UNIT 9 STRUCTURES AND UNIONS

Structure

9.0 INTRODUCTION

We have seen so far how to store numbers, characters, strings, and even large sets of these primitives using arrays, but what if we want to store collections of different kinds of data that are somehow related. For example, a file about an employee will probably have his/her name, age, the hours of work, salary, etc. Physically, all of that is usually stored in someone's filing cabinet. In programming, if you have lots of related information, you group it together in an organized fashion. Let's say you have a group of employees, and you want to make a database! It just wouldn't do to have tons of loose variables hanging all over the place. Then we need to have a single data entity where we will be able to store all the related information together. But this can't be achieved by using the arrays alone, as in the case of arrays, we can group multiple data elements that are of the same data type, and is stored in consecutive memory locations, and is individually accessed by a subscript. That is where the user-defined datatype *Structures* come in.

Structure is commonly referred to as a user-defined data type. C's *structures* allow you to store multiple variables of any type in one place (the structure). A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. They can hold any number of variables, and you can make arrays of structures. This flexibility makes structures ideally useful for creating databases in C. Similar to the structure there is another user defined data type called *Union* which allows the programmer to view a single storage in more than one way i.e., a variable declared as union can store within its storage space, the data of different types, at different times. In this unit, we will be discussing the user-defined data type structures and unions.

9.1 OBJECTIVES

After going through this unit you should be able to:

- declare and initialize the members of the structures;
- access the members of the structures;
- pass the structures as function arguments;
- declare the array of structures;
- declare and define union; and
- perform all operations on the variables of type Union.

9.2 DECLARATION OF STRUCTURES

To declare a structure you must start with the keyword ***struct*** followed by the *structure name* or *structure tag* and within the braces the list of the structure's member variables. Note that the structure declaration does not actually create any variables. The syntax for the structure declaration is as follows:

```
struct structure-tag {
                    datatype variable1;
                    datatype variable2;
                    dataype variable 3;
                        ...
                    };
```

For example, consider the student database in which each student has a roll number, name and course and the marks obtained. Hence to group this data with a structure-tag as ***student***, we can have the declaration of structure as:

```
struct student {
                    int roll_no;
                    char name[20];
                    char course[20];
                    int marks_obtained ;
                  };
```

The point you need to remember is that, till this time no memory is allocated to the structure. This is only the definition of structure that tells us that there exists a user-defined data type by the name of student which is composed of the following members. Using this structure type, we have to create the structure variables:

```
 struct student stud1, stud2 ;
```

At this point, we have created two instances or structure variables of the user-defined data type student. Now memory will be allocated. The amount of memory allocated will be the sum of all the data members which form part of the structure template.

The second method is as follows:

```
struct {
          int roll_no;
          char name[20];
          char course[20];
          int marks_obtained ;
       }  stud1, stud2 ;
```

In this case, a tag name *student* is missing, but still it happens to be a valid declaration of structure. In this case the two variables are allocated memory equivalent to the members of the structure.

The advantage of having a tag name is that we can declare any number of variables of the tagged named structure later in the program as per requirement.

If you have a small structure that you just want to define in the program, you can do the definition and declaration together as shown below. This will define a structure of type *struct telephone* and declare three instances of it.

Consider the example for declaring and defining a structure for the telephone billing with three instances:

```
struct telephone{
                        int tele_no;
                        int cust_code;
```

```
                char cust_address[40];
                int bill_amt;
        }       tele1, tele2, tele3;
```

The structure can also be declared by using the typedefinition or typedef. This can be done as shown below:

```
typedef struct country{
                char name[20];
                int population;
                char language[10];
              } Country;
```

This defines a structure which can be referred to either as *struct country* or *Country*, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you are not going to use one or the other. But it is a standard practice to put them both in and to give them the same name, but the one after the braces starts with an uppercase letter.

The *typedef* statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the *typedef* above will be of type ***struct country,*** just like population is of type integer. The structure variables can be now defined as below:

Country Mexico, Canada, Brazil;

9.3 ACCESSING THE MEMBERS OF A STRUCTURE

Individual structure members can be used like other variables of the same type. Structure members are accessed using *the structure member operator* (.), also called the *dot operator*, between the structure name and the member name. The syntax for accessing the member of the structure is:

structurevariable. member-name;

Let us take the example of the coordinate structure.

```
struct coordinate{
                int x;
                int y;
            };
```

Thus, to have the structure named first refer to a screen location that has coordinates

x=50, y=100, you could write as,

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write,

```
printf ("%d,%d", second.x, second.y);
```

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Now let us see the following program to clarify our concepts. For example, let us see, how will we go about storing and retrieving values of the individual data members of the student structure.

Example 9.1

```
/*Program to store and retrieve the values from the student structure*/
```

```
#include<stdio.h>
struct student {
                    int roll_no;
                    char name[20];
                    char course[20];
                    int marks_obtained ;
          };
main()
{
student s1 ;
printf (“Enter the  student roll number:”);
scanf (“%d”,&s1.roll_no);
printf (“\nEnter the student name: “);
scanf (“%s”,s1.name);
printf (“\nEnter the student  course”);
scanf (“%s”,s1.course);
printf (“Enter the student percentage\n”);
scanf (“%d”,&s1.marks_obtained);
printf (“\nData entry is complete”);
printf ( “\nThe data entered is as follows:\n”);
printf (“\nThe student roll no is %d”,s1.roll_no);
printf (“\nThe student name is %s”,s1.name);
printf (“\nThe student course is %s”,s1.course);
printf (“\nThe student percentage is %d”,s1.marks_obtained);
}
```

OUTPUT

```
Enter the student roll number: 1234
Enter the student name: ARUN
Enter the student course: MCA
Enter the student percentage: 84
Date entry is complete

The data entered is as follows:
The student roll no is 1234
The student name is ARUN
The student course is MCA
The student percentage is 84
```

Another way of accessing the storing the values in the members of a structure is by initializing them to some values at the time when we create an instance of the data type.

9.4 INITIALIZING STRUCTURES

Like other C variable types, structures can be initialized when they’re declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces. For example, look at the following statements for initializing the values of the members of the *mysale* structure variable.

Example 9.2

```
struct sale {
              char customer[20];
             char item[20];
              float amt;
               } mysale = { "XYZ Industries",
```

```
                    "toolskit",
                    600.00
                  };
```

In a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

Example 9.3

```
struct customer {
                char firm[20];
                char contact[25];
                }

struct sale {
            struct customer buyer1;
            char item [20];
            float amt;
            } mysale = {
                        { "XYZ Industries", "Tyran Adams"},
                        "toolskit",
                        600.00
                       };
```

These statements perform the following initializations:

- the structure member *mysale.buyer1.firm* is initialized to the string "XYZ Industries".
- the structure member *mysale.buyer1.contact* is initialized to the string "Tyran Adams".
- the structure member *mysale.item* is initialized to the string "toolskit".
- the structure member *mysale.amount* is initialized to the amount 600.00.

For example let us consider the following program where the data members are initialized to some value.

Example 9.4

Write a program to access the values of the structure initialized with some initial values.

```
/* Program to illustrate to access the values of the structure initialized with some
initial values*/

#include<stdio.h>
struct telephone{
        int tele_no;
        int cust_code;
        char cust_name[20];
        char cust_address[40];
        int bill_amt;
        };
main()
{
 struct telephone tele = {2314345,
                          5463,
                          "Ram",
                          "New Delhi",
```

```
                                2435    };

printf("The values are initialized in this program.");
printf("\nThe telephone number is  %d",tele.tele_no);
printf("\nThe customer code is  %d",tele.cust_code);
printf("\nThe customer name is  %s",tele.cust_name);
printf("\nThe customer address is  %s",tele.cust_address);
printf("\nThe bill amount is   %d",tele.bill_amt);
}
```

OUTPUT

```
The values are initialized in this program.
The telephone number is 2314345
The customer code is 5463
The customer name is Ram
The customer Address is New Delhi
The bill amount is 2435
```

Check Your Progress 1

1. What is the difference between the following two declarations?
   ```
   struct x1{…….};
   typedef struct{……}x2;
   ```

……

……

2. Why can't you compare structures?

……

……..

3. Why does size of report a larger size than, one expects, for a structure type, as if there were padding at the end?

……

……

4. Declare a structure and instance together to display the date.

……

……

9.5 STRUCTURES AS FUNCTION ARGUMENTS

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. Let us look into the techniques of passing the structures to the functions. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers. We will be concentrating on the first method in this unit and passing using pointers will be taken up in the next unit. Like other data types, a structure can be passed as an argument to a function. The program listing given below shows how to do this. It uses a function to display data on the screen.

Example 9.5

Write a program to demonstrate passing a structure to a function.

```
/*Program to demonstrate passing a structure to a function.*/

#include <stdio.h>

/*Declare and define a structure to hold the data.*/

struct data{
        float amt;
        char fname [30];
        char lname [30];
        } per;

main()
{
void print_per (struct data x);
printf("Enter the donor's first and last names separated by a space:");
scanf ("%s %s", per.fname, per.lname);
printf ("\nEnter the amount donated in rupees:");
scanf ("%f", &per.amt);
print_per (per);
return 0;
 }

void print_per(struct data x)
{
  printf ("\n %s %s gave donation of amount Rs.%.2f.\n", x.fname, x.lname, x.amt);
}
```

OUTPUT

```
Enter the donor's first and last names separated by a space: RAVI KANT
Enter the amount donated in rupees: 1000.00
RAVI KANT gave donation of the amount Rs. 1000.00.
```

You can also pass a structure to a function by passing the structure's address (that is, a pointer to the structure which we will be discussing in the next unit). In fact, in the older versions of C, this was the only way to pass a structure as an argument. It is not necessary now, but you might see the older programs that still use this method. If you pass a pointer to a structure as an argument, remember that you must use the indirect membership operator (→) to access structure members in the function.

Please note the following points with respect to passing the structure as a parameter to a function.

- The return value of the called function must be declared as the value that is being returned from the function. If the function is returning the entire structure then the return value should be declared as *struct* with appropriate tag name.
- The actual and formal parameters for the structure data type must be the same as the *struct* type.
- The return statement is required only when the function is returning some data.
- When the return values of type is *struct*, then it must be assigned to the structure of identical type in the calling function.

Let us consider another example as shown in the Example 9.6, where *structure salary* has three fields related to an employee, namely - *name, no_days_worked* and d*aily_wage*. To accept the values from the user we first call the function *get_data* that

gets the values of the members of the structure. Then using the *wages* function we calculate the salary of the person and display it to the user.

Example 9.6

Write a program to accept the data from the user and calculate the salary of the person using concept of functions.

```
/* Program to accept the data from the user and calculate the salary of the person*/

#include<stdio.h>
main()
{
  struct sal     {
           char name[30];
           int no_days_worked;
           int daily_wage;       };
           struct sal salary;
           struct sal get_dat(struct);          /* function prototype*/
           float  wages(struct);                /*function prototype*/
           float amount_payable;                /* variable declaration*/
           salary = get_data(salary);
           printf("The name of employee is %s",salary.name);
           printf("Number of days worked is %d",salary.no_daya_worked);
           printf("The daily wage of the employees is %d",salary.daily_wage);
          amount_payable = wages(salary);
           printf("The amount payable to %s is %f",salary.name,amount_payable);
}

struct sal get_data(struct sal income)
           {
           printf("Please enter the employee name:\n");
           scanf("%s",income.name);
           printf("Please enter the number of days worked:\n");
           scanf("%d",&income.no_days_worked);
           printf('Please enter the employee daily wages:\n");
           scanf("%d",&income.daily_wages);
          return(income);
         }

float wages(struct)
{
   struct sal amt;
   int total_salary ;
   total_salary = amt.no_days_worked * amt.daily_wages;
   return(total_salary);     }
```

Check Your Progress 2

1. How is structure passing and returning implemented?

……

……

……

2. How can I pass constant values to functions which accept structure arguments?

……

……

……

3. What will be the output of the program?

```
#include<stdio.h>
main( )
{
struct pqr{
      int x ;
     };
struct pqr  pqr ;
pqr.x =10 ;
printf ("%d", pqr.x);
}
```

……

……

……

9.6 STRUCTURES AND ARRAYS

Thus far we have studied as to how the data of heterogeneous nature can be grouped together and be referenced as a single unit of structure. Now we come to the next step in our real world problem. Let's consider the example of students and their marks. In this case, to avoid declaring various data variables, we grouped together all the data concerning the student's marks as one unit and call it student. The problem that arises now is that the data related to students is not going to be of a single student only. We will be required to store data for a number of students. To solve this situation one way is to declare a structure and then create sufficient number of variables of that structure type. But it gets very cumbersome to manage such a large number of data variables, so a better option is to declare an array.

So, revising the array for a few moments we would refresh the fact that an array is simply a collection of homogeneous data types. Hence, if we make a declaration as:

int temp[20];

It simply means that temp is an array of twenty elements where each element is of type integer, indicating homogenous data type. Now in the same manner, to extend the concept a bit further to the structure variables, we would say,

struct student stud[20] ;

It means that *stud* is an array of twenty elements where each element is of the type *struct student* (which is a user-defined data type we had defined earlier). The various members of the *stud* array can be accessed in the similar manner as that of any other ordinary array.

For example,
struct student stud[20], we can access the *roll_no* of this array as

stud[0].roll_no;
stud[1].roll_no;
stud[2].roll_no;
stud[3].roll_no;

```
…
…
…
stud[19].roll_no;
```

Please remember the fact that for an array of twenty elements the subscripts of the array will be ranging from 0 to 19 (a total of twenty elements). So let us now start by seeing how we will write a simple program using array of structures.

