Chapter X
Pointers

1. Introduction

A Pointer is a variable that represents the location of a data item, either a variable or an array element. Pointers are frequently used in as they have a number of useful applications and partly because they usually lead to a more compact and efficient code than can be obtained in other ways. Suppose **'v'** is a variable that represents some particular data item. The data item can be accessed if we Know the location of the memory cell. The address of v's memory location can be determined by the expression **&v**, where & (ampersand) is a unary operator, called the address operator, that evaluates the address of its operand.

Note: The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

Now if we assign the address of **v** to another variable, **pv**

```
pv = &v;
```

Example :

Variable name --->	pv	v
Contents --->	1005	5
Address --->	2005	1005

The new variable is called a pointer to v since it points to the location where v is stored in memory. However, please note that p represent v 's address , not its value . Thus, is referred to as a pointer variable.

The data item represented by v can be accessed by the expression ***pv** where * is a unary operator called the indirection operator that operates only on a pointer variable. Therefore, ***pv** and **v** both refer to the same data item

Example:

```
main ( )
{
  int  u = 8, v ;
  int   *pu ;        / * pointer to an integer * /
  int   *pv ;        / * pointer to an integer */
  pu  = &u ;         / * assign address of u to pu */
  v   = *pu ;        / * assign value of u to v * /
  pv  = &v ;         / * assign address of u to pv */
  printf ("u = % d, &u = %u, pu =  % d,  * pu  % d \ n '' ,  u ,  &u, pu,  * pu ) ;
  printf ("v = % d, &v =  %u,  pv  =  % d,  * pv  % d \ n " ,  v ,  &v,  pv, * pv ) ;
}
```

Executing this program results in the following output:

```
u = 8, 8u = 7000, p u = 7000, *pu = 8
v = 8, 8v = 2500, p v = 2500, *pu = 8
```

The indirection operator (*) can only act upon operands that are pointers.

2. Pointer Declaration

Pointer variables, like all other variables, must be declared before they may be used in a c program. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the object of the pointer i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus a pointer declaration may be written in general terms as

```
Data-type    *paver;
```

Where paver is the name of the pointer variable, and data-type refers to the data type of the pointer's object. It should be noted that a pointer is constrained to point to a particular kind of object; every pointer points to a specific data type.

3. Pointers and Function Arguments

Pointers are often passed to a function as arguments. This allows data items within the calling pointer of the program to be accessed by the function, altered within the function and then returned to the calling portion of the program in altered from. This concept of use of pointers as passing arguments by reference in contrast to passing arguments by value.

When an argument is passed by value, the data item is copied to the function. Thus, any alteration made to he data item within the function is not carried over into the calling routine. When an argument is passed by reference, however the address of the data item is passed to the function. The contents of that address can be accessed freely either within the function or within then calling routine. Moreover, any change that is made to the data item will be recognized in both the function and the calling routine. Thus, the use of pointer as a function argument permits the corresponding data item to be altered globally from within the function,

Example:

```
/* program to swap values using pointers */
# include <studio. h>
void swap (int *a, int *b);

main ( )
   {
     int  x =  5;
     int  y =  7;
     swap (&x, &y);
     printf (“%d %d\n”,  x,  y);
   }
   void swap (int *a, int *b)
   {
        int  t ;
        t   = * a ;
       *a   = *b ;
       *b   = t ;
    }
```

Example:

```
    # include <studio. h>

    void func1 (int ,  int );
    void func2 (int  *,  int   *);
main ( )
{
       int  u  = 1;
       int  v  = 3;

       printf (‘’ u  =  % d,  v  = %d\n’’,  u,  v);
       func1 (u,  v);
       printf (‘’ u  =  % d,  v  = %d\n’’,  u,  v);
       func2 (&u,  &v);
       printf (‘’ u  =  % d,   v  = %d\n’’,  u,  v);
```

```
 }

void func1 (int   u, int   v )
{
              u   =  0;
              v   =   0;
              printf (“ u   =  % d,  v  =  %d\n’’,  u,  v);
}
void func2 (int   *pu,  int  *pv )
{
             *pu  =  0;
             *pv  =  0;
             printf (“*pu  =  % d,  *p v  = %d\n”,  *p u,  *p v);
}
```

Output

```
    u   =   1              v     =   3
    u   =   0              v     =   0
    u   =   1              v     =   3
    *pu = 0               *pv  =   0
     u   =  0              v     =   0
```

A function can also return a pointer to the calling pointer of the program. To do so, the functions definition and any corresponding function declarations must indicate that the function will return a pointer. This is accomplished by preceding the function name with an asterisk. The asterisk must appear in both the function definition and the function declaration.

