UNIT – III

FUNCTIONS

- ✓ 'C' programs are comprised of a set of functions.
- ✓ The functions are normally used to divide a large program into smaller modules which are easier to handle.
- ✓ Each function normally performs a specific task.
- ✓ Every program must contain one function named as main where the program always begins execution. The main program may call other functions within which it may call still other functions.
- ✓ **When a function is called program execution control is transferred to the first statement of the called function**.
- ✓ The function execution is completed when it executes the last statement in the function or it executes a return statement.

DESIGNING STRUCTURED FUNCTIONS:

- ✓ To solve simple programs it is easier, but to solve big programs it is difficult.
- ✓ To make the process simple, we break program into smaller pieces called modules.
- ✓ For example, if our task is to solve college problem then we break it into department pieces.
- ✓ Each piece is separately solved and finally all the pieces are combined to solve the entire problem.
- ✓ The process of subdividing the problem into manageable parts is called top down design.
- ✓ The technique used for passing data to a function is called parameter passing.

FUNCTIONS IN C:

- ✓ A program in C contains several functions and one function must be main() function
- ✓ The program execution starts from opening brace of main() function and ends at the ending brace of main().
- ✓ It is an independent module that performs a specific task
- ✓ **"A function is defined as a self-contained block of statements which are used to perform a specific task"**
- ✓ A function definition is also known as function implementation that includes the following:
    1. Function name
    2. Function type
    3. List of parameters

4. Local variable declarations
5. Function statements
6. A return statement

- ✓ All the six elements are grouped into two parts. They are:
  - ✓ Function header
  - ✓ Function body

General Format of 'C' Function:

```
type function_name(parameters declaration)
{
        local variable declaration;
        statement 1;
        statement 2;
        statement 3;
        …….
        …….
        statement n;
        return(expression);
}
```

ADVANTAGES OF FUNCTIONS:

- ✓ It provides reuse of code
- ✓ It provides understandable and manageable steps
- ✓ It provides rich library usage of code
- ✓ They protect data

USER DEFINED FUNCTIONS:

A function must be declared and defined. When the function is created we need to consider both the function itself and how it interfaces with many other functions. This includes passing of data to the function and returning value back from the function. A function is called with function name and passing any data to it in the parentheses. The calling and passing data to a function is shown below:

functionName(data passed to function);

To see the process of sending and receiving data, let us consider two functions main(), findmax().

```
#include<stdio.h>
main()
{
        void findmax(int , int);
        int f, s;
        printf("Enter the first and second numbers: ");
        scanf("%d%d", &f, &s);
        findmax(f,s);
}
void findmax(int x, int y)
{
        int m;
        if(x > y)
                m=x;
        else
                m=y;
        printf("%d", m);
}
```

In the above program, findmax() function is the called function and the main() function is the calling function. For every function we need to have the following:

- ✓ Function Prototype
- ✓ Function Call
- ✓ Function Definition

Function Prototype:

Before the function is called it must be declared. The declaration statement for the function is called function prototype. It declares the data type of the value that will be returned directly from the function.

void findmax(int, int);

From the above declaration the function findmax() expects two integer values to be sent when the function is called and returns no value. Function prototype may be placed within the function or outside the main function. If placed inside the function then it will be available to only main function. If it is placed outside the main function then it will be available to all the functions in the file. The general format for the function prototype is as follows:

returntype function_name(list of arguments data types);

The use of function prototypes allows compiler to check for data type errors. If the function prototype specified is different to the specified type then an error message is displayed. It

ensures the conversion of all arguments passed to function when it is called to their declared argument data types in the prototype.

Function Call (Calling a Function):

Calling a function is an easy operation. The requirement is the name of the function and the data passed to the function in the parenthesis. The data enclosed in the parenthesis of call statement are called arguments or actual arguments or actual parameters.

The data passed are generally numbers, but sometimes data may not be numbers also. Data can be an expression, then first the expression is solved and the value obtained is passed to function.

findmax(f, s);

In the above syntax, the function name is findmax and two values are passed to the function. As we pass the values to the function, thus this method is called pass by value or call by value method.

Function Definition:

The function we write must begin with a function header line. Each function is defined once but can be used by any function in the program. Every C function consists of two parts, function header and function body.

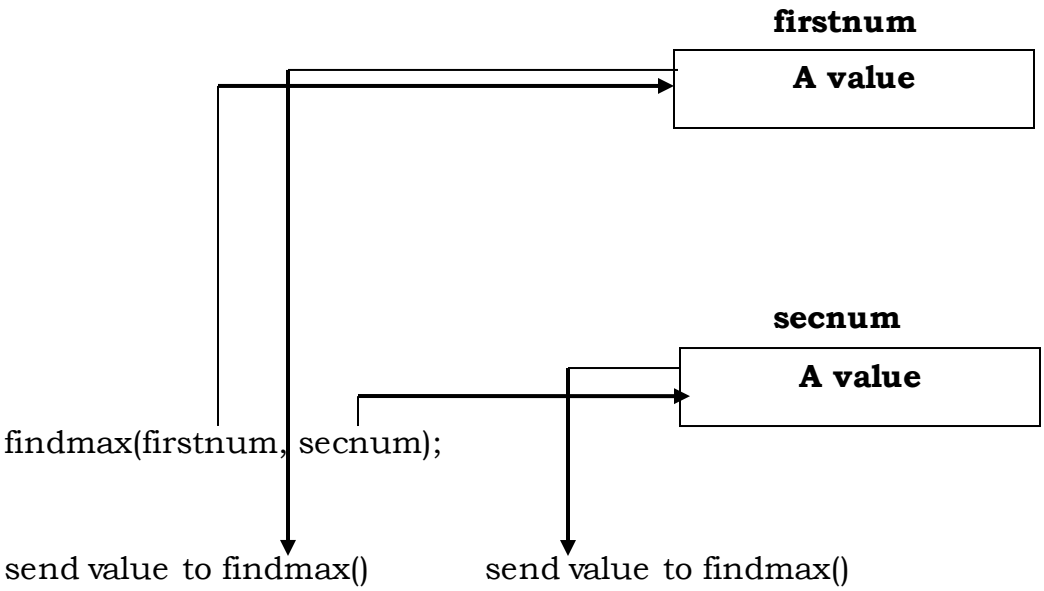
The purpose of the function header is to identify the data type returned, name and number, type and order of values expected by the function. The purpose of function body is to operate on data passed and return at most one value back to the calling function.

The function header line for the findmax() function is as follows.

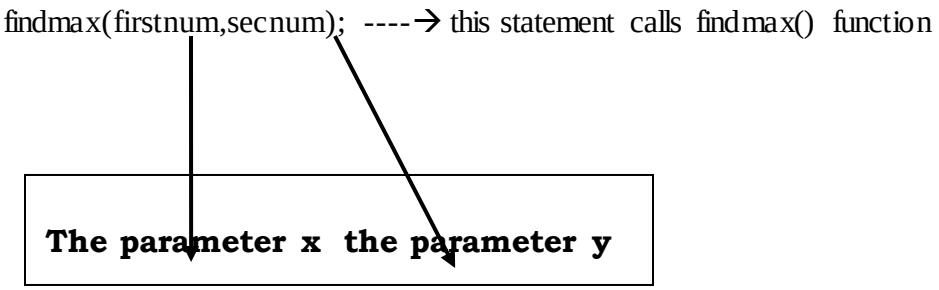
void findmax(int x, int y)

The format of the function definition is as shown below.

```
returntype function_name(arguments) <------ function header
{
        Variable declaration;
        Any other C statements; <---------- function body
}
```



The argument names in the header line are called formal parameters or formal arguments. Thus the arguments x and y are used to store the firstnum and secnum values.



