.5.3 Deleting an Element from an Array

Deleting an element from an array means removing a data element from an already existing array. If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple. We just have to subtract 1 from the `upper_bound`. Figure 3.15 shows an algorithm to delete an element from the end of an array.

For example, if we have an array that is declared as

```
int marks[60];
```

The array is declared to store the marks of all the students in the class. Now, suppose there are 54 students and the student with roll number 54 leaves the course. The score of this student was stored in `marks[54]`. We just have to decrement the `upper_bound`. Subtracting 1 from the `upper_bound` will indicate that there are 53 valid data in the array.

However, if we have to delete an element from the middle of an array, then it is not a trivial task. On an average, we might have to move as much as half of the elements from their positions in order to occupy the space of the deleted element.

```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Figure 3.15 Algorithm to delete the last element of an array

For example, consider an array whose elements are arranged in ascending order. Now, suppose an element has to be deleted, probably from somewhere in the

```
        return pos;
}
```

Output

```
Enter the size of the array : 5
arr[0] = 5
arr[1] = 1
arr[2] = 6
arr[3] = 3
arr[4] = 2
The new array is :
arr[0] = 5
arr[1] = 6
arr[2] = 1
arr[3] = 3
arr[4] = 2
```

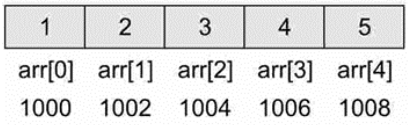


Figure 3.23 Memory representation of `arr[]`

3.7 POINTERS AND ARRAYS

The concept of array is very much bound to the concept of pointer. Consider Fig. 3.23. For example, if we have an array declared as,

```
int arr[] = {1, 2, 3, 4, 5};
```

then in memory it would be stored as shown in Fig. 3.23.

Array notation is a form of pointer notation. The name of the array is the starting address of the array in memory. It is also known as the base address. In other words, base address is the address of the first element in the array or the address of `arr[0]`. Now let us use a pointer variable as given in the statement below.

```
int *ptr;
ptr = &arr[0];
```

Programming Tip

The name of an array is actually a pointer that points to the first element of the array.

Here, `ptr` is made to point to the first element of the array. Execute the code given below and observe the output which will make the concept clear to you.

```
main()
{
    int arr[]={1,2,3,4,5};
    printf("\n Address of array = %p %p %p", arr, &arr[0], &arr);
}
```

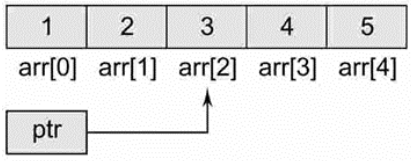


Figure 3.24 Pointer pointing to the third element of the array

Similarly, writing `ptr = &arr[2]` makes `ptr` to point to the third element of the array that has index `2`. Figure 3.24 shows `ptr` pointing to the third element of the array.

If pointer variable `ptr` holds the address of the first element in the array, then the address of successive elements can be calculated by writing `ptr++`.

```
int *ptr = &arr[0];
ptr++;
printf("\n The value of the second element of the array is %d",
*ptr);
```

Programming Tip

An error is generated if an attempt is made to change the address of the array.

The `printf()` function will print the value `2` because after being incremented `ptr` points to the next location. One point to note here is that if `x` is an integer variable, then `x++;` adds `1` to the value of x. But `ptr`

is a pointer variable, so when we write `ptr+i`, then adding `i` gives a pointer that points `i` elements further along an array than the original pointer.

Since `++ptr` and `ptr++` are both equivalent to `ptr+1`, incrementing a pointer using the unary `++` operator, increments the address it stores by the amount given by `sizeof(type)` where `type` is the data type of the variable it points to (i.e., `2` for an integer). For example, consider Fig. 3.25.

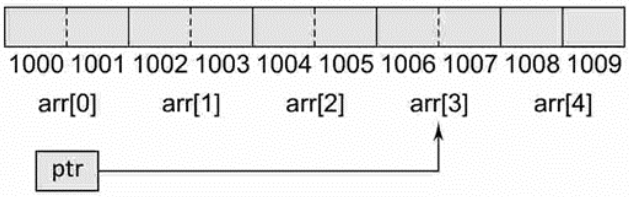


Figure 3.25 Pointer (ptr) pointing to the fourth element of the array

Programming Tip

When an array is passed to a function, we are actually passing a pointer to the function. Therefore, in the function declaration you must declare a pointer to receive the array name.

If `ptr` originally points to `arr[2]`, then `ptr++` will make it to point to the next element, i.e., `arr[3]`. This is shown in Fig. 3.25.

Had this been a character array, every byte in the memory would have been used to store an individual character. `ptr++` would then add only 1 byte to the address of `ptr`.

When using pointers, an expression like `arr[i]` is equivalent to writing `*(arr+i)`.

Many beginners get confused by thinking of array name as a pointer. For example, while we can write

```
ptr = arr;   // ptr = &arr[0]
```

we cannot write

```
arr = ptr;
```

This is because while `ptr` is a variable, `arr` is a constant. The location at which the first element of `arr` will be stored cannot be changed once `arr[]` has been declared. Therefore, an array name is often known to be a constant pointer.

To summarize, the name of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the element that it points to. Therefore, arrays and pointers use the same concept.

Note arr[i], i[arr], *(arr+i), *(i+arr) gives the same value.

Look at the following code which modifies the contents of an array using a pointer to an array.

```
int main()
{
      int arr[]={1,2,3,4,5};
      int *ptr, i;
      ptr=&arr[2];
      *ptr = -1;
      *(ptr+1) = 0;
      *(ptr-1) = 1;
      printf("\n Array is: ");
      for(i=0;i<5;i++)
            printf(" %d", *(arr+i));
      return 0;
}
```

Output

```
Array is: 1 1 -1 0 5
```

In C we can add or subtract an integer from a pointer to get a new pointer, pointing somewhere other than the original position. C also permits addition and subtraction of two pointer variables. For example, look at the code given below.

```
int main()
{
      int arr[]={1,2,3,4,5,6,7,8,9};
      int *ptr1, *ptr2;
      ptr1 = arr;
      ptr2 = arr+2;
      printf("%d", ptr2-ptr1);
      return 0;
}
```

Output

```
2
```

In the code, ptr1 and ptr2 are pointers pointing to the elements of the same array. We may subtract two pointers as long as they point to the same array. Here, the output is 2 because there are two elements between ptr1 and ptr2 in the array arr. Both the pointers must point to the same array or one past the end of the array, otherwise this behaviour cannot be defined.

Moreover, C also allows pointer variables to be compared with each other. Obviously, if two pointers are equal, then they point to the same location in the array. However, if one pointer is less than the other, it means that the pointer points to some element nearer to the beginning of the array. Like with other variables, relational operators (>, <, >=, etc.) can also be applied to pointer variables.

PROGRAMMING EXAMPLE

15. Write a program to display an array of given numbers.

```
#include <stdio.h>
int main()
{
        int arr[]={1,2,3,4,5,6,7,8,9};
        int *ptr1, *ptr2;
        ptr1 = arr;
        ptr2 = &arr[8];
        while(ptr1<=ptr2)
        {
                printf("%d", *ptr1);
                ptr1++;
        }
        return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9
```

3.8 ARRAYS OF POINTERS

An array of pointers can be declared as

```
int *ptr[10];
```

The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];
int p = 1, q = 2, r = 3, s = 4, t = 5;
ptr[0] = &p;
ptr[1] = &q;
ptr[2] = &r;
ptr[3] = &s;
ptr[4] = &t;
```

Can you tell what will be the output of the following statement?

```
printf("\n %d", *ptr[3]);
```

The output will be `4` because `ptr[3]` stores the address of integer variable `s` and `*ptr[3]` will therefore print the value of s that is `4`. Now look at another code in which we store the address of three individual arrays in the array of pointers:

```
int main()
{
      int arr1[]={1,2,3,4,5};
      int arr2[]={0,2,4,6,8};
      int arr3[]={1,3,5,7,9};
      int *parr[3] = {arr1, arr2, arr3};
      int i;
      for(i = 0;i<3;i++)
            printf(«%d», *parr[i]);
      return 0;
}
```

Output

```
1 0 1
```

Surprised with this output? Try to understand the concept. In the `for` loop, `parr[0]` stores the base address of `arr1` (or, `&arr1[0]`). So writing `*parr[0]` will print the value stored at `&arr1[0]`. Same is the case with `*parr[1]` and `*parr[2]`.

3.9 TWO-DIMENSIONAL ARRAYS

Till now, we have only discussed one-dimensional arrays. One-dimensional arrays are organized linearly in only one direction. But at times, we need to store data in the form of grids or tables. Here, the concept of single-dimension arrays is extended to incorporate two-dimensional data structures. A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column. The C compiler treats a two-dimensional array as an array of one-dimensional arrays. Figure 3.26 shows a two-dimensional array which can be viewed as an array of arrays.

3.9.1 Declaring Two-dimensional Arrays

Any array must be declared before being used. The declaration statement tells the compiler the name of the array, the data type of each element in the array, and the size of each dimension. A two-dimensional array is declared as:

```
data_type array_name[row_size][column_size];
```

Figure 3.26 Two-dimensional array

Therefore, a two-dimensional `m×n` array is an array that contains `m × n` data elements and each element is accessed using two subscripts, `i` and `j`, where `i<=m` and `j<=n`.

For example, if we want to store the marks obtained by three students in five different subjects, we can declare a two-dimensional array as:

```
int marks[3][5];
```

In the above statement, a two-dimensional array called `marks` has been declared that has `m(3)` rows and `n(5)` columns. The first element of the array is denoted by `marks[0][0]`, the second element as `marks[0][1]`, and so on. Here, `marks[0][0]` stores the marks obtained by the first student in the first subject, `marks[1][0]` stores the marks obtained by the second student in the first subject.

The pictorial form of a two-dimensional array is shown in Fig. 3.27.

Rows / Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Figure 3.27 Two-dimensional array

Hence, we see that a 2D array is treated as a collection of 1D arrays. Each row of a 2D array corresponds to a 1D array consisting of `n` elements, where `n` is the number of columns. To understand this, we can also see the representation of a two-dimensional array as shown in Fig. 3.28.