Example:

```
  /* function to reverse a string */

  char   *rev (char *  ptr )
  {
              char  *ptr1,   *ptr2,    tmp;
              ptr1 =   ptr2 =  ptr;
              while( *ptr2)
                    ++ptr2;
                ptr2;
               while( ptr1< ptr2)
                {
                         tmp=*ptr1;
                         *ptr1=*ptr2;
                         *ptr2=tmp;
                }
                return  ptr;
  }
main( )
{
            char  a[100],   *p;
            gets (a);
             p =  rev(a);
             puts(p);
  }
```

An array name is actually a pointer to the array, i.e., the array name represent the address of the first element in the array. Therefore, an array name is treated as a pointer when it is passed to a function. However, it is not necessary to precede the

array name with an ampersand within the function call. Ampersands are not require with array names, since array names themselves represent addresses.
Note: it is possible to pass a pointer of an array rather than an entire array to a function. To do so, the address of the first array element to be passed must be specified as a argument. The remainder of the array, starting with the specified array element will then be passed to the function.

4. Operations on pointers

i) A pointer variable can be assigned the address of an ordinary variable.

Example: pv = &v

ii) A pointer variable can be assigned the value of another pointer variable provided both pointers point to objects of the same data type.

iii) A pointer variable can be assigned a NULL (zero) value (as a special case).

iv) An integer quantity can be added or subtracted from a pointer variable.

v) One pointer variable can be subtracted from another provided both pointers point to elements of the same array.

Example:

```
int  mystr(char   *s )
{
     char  *ss  =  s;
     int   i;
     for ( ; *s  !=  '\0';  s++);
     return  s  -  ss;
}
```

vi) Two pointer variables can be compared provided both pointers point to objects of the same data type.

Other arithmetic operations on pointers are not allowed. Thus, a pointer variable cannot be multiplied by a constant, two pointer variables cannot be added and so on.

5. Pointers and single-dimensional Arrays

We have noted that an array name is really a pointer to the first element in that array.

Therefore, if n is a 1-d array then the address of the first array element can be expressed as either &n[0] or simply as n. Moreover, the address of the second array element can be written as either &n[1] or as (n+1) and so on. In general, the address of the (i + 1) array element can be expressed as either &n[i] or as (n+i). The value of i is sometimes referred to as an offset when used in this manner.

Since &n[1] and (n+1) both represent the address of the second element of array n, it follows that n[1] and *(n+1) both represent the contents of that address i.e. the value of the second element of n.

Note: since an array name is actually a pointer to the first element within the array, therefore we can define the array as a pointer variable rather than as a conventional array.

Numerical array element cannot be assigned initial values if the array is defined as a pointer variable. However. a character type pointer variable can be assigned an entire string as a part of the variable declaration. Thus, a string can be represented by either a 1-d character array or by a character pointer.

Note : consider the following declarations :

```
char   *p, str [20];
```

Here p is a pointer and str is an array name.

Example:

```
/* to print all the characters in an array */

#include <stdio.h >

main ()
{
          char   str [20],   *a;
          for(a = str ;  *a  ! = '\0' ; a++)
          putchar(*a);
}
```

Note : Since a pointer can point to any of the data type, the increment of a++ will automatically be according to the data type i.e. in case of int it well be 2 bytes.

Example :

```
/*function to find string length */

int mystylen(char  *s)
 {
        int   i;
        for (i = 0; *s! = '\0'; i++, s++)
        return i;
}
```

Example:

```
/*function to reverse a string using pointers and recursion */ void rev(char  *s)

rev(char *s)
        {
         if(*s)
         rev(s+1);
         putchar(*s);
     }
```

Chapter X

Advanced pointer Topics

1. Multi-Dimensional Arrays

These can be declared as follows:

```
int a [2] [4];
```

Declares an array having 2 rows and 4 columns
The rows reserved are 0 & 1 and columns are 0, 1, 2 & 3.

Example:

```
int m a t [2] [4] = {{1, 2, 3, 8}, {0, 1, 5, 6}};
char weekday [7] [10] = {“Sunday”, “Monday”, …,“Saturday”};

puts(weekday[3]);
```

will print the string “Wednesday”.

Note:

```
&a[0]   === >  a

&a[0] [0]   === >  a

&a[3] [3]  === >  a[3]
```

Example :

```
/*program to find the day of year */

#include <stdio.h>

int doy (int  d, int m, int y);

main ( )
{
          int d, m, y,  dd;
          printf(“date (dd mm yyyy): “);
          scanf(“% d % d %d”, &d,  &m,  &y);
          dd =  doy (d , m , y);
          if(dd  <  0)
             printf (“error “);
           else
                printf(“the day of year is % d\n”, dd);

 }

int doy (int  d, int  m , int  y)

{
     int dt[2] [13]  = {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
                        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30,31, 30, 31, }};
   int i , leap;
   if(m  <   1  ||   m  >  12)
            d   = -1;
  leap  =  (((y  % 4)  ==  0  &&  (y  %  100)  ! = 0) || (y %  400)  == 0);
   if(dt[leap] [m] < d  ||  d <  1)
            d = -2;
```

```
    if(d  >  0)
            for(i =  1;i < m, i+++)
                d += dt [leap][i];
     return d;
}
```