The function name and all parameters in the header line for findmax() function are chosen by the programmer. All the parameters are separated by commas. After the function header, now let us see the function body. It contains an opening brace and a closing brace. All the statements between them belong to function body.

```
{
        Variable declarations;
        C statements;
}
```

```
#include<stdio.h>

main()
{
        void findmax(int , int);
        int firstnum, secnum;
        printf("Enter the first and second number: ");
        scanf("%d%d", &f, &s);
        findmax(firstnum,secnum);
}
```

```
void findmax(int x, int y)
{
        int m;
        if(x >= y)
                m=x;
        else
                m=y;
        printf("%d", m);
}
```

TYPES OF FUNCTIONS

A function depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following:

1. Functions with no parameters / arguments and no return values.
2. Functions with parameters / arguments and no return values.
3. Functions with parameters / arguments and return values.
4. Functions with no parameters / arguments and return values

/* Functions with No Arguments and No Return Values */

```
#include<stdio.h>

void sum();

int main( )
{
        sum();
        return 0;
}

void sum()
{
        int a, b;
        printf("Enter the values of a and b: ");
        scanf("%d%d", &a, &b);
        printf("%d", a+b);
}
```

/* Functions with Arguments and No Return Values*/

```
#include<stdio.h>

void sum(int, int);

int main( )
{
        int a, b;
        printf("Enter the values of a and b: ");
        scanf("%d%d", &a, &b);
        sum(a,b);
        return 0;
}
```

```
void sum(int  x, int  y)
{
        printf("%d",  x+y);
}
```

/* Function with Arguments and Return Values*/

```
#include  <stdio.h>

int  sum(int,  int);

int main( )
{
        int a,b,c;
        printf("Enter  the values  of a and b: ");
        scanf("%d%d",  &a, &b);
        c=sum(a,b);
        printf("%d",  c);
        return  0;
}

int sum(int  x, int  y)
{
        return(x+y);
}
```

/* Functions with No Arguments and Return Values */

```
#include<stdio.h>

int sum();

int main( )
{
        int  c;
        c=sum();
        printf("%d",  c);
        return  0;
}

int sum()
{
        int a, b;
        printf("Enter  the values  of a and b: ");
        scanf("%d%d",  &a, &b);
        return(a+b);
}
```

STANDARD / PRE-DEFINED / LIBRARY FUNCTIONS

Using C arithmetic operators within an assignment statement we can perform addition, subtraction, multiplication and division very easily. But for finding power of a number, finding

square root of a number or determining the absolute values it is not easy. To do such calculations we have standard mathematical functions.

Function	Description	Example	Returned Value
sqrt(x)	square root of x	sqrt(16.00)	4.000000
pow(x,y)	x raised to y power	pow(2,3)	8.000000
exp(x)	e raised to x power	exp(-3.2)	0.040762
log(x)	natural log of x	log(18.697)	2.928363
log10(x)	common log of x	log10(18.697)	1.271772
fabs(x)	absolute value of x	fabs(-3.5)	3.5000000
abs(x)	absolute value of x	abs(-2)	2

To explain about C Mathematical functions, let us consider sqrt() function that calculates the square root value of a number. The square root of a number is computed using the expression

sqrt(number)

where the function name is sqrt and following the parenthesis contains a number for which we have to find the square root. The number passed to the function is called function argument. The argument to the sqrt function is number that is floating point value.

To access the mathematical functions we must include preprocessor directive statement

#inlcude<math.h>

is called preprocessor directive symbol and math.h is a header file. On including this header file the mathematical functions can be used in our programs. The different forms of mathematical functions are as follows:

sqrt(4.0 + 5.3 * 4.0)

sqrt(x)

pow(x, y)

x=3.0 * sqrt(5 * 33 – 13.99) / 5;

sqrt(pow(fabs(x) , y))

PASSING ARRAYS TO FUNCTIONS

1. Passing individual array elements to a function

Array elements can be passed to a function by calling the function:

- ✓ By value i.e., by passing values of array elements to the function.
- ✓ By reference i.e., passing addresses of array elements to the function.

/*Program for call by value*/

```
#include<stdio.h>

main ()
{
        int i;
        int a [5] = {33, 44, 55, 66, 77};
        for (i=0; i<5; i++)
        write (a[i]);
}

write (int n)
{
        printf ("%d\n", n);
}
```

/*Program for call by reference */

```
#include<stdio.h>

main ( )
{
        int i;
        int a[5]={33, 44, 55, 66, 77};
        for (i=0; i<5; i++)
        write (&a[i]);
}

write (int *n)
{
        printf ("%d\n", n);
}
```

2. Passing an Entire Array to a Function

Let us now see how to pass the entire array to a function rather individual elements.

```
#include<stdio.h>

main ()
{
        int a [] = {32, 43, 54, 65, 78};
        display (a, 5);
}

display (int *i, int x)
{
        int j;
        for (j=0; j<5;j++)
        {
                printf ("Address = %u\n", i);
                printf ("Element = %d\n",*i);
                i++;
        }
}
```

Here the display() function is needed to print the array elements. Note that address of the zeroth element is being passed to the display() function.

PASSING POINTERS TO FUNCTIONS:

Passing Addresses:

Let us consider the function scanf(), passing the address of a variable to a function requires placing the address operator "&" before the variable name. Here "&" is referred as "the address of". For example,

&n refers to address of n

&x refers to address of x

```
#include<stdio.h>
main()
{
        int num;
        num=20;
        printf("Enter the number: ");
        printf("%d", num);
        pritnf("%u", &num);
}
```

Functions Returning Pointers

To explain about function returning pointers let us consider an example program.

```
#include <stdio.h>
int* findLarger(int*, int*);
int main()
{
        int numa=0;
        int numb=0;
        int *result;
        printf("\nPointer: Show a function returning pointer :\n");
        printf("-------------------------------------------------\n");
        printf("Input the first number: ");
        scanf("%d", &numa);
        printf("Input the second number: ");
        scanf("%d", &numb);
        result=findLarger(&numa, &numb);
        printf("The number %d is larger.\n",*result);
        return 0;
}
int* findLarger(int *n1, int *n2)
{
```

```
        if(*n1 > *n2)
                return n1;
        else
                return n2;
}
```

RECURSION:

The function called by itself is called self-recursive function or recursive function or direct recursion and this process is often referred as recursion. A function can call second function which in turn calls the first function is called mutual recursion or indirect recursion.

Mathematical Recursion:

The problems solved using an algebraic formula shows recursion explicitly. For example, factorial of a given number **n** denoted as **n!,** where **n** is a positive integer.

0! = 1
1! = 1 * 1 = 1 * 0!
2! = 2 * 1 = 2 * 1!
3! = 3 * 2 * 1 = 3 * 2!
4! = 4 * 3 * 2 * 1 = 4 * 3! and so on …..