Example 9.7

Write a program to read and display data for 20 students.

```
/*Program to read and print the data for 20 students*/

#include <stdio.h>
struct student {  int roll_no;
                  char name[20];
                  char course[20];
                  int marks_obtained ;
                 };
main( )
{
 struct student stud [20];
 int i;
 printf (“Enter the student data one by one\n”);
 for(i=0; i<=19; i++)
  {
     printf (“Enter the roll number of %d student”,i+1);
     scanf (“%d”,&stud[i].roll_no);
     printf (“Enter the name of %d student”,i+1);
     scanf (“%s”,stud[i].name);
     printf (“Enter the course of %d student”,i+1);
     scanf (“%d”,stud[i].course);
     printf (“Enter the marks obtained of %d student”,i+1);
     scanf (“%d”,&stud[i].marks_obtained);
   }
      printf (“the data entered is as follows\n”);
      for (i=0;i<=19;i++)
  {
     printf (“The roll number of  %d student is %d\n”,i+1,stud[i].roll_no);
     printf (“The name of  %d student is %s\n”,i+1,stud[i].name);
     printf (“The course of  %d student is %s\n”,i+1,stud[i].course);
     printf (“The marks of  %d student is %d\n”,i+1,stud[i].marks_obtained);
    }
}
```

The above program explains to us clearly that the array of structure behaves as any other normal array of any data type. Just by making use of the subscript we can access all the elements of the structure individually.

Extending the above concept where we can have arrays as the members of the structure. For example, let’s see the above example where we have taken a structure for the student record. Hence in this case it is a real world requirement that each student will be having marks of more than one subject. Hence one way to declare the structure, if we consider that each student has 3 subjects, will be as follows:

```
struct student {
                  int roll_no;
                  char name [20];
```

```
    char course [20];
    int subject1 ;
    int subject2;
    int subject3;
  };
```

The above described method is rather a bit cumbersome, so to make it more efficient we can have an array inside the structure, that is, we have an array as the member of the structure.

```
struct student {
    int roll_no;
    char name [20];
    char course [20];
    int subject [3] ;
  };
```

Hence to access the various elements of this array we can the program logic as follows:

Example 9.8

/*Program to read and print data related to five students having marks of three subjects each using the concept of arrays */

```
#include<stdio.h>
struct student {
    int roll_no;
    char name [20];
    char course [20];
    int subject [3] ;
  };
main( )
{
  struct student stud[5];
  int i,j;
printf (“Enter the data for all the students:\n”);
for (i=0;i<=4;i++)
{
printf (“Enter the roll number of %d student”,i+1);
scanf (“%d”,&stud[i].roll_no);
printf(“Enter the name of %d student”,i+1);
scanf (“%s”,stud[i].name);
printf (“Enter the course of %d student”,i+1);
scanf (“%s”,stud[i].course);
for (j=0;j<=2;j++)
  {
   printf (“Enter the marks of the %d subject of the student %d:\n”,j+1,i+1);
  scanf (“%d”,&stud[i].subject[j]);
  }
}
printf (“The data you have entered is as follows:\n”);
for (i=0;i<=4;i++)
  {
      printf (“The %d th student's roll number is %d\n”,i+1,stud[i].roll_no);
      printf (“The %d the student's name is %s\n”,i+1,stud[i].name);
      printf (“The %d the student's course is %s\n”,i+1,stud[i].course);
      for (j=0;j<=2;j++)
      {
      printf (“The %d the student's marks of %d  I  subject are %d\n”,i+1, j+1,
      stud[i].subject[j]);
```

```
        }
    }
  printf ("End of the program\n");
}
```

Hence as described in the example above, the array as well as the arrays of structures can be used with efficiency to resolve the major hurdles faced in the real world programming environment.

9.7 UNIONS

Structures are a way of grouping homogeneous data together. But it often happens that at any time we require only one of the member's data. For example, in case of the support price of shares you require only the latest quotations. And only the ones that have changed need to be stored. So if we declare a structure for all the scripts, it will only lead to crowding of the memory space. Hence it is beneficial if we allocate space to only one of the members. This is achieved with the concepts of the *UNIONS*. *UNIONS* are similar to *STRUCTURES* in all respects but differ in the concept of storage space.

A *UNION* is declared and used in the same way as the structures. Yet another difference is that only one of its members can be used at any given time. Since all members of a Union occupy the same memory and storage space, the space allocated is equal to the largest data member of the Union. Hence, the member which has been updated last is available at any given time.

For example a union can be declared using the syntax shown below:

```
union union-tag {
                    datatype variable1;
                    datatype variable2;
                        ...
                };
```

For example,

```
union temp{
            int   x;
            char y;
            float z;
           };
```

In this case a float is the member which requires the largest space to store its value hence the space required for float (4 bytes) is allocated to the union. All members share the same space. Let us see how to access the members of the union.

Example 9.9

Write a program to illustrate the concept of union.

```
/* Declare a union template called tag */
union tag {
            int nbr;
            char character;
            }
/* Use the union template */
union tag mixed_variable;
/* Declare a union and instance together */
union generic_type_tag {
                        char c;
                        int i;
```

```
float f;
double d;
} generic;
```

9.8 INITIALIZING AN UNION

Let us see, how to initialize a Union with the help of the following example:

Example 9.10

```
union date_tag {
        char complete_date [9];
        struct part_date_tag {
                char month[2];
                char break_value1;
                char day[2];
                char break_value2;
                char year[2];
                    } parrt_date;
      }date = {"01/01/05"};
```

9.9 ACCESSING THE MEMBERS OF AN UNION

Individual union members can be used in the same way as the structure members, by using the member operator or dot operator (.). However, there is an important difference in accessing the union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Trying to access the previously stored values will result in erroneous output.

Check Your Progress 3

1. What will be the output?

```
#include<stdio.h>
main()
{
union{
      struct{
            char x;
            char y;
            char z;
            char w;
           }xyz;

     struct{
           int p;
           int q ;
           }pq;
           long  a ;
           float b;
           double d;
           }prq;
     printf ("%d",sizeof(prq));
           }
```

9.10 SUMMARY

In this unit, we have learnt how to use structures, a data type that you design to meet the needs of a program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a *member,* is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and can also be used in arrays.

Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all its members in the same area. This means that only one member of a union can be used at a time.

9.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. The first form declares a *structure tag*; the second declares a *typedef*. The main difference is that the second declaration is of a slightly more abstract type - users do not necessarily know that it is a structure, and the keyword struct is not used while declaring an instance.

2. There is no single correct way for a compiler to implement a structure comparison consistent with C's low-level flavor. A simple byte-by-byte comparison could detect the random bits present in the unused "holes" in the structure (such padding is used to keep the alignment of later fields correct). A field-by-field comparison for a large structure might require an inordinate repetitive code.

3. Structures may have this padding (as well as internal padding), to ensure that alignment properties will be preserved when an array of contiguous structures is allocated. Even when the structure is not part of an array, the end padding remains, so that *sizeof* can always return a consistent size.

4.
```
struct date {
        char month[2];
        char day[2];
        char year[4];
    } current_date;
```

Check Your Progress 2

1. When structures are passed as arguments to functions, the entire structure is typically pushed on the stack, using as many words. (Programmers often choose to use pointers instead, to avoid this overhead). Some compilers merely pass a pointer to the structure, though they may have to make a local copy to preserve pass-by value semantics.

   Structures are often returned from functions in a pointed location by an extra, compiler-supplied "hidden" argument to the function. Some older compilers used a special, static location for structure returns, although this made structure - valued functions non-reentrant, which ANSI C disallows.

2. C has no way of generating anonymous structure values. You will have to use a temporary structure variable or a little structure - building function.

3. 10

Check Your Progress 3

1. 8

9.12 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. Computer Science A structured programming approach using C, *Behrouza .Forouzan, Richard F. Gilberg*, Second Edition, Brooks/Cole, Thomson Learning, 2001.
3. Programming with C, Schaum Outlines, Second Edition, *Gottfried*, Tata McGraw Hill, 2003.

UNIT 10 POINTERS

Structure

10.0 INTRODUCTION

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. One of those things, beginners in C find difficult is the concept of pointers. The purpose of this unit is to provide an introduction to pointers and their efficient use in the C programming. Actually, the main difficulty lies with the C's pointer terminology than the actual concept.

C uses pointers in three main ways. First, they are used to create *dynamic data structures*: data structures built up from blocks of memory allocated from the heap at run-time. Second, C uses pointers to handle *variable parameters* passed to functions. And third, pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when you work with strings.

A normal variable is a location in memory that can hold a value. For example, when you declare a variable *i* as an integer, four bytes of memory is set aside for it. In your program, you refer to that location in memory by the name *i*. At the machine level, that location has a memory address, at which the four bytes can hold one integer value. A *pointer* is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is the pointer and the value pointed to. As long as you're careful to ensure that the pointers in your programs always point to valid memory locations, pointers can be useful, powerful, and relatively trouble-free tools.

We will start this unit with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Thorough knowledge of the *pointers* is very much essential for your future courses like the *datastructures, design and analysis of algorithms etc.*.

10.1 OBJECTIVES

After going through this unit you should be able to:

- understand the concept and use pointers;
- address and use of indirection operators;
- make pointer type declaration, assignment and initialization;
- use null pointer assignment;
- use the pointer arithmetic;
- handle pointers to functions;
- see the underlying unit of arrays and pointers; and
- understand the concept of dynamic memory allocation.

10.2 POINTERS AND THEIR CHARACTERISTICS

Computer's memory is made up of a sequential collection of storage cells called bytes. Each byte has a number called an address associated with it. When we declare a variable in our program, the compiler immediately assigns a specific block of memory to hold the value of that variable. Since every cell has a unique address, this block of memory will have a unique starting address. The size of this block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PC's integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. If you want to know the size of the various data types on your system, running the following code given in the Example 10.1 will give you the information.

Example 10.1

Write a program to know the size of the various data types on your system.

```
# include <stdio.h>
main( )
{
  printf ("n Size of a int = %d bytes", sizeof (int));
  printf ("\n Size of a float = %d bytes", sizeof (float));
  printf ("\n Size of a char = %d bytes", sizeof (char));
}
```

OUTPUT

Size of int = 2 bytes
Size of float = 4 bytes
Size of char = 1 byte

An *ordinary variable* is a location in memory that can hold a value. For example, when you declare a variable *num* as an integer, the compiler sets aside 2 bytes of memory (depends up the PC) to hold the value of the integer. In your program, you refer to that location in memory by the name *num*. At the machine level that location has a memory address.

int num = 100;

We can access the value 100 either by the name num or by its memory address. Since addresses are simply digits, they can be stored in any other variable. Such variables that hold addresses of other variables are called *Pointers*. In other words, a *pointer* is

simply a variable that contains an address, which is a location of another variable in memory. A pointer variable "points to" another variable by holding its address. Since a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is a pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get familiar with it, then it is extremely easy and very powerful. One good way to visualize this concept is to examine the figure 10.1 given below:

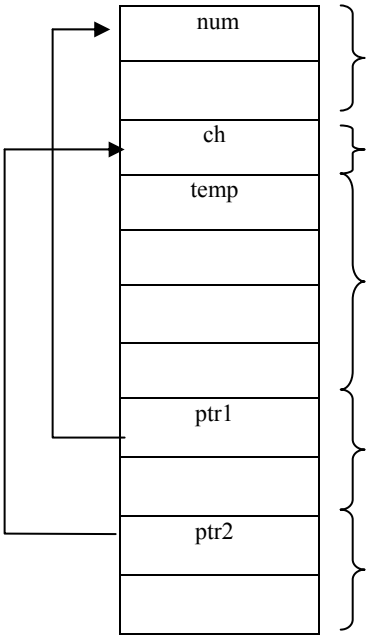


Figure 10.1: Concept of pointer variables

Let us see the important features of the pointers as follows:

Characteristic features of Pointers:

With the use of pointers in programming,

i. The program execution time will be faster as the data is manipulated with the help of addresses directly.
ii. Will save the memory space.
iii. The memory access will be very efficient.
iv. Dynamic memory is allocated.

10.3 THE ADDRESS AND INDIRECTION OPERATORS

Now we will consider how to determine the address of a variable. The operator that is available in C for this purpose is "&" (*address of*) operator. The operator & and the immediately preceding variable returns the address of the variable associated with it. C's other unary pointer operator is the "*", also called as *value at address* or indirection operator. It returns a value stored at that address. Let us look into the illustrative example given below to understand how they are useful.

Example 10.2

Write a program to print the address associated with a variable and value stored at that address.

/* Program to print the address associated with a variable and value stored at that address*/

```
# include <stdio.h>
main( )
{
  int qty = 5;
  printf ("Address of qty = %u\n",&qty);
  printf ("Value of qty = %d \n",qty);
  printf("Value of qty = %d",*(&qty));
 }
```

OUTPUT

```
Address of qty = 65524
Value of qty = 5
Value of qty = 5
```

Look at the *printf* statement carefully. The format specifier *%u* is taken to increase the range of values the address can possibly cover. The system-generated address of the variable is not fixed, as this can be different the next time you execute the same program. Remember unary operator operates on single operands. When *&* is preceded by the variable qty, has returned its address. Note that the *&* operator can be used only with simple variables or array elements. It cannot be applied to expressions, constants, or register variables.

Observe the third line of the above program. **(&qty)* returns the value stored at address 65524 i.e. 5 in this case. Therefore, *qty* and **(&qty)* will both evaluate to 5.