2. Pointer arrays

Example:

```
#include <stdio.h>
main ()
{
     int    i;
     char line [10][80];
     char  *p[10];
     for(i  =  0;  i  < 10;  i++)
     {
          printf ("string: ");
          gets(line[i]);
          p[i] =line[i];
     }
     for(i =0; i < 10; i++)
          printf("%s\n", p[i]);
 }
```

pointer array can be initialized as:

char *lines[5] = {"xyz", "pq", "abcd", "abc", "abex"};

The storage in pointer array is of addresses and not the actual string

Example:

```
/* to sort strings using a function */

#include <staio.h>
#include <staing.h>

viod  sSort (char *string[ ], int num);

main()
{
    char *lines[5] = {"abcd", "abc", "xy", "abd", "xyz"};
    int    i;
    sSort(lines, 5);
    for(i = 0; i < 5, i++)
        printf("%s\n", lines[i]);
 }
 void sSort(char *string[ ], int num)
 {
     int  i,  j;
     char  *tmp;
     for(i = 0; i < num - 1; i++)
        for(j= i + 1; j < num; j++)
```

```
                    if((strcmp(strings[i], string[j])) > 0)
                    {
                        tmp     = string[i];
                        string[i]  = string[j];
                        string[j]  = tmp,
                     }
        }
```

Example:

```
      /*program to return the month name */
      #include <stdio.h>
      char  *fn(int)
      main()
      {
          int  n;
          printf("month number:");
          scanf("%d", &n);
          puts(fn(n));
      }
      char *fn(int i)
      {
         char   *mn[ ] = {"Error", "Jun", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", "Nov", "Dec"
         return (i > 12 || i < 1) ? mn[0] : mn[i]
      }
```

3. Pointers Vs. Multi-Dimensional Array

To define a Multidimensional Array, We write

```
    char wDays[7][10]  =  {"Sunday",
                    "Monday",
                      -
                      -
                      -
                   "Saturday"};
```

and a pointer array can be defined as

```
    char  *ptr[7]  =  =  {"Sunday",
                    "Monday",
                      -
                      -
                      -
                   "Saturday"};
```

Storage in multidimensional array is always lines, in a pointer array address of the string is stored. Storage is more in multidimensional array (as there is wastage) but not there in pointer array.
In pointer array overhead is of addresses (each occupying 2 bytes) i.e. space for storage of pointers. In speed pointer arrays are much faster as compared to multidimensional Arrays.

3.Pointer to pointer

Let us take an array of pointers

```
 char *lines [5];
```

Now this array will also have a certain base address. It can be referred to as

&lines[0]   OR lines

- if this base address is stored in a pointer it will be referred to as a pointer to a pointer of type of char .if is defined as follows:

```
      char **pp;
              pp=lines;
        pp                        Lines

   2000  -------->         1000  --->     a  b  c  d  \0

   3000                      1005 --->      x  y  z  d  \0
```

The number of stars that must be attached to the variable to reference the value to which it is pointing to, is called the level of indirection of the pointer variable.

Example:

```
char  ***ptr3,**ptr2,*ptr1,char1='A';
ptr1  = &charl,;
ptr2  = &ptr1;
ptr3  = &ptr2;
```

```
   char1            ptr1                 ptr2                 ptr3

   'A'<------         4343   <------   1990   <-------  2727

   4343              1990              2727               9944
```

Example :

```
    #include <stdio.h>
    main()
    {
        char *lines[5] = {"xyz", "pqr", "abc", "abc", "abc"},;
        char  **pp;
        pp  = lines;
        printf("%c", **pp);
    }
```

output is : x

Example :

```
        variable     address        value

           i              10001           5

           p              20002         10001

           pp             30003         20002
```

```
    #include <stdio.h>
    main()
    {
        int i,  *p,  **pp;
        i  =  5;
        p  =  &i;
       pp =  &p;
       pringf("%d\n", i);
       pringf("%d\n", p);
```

```
     pringf("%d\n", *p);
     pringf("%d\n", pp);
     pringf("%d\n", *pp);
     pringf("%d\n", **pp);
     pringf("%d\n", &pp);
}

output  is :
     5
     10001
     5
     20002
     10001
     5
     30003
```

Example :

```
#include <stdio.h>
 viod  sSort (char **,int);
 main( )
 {
     int i;
     char *str[] = {"ars", "sfgfgr", "dsfg","dewadf", "aGHn",};
     char **p;
     p = str;
     sSort(p, 5);
     for(i = 0; i < 5; ++i)
          puts(str[i]);
  }

  viod  sSort (char **,int);
  {
      int  i,  j;
      char  *s,
        for(i = 0; i < n - 1; ++i)
        {
           for(j = i + 1; j < n; ++j)
           {
              if((strcmp(*(p + i), *(p + j))) > 0)
              {
                     s      =  *(p + i);
                 *(p + i)  =  *(p + j);
                 *(p + j);  =  s;
               }
            }
        }
  }
```

This program sorts an array of strings using a pointer to a pointer.

