POINTERS

POINTER

Pointer is a variable which contains the address of another variable. **or**
A variable which holds address of another variable is called a pointer variable.
Example:
If x is a variable and address of x is stored in a variable p (i.e. p=&x), then the variable p is called a pointer variable.
(Note: A pointer variable should contain only the addresses)

Syntax of declaring pointer variable:

datatype *identifier;

Here datatype indicates any data type like int, char, float etc. It can be derived or user defined data type also.
Identifier is the name given to the pointer variable. It should be a valid user-defined variable name.
The asterisk(*) in between type and identifier indicates that the identifier is a pointer variable.
Example: int *p; char *x; float *z,*a,*b;

Pointer operators or **Operators used in Pointers:**

a) Address operator or Reference operator(&): It is used to get the address of the variable where it is stored in the memory.

b) Indirection operator or Dereferencing operator or Asterisk(*): It used to get the value stored in the particular address.

Accessing variables through pointers:

We use indirection operator(*) to access the values of the variables through pointers.

Example:
```
int  x=10, *p;

p=&x;

printf("%d",*p);
```

Here *p prints the value of the variable x.(i.e. 10).

Write a program to define three pointer variables for character, integer and float variables. Display the values and address of all the variables.

```
#include<stdio.h>
int main()
{
        char  c,*cptr;
        int  i,*iptr;
        float  f,*fptr;
        printf("Enter the character value\n");
        scanf("%c",&c);
        printf("Enter the integer value\n");
        scanf("%d",&i);
        printf("Enter the float value\n");
        scanf("%f",&f);
        cptr=&c;
        iptr=&i;
        fptr=&f;
        printf("character value=%c and its address=%u\n",*cptr,cptr);
        printf("integer value=%d and its address=%u\n",*iptr,iptr);
        printf("float value=%f and its address=%u\n",*fptr,fptr);
        return 0;
}
```

Arithmetic operators & Expressions using operators

```
int   a=10,b=20,*p1,*p2;
   int  sum,diff,prod,quo,rem;
p1=&a;                          p2=&b;
```

$\boxed{a=10}$ p1 $\boxed{b=20}$ p2

```
sum=*p1+*p2;              diff=*p1-*p2;               prod=(*p1)*(*p2);
     =10 + 20                  =10 - 20                    =10 * 20
     =30                       =-10                        =200

quo=*p1/*p2;              rem=*p1%*p2;
     =10/ 20                   =10 % 20
     =10.5                     =10(numerator    is    less
                          than denominator )
```

Compatibility:

Compatibility means the type of data variable and type of the pointer variable should be same or compatible.

Example: int a,*p; p=&a is compatible

int x; char *y; y=&x; it is incompatible, since type of the data variable and the type of the pointer variables are different.

Pointers and Functions:

There are two ways of passing parameters to the functions:

a) Pass by value (also called call by value).

b) Pass by reference (also called call by reference).

a)**Pass by value**: Whatever the changes done inside the function is not reflected outside the function.

If the values of the formal parameters changes in the function, the values of the actual parameters are not changed. This way of passing parameters is called pass by value or call by value.

Example:

```
#include<stdio.h>
int  main()
{
        int  x=10,y=20;
        printf("x=%d  y=%d",x,y);
        swap(x,y);
        printf("x=%d  y=%d",x,y);
        return 0;
}
```

```
void swap(int  x, int  y)
{
                int  temp;
        temp=x;
        x=y;
        y=x;
        printf("x=%d  y=%d",x,y);
}
```

After execution of this program, the first printf() function prints the value x=10,y=20, then it calls the function swap(x,y), In the function swap(x,y), the values of x and y are interchanged. So in the swap(x,y), the printf() function prints x=20 and y=10, then the control will transferred to main() function and the printf function prints the value x=10 and y=20. This indicates that, in call by value, whatever modifications done to the formal parameters, it will be reflected in that particular functions , outside the functions the value will be remain same as actual parameter.

b) Pass by reference: Whatever the changes done inside the function is reflected inside the function as well as outside the function.

