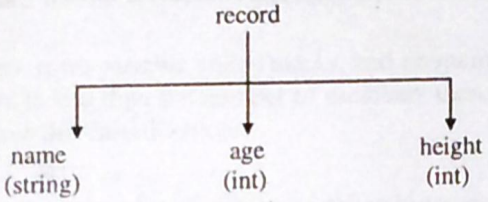
```
int *func()
{
        int *ptr;
        ptr = (int*)malloc(10*sizeof(int));
        .....
        return ptr;
}
```

Here we have allocated memory through `malloc()` in `func()`, and returned a pointer to this memory. Now the calling function receives the starting address of this memory, so it can use this memory. Note that now the call to function `free()` should be placed in the calling function when it has finished working with this memory. Here `func()` is declared as a function returning pointer. Recall that it is not valid to return address of a local variable since it vanishes after the termination of function.

## 2.4 Structure

Array is a collection of same type of elements but in many real life applications we may need to group different types of logically related data. For example if we want to create a record of a person that contains name, age and height of that person, then we can't use array because all the three data elements are of different types.




To store these related fields of different data types we can use a structure, which is capable of storing heterogeneous data. Data of different types can be grouped together under a single name using structures. The data elements of a structure are referred to as members.

### 2.4.1 Defining a Structure

Definition of a structure creates a template or format that describes the characteristics of its members. All the variables that would be declared of this structure type, will take the form of this template. The general syntax of a structure definition is-

```
struct tagname{
            datatype member1;
            datatype member2;
            .....
            .....
            datatype memberN;
           };
```

Here `struct` is a keyword, which tells the compiler that a structure is being defined. `member1`, `member2`, ....., `memberN` are members of the structure and are declared inside curly braces. There should be a semicolon at the end of the curly braces. These members can be of any data type like int, char, float, array, pointer or another structure type. `tagname` is the name of the structure and it is used further in the program to declare variables of this structure type.

Definition of a structure provides one more data type in addition to the built in data types. We can declare variables of this new data type that will have the format of the defined structure. It is important to note that definition of a structure template does not reserve any space in memory for the members; space is reserved only when actual variables of this structure type are declared. Although the syntax of declaration of members inside the template is identical to the syntax we use in declaring variables, these members are not variables, they don't

have any existence until they are attached with a structure variable. The member names inside a structure should be different but these names can be similar to any other variable name declared outside the structure. The member names of two different structures may also be same. Let us take an example of defining a structure template.

```
struct student{
                char name[20];
                int rollno;
                float marks;
              };
```

Here `student` is the structure tag and there are three members of this structure viz `name`, `rollno` and `marks`. Structure template can be defined globally or locally i.e. it can be placed before all functions in the program or it can be locally present in a function. If the template is global then it can be used by all functions while if it is local then only the function containing it can use it.

## 2.4.2 Declaring Structure Variables

By defining a structure we have only created a template or format, the actual use of structures will be when we declare variables based on this template. We can declare structure variables in two ways-

1. With structure definition
2. Using the structure tag

### 2.4.2.1 With Structure Definition

```
struct student{
                char name[20];
                int rollno;
                float marks;
              }stu1,stu2,stu3;
```

Here `stu1`, `stu2` and `stu3` are variables of type `struct student`. When we declare a variable while defining the structure template, the tagname is optional. So we can also declare them as-

```
struct{
        char name[20];
        int rollno;
        float marks;
      }stu1,stu2,stu3;
```

If we declare variables in this way, then we'll not be able to declare other variables of this structure type anywhere else in the program nor can we send these structure variables to functions. If a need arises to declare a variable of this type in the program then we'll have to write the whole template again. So although the tagname is optional it is always better to specify a tagname for the structure.

### 2.4.2.2 Using Structure Tag

We can also declare structure variables using structure tag. This can be written as-

```
struct student{
                char name[20];
                int rollno;
                float marks;
              };
struct student stu1,stu2;
struct student stu3;.
```

Here `stu1`, `stu2` and `stu3` are structure variables that are declared using the structure tag `student`. Declaring a structure variable reserves space in memory. Each structure variable declared to be of type `struct student` has three members viz. `name`, `rollno` and `marks`. The compiler will reserve space for each variable sufficient to hold all the members.

