CONTENTS

1. POINTERS

A pointer is a variable that stores the address of some other variable. It is derived data type that is created from fundamental data types. In some situation is is easy to access the variable through its storage address in the main memory. Now you may wonder what is storage address of a variable is. It is the address at which the variable is stored when you declare it.

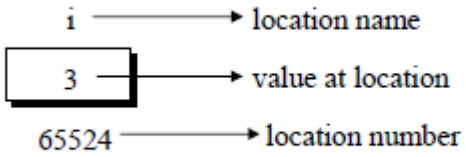
Consider the declaration:

int i=3;

This declaration tells the C compiler to:

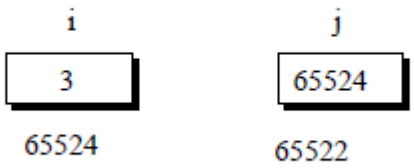
- Reserve space in memory to hold the integer value.
- Associate the name i with this memory location.
- Store the value 3 at this location.

We may represent i's location in memory by the following memory map.



We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a fixed number, because some other time the computer may choose a different location for storing the value 3. The important point is, i's location at which it is stored is a number and it is called as i's address.

Now, we know that pointer is a variable that stores address of other variable. For example, if a variable 'i' having value 3 has been stored at a memory address 65524 in the memory, the pointer to variable 'i' (temporarily named as j) will contain value '65524'. A pointer to 'i' can manipulate the value stored at the location of 'i' i.e. 3.



Pointers give us the freedom that now, the value of variable 'i' can be accessed either from its name i.e. 'i' or by using the variable 'j' which is a pointer to 'i'. We will see how to access the values in upcoming topics.

Advantages of Pointers

The pointers are used to implement the concept of dynamic memory allocation (allocation of memory at run-time) in C language. They are mainly used due to following advantages:

- With the use of pointers, one can return multiple values from a function.
- We can allocate or de-allocate space in memory by using pointers.
- Pointers are used to efficiently handle arrays.
- Pointers are used to implement several Data Structures like Stacks, Queues, Linked Lists and Tress.
- By using pointers, one can pass an array as a parameter to a function.
- The use of pointers results into faster and more efficient code.

All of the above qualities, combined make pointers one of the most powerful features of the C programming language.

C language supports two special operators to determine the address of the variable and the value of variable.

The "address of" operator '&' returns the address of the variable, to which it is precedes. You have already used this operator while using "scanf()" function.

The unary operator '*', also called 'value at address' operator gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Below are few examples that shows the use of these two operators:

```
/*Use of '&' operator*/
#include<stdio.h>
#include<conio.h>
void main()
{
     int i=3;
     printf("\nAddress of i = %u",&i);
```

```
     printf("\nValue of i = %d",i);
     getch();
}
```

Output :

Address of i = 65524
Value of i = 3

The expression '&i' returns the address of the variable i, which in this case happens to be 65524. Since 65524 represent an address, there is no question of a sign being associated with it. Hence it is printed out using %u, which is a format specifier for printing an unsigned integer.

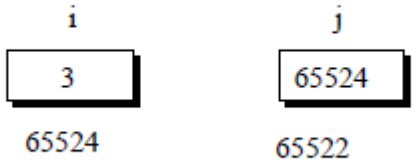
The following program shows the use of both '&' and '*' operators:

```
/*Use of '&' and '*' operator*/
#include<stdio.h>
#include<conio.h>
void main()
{
     int i=3;
     printf("\nAddress of i = %u",&i);
     printf("\nValue of i = %d",i);
     printf("\nValue of i = %d",*(&i));
     getch();
}
```

Output :

Address of i = 65524
Value of i = 3
Value of i = 3

Note that printing the value of *(&i) is same as printing the value of i. The expression &i gives the address of the variable i. This address can be collected in a variable, by saying, 'j = &i;' But remember that j is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (i in this case). Since j is a variable the compiler must provide it space in the memory. The following memory map would illustrate the contents of i and j.