4. Pointer to function

- ✓ The address of a function can be stored in a pointer variable.
- ✓ In this case the C compiler needs to know not only that the pointer variable pointer to a function , but also what type of value is returned by the function.
- ✓ To declare a variable pfn to be of the type "pointer to a function that returns an in integer", the declaration the declaration to be given is as follows:

```
int  (*pfn)();
```

If testfn is a function that returns an int, then the statement

```
pfn  =  testfn;
```

stores a pointer to a function.
We can call the function indirectly by using the pointer to the function.
Example:

```
#include <stdio.h>
int testfn(void);
main()
{
    int  i;
    int (*pfn)(void);
    pfn  = testfn;
    i    = (*pfn)();
    printf("%d\n",i);
  }
  int testfn(void)
  {
     puts("hello");
     return 1;
  }
```

Output: hello

```
1
```

Chapter XI

Structures and Unions

A structure is a collection of one or more variable possibly of different type types, grouped together under a single name for convenient. handling Structures help to organize Complicated data, particularly in large programs because they permit a group of separate entities.

In general, a structure may be declared as

```
struct tag
{
    member1;
    member2;
    .
    .
    .
 };
```

In this declaration, strut is a required keyword, tag is a name that identifies structures of this type and member1, member2, ... are individual member declarations. A structure member a tag and an ordinary variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures.

The individual member can be ordinary variables, ponies, arrays, or other structure. A storage class cannot be assigned to an individual member, and individual members can't be initialized within a structure-type declaration.

Once the composition of the structure has been defined, individual structure type variable can be declared as follows:

```
storage-class struck tag variable 1, variable 2,.......;
```

Where storage-class is an optional storage-class specified, strict is a required keyword, tag is the name that appeared in the structure type declaration, and variable 1, variable 2,.... are structure variables of type tag.

It is possible to combine the declaration of the structure composition with that of the structure variables

```
storage-class struct tag {
    member 1;
    member 2;
    ------
    ------
    member v;
  } variable 1, variable 2,.....;
```

The tag is optional in this situation.

The member of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in brackets and separated by commas. The general form is

```
struct Date
{
    int   dd;
    int   mm;
   int   yy;
 };

 struct Date d1, d2;
```

The tag will make a template of the fields. The tag is useful as you do not have to repeat the fields again and again. If no tag were present in the definition, every time the definition has to be repeated.

storage-class struct tag variable = {value 1, value 2, ... value n };

The members of a structure are usually processed individually, as separate entities.

A structure member can be accessed by writing

```
          variable. member
```

Where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure.

If a structure member is itself a structure, then a member of the embedded structure can be accessed by writhing.

```
       variable.member1. member2
```

where member1 refers to the name of the member within the outer structure, and member2 refers to the name of the member within the embedded structure. Similarly, if a structure member is an array then an individual array element can be accessed by writing

```
       variable. member [expression]
```

1. Structures and Pointer

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if var represents a structure-type variable, then & var represents the starting address of that variable. Moreover, Ae can declare a pointer variable for a structure by written

```
    type  *ptv;
```

Where type is a data-type that identifies the composition of the structure, ptv represents the name of the pointer variable. We can then assign the beginning address of a structure variable to this pointer by writing

```
    ptv = &var
```

Example

```
    typedef struct
    {

        int   accNum;
        char  accType;
        char  name[100];
        float  sal;
     }  Account;
```

```
Account customer,  *pc;
pc = &customer;
```

The variable and pointer declarations can be combined as

```
struct
{
   member1;
   member2;
   ......
} var, *ptv;
```

Where var again represents a structure type variable and ptv represents the name of a pointer variable.

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

```
Ptv->member
```

Where ptv refers to a structure type pointer variable and the operator->is compatible to the period (.) operator. thus, the expression

```
Ptv->member
```

is equivalent to writing

```
var. member or (*ptv).member
```

Where var is a structure type variable. The operator-> falls into the highest precedence group. Its associatively is left-to-right.

2. Passing structures to a Function

There are several different ways to pass structure type information to or from a function. structure member can be transferred individually, or entire structures can be transferred.

Individual structure member and can be passed to a function as arguments in the function call, and a single structure member can be returned via the return statement. To do so, each structure member is treated in the same way as an ordinary single-valued variable.

A complete structure can be transferred to a function by passing a structure type pointer as an argument. A structure passed in this manner will be passed by reference rather than by value. Hence if any of the structure members are altered within the function, the alterations will be recognized outside the function.