### 2.4.3 Initialization of Structure Variables

The syntax of initializing structure variables is similar to that of arrays. All the values are given in curly braces and the number, order and type of these values should be same as in the structure template definition. The initializing values can only be constant expressions.

```
struct student{
                    char name[20];
                    int rollno;
                    float marks;
                }stu1 = {"Mary",25,98};
struct student stu2 = {"John",24,67.5};
```

Here value of members of stu1 will be "Mary" for name, 25 for rollno, 98 for marks. The values of members of stu2 will be "John" for name, 24 for rollno, 67.5 for marks.

We cannot initialize members while defining the structure.

```
struct student {
                    char name[20];
                    int rollno;
                    float marks = 99;  /*Invalid*/
                }stu;
```

This is invalid because there is no variable called marks, and no memory is allocated for structure definition. If the number of initializers is less than the number of members then the remaining members are initialized with zero. For example if we have this initialization-

```
struct student stu1 = {"Mary"};
```

Here the members rollno and marks of stu1 will be initialized to zero. This is equivalent to the initialization-

```
struct student stu1 = {"Mary",0,0};
```

### 2.4.4 Accessing Members of a Structure

For accessing any member of a structure variable, we use the dot ( . ) operator which is also known as the period or membership operator. The format for accessing a structure member is-

```
structvariable.member
```

Here on the left side of the dot there should be a variable of structure type and on right hand side there should be the name of a member of that structure. For example consider the following structure-

```
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                };
struct student stu1,stu2;
```

name of stu1 is given by - `stu1.name`
rollno of stu1 is given by - `stu1.rollno`
marks of stu1 is given by - `stu1.marks`
name of stu2 is given by - `stu2.name`
rollno of stu2 is given by - `stu2.rollno`
marks of stu2 is given by - `stu2.marks`

We can use `stu1.name`, `stu1.marks`, `stu2.marks` etc like any other ordinary variables in the program. They can be read, displayed, processed, assigned values or can be sent to functions as arguments. We can't use `student.name` or `student.rollno` because `student` is not a structure variable, it is a structure tag.

```
/*P2.22 Program to display the values of structure members*/
#include<stdio.h>
#include<string.h>
```

```
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                };
main()
{
        struct student stu1 = {"Mary",25,68};
        struct student stu2,stu3;
        strcpy(stu2.name,"John");
        stu2.rollno = 26;
        stu2.marks = 98;
        printf("Enter name, rollno and marks for stu3 : ");
        scanf("%s %d %f",stu3.name,&stu3.rollno,&stu3.marks);
        printf("stu1 : %s  %d  %.2f\n",stu1.name,stu1.rollno,stu1.marks);
        printf("stu2 : %s  %d  %.2f\n",stu2.name,stu2.rollno,stu2.marks);
        printf("stu3 : %s  %d  %.2f\n",stu3.name,stu3.rollno,stu3.marks);
}
```

In this program we have declared three variables of type `struct student`. The first variable `stu1` has been initialized, the members of second variable `stu2` are given values using separate statements and the values for third variable `stu3` are input by the user.

The dot operator is one of the highest precedence operators; its associativity is from left to right. Hence it will take precedence over all other unary, relational, logical, arithmetic and assignment operators. So in an expression like `++stu.marks`, first `stu.marks` will be accessed and then its value will be increased by 1.

## 2.4.5 Assignment of Structure Variables

We can assign values of a structure variable to another structure variable, if both variables are of the same structure type. For example-

```
/*P2.23 Program to assign a structure variable to another structure variable*/
#include<stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                };
main()
{
        struct student stu1 = {"Oliver",12,98},stu2;
        stu2 = stu1;
        printf("stu1 : %s   %d   %.2f\n",stu1.name,stu1.rollno,stu1.marks);
        printf("stu2 : %s   %d   %.2f\n",stu2.name,stu2.rollno,stu2.marks);
}
```

Unary, relational, arithmetic, bitwise operators are not allowed with structure variables. We can use these variables with the members provided the member is not a structure itself.