Declaring Pointer Variables

Like other variables in C, a pointer variable has to be declared before it can be used in the program. The general syntax of declaring a pointer is as follows:

```
Data_Type *name
```

Where, "Data_Type" is the type of the pointer variable and "name" is the name of the pointer variable. The asterisk sign (*) is used to inform the compiler that the variable being declared is a pointer. It is very important to note that a pointer can point to the variable having same type as of the pointer variable. For example, the following declaration

```
int *j;
```

tells the compiler that j will be used to store the address of an integer value. In other words j points to an integer. Thus, int *j would mean, the value at the address contained in j is an int.

The declaration of pointer variables can be done with the declaration of other scalar variables of same type. This means the following statements are valid:

```
int i,*iptr;
int j,*jptr
```

Look at the following declarations,

```
int *alpha;
char *ch;
float *s;
```

Here, alpha, ch and s are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers; therefore pointers always contain whole numbers (unsigned integer). The declaration 'float *s' does not mean that 's' is going to contain a floating-point value. What it means is, 's' is going to contain the address of a floating-point value. Similarly, 'char *ch' means that 'ch' is going to contain the address of a char value. Or in other words, the value at address stored in 'ch' is going to be a char.

Like other variables in C, pointers will contain garbage value if you don't initialize them. Therefore, the pointer variables must be properly initialized.

Initializing Pointers

The process of initializing pointers is also similar to that of normal variables in C. They are also initialized by using the assignment operator (=) of C. The difference in the initialization of scalar variables and pointers is that the scalar variables are directly initialized with values but pointers are initialized with addresses. Consider the following declarations:

```
int a=90;    //Correct
int *ptr=655;    //Incorrect
int *ptr=&a;    //Correct
```

The first statement is valid as the value is being assigned to a scalar variable but the second statement will result into an error. The pointer variables are initialized using address of operator. This way, the third statement is completely correct.

The following lines initializes a pointer variable 'ptr' to an address of variable 'var'.

```
int *ptr,var;   //Declaring Pointer variable
ptr=&var;   //Putting the address of var into ptr
```

```
/*Use of '&' and '*' operator*/
#include<stdio.h>
#include<conio.h>
void main()
{
     int i=3;
     int *j;
     j=&i;
     printf("\nAddress of i = %u",&i);
     printf("\nAddress of i = %u",j);
     printf("\nAddress of j = %u",&j);
     printf("\nValue of j = %u",j);
     printf("\nValue of i = %d",i);
     printf("\nValue of i = %d",*(&i));
     printf("\nValue of i = %d",*j);
}
```

Output :

Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3

Pointers can be used to access the address as well as the data of the variables, whose address it contains. To access the address, the name of the pointer variable is used, whereas to access the data stored in the variable pointed by the pointer, * is preceded with the name of the pointer variable. The asterisk sign, when used while declaring a pointer, instructs the compiler that the variable being declared is a pointer. The following program also illustrates the use of pointers.

```
#include<stdio.h>
void main()
{
     int i=100,j=200;
     int *ptr;
     ptr=&i;
     printf("\nThe pointer points to i");
     printf("\nAddress of ptr is = %u",ptr);
     printf("\nValue of i = %d",*ptr);
     ptr=&j;
     printf("\nThe pointer points to j");
     printf("\nAddress of ptr is = %u",ptr);
     printf("\nValue of j = %d",*ptr);
}
```

Output :
The pointer points to i
Address of ptr is 65524
Value of i = 100
The pointer points to j
Address of ptr is 65526
Value of j = 200

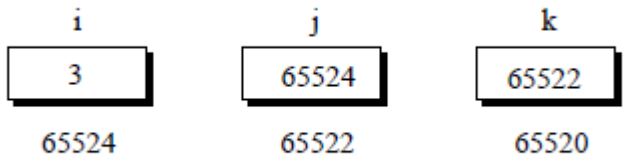
In the above program, a pointer variable "ptr" has been initialized to the address of the variable "i". If we display "ptr" in the printf statement, the address pointed by "ptr" is printed. If we display "*ptr" in the printf statement, the data stored in the location pointed by "ptr" is printed. After this, the pointer is reassigned with the value of variable "j". When we print the same details using printf function, the address and data of variable "j" is printed.