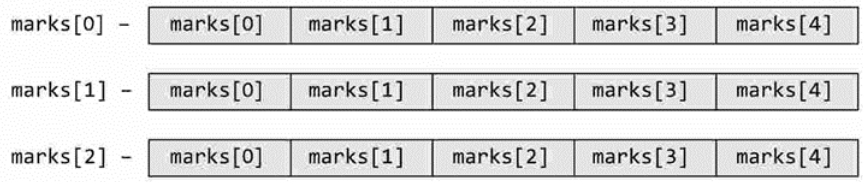


Figure 3.28 Representation of two-dimensional array `marks[3][5]`

Although we have shown a rectangular picture of a two-dimensional array, in the memory, these elements actually will be stored sequentially. There are two ways of storing a two-dimensional array in the memory. The first way is the *row major order* and the second is the *column major order*. Let us see how the elements of a 2D array are stored in a row major order. Here, the elements of the first row are stored before the elements of the second and third rows. That is, the elements of the array are stored row by row where `n` elements of the first row will occupy the first `n` locations. This is illustrated in Fig. 3.29.

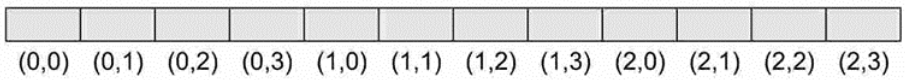


Figure 3.29 Elements of a 3 × 4 2D array in row major order

However, when we store the elements in a column major order, the elements of the first column are stored before the elements of the second and third column. That is, the elements of the array are stored column by column where `m` elements of the first column will occupy the first `m` locations. This is illustrated in Fig. 3.30.

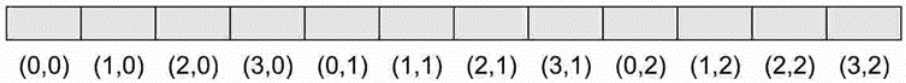


Figure 3.30 Elements of a 4 × 3 2D array in column major order

In one-dimensional arrays, we have seen that the computer does not keep track of the address of every element in the array. It stores only the address of the first element and calculates the address of other elements from the base address (address of the first element). Same is the case with a two-dimensional array. Here also, the computer stores the base address, and the address of the other elements is calculated using the following formula.

If the array elements are stored in column major order,

```
Address(A[I][J]) = Base_Address + w{M ( J - 1) + (I - 1)}
```

And if the array elements are stored in row major order,

```
Address(A[I][J]) = Base_Address + w{N ( I - 1) + (J - 1)}
```

where `w` is the number of bytes required to store one element, `N` is the number of columns, `M` is the number of rows, and `I` and `J` are the subscripts of the array element.

Example 3.5 Consider a 20 × 5 two-dimensional array `marks` which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, `marks[18][4]` assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J]) = Base_Address + w{N (I - 1) + (J - 1)}
Address(marks[18][4]) = 1000 + 2 {5(18 - 1) + (4 - 1)}
                      = 1000 + 2 {5(17) + 3}
                      = 1000 + 2 (88)
                      = 1000 + 176 = 1176
```

3.9.2 Initializing Two-dimensional Arrays

Like in the case of other variables, declaring a two-dimensional array only reserves space for the array in the memory. No values are stored in it. A two-dimensional array is initialized in the same way as a one-dimensional array is initialized. For example,

```
int marks[2][3]={90, 87, 78, 68, 62, 71};
```

Note that the initialization of a two-dimensional array is done row by row. The above statement can also be written as:

```
int marks[2][3]={{90,87,78},{68, 62, 71}};
```

The above two-dimensional array has two rows and three columns. First, the elements in the first row are initialized and then the elements of the second row are initialized.

Therefore,

```
marks[0][0] =   90      marks[0][1] =   87      marks[0][2] =   78
marks[1][0] =   68      marks[1][1] =   62      marks[1][2] =   71
```

In the above example, each row is defined as a one-dimensional array of three elements that are enclosed in braces. Note that the commas are used to separate the elements in the row as well as to separate the elements of two rows.

In case of one-dimensional arrays, we have discussed that if the array is completely initialized, we may omit the size of the array. The same concept can be applied to a two-dimensional array, except that only the size of the first dimension can be omitted. Therefore, the declaration statement given below is valid.

```
int marks[][3]={{90,87,78},{68, 62, 71}};
```

In order to initialize the entire two-dimensional array to zeros, simply specify the first value as zero. That is,

```
int marks[2][3] = {0};
```

The individual elements of a two-dimensional array can be initialized using the assignment operator as shown here.

```
marks[1][2] = 79;
```

or

```
marks[1][2] = marks[1][1] + 10;
```

3.9.3 Accessing the Elements of Two-dimensional Arrays

The elements of a 2D array are stored in contiguous memory locations. In case of one-dimensional arrays, we used a single `for` loop to vary the index `i` in every pass, so that all the elements could be scanned. Since the two-dimensional array contains two subscripts, we will use two `for` loops to scan the elements. The first `for` loop will scan each row in the 2D array and the second `for` loop will scan individual columns for every row in the array. Look at the programs which use two `for` loops to access the elements of a 2D array.

PROGRAMMING EXAMPLES

16. Write a program to print the elements of a 2D array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int arr[2][2] = {12, 34, 56,32};
        int i, j;
        for(i=0;i<2;i++)
        {
              printf("\n");
              for(j=0;j<2;j++)
                      printf("%d\t", arr[i][j]);
        }
        return 0;
}
```

Output

```
12      34
56      32
```

17. Write a program to generate Pascal's triangle.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int arr[7][7]={0};
        int row=2, col, i, j;
        arr[0][0] = arr[1][0] = arr[1][1] = 1;
        while(row <= 7)
        {
                arr[row][0] = 1;
                for(col = 1; col <= row; col++)
                arr[row][col] = arr[row-1][col-1] + arr[row-1][col];
                row++;
        }
        for(i=0; i<7; i++)
        {
                printf("\n");
                for(j=0; j<=i; j++)
```

```
marks[4][0] = 67
marks[4][1] = 78
marks[4][2] = 89
The highest marks obtained in the subject 1 = 99
The highest marks obtained in the subject 2 = 90
The highest marks obtained in the subject 3 = 100
```

3.10 OPERATIONS ON TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays can be used to implement the mathematical concept of matrices. In mathematics, a matrix is a grid of numbers, arranged in rows and columns. Thus, using two-dimensional arrays, we can perform the following operations on an $m \times n$ matrix:

Transpose Transpose of an $m \times n$ matrix A is given as a $n \times m$ matrix B, where $B_{i,j} = A_{j,i}$.

Sum Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing:

$C_{i,j} = A_{i,j} + B_{i,j}$

Difference Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be subtracted by writing:

$C_{i,j} = A_{i,j} - B_{i,j}$

Product Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, $m \times n$ matrix A can be multiplied with a $p \times q$ matrix B if $n=p$. The dimension of the product matrix is $m \times q$. The elements of two matrices can be multiplied by writing:

$C_{i,j} = \Sigma A_{i,k} B_{k,j}$ for $k=1$ to n

PROGRAMMING EXAMPLES

20. Write a program to read and display a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, j, mat[3][3];
        clrscr();
        printf("\n Enter the elements of the matrix ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                     scanf("%d",&mat[i][j]);
                }
        }
        printf("\n The elements of the matrix are ");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
```

```
                                printf("\t %d",mat[i][j]);
        }
        return 0;
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
```

21. Write a program to transpose a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, j, mat[3][3], transposed_mat[3][3];
        clrscr();
        printf("\n Enter the elements of the matrix ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        scanf("%d", &mat[i][j]);
                }
        }
        printf("\n The elements of the matrix are ");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                        printf("\t %d", mat[i][j]);
        }
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                        transposed_mat[i][j] = mat[j][i];
        }
        printf("\n The elements of the transposed matrix are ");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                        printf("\t %d",transposed_ mat[i][j]);
        }
        return 0;
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
The elements of the transposed matrix are
1 4 7
2 5 8
3 6 9
```

```
                {
                        res[i][j]=0;
                        for(k=0; k<res_cols;k++)
                        res[i][j] += mat1[i][k] * mat2[k][j];
                }
        }
        printf("\n The elements of the product matrix are ");
        for(i=0;i<res_rows;i++)
        {
                printf("\n");
                for(j=0;j<res_cols;j++)
                        printf("\t %d",res[i][j]);
        }
        return 0;
}
```

Output

```
Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the product matrix are
19 22
43 50
```

3.11 PASSING TWO-DIMENSIONAL ARRAYS TO FUNCTIONS

There are three ways of passing a two-dimensional array to a function. First, we can pass individual elements of the array. This is exactly the same as passing an element of a one-dimensional array. Second, we can pass a single row of the two-dimensional array. This is equivalent to passing the entire one-dimensional array to a function that has already been discussed in a previous section. Third, we can pass the entire two-dimensional array to the function. Figure 3.31 shows the three ways of using two-dimensional arrays for inter-functon communication.

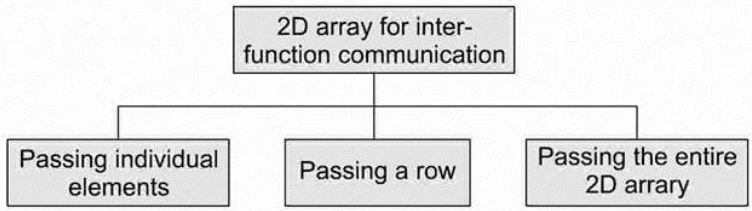


Figure 3.31 2D arrays for inter-function communication

Passing a Row

A row of a two-dimensional array can be passed by indexing the array name with the row number. Look at Fig. 3.32 which illustrates how a single row of a two-dimensional array can be passed to the called function.

Calling function	Called function

`main()`
`{`
`  int arr[2][3] = ({1, 2, 3}, {4, 5, 6});`
`  func(arr[1]);`
`}`
	`void func(int arr[])`
	`{`
	`  int i;`
	`  for(i=0;i<3;i++)`
	`  printf("%d", arr[i] * 10);`
	`}`

Figure 3.32 Passing a row of a 2D array to a function

Passing the Entire 2D Array

To pass a two-dimensional array to a function, we use the array name as the actual parameter (the way we did in case of a 1D array). However, the parameter in the called function must indicate that the array has two dimensions. Look at the following program which passes entire 2D array to a function.

Programming Example

24. Write a program to fill a square matrix with value zero on the diagonals, 1 on the upper right triangle, and –1 on the lower left triangle.