If the values of the formal parameters changes in the function, the values of the actual parameters are also changed.

```
#include<stdio.h>

int  main()
{
        int x=10, y=20;
        printf("x=%d  y=%d",x,y);
        swap(&x,&y);
        printf("x=%d  y=%d",x,y);
```

```
void swap(int *x, int *y)
{
        int  temp;
        temp=*x;
        *x=*y;
        *y=*x;
        return 0;
}
```

```
        printf("x=%d  y=%d",*x,*y);
}
```

After execution of this program, the first printf() function prints the value x=10,y=20, then it calls the function swap(&x,&y), In the function swap(&x,&y), the values of x and y are interchanged. So in the swap(&x,&y), the printf() function prints x=20 and y=10, then the control will transferred to main() function and the printf function prints the value x=20 and y=10. This indicates that, in call by reference, whatever modifications done to the formal parameters, it will be reflected in that functions and also outside the functions the value will be changed.

Differences between pass by value and pass by reference:

Pass by value	**Pass by reference**
1. When a function is called the values of variables are passed.	1. When a function is called the addresses of variables are passed.
2. The type of formal parameters should be same as type of actual parameters.	2. The type of formal parameters should be same as type of actual parameters, but they have to be declared as pointers.
3. Formal parameters contains the values of actual parameters	3. Formal parameters contains the addresses of actual parameters.
4. Change of formal parameter in the function will not affect the actual parameters in the calling function.	4. The actual parameters are changed, since the formal parameters indirectly manipulate the actual parameters.
5. Execution is slower since all the values have to be copied into formal parameters.	5. Execution is faster since only Addresses are copied.
6. Example: Any programming example	6. Example: Any programming example

Initialization of Pointer variable:

Pointer Initialization is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

Example: int a=10;

int *ptr;

ptr=&a;

or int *ptr=&a;

Arrays and Pointers:

Consider the following declaration:

int a[5]={5,10,15,20,25}; This declaration informs the compiler to allocate five locations and initialize all memory locations with initial values as shown below:

a	0100	0102	0104	0106	0108
	5	10	15	20	25
	0	1	2	3	4

The starting address of the first byte of the array is called base address or starting address. Once the memory is allocated, the variable a contains 0100 which is the starting address of the 0th item. This is called base address. But, the value 0100 stored in **a** cannot be changed. So even though **a** contains an address, since its value cannot be changed. This is called pointer constant.

The array **a** is pointer constant only to the first element and not for the whole array.

So &a[0] and **a** have the same pointer value 0100 and hence they are same.

i.e. **a , &a[0] and (a+0) same.**

Note: In general

a) The address of ith item in an array is accessed using &a[i], that is same as (a+i) using pointers.

b) The ith item in an array is accessed using a[i], that is same as *(a+i) using pointers.

c) Loc(a[i])=base address + index of item*sizeof(data type)

Eg:1) Consider an array of integers with the following statement:

int a[5]={5,10,15,20,25}; Assuming base address of a=0100, the memory map along with calculation of address of a[i] is as shown below:

a+0 or &a[0]	0100=0100+0=0100+0*2
a+1 or &a[1]	0102=0100+2=0100+1*2
a+2 or &a[2]	0104=0100+4=0100+2*2
a+3 or &a[3]	0106=0100+6=0100+3*2
a+4 or &a[4]	0108=0100+8=0100+4*2

2) Consider an array of floating point numbers with the flowing statement:

```
float a[5]={1.5,2.5,3.5,4.4,5.2};
```

Assuming base address of a=0100, the memory map along with calculation of address of a[i] is as shown below:

a+0 or &a[0]	0100=0100+0=0100+0*4
a+1 or &a[1]	0104=0100+4=0100+1*4
a+2 or &a[2]	0108=0100+8=0100+2*4
a+3 or &a[3]	0112=0100+12=0100+3*4
a+4 or &a[4]	0116=0100+16=0100+4*4

Array of pointers:

As we can have array of integers or an array of float, similarly there can be an array of pointers. Since a pointer variable always contains an address, **an array of pointers would be nothing but a collection of addresses**. The address present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other address. All rules that apply to an ordinary array apply to the array of pointers also.