10.4 POINTER TYPE DECLARATION AND ASSIGNMENT

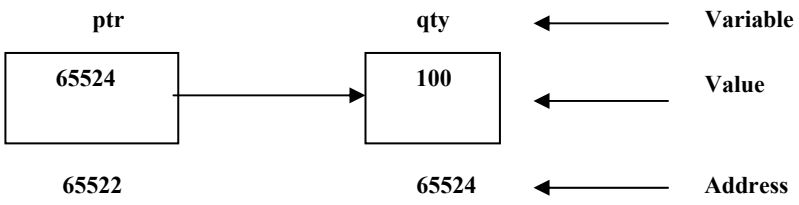
We have seen in the previous section that *&qty* returns the address of *qty* and this address can be stored in a variable as shown below:

ptr = &qty;

In C, every variable must be declared for its data type before it is used. Even this holds good for the pointers too. We know that *ptr* is not an ordinary variable like any integer variable. We declare the data type of the pointer variable as that of the type of the data that will be stored at the address to which it is pointing to. Since *ptr* is a variable, which contains the address of an integer variable *qty*, it can be declared as:

*int *ptr;*

where *ptr* is called a *pointer variable*. In C, we define a pointer variable by preceding its name with an asterisk(*). The "*" informs the compiler that we want a pointer variable, i.e. to set aside the bytes that are required to store the address in memory. The *int* says that we intend to use our pointer variable to store the address of an integer. Consider the following memory map:



Let us look into an example given below:

Example 10.3

/* Program below demonstrates the relationships we have discussed so far */

```
# include <stdio.h>
main( )
{
  int qty = 5;
  int *ptr;         /* declares ptr as a pointer variable that points to an integer variable
*/
  ptr = &qty;   /* assigning qty's address to ptr -> Pointer Assignment */

  printf ("Address of qty = %u \n", &qty);
  printf ("Address of qty = %u \n", ptr);
  printf ("Address of ptr = %u \n", &ptr);
  printf ("Value of ptr = %d \n", ptr);
  printf ("Value of qty = %d \n", qty);
  printf ("Value of qty = %d \n", *(&qty));
  printf ("Value of qty = %d", *ptr);
}
```

OUTPUT

```
Address of qty = 65524
Address of ptr = 65522
Value of ptr = 65524
Value of qty = 5
Value of qty = 5
Value of qty = 5
```

Try this as well:

Example 10.4

/* Program that tries to reference the value of a pointer even though the pointer is uninitialized */

```
# include <stdio.h>
main()
{
   int *p;   /* a pointer to an integer */
    *p = 10;
   printf("the value is %d", *p);
   printf("the value is %u",p);
}
```

This gives you an error. The pointer *p* is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system. When you say **p=10;* the program will simply try to write a 10 to whatever random location ***p*** points to. The program may explode immediately. It may subtly corrupt data in another part of your program and you may never realize it. Almost always, an uninitialized pointer or a bad pointer address causes the fault.

This can make it difficult to track down the error. Make sure you initialize all pointers to a valid address before dereferencing them.

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable. Remember the variable whose address is assigned to the pointer variable must be declared earlier in the program. In the example given below, let us assign the pointer *p* with an address and also a value 10 through the **p*.

Example 10.5

Let us say,

```
int x; /* x is initialized to a value 10*/
p = &x;   /* Pointer declaration & Assignment */
*p=10;
```

Let us write the complete program as shown below:

```
# include <stdio.h>
main( )
{
   int *p;   /* a pointer to an integer */
   int x;
   p = &x;
   *p=10;
   printf("The value of x is %d",*p);
   printf("\nThe address in which the x is stored is %d",p);
}
```

OUTPUT

```
The value of x is 10
The address in which the x is stored is 52004
```

This statement puts the value of 20 at the memory location whose address is the value of *px*. As we know that the value of *px* is the address of *x* and so the old value of *x* is replaced by 20. This is equivalent to assigning 20 to *x*. Thus we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

10.4.1 Pointer to a Pointer

The concept of pointer can be extended further. As we have seen earlier, a pointer variable can be assigned the address of an ordinary variable. Now, this variable itself could be another pointer. This means that a pointer can contain address of another pointer. The following program will makes you the concept clear.

Example 10.6

```
/* Program that declares a pointer to a pointer */

# include<stdio.h>
main( )
{
  int i = 100;
  int *pi;
  int **pii;
  pi = &i;
  pii = &pi;

 printf ("Address of i = %u \n", &i);
```

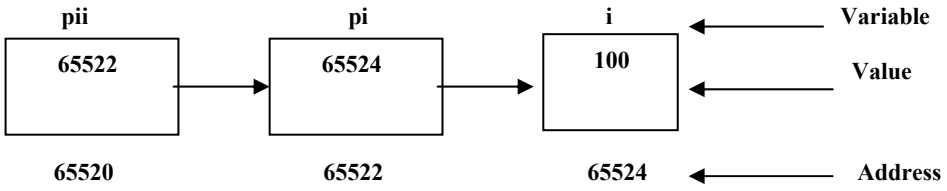
```
 printf ("Address of i = %u \n", pi);
 printf ("Address of i = %u \n", *pii);
 printf ("Address of pi = %u \n", &pi);
 printf ("Address of pi = %u \n", pii);
 printf ("Address of pii = %u \n", &pii);
 printf ("Value of i = %d \n", i);
 printf ("Value of i = %d \n", *(&i));
 printf ("Value of i = %d \n", *pi);
 printf ("Value of i = %d", **pii);
}
```

OUTPUT

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of pi = 65522
Address of pi = 65522
Address of pii = 65520

Value of i = 100
Value of i = 100
Value of i = 100
Value of i = 100
```

Consider the following memory map for the above shown example:



10.4.2 Null Pointer Assignment

It does make sense to assign an integer value to a pointer variable. An exception is an assignment of 0, which is sometimes used to indicate some special condition. A macro is used to represent a null pointer. That macro goes under the name *NULL*. Thus, setting the value of a pointer using the *NULL*, as with an assignment statement such as *ptr = NULL*, tells that the pointer has become a *null* pointer. Similarly, as one can test the condition for an integer value as zero or not, like *if (i == 0)*, as well we can test the condition for a null pointer using *if (ptr == NULL)* or you can even set a pointer to *NULL* to indicate that it's no longer in use. Let us see an example given below.

Example 10.7

```
# include<stdio.h>
# define NULL 0
main()
{
  int *pi = NULL;
  printf("The value of pi is %u", pi);
}
```

OUTPUT

The value of pi is 0

Check Your Progress 1

1. How is a pointer variable being declared? What is the purpose of data type included in the pointer declaration?

……

……

……

2. What would be the output of following programs?

(i)
```
void main( )
{
   int i = 5;
   printf ("Value of i = %d   Address of i = %u", i, &i);
   &i = 65534;
   printf ("\n New value of i = %d  New Address of i = %u", i, &i);
   }
```

(ii)
```
void main( )
{
   int *i, *j;
   j = i * 2;
   printf ("j = %u", j);
}
```

……

……

……

3. Explain the effect of the following statements:

(i) `int x = 10, *px = &x;`

(ii) `char *pc;`

(iii)
```
int x;
void *ptr = &x;
*(int *) ptr = 10;
```

……

……

……

10.5 POINTER ARITHMETIC

Pointer variables can also be used in arithmetic expressions. The following operations can be carried out on pointers:

1. Pointers can be incremented or decremented to point to different locations like

```
ptr1 = ptr2 + 3;
ptr ++;
-- ptr;
```

However, *ptr++* will cause the pointer *ptr* to point the next address value of its type. For example, if *ptr* is a pointer to float with an initial value of 65526, then after the operation *ptr* **++** or *ptr = ptr+1*, the value of *ptr* would be 65530. Therefore, if we increment or decrement a pointer, its value is increased or decreased by the length of the data type that it points to.

2. If *ptr1* and *ptr2* are properly declared and initialized pointers, the following operations are valid:

```
res = res + *ptr1;
*ptr1 = *ptr2 + 5;
prod = *ptr1 * *ptr2;
quo = *ptr1 /  *ptr2;
```

Note that there is a blank space between / and * in the last statement because if you write **/*** together, then it will be considered as the beginning of a comment and the statement will fail.

3. Expressions like *ptr1 == ptr2*, *ptr1 < ptr2*, and *ptr2 != ptr1* are permissible provided the pointers *ptr1* and *ptr2* refer to same and related variables. These comparisons are common in handling arrays.

Suppose *p1* and *p2* are pointers to related variables. The following operations cannot work with respect to pointers:

1. Pointer variables cannot be added. For example, *p1 = p1 + p2* is not valid.

2. Multiplication or division of a pointer with a constant is not allowed. For example, *p1 * p2* or *p2 / 5* are invalid.

3. An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block. Suppose *p* and *q* are two pointers. If we say, *p = q;* when *q* is uninitialized. The pointer *p* will then become uninitialized as well, and any reference to **p* is an invalid pointer reference.

10.6 PASSING POINTERS TO FUNCTIONS

As we have studied in the FUNCITONS that arguments can generally be passed to functions in one of the two following ways:

1. Pass by value method
2. Pass by reference method

In the first method, when arguments are passed by value, a copy of the *values* of actual arguments is passed to the calling function. Thus, any changes made to the variables inside the function will have no effect on variables used in the actual argument list.

However, when arguments are passed by reference (i.e. when a pointer is passed as an argument to a function), the *address* of a variable is passed. The contents of that address can be accessed freely, either in the called or calling function. Therefore, the function called by reference can change the value of the variable used in the call.

Here is a simple program that illustrates the difference.

Example 10.8

Write a program to swap the values using the pass by value and pass by reference methods.

/* Program that illustrates the difference between ordinary arguments, which are passed by value, and pointer arguments, which are passed by reference */

```
# include <stdio.h>
main()
{
        int x = 10;
        int y = 20;
       void swapVal ( int, int );          /* function prototype */
       void swapRef ( int *, int * );      /*function prototype*/
        printf("PASS BY VALUE METHOD\n");
        printf ("Before calling function swapVal    x=%d   y=%d",x,y);
        swapVal (x, y);             /* copy of the arguments are passed */
        printf ("\nAfter calling function swapVal   x=%d   y=%d",x,y);
               printf("\n\nPASS BY REFERENCE METHOD");
        printf ("\nBefore calling function swapRef   x=%d   y=%d",x,y);
        swapRef (&x,&y);      /*address of arguments are passed */
        printf("\nAfter calling function swapRef     x=%d   y=%d",x,y);
}

/* Function using the pass by value method*/
void swapVal (int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
        printf ("\nWithin function swapVal          x=%d   y=%d",x,y);
        return;
}

/*Function using the pass by reference method*/
void swapRef (int *px, int *py)
{
         int temp;
         temp = *px;
         *px = *py;
         *py = temp;
          printf ("\nWithin function swapRef       *px=%d   *py=%d",*px,*py);
          return;
}
```

OUTPUT

```
PASS BY VALUE METHOD
Before calling function swapVal   x=10        y=20
Within function swapVal           x=20        y=10
After calling function swapVal    x=10        y=20
```

```
PASS BY REFERENCE METHOD
Before calling function swapRef    x=10        y=20
Within function swapRef            *px=20    *py=10
After calling function swapRef      x=20        y=10
```

This program contains two functions, *swapVal* and *swapRef*.

In the function *swapVal*, arguments *x* and *y* are passed by *value*. So, any changes to the arguments are local to the function in which the changes occur. Note the values of *x* and *y* remain unchanged even after exchanging the values of *x* and *y* inside the function ***swapVal***.

Now consider the function *swapRef*. This function receives two *pointers* to integer variables as arguments identified as pointers by the indirection operators that appear in argument declaration. This means that in the function *swapRef*, arguments *x* and *y* are passed by *reference.* So, any changes made to the arguments inside the function *swapRef* are reflected in the function *main( ).* Note the values of *x* and *y* is interchanged after the function call *swapRef*.

10.6.1 A Function returning more than one value

Using *call by reference* method we can make a function return more than one value at a time, which is not possible in the *call by value* method. The following program will makes you the concept very clear.

Example 10.9

Write a program to find the perimeter and area of a rectangle, if length and breadth are given by the user.

```
/* Program to find the perimeter and area of a rectangle*/

#include <stdio.h>
void main()
{
float len,br;
float peri, ar;
void periarea(float length, float breadth, float *, float *);
printf("\nEnter the length and breadth of  a rectangle in metres: \n");
scanf("%f %f",&len,&br);
periarea(len,br,&peri,&ar);
printf("\nPerimeter of the rectangle is %f metres", peri);
printf("\nArea of the rectangle is %f sq. metres", ar);
}

void periarea(float length, float breadth, float *perimeter, float *area)
{
*perimeter = 2 * (length +breadth);
*area = length * breadth;
}
```

OUTPUT

```
Enter the length and breadth of  a rectangle in metres:
23.0 3.0
Perimeter of the rectangle is 52.000000 metres
Area of the rectangle is 69.000000 sq. metres
```

Here in the above program, we have seen that the function *periarea* is returning two values. We are passing the values of *len* and *br* but, addresses of peri and *ar*. As we are passing the addresses of *peri* and *ar*, any change that we make in values stored at addresses contained in the variables **perimeter* and **area*, would make the change effective even in *main()* also.

10.6.2 Function returning a pointer

A function can also return a pointer to the calling program, the way it returns an int, a float or any other data type. To return a pointer, a function must explicitly mention in the calling program as well as in the function prototype. Let's illustrate this with an example:

Example: 10.10

Write a program to illustrate a function returning a pointer.