```
#include <stdio.h>
#include <conio.h>
void read_matrix(int mat[5][5], int);
void display_matrix(int mat[5][5], int);
int main()
{
        int row;
        int mat1[5][5];
        clrscr();
        printf("\n Enter the number of rows and columns of the matrix:");
        scanf("%d", &row);
        read_matrix(mat1, row);
        display_matrix(mat1, row);
        getch();
        return 0;
}

void read_matrix(int mat[5][5], int r)
{
        int i, j;
        for(i=0; i<r; i++)
        {
              for(j=0; j<r; j++)
              {
                    if(i==j)
                        mat[i][j] = 0;
                    else if(i>j)
                        mat[i][j] = -1;
                    else
                        mat[i][j] = 1;
              }
        }
}
void display_matrix(int mat[5][5], int r)
{
        int i, j;
```

A pointer to a three-dimensional array can be declared as,

```
int arr[2][2][2]={1,2,3,4,5,6,7,8};
int (*parr)[2][2];
parr = arr;
```

We can access an element of a three-dimensional array by writing,

```
arr[i][j][k] = *(*(*(arr+i)+j)+k)
```

PROGRAMMING EXAMPLE

27. Write a program which illustrates the use of a pointer to a three-dimensional array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i,j,k;
        int arr[2][2][2];
        int (*parr)[2][2]= arr;
        clrscr();
        printf("\n Enter the elements of a 2 × 2 × 2 array: ");
        for(i = 0; i < 2; i++)
        {
                for(j = 0; j < 2; j++)
                {
                        for(k = 0; k < 2; k++)
                                scanf("%d", &arr[i][j][k]);
                }
        }
        printf("\n The elements of the 2 × 2 × 2 array are: ");
        for(i = 0; i < 2; i++)
        {
                for(j = 0; j < 2; j++)
                {
                        for(k = 0; k < 2; k++)
                                printf("%d", *(*(*(parr+i)+j)+k));
                }
        }
        getch();
        return 0;
}
```

Output

```
Enter the elements of a 2 × 2 × 2 array: 1 2 3 4 5 6 7 8
The elements of the 2 × 2 × 2 array are: 1 2 3 4 5 6 7 8
```

Note In the printf statement, you could also have used `*(*(*(arr+i)+j+)+k)` instead of `*(*(*(parr+i)+j)+k))`.

3.15 SPARSE MATRICES

Sparse matrix is a matrix that has large number of elements with a zero value. In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure should be used. If we apply the operations using standard matrix structures and algorithms to sparse matrices, then the execution will slow down and the matrix will consume large amount of memory. Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

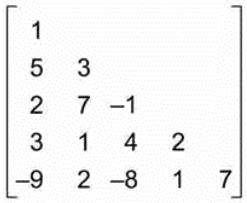


Figure 3.34 Lower-triangular matrix

There are two types of sparse matrices. In the first type of sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also called a (*lower*) *triagonal matrix* because if you see it pictorially, all the elements with a non-zero value appear below the diagonal. In a lower triangular matrix, $A_{i,j}=0$ where `i<j`. An `n×n` lower-triangular matrix `A` has one non-zero element in the first row, two non-zero elements in the second row and likewise `n` non-zero elements in the *n*th row. Look at Fig. 3.34 which shows a lower-triangular matrix.

Figure 3.35 Upper-triangular matrix

To store a lower-triangular matrix efficiently in the memory, we can use a one-dimensional array which stores only non-zero elements. The mapping between a two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:

(a) Row-wise mapping—Here the contents of array `A[]` will be {1, 5, 3, 2, 7, –1, 3, 1, 4, 2, –9, 2, –8, 1, 7}

(b) Column-wise mapping—Here the contents of array `A[]` will be {1, 5, 2, 3, –9, 3, 7, 1, 2, –1, 4, –8, 2, 1, 7}

In an *upper-triangular matrix*, $A_{i,j}=0$ where `i>j`. An `n×n` upper-triangular matrix `A` has `n` non-zero elements in the first row, `n-1` non-zero elements in the second row and likewise one non-zero element in the *n*th row. Look at Fig. 3.35 which shows an upper-triangular matrix.

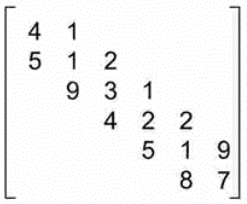


Figure 3.36 Tri-diagonal matrix

There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also called a *tri-diagonal matrix*. Hence in a tridiagonal matrix, $A_{i,j}=0$, where `|i - j| > 1`. In a tridiagonal matrix, if elements are present on

(a) the main diagonal, it contains non-zero elements for `i=j`. In all, there will be `n` elements.

(b) below the main diagonal, it contains non-zero elements for `i=j+1`. In all, there will be `n-1` elements.

(c) above the main diagonal, it contains non-zero elements for `i=j-1`. In all, there will be `n-1` elements.

Figure 3.36 shows a tri-diagonal matrix. To store a tri-diagonal matrix efficiently in the memory, we can use a one-dimensional array that stores only non-zero elements. The mapping between a two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:

(a) Row-wise mapping—Here the contents of array `A[]` will be {4, 1, 5, 1, 2, 9, 3, 1, 4, 2, 2, 5, 1, 9, 8, 7}

(b) Column-wise mapping—Here the contents of array `A[]` will be {4, 5, 1, 1, 9, 2, 3, 4, 1, 2, 5, 2, 1, 8, 9, 7}

(c) Diagonal-wise mapping—Here the contents of array `A[]` will be {5, 9, 4, 5, 8, 4, 1, 3, 2, 1, 7, 1, 2, 1, 2, 9}

3.16 APPLICATIONS OF ARRAYS

Arrays are frequently used in C, as they have a number of useful applications. These applications are

- Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.
- Many databases include one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables. We will read about these data structures in the subsequent chapters.
- Arrays can be used for sorting elements in ascending or descending order.

Points to Remember

- An array is a collection of elements of the same data type.
- The elements of an array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- The index specifies an offset from the beginning of the array to the element being referenced.
- Declaring an array means specifying three parameters: data type, name, and its size.
- The length of an array is given by the number of elements stored in it.
- There is no single function that can operate on all the elements of an array. To access all the elements, we must use a loop.
- The name of an array is a symbolic reference to the address of the first byte of the array. Therefore, whenever we use the array name, we are actually referring to the first byte of that array.
- C considers a two-dimensional array as an array of one-dimensional arrays.
- A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second subscript denotes the column of the array.
- Using two-dimensional arrays, we can perform the different operations on matrices: transpose, addition, subtraction, multiplication.
- A multi-dimensional array is an array of arrays. Like we have one index in a one-dimensional array, two indices in a two-dimensional array, in the same way we have n indices in an n-dimensional or multi-dimensional array. Conversely, an n-dimensional array is specified using n indices.
- Multi-dimensional arrays can be stored in either row major order or column major order.
- Sparse matrix is a matrix that has large number of elements with a zero value.
- There are two types of sparse matrices. In the first type, all the elements above the main diagonal have a zero value. This type of sparse matrix is called a lower-triangular matrix. In the second type, all the elements below the main diagonal have a zero value. This type of sparse matrix is called an upper-triangular matrix.
- There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of sparse matrix is called a tridiagonal matrix.

Exercises

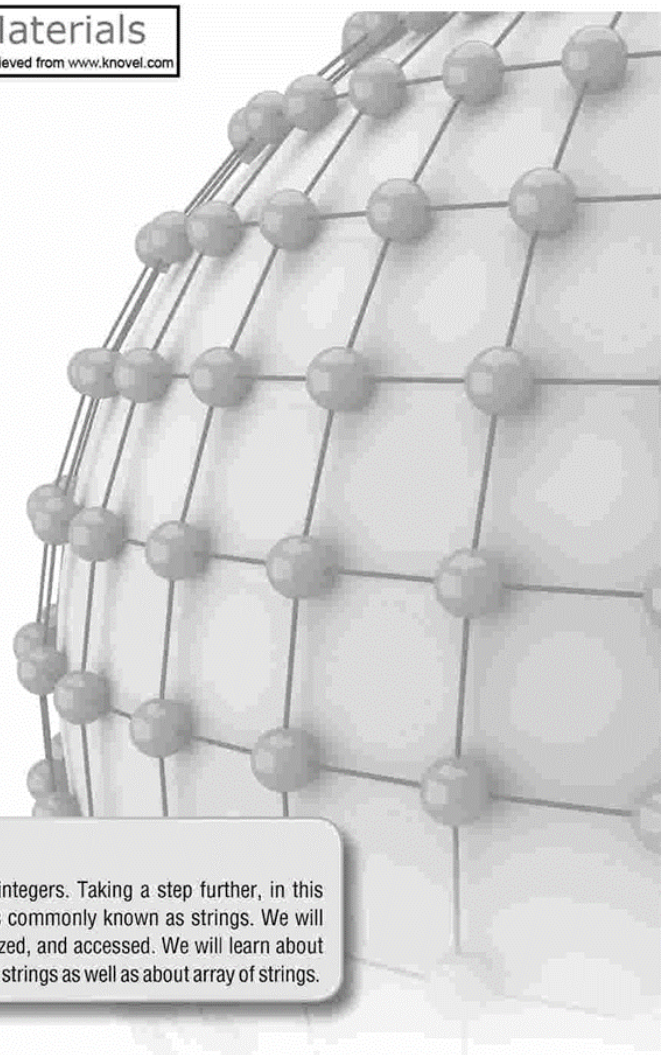
Review Questions

1. What are arrays and why are they needed?
2. How is an array represented in the memory?
3. How is a two-dimensional array represented in the memory?
4. What is the use of multi-dimensional arrays?
5. Explain sparse matrix.
6. How are pointers used to access two-dimensional arrays?
7. Why does storing of sparse matrices need extra consideration? How are sparse matrices stored efficiently in the computer's memory?
8. For an array declared as `int arr[50]`, calculate the address of `arr[35]`, if `Base(arr)=1000` and `w=2`.
9. Consider a two-dimensional array `Marks[10][5]` having its base address as 2000 and the number of bytes per element of the array is 2. Now, compute the address of the element, `Marks[8][5]`, assuming that the elements are stored in row major order.
10. How are arrays related to pointers?
11. Briefly explain the concept of array of pointers.
12. How can one-dimensional arrays be used for inter-function communication?
13. Consider a two-dimensional array `arr[10][10]` which has base address = 1000 and the number of bytes per element of the array = 2. Now, compute the address of the element `arr[8][5]` assuming that the elements are stored in column major order.
14. Consider the array given below:

`Name[0]`	Adam
`Name[1]`	Charles
`Name[2]`	Dicken
`Name[3]`	Esha
`Name[4]`	Georgia
`Name[5]`	Hillary
`Name[6]`	Mishael

CHAPTER 4

Strings



LEARNING OBJECTIVE

In the last chapter, we discussed array of integers. Taking a step further, in this chapter, we will discuss array of characters commonly known as strings. We will see how strings are stored, declared, initialized, and accessed. We will learn about different operations that can be performed on strings as well as about array of strings.