## 2.4.6 Array of Structures

We know that array is a collection of elements of same datatype. We can declare array of structures where each element of array is of structure type. Array of structures can be declared as-

```
struct student stu[10];
```

Here `stu` is an array of 10 elements, each of which is a structure of type `struct student`, means each element of `stu` has 3 members, which are `name`, `rollno` and `marks`. These structures can be accessed through subscript notation. To access the individual members of these structures we'll use the dot operator as usual.

```
stu[0].name     stu[0].rollno     stu[0].marks
stu[1].name     stu[1].rollno     stu[1].marks
stu[2].name     stu[2].rollno     stu[2].marks
.............   ..............    ..............
.............   ..............    ..............
stu[9].name     stu[9].rollno     stu[9].marks
```

All the structures of an array are stored in consecutive memory locations.

```
/*P2.24 Program to understand array of structures*/
#include<stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    float marks;
                };
main()
{
        int i;
        struct student stuarr[10];
        for(i=0; i<10; i++)
        {
           printf("Enter name,rollno and marks : ");
           scanf("%s%d%f",stuarr[i].name,&stuarr[i].rollno,&stuarr[i].marks);
        }
        for(i=0; i<10; i++)
           printf("%s %d %f\n",stuarr[i].name,stuarr[i].rollno,stuarr[i].marks);
}
```

An array of structure can be initialized as-

```
struct student stuarr[3] = {
                                {"Mary",12,98.5},
                                {"John",11,97},
                                {"Tom",12,89.5}
                              };
```

The inner pairs of braces are optional if all the initializers are present in the list.

## 2.4.7 Arrays within Structures

We can have an array as a member of structure. In structure `student`, we have taken the member `name` as an array of characters. Now we'll declare another array inside the structure `student`.

```
struct student {
                    char name[20];
                    int rollno;
                    int submarks[4];
                  };
```

The array `submarks` denotes the marks of students in 4 subjects.

If `stu` is a variable of type `struct student` then-

`stu.submarks[0]` - Denotes the marks of the student in first subject
`stu.submarks[1]` - Denotes the marks in second subject.
`stu.name[0]` - Denotes the first character of the name member.
`stu.name[4]` - Denotes the fifth character of the name member.

If `stuarr` is an array of size 10 of type `struct student` then-

`stuarr[0].submarks[0]` - Denotes the marks of first student in first subject
`stuarr[4].submarks[3]` - Denotes the marks of fifth student in fourth subject.
`stuarr[0].name[0]` - Denotes the first character of name member of first student
`stuarr[5].name[7]` - Denotes the eighth character of name member of sixth student

```
/*P2.25 Program to understand arrays within structures*/
#include<stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    int submarks[4];
                  };
main()
{
    int i,j;
    struct student stuarr[3];

    for(i=0; i<3; i++)
    {
        printf("Enter data for student %d\n",i+1);
        printf("Enter name : ");
        scanf("%s",stuarr[i].name);
        printf("Enter roll number : ");
        scanf("%d",&stuarr[i].rollno);
        for(j=0; j<4; j++)
        {
                printf("Enter marks for subject %d : ",j+1);
                scanf("%d",&stuarr[i].submarks[j]);
        }
    }
    for(i=0; i<3; i++)
    {
        printf("Data of student %d\n",i+1);
        printf("Name:%s,Roll number:%d\nMarks:",stuarr[i].name,stuarr[i].rollno);
        for(j=0; j<4; j++)
                printf("%d   ",stuarr[i].submarks[j]);
        printf("\n");
    }
}
```

## 2.4.8 Nested Structures (Structure within Structure)

The members of a structure can be of any data type including another structure type i.e. we can include a structure within another structure. A structure variable can be a member of another structure. This is called nesting of structures.

```
struct tag1{
                member1;
                member2;
                ..........
                struct tag2{
                                member1;
                                member2;
                                ..........
                                member m;
                              }var1;

                ..........
                member n;
            }var2;
```

For accessing member1 of inner structure we will write -

```
var2.var1.member1
```

Here is an example of a nested structure -

```
struct student{
            char name[20];
            int rollno;
            struct date{
                        int day;
                        int month;
                        int year;
                        }birthdate;
            float marks;
      }stu1,stu2;
```

Here we have defined a structure `date` inside the structure `student`. This structure `date` has three members `day`, `month`, `year` and `birthdate` is a variable of type `struct date`. We can access the members of inner structure as-