The concept of pointers can be further extended. Pointer, we know is a variable that contains address of some another variable. Now this variable itself might be another pointer. Thus, we now have a pointer that contains another pointer's address. The following example should make this point clear.

```
#include<stdio.h>
void main()
{
     int i=3,*j,**k;
     j = &i;
     k = &j;
     printf("\nAddress of i = %u",&i);
     printf("\nAddress of i = %u",j);
     printf("\nAddress of i = %u",*k);
     printf("\nAddress of j = %u",&j);
     printf("\nAddress of j = %u",k);
     printf("\nAddress of k = %u",&k);
     printf("\nValue of j = %u",j);
     printf("\nValue of k = %u",k);
     printf("\nValue of i = %d",i);
     printf("\nValue of i = %d",*(&i));
     printf("\nValue of i = %d",*j);
     printf("\nValue of i = %d",**k);
}
```

Output :
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j = 65524
Value of k = 65522
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3

The following figure would help you in tracing out how the program prints the above output.



Remember that when you run this program the addresses that get printed might turn out to be something different than the ones shown in the figure. However, with these addresses too the relationship between i, j and k can be easily established.

2. Pointer Arithmetic

The arithmetic operations on pointer can affect the memory location pointed by pointers or the data stored at the pointed location. Consider the program:

```
#include<stdio.h>
void main()
{
     int i=3,*x;
     float j=1.5,*y;
     char k='c',*z;
     x=&i;
     y=&j;
     z=&k;
     printf("\nOriginal address in x = %u",x);
     printf("\nOriginal address in y = %u",y);
     printf("\nOriginal address in z = %u",z);
     x++;
     y++;
     z++;
     printf("\nNew address in x = %u",x);
     printf("\nNew address in y = %u",y);
     printf("\nNew address in z = %u",z);
}

Output:

Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520
```

Observe the last three lines of the output. 65526 is original value in x plus 2, 65524 is original value in y plus 4, and 65520 is original value in z plus 1. This so happens because

every time a pointer is incremented it points to the immediately next location of its type. That is why, when the integer pointer x is incremented, it points to an address two locations after the current location, since an int is 2 bytes long. Similarly, y points to an address 4 locations after the current location and z points 1 location after the current location.

- Adding Numbers to Pointers : Adding a number to a pointer leads the pointer to a new location which is the given by: Original location+(number added*bytes taken by the variable),For example, if an integer pointer "ptr" points to 65524, the statement "ptr=ptr+5" would make to pointer now point to 65524+(2*5) i.e. 65534.
- Subtracting numbers from pointers: The subtraction of pointers is similar to that of addition. The only difference is that in Subtraction the new location will be: original location - (number*bytes taken by variable).

- Subtracting Pointers: If you subtract a pointer from other pointer, the result will be the number of memory locations existing between the address pointed by both the pointers. For example, if a pointer "ptr1" pointing to location 65524, is subtracted from a pointer "ptr2" pointing to location 65532, the result will be 4(if the pointer variable is of integer type).