4.1 INTRODUCTION

Nowadays, computers are widely used for word processing applications such as creating, inserting, updating, and modifying textual data. Besides this, we need to search for a particular pattern within a text, delete it, or replace it with another pattern. So, there is a lot that we as users do to manipulate the textual data.

In C, a string is a null-terminated character array. This means that after the last character, a `null` character (`'\0'`) is stored to signify the end of the character array. For example, if we write

```
char str[] = "HELLO";
```

then we are declaring an array that has five characters, namely, `H`, `E`, `L`, `L`, and `O`. Apart from these characters, a `null` character (`'\0'`) is stored at the end of the string. So, the internal representation of the string becomes `HELLO'\0'`. To store a string of length 5, we need 5 + 1 locations (1 extra for the `null` character). The name of the character array (or the string) is a pointer to the beginning of the string. Figure 4.1 shows the difference between character storage and string storage.

If we had declared `str` as

```
char str[5] = "HELLO";
```

then the `null` character will not be appended automatically to the character array. This is because `str` can hold only 5 characters and the characters in `HELLO` have already filled the space allocated to it.

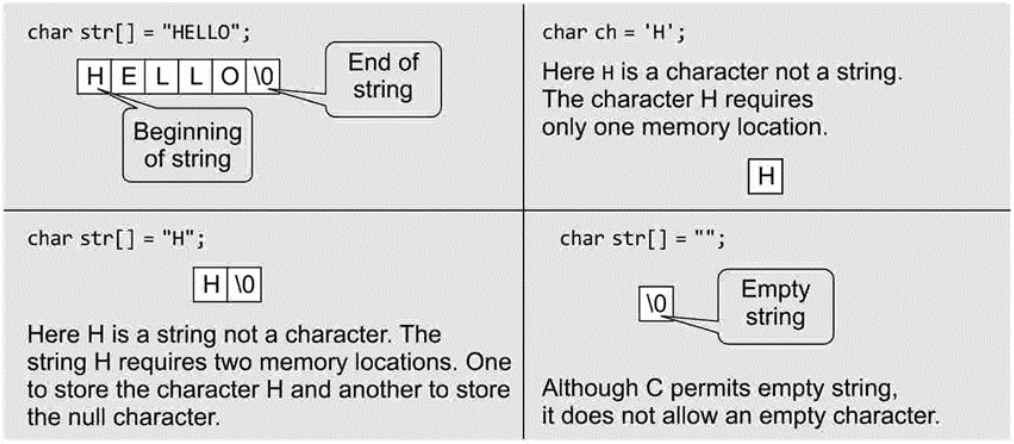


Figure 4.1 Difference between character storage and string storage

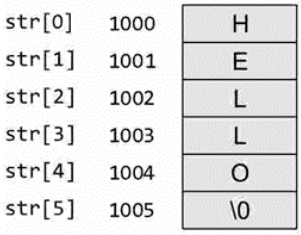


Figure 4.2 Memory representation of a character array

Programming Tip

When allocating memory space for a string, reserve space to hold the null character also.

Like we use subscripts (also known as index) to access the elements of an array, we can also use subscripts to access the elements of a string. The subscript starts with a zero (0). All the characters of a string are stored in successive memory locations. Figure 4.2 shows how `str[]` is stored in the memory.

Thus, in simple terms, a string is a sequence of characters. In Fig. 4.2, 1000, 1001, 1002, etc., are the memory addresses of individual characters. For simplicity, the figure shows that `H` is stored at memory location 1000 but in reality, the ASCII code of a character is stored in the memory and not the character itself. So, at address 1000, 72 will be stored as the ASCII code for `H` is 72.

The statement

```
char str[] = "HELLO";
```

declares a constant string, as we have assigned a value to it while declaring the string. However, the general form of declaring a string is

```
char str[size];
```

When we declare the string like this, we can store `size-1` characters in the array because the last character would be the null character. For example, `char mesg[100];` can store a maximum of 99 characters.

Till now, we have only seen one way of initializing strings. The other way to initialize a string is to initialize it as an array of characters. For example,

```
char str[] = {'H', 'E', 'L', 'L', 'O', '\0'};
```

In this example, we have explicitly added the null character. Also observe that we have not mentioned the size of the string. Here, the compiler will automatically calculate the size based on the number of characters. So, in this example six memory locations will be reserved to store the string variable, `str`.

We can also declare a string with size much larger than the number of elements that are initialized. For example, consider the statement below.

```
char str [10] = "HELLO";
```

In such cases, the compiler creates an array of size 10; stores "HELLO" in it and finally terminates the string with a null character. Rest of the elements in the array are automatically initialized to NULL.

Now consider the following statements:

```
char str[3];
str = "HELLO";
```

The above initialization statement is illegal in C and would generate a compile-time error because of two reasons. First, the array is initialized with more elements than it can store. Second, initialization cannot be separated from declaration.

4.1.1 Reading Strings

If we declare a string by writing

```
char str[100];
```

Then str can be read by the user in three ways:

1. using scanf function,
2. using gets() function, and
3. using getchar(),getch()or getche() function repeatedly.

Strings can be read using scanf() by writing

```
scanf("%s", str);
```

Although the syntax of using scanf() function is well known and easy to use, the main pitfall of using this function is that the function terminates as soon as it finds a blank space. For example, if the user enters Hello World, then the str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function. You may also specify a field width to indicate the maximum number of characters that can be read. Remember that extra characters are left unconsumed in the input buffer.

Programming Tip

Using & operand with a string variable in the scanf statement generates an error.

Unlike int, float, and char values, %s format does not require the ampersand before the variable str.

The next method of reading a string is by using the gets() function. The string can be read by writing

```
gets(str);
```

gets() is a simple function that overcomes the drawbacks of the scanf() function. The gets() function takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

Strings can also be read by calling the getchar() function repeatedly to read a sequence of single characters (unless a terminating character is entered) and simultaneously storing it in a character array as shown below.

```
i=0;()
ch = getchar;// Get a character
while(ch != '*')
{
      str[i] = ch;// Store the read character in str
      i++;
      ch = getchar();// Get another character
}
str[i] = '\0';// Terminate str with null character
```

Note that in this method, you have to deliberately append the string with a null character. The other two functions automatically do this.

4.1.2 Writing Strings

Strings can be displayed on the screen using the following three ways:

1. using `printf()` function,
2. using `puts()` function, and
3. using `putchar()` function repeatedly.

Strings can be displayed using `printf()` by writing

```
printf("%s", str);
```

We use the format specifier `%s` to output a string. Observe carefully that there is no '&' character used with the string variable. We may also use width and precision specifications along with `%s`. The width specifies the minimum output field width. If the string is short, the extra space is either left padded or right padded. A negative width left pads short string rather than the default right justification. The precision specifies the maximum number of characters to be displayed, after which the string is truncated. For example,

```
printf ("%5.3s", str);
```

The above statement would print only the first three characters in a total field of five characters. Also these characters would be right justified in the allocated width. To make the string left justified, we must use a minus sign. For example,

```
printf ("%-5.3s", str);
```

Note When the field width is less than the length of the string, the entire string will be printed. If the number of characters to be printed is specified as zero, then nothing is printed on the screen.

The next method of writing a string is by using `puts()` function. A string can be displayed by writing

```
puts(str);
```

`puts()` is a simple function that overcomes the drawbacks of the `printf()` function.

Strings can also be written by calling the `putchar()` function repeatedly to print a sequence of single characters.

```
i=0;
while(str[i] != '\0')
{
      putchar(str[i]);// Print the character on the screen
      i++;
}
```

4.2 OPERATIONS ON STRINGS

In this section, we will learn about different operations that can be performed on strings.

Finding Length of a String

The number of characters in a string constitutes the length of the string. For example, `LENGTH("C PROGRAMMING IS FUN")` will return 20. Note that even blank spaces are counted as characters in the string.

Figure 4.3 shows an algorithm that calculates the length of a string. In this algorithm, `I` is used as an index for traversing string `STR`. To traverse each and every character of `STR`, we increment the value of `I`.

```
Step 1: [INITIALIZE] SET I = 0
Step 2: Repeat Step 3 while STR[I] != NULL
Step 3:    SET I = I + 1
       [END OF LOOP]
Step 4: SET LENGTH = I
Step 5: END
```

Figure 4.3 Algorithm to calculate the length of a string

Once we encounter the `null` character, the control jumps out of the `while` loop and the length is initialized with the value of `I`.

Note The library function `strlen(s1)` which is defined in `string.h` returns the length of string `s1`.