```
/*Program that shows how a function returns a pointer */

# include<stdio.h>

void main( )
{
  float *a;
  float *func( );        /* function prototype */
  a = func( );
  printf ("Address = %u", a);
}
float *func( )
{
   float r = 5.2;
   return (&r);
}
```

OUTPUT

Address = 65516

This program only shows how a function can return a pointer. This concept will be used later while handling arrays.

Check Your Progress 2

1. Tick mark (√)whether each of the following statements are true or false.

(i) An integer is subtracted from a pointer variable. ☐ True ☐ False

(ii) Pointer variables can be compared. ☐ True ☐ False

(iii) Pointer arguments are passed by value. ☐ True ☐ False

(iv) Value of a local variable in a function can be changed by another function. ☐ True ☐ False

(v) A function can return more than one value. ☐ True ☐ False

(vi) A function can return a pointer. ☐ True ☐ False

10.7 ARRAYS AND POINTERS

Pointers and arrays are so closely related. An array declaration such as *int arr[5]* will lead the compiler to pick an address to store a sequence of 5 integers, and *arr* is a name for that address. The array name in this case is the *address* where the sequence of integers starts. Note that the value is not the first integer in the sequence, nor is it the sequence in its entirety. The value is just an address.

Now, if *arr* is a one-dimensional array, then the address of the first array element can be written as *&arr[0]* or simply *arr*. Moreover, the address of the second array element can be written as *&arr[1]* or simply *(arr+1)*. In general, address of array element *(i+1)* can be expressed as either *&arr[i]* or as *(arr+ i)*. Thus, we have two different ways for writing the address of an array element. In the latter case i.e, expression *(arr+ i)* is a symbolic representation for an address rather than an arithmetic expression. Since *&arr[i]* and *(ar+ i)* both represent the address of the i^{th} element of *arr,* so *arr[i]* and **(ar + i)* both represent the contents of that address i.e., the value of i^{th} element of *arr*.

Note that it is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as *arr*, *(arr+ i)* and *arr[i]* cannot appear on the left side of an assignment statement. Thus we cannot write a statement such as:

```
&arr[0] = &arr[1];      /*  Invalid  */
```

However, we can assign the value of one array element to another through a pointer, for example,

```
ptr = &arr[0];       /*   ptr is a pointer to arr[ 0]  */
arr[1] = *ptr;       /* Assigning the value stored at address to arr[1]  */
```

Here is a simple program that will illustrate the above-explained concepts:

Example 10.11

```
/* Program that accesses array elements of a one-dimensional array using pointers */

# include<stdio.h>
main()
{

  int arr[ 5 ] = {10, 20, 30, 40, 50};
  int i;

  for (i = 0;  i < 5;  i++)
     {
         printf ("i=%d\t arr[i]=%d\t *(arr+i)=%d\t", i, arr[i], *(arr+i));
         printf ("&arr[i]=%u\t arr+i=%u\n", &arr[i], (arr+i));     }
 }
```

OUTPUT:

```
i=0     arr[i]=10      *(arr+i)=10    &arr[i]=65516    arr+i=65516
i=1     arr[i]=20      *(arr+i)=20    &arr[i]=65518    arr+i=65518
i=2     arr[i]=30      *(arr+i)=30    &arr[i]=65520    arr+i=65520
i=3     arr[i]=40      *(arr+i)=40    &arr[i]=65522    arr+i=65522
i=4     arr[i]=50      *(arr+i)=50    &arr[i]=65524    arr+i=65524
```

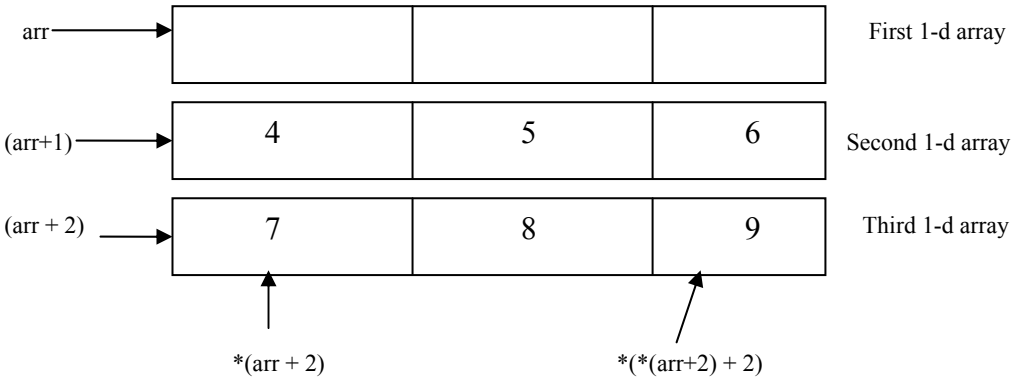
Note that *i* is added to a pointer value (address) pointing to integer data type (i.e., the array name) the result is the pointer is increased by i times the size (in bytes) of integer data type. Observe the addresses 65516, 65518 and so on. So if *ptr* is a char pointer, containing addresses *a*, then *ptr+1* is *a+1*. If *ptr* is a float pointer, then *ptr+ 1* is *a+ 4*.

Pointers and Multidimensional Arrays

C allows multidimensional arrays, lays them out in memory as contiguous locations, and does more behind the scenes address arithmetic. Consider a 2-dimensional array.

int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

The compiler treats a 2 dimensional array as an array of arrays. As you know, an array name is a pointer to the first element within the array. So, **arr** points to the first 3-element array, which is actually the first row (i.e., row 0) of the two-dimensional array. Similarly, *(arr + 1)* points to the second 3-element array (i.e., row 1) and so on. The value of this pointer, *(arr + 1), refers to the entire row. Since row 1 is a one-dimensional array, *(arr + 1)* is actually a pointer to the first element in row 1. Now add 2 to this pointer. Hence, *(*(arr + 1) + 2)* is a pointer to element 2 (i.e., the third element) in row 1. The value of this pointer, **(*(arr + 1) + 2),* refers to the element in column 2 of row 1. These relationships are illustrated below:



10.8 ARRAY OF POINTERS

The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too. Since a pointer contains an address, an array of pointers would be a collection of addresses. For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.

Two-dimensional array can be defined as a one-dimensional array of integer pointers by writing:

*int *arr[3];*

rather than the conventional array definition,

int arr[3][5];

Similarly, an n-dimensional array can be defined as (n-1)-dimensional array of pointers by writing

*data-type *arr[subscript 1] [subscript 2]…. [subscript n-1];*

The subscript1, subscript2 indicate the maximum number of elements associated with each subscript.

Example 10.12

Write a program in which a two-dimensional array is represented as an array of integer pointers to a set of single-dimensional integer arrays.

```
/* Program calculates the difference of the corresponding elements of two table of integers */

# include <stdio.h>
# include <stdlib.h>
# define MAXROWS 3
void main( )
{
     int *ptr1[MAXROWS], *ptr2 [MAXROWS], *ptr3 [MAXROWS];
     int rows, cols, i, j;
     void inputmat (int *[ ], int, int);
     void dispmat (int *[ ], int, int);
     void calcdiff (int *[ ], int *[ ], int *[ ], int, int);

     printf ("Enter no. of rows & columns \n");
     scanf ("%d%d", &rows, &cols);

     for (i = 0; i < rows; i++)
     {
          ptr1[ i ] = (int *) malloc (cols * sizeof (int));
          ptr2[ i ] = (int *) malloc (cols * sizeof (int));
          ptr3[ i ] = (int *) malloc (cols * sizeof (int));
    }

    printf ("Enter values in first matrix \n");
    inputmat (ptr1, rows, cols);
    printf ("Enter values in second matrix \n");
    inputmat (ptr2, rows, cols);
    calcdiff (ptr1, ptr2, ptr3, rows, cols);
    printf ("Display difference of the two matrices \n");
    dispmat (ptr3, rows, cols);
}

 void inputmat (int *ptr1[MAXROWS], int m, int n)
 {
     int i, j;
     for (i = 0; i < m; i++)
     {
          for (j = 0; j < n; j++)
          {
               scanf ("%d", (*(ptr1 + i) + j));
          }
      }
      return;
 }

 void dispmat (int *ptr3[  MAXROWS ], int m, int n)
 {
     int i, j;
```

```
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf ("%d ", *(*(ptr3 + i) + j));
        }
        printf("\n");
    }
    return;
}

void calcdiff (int *ptr1[ MAXROWS ], int *ptr2 [ MAXROWS ],
                        int *ptr3[MAXROWS], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            *(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);
        }
    }
    return;
}
```

OUTPUT

```
Enter no. of rows & columns
3  3
Enter values in first matrix
2    6   3
5    9   3
1    0   2
Enter values in second matrix
3    5   7
2    8   2
1    0   1
Display difference of the two matrices
-1   1  -4
 3   1   1
 0   0   1
```

In this program, *ptr1*, *ptr2*, *ptr3* are each defined as an array of pointers to integers. Each array has a maximum of MAXROWS elements. Since each element of ptr1, ptr2, ptr3 is a pointer, we must provide each pointer with enough memory for each row of integers. This can be done using the library function *malloc* included in *stdlib.h* header file as follows:

*ptr1[i] = (int *) malloc (cols * sizeof (int));*

This function reserves a block of memory whose size (in bytes) is equivalent to *cols * sizeof(int).* Since cols = 3, so 3 * 2 (size of int data type) i.e., 6 is allocated to each *ptr1[1], ptr1[2]* and *ptr1[3].* This *malloc* function returns a pointer of type *void*. This means that we can assign it to any type of pointer. In this case, the pointer is type-casted to an integer type and assigned to the pointer *ptr1[1], ptr1[2]* and *ptr1[3].* Now, each of *ptr1[1], ptr1[2]* and *ptr1[3]* points to the first byte of the memory allocated to the corresponding set of one-dimensional integer arrays of the original two-dimensional array.

The process of calculating and allocating memory at run time is known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

Instead of using conventional array notation, pointer notation has been used for accessing the address and value of corresponding array elements which has been explained to you in the previous section. The difference of the array elements within the function *calcdiff* is written as

**(*(ptr3 + i) + j) = *(*(ptr1 + i) + j) - *(*(ptr2 + i) + j);*

10.9 POINTERS AND STRINGS

As we have seen in strings, a string in C is an array of characters ending in the null character (written as '\0'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the (constant) address of its first character. Thus, it is appropriate to say that a string is a constant pointer.
A string can be declared as a character array or a variable of type *char **. The declarations can be done as shown below:

```
char country[ ] = "INDIA";
char *country = "INDIA";
```

Each initialize a variable to the string “INDIA”. The second declaration creates a pointer variable *country* that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable *country*, **country* would yield the value at this address, which gets printed through,

```
printf ("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using *the* library function *malloc* at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.

Example 10.13

Write a program to test whether the given string is a palindrome or not.

/* Program tests a string for a palindrome using pointer notation */

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

main()
{
    char *palin, c;
    int i, count;

    short int palindrome(char,int);          /*Function Prototype */
    palin = (char *) malloc (20 * sizeof(char));
    printf("\nEnter a word: ");
    do
```

```
    {
        c = getchar( );
        palin[i] = c;
        i++;
    }while (c != '\n');

    i = i-1;
    palin[i] = '\0';
    count = i;

    if (palindrome(palin,count) == 1)
        printf ("\nEntered word is not a palindrome.");
    else
        printf ("\nEntered word is a palindrome");
    }

short int palindrome(char *palin, int len)
{
    short int i = 0, j = 0;
    for(i=0 , j=len-1; i < len/2;i++,j--)
    {
        if (palin[i] == palin[j])
            continue;
        else
            return(1);
    }
    return(0);
}
```

OUTPUT

Enter a word: malayalam

Entered word is a palindrome.

Enter a word: abcdba

Entered word is not a palindrome.

Array of pointers to strings

Arrays may contain pointers. We can form an array of strings, referred to as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character, so each entry in an array of strings is actually a pointer to the first character of a string. Consider the following declaration of a string array:

```
char *country[ ] = {
                    "INDIA", "CHINA", "BANGLADESH", "PAKISTAN", "U.S"
                      };
```

The **country[ ]* of the declaration indicates an array of five elements. The *char** of the declaration indicates that each element of array country is of type "pointer to char". Thus, *country [0]* will point to INDIA, *country[1]* will point to CHINA, and so on.

Thus, even though the array *country* is fixed in size, it provides access to character strings of any length. However, a specified amount of memory will have to be allocated for each string later in the program, for example,

```
country[ i ] = (char *) malloc(15 * sizeof (char));
```

The *country* character strings could have been placed into a two-dimensional array but such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when a large number of strings are stored with most strings shorter than the longest string.

As individual strings can be accessed by referring to the corresponding array element, individual string elements be accessed through the use of the indirection operator. For example, ** (* country + 3) + 2)* refers to the third character in the fourth string of the array *country*. Let us see an example below.

Example 10.14

Write a program to enter a list of strings and rearrange them in alphabetical order, using a one-dimensional array of pointers, where each pointer indicates the beginning of a string:

```
/* Program to sort a list of strings in alphabetical order using an array of pointers */

# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# include <string.h>

void readinput (char *[ ], int);
void writeoutput (char *[ ], int);
void reorder (char *[ ], int);

main( )
{
      char *country[ 5 ];
      int i;
      for (i = 0; i < 5; i++)
          {
           country[ i ] = (char *) malloc (15 * sizeof (char));
          }
     printf ("Enter five countries on a separate line\n");
     readinput (country, 5);
     reorder (country, 5);
     printf ("\nReordered list\n");
     writeoutput (country, 5);
     getch( );
}

void readinput (char *country[ ], int n)
{
       int i;
       for (i = 0; i < n; i++)
                {   scanf ("%s", country[ i ]);   }
        return;
}

void writeoutput (char *country[ ], int n)
{
       int i;
```

```
        for (i = 0; i < n; i++)
        {   printf ("%s", country[ i ]);
            printf ("\n");   }
         return;
}

