MODULE - 5

Structure and Pointers, Preprocessor Directives

CHAPTER 1: STRUCTURES

1. INTRODUCTION

We have seen that arrays can be used to represent a group of data items that belong to the same type, such as int or float. However, we cannot use an array if we want to represent a collection of data items of different types using a single name.

C supports a constructed data type known as structures, a mechanism for packing data of different types.A structure is a convenient tool a group of logically related data items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a record many other languages. More examples of such structures are:

time	:	seconds, minutes, hours
date	:	day, month, year
book	:	author, title, price, year
city	:	name, country, population
address	:	name, door-number, street, city
inventory	:	item, stock, value
customer	:	name, telephone, city, category

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

2. DEFINING A STRUCTURE

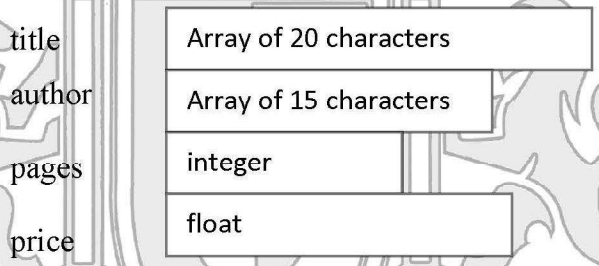
Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book _bank
{
```

```
        char title[20]:
         char author [15]
         int pages;
      float price;
}
```

The keyword struct declares a structure to hold the details of four data fields, namely title, author pages, and price. These fields are called structure elements or members. Each member may belong to a different type of data. book_bank is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called template to represent information as shown below



The general format of a structure definition is as follows:

```
Struct: tag_name
{
        data_type     member1;
        data_type     member2;
        ------        -------
        ------        -------
};
```

In defining a structure you may note the following syntax

1.The template is terminated with a semicolon

2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template

3. The tag name such as book_bank can be used to declare structure variables of its type, later in the program.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

3. DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword **struct**.
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

For example, the statement

struct book_bank, bookl, book2, book3;

declares **book1, book2**, and **book3** as variables of type **struct book_bank**
Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
struct book_bank
{
        char title[20];
        char author [15];
        int pages;
        float price;
};
```

struct book bank book1, book2, book3;

Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as book1. When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

```
struct book bank
{
        char title[20];
        char author [15];
        int pages;
        float price;
};  book1, book2, book3;
```

is valid.

The use of tag name is optional here. For example:

```
struct
{
        ..........
        ..........
        ..........
} book1, book2, book3;
```

Declares book1, book2, and book3 as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for two reasons.
1. Without a tag name, we cannot use it for future declarations:
2. Normally, structure definitions appear at the beginning of the program file, before any variables orfunctions are defined. They may also appear before the main, along with macro definitions, such as #define In such cases, the definition is global and can be used by other functions as well.

1. Type Defined Structures

We can use the keyword typedef to define a structure as follows:

typedef struct

{

```
. . . . . .
type nember1;
type menber2;

. . . . . .
. . . . . .
} type_name;
```

The type_name represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

type_name variable1, variable2,……;

Remember that (1) the name type_name is the type definition name, not a variable and (2) we cannot define a variable with typedef declaration.

4 .ACCESSING STRUCTURE MEMBERS

We can access and assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word title, has no meaning whereas the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the member operator '.' which is also known as 'dot operator' or 'period operator'.

For example,

book1.price

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**.

```
strcpy (book1.title, "BASIC");
strcpy (book1.author, "Balagurusamy");

book1.pages 250;
book1.price 120.50;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s\n", book1.title);
scanf("%d\n", &book1.pages);
```

are valid input statements.

Program

```
struct  personal
 {
        char  name[20];
        int   day;
        char  month[10];
        int   year;
        float salary;
  };


main()
{
        struct personal person;
        printf("Input Values\n");
        scanf("%s %d %s %d %f",
        person.name,
        &person.day,
        person.month,
        &person.year,
        &person.salary);
        printf("%s %d %s %d %f\n",
         person.name,
        person.day,
        person.month,
        person.year,
        person.salary);
  }
```


Output

Input Values
M.L.Goel 10 January 1945 4500
M.L.Goel 10 January 1945 4500.00

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables. Note that the compile-time initialization of a structure variable must have the following elements:

1. The keyword struct.
2. The structure tag name
3. The name of the variable to be declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces
6. A terminating semicolon.

Rules for Initializing Structures

There are a few rules to keep in mind while initializing structure variables at compile-time.
1. We cannot initialize individual members inside the structure template
2. The order of values enclosed in braces must match the order of members in the structure definition
3. It is permitted to have a partial initialization. We can initialize only the first few members and
4. The uninitialized members will be assigned default values as follows:

- Zero for integer and floating point numbers.
- '\0' for characters and strings

6. COPYING AND COMPARING STRUCTURE VARIABLES

Two variables of the same structure type can be copied the same way as ordinary variables. If person1 and person2 belong to the same structure, then the following statements are valid

```
person1 = person2;
person2 = person1;
```


However, the statements such as

```
person1 = person2
person1 =person2
```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Program

```
struct class
{
        int  number;
        char name[20];
        float marks;
```

```
};

main()
{
        int  x;
        struct class student1 = {111,"Rao",72.50};
        struct class student2 = {222,"Reddy", 67.00};
        struct class student3;
         student3 = student2;

         x = ((student3.number ==  student2.number) &&
        (student3.marks  ==  student2.marks)) ? 1 : 0;

        if(x == 1)
         {
                printf("\nstudent2 and student3 are same\n\n");
                printf("%d %s %f\n", student3.number,
                     student3.name,
                     student3.marks);
         }
else
        printf("\nstudent2 and student3 are different\n\n");

  }
```

Output

student2 and student3 are same

222 Reddy 67.000000

7 .OPERATIONS ON INDIVIDUAL MEMBERS

The individual members are identified using the member operator, the dot. A member with the dot operator along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators.