PROGRAMMING EXAMPLE

1. Write a program to find the length of a string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        char str[100], i = 0, length;
        clrscr();
        printf("\n Enter the string : ");
        gets(str)
        while(str[i] != '\0')
                i++;
        length = i;
        printf("\n The length of the string is : %d", length);
        getch()
        return 0;
}
```

Output

```
Enter the string : HELLO
The length of the string is : 5
```

```
Step 1: [INITIALIZE] SET I=0
Step 2: Repeat Step 3 while STR[I] != NULL
Step 3:   IF STR[I] >= 'a' AND STR[I] <= '2'
            SET UPPERSTR[I] = STR[I] -32
          ELSE
            SET UPPERSTR[I] = STR[I]
          [END OF IF]
          SET I = I + 1
        [END OF LOOP]
Step 4: SET UPPERSTR[I] = NULL
Step 5: EXIT
```

Figure 4.4 Algorithm to convert characters of a string into upper case

Converting Characters of a String into Upper/ Lower Case

We have already discussed that in the memory ASCII codes are stored instead of the real values. The ASCII code for `A-Z` varies from 65 to 91 and the ASCII code for `a-z` ranges from 97 to 123. So, if we have to convert a lower case character into uppercase, we just need to subtract 32 from the ASCII value of the character. And if we have to convert an upper case character into lower case, we need to add 32 to the ASCII value of the character. Figure 4.4 shows an algorithm that converts the lower case characters of a string into upper case.

Note The library functions `toupper()` and `tolower()` which are defined in `ctype.h` convert a character into upper and lower case, respectively.

In the algorithm, we initialize `I` to zero. Using `I` as the index of `STR`, we traverse each character of `STR` from Step 2 to 3. If the character is in lower case, then it is converted into upper case by subtracting 32 from its ASCII value. But if the character is already in upper case, then it is copied into the `UPPERSTR` string. Finally, when all the characters have been traversed, a `null` character is appended to `UPPERSTR` (as done in Step 4).

PROGRAMMING EXAMPLE

2. Write a program to convert the lower case characters of a string into upper case.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        char str[100], upper_str[100];
        int i=0;
        clrscr();
        printf("\n Enter the string :");
        gets(str);
        while(str[i] != '\0')
        {
                if(str[i]>='a' && str[i]<='z')
                        upper_str[i] = str[i] - 32;
                else
                        upper_str[i] = str[i];
                i++;
        }
        upper_str[i] = '\0';
        printf("\n The string converted into upper case is : ");
        puts(upper_str);
        return 0;
}
```

Output

```
Enter the string : Hello
The string converted into upper case is : HELLO
```

Appending a String to Another String

Appending one string to another string involves copying the contents of the source string at the end of the destination string. For example, if `S1` and `S2` are two strings, then appending `S1` to `S2` means we have to add the contents of `S1` to `S2`. So, `S1` is the source string and `S2` is the destination string. The appending operation would leave the source string `S1` unchanged and the destination string `S2 = S2 + S1`. Figure 4.5 shows an algorithm that appends two strings.

Note The library function `strcat(s1, s2)` which is defined in `string.h` concatenates string `s2` to `s1`.

```
Step 1: [INITIALIZE] SET I=0 and J=0
Step 2: Repeat Step 3 while DEST_STR[I] != NULL
Step 3:         SET I = I + 1
        [END OF LOOP]
Step 4: Repeat Steps 5 to 7 while SOURCE_STR[J] != NULL
Step 5:         DEST_STR[I] = SOURCE_STR[J]
Step 6:         SET I = I + 1
Step 7:         SET J = J + 1
        [END OF LOOP]
Step 8: SET DEST_STR[I] = NULL
Step 9: EXIT
```

Figure 4.5 Algorithm to append a string to another string

In this algorithm, we first traverse through the destination string to reach its end, that is, reach the position where a `null` character is encountered. The characters of the source string are then

```
                {
                        k++;
                        j++;
                }
                if(pat[j]=='\0')
                {
                        copy_loop=k;
                        while(rep_pat[rep_index] !='\0')
                        {
                        new_str[n] = rep_pat[rep_index];
                        rep_index++;
                        n++;
                        }
                }
                new_str[n] = str[copy_loop];
                i++;
                copy_loop++;
                n++;
        }
        new_str[n]='\0';
        printf("\n The new string is : ");
        puts(new_str);
        getch();
        return 0;
}
```

Output

```
Enter the string : How ARE you?
Enter the pattern to be replaced : ARE
Enter the replacing pattern : are
The new string is : How are you?
```

4.3 ARRAYS OF STRINGS

Till now we have seen that a string is an array of characters. For example, if we say `char name[] = "Mohan"`, then the name is a string (character array) that has five characters.

Now, suppose that there are 20 students in a class and we need a string that stores the names of all the 20 students. How can this be done? Here, we need a string of strings or an array of strings. Such an array of strings would store 20 individual strings. An array of strings is declared as

```
char names[20][30];
```

Here, the first index will specify how many strings are needed and the second index will specify the length of every individual string. So here, we will allocate space for 20 names where each name can be a maximum 30 characters long.

Let us see the memory representation of an array of strings. If we have an array declared as

```
char name[5][10] = {"Ram", "Mohan", "Shyam", "Hari", "Gopal"};
```

Then in the memory, the array will be stored as shown in Fig. 4.13.

name[0]	R	A	M	'\0'						
name[1]	M	O	H	A	N	'\0'				
name[2]	S	H	Y	A	M	'\0'				
name[3]	H	A	R	I	'\0'					
name[4]	G	O	P	A	L	'\0'				

Figure 4.13 Memory representation of a 2D character array

```
Step 1: [INITIALIZE] SET I=0
Step 2: Repeat Step 3 while I< N
Step 3:     Apply Process to NAMES[I]
        [END OF LOOP]
Step 4: EXIT
```

Figure 4.14 Algorithm to process individual string from an array of strings

By declaring the array names, we allocate 50 bytes. But the actual memory occupied is 27 bytes. Thus, we see that about half of the memory allocated is wasted. Figure 4.14 shows an algorithm to process individual string from an array of strings.

In Step 1, we initialize the index variable `I` to zero. In Step 2, a `while` loop is executed until all the strings in the array are accessed. In Step 3, each individual string is processed.

Programming Examples

10. Write a program to sort the names of students.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
        char names[5][10], temp[10];
        int i, n, j;
        clrscr();
        printf("\n Enter the number of students : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n Enter the name of student %d : ", i+1);
                gets(names[i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(strcmp(names[j], names[j+1])>0)
                        {
                                strcpy(temp, names[j]);
                                strcpy(names[j], names[j+1]);
                                strcpy(names[j+1], temp);
                        }
                }
        }
        printf("\n Names of the students in alphabetical order are : ");
        for(i=0;i<n;i++)
                puts(names[i]);
        getch();
        return 0;
}
```

Output

```
Enter the number of students : 3
Enter the name of student 1 : Goransh
Enter the name of student 2 : Aditya
Enter the name of student 3 : Sarthak
Names of the students in alphabetical order are : Aditya Goransh Sarthak
```

11. Write a program to read multiple lines of text and then count the number of characters, words, and lines in the text.

```
#include <stdio.h>
```

Points to Remember

- A string is a null-terminated character array.
- Individual characters of strings can be accessed using a subscript that starts from zero.
- All the characters of a string are stored in successive memory locations.
- Strings can be read by a user using three ways: using `scanf()` function, using `gets()` function, or using `getchar()` function repeatedly.
- The `scanf()` function terminates as soon as it finds a blank space.
- The `gets()` function takes the starting address of the string which will hold the input. The string inputted using `gets()` is automatically terminated with a `null` character.
- Strings can also be read by calling `getchar()` repeatedly to read a sequence of single characters.
- Strings can be displayed on the screen using three ways: using `printf` function, using `puts()` function, or using `putchar()` function repeatedly.
- C standard library supports a number of pre-defined functions for manipulating strings or changing the contents of strings. Many of these functions are defined in the header file `string.h`.
- Alternatively we can also develop functions which perform the same task as the pre-defined string handling functions. The most basic function is the `length` function which returns the number of characters in a string.
- Name of a string acts as a pointer to the string. In the declaration `char str[5] = "hello";` `str` is a pointer which holds the address of the first character, i.e., `'h'`.
- An array of strings can be declared as `char strings [20][30];` where the first subscript denotes the number of strings and the second subscript denotes the length of every individual string.

Exercises

Review Questions

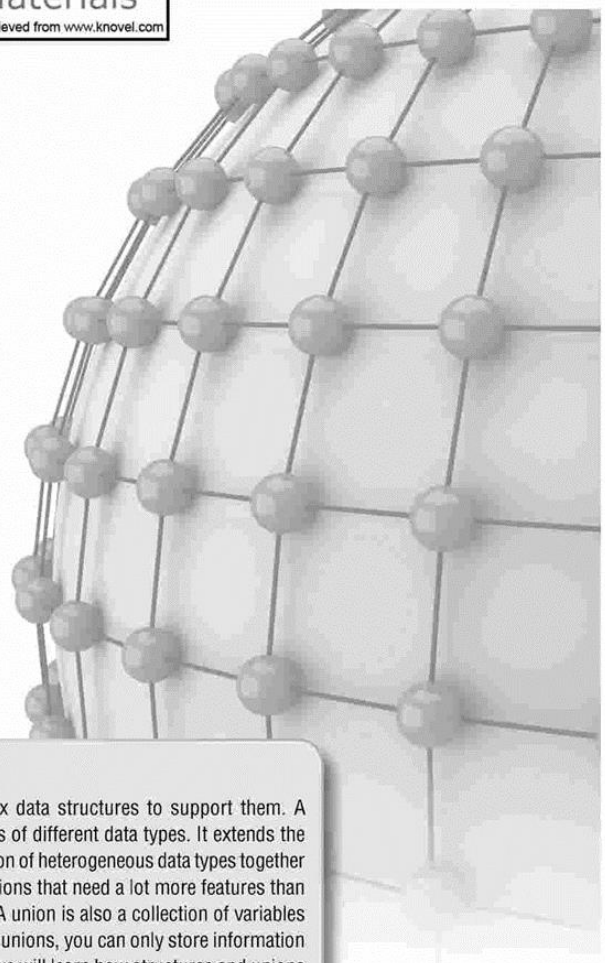
1. What are strings? Discuss some of the operations that can be performed on strings.
2. Explain how strings are represented in the main memory.
3. How are strings read from the standard input device? Explain the different functions used to perform the string input operation.
4. Explain how strings can be displayed on the screen.
5. Explain the syntax of `printf()` and `scanf()`.
6. List all the substrings that can be formed from the string 'ABCD'.
7. What do you understand by pattern matching? Give an algorithm for it.
8. Write a short note on array of strings.
9. Explain with an example how an array of strings is stored in the main memory.
10. Explain how pointers and strings are related to each other with the help of a suitable program.
11. If the substring function is given as `SUBSTRING (string, position, length)`, then find `S(5,9)` if `S = "Welcome to World of C Programming"`
12. If the index function is given as `INDEX(text, pattern)`, then find `index(T, P)` where `T = "Welcome to World of C Programming"` and `P = "of"`
13. Differentiate between `gets()` and `scanf()`.
14. Give the drawbacks of getchar() and scanf().
15. Which function can be used to overcome the shortcomings of `getchar()` and `scanf()`?
16. How can `putchar()` be used to print a string?
17. Differentiate between a character and a string.
18. Differentiate between a character array and a string.

Programming Exercises

1. Write a program in which a string is passed as an argument to a function.
2. Write a program in C to concatenate first `n` characters of a string with another string.
3. Write a program in C that compares first `n` characters of one string with first `n` characters of another string.
4. Write a program in C that removes leading and trailing spaces from a string.
5. Write a program in C that replaces a given character with another character in a string.

CHAPTER 5

Structures and Unions



LEARNING OBJECTIVE

Today's modern applications need complex data structures to support them. A structure is a collection of related data items of different data types. It extends the concept of arrays by storing related information of heterogeneous data types together under a single name. It is useful for applications that need a lot more features than those provided by the primitive data types. A union is also a collection of variables of different data types, except that in case of unions, you can only store information in one field at any one time. In this chapter, we will learn how structures and unions are declared, defined, and accessed using the C language.

5.1 INTRODUCTION

A structure is in many ways similar to a record. It stores related information about an entity. Structure is basically a user-defined data type that can store related information (even of different data types) together. The major difference between a structure and an array is that an array can store only information of same data type.

A structure is therefore a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure.

5.1.1 Structure Declaration

A structure is declared using the keyword `struct` followed by the structure name. All the variables of the structure are declared within the structure. A structure type is generally declared by using the following syntax:

```
struct struct-name
{
      data_type var-name;
```

Programming Tip

Do not forget to place a semicolon after the declaration of structures and unions.

```
    data_type var-name;
    ..............
};
```

For example, if we have to define a structure for a student, then the related information for a student probably would be: `roll_number`, `name`, `course`, and `fees`. This structure can be declared as:

```
struct student
{
      int r_no;
      char name[20];
      char course[20];
      float fees;
};
```

Now the structure has become a user-defined data type. Each variable name declared within a structure is called a member of the structure. The structure declaration, however, does not allocate any memory or consume storage space. It just gives a template that conveys to the C compiler how the structure would be laid out in the memory and also gives the details of member names. Like any other data type, memory is allocated for the structure when we declare a variable of the structure. For example, we can define a variable of `student` by writing:

```
struct student stud1;
```

Here, `struct student` is a data type and `stud1` is a variable. Look at another way of declaring variables. In the following syntax, the variables are declared at the time of structure declaration.

```
struct student
{
   int r_no;
   char name[20];
   char course[20];
   float fees;
} stud1, stud2;
```

In this declaration we declare two variables `stud1` and `stud2` of the structure `student`. So if you want to declare more than one variable of the structure, then separate the variables using a comma. When we declare variables of the structure, separate memory is allocated for each variable. This is shown in Fig. 5.1.

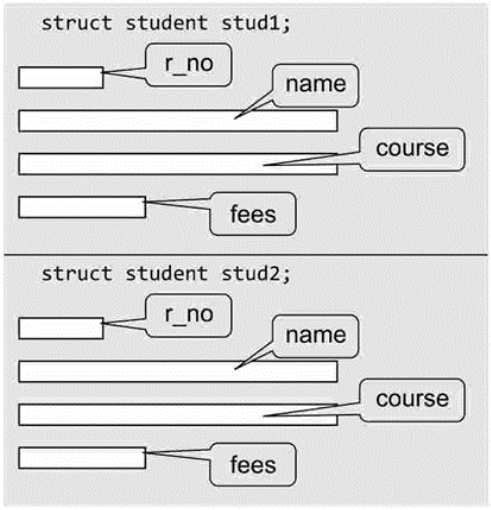


Figure 5.1 Memory allocation for a structure variable

Note Structure type and variable declaration of a structure can be either local or global depending on their placement in the code.

Last but not the least, structure member names and names of the structure follow the same rules as laid down for the names of ordinary variables. However, care should be taken to ensure that the name of structure and the name of a structure member should not be the same. Moreover, structure name and its variable name should also be different.

5.1.2 `Typedef` Declarations

The `typedef` (derived from type definition) keyword enables the programmer to create a new data type name by using an existing data type. By using `typedef`, no new data is created, rather an

alternate name is given to a known data type. The general syntax of using the `typedef` keyword is given as:

```
typedef existing_data_type new_data_type;
```

Programming Tip

C does not allow declaration of variables at the time of creating a typedef definition. So variables must be declared in an independent statement.

Note that `typedef` statement does not occupy any memory; it simply defines a new type. For example, if we write

```
typedef int INTEGER;
```

then `INTEGER` is the new name of data type `int`. To declare variables using the new data type name, precede the variable name with the data type name (new). Therefore, to define an integer variable, we may now write

```
INTEGER num=5;
```

When we precede a `struct` name with the `typedef` keyword, then the `struct` becomes a new type. It is used to make the construct shorter with more meaningful names for types already defined by C or for types that you have declared. For example, consider the following declaration:

```
typedef struct student
{
      int r_no;
      char name[20];
      char course[20];
      float fees;
};
```

Now that you have preceded the structure's name with the `typedef` keyword, `student` becomes a new data type. Therefore, now you can straightaway declare the variables of this new data type as you declare the variables of type `int`, `float`, `char`, `double`, etc. To declare a variable of structure `student`, you may write

```
student stud1;
```

Note that we have not written `struct student stud1`.

5.1.3 Initialization of Structures

A structure can be initialized in the same way as other data types are initialized. Initializing a structure means assigning some constants to the members of the structure. When the user does not explicitly initialize the structure, then C automatically does it. For `int` and `float` members, the values are initialized to zero, and `char` and string members are initialized to `'\0'` by default.

The initializers are enclosed in braces and are separated by commas. However, care must be taken to ensure that the initializers match their corresponding types in the structure definition.

The general syntax to initialize a structure variable is as follows:

```
struct struct_name
{
      data_type member_name1;
      data_type member_name2;
      data_type member_name3;
      .....................
}struct_var = {constant1, constant2, constant3,...};
```

or

```
struct struct_name
{
      data_type member_name1;
      data_type member_name2;
      data_type member_name3;
      .....................
};
```

```
struct struct_name struct_var = {constant1, constant2, constant 3,...};
```

For example, we can initialize a student structure by writing,

```
struct student
{
   int r_no;
   char name[20];
   char course[20];
   float fees;
}stud1 = {01, "Rahul", "BCA", 45000};
```

Programming Tip

It is an error to assign a structure of one type to a structure of another type.

Or, by writing,

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
```

Figure 5.2 illustrates how the values will be assigned to individual fields of the structure.

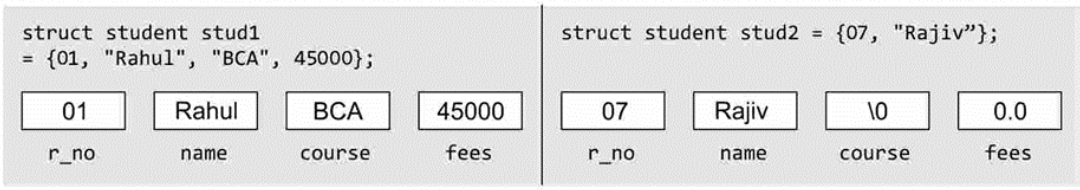


Figure 5.2 Assigning values to structure elements

When all the members of a structure are not initialized, it is called partial initialization. In case of partial initialization, first few members of the structure are initialized and those that are uninitialized are assigned default values.

5.1.4 Accessing the Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a `'.'` (dot) operator. The syntax of accessing a structure or a member of a structure can be given as:

```
struct_var.member_name
```

The dot operator is used to select a particular member of the structure. For example, to assign values to the individual data members of the structure variable `stud1`, we may write

```
stud1.r_no = 01;
stud1.name = "Rahul";
stud1.course = "BCA";
stud1.fees = 45000;
```

To input values for data members of the structure variable `stud1`, we may write

```
scanf("%d", &stud1.r_no);
scanf("%s", stud1.name);
```

Similarly, to print the values of structure variable `stud1`, we may write

```
printf("%s", stud1.course);
printf("%f", stud1.fees);
```

Memory is allocated only when we declare the variables of the structure. In other words, the memory is allocated only when we instantiate the structure. In the absence of any variable, structure definition is just a template that will be used to reserve memory when a variable of type `struct` is declared.

Once the variables of a structure are defined, we can perform a few operations on them. For example, we can use the assignment operator `(=)` to assign the values of one variable to another.

Note Of all the operators –>, . , ( ), and [] have the highest priority. This is evident from the following statement
`stud1.fees++` will be interpreted as `(stud1.fees)++`.

5.1.5 Copying and Comparing Structures

We can assign a structure to another structure of the same type. For example, if we have two structure variables `stud1` and `stud2` of type `struct student` given as

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
struct student stud2;
```

Then to assign one structure variable to another, we will write

```
stud2 = stud1;
```

This statement initializes the members of `stud2` with the values of members of `stud1`. Therefore, now the values of `stud1` and `stud2` can be given as shown in Fig. 5.3.

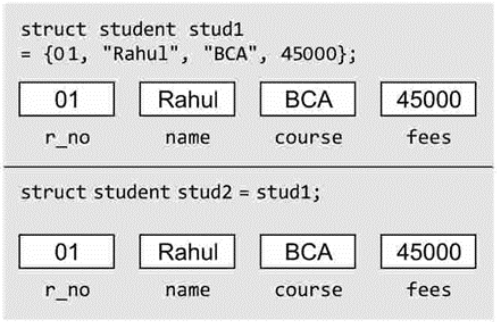


Figure 5.3 Values of structure variables

Programming Tip

An error will be generated if you try to compare two structure variables.

C does not permit comparison of one structure variable with another. However, individual members of one structure can be compared with individual members of another structure. When we compare one structure member with another structure's member, the comparison will behave like any other ordinary variable comparison. For example, to compare the fees of two students, we will write