The example below shows the array of pointers.

```
int *ptr[5];
int a=100, b=200, c=300, d=400, e=500;
ptr[0]=&a;    *ptr[0]=100;
ptr[1]=&b;    *ptr[1]=200;
ptr[2]=&c;    *ptr[2]=300;
ptr[3]=&d;    *ptr[3]=400;
ptr[4]=&e;    *ptr[4]=500;
```

Pointers and Strings:

String is sequence of characters ends with NULL (‘\0’) character.

Suppose we have declared an array of 5 elements of data type character.

```
char str[5], *cptr;
cptr=str;
```

Here str is an array of characters, cptr is character pointer to string.

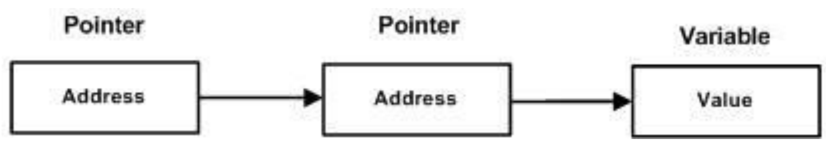
Here str gives the base addresses of the array. i.e the address of the first character in the string variable and hence can be regarded as pointer to character.

Write a program to find the length of the string using pointers:

```
#include<iostream.h>
#include<conio.h>
int main()
{
        char str[25], *ptr;
        int len=0;
        printf("Enter the string");
        gets(str);
        ptr=str;
        while(*ptr!='\0')
        {
                printf("%c",*ptr);
                ptr++;
                len++;
        }
        printf("Length=%d",len);
        return 0;
}
```

Pointers to Pointers:

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int

int **variable;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice.

Example:

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

```
int main ()
{

   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   /* take the address of var */
   ptr = &var;

   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);

   return 0;
}
```

Memory allocation functions:

There are two types of memory allocation techniques. They are

1) **Static memory allocation:** Allocation of memory during the compile time is called static memory allocation.
   Example: arrays

2) **Dynamic memory allocation:** Allocation of memory during run time is called dynamic memory allocation.
   Example: pointers, linked lists, trees etc

Advantages and disadvantages of pointers:

Advantages:

1) More than one value can be returned using pointer concept.
2) Very compact code can be written using pointers.
3) Data accessing is much faster when compared to arrays.
4) The pointers in C language are mainly useful in non-primitive data structures such as arrays, linked list and trees.

Disadvantages:

1) Un-initialized pointers or pointers containing invalid addresses can cause system crash.
2) It is very easy to use pointers incorrectly, causing bugs that are very difficult to identify and correct.
3) They are confusing and difficult to understand in the beginning and if they are misused, the result is not predictable.

STRUCTURE

STRUCTURE

Structure is collection of similar or dissimilar data type sharing a common name.
The general format of a structure definition is shown below:

```
struct   tag_name
{
          datatype1  member1;
          datatype2  memebr2;
          ……. ……….
          ……… ……..
          datatypen  memebr n;

};
```

Where

- struct is the keyword which tells the compiler that a structure is being defined.
- memeber1, member2,…are called members of the structures. They are also called fields of the structure.
- The members are declared within curly braces. The members can be any of the data types such as int, char, float etc.
- There should be semicolon at the end of closing brace.

Example:

```
struct  student
{
          int  regno;
          char  name[10];
          float  fees;
};
```

The structure definition does not reserve any space in memory for the members.

Structure declaration: Structure declaration is used to allocate the total memory required for the given the structure.

```
struct   tag_name   variable;
```

Here the variable allocates the total memory required for the structure.
Example:

```
struct   student
{
          int  regno;
          char  name[30];
          float  fees;
```

```
                    };
                    struct  student  S;
```

Here S is a variable name which allocates the total memory required for the given structure. For the above structure total memory required will be as follows

2 bytes are allocated for the field name regno,
30 bytes are allocated for the field name
and 4 bytes will be allocated to the field fees.
Total size of the structure student is 2+30+4=36 bytes.