We can perform the following operations:

if (student1.number 111)

```
student1.marks +10.00;
float sum student1.marks + student2.marks;
student2.marks * 0.5;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

```
student1.number++;
++ student1.number;
```

The precedence of the member operator is higher than all arithmetic and relational operators and relational operations and therefore no parentheses are required.

1. Three Ways to Access Members

3 ways to access the members of structure variables.

```
typedef struct
{
    int x;
    int y;
} VECTOR;
VECTOR  v, * ptr;
ptr = & v;
```

The identifier ptr is known as pointer that has been assigned the address of the structure variable n.
Now, the members can be accessed in three ways:

- using dot notation : v.x
- using indirection notation : (*ptr).x
- using selection notation : ptr->x

8. ARRAYS OF STRUCTURES

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student

name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we example:

struct class student [100];

defines an array called student, that consists of 100 elements. Each element is defined to be of the type struct class. Consider the following declaration

```
struct marks
{
        int subject1;
        int subject2;
        int subject3;
};

main()
{
        struct marks student [3] ={{45,68,81}, {75,53,69},{57,36,71}};
```

This declares the student as an array of three elements student[0], student[1], and student[2] and initializes their members as follows:

```
student [o].subject1 45;
student[o].subject2-65;
. . . . . .
. . . . . .
student [2].subject3 71;
```

Note that the array is declared just as it would have been with any other array. Since student is an array, we use the usual array-accessing methods to access individual elements and then the memberoperator to access members.

Program