Therefore the definition can be written as:

0! = 1

n! = n * (n-1)! for n > = 1

The recursive function in C for factorial function is

```
if n=0
        factorial = 1;
else
        factorial = n*factorial(n-1);
```

/*Recursive program to compute Factorial of an integer*/

```
#include<stdio.h>
long int factorial(int);            /*Function prototype*/
void main()
{
  int n;
  long int f;
  printf("Enter the number: ");
  scanf("%d",&n);
  f=factorial(n);                   /*Function call*/
  printf("Factorial of %d: %ld\n",n,f);
}
/*Function definition*/
```

```
long int factorial(int n)
{
 if(n==0 || n==1)              /*Base criteria*/
   return 1;
 else
   return n*factorial(n-1);    /*Recursive Function call*/
}
```

OUTPUT:

Enter the number: 6

Factorial of 6: 720

/*Recursive program to compute the nth Fibonacci number*/

```
#include<stdio.h>

int fib(int);                  /*Function prototype*/

void main()
{
 int n,f;
 printf("Enter the term: ");
 scanf("%d",&n);
 f=fib(n-1);                   /*Function call*/
 printf("%dth term of Fibonacci series: %d\n",n,f);
}

/*Function definition*/
int fib(int n)
{
 if(n==0||n==1)                /*Base criteria*/
   return n;
 else
   return fib(n-1)+fib(n-2);   /*Recursive Function call*/
}
```

OUTPUT:

Enter the term: 8

8th term of Fibonacci series: 13

/*Recursive program for calculation of GCD(n,m)*/

```
#include<stdio.h>

int gcd(int,int);              /*Function prototype*/

void main()
{
 int n,m,g;
 printf("Enter two numbers: ");
 scanf("%d%d",&n,&m);
 g=gcd(n,m);                   /*Function call*/
```

```
  printf("GCD  of %d & %d: %d\n",n,m,g);
}

/*Function  definition*/
int gcd(int  n,int  m)
{
  if(n==0)                    /*Base criteria*/
    return  m;
  else if(m==0)               /*Base criteria*/
    return  n;
  else
    return  gcd(m,n%m);       /*Recursive  Function  call*/

}
```

OUTPUT:

```
Enter  two numbers:  63 45

GCD of 63 & 45: 9
```

/*Recursive program for Towers of Hanoi: N disks are to be transferred from peg S to peg D with peg A as the intermediate peg*/

```
#include<stdio.h>

void transfer(int,int,int,int);          /*Function  prototype*/

void main()
{
  int  n;
  clrscr();
  printf("Enter  number  of disks: ");
  scanf("%d",&n);
  transfer(n,1,3,2);                     /*Function  call*/
}

void transfer(int  n,int s,int  d,int a)     /*Function  definition*/
{
  if(n==0)                                   /*Base criteria*/
    printf("Move  disk %d from %d to %d\n",n,s,d);
  else
  {
    transfer(n-1,s,a,d);                     /*Recursive  Function  call*/
    printf("Move  disk %d from %d to %d\n",n,s,d);
    transfer(n-1,a,d,s);                     /*Recursive  Function  call*/
  }
}
```

OUTPUT

```
Enter  number  of disks: 3
Move disk  1 from  1 to  3
Move disk  2 from  1 to  2
Move disk  1 from  3 to  2
Move disk  3 from  1 to  3
```

Move disk 1 from 2 to 1
Move disk 2 from 2 to 3
Move disk 1 from 1 to 3

/* Program to perform binary search using recursive function*/

```
#include<stdio.h>

int rbinary_search(int [],int,int,int);      /*Function prototype*/

void main()
{
  int n,key,i,pos;
  int a[100];
  printf("Enter size of the list: ");
  scanf("%d",&n);
  printf("Enter %d elements in ascending order\n",n);
  for(i=0;i<n;i++)
  {
    printf("Enter a[%d]: ",i);
    scanf("%d",&a[i]);
  }
  printf("Enter element to be searched: ");
  scanf("%d",&key);
  pos=rbinary_search(a,0,n-1,key);      /*Recursive function call*/
  if(pos != -1)
    printf("Element found successfully at position %d\n",pos+1);
  else
    printf("Element not found\n");
}

/*Recursive binary search function definition*/
int rbinary_search(int a[],int lb,int ub,int key)
{
  int mid;
  if(lb>ub)                    /*Base criteria*/
    return -1;
  else
  {
    mid=(lb+ub)/2;
    if(a[mid]==key)
      return mid;
    else if(key<a[mid])
      return rbinary_search(a,lb,mid-1,key);      /*Recursive Function call*/
    else
      return rbinary_search(a,mid+1,ub,key);    /*Recursive Function call*/
  }
}
```

OUTPUT:

Enter size of the list: 6
Enter 6 elements in ascending order

Enter a[0]: 2
Enter a[1]: 4
Enter a[2]: 6
Enter a[3]: 8
Enter a[4]: 10
Enter a[5]: 12
Enter element to be searched: 10
Element found successfully at position 5

STRUCTURE:

"A structure is defined as a collection of dissimilar data types under one single name".

or

"A structure is defined as heterogeneous collection of data items specified under a single name".

or

"A structure is defined as collection of elements of different data types".

Structure Definition:

- ✓ There are two ways of defining a structure. They are listed as follows:
  - ✓ Tagged structure
  - ✓ Type definition structure

Tagged Structure:

- ✓ A tagged structure is used to define variables, parameters and return types.
- ✓ The format is as follows:

```
struct tag_name
{
        data type member 1;
        data type member 2;
        data type member 3;
        .
        .
        .
        data type member n;
};
```

Example:

```
struct book_bank
{
        char title[25];
        char author[20];
        int pages;
        float price;
};
```

Type definition with typedef

- ✓ Using the type definition to define a structure is called type declaration structure.
- ✓ We use the keyword **typedef** here.
- ✓ The syntax is:

```
typedef struct
{
        data type  member 1;
        data type  member 2;
        data type  member 3;
        .
        .
        .
        .
        .
        data type  member n;
}tag_name;
```

Example:

```
typedef struct
{
        char  title[25];
        char author[20];
        int  pages;
        float price;
}book_bank;
```

Declaring Structure Variables:

After defining the structure format we declare variables of the structure data type. A structure variable declaration is similar to that of an ordinary variable declaration. It includes

- ✓ **struct keyword**
- ✓ **Structure tag name**
- ✓ **List of variables separated by commas**
- ✓ **A terminating semicolon**

Example:

```
struct book_bank book1, book2, book3;
```

Here book1, book2, book3 are the structure variables of type struct book_bank. Each of these variables can have the four members of the structure book_bank. Then it is

```
struct  book_bank
{
        char  title[25];
        char author[20];
        int  pages;
        float  price;
};
struct book_bank book1, book2, book3;
```

or

```
struct  book_bank
{
        char  title[25];
        char author[20];
        int  pages;
        float  price;
} book1, book2, book3;
```

Accessing Structure Elements / Members:

The members of structure themselves are not variables. The members should be linked to the structure variable in order to make them meaningful members. The link between a member and a variable are established using an operator **"."** which is also known as **dot operator or period operator or member operator**.

Example: **book2.price** is the variable representing the price of book2 and can be treated like any other ordinary variables.