void reorder (char *country[ ], int n)
{
          int i, j;
          char *temp;
          for (i = 0; i < n-1; i++)
          {
              for (j = i+1; j < n; j++)
              {
                  if (strcmp (country[ i ], country[ j ]) > 0)
                  {
                       temp = country[ i ];
                       country[ i ] = country[ j ];
                       country[ j ] = temp;
                   }
               }
           }
           return;
 }
```

OUTPUT

Enter five countries on a seperate line
INDIA
BANGLADESH
PAKISTAN
CHINA
SRILANKA

Reordered list
BANGLADESH
CHINA
INDIA
PAKISTAN
SRILANKA

The limitation of the string array concept is that when we are using an array of pointers to strings we can initialize the strings at the place where we are declaring the array, but we cannot receive the strings from keyboard using *scanf( )*.

Check Your Progress 3

1. What is meant by array of pointers?

……

……

2. How the indirection operator can be used to access a multidimensional array element.

……

……

3. A C program contains the following declaration.
float temp[3][2] = {{13.4, 45.5}, {16.6, 47.8}, {20.2, 40.8}};

(i) What is the meaning of temp?
(ii) What is the meaning of (temp + 2)?
(iii) What is the meaning of *(temp + 1)?
(iv) What is the meaning of (*(temp + 2) + 1)?
(v) What is the meaning of *(*(temp) + 1) + 1)?
(vi) What is the meaning of *(*(temp + 2))?

……

……

……

10.10 SUMMARY

In this unit we have studied about pointers, pointer arithmetic, passing pointers to functions, relation to arrays and the concept of dynamic memory allocation. A pointer is simply a variable that contains an address which is a location of another variable in memory. The unary operator *&,* when preceded by any variable returns its address. C's other unary pointer operator is ***,** when preceded by a pointer variable returns a value stored at that address.

Pointers are often passed to a function as arguments by reference. This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form. There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array. Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.

As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as *dynamic memory allocation.* The library routine *malloc* can be used for this purpose.

10.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1. Refer to section 10.4. The data type included in the pointer declaration, refers to the type of data stored at the address which we will be storing in our pointer.

2. (i) Compile-time Error : Lvalue Required. Means that the left side of an assignment operator must be an addressable expression that include a variable or an indirection through a pointer.

(ii) Multiplication of a pointer variable with a constant is invalid.

3. (i) Refer section 10.4
(ii) Refer section 10.4
(iii) This means pointers can be of type void but can't be de-referenced without explicit casting. This is because the compiler can't determine the size of the object the pointer points to.

Check Your Progress 2

1 (i) True.
(ii) True.
(iii) False.
(iv) True.
(v) True.
(vi) True.

Check Your Progress 3

1. Refer section 10.4.

2. Refer section 10.4 to comprehend the convention followed.

3. (i) Refers to the base address of the array temp.

(ii) Address of the first element of the last row of array temp i.e. address of element 20.2.

(iii) Will give you 0. To get the value of the last element of the first array i.e. the correct syntax would be *(*(temp+0)+1).

(iv) Address of the last element of last row of array temp i.e. of 40.8.

(v) Displays the value 47.8 i.e., second element of last row of array temp.

(vi) Displays the value 20.2 i.e., first element of last row of array temp.

10.12 FURTHER READINGS

1. Programming with C, Second Edition, *Gottfried Byron S*, Tata McGraw Hill, India.
2. The C Programming Language, Second Edition, *Brian Kernighan and Dennis Richie*, PHI, 2002.
3. Programming in ANSI C, Second Edition, *Balaguruswamy E*, Tata McGraw Hill, India, 2002.
4. How to Solve it by Computer, *R.G.Dromey*, PHI, 2002.
5. C Programming in 12 easy lessons, *Greg Perry*, SAMS, 2002.
6. Teach Yourself C in 21 days, Fifth Edition, *Peter G*, Fifth edition,SAMS, 2002.

UNIT 11 THE C PREPROCESSOR

Structure

11.0 INTRODUCTION

Theoretically, the "preprocessor" is a translation phase that is applied to the source code before the compiler gets its hands on it. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. C Preprocessor is just a text substitution tool, which filters your source code before it is compiled. The preprocessor more or less provides its own language, which can be a very powerful tool for the programmer. All preprocessor directives or commands begin with the symbol #.

The preprocessor makes programs easier to develop, read and modify. The preprocessor makes C code portable between different machine architectures & customizes the language.

The preprocessor performs textual substitutions on your source code in three ways:

File inclusion: Inserting the contents of another file into your source file, as if you had typed it all in there.
Macro substitution: Replacing instances of one piece of text with another.
Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The next three sections will introduce these three preprocessing functions. The syntax of the preprocessor is different from the syntax of the rest of C program in several respects. The C preprocessor is not restricted to use with C programs, and programmers who use other languages may also find it useful. However, it is tuned to recognize features of the C language like comments and strings.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- define, declare preprocessor directives;
- discuss various preprocessing directives, for example file inclusion, macro substitution, and conditional compilation; and
- discuss various syntaxes of preprocessor directives and their applications.

11.2 # *define* TO IMPLEMENT CONSTANTS

The preprocessor allows us to customize the language. For example to replace **{** and **}** of C language to *begin* and *end* as block-statement delimiters (as like the case in PASCAL) we can achieve this by writing:

```
# define begin {
# define end   }
```

During compilation all occurrences of *begin* and *end* get replaced by corresponding **{** and **}**. So the subsequent C compilation stage does not know any difference!

#define is used to define constants.

The syntax is as follows:

 # define <literal> <replacement-value>

literal is identifier which is replaced with *replacement-value* in the program.

For Example,

```
#define MAXSIZE 256
#define PI 3.142857
```

The C preprocessor simply searches through the C code before it is compiled and replaces every instance of *MAXSIZE* with 256.

```
# define  FALSE  0
# define  TRUE   !FALSE
```

The literal *TRUE* is substituted by *!FALSE* and *FALSE* is substituted by the value 0 at every occurrence, before compilation of the program. Since the values of the literal are constant throughout the program, they are called as constant.

The syntax of above *# define* can be rewritten as:

 # define <constant-name> <replacement-value>

Let us consider some examples,

```
# define        M             5
# define        SUBJECTS      6
# define        PI            3.142857
# define        COUNTRY       INDIA
```

Note that no semicolon (;) need to be placed as the delimiter at the end of a # define line. This is just one of the ways that the syntax of the preprocessor is different from the rest of C statements (commands). If you unintentionally place the semicolon at the end as below:

```
#define MAXLINE 100;          /* WRONG */
```

and if you declare as shown below in the declaration section,

```
char line[MAXLINE];
```

the preprocessor will expand it to:

```
char line[100;];                    /* WRONG */
```

which gives you the syntax error. This shows that the preprocessor doesn't know much of anything about the syntax of C.

11.3 # *define* TO CREATE FUNCTIONAL MACROS

Macros are inline code, which are substituted at compile time. The definition of a macro is that which accepts an argument when referenced. Let us consider an example as shown below:

Example 11.1

Write a program to find the square of a given number using macro.

```
/* Program to find the square of a number using marco*/
#include <stdio.h>
# define  SQUARE(x)   (x*x)
main()
    {
     int v,y;
     printf("Enter any number to find its square: ");
     scanf("%d", &v);
     y = SQUARE(v);
     printf("\nThe square of %d is %d", v, y);
}
```

OUTPUT

```
Enter any number to find its square: 10
The square of 10 is 100
```

In this case, *v* is equated with *x* in the macro definition of *square*, so the variable *y* is assigned the square of *v*. The brackets in the macro definition of *square* are necessary for correct evaluation. The expansion of the macro becomes:

*y =(v * v);*

Macros can make long, ungainly pieces of code into short words. Macros can also accept parameters and return values. Macros that do so are called *macro functions*. To create a macro, simply define a macro with a parameter that has whatever name you like, such as *my_val*. For example, one macro defined in the standard libraries is "abs", which returns the absolute value of its parameter. Let us define our own version of ***ABS*** as shown below. Note that we are defining it in upper case not only to avoid conflicting with the existing "abs".

```
#define ABS(my_val) ((my_val) < 0) ? -(my_val) : (my_val)
```

#define can also be given arguments which are used in its replacement. The definitions are then called macros. Macros work rather like functions, but with the following minor differences:

- Since macros are implemented as a textual substitution, by this the performance of program improves compared to functions.
- Recursive macros are generally not a good idea.

- Macros don't care about the type of their arguments. Hence macros are a good choice where we want to operate on reals, integers or a mixture of the two. Programmers sometimes call such type flexibility polymorphism.
- Macros are generally fairly small.

Let us look more illustrative examples to understand the *macros* concept.

Example 11.2

Write a program to declare constants and macro functions using *#define*.

```
/* Program to illustrate the macros */
#include <stdio.h>
#include <string.h>
#define STR1        "A macro definition!\n"
#define STR2        "must be all on one line!\n"
#define EXPR1        1+2+3
#define EXPR2        EXPR1+5
#define ABS(x)       (((x) < 0) ? – (x):(x))
#define MAX(p,q)    ((p < q) ? (q):(p))
#define BIGGEST(p,q,r)   (MAX(p, q) < r)?(r):(MAX(p, q))
main()
{
  printf(STR1);
  printf(STR2);
  printf("Largest number among  %d, %d and %d is %d\n",EXPR1, EXPR2, ABS (–3),
                                                        BIGGEST(1,2,3));
}
```

OUTPUT

```
A macro definition
must be all on one line!
Largest number among 6, 11 and 3 is 3
```

The macro STR1 is replaced with "A macro definition \n" in the first *printf()* function. The macro STR2 is replaced with "must be all on one line! \n" in the second *printf* function. The macro EXPR1 is replaced with 1+2+3 in third *printf* statement. The macro EXPR2 is replaced with EXPR1 +5 in fourth *printf* statement. The macro ABS(–3) is replaced with (– 3<0) ? – (– 3) : 3. And evaluation 3 is replaced. The largest among the three numbers is diplayed.

Example 11.3

Write a program to find out square and cube of any given number using macros.

```
/* Program to find  the square and cube of any given number using macro directive */
# include<stdio.h>
# define   sqr(x)          (x  * x)
# define   cub(x)          (sqr(x) *  x)
main()
{
        int  num;
        printf("Enter a number: ");
        scanf("%d", &num);
        printf(" \n Square of  the number is %d", sqr(num));
        printf(" \n Cube of the number is %d\n", cub(num));
}
```

OUTPUT

```
Enter a number: 5
Square of the number is 25
Cube of the number is 125
```

Note: Multi-line macros can be defined by placing a backward slash (\ ) at end of each line except the last line. This feature permits a single macro (i.e. a single identifier) to represent a compound statement.

Example 11.4

Write a macro to display the string COBOL in the following fashion

```
C
CO
COB
COBO
COBOL
COBOL
COBO
COB
CO
C
```

```
/* Program to display the string as given in the problem*/
# include<stdio.h>
# define        LOOP    for(x=0; x<5; x++)              \
                        {       y=x+1;                  \
                                printf("%-5.*s\n", y, string);  }   \
                        for(x=4; x>=0; x--)             \
                        {           y=x+1;              \
                        printf("%-5.*s \n", y, string); }

main()
{
        int x, y;
        static char string[ ] = "COBOL";
        printf("\n");
        LOOP;
}
```

When the above program is executed the reference to macro (loop) is replaced by the set of statements contained within the macro definition.

OUTPUT

```
C
CO
COB
COBO
COBOL
COBOL
COBO
COB
CO
C
```

Recollect that CALL BY VALUE Vs CALL BY REFERENCE given in the previous uint. By CALL BY VALUE, the swapping was not taking place, because the visibility of the variables was restricted to with in the function in the case of local variables. You can resolve this by using a macro. Here is ***swap*** in action when using a macro:

#define swap(x, y) {int tmp = x; x = y; y = tmp; }

Now we have swapping code that works. Why does this work? It is because the CPP just simply replaces text. Wherever swap is called, the CPP will replace the macro call with the macro meaning, (defined text).

Caution in using macros

You should be very careful in using Macros. In particular the textual substitution means that arithmetic expressions are liable to be corrupted by the order of evaluation rules (precedence rules). Here is an example of a macro, which won't work.

```
#define     DOUBLE(n)          n + n
```

Now if we have a statement,

```
z = DOUBLE(p) *  q;
```

This will be expanded to

```
z = p + p * q;
```

And since * has a higher priority than +, the compiler will treat it as.

```
z = p + (p * q);
```

The problem can be solved using a more robust definition of DOUBLE

```
#define     DOUBLE(n)          (n + n)
```

Here, the braces around the definition force the expression to be evaluated before any surrounding operators are applied. This should make the macro more reliable.

Check Your Progress 1

1. Write a macro to evaluate the formula *f(x) = x*x + 2*x + 4.*

……..……

……..

2. Define a preprocessor macro *swap(t, x, y)* that will swap two arguments *x* and *y* of a given type *t*.

……

……

3. Define a macro called *AREA*, which will calculate the area of a circle in terms of radius.

……

……

4. Define a macro called *CIRCUMFERENCE*, which will calculate the circumference of a circle in terms of radius.

……

……

5. Define a macro to display multiplication table.

……

……

6. Define a macro to find sum of *n* numbers.

……

……

……..

11.4 READING FROM OTHER FILES USING # *include*

The preprocessor directive *#include* is an instruction to read in the entire contents of another file at that point. This is generally used to read in header files for library functions. Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions. The syntax is as follows:

#include <filename.h>

or

#include "*filename.h*"

The above instruction causes the contents of the file "filename.h" to be read, parsed, and compiled at that point. The difference between the suing of # and " " is that, where the preprocessor searches for the *filename.h.* For the files enclosed in **< >** (less than and greater than symbols) the search will be done in standard directories (include directory) where the libraries are stored. And in case of files enclosed in " " (double quotes) search will be done in "current directory" or the directory containing the source file. Therefore, " " is normally used for header files you've written, and # is normally used for headers which are provided for you (which someone else has written).

Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions. This is ***/usr/include*** for most UNIX systems. And ***c:/tc/include*** for turbo compilers on DOS / WINDOWS based systems.

Use of *#include* for the programmer in multi-file programs, where certain information is required at the beginning of each program file. This can be put into a file by name "globals.h" and included in each program file by the following line:

#include "globals.h"

If we want to make use of inbuilt functions related to input and output operations, no need to write the prototype and definition of the functions. We can simply include the file by writing:

#include <stdio.h>

and call the functions by the function calls. The standard header file *stdio.h* is a collection of function prototype (declarations) and definition related to input and output operations.

The extension "*.h*"', simply stands for "header" and reflects the fact that *#include* directives usually sit at the top (head) of your source files. "*.h*" extension is not compulsory – you can name your own header files anything you wish to, but *.h* is traditional, and is recommended.

Placing common declarations and definitions into header files means that if they always change, they only have to be changed in one place, which is a much more feasible system.

What should you put in header files?

- External declarations of global variables and functions.
- Structure definitions.
- Typedef declarations.

However, there are a few things *not* to put in header files:

- Defining instances of global variables. If you put these in a header file, and include the header file in more than one source file, the variable will end up multiply defined.
- Function bodies (which are also defining instances), may not be put in header files. Since these headers may end you up with multiple copies of the function and hence "multiply defined" errors. People sometimes put commonly-used functions in header files and then use #include to bring them (once) into each program where they use that function, or use #include to bring together the several source files making up a program, but both of these are not good practice. It's much better to learn how to use your compiler or linker to combine together separately-compiled object files.

11.5 CONDITIONAL SELECTION OF CODE USING # *ifdef*

The preprocessor has a conditional statement similar to C's if-else. It can be used to selectively include statements in a program. The commands for conditional selection are; *#ifdef, #else* and *#endif*.

#ifdef

The syntax is as follows:

```
#ifdef  IDENTIFIER_NAME
{
  statements;
 }
```

This will accept a name as an argument, and returns true if the name has a current definition. The name may be defined using a # *define*, the *-d* option of the compiler, or certain names which are automatically defined by the UNIX environment. If the identifier is defined then the statements below #ifdef will be executed

#else

The syntax is as follows:

```
#else
{
statements;
```

}

#else is optional and ends the block started with *#ifdef*. It is used to create a 2 way optional selection. If the identifier is not defined then the statements below *#else* will be executed.

#endif

Ends the block started by *#ifdef* or *#else*.

Where the *#ifdef* is true, statements between it and a following *#else* or *#endif* are included in the program. Where it is false, and there is a following *#else*, statements between the *#else* and the following *#endif* are included. Let us look into the illustrative example given below to get an idea.

Example 11.5

Define a macro to find maximum of 3 or 2 numbers using #ifdef , #else

```
/* Program to find maximum of 2 numbers using #ifdef*/

#include<stdio.h>
#define TWO
main()
{
int a, b, c;
clrscr();

#ifdef   TWO
  {
  printf("\n Enter two numbers: \n");
  scanf("%d %d", &a,&b);
  if(a>b)
          printf("\n Maximum  of two numbers is %d", a);
    else
          printf("\n Maximum  is of two numbers is %d", b);
  }

#endif
}   /* end of main*/
```

OUTPUT

```
Enter two numbers:
33
22
Maximum  of two numbers is  33
```

Explanation

The above program demonstrate preprocessor derivative #ifdef. By using #ifdef TWO has been defined. The program finds out the maximum of two numbers.

11.5.1 Using #ifdef for Different Computer Types

Conditional selection is rarely performed using *#define* values. This is often used where two different computer types implement a feature in different ways. It allows the programmer to produce a program, which will run on either type.

A simple application using machine dependent values is illustrated below.

```
#include <stdio.h>
main()
{
#ifdef HP
{
    printf("This is a HP system \n");
    ……
    ……. /* code for HP  systems*/
     }
 #endif

#ifdef SUN
{
        printf("This is a SUN system \n");
        …….        /* code for SUN Systems
}
#endif
}
```

If we want the program to run on HP systems, we include the directive
#define HP at the top of the program.

If we want the program to run on SUN systems, we include the directive
#define SUN at the top of the program.

Since all you're using the macro HP or SUN to control the #ifdef, you don't need to give it *any replacement* text at all. *Any* definition for a macro, even if the replacement text is empty, causes an #ifdef to succeed.

11.5.2 Using #ifdef to Temporarily Remove Program Statements

#ifdef also provides a useful means of temporarily "blanking out" lines of a program. The lines in the program are preceded by *#ifdef* NEVER and followed by *#endif*. Of course, you should ensure that the name NEVER isn't defined anywhere.

```
#include <stdio.h>
main()
{
…….
#ifdef NEVER
{
    ……
    ……. /* code is skipped */
   #endif
}
```

11.6 OTHER PREPROCESSOR COMMANDS

Other preprocessor commands are:

- **#ifndef** - If this macro is not defined
- **#if** - Test if a compile time condition is true

- **#else** - The alternative for #if. This is part of an #if preprocessor statement and works in the same way with #if that the regular C else does with the regular if.
- **#elif** - enables us to establish an "if…else…if .." sequence for testing multiple conditions.

Example 11.6

```
#if  processor  == intel
#define FILE "intel.h"
#elif  processor  == amd
#define FILE "amd.h"
#if  processor  == motrola
#define FILE "motrola.h"
#endif
#include FILE
```

- # - Stringizing operator, to be used in the definition of macro. This operator allows a formal parameter within macro definition to be converted to a string.

Example 11.7

```
#define multiply (p*q) printf(#pq " = %f", pq)
main()
{
   ……..
   multiply(m*n);
}
```

The preprocessor converts the line *multiply(m*n)* into *printf("m*n" " = %f", m*n);* And then into printf("m*n = %f", m*n);

- Token merge, creates a single token from two adjacent ones within a macro definition.

Example 11.8

```
#define merge(s1,s2)  s1## s2
main()
{
   ……..
   printf("%f", merge(total, sales);
}
```

The preprocessor transforms the statement *merge(total, sales) into printf("%f", totalsales);*

#error - text of error message -- generates an appropriate compiler error message.

Example 11.9

```
#ifdef  OS_MSDOS
#include <msdos.h>
      #elifdef OS_UNIX
          #include ``default.h"
              #else
```

```
        #error Wrong OS!!
#endif
```

 # line

#line number *"string"* – informs the preprocessor that the number is the next number of line of input. *"string"* is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the original source files instead of the intermediate C (translated) source files.

#pragma

It is used to turn on or off certain features. Pragmas vary from compiler to compiler. Pragmas available with Microsoft C compilers deals with formatting source listing and placing comments in the object file generated by the compiler. Pragmas available with Turbo C compilers allows to write assembly language statements in C program.

A control line of the form

#pragma token-sequence

This causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.

Other preprocessor directives are # - Stringizing operator allows a formal parameter within macro definition to be converted to a string. ## - Token merge, creates a single token from two adjacent ones within a macro definition. **#error** - generates an appropriate compiler error message.

Example 11.10

Write a macro to demonstrate #define, #if, #else preprocessor commands.

/* The following code displays 35 to the screen. */

```
#include <stdio.h>
#define CHOICE 100
int my_int = 0;
#if (CHOICE == 100)
        void set_my_int()
         {    my_int = 35;         }
#else
         void set_my_int()
         {
              my_int = 27;
          }
#endif
main ()
{
              set_my_int();
              printf("%d\n", my_int);
}
```

OUTPUT

35

The *my_int* is initialized to zero and *CHOICE* is defined as 100. *#if* derivative checks whether *CHOICE* is equal to 100. Since *CHOICE* is defined as 100, *void set_my_int* is called and *int* is set 35. And the same is displayed on to the screen.

Example 11.11

Write a macro to demonstrate #define, #if, #else preprocessor commands.

```
/*  The following code displays 27 on the screen */

#include <stdio.h>
#define CHOICE 100
int my_int = 0;
#undef CHOICE
#ifdef CHOICE
        void set_my_int()
        {
                my_int = 35;
        }
#else
        void set_my_int()
        {
                my_int = 27;
        }
#endif

main ()
{
        set_my_int();
        printf("%d\n", my_int);
}
```

OUTPUT

27

The *my_int* is initialized to 0 and *CHOICE* is defined as 100. *#undef* is used to undefine CHOICE. *#else* is invoked , *void set_my_int* is called and *int* is set 27. And the same is displayed on to the screen.

11.7 PREDEFINED NAMES DEFINED BY PREPROCESSOR

These are identifiers defined by the preprocessor, and cannot be undefined or redefined. They are:

LINE an integer constant containing the current source line number.

FILE a string containing the name of the file being complied.

DATE a string literal containing the date of compilation, in the form “mm-dd-yyyy”.

TIME a string literal containing the time of compilation, in the form “hh:mm:ss”.

STDC the constant 1. This identifier is defined to be 1 only in the implementations conforming to the ANSI standard.

11.8 MACROS Vs FUNCTIONS

Till now we have discussed about macros. Any computations that can be done on macros can also be done on functions. But there is a difference in implementations and in some cases it will be appropriate to use macros than function and vice versa. We will see the difference between a macro and a function now.

Macros	Functions
Macro calls are replaced with macro expansions (meaning).	In function call, the control is passed to a function definition along with arguments, and definition is processed and value may be returned to call
Macros run programs faster but increase the program size.	Functions make program size smaller and compact.
If macro is called 100 numbers of times, the size of the program will increase.	If function is called 100 numbers of times, the program size will not increase.
It is better to use Macros, when the definition is very small in size.	It is better to use functions, when the definition is bigger in size.

Check Your Progress 2

1. Write an instruction to the preprocessor to include the math library `math.h`.

……

……

2. Write a macro to add user defined header file by name *madan.h* to your program.

……

……

3. What will be the output of the following program?

```
#include<stdio.h>
main()
{
    float  m=7;
  #ifdef DEF
    i*=i;
  #else
    printf("\n%f", m);
  #endif      }
```

……

……

4. Write a macro to find out whether the given character is upper case or not.

……

……

……..

11.9 SUMMARY

The preprocessor makes programs easier to develop and modify. The preprocessor makes C code more portable between different machine architectures and customize the language. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. All preprocessor lines begin with #. C Preprocessor is just a text substitution tool on your source code in three ways: File inclusion, Macro substitution, and Conditional compilation.

File inclusion - inserts the contents of another file into your source file.
Macro Substitution - replaces instances of one piece of text with another.
Conditional Compilation - arranges source code depending on various circumstances.

11.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1.
```
# include<stdio.h>
# define f(x)        x*x + 2 * x + 4
main()
{
int num;
printf("enter value x: ");
scanf("%d",&num);
printf("\nvalue of f(num) is %d", f(num));
}
```

2.
```
# include<stdio.h>
# define swap(t, x, y)  { t  tmp = x; x = y; y = tmp; }
main( )
{
 int      a, b;
 float   p, q;
printf("enter integer values for a, b: ");
scanf("%d %d", &a, &b);
printf("\n Enter float values for p, q: ");
scanf("%f %f", &p, &q);
swap(int, a, b);
printf(" \n After swap the values of  a and b are %d %d", a, b);
swap(float, p, q);
printf("\n After swap the values of p and q are %f %f", p, q);
}
```

3.
```
# include<stdio.h>
# define   AREA(radius)      3.1415 * radius * radius
main( )
{
     int radius;
     printf("Enter value of radius:  ");
     scanf("%d ", &radius );
     printf("\nArea is  %d", AREA(radius));
}
```

4.
```
# include<stdio.h>
# define   CIRCUMFERENCE(radius)        2 * 3.1415 * radius

 main( )
```

```
{
    int radius;
    printf("Enter value for radius ");
    scanf("%d ", &radius );
    printf("Circumference is  %d", CIRCUMFERENCE(radius));
}
```

5.
```
# include<stdio.h>
# define MUL_TABLE(num)              for(n=1;n<=10;n++)   \
                                  printf("\n%d*%d=%d",num,n,num*n)
 main()
{
    int number;      int n;
    printf("enter number");
    scanf("%d",&number);
    MUL_TABLE(number);
 }
```

6.
```
# include<stdio.h>
# define SUM(n)  ( (n * (n+1)) / 2 )
main()
{
    int number;
    printf("enter number");
    scanf("%d", &number);
    printf("\n sum of n numbers %d", sum(number));       }
```

Check Your Progress 2

1.
```
# include <math.h>
```

2.
```
#include "madan.h"
```

3. 7

4.
```
#  include<stdio.h>
# define isupper(c) (c>=65 && c<=90)
main()
{
  char c;
  printf("Enter character:");
  scanf("%c",&c);
  if(isupper(c))
        printf("\nUpper case");
    else
         printf("\nNo it is not an upper case character");
}
```

11.11 FURTHER READINGS

1. The C programming language, *Brian W. Kernighan & Dennis Ritchie*, Pearson Education, 2002.
2. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education, 2002.
3. Computer Programming in C, *Raja Raman. V*, PHI, 2002.

UNIT 12 FILES

Structure

12.0 INTRODUCTION

The examples we have seen so far in the previous units deal with standard input and output. When data is stored using variables, the data is lost when the program exits unless something is done to save it. This unit discusses methods of working with files, and a data structure to store data. C views file simply as a sequential stream of bytes. Each file ends either with an *end-of-file* marker or at a specified byte number recorded in a system maintained, administrative data structure. C supports two types of files called ***binary*** *files* and ***text*** *files*.

The difference between these two files is in terms of storage. In *text files*, everything is stored in terms of text *i.e.* even if we store an integer 54; it will be stored as a 3-byte string - "54\0". In a text file certain character translations may occur. For example a *newline(\n)* character may be converted to a carriage return, linefeed pair. This is what Turbo C does. Therefore, there may not be one to one relationship between the characters that are read or written and those in the external device. A *binary file* contains data that was written in the same format used to store internally in main memory.

For example, the integer value 1245 will be stored in 2 bytes depending on the machine while it will require 5 bytes in a text file. The fact that a numeric value is in a standard length makes binary files easier to handle. No special string to numeric conversions is necessary.

The disk I/O in C is accomplished through the use of library functions. The ANSI standard, which is followed by TURBO C, defines one complete set of I/O functions. But since originally C was written for the UNIX operating system, UNIX standard defines a second system of routines that handles I/O operations. The first method, defined by both standards, is called a buffered file system. The second is the unbuffered file system.

In this unit, we will first discuss buffered file functions and then the unbuffered file functions in the following sections.

12.1 OBJECTIVES

After going through this unit you will be able to:

- define the concept of file pointer and file storage in C;
- create text and binary files in C;
- read and write from text and binary files;
- deal with large set of Data such as File of Records; and
- perform operations on files such as count number of words in a file, search a word in a file, compare two files etc.

12.2 FILE HANDLING IN C USING FILE POINTERS

As already mentioned in the above section, a sequential stream of bytes ending with an *end-of-file* marker is what is called a ***file***. When the file is opened the stream is associated with the file. By default, three files and their streams are automatically opened when program execution begins - the ***standard input***, ***standard output***, and the ***standard error***. Streams provide communication channels between files and programs.

For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables to write data on the screen. Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information, such as size, current file pointer position, type of file etc., to perform operations on the file. This structure also contains an integer called a *file descriptor* which is an index into the table maintained by the operating system namely, the *open file table.* Each element of this table contains a block called *file control block (FCB)* used by the operating system to administer a particular file.

The standard input, standard output and the standard error are manipulated using file pointers *stdin, stdout* and *stderr.* The set of functions which we are now going to discuss come under the category of buffered file system. This file system is referred to as buffered because, the routines maintain all the disk buffers required for reading / writing automatically.

To access any file, we need to declare a pointer to FILE structure and then associate it with the particular file. This pointer is referred as a *file pointer* and it is declared as follows:

*FILE *fp;*

12.2.1 Open A File Using The Function *fopen()*

Once a file pointer variables has been declared, the next step is to open a file. The *fopen()* function opens a stream for use and links a file with that stream. This function returns a file pointer, described in the previous section. The syntax is as follows:

*FILE *fopen(char *filename,*mode);*

where ***mode*** is a string, containing the desired open status. The filename must be a string of characters that provide a valid file name for the operating system and may include a path specification. The legal mode strings are shown below in the table 12.1:

Table 12.1: Legal values to the *fopen( )* mode parameter

MODE	MEANING
"r" / "rt"	opens a text file for read only access
"w" / "wt"	creates a text file for write only access
"a" / "at"	text file for appending to a file
"r+t"	open a text file for read and write access
"w+t"	creates a text file for read and write access,
"a+t"	opens or creates a text file and read access
"rb"	opens a binary file for read only access
"wb"	create a binary file for write only access
"ab"	binary file for appending to a file
"r+b"	opens a binary or read and write access
"w+b"	creates a binary or read and write access,
"a+b"	open or binary file and read access

The following code fragment explains how to open a file for reading.

Code Fragment 1