3. Pointer Expressions

The pointers are generally used in expressions to manipulate the values stored in the memory location, they point to. The following program illustrates the use of pointers in arithmetic expressions to calculate the net amount after duration.

```
#include<stdio.h>
#include<conio.h>
void main()
{
     int pr,rate,time,total=0;
     int *tot,*ppr;
     tot=&total;
     ppr=&pr;
     printf("\nEnter amount, rate and time");
     scanf("%d %d %d",&pr,&rate,&time);
     *tot=*ppr+(*ppr*rate*time)/100;
     printf("\nTotal amount : %d",*tot);
     getch();
}
Output :
Enter amount, rate and time: 1000 12 2
Total amount : 1240
```

In the above program, the pointers "tot" and "ppr" are used instead of variables "total" and "pr". The asterisk sign in front of these pointers, instructs the

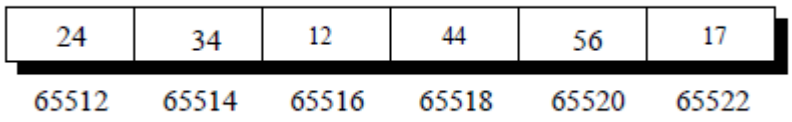
compiler that the operation is to be performed on the data saved at the location pointed by pointer variable.

4.Pointers and Arrays

We know that,

- Array elements are always stored in contiguous memory locations.
- A pointer when incremented always points to the very next location of its type.

Consider an array num[]={24,34,12,44,56,17}. The following figure is an example of how this array might be located in memory, assuming int takes 2 bytes of memory.



We can see that the array elements are stored in contiguous memory locations, where each element is occupying two bytes, because it's an integer array. Now we see the two different ways on how to access the elements of this array and print the address of each element.

```
#include<stdio.h>
#include<conio.h>
void main()
{
     int num[]={24,34,12,44,56,17};
     int i;
     for(i=0;i<=5;i++)
     {
        printf("\naddress = %u ",&num[i]);
        printf("element = %d",num[i]);
     }
     getch();
}
Output :
address = 65512 element = 24
address = 65514 element = 34
address = 65516 element = 12
address = 65518 element = 44
address = 65520 element = 56
address = 65522 element = 17
```

This method of accessing array elements and addresses is already known to us. It's just been given here for a comparison with the next method, which accesses the array elements and addresses using pointers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
     int num[]={24,34,12,44,56,17};
     int i,*j;
     j=&num[0];   /* assign address of zeroth element */
     for(i=0;i<=5;i++)
     {
        printf("\naddress = %u ",&num[i]);
        printf("element = %d",num[i]);
        j++;   /*Increment pointer to next location*/
     }
     getch();
}
Output :
address = 65512 element = 24
address = 65514 element = 34
address = 65516 element = 12
address = 65518 element = 44
address = 65520 element = 56
address = 65522 element = 17
```

In this program, we have initialized j to the address of the 0th element using the statement, "j=&num[0];". When the loop works for the first time, j contains the address 65512, and the value at this address is 24. The printf() statements print the address and the value. On incrementing j, it then points to the next memory location i.e. location no. 65514. The next time loop runs, location no. 65514 contains the second element of the array, therefore when the printf() now works print the data of second element. This way the whole array gets printed.

Now obviously, the question that arises is to which of the above two methods should be used and when? Accessing array elements by pointers is always faster and better. However, according to our convenience we should note the following:

The array elements should be accessed by using pointers if the elements are to be accessed in a fixed order, say from beginning to end, or from end to beginning, or every alternate element or any such definite logic. Instead, it would be easier to access the elements using a subscript if there is no fixed logic in accessing the elements. However, in this case also, accessing the elements by pointers would work faster.

5. Passing an Entire Array to a Function

Now let's see how to pass an entire array to a function rather than its individual elements. Consider the following example:

```
#include<stdio.h>
#include<conio.h>
void display(int *, int);
void main()
{
     int num[]={24,34,12,44,56,17};
     display(&num[0],6);
}
void display(int *j,int n)
{
     int i;
     for(i=0;i<n;i++)
     {
       printf("\nelement = %d",*j);
       j++; /* increment pointer to point to next element */
     }
}
Output :
element = 24
element = 34
element = 12
element = 44
element = 56
element = 17
```

Here we have used display() function to print the array elements. We have passed the address of the zeroth element to the display() function. The 'for' loop is used to access the array elements using pointers. Therefore, just passing the address of the zeroth element of the array to a function is same as passing the entire array to the function. It is also necessary to pass the total number of elements in the array, otherwise the function would not know when to terminate the for loop. Note that the address of the zeroth element (also called the base address) can also be passed by just passing the name of the array. Thus, the following two function calls perform the same operation:

```
display(&num[0],6);
display(num,6);
```

Now we know that on mentioning the name of the array we get its base address. Therefore, by saying *num we can refer to the zeroth element of the array, that is, 24. One can easily see that *num and *(num+0) both refer to 24. Similarly, by saying *(num+1) we can refer the first element of the array, that is, 34. In fact, this is what the C compiler does internally. When we say, num[i], the C compiler internally converts it to *(num+i). This means that all the following notations are same:

```
num[i]
*(num+i)
*(i+num)
i[num]
```

6. Pointers and 2-D Arrays

Accessing two dimensional array elements through pointer variables is similar and easier. You can think of a two dimensional array as a long linear array in which individual elements are divided into rows. You have studied that the contents of linear array can be accessed with *(x+i), where x is the base address and i is the index number. In case of two dimensional arrays, another index variable j is used.

Suppose we want to refer to the element a[2][1] using pointers. We know that s[2] would point to address of third row's first element. Obviously (s[2]+1) would give the address of third row's second element. And the value at this address can be obtained by using the value at address operator, saying *(s[2]+1). We have already studied in one-dimensional arrays that num[i] is same as *(num+i). Similarly, *(s[2]+1) is same as, *(*(s+2)+1). Thus, all the following expressions refer to the same element,

```
s[2][1]
*(s[2]+1)
*(*(s+2)+1)
```

The following program prints out each element of a two-dimensional array using pointer notations.

```
#include<stdio.h>
#include<conio.h>
void main()
{
       int ar[3][4]=
            {
            {10,20,30,40},
            {20,40,60,80},
            {30,60,90,120},
            };


      int i,j;
      for(i=0;i<3;i++)
      {
      for(j=0;j<4;j++)
      {
      printf("\nValue at [%d][%d] is %d",i,j,*(*(ar+i)+j));
      }
      }
    getch();
 }
```

```
Output :
Value at [0][0] is 10
Value at [0][1] is 20
Value at [0][2] is 30
Value at [0][3] is 40
Value at [1][0] is 20
Value at [1][1] is 40
```

```
Value at [1][2] is 60
Value at [1][3] is 80
Value at [2][0] is 30
Value at [2][1] is 60
Value at [2][2] is 90
Value at [2][3] is 120
```

The above program declares and initializes an array of size [3][4]. Then the contents of array are printed by using the pointer with formula *(*(ar+i)+j).

7. Passing 2-D Array to a Function

The following program shows how to pass a 2-D array to a function using pointers.

```
#include<stdio.h>
#include<conio.h>
void display1(int *q, int row, int col);
void display2(int (*q)[4], int row, int col);
void main()
{
      int ar[3][4]=
      {
      {10,20,30,40},
      {20,40,60,80},
      {30,60,90,120},
      };


      display1(a,3,4);
      display2(a,3,4);
      getch();
}

void display1(int *q, int row, int col)
{
     int i,j;
     for(i=0;i<row;i++)
     {
        for(j=0;j<col;j++)
        {
          printf("%d ",*(q +i*col+j));
        }
        printf("\n");
     }
     printf("\n");
}
void display2(int (*q)[4], int row, int col)
{
     int i,j;
     int *p;
     for(i=0;i<row;i++)
     {
        p=q+i;
        for(j=0;j<col;j++)
        {
```

```
                    printf("%d ",*(p+j));
                  }
                  printf("\n");
               }
               printf("\n");
          }
Output :
10 20 30 40
20 40 60 80
30 60 90 120