It is possible to combine the definition of structure and declaration of the structure composition with that of the structure variable.
Example:

```
struct   student
{
        int   regno;
        char  name[30];
        float  fees;
}S;
```

Accessing the structure member or Accessing the elements of structure:

Structure member can be accessed using member operator or dot operator(.).

Syntax

```
structure_varaible_name•member_name
```

Example: The above structure members can be accessed using dot (•) operator.

```
S•regno
S•name
S•fees.
```

Program to illustrate structure definition, declaration and accessing of structure members.

```
#include<stdio.h>
int main( )
{
        struct   student
        {
                int  regno;
                char  name[20];
                float  fees;
        };
        struct  student  S;
        printf("Enter the roll number, name and fees of a student\n");
        scanf("%d%s%f",&S. regno,S.name,&S.fees);
        printf("Entered roll number, name and fees of a student is\n");
        printf("%d\t %s\t%f ",S. regno,S.name,S.fees);
}
```

Define and Declare a Structure called EMPLOYEE with the following members

1) EmpCode
2) EmpName
3) EmpSalary
4) EmpDesg

Answer:-

```
struct   Employee
{
        int      EmpCode;
        char   EmpName [30];
        long   EmpSalary;
          char   EmpDesg [30];
};
        struct   Employee  E;
```

Define and Declare a Structure called BOOK with the following members

1) Title
2) Author
3) Price
4) Pages
5) Version

Answer:-

```
struct   Book

{
        char  title[30];
        char  author[30];
        int     price;
           int     pages;
           float  version;
};
        struct   Book   B;
```

Initialization of structure:

Like any other data type it is possible for us to initialize the members of a structure when the structure is declared.

The general format of structure initialization is as shown below:

```
struct   tag_variable   name ={v1,v2,v3,…vn};
```

Here

- * struct tag_name is derived data type
- * v1, v2, v3 …vn are all the initial values. These values are called initializers, they should be separated by commas and should be enclosed between { }.

Example:

```
struct   student
{
        int   regno;
        char  name[30];
        float  fees;
}S;

struct  student   S={1234, "Nisarga",18500.50};
```

Note:

- The initial values are assigned to members of the structure on one-to-one basis. i.e. the values are assigned to various members in the order specified from the first member.
- During partial initialization (If there are fewer initial values, than the members of a structure), the values are assigned to members in the order specified and the remaining members are initialized with default values.
- During initialization, the number of initializers should not exceed the number of members. It leads to syntax error.
- During initialization, there is no way to initialize members in the middle of a structure without initializing the previous members.

Array of Structures:

As we have an array of integers, we can have an array of structures also. For example, suppose we want to store the information of ten students consisting of regno, name and fees. The structure definition can be written as shown below:

```
struct   student
{
        int   regno;
        char  name[30];
        float  fees;
};
```

To store the information of more number of students, we can have the following declaration.

```
struct   student   S[10];
```

Write a program to create a database of N students using structures.

```
#include<stdio.h>
#include<conio.h>
void main()
{
            struct   student
            {
                    int   regno;
                    char  name[30];
                    float  fees;
            };
        struct   student  S[10];
        float  f1;
        int  i, n;
        clrscr();
        printf("Enter the number of students\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("Enter the regno,name and fees of %d student\n",i+1);
            scanf("%d%s%f",&s[i].regno,s[i].name, &f1);
            s[i].fees=f1;
        }
        printf("Entered student information is\n");
        printf("REG.NO\tNAME\tFEES\n");
        for(i=0;i<n;i++)
            printf("%d\t %s\t %f\n", s[i].regno, s[i].name, s[i].fees);
        getch();
}
```

Structures and Functions:

Structure can be passed as arguments to the function in 3 different methods:

1. **To pass each member of the structure as an actual argument of the function call.**

- The actual arguments are then treated independently like ordinary variables.
- This is the more elementary method and becomes unmanageable and inefficient when the structure size is large.

2. To pass a copy of the entire structure to the called function.

- The function is working on a copy of the structure, any changes to the structure members within the function are not reflected in the original structure.

3. Pointers to pass the structure as an argument to the called function.

- Here the address location of the structure is passes to the called function. The function can access indirectly the entire structure and work on it.

To pass a copy of the entire structure to the called function.