`stu1.birthdate.day` → day of birthdate of stu1
`stu1.birthdate.month` → month of birthdate of stu1
`stu1.birthdate.year` → year of birthdate of stu1
`stu2.birthdate.day` → day of birthdate of stu2

Here we have defined the template of structure date inside the structure `student`, we could have defined it outside and declared its variables inside the structure student using the tag. But remember if we define the inner structure outside, then this definition should always be before the definition of outer structure. Here in this case the date structure should be defined before the student structure.

```
struct date{
            int day;
            int month;
            int year;
          };
struct  student{
              char name[20];
              int rollno;
              float marks;
              struct date birthdate;
            }stu1,stu2;
```

The advantage of defining `date` structure outside is that we can declare variables of `date` type anywhere else also. Suppose we define a structure `teacher`, then we can declare variables of `date` structure inside it as-

```
struct teacher{
              char name[20];
              int age;
              float salary;
              struct date birthdate, joindate;
            }t1,t2;
```

The nested structures may also be initialized at the time of declaration. For example-

```
struct teacher t1 = {"Sam",34,9000,{8,12,1970},{1,7,1995}};
```

Nesting of a structure within itself is not valid. For example the following structure definition is invalid-

```
struct person{
            char name[20];
            int age;
            float height;
            struct person father;  /*Invalid*/
          }emp;
```

The nesting of structures can be extended to any level. The following example shows nesting at level three i.e. first structure is nested inside a second structure and second structure is nested inside a third structure.

```
struct time
{
      int hr;
      int min;
```

```
        int sec;
};
struct date
{
        int day;
        int month;
        int year;
        struct time t;
};
struct student
{
        char name[20];
        struct date dob;        /*Date of birth*/
}stu1,stu2;
```

To access hour of date of birth of student `stu1` we can write-

```
stu1.dob.t.hr
```

## 2.4.9 Pointers to Structures

We have studied that pointer is a variable which holds the starting address of another variable of any data type like int, float or char. Similarly we can have pointer to structure, which can point to the starting address of a structure variable. These pointers are called structure pointers and can be declared as-

```
struct student{
                char name[20];
                int rollno;
                int marks;
              };
struct student stu,*ptr;
```

Here `ptr` is a pointer variable that can point to a variable of type `struct student`. We will use the & operator to access the starting address of a structure variable, so `ptr` can point to `stu` by-

```
ptr = &stu;
```

There are two ways of accessing the members of structure through the structure pointer.

We know `ptr` is a pointer to a structure, so by dereferencing it we can get the contents of structure variable. Hence `*ptr` will give the contents of `stu`. So to access members of a structure variable `stu` we can write-

```
(*ptr).name
(*ptr).rollno
(*ptr).marks
```

Here parentheses are necessary because dot operator has higher precedence than the * operator. This syntax is confusing so C has provided another facility of accessing structure members through pointers. We can use the arrow operator (->) which is formed by hyphen symbol and greater than symbol. We can access the members as-

```
ptr->name
ptr->rollno
ptr->marks
```

The arrow operator has same precedence as that of dot operator and it also associates from left to right.

```
/*P2.26 Program to understand pointers to structures*/
#include<stdio.h>
struct student{
                char name[20];
                int rollno;
                int marks;
              };
main()
{
        struct student stu = {"Mary",25,68};
```

```
        struct student *ptr = &stu;
        printf("Name - %s\t",ptr->name);
        printf("Rollno - %d\t",ptr->rollno);
        printf("Marks - %d\n",ptr->marks);
}
```

We can also have pointers that point to individual members of a structure variable. For example-

```
int *p = &stu.rollno;
float *ptr = &stu.marks;
```

The expression `&stu.rollno` is equivalent to `&(stu.rollno)` because the precedence of dot operator is more than that of address operator.

## 2.4.10 Pointers within Structures

A pointer can also be used as a member of structure. For example we can define a structure like this-

```
struct student{
                char name[20];
                int *ptrmem;
              };
struct student stu,*stuptr = &stu;
```

Here `ptrmem` is pointer to int and is a member of the structure `student`.