```
struct marks
{
    int sub1;
    int sub2;
    int sub3;
```

```
    int  total;
  };
  main()
  {
    int  i;
    struct marks student[3] =  {{45,67,81,0},  {75,53,69,0},  {57,36,71,0}};
    struct marks total;
    for(i = 0; i <= 2; i++)
    {
      student[i].total = student[i].sub1 +  student[i].sub2 +  student[i].sub3;
      total.sub1 = total.sub1 + student[i].sub1;
      total.sub2 = total.sub2 + student[i].sub2;
      total.sub3 = total.sub3 + student[i].sub3;
      total.total = total.total + student[i].total;
    printf(" STUDENT          TOTAL\n\n");
     for(i = 0; i <= 2; i++)
     printf("Student[%d]     %d\n", i+1,student[i].total);
    printf("\n SUBJECT  INSTITUTE OF TECHNOLOGY TOTAL\n\n");
    printf("%s     %d\n%s     %d\n%s     %d\n",
        "Subject 1   ", total.sub1,
        "Subject 2   ", total.sub2,
        "Subject 3   ", total.sub3);
    printf("\nGrand Total = %d\n", total.total);
}
```

Output

STUDENT TOTAL

Student[1] 193

Student[2] 197

Student[3] 164

SUBJECT TOTAL

Subject 1 177

Subject 2 156

Subject 3 221

Grand Total – 554

CHAPTER 2: POINTERS

A pointer is a derived data type in C. It is built , from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since these memory addresses are the locations inthe computer memory where program instructions and data are stored, pointers can be used to accessand manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has addedpower and flexibility to the language..

Pointers are used frequently inC, as they offer a number of benefits to the programmers. They include:

1. Pointers are more efficient in handling arrays and data tables.
2.. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as argumentsto other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures,linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Of course, the real power of C lies in the proper use of pointers. In this chapter, we will examine thepointers in detail and illustrate how to use them in program development. Chapter 13 examines the useof pointers for creating and managing linked lists.

UNDERSTANDING POINTERS

The computer's memory is a sequential collection of storage cells as shown in the below Fig. Each cell, commonly known as a byte, has a number called address associated with it. Typically, thc addrcsscs arc numbcrcd consccutivcly, starting from zcro. Thc last addrcss dcpcnds on thc memory size. A computer system having 64 K memory will have its last address as 65,535.

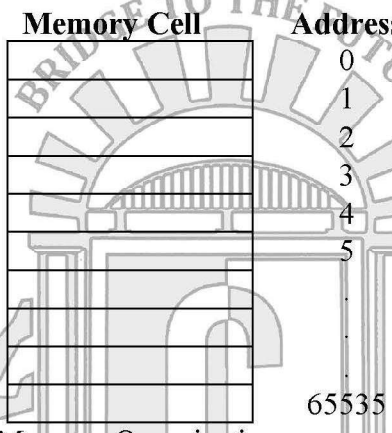


Fig : Memory Organisation

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number.

Consider the following statement

int quantity = 179;

This statement instructs the system to find a locationfor the integer variable quantity and puts the value 179in that location. let us assume that the system haschosen the address location 5000 for quantity. Wemay represent this as shown in Fig.. (Note thatthe address of a variable is the addrcss of thc first bytcoccupicd by that variablc.)

Quantity	Variable
179	Value
5000	Address

Fig : Representing of Variable

During execution of the program, the system always associates the name quantity with the address 5000.

(This is something similar to having a house number as well as a house name.) We may have access to the value 179 by using either the name quantity or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called pointer variables.

A pointer variable is therefore, nothing but a variable that contains an address, which is a location of another variable in memory.

Remember, since a pointer is a variable, its value is also stored in the memory in another location. suppose, we assign the address of quantity to a variable p. The link between the variables p and quantity can be visualized as shown in Fig. The address of p is 5048.

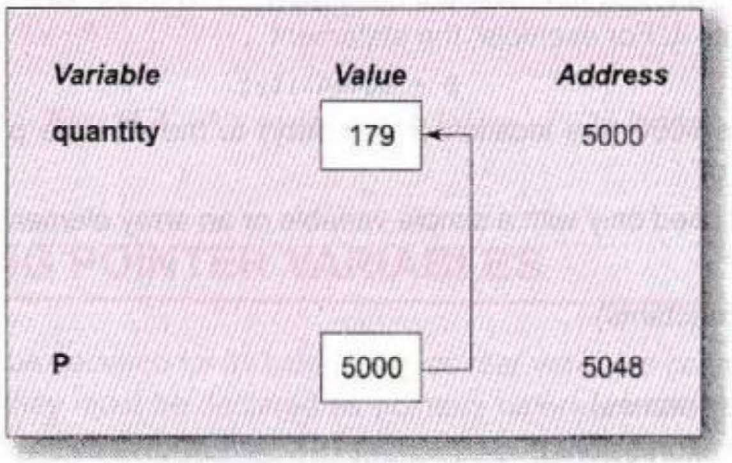
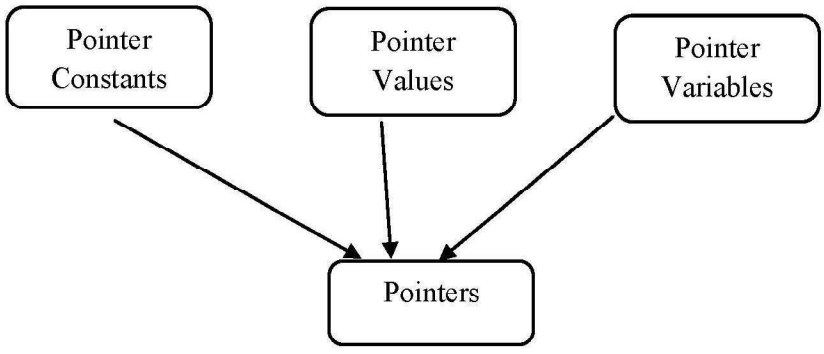


Fig : Pointer Variable

Since the value of the variable p is the address of the variable quantity, we may access the value Of quantity by using the value of p and therefore, we say that the variable p 'points' to the variablequantity. Thus, p gets the name 'pointer'.

Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:



Memory addresses within a computer are referred to as pointer constants. We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator(&). The value thus obtained is known as pointervalue. The pointer value (i.e. the address of a variable) may change from one run of the program toanother.

Once we have a pointer value, it can be stored into another variable. The variable that contains apointer value is called a pointer variable.

ACCESSING THEADDRESS OF A VARIALBE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately.
How can we then determine the address of a variable?
This can be done with the help t of the operator & available in C. We have already seen the use of this address operator in the scanf function. The operator & immediately preceding a variable returns the address of the variable associated with it.

For example, the statement
p =&quantity;
would assign the address 5000 (the location of quantity) to the variable p. The & operator can be remembered as 'address of'.
The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:
1. &125 (pointing at constants).
2. int x[10];
&x (pointing at array names).
3. &(x+y) (pointing at expressions).
If x is an array, then expressions such as
&x[0] and &x[i+3] are valid and represent the addresses of 0th and (i+3)th elements of x .

Program :

Write a program to print the address of a variable along with its value

The program shown in Fig, declares and initializes four variables and then prints out these values with their respective storage locations. Notice that we have used %u format for printing address values. Memory addresses are unsigned integers.

ACCESSING ADDRESSES OF VARIABLES

```
Program
  main()
  {
     char   a;
     int    x;
     float  p, q;

     a  = 'A';
     x  = 125;
     p  = 10.25, q = 18.76;
     printf("%c is stored at addr %u.\n", a, &a);
     printf("%d is stored at addr %u.\n", x, &x);
     printf("%f is stored at addr %u.\n", p, &p);
     printf("%f is stored at addr %u.\n", q, &q);

  }
```

Output

```
A is stored at addr 4436.
125 is stored at addr 4434.
10.250000 is stored at addr 4442.
18.760000 is stored at addr 4438.
```

DECLARING POINTER VARIABLES

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them.

The declaration of a pointer variable takes the following form:

```
data_type    *pt_name;
```

This tells the compiler three things about the variable pt_name.
1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data_type.
For example,

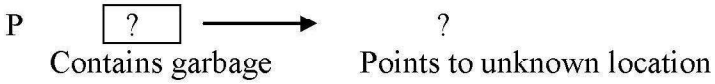
```
int *p; /* integer pointer */
```

declares the variable p as a pointer variable that points to an integer data type. Remember that thetype int refers to the data type of the variable being pointed to by p and not the type of the value of thepointer. Similarly, the statement

```
float *x;      /* float pointer */
```

declares x as a pointer to a floating-point variable.
The declarations cause the compiler to allocate memory locations for the pointer variables p andx. Since the memory locations have not been assigned any values, these locations may contain someunknown values in them and therefore they point to unknown locations as shown:

int *p ;



POINTER DECLARATION STYLE

Pointer variables are declared similarly as normal variables except for the addition of the unary *operator. This symbol can appear anywhere between the type name and the pointer variable name.
Programmers use the following styles:

```
int*    p;      /*style 1 */
int     *p;     /*style 2 */
int     * p;    /*style 3 */
```

However, the style 2 is becoming increasingly popular due to the following reasons:
1. This style is convenient to have multiple declarations in the same statement. Example:
int *p, x, *q;
2. This style matches with the format used for accessing the target values. Example:

```
int x, *p, y;
x= 10;
p = &x;
y = *p;         /* accessing x through p */
*p – 20;        /*assigning 20 to x */
```

INITIALIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as initialization. Aspointed out earlier, all uninitialized pointers will have some unknown values that will be interpreted asmemory addresses. They may not be valid addresses or they may point to some values that are wrong
Since the compilers do not detect these errors, the programs with uninitialized pointers will produceerroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;          /* declaration */
p =&quantity; /* initialization */
```

We can also combine the initialization with the declaration. That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable quantity must be declared before the initialization takes place. Remember, this is an initialization of p and not *p.
We must ensure that the pointer variables always point to the corresponding type of data. For example,

```
float a, b;
int x, *p;
p= &a;/* wrong */
b = *p;
```

will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer. When we declare a pointer to be of int type, the system assumes that any address thatthe pointer will hold will point to an 'integer variable. Since the compiler will not detect such errors, careshould be taken to avoid wrong pointer assignments.
It is also possible to combine the declaration of data variable, the declaration of pointer variable andthe initialization of the pointer variable in one step. For example,

```
int x, *p =&x; /* three in one */
```

is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes p tothe address of x. And also remember that the target variable x is declared first.

The statement

```
int *p = &x, x;
```

is not valid . we could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued.

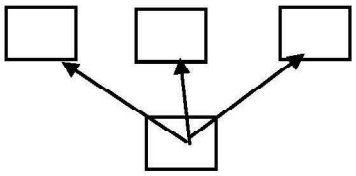
```
int *p = NULL;
int *p = 0;
```

POINTER FLEXIBILITY

Pointers are flexible. We can make the same pointer to point to different data variables in different Statements. Example;

```
int x, y, z, *p;
p = &x;
p = &y;
p = &z;
```

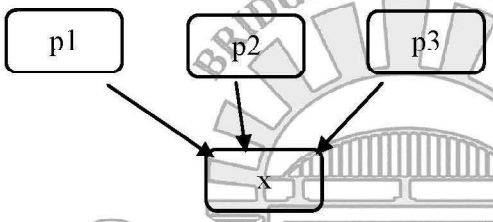
x y z



p

We can also use different pointers to point to the same data variable. Example;

```
int x;
int *pl = &x;
int *p2 = &x;
int *p3 =&x;
```



With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;   / *absolute address */
```

ACCESSING A VARIABLE THROUGH ITS POINTER

once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is done by using another unary operator * (asterisk), usually known as the indirection operator.
Another name for the indirection operator is the dereferencing operator. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

The first line declares quantity and n as integer variables and p as a pointer variable pointing toan integer.
The second line assigns the value 179 to quantity and the third line assigns the address of quantity to the pointer variable p. The fourth line contains the indirection operator *.

When the operator*is placed before a pointer variable in an expression , the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns value of

the variable quantity, because p is the address of quantity. The * can be remembered as 'value at address'. Thus the value of n would be 179.

The two statementsare equivalent to

```
p = &quantity;
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

Program : Write a program to illustrate the use of indirection operator * to access the value pointed to by a pointer.