- A structure variable can be passed to the function as an argument as normal variable.

  If structure is passed by value, changes made in structure variable in function definition does not reflect in original structure variable in calling function.

```
#include <stdio.h>
 struct student
{
        char name[50];
         int rollno;
};
 void Display(struct student s);

int main()
{
         struct student s1;
         printf("Enter student's name: ");
         scanf("%s",&s1.name);
         printf("Enter roll number:");
         scanf("%d",&s1.roll);
        Display(s1);    // passing structure variable s1 as argument
         return(0);
}
void Display(struct student stu)
{
         printf("\nName: %s",stu.name);
         printf("\nRoll: %d",stu.roll);
 }
```

Pointers to Structures

The pointer variable in association with the structure data type is very much used to construct complex databases using the data structures such as linked list, trees etc. The syntax for defining pointer to structures is given below:

```
struct Structure_name
{
        Datatype1 member1;
        Datatype2 member2;
        .........
        .........
        .........
        Datatypen member;
 } *ptr;
OR
struct Structure_name *ptr;
```

Example:

```
struct Date
{
        int day;
        int month;
        int year;
};
struct Date *ptr;
```

Here ptr is a pointer variable holding the address of the structure Date and is having three members such as day, month and year.

The pointer to structure variable can be accessed and processed in one of the following methods.

Method1:

(*ptr).member_name;

The parenthesis is essential because the structure member dot(.) has a higher precedence over the indirection operator(*).

The above structure members can be accessed as:

(*ptr).day, (*ptr).month, (*.ptr).year and so on.

Method2:

Pointer_variable->member_name;

The above structure members can be accessed as:

ptr->day, ptr->month, ptr->year and so on.

Example: Program to illustrate pointers to structures:

```
#include<stdio.h>
struct student
{
        int regno;
        char name[20];
        float marks;
};
struct student s,*ptr;

int main()
{
        ptr=&s;
        printf("Enter the register number, name and marks:\n");
        scanf("%d %s %f",&ptr->regno,ptr->name,&ptr->marks);
        printf("\n Register Number=%d",ptr->regno);
        printf("\n Name=%s",ptr->name);
        printf("\n Marks=%f",ptr->marks);
        return 0;
}
```

Self-referential structures:

- A self referential structure is a structure , which refers to the pointer to points to another structure of the same type.
- Syntax:

```
struct name
 {
        member1;
        member2;
          …….
        struct name *pointer;
 };
```

Example:

```
struct list
{
        int no;
        struct list *ptr;
};
```

UNIONS:

- Union is a collection of similar or dissimilar data elements sharing a common name and all the union members uses the same memory location to store data.
  [NOTE: The syntax of union definition, union declaration and accessing union members are same as structure. Instead of the keyword structure, keyword union is used.]

The general format of a union definition is shown below:

```
union   tag_name
{
        datatype1  member1;
        datatype2  memebr2;
        ……. ……….
        ……… ……..
        datatypen  memebr n;

};
```

Where

- union is the keyword which tells the compiler that a union is being defined.
- memeber1, member2,…are called members of the unions. They are also called fields of the unions.
- The members are declared within curly braces. The members can be any of the data types such as int, char, float etc.
- There should be semicolon at the end of closing brace.

Example:

```
union  student
{
        int  regno;
        char  name[10];
        float  fees;
};
```

The union definition does not reserve any space in memory for the members.

Union declaration: Union declaration is used to allocate the memory required for the given the union.

```
union   tag_name   variable;
```

Here the variable allocates the largest memory required for any one of the union members.

Example:

```
union   student
{
        int  regno;
        char  name[30];
```

```
            float  fees;
    };
             union  student  S;
```

Differences between Structures and Unions:

Sl No	Structures	Unions
1	The keyword struct is used to define a structure.	The keyword union is used to define a union.
2	When a variable is associated with a structure, the compiler allocates the memory for each. The size of structure is equal to the sum of sizes of its members.	When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member.
3	Each member within a structure is assigned unique storage area.	Memory allocated is shared by individual members of union.
4	All members can be accessed at a time.	Only one member can be accessed at a time.
5	All the members of a structure can be initialized at once.	Only the first member of a union can be initialized.
6	Example	Example

Uses of Unions:

- Unions are particularly useful in Embedded programming or in situations where direct access to the hardware/memory is needed.
- It is used in bit modifier (to access directly a single bit in the register/memory address).
- Low level system programming.
- It is useful when we want to model structs defined by hardware, devices or network protocols, or when we are creating a large number of objects and want to save space.