Where a structure is passed directly to a function, the transfer is by value rather than by reference. Therefore, the of the structure member are altered within the function, the alteration will not be recognized outside of the function. However, if the altered structure is returned to the calling portion of the program, then the changes will be recognized within this broader scope.

Example :

i) struct Date

```
{
     int dd;
     int mm;
     int yy;
}  d1,    d2,
```

To assign value 21 to dd of variable d1, we write

```
d1.dd = 21
```

ii)

```
struct Employee
{
     char name[50];
     char deptcode;
     float bppay;
};
main()
{
     struct Employee abc = {"xyz", 'A', 500.00};
}
```

iii)

```
#include <stdio.h>
main()
{
    struct Date
    {
            int dd;
            int mm;
            int yy;
     };
      struct Date today;
      today.dd = 20;
      today.mm = 8;
      today.yy = 1991;
      printf(Today's date is %d/%d/d\n", today.dd, today.mm, today. yy %100);
}
```

Note : Structure which is to be referred by all the functions of a c program, should be declared as external i.e. it should be declared before the main().

iv)

```
#include <stdio.h>
struct Employee
{
     char name[50];
     char deptcode;
     float bpay;
};

main()
{
     struct Employee abc;

     gets(abc.name);
     abc.deptCode = 'A';
     scanf("%f", &abc. bpay);
     printf("Name = %s dept code = %c Basic Pay = %f", abc.name, abc.deptCode,       abc.bpay);
}
```

Note : A structure as a whole can be assigned to another structure.

v) /*program to interchange two dates using structures */

```
#include <stdio.h>

struct Date
{
     int dd;
     int mm;
     int yy;
};

main()
{
     struct   Date d1, d2, t;
     printf("1st date:");
     scanf("%d%d", &d1.dd, &d1.mm, &d1.yy);
     printf("2nd date:");
     scanf("%d%d", &d2.dd, &d2.mm, &d2.yy);
     t  = d1;
     d1 = d2;
     d2 = t;
     printf("%d/%d/%d\n", d1.dd, d1.mm, d1.yy);
     printf("%d/%d/%d\n", d2.dd, d2.mm, d2.yy);
}
```

vi)   /*nested structure definition*/

```
struct Date
{
       int day;
       int month;
       int year;
 }
 struct Employee
 {
         char name[50];
         char address[50]
         struct data dob;
         struct data doj;
  };
  struct Employee abc;
```

vii)   /*function which returns a type structure*/

```
#include <stdio.h>
struct date makeDate(void);
struct Date
{
     int dd;
     int mm;
     int yy;
};
main()
{
    struct Date d1, d2, t;

    d1 = makeDate();
    d2 = makeDate();
    t  = d1;
```

```
            d1 = d2;
            d2 = t;
            printf("%d/%d/%d\n", d1.dd, d1.mm, d1.yy);
            printf("%d/%d/%d\n", d2.dd, d2.mm, d2.yy);
        }
        struct date makeDate(void);
        {
            struct Date d;

            printf("Date:");
            scanf("%d %d %d", &d.dd, &d.mm, &d.yy);
            return d;
        }
```

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

void swapDates(struct Date *d1, struct Date *d2);

struct Date
{
    int dd;
    int mm;
    int yy;
};

main()
{
   struct   Date d1, d2;

   printf("date:");
   scanf("%d %d %d", &d1.dd, &d1.mm, &d1.yy);
   printf("Next date:");
   scanf("%d %d %d", &d2.dd, &d2.mm, &d2.yy);
   swapDates(&d1, &d2);
   printf("%d/%d/%d\n", d1.dd, d1.mm, d1.yy);
   printf("%d/%d/%d\n", d2.dd, d2.mm, d2.yy);
 }
 void swapDates(struct date *d1,struct Date *d2
 {
    struct Date t;
      t   = *d1;
      *d1 = *d2;
      *d2 = t;
 {
```

ix)
```
struct Employee
{
    char name[50];
    char deptCode;
    float bpay;
};
struct Employee e[100];
```

New 100 element of type strut employee have been declared and to reference a particular structure element we use a subscript, as

```
e[1].bpay
e[1].deptcode
e[1].name
```

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

struct studen
{
    char name[30];
    int rollNum;
    int m[5]
 };
 main()
 {
    struct student s[1000];
    int i;

    for (i = 0; i < 100; i++)
    {
        prinf("name, rollno & 5 marks:\n");
        scanf("%s%d%d%d%d%d%d", s[i].name,&s[i].roll no, &s[i].m[0] &s[i].m[1],
            &s[i].m[2], &s[i].m[3], &s[i].m[4]);
     }
 }
```

xi)
```
 main()
 {
    struct Date
    {
        int dd;
        int mm;
        int yy;
    };
    struct   Date today, *dp;

    dp     = &today;
    dp->mm = 9;
    dp->dd = 25
    dp->yy  =  1998;
    printf("today"s  date is  %d/%d/%d\n", dp->dd, dp->mm,  dp->yy);
  }
```

Unions

Unisons, like structures, contain member whose individual late type may differ from one another. However, the member that compose a union all share the same storage area within the computer's memory. Thus, unions are used to multiple member. they are useful for application involving multiple member, where values need not be assigned to all members at any one time.