To access the value of `ptrmem`, we'll write-

`stu.ptrmem` or `stuptr->ptrmem`

To access the value pointed to by `stu.ptrmem`, we'll write-

`*stu.ptrmem` or `*stuptr->ptrmem`

Since the priority of dot and arrow operators is more than that of dereference operator, the expression `*stu.ptrmem` is equivalent to `*(stu.ptrmem)`, and the expression `*stuptr->ptrmem` is equivalent to `*(stuptr->ptrmem)`.

## 2.4.11 Structures and Functions

Structures may be passed as arguments to function in different ways. We can pass individual members, whole structure variable or structure pointers to the function. Similarly a function can return either a structure member or whole structure variable or a pointer to structure.

### 2.4.11.1 Passing Structure Members as Arguments

We can pass individual structure members as arguments to functions like any other ordinary variable.

```
/*P2.27 Program to understand how structure members are sent to a function*/
#include<stdio.h>
#include<string.h>
struct student{
                char name[20];
                int rollno;
                int marks;
              };
void display(char name[],int rollno,int marks);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2;
        strcpy(stu2.name,"Mary");
        stu2.rollno = 18;
        stu2.marks = 90;
```

```
        display(stu1.name,stu1.rollno,stu1.marks);
        display(stu2.name,stu2.rollno,stu2.marks);
}
void display(char name[],int rollno,int marks)
{
        printf("Name   - %s\t",name);
        printf("Rollno - %d\t",rollno);
        printf("Marks  - %d\n",marks);
}
```

Here we have passed members of the variables `stu1` and `stu2` to the function `display()`. We can pass the arguments using call by reference also so that the changes made in the called function will be reflected in the calling function. In that case we'll have to send the addresses of the members. It is also possible to return a single member from a function.

### 2.4.11.2 Passing Structure Variable as Argument

Passing individual members to function becomes cumbersome when there are many members and the relationship between the members is also lost. We can pass the whole structure as an argument.

```
/*P2.28 Program to understand how a structure variable is sent to a function*/
#include<stdio.h>
struct student {
                    char name[20];
                    int rollno;
                    int marks;
                };
void display(struct student);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2 = {"Mary",18,90};
        display(stu1);
        display(stu2);
}
void display(struct student stu)
{
        printf("Name   - %s\t", stu.name);
        printf("Rollno - %d\t", stu.rollno);
        printf("Marks  - %d\n", stu.marks);
}
```

Here it is necessary to define the structure template globally because it is used by both functions to declare variables.

The name of a structure variable is not a pointer unlike arrays, so when we send a structure variable as argument to a function, a copy of the whole structure is made inside the called function and all the work is done on that copy. Any changes made inside the called function are not visible in the calling function since we are only working on a copy of the structure variable, not on the actual structure variable.

### 2.4.11.3 Passing Pointers to Structures as Arguments

If the size of a structure is very large, then it is not efficient to pass the whole structure to the function since a copy of it has to be made inside the called function. In this case it is better to send address of the structure which will improve the execution speed.

We can access the members of the structure variable inside the calling function using arrow operator. In this case any changes made to the structure variable inside the called function, will be visible in the calling function since we are actually working on the original structure variable.

```
/*P2.29 Program to understand how a pointer to structure variable is sent to a function*/
#include<stdio.h>
struct student{
```

```
                char name[20];
                int rollno;
                int marks;
            };
void display(struct student *);
void inc_marks(struct student *);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2 = {"Mary",18,90};
        inc_marks(&stu1);
        inc_marks(&stu2);
        display(&stu1);
        display(&stu2);
}
void inc_marks(struct student *stuptr)
{
        (stuptr->marks)++;
}
void display(struct student *stuptr)
{
        printf("Name   - %s\t",stuptr->name);
        printf("Rollno - %d\t",stuptr->rollno);
        printf("Marks  - %d\n",stuptr->marks);
}
```

### 2.4.11.4 Returning a Structure Variable from Function

Structure variables can be returned from functions as any other variable. The returned value can be assigned to a structure of the appropriate type.

```
/*P2.30 Program to understand how a structure variable is returned from a function*/
#include<stdio.h>
struct student{
                char name[20];
                int rollno;
                int marks;
            };
void display(struct student);
struct student change(struct student stu);
main()
{
        struct student stu1 = {"John",12,87};
        struct student stu2 = {"Mary",18,90};
        stu1 = change(stu1);
        stu2 = change(stu2);
        display(stu1);
        display(stu2);
}
struct student change(struct student stu)
{
        stu.marks = stu.marks + 5;
        stu.rollno = stu.rollno - 10;
        return stu;
}
void display(struct student stu)
{
        printf("Name   - %s\t",stu.name);
        printf("Rollno - %d\t",stu.rollno);
        printf("Marks  - %d\n",stu.marks);
}
```