Bitfields:

- It is a structure member whose width is given in a specific number of bits.
- A bit field is a data structure used in programming. It consists of a number of adjacent computer memory locations which have been allocated to hold a sequence of bits, stored so that any single group of bits within the set can be addressed.
- The name and size of bit fields are defined using a structure.

```
struct name
 {
        Datatype1 member1:bit-length;
        Datatype2 member2:bit-length;
        Datatype3 member13:bit-length;
          ……
        Datatype n member n:bit-length;
};
```

Here datatype is either int or unsigned int and the bit length is the number of bits used for the specified name. (signed bit field should have atleast 2 bits(one bit for sign).

```
#include <stdio.h>
// A space optimized representation of date
struct date
{
  // d has value between 1 and 31, so 5 bits   are sufficient
  unsigned int d: 5;
  // m has value between 1 and 12, so 4 bits  are sufficient
  unsigned int m: 4;
  unsigned int y;
};

int main()
{
  printf("Size of date is %d bytes\n", sizeof(struct date));
  struct date dt = {31, 12, 2014};
  printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
  return 0;
}
```

Note:

1. The first field always starts with the first bit of the word.

2. We cannot take the address of the bit field variable. That is , we cannot use scanf to read values into bit fields. We can neither use pointer to access the bit fields.
3. Bit fields cannot be arrayed.
4. Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behavior would be unpredicted.
5. There can be unused bits in a word.
6. A bit field cannot overlap integer boundaries. That is the sum of lengths of all the fields in a structure should not be more than the size of the word. In case , it is more, the overlapping field is automatically forced to the beginning of the next word.

(1) What will be output if you will compile and execute the following c code?

```
struct marks{
  int p:3;
  int c:3;
  int m:2;
};
void main(){
  struct marks s={2,-6,5};
  printf("%d %d %d",s.p,s.c,s.m);
}
```

(a) 2 -6 5
(b) 2 -6 1
(c) 2 2 1
(d) Compiler error
(e) None of these

Answer: (c)

Explanation:
Binary value of 2: 00000010 (Select three two bit)
Binary value of 6: 00000110
Binary value of -6: 11111001+1=11111010
(Select last three bit)
Binary value of 5: 00000101 (Select last two bit)

Typedef:

- **typedef** is a keyword used in C language to assign alternative names to existing types.
- It is mostly used with user defined data types, when names of data types get slightly complicated.
- Following is the general syntax for using typedef.

```
        typedef existing_name  alternate_name
```

Example:

```
    typedef unsigned long ulong;
    ulong i,j,k;
```

UNION:

Union is a collection of similar or dissimilar data items sharing a common name and all the union members uses the same memory location to store data.

NOTE: The syntax of union definition, union declaration and accessing union members are same as structure. Instead of the keyword structure, keyword union is used.

The general format of a union definition is shown below:

```
        union   tag_name
```

```
    {
            datatype1   member1;
            datatype2   memebr2;
            ……. ……….
            ……… ……..
    };
   union   tag_name   variable;
```

Example:

```
        union   student
        {
                int  regno;
                char  name[30];
                float  fees;
        };
      union  student  S;
```

Differences between structures and unions:

Structures	**Unions**
1. The keyword struct is used to define a structure.	1. The keyword union is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each. The size of structure is equal to the sum of sizes of its members.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member.
3. Each member within a structure is assigned unique storage area.	3. Memory allocated is shared by individual members of union.

4. Altering the value of a member will not affect other members of the structure.	4. Altering the value any of the member will alter other member values.
5. Individual members can be accessed at a time.	5. Only one member can be accessed at a time.
6. Several members of a structure can be initialized at once.	6. Only the first member of a union can be initialized.