In general terms, the composition of a unions may be defined as

```
union  tag
{
        member    1;
        member    2;
        ---
```

```
          ---
          member     n;
       };
```

Where union is a required keyword; the other terms having the same meaning as in a structure definition. Individual union variables can then be declared as

storage-class union tag variable 1, variable 2,..., variable n;

The two declaration may be combined. Thus we can write

```
storage-class union tag
{
      member    1;
      member    2;
      ---
      ---
     member     n;
  }    variable;
```

The tag is optional in this type of declaration.

A union may be a member of a structure, and a structure may be a member of a union. However, it should be noted that only one member of a union can be assigned a value at any one time.

A union variable will be large enough to hold the largest of the data type of its member. In effect a union is a structure in which all members have offset zero from the base; the structure is big enough to hold the widest member.

Example :

```
        struct
        {
              char  name[30];
              int   flag;
              int   utype;
              union    u
              {
                   int       f1;
                   char        f2;
                   float        f3;
               }      u1;
         }  v;

         if(v.utype       = =  1)
           printf("%d\n",  v.u1.f1);
         else if(v.utype   = =  2)
           printf("%c\n",  v.u1.f2);
         else if(v.utype  = = 3)
           printf("%f\n",  v.u1.f3);
         else
           printf(ïnvalid  type\n",);
```

Chapter XII
Special Features

1. User-defined Data Type (typedef)

The typedef feature allow users to define new data type that are equivalent to existing data type. Once a user defined data type has been established, then new variable, arrays, structures, and so on can be declared in terms of this new data type.

In general terms, a new data type is defined as

```
typedef      type        new-type  ;
```

Where type refers to an existing data type, and new type refers to the new user defined data type .

Example:

```
typedef    int     age ;
age      male,   female;

typedef    float height;
height     men,   women ;
```

The typedef feature is particularly convenient when defining structures since it eliminates the need to repeatedly write strict tag whenever a structure is referenced.

In general terms, a user-defined structure type can be written as

```
          typedef      struct{
           member    1;
           member    2 ;
           ---
           ---
          member  n;
     }  new -type;
```

Example:

```
typedef sturct
{
             int     acc Num;
             char    accType;
             char    name[50];
             float    balance;
}    Record;
```

Record oldCust , newCust;

It must be emphasized that a typedef declaration does not create a new type in any case, it merely adds a new name for some existing type. Thus the basic advantage is a purely aesthetic one.

2. Bit Fields

In some application, it may be desirable to work with data items that consist of only a few bits (e.g.: a single-bit flag to indicate a T/F condition, a 3-bit integer whose values can range individual word of memory. To do so, the word is sub-divided into individual bit-fields. These bit fields are defined as individual, like any other member of a structure.

In general terms, the decomposition of a word into distinct bit fields can be written as.

```
struct      tag
{
       member    1;
       member    2 ;
 -----
```

```
    -----
     };
```

Each member declaration must include a specification indicating the size of the corresponding bit field. To do so, the member name must be followed by a colon & an unsigned integer indicating the field size.

A field within a structure can't overlap a word within the computer's memory. If the sum of the field widths does exceed the word size, then any overlapping field will automatically be forced to the beginning of the next word.

Unnamed fields can be used to control the alignment of bit fields within a word of memory. Such fields provide padding within the word. The size of the unnamed field determines the extent of the padding.

Another way to control the alignment of bit field is to include an unnamed field whose width is zero. This will automatically force the next Field to be aligned with the beginning of a new word. Bit-field to be aligned with the manner as other structure members, and they may appear within arithmetic expressions as unsigned integer quantities. There are, however, several restrictions on their use. In particular, arrays of bit fields are not permitted, the address particular, (&) cannot be applied to a bit field, a pointer can't access a bit field, and a function can't return a bit field.

Example:

```
    struct   flagstruct
    {
          unsigned      f1  :1:
          unsigned      f2  :1;
          unsigned      f3  :1;
          unsigned      f4  :1;
     }  flag;
```

This definition defines 4 flags each having 1 bit and can store either 0 or 1

```
          flag . f1    =  0;          (means    off )
          flag . f1    =  1;          (means    on )
          flags. f4    =  0;
```

3. Bit Operators

The operations are carried out independently on each pair of corresponding bits within the operands. The least significant bits within the two operands will be compared, then the next least significant bits, and so on, until all of the bits have been compared. The results of these comparisons are:

- A bitwise AND expression will return 1 if both bits have a value of 1 (i.e. If both bits are TRUE). Otherwise, it will return a value of 0.