```
if(stud1.fees > stud2.fees) //to check if fees of stud1 is
greater than stud2
```

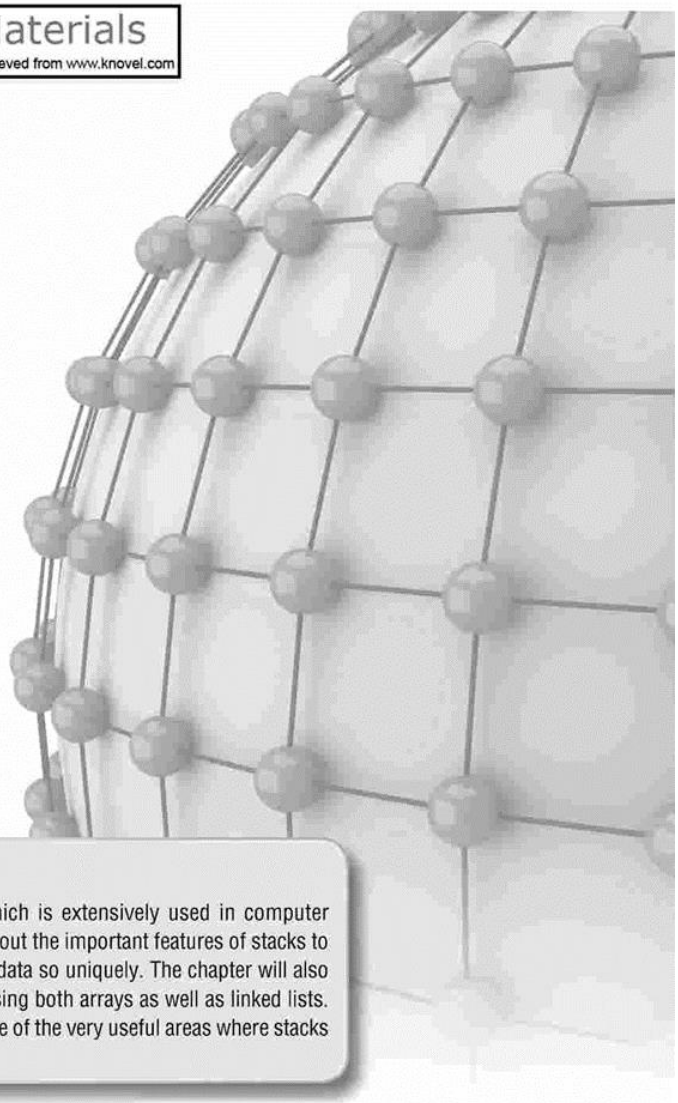
Programming Examples

1. Write a program using structures to read and display the information about a student.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        struct student
        {
                int roll_no;
                char name[80];
                float fees;
                char DOB[80];
        };
        struct student stud1;
        clrscr();
        printf("\n Enter the roll number : ");
        scanf("%d", &stud1.roll_no);
        printf("\n Enter the name : ");
        scanf("%s", stud1.name);
        printf("\n Enter the fees : ");
        scanf("%f", &stud1.fees);
        printf("\n Enter the DOB : ");
        scanf("%s", stud1.DOB);
        printf("\n ********STUDENT'S DETAILS *******");
        printf("\n ROLL No. = %d", stud1.roll_no);
        printf("\n NAME = %s", stud1.name);
        printf("\n FEES = %f", stud1.fees);
```

CHAPTER 7

Stacks



LEARNING OBJECTIVE

A stack is an important data structure which is extensively used in computer applications. In this chapter we will study about the important features of stacks to understand how and why they organize the data so uniquely. The chapter will also illustrate the implementation of stacks by using both arrays as well as linked lists. Finally, the chapter will discuss in detail some of the very useful areas where stacks are primarily used.

7.1 INTRODUCTION

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. 7.1. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

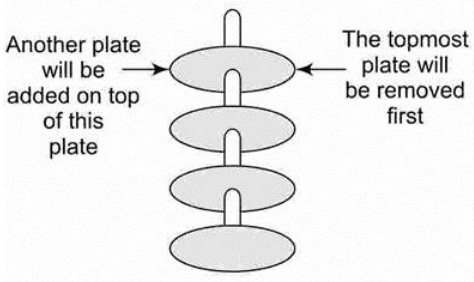


Figure 7.1 Stack of plates

Now the question is where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

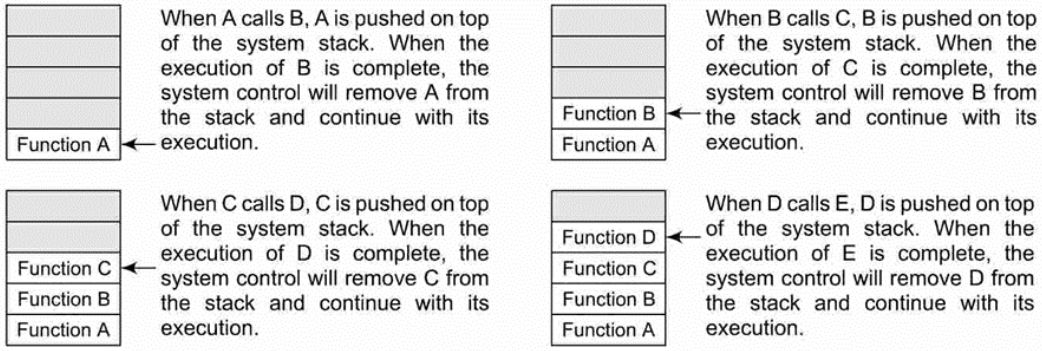


Figure 7.2 System stack in the case of function calls

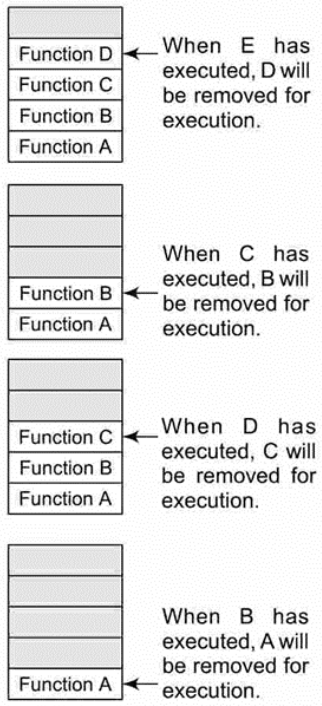


Figure 7.3 System stack when a called function returns to the calling function

In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used. Whenever a function calls another function, the calling function is pushed onto the top of the stack. This is because after the called function gets executed, the control is passed back to the calling function. Look at Fig. 7.2 which shows this concept.

Now when function `E` is executed, function `D` will be removed from the top of the stack and executed. Once function `D` gets completely executed, function `C` will be removed from the stack for execution. The whole procedure will be repeated until all the functions get executed. Let us look at the stack after each function is executed. This is shown in Fig. 7.3.

The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.

Stacks can be implemented using either arrays or linked lists. In the following sections, we will discuss both array and linked list implementation of stacks.

7.2 ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called `TOP` associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called `MAX`, which is used to store the maximum number of elements that the stack can hold.

If `TOP = NULL`, then it indicates that the stack is empty and if `TOP = MAX-1`, then the stack is full. (You must be wondering why we have written `MAX-1`. It is because array indices start from 0.) Look at Fig. 7.4.

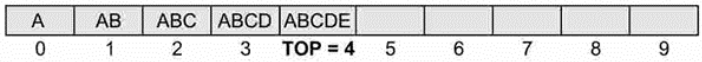


Figure 7.4 Stack

The stack in Fig. 7.4 shows that `TOP = 4`, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

7.3 OPERATIONS ON A STACK

A stack supports three basic operations: `push`, `pop`, and `peek`. The `push` operation adds an element to the top of the stack and the `pop` operation removes the element from the top of the stack. The `peek` operation returns the value of the topmost element of the stack.

7.3.1 Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if `TOP=MAX-1`, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an `OVERFLOW` message is printed. Consider the stack given in Fig. 7.5.

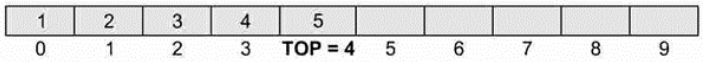


Figure 7.5 Stack

To insert an element with value 6, we first check if `TOP=MAX-1`. If the condition is false, then we increment the value of `TOP` and store the new element at the position given by `stack[TOP]`. Thus, the updated stack becomes as shown in Fig. 7.6.

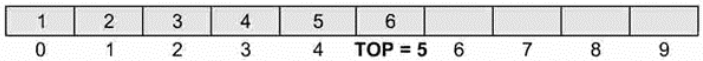


Figure 7.6 Stack after insertion

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Figure 7.7 Algorithm to insert an element in a stack

Figure 7.7 shows the algorithm to insert an element in a stack. In Step 1, we first check for the `OVERFLOW` condition. In Step 2, `TOP` is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by `TOP`.

7.3.2 Pop Operation

The `pop` operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if `TOP=NULL` because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an `UNDERFLOW` message is printed. Consider the stack given in Fig. 7.8.

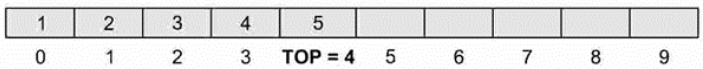


Figure 7.8 Stack

To delete the topmost element, we first check if `TOP=NULL`. If the condition is false, then we decrement the value pointed by `TOP`. Thus, the updated stack becomes as shown in Fig. 7.9.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Figure 7.10 Algorithm to delete an element from a stack

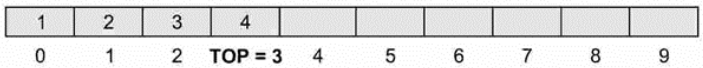


Figure 7.9 Stack after deletion

Figure 7.10 shows the algorithm to delete an element from a stack. In Step 1, we first check for the `UNDERFLOW` condition. In Step 2, the value of the location in the stack pointed by `TOP` is stored in `VAL`. In Step 3, `TOP` is decremented.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Figure 7.11 Algorithm for Peek operation

7.3.3 Peek Operation

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack. The algorithm for Peek operation is given in Fig. 7.11.

However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned. Consider the stack given in Fig. 7.12.