7. Example:
`struct student`
`{`
`int regno;`
`char name[30];`
`float fees;`
`};`
`struct student S;`
	7. Example:
	`union student`
	`{`
	`int regno;`
	`char name[30];`
	`float fees;`
	`};`
	`union student S;`

8. Structure member can be retrieved at any time.	8. only the last value can be retrieved

C Pre-Processor

C Preprocessor

- The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform the program before actual compilation.
- It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.
- The C preprocessor provides separate facilities that we can use as it needs:

1) **Inclusion of header files:** These are files of declarations that can be substituted into our program.
2) **Macro expansion:** We can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
3) **Conditional compilation:** Using special preprocessing directives, we can include or exclude parts of the program according to various conditions.

Header Files

- A header file is a file containing C declarations and macro definitions to be shared between several source files. We request the use of a header file in our program with the C preprocessing directive `#include'.
- **Uses of Header Files**
- System header files declare the interfaces to parts of the operating system. We include them in our program to supply the definitions and declarations we need to invoke system calls and libraries.
- Our own header files contain declarations for interfaces between the source files of our program. Each time we have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.
- The usual convention is to give header files names that end with `.h'.

#include

- In the C Programming Language, the #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found.
- Include directives are typically used to include the C header files for C functions that are held outside of the current source file.
- The syntax for the #include directive in the C language is:

#include <*header_file*>

OR

#include "*header_file*“

Here header_file is the name of the header file that we wish to include. A header file is a C file that typically ends in ".h" and contains declarations and macro definitions which can be shared between several source files.

- The difference between these two syntaxes is:
- If a header file is included within <>, the preprocessor will search a predetermined directory path to locate the header file.
- If the header file is enclosed in “ ", the preprocessor will look for the header file in the same directory as the source file.

- Example:

- #include<stdio.h>
- #include<conio.h>
- #include “myfile.h”

#define

- In the C Programming Language, the #define directive allows the definition of macros within our source code. These macro definitions allow constant values to be declared for use throughout our code.
- Macro definitions are not variables and cannot be changed by our program code like variables. We generally use this syntax when creating constants that represent numbers, strings or expressions.
- **Syntax:**

#define CNAME value

OR

#define CNAME (expression)

- Here CNAME is the name of the constant.
- Most C programmers define their constant names in uppercase, but it is not a requirement of the C Language.
- Value is the value of the constant.
- Expression is the expression whose value is assigned to the constant. The *expression* must be enclosed in parentheses if it contains operators.

Note:

- Do NOT put a semicolon character at the end of #define statements. This is a common mistake.

Example

```
#include<stdio.h>

#define PI 3.142
int main()
{
      float r,area,circum;

      printf("\n Enter the radius:");
      scanf("%f",&r);
      area=PI*r*r;
      circum=2*PI*r;
      printf("\n Area of Circle=%f",area);
      printf("\n Circumference of circle=%f",circum);
      return 0;
}
```

#undef

- The #undef directive undefines a constant or preprocessor macro defined previously using #define.
- Syntax:

  #undef *token*

Example:

```
#define E 2.71828
    int e_square = E * E;
#undef E
```

Conditional Compilation(Inclusion) Directives

- Conditional compilation as the name implies code is compiled if certain condition(s) hold true.
- Conditional Compilation Directives allow us to include certain portion of the code depending upon the output of constant expression.
- The preprocessor cannot use variables from the C program in expressions , it can only act on preprocessor macros.
- The preprocessor is run on the text, before any compilation is done.
- The preprocessor does not evaluate c variables. It "preprocesses" the source code before it is compiled and thus has its own language.

Types Conditional Compilation Directives

- #if
- #ifdef
- #ifndef
- #else
- #elif
- #endif

Example

```
#include <stdio.h>
int main()
 {
    #define COMPUTER "An amazing device“
    #ifdef COMPUTER printf(COMPUTER);
    #endif
   return o;
}
```

```
#include <stdio.h>
 #define x 10
int main()
 {
     #ifdef x
              printf("hello\n"); // this is compiled as x is defined
     #else
               printf("bye\n");                  // this is not compiled
      #endif
      return(0);
}
```