The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer ptr is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
      x = *(&x) = *ptr = y
&x = &*ptr
```

ACCESSING VARIABLES USING POINTERS

Program

```
  main()
  {
     int   x, y;
     int   *ptr;
     x = 10;
     ptr = &x;
     y = *ptr;
     printf("Value of x is %d\n\n",x);
     printf("%d is stored at addr %u\n", x, &x);
     printf("%d is stored at addr %u\n", *&x, &x);
     printf("%d is stored at addr %u\n", *ptr, ptr);
     printf("%d is stored at addr %u\n", y, &*ptr);
     printf("%d is stored at addr %u\n", ptr, &ptr);
     printf("%d is stored at addr %u\n", y, &y);
     *ptr = 25;
     printf("\nNow x = %d\n",x);

  }
```

Output

```
  Value of x is 10
  10    is stored at addr 4104
  10    is stored at addr 4104
  10    is stored at addr 4104
```

10 is stored at addr 4104
4104 is stored at addr 4106
10 is stored at addr 4108
Now x = 25

The statement ptr = &x assigns the address of x to ptr and y = *ptr assigns the value pointed to by the pointer ptr to y.
Note the use of the assignment statement
*ptr =25;

This statement puts the value of 25 at the memory location whose address is the value of ptr. We know that the value of ptr is the address of x and therefore, the old value of x is replaced by 25. This in effect, is equivalent to assigning 25 to x. This shows how we can change the value of a variable a variable indirectly using a pointer and the indirection operator.

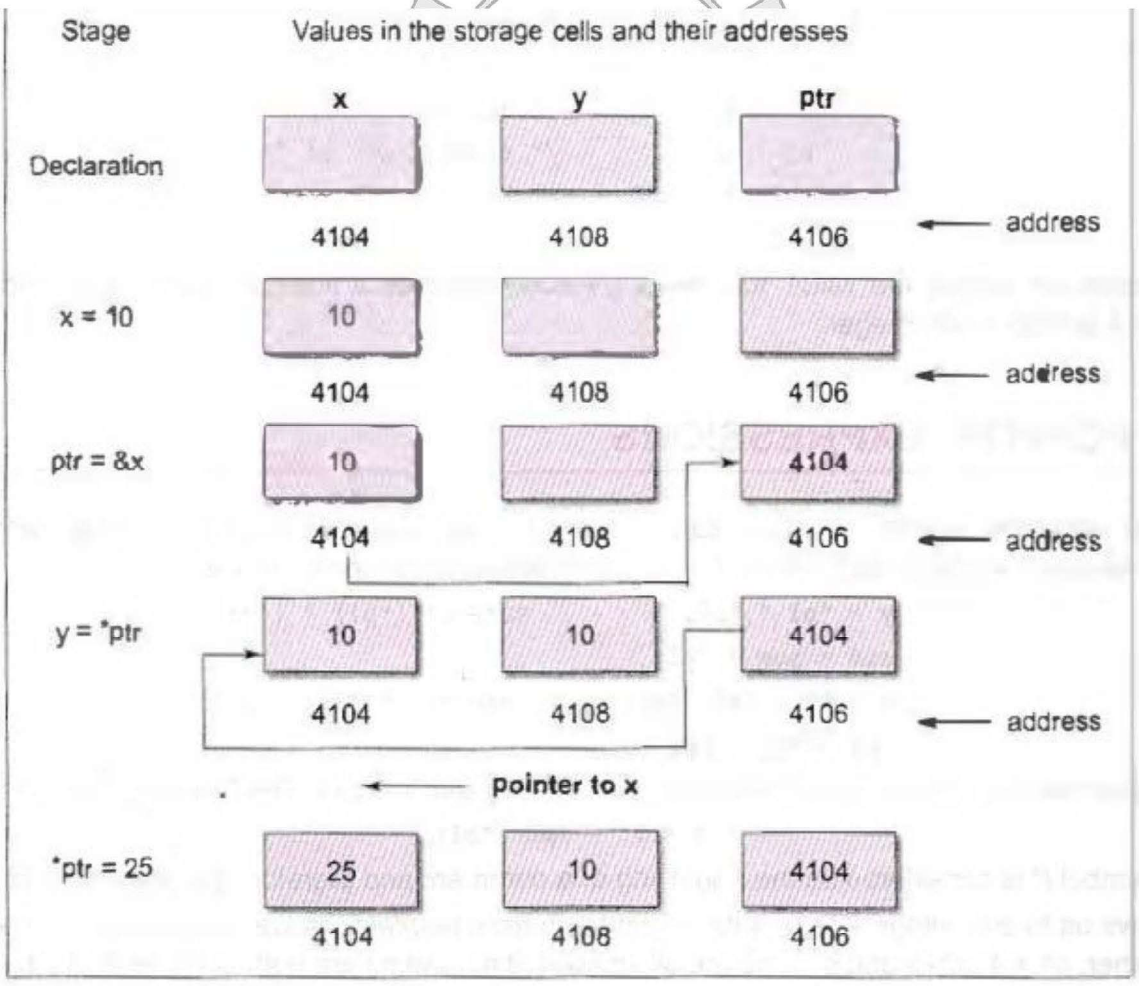


Fig : illustration of the Pointer assignment

POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose product is an array variable of struct type the name product represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
char name [30] ;
int number;
float price;
} product [2], *ptr;
```

This statement declares product as an array of two elements, each of the type struct inventory and ptr as a pointer to data objects of the type struct inventory. The assignment

```
ptr = product;
```

would assign the address of the zeroth element of product to ptr. That is, the pointer ptr will now point to product[0]. Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol -> is called the arrow operator (also known as member selection operator) and is made up of a minus sign and a greater than sign. Note that ptr-> is simply another way of writing product [0] .
When the pointer ptr is incremented by one, it is made to point to the next record, i.e., product[1].
The following for statement will print the values of members of all the elements of product array.

```
For(ptr = product; ptr < product+2; ptr++)
printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr) .number
```

to access the member number. The parentheses around *ptr are necessary because the member operator '.' has a higher precedence than the operator*.