### 2.4.11.5 Returning a Pointer to Structure from a Function

Pointers to structures can also be returned from functions. In the following program, the function `func()` returns a pointer to structure.

```
/*P2.31 Program to understand how a pointer to structure is returned from a function*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct student{
                char name[20];
                int rollno;
                int marks;
              };
void display(struct student *);
struct student *func();
struct student *ptr;
main()
{
        struct student *stuptr;
        stuptr = func();
        display(stuptr);
        free(stuptr);
}
struct student *func()
{
        ptr = (struct student *)malloc(sizeof(struct student));
        strcpy(ptr->name,"Joseph");
        ptr->rollno = 15;
        ptr->marks = 98;
        return ptr;
}
void display(struct student *stuptr)
{
        printf("Name   - %s\t",stuptr->name);
        printf("Rollno  - %d\t",stuptr->rollno);
        printf("Marks  - %d\n",stuptr->marks);
}
```

### 2.4.11.6 Passing Array of Structures as Argument

We can pass an array of structure to function, where each element of array is of structure type. This can be written as-

```
/*P2.32 Program to understand how an array of structures is sent to a function*/
#include<stdio.h>
struct student{
                char name[20];
                int rollno;
                int marks;
              };
void display(struct student);
void dec_marks(struct student stuarr[]);
main()
{
       int i;
       struct student stuarr[3] = {
                                    {"Mary", 12, 98},
                                    {"John", 11, 97},
                                    {"Tom", 12, 89}
                                  };
       dec_marks(stuarr);
       for(i=0; i<3; i++)
              display(stuarr[i]);
```

```
}
void dec_marks(struct student stuarr[])
{
        int i;
        for(i=0; i<3; i++)
                stuarr[i].marks = stuarr[i].marks-10;

}
void display(struct student stu)
{
        printf("Name   - %s\t", stu.name);
        printf("Rollno  - %d\t", stu.rollno);
        printf("Marks  - %d\n", stu.marks);

}
```

All the changes made in the array of structures inside the called function will be visible in the calling function.

### 2.4.11.7 Self Referential Structures

A structure that contains pointers to structure of its own type is known as self referential structure. For example-

```
struct tag{
            datatype member1;
            datatype member2;
            ..................
            ..................
            struct tag *ptr1;
            struct tag *ptr2;
        };
```

Here `ptr1` and `ptr2` are structure pointers that can point to structure variables of type `struct tag`, so `struct tag` is a self referential structure. These types of structures are helpful in implementing data structures like linked lists and trees.

## Exercise

Find the output of the following programs.

**(1)**
```
main()
{
    int i,size=5,arr[size];
    for(i=0; i<size; i++)
            scanf("%d",&arr[i]);
    for(i=0; i<size; i++)
            printf("%d  ",arr[i]);
}
```

**(2)**
```
main()
{
    int arr[4]={2,4,8,16},i=4,j;
    while(i)
    {
            j = arr[i] + i;
            i--;
    }
    printf("j = %d\n",j);
}
```

**(3)**
```
main()
{
    int i=0,sum=0,arr[6]={4,2,6,0,5,10};
    while(arr[i])
    {
```

```
            sum = sum+arr[i];
            i++;
        }
        printf("sum = %d\n",sum);
    }
```

**(4)**

```
void func(int arr[]);
main()
{
    int arr[5] = {5,10,15,20,25};
    func(arr);
}
void func(int arr[])
{
    int i=5,sum=0;
    while(i>2)
            sum = sum+arr[--i];
    printf("sum = %d\n",sum);
}
```