```
#include <stdio.h>

main ()
  {

    FILE *fp;
    if ((fp=fopen("file1.dat", "r"))==NULL)
    {
   printf("FILE DOES NOT EXIST\n");
   exit(0);
    }
 }
```

The value returned by the *fopen( )* function is a file pointer. If any error occurs while opening the file, the value of this pointer is *NULL*, a constant declared in <*stdio.h*>. Always check for this possibility as shown in the above example.

12.2.2 Close A File Using The Function Fclose( )

When the processing of the file is finished, the file should be closed using the fclose() function, whose syntax is:

*int fclose(FILE *fptr);*

This function flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, and then closes the stream. The return value is 0 if the file is closed successfully or a constant *EOF*, an end-of file marker, if an error occurred. This constant is also defined in <*stdio.h*>. If the function *fclose()* is not called explicitly, the operating system normally will close the file when the program execution terminates.

The following code fragment explains how to close a file.

Code Fragment 2

```
# include <stdio.h>
main ( )
{
 FILE *fp;
 if ((fp=fopen("file1.dat", "r"))==NULL)
    {
     printf("FILE DOES NOT EXIST\n");
     exit(0);
    }
……..
……..
……..
…….
/* close the file */
fclose(fp);
}
```

Once the file is closed, it cannot be used further. If required it can be opened in same or another mode.

Check Your Progress 1

1. How does fopen( ) function links a file to a stream?

……

……

……..….

2. Differentiate between text files and binary files.

……

……

……..

3. What is EOF and what is its value?

……

……

……..….

12.3 INPUT AND OUTPUT USING FILE POINTERS

After opening the file, the next thing needed is the way to read or write the file. There are several functions and macros defined in <*stdio.h*> header file for reading and writing the file. These functions can be categorized according to the form and type of data read or written on to a file. These functions are classified as:

- Character input/output functions
- String input/output functions
- Formatted input/output functions
- Block input/output functions.

12.3.1 Character Input and Output in Files

ANSI C provides a set of functions for reading and writing character by character or one byte at a time. These functions are defined in the standard library. They are listed and described below:

- getc()
- putc()

getc( ) is used to read a character from a file and *putc( )* is used to write a character to a file. Their syntax is as follows:

*int putc(int ch, FILE *stream);*
*int getc(FILE *stream);*

The file pointer indicates the file to read from or write to. The character ***ch*** is formally called an integer in *putc( )* function but only the low order byte is used. On success *putc( )* returns a character(in integer form) written or EOF on failure. Similarly getc( ) returns an integer but only the low order byte is used. It returns *EOF* when end-of-file is reached. *getc( )* and *putc( )* are defined in <stdio.h> as macros not functions.

fgetc() and fputc()

Apart from the above two macros, C also defines equivalent functions to read / write characters from / to a file. These are:

*int fgetc(FILE *stream);*
*int fputc(int c, FILE *stream);*

To check the end of file, C includes the function *feof( )* whose prototype is:

*int feof(FILE *fp);*

It returns **1** if end of file has been reached or **0** if not. The following code fragment explains the use of these functions.

Example 12.1

Write a program to copy one file to another.

```
/*Program to copy one file to another */

 #include <stdio.h>
main( )
{
  FILE *fp1;
  FILE *fp2;
  int ch;
  if((fp1=fopen("f1.dat","r")) == NULL)

   {
    printf("Error opening input file\n");
     exit(0);
    }
       if((fp2=fopen("f2.dat","w")) == NULL)
  {
   printf("Error opening output file\n");
          exit(0);
```

```
        }

        while (!feof(fp1))
        {
          ch=getc(fp1);
          putc(ch,fp2);
        }
        fclose(fp1);
        fclose(fp2);
}
```

OUTPUT

If the file ”f1.dat” is not present, then the output would be:
Error opening input file
If the disk is full, then the output would be:
Error opening output file

If there is no error, then “f2.dat” would contain whatever is present in “f1.dat” after the execution of the program, if “f2.dat” was not empty earlier, then its contents would be overwritten.

12.3.2 String Input/Output Functions

If we want to read a whole line in the file then each time we will need to call character input function, instead C provides some string input/output functions with the help of which we can read/write a set of characters at one time. These are defined in the standard library and are discussed below:

- *fgets( )*
- *fputs( )*

These functions are used to read and write strings. Their syntax is:

*int fputs(char *str, FILE *stream);*
*char *fgets(char *str, int num, FILE *stream);*

The integer parameter in *fgets( )* is used to indicate that at most num-1 characters are to be read, terminating at end-of-file or end-of-line. The end-of-line character will be placed in the string ***str*** before the string terminator, if it is read. If end-of-file is encountered as the first character, EOF is returned, otherwise str is returned. The *fputs( )* function returns a non-negative number or EOF if unsuccessful.

Example 12.2

Write a program read a file and count the number of lines in the file, assuming that a line can contain at most 80 characters.

```
/*Program to read a file and count the number of lines in the file */
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
  FILE *fp;
  int cnt=0;
  char str[80];
```

```
/* open a file in read mode */

  if ((fp=fopen("lines.dat","r"))== NULL)
  {    printf("File does not exist\n");
       exit(0);
  }
/* read the file till end of file is encountered */
  while(!(feof(fp)))
  {  fgets(str,80,fp);      /*reads at most 80 characters in str */
     cnt++;                 /* increment the counter after reading a line */
  }
}/* print the number of lines */
printf("The number of lines in the file is :%d\n",cnt);
fclose(fp);
}
```

OUTPUT

Let us assume that the contents of the file "***lines.dat***" are as follows:

This is C programming.
I love C programming.

To be a good programmer one should have a good logic. This is a must.
C is a procedural programming language.

After the execution the output would be:

The number of lines in the file is: 4

12.3.3 Formatted Input/Output Functions

If the file contains data in the form of digits, real numbers, characters and strings, then character input/output functions are not enough as the values would be read in the form of characters. Also if we want to write data in some specific format to a file, then it is not possible with the above described functions. Hence C provides a set of formatted input/output functions. These are defined in standard library and are discussed below:

fscanf() and ***fprintf()***

These functions are used for formatted input and output. These are identical to *scanf()* and *printf()* except that the first argument is a file pointer that specifies the file to be read or written, the second argument is the format string. The syntax for these functions is:

```
int  fscanf(FILE *fp, char *format,. . .);
int fprintf(FILE *fp, char *format,. . .);
```

Both these functions return an integer indicating the number of bytes actually read or written.

Example 12.3

Write a program to read formatted data (account number, name and balance) from a file and print the information of clients with zero balance, in formatted manner on the screen.

/* Program to read formatted data from a file */

```
#include<stdio.h>
main()
{
  int account;
  char name[30];
  double bal;
  FILE *fp;

 if((fp=fopen("bank.dat","r"))== NULL)
        printf("FILE not present \n");
     else
        do{
          fscanf(fp,"%d%s%lf",&account,name,&bal);
           if(!feof(fp))
               {
               if(bal==0)
               printf("%d %s %lf\n",account,name,bal);
                 }
            }while(!feof(fp));
 }
```

OUTPUT

This program opens a file "***bank.dat***" in the read mode if it exists, reads the records and prints the information (account number, name and balance) of the zero balance records.

Let the file be as follows:

```
101     nuj      1200
102     Raman  1500
103     Swathi  0
104     Ajay     1600
105     Udit     0
```

The output would be as follows:

```
103     Swathi  0
105     Udit     0
```

12.3.4 Block Input/Output Functions

Block Input / Output functions read/write a block (specific number of bytes from/to a file. A block can be a record, a set of records or an array. These functions are also defined in standard library and are described below.

- *fread( )*
- *fwrite( )*

These two functions allow reading and writing of blocks of data. Their syntax is:

*int fread(void *buf, int num_bytes, int count, FILE *fp);*

*int fwrite(void *buf, int num_bytes, int count, FILE *fp);*

In case of *fread(),* buf is the pointer to a memory area that receives the data from the file and in *fwrite(),* it is the pointer to the information to be written to the file. *num_bytes* specifies the number of bytes to be read or written. These functions are quite helpful in case of binary files. Generally these functions are used to read or write array of records from or to a file. The use of the above functions is shown in the following program.

Example 12.4

Write a program using *fread( )* and *fwrite()* to create a file of records and then read and print the same file.

```
/* Program to illustrate the fread() and fwrite() functions*/
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<string.h>

void main()
{
  struct stud
         {
          char name[30];
          int age;
          int roll_no;
         }s[30],st;
  int i;
  FILE *fp;

/*opening the file in write mode*/
       if((fp=fopen("sud.dat","w"))== NULL)
         {   printf("Error while creating a file\n");
                    exit(0);    }

/* reading an array of students */
          for(i=0;i<30;i++)
         scanf("%s %d %d",s[i].name,s[i].age,s[i].roll_no);

 /* writing to a file*/
         fwrite(s,sizeof(struct stud),30,fp);
         fclose(fp);

/* opening a file in read mode */
         fp=fopen("stud.dat","r");

/* reading from a file and writing on the screen */
         while(!feof(fp))
         {
           fread(&st,sizeof(struct stud),1,fp);
           fprintf("%s %d %d",st.name,st.age,st.roll_no);
          }
         fclose(fp);       }
```

OUTPUT

This program reads 30 records (name, age and roll_number) from the user, writes one record at a time to a file. The file is closed and then reopened in read mode; the records are again read from the file and written on to the screen.

Check Your Progress 2

1. Give the output of the following code fragment:

```
#include<stdio.h>
#include<process.h>
#include<conio.h>
main()
{
FILE * fp1, * fp2;
       double a,b,c;

fp1=fopen("file1", "w");
fp2=fopen("file2", "w");

fprintf(fp1,"1 5.34 –4E02");
fprintf(fp2,"-2\n1.245\n3.234e02\n");
     fclose(fp1);
     fclose(fp2);

fp1=fopen("file1", "r");
fp2=fopen("file2","r");

 fscanf(fp1,"%lf %lf %lf",&a,&b,&c);
 printf("%10lf %10lf %10lf",a,b,c);
 fscanf(fp2,"%lf %lf %lf",&a,&b,&c);
 printf("%10.1e %10lf %10lf",a,b,c);

      fclose(fp1);
      fclose(fp2);
  }
```

……

……

……

2. What is the advantage of using fread/fwrite functions?

……

……

……

3. _____ and _____ functions are used for formatted input and output from a file.

……

……

……

12.4 SEQUENTIAL Vs RANDOM ACCESS FILES

We have seen in section 12.0 that C supports two type of files – text and binary files, also two types of file systems – buffered and unbuffered file system. We can also differentiate in terms of the type of file access as Sequential access files and random access files. Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence. All the above examples

that we have considered till now in this unit are performing sequential access. Random access files allow reading data from any location in the file. To achieve this purpose, C defines a set of functions to manipulate the position of the file pointer. We will discuss these functions in the following sections.

12.5 POSITIONING THE FILE POINTER

To support random access files, C requires a function with the help of which the file pointer can be positioned at any random location in the file. Such a function defined in the standard library is discussed below:

The function *fseek( )* is used to set the file position. Its prototype is:

*int fseek(FILE *fp, long offset, int pos);*

The first argument is the pointer to a file. The second argument is the number of bytes to move the file pointer, counting from zero. This argument can be positive, negative or zero depending on the desired movement. The third parameter is a flag indicating from where in the file to compute the offset. It can have three values:

SEEK_SET(or value 0) the beginning of the file,
SEEK_CUR(or value 1) the current position and
SEEK_END(or value 2) the end of the file

These three constants are defined in <*stdio.h*>. If successful *fseek( )* returns zero. Another function *rewind()* is used to reset the file position to the beginning of the file. Its prototype is:

*void rewind(FILE *fp);*

A call to rewind is equivalent to the call

fseek(fp,0,SEEK_SET);

Another function *ftell()* is used to tell the position of the file pointer. Its prototype is:

*long ftell(FILE *fp);*

It returns –1 on error and the position of the file pointer if successful.

Example 12.5

Write a program to search a record in an already created file and update it. Use the same file as created in the previous example.