Program :

Write a program to illustrate the use of structure pointers

```
struct invent
```

```
{
   char  *name[20];
   int   number;
   float price;
};
main()
{
  struct invent product[3], *ptr;
  printf("INPUT\n\n");
  for(ptr = product; ptr < product+3; ptr++)
    scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);
  printf("\nOUTPUT\n\n");
   ptr = product;
   while(ptr < product + 3)
   {
      printf("%-20s %5d %10.2f\n",
            ptr->name,
            ptr->number,
            ptr->price);
      ptr++;
   }
}
```

Output

```
INPUT
Washing_machine   5    7500
Electric_iron    12     350
Two_in_one        7    1250

OUTPUT
Washing machine    5   7500.00
Electric_iron          12   350.00
Two_in_one             7   1250.00
```


While using structure pointers, we should take care of the precedence of operators.
The operators '->' and '.', and ( ) and [ ] enjoy the highest priority among the operators. They bind verytightly with their operands.
For example, given the definition

```
struct
{
int count;
float *p;/* pointer inside the struct */

} ptr;     /* struct type pointer */
```

Then the statement.
++ptr->count;

increments count, not ptr. However,
(++ptr)->count;
increments ptr first, and then links count. The statement
ptr++ -> count;
is legal and increments ptr after accessing count.

The following statements also behave in the similar fashion.

*ptr->p	Fetches whatever p points to.
*ptr->p++	Increments p after accessing whatever it points to.
(*ptr->p)++	Increments whatever p points to.
*ptr++->p	Increments ptr after accessing whatever it points to.

CHAPTER 3 : PREPROCESSOR DIRECTIVES

Definition

The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines* or *directives.*

Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.
They all begin with the symbol #, and do not require a semicolon at the end. We have already' used the directives **#define** and **#include**.
A set of commonly used preprocessor directives are:

Directive	*Function*
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if.
#ifndef	Tests whether a macro is not defined.
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails.

These directives can be divided into three categories:
1. Macro substitution directives

2. File inclusion directives
3. Compiler control directives

1. Macro Substitution

Macro substitution is a process where an identifier in a program is replaced by a predefined string
composed of one or more tokens. The preprocessor accomplishes this task under the direction of #define statement. This statement, usually known as a *macro definition* (or simply a macro) takes the
following general form:

#define identifier string

The string may be any text, while the identifier must be a valid C name. The most common forms are:

- Simple macro substitution
- Argumented macro substitution
- Nested macro substitution

Simple Macro Substitution

Simple string replacement is commonly used to define constants. Examples of definition of constants
are:

```
#define   COUNT      100
#define   FALSE      0
#define   SUBJECTS   6
#define   PI         3.1415926
#define   CAPITAL    "DELHI"
```

It is a convention to write all macros in capitals to identify them as symbolic constants. A definition, such as

```
#define M 5
```

will replace all occurrences of M with 5, starting from the line of definition to the end of the program.
However, a macro inside a string does not get replaced. Consider the following two lines:

```
total = M * value;
printf("M = %d\n", M);
```

These two lines would be changed during preprocessing as follows:

```
total = 5 * value;
printf("M = %d\n", 5);
```

Notice that the string "M=%d\n" is left unchanged.
A macro definition can include more than a simple constant value. It can include expressions as well.
Following are valid definitions:

```
#define   AREA   5 * 12.46
#define   SIZE   sizeof(int) * 4
```

```
#define        TWO-PI        2.0 * 3.1415926
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. Consider the evaluation of the equation
ratio =D/A;

where D and A are macros defined as follows:

```
#define  D    45 - 22
#define  A    78 + 32
```

The result of the preprocessor's substitution for Dand A is: ratio=45-22/78+32;
This is different from the expected expression (45 - 22)/(78+32)

The preprocessor performs a literal text substitution, whenever the defined name occurs. This explains why we cannot use a semicolon to terminate the #define statement.

```
#define    TEST     if (x > y)
#define    AND
#define    PRINT    printf("Very Good. \n");
```

to build a statement as follows: TEST AND PRINT
The preprocessor would translate this line to **if(x>y) printf("Very Good. \n");**

Following are a few definitions that might be useful in building error free and more readable programs:

```
#define    EQUALS        ==
#define    AND           &&
#define    OR            ||
#define    NOT_EQUAL     !=
#define    START         main(){
#define    END           }
#define    MOD           %
#define    BLANK_LINE    printf("\n");
#define    INCREMENT     ++
```

An example of the use of syntactic replacement is:

```
START
---
---
if(total EQUALS 240 AND average
EQUALS                    60)

INCREMENT count;
---
---
END
```

Macros with Arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes
the form: **#define identifier(f1,f2,….fn) string**

The identifiers f1, f2,…,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.
Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called. the preprocessor substitutes the string, replacing the
formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

```
#define CUBE(x) (x*x*x)
```

If the following statement appears later in the program

```
volume = CUBE(side);
```

Then the preprocessor would expand this statement to:

```
volume = (side * side * side );
```

Consider the following statement: `volume = CUBE(a+b);`

This would expand to: `volume = (a+b * a+b * a+b);`

which would obviously not produce the correct results. This is because the preprocessor performs a
blind test substitution of the argument a+b in place of x. This shortcoming can be corrected by using
parentheses for each occurrence of a formal argument in the *string*.
Example:

```
# define CUBE(x) ((x)*(x)*(x))
```

This would result in correct expansion of CUBE(a+b) as:

```
volume = (a+b) * (a+b) * (a+b));
```

Remember to use parentheses for each occurrence of a formal argument, as well as the whole *string*.
Some commonly used definitions are:

```
#define MAX(a,b) (((a)>(b))?(a):(b))
#define MIN(a,b) (((a)<(b))?(a):(b))
#define ABS(x) (((x) > 0) ? (x) : (-(x)))
#define STREQ(s1,s2) (strcmp((s1,) (s2)) == 0)
#define STRGT(s1,s2) (strcmp((s1,) (s2)) > 0)
```

The argument supplied to a macro can be any series or characters. For example, the definition

```
#define PRINT(variable, format) printf("variable = %format \n", variable)
```

can be called-in by PRINT(price*quantity,f); The preprocessor will expand this as

```
printf( "price * quantity = %f\n", price * quantity);
```

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be

nested.
For instance, consider the following macro definitions.

```
#define   M          5
#define   N          M+1
#define   SQUARE(x)  ( (x) * (x) )
#define   CUBE(x)    (SQUARE (x) * (x))
#define   SIXTH(x)   (CUBE(x) * CUBE(x))
```

The preprocessor expands each #define macro, until no more macros appear in the text. For example, the last definition is first expanded into

((SQUARE(x) * (x) * (SQUARE(x) * (x)))

Since SQUARE (x) is still a macro, it is further expanded into ((((x)*(x)) * (x)) * (((x) * (x)) * (x)))
which is finally evaluated as x^6.
Macros can also be used as parameters of other macros. For example, given the definitions of M and
N, we can define the following macro to give the maximum of these two:

```
#define MAX(M,N) (((M) > (N) ) ? (M) : (N))
```

Macro calls can be nested in much the same fashion as function calls. Example:

```
#define HALF(x)   ( (x)/2.0)
#define   Y       HALF(HALF(x))
```

Similarly, given the definition of MAX(a,b) we can use the following nested call to give the maximum
of the three values x,y, and z: MAX (x, MAX(y,z))

Undefining a Macro

A defined macro can be undefined, using the statement **#undef *identifier***. This is useful when we want to restrict the definition only to a particular part of the program.

2. *File Inclusion*

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive

```
#include "filename"
```

where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.
Alternatively, this directive can take the form #include <filename> without double quotation

marks. In this case, the file is searched only in the standard directories.
If an included file is not found, an error is reported and compilation is terminated. Assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the
program as:

```
#include <stdio.h>
#include "SYNTAX.C"
#include "STAT.C"
#include "TEST.C"
#define M     100
main()
{
        ---
        ---
        ---
}
```

3. Compiler Control Directives

Suppose a customer has two different types of computers and you are required to write a program
that will run on both the systems. We want to use the same program, although certain lines of code must be different for each system.
While developing large programs, we may face one or more of the following situations:

1. We have included a file containing some macro definitions. It is not known whether a particular
   macro (say, TEST) has been defined in that header file. However, you want to be certain that Test
   is defined (or not defined).
2. We have included a file containing some macro definitions. It is not known whether a particular
   macro (say, TEST) has been defined in that header file. However, you want to be certain that Test
   is defined (or not defined).
3. We are developing a program (say, for sales analysis) for selling in the open market. Some customers may inelet on having certain additional features. However, you would like to have a
   single program that would satisfy both types of customers.
4. Suppose we are in the process of testing the program, which is rather a large one. You would
   like to have print calls inserted in certain places to display intermediate results and messages

in order to trace the flow of execution and errors, if any. Such statements are called 'debugging'
statements. You want these statements to be a part of the program and to become 'active' only
when you decide so.

One solution to these problems is to develop different programs to suit the needs of different situations. Another method is to develop a single, comprehensive program that includes all optional codes and then directs the compiler to skip over certain parts of source code when they are not required. Fortunately,
the C preprocessor offers a feature known as *conditional compilation,* which can be used to 'switch' on
or off a particular line or group of lines in a program.

Situation 1

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST
is always defined, irrespective of whether it has been defined in the header file or not. This can be
achieved as follows:

```
#include    "DEFINE.H"
#ifndef     TEST
#define     TEST 1
#endif
.....
```

DEFINE.H is the header file that is supposed to contain the definition of **TEST** macro. The directive.

#ifndef TEST

searches for the definition of **TEST** in the header file and *if not defined,* then all the lines between the
#ifndef and the corresponding **#endif** directive are left 'active' in the program. That is, the preprocessor
directive # **define** TEST is processed.

In case, the TEST has been defined in the header file, the **#ifndef** condition becomes false, therefore
the directive **#define TEST** is ignored. We cannot simply write # **define** TEST 1 because if **TEST** is already defined, an error will occur.

Similar is the case when we want the macro **TEST** never to be defined. Looking at the following code:

```
.....
#ifdef      TEST
#undef      TEST
#endif
....
....
```

This ensures that even if **TEST** is defined in the header file, its definition is removed. Here again

we
cannot simply say **#undef** TEST because, if TEST is not defined, the directive is erroneous.

Situation 2

The main concern here is to make the program portable. This can be achieved as follows:

```
…
…
main()
{
        …
        …
#ifdef   IBM_PC
{
        … …
        … …   code for IBM_PC
        … …
}
#else
{
        … …
        … …   code for HP machine
        … …
}
#endif
        … …
        … …
}
```

If we want the program to run on IBM PC, we include the directive #define IBM_PC in the program; otherwise we don't.

The compiler complies the code for IBM PC if IBM-PC is defined, or the code for the HP machine if it is not.

Situation 3

This is similar to the above situation and therefore the control directives take the following form:

```
#ifdef  ABC
        group-A lines
#else
        group-B lines
#endif
```

Group-A lines are included if the customer ABC is defined. Otherwise, group-B lines are included.

Situation 4

Debugging and testing are done to detect errors in the program. While the Compiler can detect syntactic
and semantic errors, it cannot detect a faulty algorithm where the program executes, but produces

wrong results.

The process of error detection and isolation begins with the testing of the program with a known set
of test data. The program is divided down and printf statements are placed in different parts to see
intermediate results. Such statements are called debugging statements and are not required once the
errors are isolated and corrected. We can either delete all of them or, alternately, make them inactive
using control directives as:

```
… …
… …
#ifdef  TEST
{
        printf("Arrayelements\n");
        for (i = 0; i < m; i++)
          printf("x[%d] = %d\n",I,x[i]);
}
#endif
… …
… …
#ifdef TEST
     printf ( .... ) ;
#endif
```

The statements between the directives #ifdef and #endif are included only if the macro TEST is defined. Once everything is OK, delete or undefine the TEST. This makes the #ifdef TEST conditions
false and therefore all the debugging statements are left out.
The C preprocessor also supports a more general form of test condition - #if directive. This takes the
following form:

```
#if constant expression
{
        statement-1;
        statement-2;
        … …
        … …
}
#endif
```

The *constant-expression* may be any logical expression such as:

```
TEST <= 3
(LEVEL = = 1 || LEVEL == 2)
MACHINE = = 'A'
```

If the result of the constant-expression is nonzero (true), then all the statements between the #if and

#endif are included for processing; otherwise they are skipped. The names TEST, LEVEL, etc. may be defined as macros.

ANSI ADDITIONS

ANSI committee has added some more preprocessor directives. They are:

#elif	Provides alternative test facility
#pragma	Specifies certain instructions
#error	Stops compilation when an error occurs

The ANSI standard also includes two new preprocessor operations:

#	Stringizing operator
##	Token-pasting operator

#elif Directive

The #elif enables us to establish an "if..else..if.." sequence for testing multiple conditions. The general form of use of #elif is:

```
#if expression1
    statement sequence 1
    #elif   expression2
        Statement sequence 2
        ....
        ....
        #elif expression N
    #endif
```

For example:

```
#if MACHINE == HCL
    #define FILE "hcl.h"
    #elif MACHINE==WIPRO
        #define FILE "wipro.h"
        #elif MACHINE==DCM
            #define FILE "dcm.h"
#endif
#include FILE
```

#pragma Directive

The **#pragma** is an implementation oriented directive that allows us to specify various instructions to be given to the compiler. It allows certain functions to be called before main and after main. (functions have no parameters and does not return anything)

#pragma startup – allows program to specify functions that should be called before main().

Syntax: #pragma startup functionname

```
#pragma startup school
School is a function which is called before main().
```

#pragma exit – allows program to specify functions that should be called after main().

Syntax: #pragma exit functionname

```
#pragma exit college
college is a function which is called after main().
#include<stdio.h>
```

```
void first();
void last();
#pragma startup first
#pragma exit last
void main()
{
        printf("\n I'am main");
}
void first()
{
        printf("\n I'am first");
}
void last()
{
        printf("\n I'am last");
}
```

Output:

I'am first
I'am main
I'am last

Stringizing Operator #

ANSI C provides an operator # called stringizing operator to be used in the definition of macro functions. This operator allows a formal argument within a macro definition to be converted to a string.

Consider the example below:

```
#define sum(xy)      printf(#xy "=%f\n",xy)
main()
{
        ……
        …  …
        sum(a+b);
        …  …
}
```

The preprocessor will convert the line sum(a+b); into printf("a+b" "%f\n", a+b); which is equivalent to

printf("a+b" "%f\n", a+b); ANSI standard also stipulates that adjacent strings will be concatenated.

Token Pasting Operator ##

The token pasting operator ## defined by ANSI standard enables us to combine two tokens within a macro definition to form a single token. For example:

```
#define combine(s1,s2) s1 ## s2
main()
{
```

```
        … ..
        … ..
        printf("%f",combine(total,sales));
        … ..
        … ..
    }
```

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

Consider another macro definition: #define print(i) printf("a" #i "=%f", a##i)

This macro will convert the statement print(5); into the statement printf("a5=%f",a5);