10 20 30 40
20 40 60 80
30 60 90 120
```

In the display1() function we have collected the base address of the 2-D array and passed to it in an ordinary int pointer q. Then through the two for loops using the expression *(q+i*col+j) we have reached the appropriate element in the array. Suppose i is equal to 2 and j is equal to 3, then we wish to reach the element a[2][3]. If the base address i.e. address of q is 65502, the expression *(q+i*col+j) becomes *(65502+2*4+3). This becomes *(65502+11). As 65502 is address of an integer, *(65502+11) evaluates to be *(65524). We can check that value at this address is 120 as each element consumes 2 bytes. This is indeed same as a[2][3].

In the display2() function we have defined variable q to be a pointer to an array of 4 integers from the declaration (*q)[4]. Initially, q holds the base address of the zeroth 1-D array, let's say 65502. This address is assigned to an int pointer p, and then using this pointer all elements of the zeroth 1-D array are accessed. Next time through the loop when i takes a value 1, the expression q+i fetches the address of the first 1-D array. This is because, q is a pointer to zeroth 1-D array and adding 1 to it would give us the address of the next 1-D array. This address is once again assigned to p, and using it all elements of the next 1-D array are accessed.

8. Arrays of pointer

Till now we have studied array of ints or an array of floats, now we will study how to handle an array of pointers. We know that a pointer variable always contains an address, so an array of pointers would be nothing but just a collection of addresses. The addresses contained in the array of pointers can be any addresses like addresses of isolated variables or addresses of array elements any other addresses. Let's see an example which will clarify the concept:

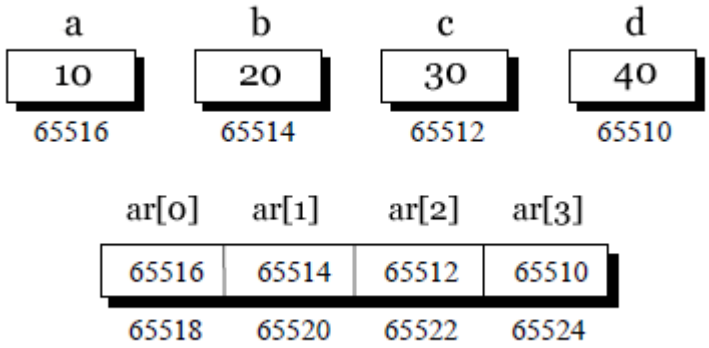
```
#include<stdio.h>
#include<conio.h>
void main()
{
     int *ar[4];
```

```
    int a=10,b=20,c=30,d=40,i;
    ar[0]=&a;
    ar[1]=&c;
    ar[2]=&b;
    ar[3]=&d;
    for(i=0;i<=3;i++)
    {
       printf("\n%d",*(ar[m]));
    }
    getch();
}
```

Output :

10
20
30
40

The following figure shows the contents and the arrangement of the 4 int variables a,b,c,d and array of pointers in memory.



As you can observe, the array 'ar' contains addresses of isolated int variables a,b,c and d. The for loop picks up the addresses present in ar and prints the values present at these addresses

9. Pointers to function

Following are the main usages of pointers in relation to functions:

- Pointers as Arguments
- Functions returning Pointers

We have already studied about pointers as arguments in the section 7.4, so it is not discussed here.

We can create functions that return pointer variables. With this method, we can return multiple values from a function, depending on some condition. The following program illustrates the functions returning data of pointer type.