- A bitwise exclusive OR expression will return a 1 if one of the bits has a value of 1 and the other has a value of 0 (one bit is true, the other false). Otherwise, it will return a value of 0.
- A bitwise inclusive OR expression will return a 1 if one or more of the bits have a value of 1 (one of both bits are true). Otherwise, it will return a value of 0.

The two bitwise shift operators also require 2 operands. The first is an integer type operand that represents the bit pattern to be shifted. The second is an unsigned integer that indicates the number of displacements. This value cannot exceed the number of bits associated with the word size of the first operand.

The left-shift operator causes all the bits in the first operand to be shifted to the right by the number of positions indicated by the second operand. The leftmost bits in the original bit pattern will be lost. The rightmost bit positions that become vacant will be filled with 0's.

The right-shift operator causes all the bits in the first operand to be shifted to the right by the number of positions indicated in the second operand. The rightmost bits in the original bit pattern will be lost. If the bit pattern being shifted represents an unsigned integer, then the left most bit positions that become vacant will be filled with 0's.

The one's complement operator causes the bits of its operand to be inverted so that 1s become 0s and 0s become 1s. This operator always precedes it operand. The operand must be an

Example:

```
    a        :     01111010
    b        :     11001011
```

Then

```
    a &  b   :     01001010
    a !  b   :     11111011
    a ^  b   :     10110001
    a >>  2  :     00011110
    a <<  3  :     01011000
```

Example:

```
     /* to change case */

     #include <stdio.h>
     #include <ctype.h>

    main()
    {
        char c;
        while ((c =getchar ())  !  = EOF)
                if((c>='a'&&c<='z')||(c>='A'&&c<='Z'))
               putchar(c^32);
     }
```

Example:

```
   /*  To exchange two values without the need for a temporary storage location */

        int  i1, i2;

        i1  ^  i2;
        i2  ^  i1;
        i1  ^  i2;
```

Masking

Masking is a process in which a given bit patterns is transformed into another bit pattern by means of a logical bitwise operation. The original bit pattern is one of the operands in the bitwise operation. The second operand called the mask, is a specially selected bit pattern that brings about the desired transformation.

Example:

To switch off the rightmost bit of a character variable.

```
    char swoff(char a)
    {
       return(a  &  ((-0) >> 1 ));
    }
```

4. Formatted Input / Output

i) scanf

The consecutive non-whitespace characters that compare a data item collectively define a field. It is possible to limit the number of such characters by specifying the maximum field width for that data item. To do so, an unsigned integer indication the field width is placed within the control string, indicating the field width is placed within the control string, between the percent sign (%) and the conversion character. The number of characters in the actual data item cannot exceed the specified field width. Any characters that extend beyond the specified field width will not be read. Such leftover characters may be incorrectly interpreted as the components of the next data item.

Example:

```
#include <stdio.h>

main( )
{
     int  I;
     float  x;
     char  c;
     ...
     Scanf("%3d%5f%c", &I, &x, &c);
     …
 }

 Input :  10  256.875  T
```

The 10 will be assigned to I, 256.8 will be assigned to x and the character 7 will be assigned to c. The remaining two input characters will be ignored.

It is also possible to skip over a data-item without assigning it to the designated variable or array. To do so, the %sign within the appropriate control group is followed by an asterisk (*).

Example:

```
#include <stdio.h>

main( )
{
    char item[20];
    float cost;
    …
    scanf("%s%*d%f", item, &cost);
    …
}
fast 12345 0.05
```

then fast will be assigned to item and 0.05 will be assigned to cost. However, 12345 will be ignored.

Some more conversion characters

%f	:	prints as [-]m.dddddd
%e	:	prints as [-]m.dddddde-xx
%g	:	prints using %e if exponent is less than -4 or greater than or equal to the precision, otherwise uses %f
%h	:	short integer
%i	:	decimal, octal or hexadecimal integer
%o	:	octal integer

%x : hexadecimal integer

Note : In the printf function, s-type conversion is used to output a string terminated by the null character ('\0'). whitespace characters may be included in the string.

A minimum field width can be specified by preceding the conversion character by an unsigned integer. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field. If the number of characters the data item exceeds the specified field width, however, then additional space will be allocated to the data item, so that the entire data item will be displayed. This is just the opposite of the field width indicator in the scanf function, which specifies a maximum field width. It is possible to specify the maximum number of decimal places for a floating-point value or the maximum number of characters for a string. This specification is known as precision. The precision is an unsigned integer that is always preceded by a decimal point.

A floating-point number will be rounded if it must be shortened to confirm to a precision specification.

It is possible to specify the precision without the minimum field width, though the precision must still be preceded by a decimal point.