Structure Initialization:

Like other data type variables we can also initialize a structure variable. However a structure must be declared as static.

```
main()
{
        static  struct
        {
                int  age;
                float  height;
        } student = {20, 180.75};
……….
……….
}
```

This assigns the value 20 to student.age and 180.75 to student.height. Suppose you want to initialize more than one structure variable:

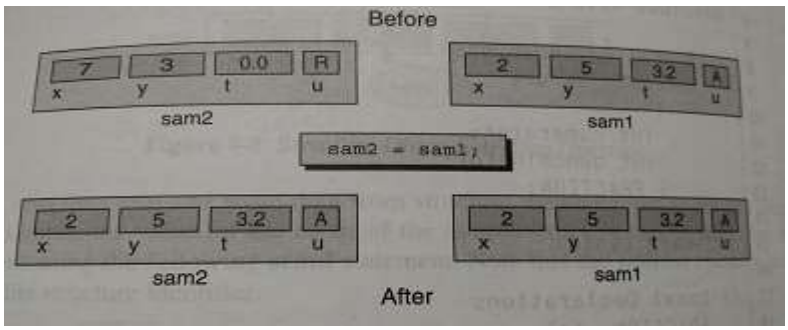
```
main()
{
        struct  st_record
        {
                int  age;
                float  height;
        };
                static  struct st_record student1={20,180.75};
                static  struct st_record student2={22,177.25};
                ……..
}
```

Another method is to initialize a structure variable outside the function.

```
struct st_record
{
        int  age;
        float  height;
} student1 = {20, 180.75};
main()
{
        static struct st_record student2={22, 177.25};
        ……..
}
```

Operations on Structures

- ✓ The only operation that can be performed on structures is assignment operation.
- ✓ By using assignment operator it is possible to copy one structure to another structure if they are same type.
- ✓ For example



ARRAY OF STRUCTURES:

We may declare an array of structures, where each element of the array represents a structure variable. For example:

struct marks student[100];

defines an array called 'student' that consists of 100 elements. Each element is defined to be of the type **struct** marks. Consider the following declaration:

```
struct marks
{
        int sub1;
        int sub2;
        int sub3;
};
main()
{
        struct marks student[3]={ {45,76,87},{78,68,79},{34,23,14} };
}
```

The above declaration shows the student is an array of three elements student[0], student[1], student[2] and initializes their members as follows

student[0].sub1=45;

student[0].sub2=76;

.

.

.

student[2].sub3=14;

student[0].sub1	45
student[0].sub2	76
student[0].sub3	87
student[1].sub1	78
student[1].sub2	68
student[1].sub3	79
student[2].sub1	34
student[2].sub2	23
student[2].sub3	14

An array of structures is stored inside the memory in the same way as multi-dimensional arrays. The array student is as above.

/*WRITE A PROGRAM TO CALCULATE THE SUBJECT-WISE AND STUDENT-WISE TOTALS AND STORE AS A PART OF THE STRUCTURE*/

```
#include<stdio.h>

struct marks
{
        int total;
        int sub1;
        int sub2;
        int sub3;
};

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<3;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<3;i++)
        {
                printf(" stu[%d]: %d\n",i+1,student[i].total);
                printf(" SUBJECT TOTAL\n\n");
                printf("sub1:%d\nsub2:%d\nsub3:%d\n",total.sub1, total.sub2, total.sub3);
                printf("\n Grand total : %d\n",total.total);
        }
}
```

ARRAYS WITHIN STRUCTURES:

C permits the use of array as structure member. We can use single or multi-dimensional array.

```
struct marks
{
        int sub[3];
        int total;
};

struct marks student[2];
```

In the above structure, the member sub contains three elements such as sub[0], sub[1] and sub[2]. These elements can be accessed by using the subscripts. For example **student[0].sub[2],** indicates the marks obtained by student 1 and third subject.

/*WRITE A PROGRAM TO CALCULATE THE SUBJECT-WISE AND STUDENT-WISE TOTALS AND STORE AS A PART OF THE STRUCTURE*/

```
#include<stdio.h>

struct marks
{
        int sub[3];
        int total;
};

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

STRUCTURES WITHIN STRUCTURES OR NESTED STRUCTURES:

Structures within a structure means nesting of structures. To explain about nested structures let us consider the following example.

```
struct salary
{
        char name[30];
        int age;
        float sal;
        struct
        {
                int day;
                char month[20];
                int year;
        }j_date;
}employee;
```

The salary structure contains a member named j_date, which itself is a structure with three members. The members contained in the inner structure are

employee.j_date.day
employee.j_date.month
employee.j_date.year

An inner structure can have more than one variable. For example:

```
struct salary
{
        ……..
        struct
        {
                int day;
                ……..
                ……
        }j_date,r_date;
}employee[10];
```

The accessing of the structure members is shown below:

employee[1].j_date.day

employee[1].r_date.day

We can also use tag names to define inner structures. It is as follows:

```
struct date
{
        int day;
        char month[20];
        int year;
};

struct salary
{
        char name[30];
        int age;
        float sal;
        struct date j_date;
        struct date r_date;
};

struct salary employee[10];
```

UNIONS:

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. The major difference between them is in terms of storage. In structures each member has its own storage location, whereas all the members of a union use the same location. It can handle only one member at a time.

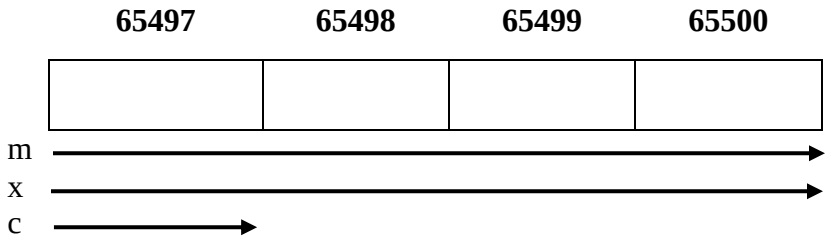
General Format of union:

```
union name
{
        datatype var1;
        datatype var2;
        ---------------
};
```

Example:

```
union item
{
        int m;
        float x;
        char c;
}code;
```

This declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above the member x requires 4 bytes which is the largest among the members. The above figure shows how all the three variables share the same address.

Accessing Union Members:

To access a union member we can use the same syntax that we used for the structure members. The operator is member operator indicated by **". "**

Example:

```
code.m;
code.x;
```

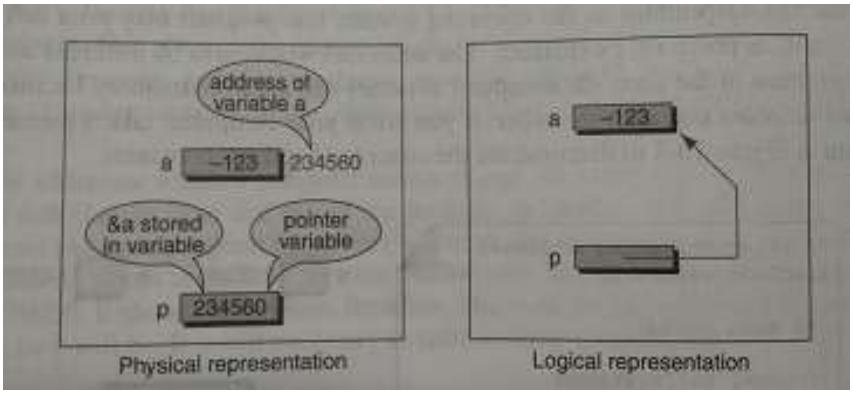
Initialization of Unions

- ✓ The union is initialized with only the first member declared in the union when the variable is declared
- ✓ The other members can be initialized by assigning values by reading
- ✓ When intializing, we must enclose the values with a set of braces even if there is one value

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 address as their values.**

"A pointer is a variable which contains the address of another variable".



Need of Pointers:

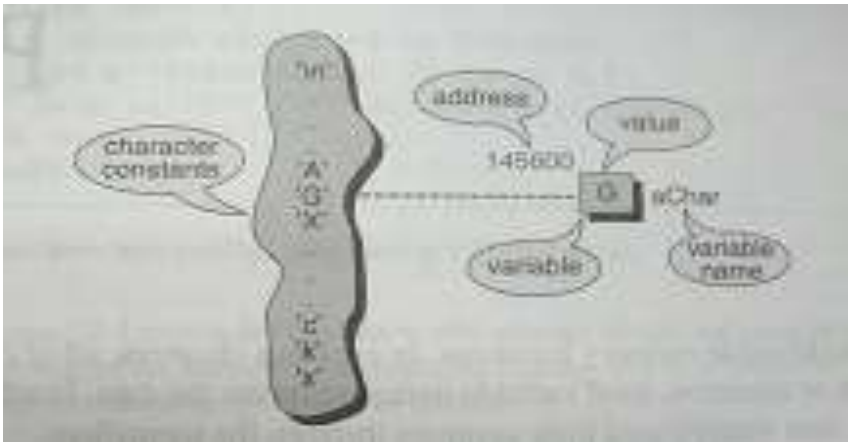
- Basically arrays are static. It means that the maximum possible size of the array has to be declared before its use (i.e., at compile time). It is not always possible to guess the maximum size of an array, because for some applications we need the size of an array to be changed during the program execution. This can be achieved by using the pointers. Pointers allows memory allocation and de-allocation dynamically.
- Pointers are used for establishing links between data elements or objects for some complex data structures such as stacks, queues, linked lists, binary trees and graphs.

Advantages of Pointers

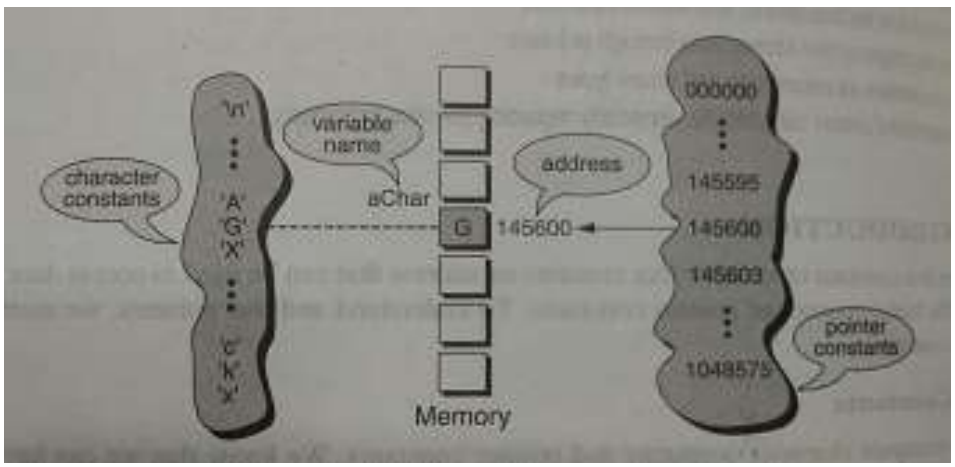
- Pointers are more efficient in handling arrays and data tables.
- Pointers can be used to return multiple values from a function via function arguments.
- Pointers permit reference to functions and there by facilitating passing of functions as arguments to other functions.
- The use of pointer arrays to character string results in saving of data storage space in memory.
- Pointers allow C to support dynamic memory management.
- Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
- Pointers reduce length and complexity of programs.
- They increase the execution speed and reduce the program execution time.
- With the help of pointers, variables can be swapped without physically moving them.

Pointer Constants:

- ✓ To understand pointer constants we compare it with character constants.
- ✓ A character constant is a letter between single quotes. The character is taken from ASCII set.
- ✓ A character constant is a value that can be stored in a variable.
- ✓ For example,

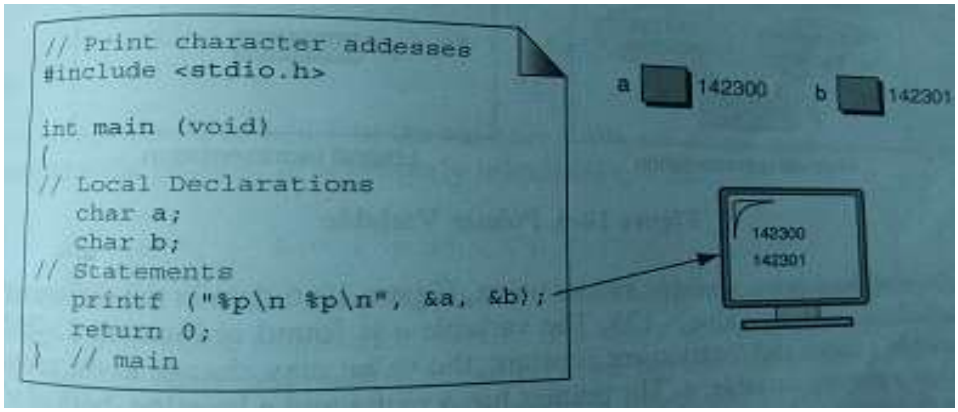


- ✓ Like character constants we have pointer constants that cannot change.
- ✓ For example,

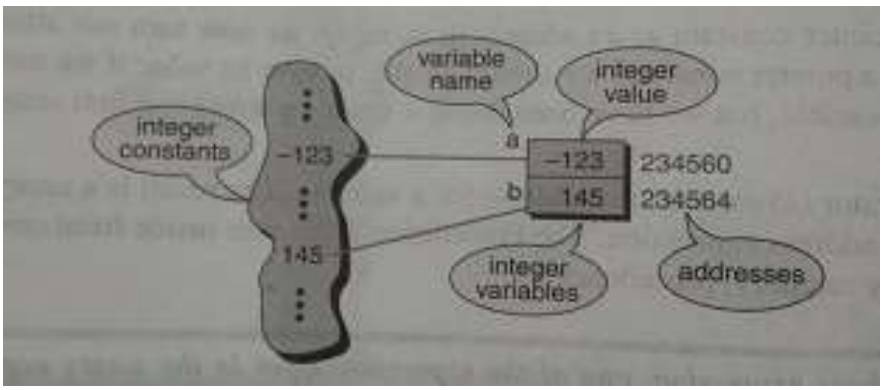


Pointer Values:

- ✓ As we have pointer constant as an address in memory it is possible to store the value at this address known as pointer value.
- ✓ For example, the scanf() with an address operator extracts the address for a variable and known as address expression.
- ✓ The format is **&variable_name**
- ✓ For example,



- ✓ The situation is different when we talk about integers. Since integer needs 4 memory locations then which of these memory locations is used to find the address of the variable. The answer is the location of the first byte.
- ✓ For example,



Declaration of Pointers:

The pointer is declared just like ordinary variable. They have to be declared before they are used. The general syntax of a pointer declaration is as follows:

data_type *Pointer_Name;

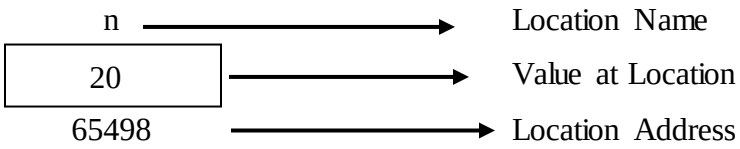
Here the **"*"** tells that variable **Pointer_Name** is pointer type variable, i.e., it holds the address of another variable specified by the **data_type**. Pointer_Name needs a memory location. Pointer_Name points to a variable of type data_type.

Consider the declaration: **int n=20;**

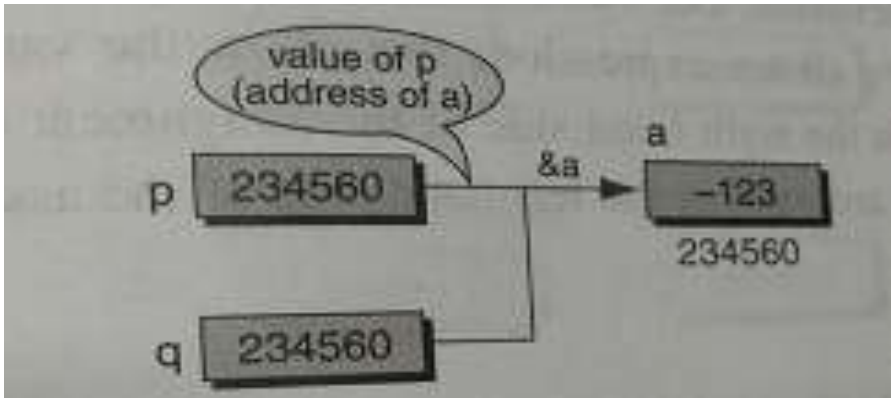
This declaration tells the C compiler to:

1. Reserve space in memory to hold the integer value.
2. Associate the name with this memory location.
3. Store the value 20 at this location.

We may represent **n's** location in the memory by the following memory map:



Pointers are flexible. Multiple pointers can point to the same variable or a pointer can point to different variables during the execution of the program.



Accessing Address of a Variable

The address of the variable is not known immediately when the variable is declared. We can determine the variable address by using an operator called **"address operator".** It is represented as **"&"**. The operator **&** preceding the variable will return the address of the variable that is associated with it.

Example:

```
x=10;
p = &x;
```

Here let **x** is the variable whose value is **10** and suppose if its address is **1000**. Then address of **x** is **1000** which is returned to **p.**

Accessing a Variable through its Pointer:

Once the pointer is assigned with the address of the variable now how can we access the value of the variable using a pointer? It is done by using an operator ***.** This operator is called **value at address operator**. The **'value at address'** operator is also called as **'indirection'** operator. It is also called as **"dereferencing"** operator.

Example:

```
int  x, n, *p;
x = 10;
p = &x;
n = *p;
```

/*PROGRAM TO PRINT ADDRESS AND THE VALUE OF A VARIABLE BY USING '&' AND '*' OPERATORS */

```
#include< stdio.h>
main ()
{
        int n=20;
        printf ("address of n is: %u \n", &n);
        printf ("value of n is: %d \n", n);
        printf ("value of n is: %d",*(&n));
}
```

In the first printf( ) statement **'&'** is used it is **address of operator**. The expression **&n** returns the address of the variable **n**. In the third printf( ) statement we used other pointer operator **'*'** called **'value at address'** operator. It returns the value stored at address n. The above program says the value of ***(&n)** is same as n.

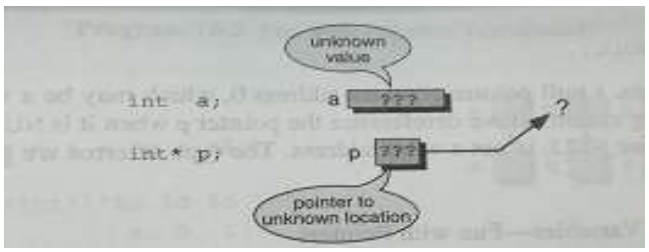
Initialization of Pointer Variables:

The process of assigning the address of a variable to a pointer variable is known as **initialization**. The initialized pointers will produce good results otherwise they will produce erroneous results. Once a pointer variable has been declared we can use assignment operator to initialize the pointer variable.

Example:

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

If they are not initialized then they are left uninitialized as follows:



Pointer Expressions:

In the above example **&n** returns the address of **n**, if we desire this address can be collected into a variable by using

m=&n;

But remember that **m** is not an ordinary variable. It is a variable which contains the address of another variable (n in this case). The following memory map would illustrate the contents of **n** and **m**.

As you can see **n's** value is 20 and **m's** value is **n's** address. Here we can't use **m** in a program without declaring it. And since **m** is a variable which contains the address of **n**, it is declared as

int *m;

This declaration tells compiler that **m** will be used to store the address of an integer variable. In other words **m** points to an integer.

/* PROGRAM TO PRINT ADDRESS AND THE VALUE OF A VARIABLE BY USING & AND * OPERATORS */

```
#include<stdio.h>
main()
{
        int n=20;
        int *m;
        m=&n;
        printf("address of n is: %u \n", &n);
        printf("address of m is:" %u\n", m);
        printf("address of m is: %u\n", &m);
        printf("value of n is: %d\n",*(&n));
        printf("value of n is: %d",*m);
}
```

Declaration versus Redirection:

- ✓ We use the asterisk operator in two ways: for declaration and for redirection.
- ✓ For declaration it is associated with type.
- ✓ For example,

int *pa;

int *pb;

- ✓ Here the asterisk declares that **pa** and **pb** are two integer pointers.

- ✓ We can use asterisk for redirection. When used, the asterisk operator redirects the operation from the pointer variable to data variable.
- ✓ For example, **sum = *pa + *pb;**
- ✓ Here the ***pa** uses the pointer value stored in **pa** and ***pb** uses the pointer value stored in **pb** and redirects the value through the asterisk to desired value and stores in sum.

ARITHMETIC OPERATIONS ON POINTERS:

- ✓ As pointers contain address then we refer address arithmetic as pointer arithmetic.
- ✓ The arithmetic operations performed on pointers is called **pointer arithmetic**.
- ✓ The following operations can be performed on pointers:
  - ✓ **Addition of a number to pointer is possible**
  - ✓ **Subtraction of a number from a pointer is possible**
  - ✓ **Subtraction of pointer from a pointer is possible**
  - ✓ **Comparison of two pointers is possible**
  - ✓ **We can manipulate a pointer with postfix and prefix increment and decrement operations**
  - ✓ **For example,**

p + 5 5 + p p -5 p1 – p2 p1>p2 p++ --p

- ✓ The following operation do not work on pointers:
  - ✓ **Addition of two pointers.**
  - ✓ **Multiplying a pointer with a constant.**
  - ✓ **Division of pointer with a constant.**
  - ✓ **For example,**

p1 + p2 p1 * 5 p1 / 6

/* PROGRAM TO PERFORM POINTER ARITHMETIC */

```
#include<stdio.h>
main( )
{
        int i=5,*i1;
        float j=5.8,*j1;
        char k='z',*k1;
        printf ("Value of i=%d\n", i);
        printf ("Value of j=%f\n", j);
        printf ("Value of k=%c\n", k);
        i1=&i;
        j1=&j;
        k1=&k;
```