**(5)**

```
void swapvar(int a,int b);
void swaparr(int arr1[5],int arr2[5]);
main()
{
    int a=4,b=6;
    int arr1[5] = {1,2,3,4,5};
    int arr2[5] = {6,7,8,9,10};
    swapvar(a,b);
    swaparr(arr1,arr2);
    printf("a = %d, b = %d\n",a,b);
    printf("arr1[0] = %d, arr1[4] = %d\n",arr1[0],arr1[4]);
    printf("arr2[0] = %d, arr2[4] = %d\n",arr2[0],arr2[4]);
}
void swapvar(int a,int b)
{
    int temp;
    temp=a, a=b, b=temp;
}
void swaparr(int arr1[5],int arr2[5])
{
    int i,temp;
    for(i=0; i<5; i++)
    { temp=arr1[i], arr1[i]=arr2[i], arr2[i]=temp; }
}
```

**(6)**

```
main()
{
    int i,arr[5]={25,30,35,40,45},*p;
    p = arr;
    for(i=0; i<5; i++)
            printf("%d\t%d\t",*(p+i),p[i]);
}
```

**(7)**

```
main()
{
    int i,arr[5]={25,30,35,40,55};
    for(i=0; i<5; i++)
    {
            printf("%d   ", *arr);
            arr++;
    }
}
```

**(8)**
```
main()
{
   int i,arr[5] = {25,30,35,40,45},*p = arr;
   for(i=0; i<5; i++)
   {
      (*p)++;
      printf("%d ",*p);
      p++;
   }
}
```

**(9)**
```
main()
{
   int arr[5]={25,30,35,40,55},*p;
   for(p=&arr[0]; p<arr+5; p++)
         printf("%d   ",*p);
}
```

**(10)**
```
main()
{
   int arr[10]={25,30,35,40,55,60,65,70,85,90},*p;
   for(p=arr+2; p<arr+8; p=p+2)
         printf("%d   ",*p);
}
```

**(11)**
```
main()
{
   int i, arr[10]={25,30,35,40,55,60,65,70,85,90};
   int *p = arr+9;
   for(i=0; i<10; i++)
         printf("%d   ",*p--);
}
```

**(12)**
```
main()
{
   int arr[10] = {25,30,35,40,55,60,65,70,85,90},*p;
   for(p=arr+9; p>=arr; p--)
         printf("%d   ",*p);
}
```

**(13)**
```
int a=5, b=10;
change1(int *p);
change2(int **pp);
main()
{
     int x=20, *ptr=&x;
     printf("%d  ",*ptr);
     change1(ptr);
     printf("%d  ",*ptr);
     change2(&ptr);
     printf("%d\n",*ptr);
}
change1(int *p)
{
     p = &a;
}
change2(int **pp)
{
     *pp = &b;
}
```

**(14)**
```
void func(int x,int *y);
```

```
main()
{
     int a=2,b=6;
     func(a,&b);
     printf("a = %d, b = %d\n",a,b);
}
void func(int x,int *y)
{
     int temp;
     temp = x;
     x = *y;
     *y = temp;
}
```

**(15)**

```
void func(int a[]);
main()
{
     int arr[10] = {1,2,3,4,5,6,7,8,9,10};
     func(arr+3);
}
void func(int a[])
{
     int i;
     for(i=0; a[i]!=8; i++)
          printf("%d ",a[i]);
}
```

**(16)**

```
main()
{
     struct A {
                 int marks;
                 char grade;
              };
     struct A A1,B1;
     A1.marks = 80;
     A1.grade = 'A';
     printf("Marks = %d\t",A1.marks);
     printf("Grade = %c\t",A1.grade);
     B1 = A1;
     printf("Marks = %d\t",B1.marks);
     printf("Grade = %c\n",B1.grade);
}
```

**(17)**

```
void func(struct tag v);
main()
{
     struct tag{
                 int i;
                 char c;
               };
     struct tag var = {2,'s'};
     func(var);
}
void func(struct tag v)
{
     printf("%d  %c\n",v.i,v.c);
}
```

**(18)**

```
struct tag{int i; char c;};
void func(struct tag);
main()
{
     struct tag var = {12,'c'};
```

```
        func(var);
        printf("%d\n",var.i);
    }
    void func(struct tag var)
    {
        var.i++;
    }
```

**(19)**

```
    struct tag{int i; char c;};
    void func(struct tag *);
    main()
    {
        struct tag var = {12,'c'};
        func(&var);
        printf("%d\n",var.i);
    }
    void func(struct tag *ptr)
    {
        ptr->i++;
    }
```