```
#include<stdio.h>
#include<conio.h>
int *large(int *,int *);
void main()
{
     int a,b;
     int *ans;
     printf("\nEnter two numbers:");
     scanf("%d %d,&a,&b);
     ans=large(&a,&b);
     printf("Larger is %d",*ans);
     getch();
}
int *large(int *n1,int *n2)
{
     if(*n1>*n2)
       return(n1);
     else
       return(n2);
}
```

Output :

Enter two numbers: 13 56
Larger is 56

This program declares a function "large" that takes two parameters of integer pointers and returns an integer pointer. The main program calls the functions by using a pointer variable "ans". The "large" function returns the pointer that is having bigger value of the two and the value is printed.

10. POINTERS AND STRUCTURES

The structure variables take space in the memory of a computer when the program is executed. You can define a pointer variable of structure data type. The pointer variable to a structure variable is declared by using the same syntax that is used to define pointer variable of int or float or other built in types. The syntax for declaring a pointer to structure variable is:

```
struct structure_name*pointer_variable_name;
```

For example, the statement "struct student *ptr" declares a pointer variable, "ptr" of "student" type. The members of a structure can be accessed by using the pointer variable. The difference between accessing a structure member with normal variable and pointer variable is that, you use period operator(.) to

access the members with normal structure variable and arrow operator(->) with pointer variable of structure type to access the members. For example, "ptr->empno" will access the "empno" member of the structure. The following program illustrates the use of pointers with the help of structures.

```
#include<stdio.h>
#include<conio.h>
struct student
{
int rollno;
char name[10];
float marks;
};
void main()
{
     struct student s;
     struct student *ptr;  //pointer of structure type
     printf("\nEnter the rollno of the student : ");
     scanf("%d",&ptr->rollno);
     printf("\nEnter the name of the student : ");
     scanf("%s",&ptr->name);
     printf("\nEnter the marks of the student : ");
     scanf("%f",&ptr->marks);
     printf("\nThe name of student is : %s",ptr->name);
     printf("\nThe marks of student is : %f",ptr->marks);
     printf("\nThe rollno of student is : %d",ptr->rollno);
     getch();
}
```

Output :

Enter the rollno of the student : 8
nEnter the name of the student : Pooja
Enter the marks of the student : 78

The name of student is : Pooja
The marks of student is : 78.0000
The rollno of student is : 8

In the above program, a pointer variable "ptr" points to the variable "s" of the structure "student". The arrow operator has been used to access the member variables of structure.

Pointers are used with functions in different ways. Following are the main usages of pointers in relation to functions:

- Pointers as Arguments
- Functions returning Pointers

We have studied about pointers as arguments in pass by reference.

We can create functions that return pointer variables. With this method, we can return multiple values from a function, depending on some condition. The following program illustrates the functions returning data of pointer type.

```
#include<stdio.h>
#include<conio.h>
int *large(int *,int *);
void main()
{
     int a,b;
     int *ans;
     printf("\nEnter two numbers:");
     scanf("%d %d,&a,&b);
     ans=large(&a,&b);
     printf("Larger is %d",*ans);
     getch();
}
int *large(int *n1,int *n2)
{
     if(*n1>*n2)
       return(n1);
     else
       return(n2);
}
```

Output :
Enter two numbers: 13 56
Larger is 56

This program declares a function "large" that takes two parameters of integer pointers and returns an integer pointer. The main program calls the functions by using a pointer variable "ans". The "large" function returns the pointer that is having bigger value of the two and the value is printed.

11. Command Line Arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line argumentsThe command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.

```
#include <stdio.h>

int main( int argc, char *argv[] )
{

 if( argc == 2 )
{
       printf("The argument supplied is %s\n", argv[1]);
}
else if( argc > 2 )
{
       printf("Too many arguments supplied.\n");
}
 else
{
       printf("One argument expected.\n");
}
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2 testing3
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that argv[0] holds the name of the program itself and argv[1] is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then argc is set at 2.

12. Dynamic Memory Allocation: malloc(), calloc(), free() & realloc()

The exact size of array is unknown until the compile time,ie. time when a compiler compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes

more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr=(cast-type*)malloc(byte-size)
```

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement causes the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory
allocated
                                          using
malloc
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
```

```
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient,then, we can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, ptr is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory:
                                                ");
    for(i=0;i<n1;++i)
    printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
```

```
    for(i=0;i<n2;++i)
  printf("%u\t",ptr+i);
    return 0;
}
```