```
        printf ("The original value of i1 =%u\n", i1);
        printf ("The original value of j1=%u\n", j1);
        printf ("The original value of k1=%u\n", k1);
        i1++;
        j1++;
        k1++;
        printf ("New value in i1=%u\n", i1);
        printf ("New value in j1=%u\n"j1);
        printf ("New value in k1=%u\n", k1);
}
```

Suppose i, j, k are stored in memory at addresses 65490, 65492 & 65497 the output would be:

Value of i= 5

Value of j= 5.800000

Value of k= z

The original value of i1=65490

The original value of j1=65492

The original value of k1=65497

New value in i1=65492

New value in j1= 65496

New value in k1= 65498

Observe last three lines of the output 65492 is original value in i1 plus 2, 65496 is original value in j1 plus 4 & 65498 is original value in k1 plus 1. This so happens because every time a pointer is incremented its pointer points to the immediately next location of its type.

That is why, when the integer pointer i1 is incremented it points to an address two locations after current location, since an int is always two bytes long. Similarly j1 points to an address four locations after current location and k1 point's one location after the current location.

COMPATIBILITY

Pointers are associated with types. They are not just pointer types but rather they are pointers to specific type. For example integer, char, float pointers.

Pointer Size Compatibility

- ✓ The size of all pointers is same.
- ✓ Every pointer variable holds the address of one memory location.
- ✓ The size of the variable that the pointer reference might be different.
- ✓ It takes the attributes that it references.

```
#include<stdio.h>
int main()
{
        char c;
        char *pc;
        int sizeofc   = sizeof(c);
        int sizeofpc   = sizeof(pc);
        int sizeofstarpc   = sizeof(*pc);
        int a;
        int  *pa;
        int sizeofa   = sizeof(a);
        int sizeofpa   = sizeof(pa);
        int sizeofstarpa   = sizeof(*pa);
        printf("%d",  sizeof(c));
        printf("%d",  sizeof(pc));
        printf("%d",  sizeof(*pc));
        printf("%d",  sizeof(a));
        printf("%d",  sizeof(pa));
        printf("%d",  sizeof(*pa));
        return 0;
}
```

Dereference Type Compatibility:

- ✓ It is the type of the variable that the pointer is referencing.
- ✓ It is invalid to assign one pointer type to another even both contain memory addresses.
- ✓ We cannot use assignment operator with pointers to different types. If we try we can get compiler errors.
- ✓ A pointer to char is compatible with a pointer to a char and a pointer to int is compatible with a pointer to a int.
- ✓ We cannot assign a pointer to a char to a pointer to an int.

Pointer to Void:

- ✓ We can have a pointer to void.
- ✓ A pointer to void is a generic pointer that is not associated with any type but it can allow any type.
- ✓ A void pointer has no object.
- ✓ A void pointer can be assigned to a pointer of reference type and vice versa.
- ✓ For example: **void *pvoid;**
- ✓ A null pointer is a pointer of any type that is assigned with constant NULL. Its value cannot be changed.

- ✓ A variable of pointer to void is a pointer with no reference type that can store only the address of any variable.
- ✓ For example,

void *pvoid;

int *pint = NULL;

char *pchar = NULL;

Casting Pointers

- ✓ Casting is used to solve the problem of incompatibility.
- ✓ We can have explicit assignment between incompatible pointer types using casting.
- ✓ For example, we can cast int to char.

pc = (char *) &a;

LVALUE and RVALUE

- ✓ An expression in C has either **lvalue** or **rvalue**.
- ✓ Every expression has a value and it can be used in two ways.
- ✓ **lvalue –** must be used whenever the object is receiving a value
- ✓ **rvalue –** can be used to supply a value for further use.
- ✓ When can an expression have lvalue and rvalue?
- ✓ There are seven types of expressions that have lvalue.
  - ✓ **identifier**
  - ✓ **expression[…]**
  - ✓ **(expression)**
  - ✓ ***expression**
  - ✓ **expression.name**
  - ✓ **expression->name**
  - ✓ **function call**
- ✓ For example, lvalue expressions

  a = … **a[5] = ….** **(a) = ….** ***p = ….**
- ✓ For example, rvalue expressions

  5 **a + 4** **a * 6** **a[2] + 3** **a++**
- ✓ The right operand of an assignment operator must be an expression.
- ✓ There are 6 operators need an lvalue expression as an operand.
  - ✓ **Address operator**
  - ✓ **Prefix increment**
  - ✓ **Postfix increment**

- ✓ **Prefix decrement**
- ✓ **Postfix decrement**
- ✓ **Assignment (left operand)**

✓ Compiler reports errors when we use rvalue in place of lvalue.

✓ For example

a + 2 = 6 **&(a + 2)** **&3** **(a + 2)++ or ++(a + 2)**

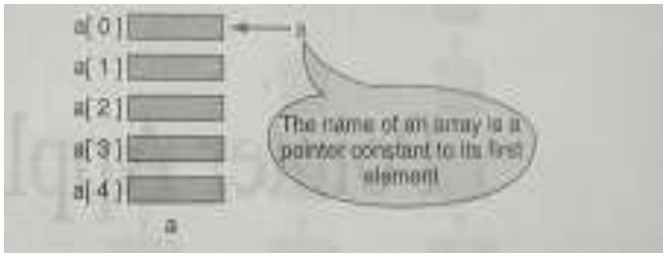
✓ A variable can be used for both lvalue and rvalue depending on its role.

✓ For example, **a = b**

ARRAYS AND POINTERS

✓ The name of the array is the pointer constant to the first element. Since array name is a pointer constant its value cannot be changed.

✓ For example,



✓ The name of the array and the address of the first element both represent the same location in the memory.

✓ We can use the array name as the pointer anywhere as long as it is rvalue.

✓ We can use indirection operator to refer the array name as we dereference first element of the array as array[0].

✓ When we refer the array name we are referring only its first element not the whole array.

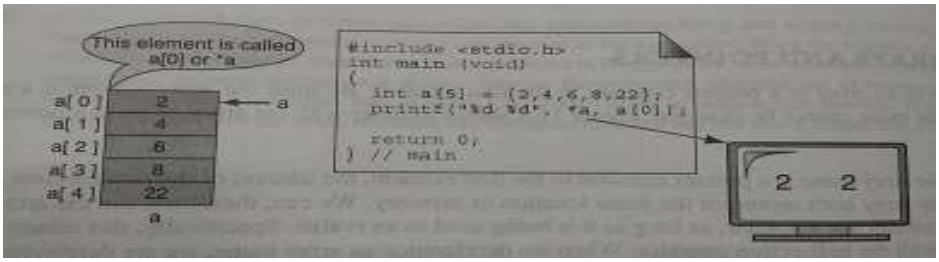
✓ For example:

```
int a[5];
printf("%u  %u", &a[0], a);
```

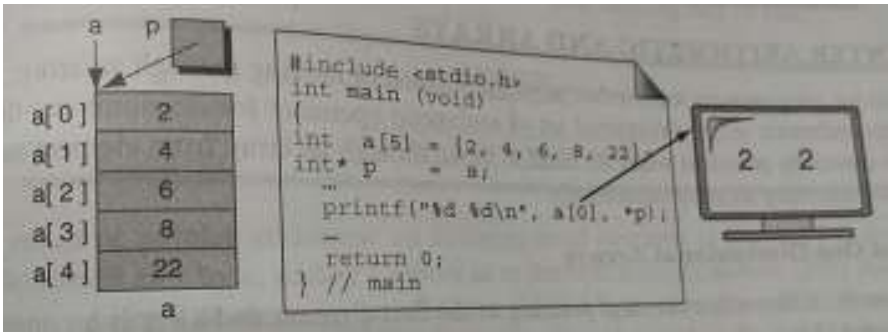
✓ From the above code, we cannot tell the values that are printed but they print the addresses of the first element in both cases.

✓ The name of the array is a pointer constant therefore it cannot be used as lvalue.

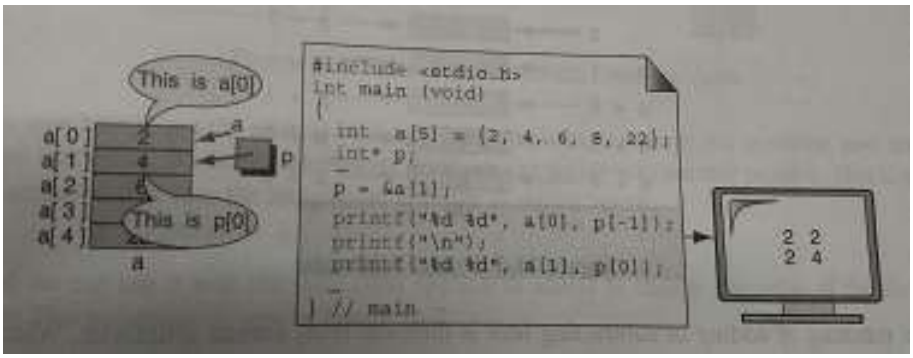
✓ If the first element of the array is holding both pointer and index then what is printed is shown below



✓ If the name of the array is a pointer then see the following example



✓ Another example to differentiate the array and pointer relationship is shown below



POINTERS AND ONE DIMENSIONAL ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of memory to contain all the elements of the array in continuous memory locations. The base address is the location of the first element of the array denoted by a[0]. The compiler also defines the array name as constant pointer to the first element.

Example: **int a [5] = {1, 2, 3, 4, 5};**

Here if the base address is 1000 for "a" and integer occupies 2 bytes then the five elements requires 10 bytes as shown below.

Element →	a[0]	a[1]	a[3]	a[4]	a[5]
Value →	1	2	3	4	5
Address →	1000	1002	1004	1006	1008

The name of the array is **"a"** and it is defined as a constant pointer pointing to the first element of the array and it is a[0] whose base address is 1000 becomes the value of **"a".** It is represented as: **a = &a[0] = 1000;**

If **p** is a pointer of integer type then **p** to point the array **a** is given by the assignment statement

p = a;

which is equivalent to **p = &a[0];**

Now it is possible to access every value of **a** using **p++** to move from one element to another element as

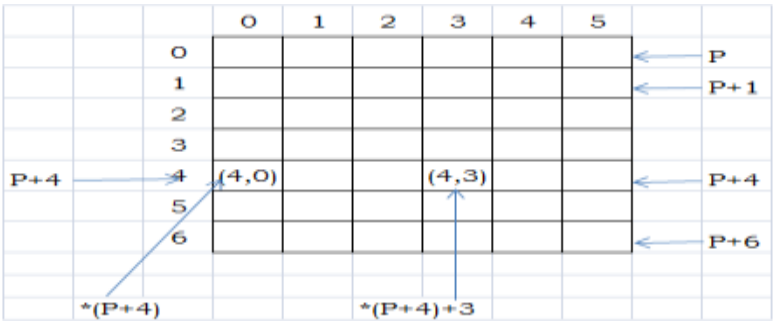
p	**= &a[0]**	**= 1000**
p+1	**= &a[1]**	**= 1002**
p+2	**= &a[2]**	**= 1004**
p+3	**= &a[3]**	**= 1006**
p+4	**= &a[4]**	**= 1008**

The address of the element is calculated by using the formula

address of a[3] = base address + (3 * scale factor of int)

When handling arrays we can use pointers to access the array elements. Hence ***(p+3)** gives the value of **a[3]**. Pointers can also be used to manipulate two dimensional arrays. In one dimensional array **"a"** the expression.

***(a+i) or *(p+i)**



Similarly an element in a 2 dimensional array can be represented by the pointer expression as

***(*(a+i)+j) or *(*(p+i)+j)**

p – **pointer to first row**

p+i – **pointer to ith row**

***(p+i)** – **pointer to the first element in the ith row**

***(p+i)+j** – **pointer to the jth element in the ith row**

***(*(p+i)+j)** – **value stored in the cell (i,j)**

Now how this expression represents the element **a[i][j].** The base address of the array is **&a[0][0]** and compiler allocates contiguous space for the row elements.

Example:

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

POINTERS TO STRUCTURES:

The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to the struct. Such pointers are known as ***'structure pointers'***.

/*EXAMPLE PROGRAM ON STRUCTURE POINTERS*/

```
#include <stdio.h>
main()
{
   struct book
    {
        char title[25];
        char author[25];
        int no;
     };
    struct book b={"SHREETECH C Notes","srinivas",102};
    struct book *ptr;
    ptr=&b;
    printf("%s %s %d\n", b.title, b.author, b.no);
    printf("%s %s %d\n", ptr->title, ptr->author, ptr->no);
}
```

Output:

```
SHREETECH C Notes Srinivas 102
SHREETECH C Notes Srinivas 102
```

The first printf() is as usual. The second printf() however is peculiar. We cannot use ptr.title, ptr.author and ptr.no because ptr is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such case C provides an **operator "->"** called an ***arrow operator*** to refer the structure elements.

/* Example Program on Passing Address of A Structure Variable */

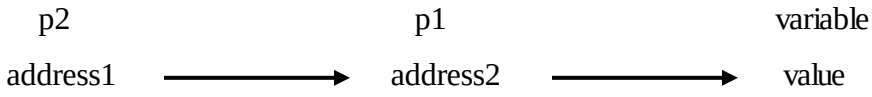
```
#include<stdio.h>
struct book
{
        char title[25];
        char author[25];
        int no;
};
main()
```

```
{
   struct book b={"SHREETECH  C Notes","srinivas",102};
   display(&b);
}
display(struct  book *b)
{
  printf(%s  %s %d\n", b->title, b->author, b->no);
}
```

POINTERS TO POINTERS:

The concept of pointer can be further extended. Pointer is a variable which contains address of another variable. Now this variable itself could be another pointer. Thus we have a pointer which contains another pointer's address.

It is possible to make a pointer variable to point to another pointer variable. Thus we can create a chain of pointers.



Here the pointer variable p2 contains the address of pointer variable p1 which points to the location that contains a value. This is known as **"multiple indirection".** A variable that is a pointer to a pointer must be declared using an additional value at address operator in front of the name.

Example: **int **p2;**

/* PROGRAM TO PRINT ADDRESS AND THE VALUE OF A VARIABLE BY USING &, * AND **OPERATORS */

```
#include<stdio.h>
main ()
{
        int n=20;
        int *m;
        int **p;
        m=&n;
        p=&m;
        printf ("Address of n is: %u \n ", &n);
        printf ("Address of n is: %u \n", m);
        printf ( "Address of m is :%u \n", &m);
```

```
    printf ("Address of m is: %u \n", p);
    printf ("Address of p is: %u \n" &p);
    printf ("Value of m is: %u \n", m);
    printf ("Value of p is: %u \n", p);
    printf ("Value of n is: %d \n", n);
    printf ("Value of n is: %d \n",*(&n));
    printf ("Value of n is %d\n ", *m);
    printf ("Value of n is: %d \n", **p);
}
```

The following memory map would help you in tracing out how the program prints the above output