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

main( )
{
   float  x = 123.456

   printf("%f %.3f %.1f\n", x, x, x);
   printf("%e % .5e %.3e\n", x, x, x );
}
Output:
   123.456000 123. 456 123.5
   1.234560e+002 1.23456e+002 1.235e+002
```

Minimum field-width and precision specification be applied to character data as well as numerical data. When applied to a string, the minimum field with is interpreted in the same manner as with a numerical quantity, i.e. leading blanks will be added if the string is shorter than the specified field width, and additional space will be allocated if the string is longer than the specified field width.

However, the precision specification will determine the maximum number of characters that can be displayed. If the precision specification is less than the total number of characters in the string, the excess rightmost characters will not be displayed.

In addition, each character group within the control string can include a flag, which affects the appearance of the output. The flag must be placed immediately after the percent sign (%).

Commonly used flags

-	data item is left-justified with the field
+	a sign (+ or -) will precede each signed numerical data item
0	causes leading zero to appear instead of leading blanks

Example :

i) `int  i, j,  k ;`

```
char   d[50];
float   a,   b,   c;
scanf("%d%d%f%s",  &i,  &j,  &a,  d);
```

ii) `scanf ("%2d%2d%2d",  &dd,  &mm,  &yy );`

input given is 200290 and no white space as delimiter is to be given.

iii) `scanf("%4d%7f",  &i,  &f,);`

if input is 50012000.00 then i = 5001 and f = 2000.00

iv) `scanf ("%7f%4d%c",   &f,  &i,  &c);`

now instead of c we can also try to give %20c which is used for a character array.

i.e., `scanf("%20c", str);`

in such a case 20 char acters can be accepted but such a string is not a NULL terminated string. also, space will not be treated as a delimiter.

v) `scanf("%20s",  str);`

The string now accepted will be a NULL-terminated string and now white space are treated as delimiter.

Ordinary characters can be given in the scanf which are to be input as such.

vi) `scanf("%2d/%2d/%2d",  &dd, &mm, &yy);`

input to be given is 20/02/90

vii) `scanf("%2d%f%*d%2s".  &i,  &x,  name );`

if input given is

56789b/0123b/45a072

i = 56, x = 789, 0123 is skipped and "45" is assigned to name.

if a specified number of characters are to be skipped we can give %*4c (now 4 characters will be skipped)

Escape sequences can be included in the control string to have desired effects

\\	-	backslash
\'	-	apostrophe
\"	-	quotes
\ddd	-	bit pattern (specify any byte value of a character ddd are octal digits)
\033		will be escape, \003 is ctrl c and \007 is for ctrl G (Bell sound)

Example :

i) `printf("%4d,  i);`

Here i is printed in at least 4 digits i.e. if value is 25 answer will be b/b/25. If value is 67853 it will be printed as such. (For a real number default precision is 6).

ii) `printf("%7.2f",    f );`

will print total 7 characters and 2 places reserved after the decimal.

if f = 603537 then   output is b/b/b/6.35 <- total seven spaces including decimal point.

if f = 6.3567 then output is b/b/b/6.36

iii)| `printf("%10s",  str );`

if str contains Hello, then output is
b/b/b/b/b/ Hello

(i.e. by default, every value is right-justified)

if string to be output is longer say "hello, world" it will be printed as such.
"%-10s"   indicates it is left-justified.
"%10.10", str
Indicates store only 10 characters, i.e. If string is bigger truncation will take place.
"%-20.10s"
Justification is left; only 10 characters are to be printed

Example:

```
"%10s"          hello, word
"%-10s"          hello, world
"%20s"           b/b/b/b/b/b/b/b/b/hello, world
"%-20s"         hello, world b/b/b/b/b/b/b/b/b
"%20.10s"       b/b/b/b/b/b/b/b/b/b/hello, world
"%-20.10s"      hello, world/b/b/b/b/b/b/b/b/b/b
"%.10s"          hello, world
```

5. Enumerations

An enumeration is a data-type similar to a structure or a union. Its members are constants that are written as identifiers, though they have signed integer values.

In general terms, an enumeration may be defined as [storage-class] enum tag {member1, member2,…. Member M} var 1,...; the member names must differ from one another.

Example:

```
enum colors{red, green, yellow, blue}  fg, bg ;
```

Enumeration Constants are automatically assigned equivalent integer values, beginning with 0 for the first constant, with each successive constant increasing by 1.

Thus, member 1 will automatically be assigned the value 0, member 2 will be assigned 1 and so on.

Enumeration variable can be processed in the same manner as other integer variables, i.e., they can be assigned new values, compared and so on. However, there are certain restrictions associated with their use. In particular, an enumeration constant cannot be read into the computer and assigned to an enumeration variable. Moreover, only the integer value of an enumeration variable can be written out.

Example:

```
fg      =      blue ;
bg      =      green ;
```

An enumeration variable can be initialized also. This can be done by assigning either an enumeration constant or an integer value to the variable.

Example:

```
enum colors {blue, cyan, green, red, white} ;
colors  fg = green, bg    =  cyan ;
```