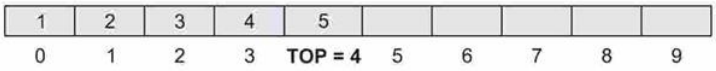


Figure 7.12 Stack

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

PROGRAMMING EXAMPLE

1. Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created

int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);

int main(int argc, char *argv[]) {
        int val, option;
        do
        {
                printf("\n *****MAIN MENU*****");
                printf("\n 1. PUSH");
                printf("\n 2. POP");
                printf("\n 3. PEEK");
                printf("\n 4. DISPLAY");
                printf("\n 5. EXIT");
                printf("\n Enter your option: ");
                scanf("%d", &option);
                switch(option)
                {
                case 1:
                        printf("\n Enter the number to be pushed on stack: ");
                        scanf("%d", &val);
                        push(st, val);
                        break;
                case 2:
                        val = pop(st);
                        if(val != -1)
                        printf("\n The value deleted from stack is: %d", val);
                        break;
                case 3:
                        val = peek(st);
                        if(val != -1)
```

```
                            case 4: val=popB();
                                    if(val!=-999)
                                       printf("\n The value popped from Stack B = %d",val);
                                    break;
                            case 5: printf("\n The contents of Stack A are : \n");
                                    display_stackA();
                                    break;
                            case 6: printf("\n The contents of Stack B are : \n");
                                    display_stackB();
                                    break;
                        }
            }while(option!=7);
            getch();
    }
```

Output

```
*****MAIN MENU*****
1. PUSH IN STACK A
2. PUSH IN STACK B
3. POP FROM STACK A
4. POP FROM STACK B
5. DISPLAY STACK A
6. DISPLAY STACK B
7. EXIT
Enter your choice : 1
Enter the value to push on Stack A : 10
Enter the value to push on Stack A : 15
Enter your choice : 5
The content of Stack A are:
15      10
Enter your choice : 4
UNDERFLOW
Enter your choice : 7
```

7.7 APPLICATIONS OF STACKS

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

7.7.1 Reversing a List

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

PROGRAMMING EXAMPLE

4. Write a program to reverse a list of given numbers.

```
#include <stdio.h>
```

```
char pop();
void main()
{
        char exp[MAX],temp;
        int i, flag=1;
        clrscr();
        printf("Enter an expression : ");
        gets(exp);
        for(i=0;i<strlen(exp);i++)
        {
                if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
                        push(exp[i]);
                if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
                        if(top == -1)
                                flag=0;
                        else
                        {
                                temp=pop();
                                if(exp[i]==')' && (temp=='{' || temp=='['))
                                        flag=0;
                                if(exp[i]=='}' && (temp=='(' || temp=='['))
                                        flag=0;
                                if(exp[i]==']' && (temp=='(' || temp=='{'))
                                        flag=0;
                        }
        }
        if(top>=0)
                flag=0;
        if(flag==1)
                printf("\n Valid expression");
        else
                printf("\n Invalid expression");
}
void push(char c)
{
        if(top == (MAX-1))
                printf("Stack Overflow\n");
        else
        {
                top=top+1;
                stk[top] = c;
        }
}
char pop()
{
        if(top == -1)
                printf("\n Stack Underflow");
        else
                return(stk[top--]);
}
```

Output

```
Enter an expression : (A + (B - C))
Valid Expression
```

7.7.3 Evaluation of Arithmetic Expressions

Polish Notations

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions.

While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, `A+B;` here, `plus` operator is placed between the two operands `A` and `B`. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and associativity rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix notation was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as `A+B` in infix notation, the same expression can be written as `AB+` in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

The expression `(A + B) * C` can be written as:

`[AB+]*C`

`AB+C*` in the postfix notation

A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation `AB+C*`. While evaluation, addition will be performed prior to multiplication.

Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, `AB+C*`, `+` is applied on `A` and `B`, then `*` is applied on the result of addition and `C`.

Although a prefix notation is also evaluated from left to right, the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if `A+B` is an expression in infix notation, then the corresponding expression in prefix notation is given by `+AB`.

While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator. Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.

Example 7.1 Convert the following infix expressions into postfix expressions.

Solution

```
(a) (A-B) * (C+D)
    [AB-] * [CD+]
    AB-CD+*
(b) (A + B) / (C + D) - (D * E)
    [AB+] / [CD+] - [DE*]
    [AB+CD+/] - [DE*]
    AB+CD+/DE*-
```

Example 7.2 Convert the following infix expressions into prefix expressions.

Solution

```
(a) (A + B) * C
    (+AB)*C
    *+ABC
(b) (A-B) * (C+D)
    [-AB] * [+CD]
    *-AB+CD
(c) (A + B) / ( C + D) - ( D * E)
    [+AB] / [+CD] - [*DE]
    [/+AB+CD] - [*DE]
    -/+AB+CD*DE
```

Conversion of an Infix Expression into a Postfix Expression

Let `I` be an algebraic expression written in infix notation. `I` may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only `+`, `-`, `*`, `/`, `%` operators. The precedence of these operators can be given as follows:

Higher priority `*`, `/`, `%`

Lower priority `+`, `-`

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression `A + B * C`, then first `B * C` will be done and the result will be added to `A`. But the same expression if written as, `(A + B) * C`, will evaluate `A + B` first and then the result will be multiplied with `C`.

The algorithm given below transforms an infix expression into postfix expression, as shown in Fig. 7.22. The algorithm accepts an infix expression that may contain operators, operands, and parentheses. For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (–) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Figure 7.22 Algorithm to convert an infix notation to postfix notation

Solution

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

Example 7.3 Convert the following infix expression into postfix expression using the algorithm given in Fig. 7.22.

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)

Programming Example

6. Write a program to convert an infix expression into its equivalent postfix notation.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char st[MAX];
int top=-1;
void push(char st[], char);
char pop(char st[]);
```

```
                    {
                            printf("\n INCORRECT ELEMENT IN EXPRESSION");
                            exit(1);
                    }
            }
            while((top!=-1) && (st[top]!='('))
            {
                    target[j] = pop(st);
                    j++;
            }
            target[j]='\0';
    }
    int getPriority(char op)
    {
            if(op=='/' || op == '*' || op=='%')
                    return 1;
            else if(op=='+' || op=='-')
                    return 0;
    }
    void push(char st[], char val)
    {
            if(top==MAX-1)
                    printf("\n STACK OVERFLOW");
            else
            {
                    top++;
                    st[top]=val;
            }
    }
    char pop(char st[])
    {
            char val=' ';
            if(top==-1)
                    printf("\n STACK UNDERFLOW");
            else
            {
                    val=st[top];
                    top--;
            }
            return val;
    }
```

Output

```
Enter any infix expression : A+B-C*D
The corresponding postfix expression is : AB+CD*-
```

Evaluation of a Postfix Expression

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack. Let us look at Fig. 7.23 which shows the algorithm to evaluate a postfix expression.

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

Figure 7.23 Algorithm to evaluate a postfix expression

Table 7.1 Evaluation of a postfix expression

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Let us now take an example that makes use of this algorithm. Consider the infix expression given as `9 - ((3 * 4) + 8) / 4`. Evaluate the expression.

The infix expression `9 - ((3 * 4) + 8) / 4` can be written as `9 3 4 * 8 + 4 / -` using postfix notation. Look at Table 7.1, which shows the procedure.

Programming Example

7. Write a program to evaluate a postfix expression.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=-1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
        float val;
        char exp[100];
        clrscr();
        printf("\n Enter any postfix expression : ");
        gets(exp);
        val = evaluatePostfixExp(exp);
        printf("\n Value of the postfix expression = %.2f", val);
        getch();
        return 0;
}
float evaluatePostfixExp(char exp[])
{
        int i=0;
        float op1, op2, value;
        while(exp[i] != '\0')
        {
                if(isdigit(exp[i]))
```

```
    > getPriority(source[i]))
                                {
                                        target[j] = pop(st);
                                        j++;
                                }
                                push(st, source[i]);
                                i++;
                        }
                        else
                        {
                                printf("\n INCORRECT ELEMENT IN EXPRESSION");
                                exit(1);
                        }
                }
                while((top!=-1) && (st[top]!='('))
                {
                        target[j] = pop(st);
                        j++;
                }
                target[j]='\0';
        }
        int getPriority( char op)
        {
                if(op=='/' || op == '*' || op=='%')
                        return 1;
                else if(op=='+' || op=='-')
                        return 0;
        }
        void push(char st[], char val)
        {
                if(top==MAX-1)
                        printf("\n STACK OVERFLOW");
                else
                {
                        top++;
                        st[top] = val;
                }
         }
        char pop(char st[])
        {
                char val=' ';
                if(top==-1)
                        printf("\n STACK UNDERFLOW");
                else
                {
                        val=st[top];
                        top--;
                }
                return val;
        }
```

Output

```
Enter any infix expression : A+B-C*D
The corresponding postfix expression is : AB+CD*-
The prefix expression is : -+AB*CD
```

Evaluation of a Prefix Expression

There are a number of techniques for evaluating a prefix expression. The simplest way of evaluation of a prefix expression is given in Fig. 7.26.

```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
        (a) Scan the prefix expression
            from right, one character at a
            time.
        (b) If the scanned character is an
            operand, push it on the
            operand stack.
        (c) If the scanned character is an
            operator, then
              (i) Pop two values from the
                  operand stack
             (ii) Apply the operator on
                  the popped operands
            (iii) Push the result on the
                  operand stack
Step 3: END
```

Figure 7.26 Algorithm for evaluation of a prefix expression

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29

For example, consider the prefix expression `+ - 9 2 7 * 8 / 4 12`. Let us now apply the algorithm to evaluate this expression.

Programming Example

9. Write a program to evaluate a prefix expression.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int stk[10];
int top=-1;
int pop();
void push(int);
int main()
{
    char prefix[10];
    int len, val, i, opr1, opr2, res;
    clrscr();
    printf("\n Enter the prefix expression : ");
    gets(prefix);
    len = strlen(prefix);
      for(i=len-1;i>=0;i--)
      {
    switch(get_type(prefix[i]))
            {
               case 0:
                    val = prefix[i] - '0';
                    push(val);
                    break;
               case 1:
                    opr1 = pop();
                    opr2 = pop();
                    switch(prefix[i])
                    {
                         case '+':
                              res = opr1 + opr2;
                              break;
                         case '-':
                              res = opr1 - opr2;
                              break;
                         case '*':
                              res = opr1 * opr2;
                              break;
                         case '/':
                              res = opr1 / opr2;
                              break;
                    }
                    push(res);
            }
      }
      printf("\n RESULT = %d", stk[0]);
      getche();
      return 0;
}
void push(int val)
{
      stk[++top] = val;
```