```
/*Program to search a record in an already created file*/

#include<stdio.h>
#include<conio.h>
#include<stdio.h>
#include<process.h>
#include<string.h>
void main()
{
        int r,found;
        struct stud
         {
```

```
        char name[30];
        int age;
        int roll_no;
       }st;
     FILE *fp;
    /* open the file in read/write mode */

     if((fp=fopen("f1.dat","r+b"))==NULL)
     {  printf("Error while opening the file \n");
       exit(0);
     }

/* Get the roll_no of the student */
    printf("Enter the roll_no of the record to be updated\n");
    found=0;
    scanf("%d",&r);

 /* check in the file for the existence of the roll_no */
    while((!feof(fp)) && !(found))
    {   fread(&st,sizeof(stud),1,fp);
       if(st.roll_no == r)

 /* if roll_no is found then move one record   backward to update it */
        {   fseek(fp,- sizeof(stud),SEEK_CUR);
           printf("Enter the new name\n");
           scanf("%s",st.name);
           fwrite(fp,sizeof(stud),1,fp);
           found=1;
         }
     }
     if (!found)
        printf("Record not present\n");
     fclose(fp);
    }
```

OUTPUT

Let the input file be as follows:

```
Geeta   18      101
Leena   17      102
Mahesh 23       103
Lokesh  21      104
Amit    19      105
```

Let the roll_no of the record to be updated be 106. Now since this roll_no is not present the output would be:

Record not present

If the roll_no to be searched is 103, then if the new name is Sham, the output would be the file with the contents:

```
Geeta   18      101
Leena   17      102
Sham    23      103
Lokesh  21      104
Amit    19       105
```

12.6 THE UNBUFFERED I/O – THE UNIX LIKE FILE ROUTINES

The buffered I/O system uses buffered input and output, that is, the operating system handles the details of data retrieval and storage, the system stores data temporarily (buffers it) in order to optimize file system access. The buffered I/O functions are handled directly as system calls without buffering by the operating system. That is why they are also known as low level functions. This is referred to as unbuffered I/O system because the programmer must provide and maintain all disk buffers, the routines do not do it automatically.

The low level functions are defined in the header file <*io.h*>.

These functions do not use file pointer of type FILE to access a particular file, but they use directly the file descriptors**,** as explained earlier, of type integer**.** They are also called *handles***.**

Opening and closing of files

The function used to open a file is *open( )*. Its prototype is:

int open(char *filename, int mode, int access);

Here *mode* indicates one of the following macros defined in <*fcntl.h*>.

Mode:

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read / Write

The *access* parameter is used in UNIX environment for providing the access to particular users and is just included here for compatibility and can be set to zero. o*pen()* function returns *–1* on failure. It is used as:

Code fragment 2

```
int fd;

if ((fd=open(filename,mode,0)) == -1)
 {    printf("cannot open file\n");
      exit(1);    }
```

If the file does not exist, *open()* the function will not create it. For this, the function *creat()* is used which will create new files and re-write old ones. The prototype is:

*int creat(char *filename, int access);*

It returns a file descriptor; if successful else it returns –1. It is not an error to create an already existing file, the function will just truncate its length to zero. The *access* parameter is used to provide permissions to the users in the UNIX environment.
The function *close()* is used to close a file. The prototype is:

int close(int fd);

It returns zero if successful and –1 if not.

Reading, Writing and Positioning in File

The functions *read()* and *write()* are used to read from and write to a file. Their prototypes are:

*int read(int fd, void *buf, int size);*
*int write(int fd, void *buf, int size);*

The first parameter is the file descriptor returned by *open()*, the second parameter holds the data which must be typecast to the format needed by the program, the third parameter indicates the number of bytes to transferred. The return value tells how many bytes are actually transferred. If this value is –1, then an error must have occurred.

Example 12.6

Write a program to copy one file to another to illustrate the use of the above functions. The program should expect two command line arguments indicating the name of the file to be copied and the name of the file to be created.

```
/* Program to copy one file to another file to illustrate the functions*/
# include<stdio.h>
# include<io.h>
#include<process.h>

typedef char arr[80];
typedef char name[30];

main()
{
arr buf;
name fname, sname;
int fd1,fd2,size;

 /* check for the command line arguments */
if (argc!=3)
 {   printf("Invalid number of arguments\n");
          exit(0);
 }
  if ((fd1=open(argv[1],O_RDONLY))<0)
    {   printf("Error in opening file %s \n",argv[1]);
          exit(0);
        }
if ((fd2=creat(argv[2],0))<0)
       {   printf("Error in creating file %s \n",argv[2]);
          exit(0);}

 open(argv[2],O_WRONLY);
 size=read(fd1,buf,80);  /* read till end of file */

while (size>0)
      {  write(fd2,buf,80);
        size=read(fd1,buf,80);
       }
       close(fd1);
       close(fd2);
}
```

OUTPUT

If the number of arguments given on the command line is not correct then output would be:

Invalid number of arguments

One file is opened in the read mode, and another file is opened in the write mode. The output would be as follows is the file to be read is not present (let the file be *f1.dat*):

Error in opening file *f1.dat*

The output would be as follows if the disk is full and the file cannot be created (let the output file be *f2.dat*):

Error in creating file *f2.dat*

If there is no error contents of *f1.dat* will be copied to *f2.dat*.

lseek()

The function *lseek()* is provided to move to the specific position in a file. Its prototype is:

long lseek(int fd, long offset, int pos);

This function is exactly the same as *fseek()* except that the file descriptor is used instead of the file pointer.

Using the above defined functions, it is possible to write any kind of program dealing with files.

Check Your Progress 3

1. Random access is possible in C files using function _____.

2. Write a proper C statement with proper arguments that would be called to move the file pointer back by 2 bytes.

……

……

3. Indicate the header files needed to use unbuffered I/O.

……

……

……..….

12.7 SUMMARY

In this unit, we have learnt about files and how C handles them. We have discussed the buffered as well as unbuffered file systems. The available functions in the standard library have been discussed. This unit provided you an ample set of programs to start with. We have also tried to differentiate between sequential access as well as random access file. The file pointers assigned to standard input, standard output and standard error are *stdin, stdout,* and *stderr* respectively. The unit clearly explains the different

type of modes oof opening the file. As seen there are several functions available to read/write from the file. The usage of a particular function depends on the application. After reading this unit one must be able to handle large data bases in the form of files.

12.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. *fopen()* function links a file to a stream by returning a pointer to a FILE structure defined in <*stdio.h*>. This structure contains an index called file descriptor to a File Control Block, which is maintained by the operating system for administrative purposes.

2. Text files and binary files differ in terms of storage. In text files everything is stored in terms of text while binary files stores exact memory image of the data i.e. in text files 154 would take 3 bytes of storage while in binary files it will take 2 bytes as required by an integer.

3. *EOF* is an end-of-file marker. It is a macro defined in <*stdio.h*>. Its value is –1.

Check Your progress 2

1. The output would be:

   1.000000 5.340000 –400.000000 -2.0e+00 1.245000 323.400000

2. The advantage of using these functions is that they are used for block read/write, which means we can read or write a large set of data at one time thus increasing the speed.

3. *fscanf()* and *fprintf()* functions are used for formatted input and output from a file.

Check Your progress 3

1. Random access is possible in C files using function *fseek()*.

2. fseek(fp, -2L, SEEK_END);

3. <io.h> and <fcntl.h>

12.9 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication, 2002.
2. C How to Program, *Deitel & Deitel*, Pearson Education, 2002.
3. Practical C Programming, *Steve Oualline*, Oreilly Publication, 2003.

APPENDIX-A

THE ASCII SET

The ASCII (American Standard Code for Information Interchange) character set defines 128 characters (0 to 127 decimal, 0 to FF hexadecimal, and 0 to 177 octal). This character set is a subset of many other character sets with 256 characters, including the ANSI character set of MS Windows, the Roman-8 character set of HP systems, and the IBM PC Extended Character Set of DOS, and the ISO Latin-1 character set used by Web browsers. They are not the same as the EBCDIC character set used on IBM mainframes. The first 32 values are non-printing **control characters**, such as *Return* and *Line feed*. You generate these characters on the keyboard by holding down the Control key while you strike another key. For example, Bell is value 7, Control plus G, often shown in documents as ^G. Notice that 7 is 64 less than the value of G (71); the Control key subtracts 64 from the value of the keys that it modifies. The table shown below gives the list of the control and printing characters.

The Control Characters

Char	Oct	Dec	Hex	Control-Key	Control Action
NUL	0	0	0	^@	Null character
SOH	1	1	1	^A	Start of heading, = console interrupt
STX	2	2	2	^B	Start of text, maintenance mode on HP console
ETX	3	3	3	^C	End of text
EOT	4	4	4	^D	End of transmission, not the same as ETB
ENQ	5	5	5	^E	Enquiry, goes with ACK; old HP flow control
ACK	6	6	6	^F	Acknowledge, clears ENQ logon hand
BEL	7	7	7	^G	Bell, rings the bell...
BS	10	8	8	^H	Backspace, works on HP terminals/computers
HT	11	9	9	^I	Horizontal tab, move to next tab stop
LF	12	10	a	^J	Line Feed
VT	13	11	b	^K	Vertical tab
FF	14	12	c	^L	Form Feed, page eject
CR	15	13	d	^M	Carriage Return
SO	16	14	e	^N	Shift Out, alternate character set
SI	17	15	f	^O	Shift In, resume defaultn character set
DLE	20	16	10	^P	Data link escape
DC1	21	17	11	^Q	XON, with XOFF to pause listings; ":okay to send".
DC2	22	18	12	^R	Device control 2, block-mode flow control
DC3	23	19	13	^S	XOFF, with XON is TERM=18 flow control
DC4	24	20	14	^T	Device control 4
NAK	25	21	15	^U	Negative acknowledge
SYN	26	22	16	^V	Synchronous idle
ETB	27	23	17	^W	End transmission block, not the same as EOT
CAN	30	24	17	^X	Cancel line, MPE echoes !!!
EM	31	25	19	^Y	End of medium, Control-Y interrupt
SUB	32	26	1a	^Z	Substitute
ESC	33	27	1b	^[	Escape, next character is not echoed
FS	34	28	1c	^\	File separator
GS	35	29	1d	^]	Group separator
RS	36	30	1e	^^	Record separator, block-mode terminator
US	37	31	1f	^_	Unit separator

Printing Characters

Char	Octal	Dec	Hex	Description
SP	40	32	20	Space
!	41	33	21	Exclamation mark
"	42	34	22	Quotation mark (" in HTML)
#	43	35	23	Cross hatch (number sign)
$	44	36	24	Dollar sign
%	45	37	25	Percent sign
&	46	38	26	Ampersand
`	47	39	27	Closing single quote (apostrophe)
(	50	40	28	Opening parentheses
)	51	41	29	Closing parentheses
*	52	42	2a	Asterisk (star, multiply)
+	53	43	2b	Plus
,	54	44	2c	Comma
-	55	45	2d	Hyphen, dash, minus
.	56	46	2e	Period
/	57	47	2f	Slant (forward slash, divide)
0	60	48	30	Zero
1	61	49	31	One
2	62	50	32	Two
3	63	51	33	Three
4	64	52	34	Four
5	65	53	35	Five
6	66	54	36	Six
7	67	55	37	Seven
8	70	56	38	Eight
9	71	57	39	Nine
:	72	58	3a	Colon
;	73	59	3b	Semicolon
<	74	60	3c	Less than sign (< in HTML)
=	75	61	3d	Equals sign
>	76	62	3e	Greater than sign (> in HTML)
?	77	63	3f	Question mark
@	100	64	40	At-sign
A	101	65	41	Uppercase A
B	102	66	42	Uppercase B
C	103	67	43	Uppercase C
D	104	68	44	Uppercase D
E	105	69	45	Uppercase E
F	106	70	46	Uppercase F
G	107	71	47	Uppercase G
H	110	72	48	Uppercase H
I	111	73	49	Uppercase I
J	112	74	4a	Uppercase J
K	113	75	4b	Uppercase K
L	114	76	4c	Uppercase L
M	115	77	4d	Uppercase M
N	116	78	4e	Uppercase N

O	117	79	4f	Uppercase O
P	120	80	50	Uppercase P
Q	121	81	51	Uppercase Q
R	122	82	52	Uppercase R
S	123	83	53	Uppercase S
T	124	84	54	Uppercase T
U	125	85	55	Uppercase U
V	126	86	56	Uppercase V
W	127	87	57	Uppercase W
X	130	88	58	Uppercase X
Y	131	89	59	Uppercase Y
Z	132	90	5a	Uppercase Z
[	133	91	5b	Opening square bracket
\	134	92	5c	Reverse slant (Backslash)
]	135	93	5d	Closing square bracket
^	136	94	5e	Caret (Circumflex)
_	137	95	5f	Underscore
`	140	96	60	Opening single quote
a	141	97	61	Lowercase a
b	142	98	62	Lowercase b
c	143	99	63	Lowercase c
d	144	100	64	Lowercase d
e	145	101	65	Lowercase e
f	146	102	66	Lowercase f
g	147	103	67	Lowercase g
h	150	104	68	Lowercase h
i	151	105	69	Lowercase i
j	152	106	6a	Lowercase j
k	153	107	6b	Lowercase k
l	154	108	6c	Lowercase l
m	155	109	6d	Lowercase m
n	156	110	6e	Lowercase n
o	157	111	6f	Lowercase o
p	160	112	70	Lowercase p
q	161	113	71	Lowercase q
r	162	114	72	Lowercase r
s	163	115	73	Lowercase s
t	164	116	74	Lowercase t
u	165	117	75	Lowercase u
v	166	118	76	Lowercase v
w	167	119	77	Lowercase w
x	170	120	78	Lowercase x
y	171	121	79	Lowercase y
z	172	122	7a	Lowercase z
{	173	123	7b	Opening curly brace
|	174	124	7c	Vertical line
}	175	125	7d	Cloing curly brace
~	176	126	7e	Tilde (approximate)
DEL	177	127	7f	Delete (rubout), cross-hatch box