UNIT-6

POINTER

A pointer provides a way of accessing a variable without referring to the variable directly. A pointer variable is a variable that holds the memory address of another variable. The pointer does not hold a value. A pointer points to that variable by holding a copy of its address. It has two parts

1) the pointer itself holds the address
2) the address points to a value

Uses of pointers:

1) Call by Address
2) Return more than one value from a function
3) Pass array and string more conveniently from one function to another
4) Manipulate arrays more easily
5) Create complex data structures
6) Communicate information about memory
7) Fast compile

Declaring Pointer: The pointer operator a variable in C is '*' called "value at address" operator. It returns the value stored at a particular memory the value at address operator is also called "indirection" operator a pointer variable is declared by preceding its name with an asterisk(*) .

Syntax: *datatype *pointer_variable;*

Where datatype is the type of data that the pointer is allowed to hold and pointer_variable is the name of pointer variable.

Example:

```
int *ptr;
char *ptr;
float *ptr;
```

Sample program:

```
main()
{
int *p;
float *q;
double *r;
printf("Size of the integer pointer %d",sizeof(p));
printf("size of the float pointer %d",sizeof(q));
printf("size of double pointer %d",sizeof(r));
}
```

Initializing Pointers:

A pointer must be initialized with a specified address operator to its use. For example to store the address of I in p, the unary & operator is to be used. P=&I;

Sample program:

```
main()
{
int num=5;
int *ptr=&num;
printf("the address num is %p",&num);
printf("the address num is %p",ptr);
}
```

Address Operator(&): It is used as a variable prefix and can be translated as address of this & variable can be read as "address of variable".

```
int *p;
int a;
p=&a;
```

Pointer arithmetic or Address Arthmatic:

Some of the valid operation on the pointer

1) Assignment of pointers to the same type of pointers.
2) Adding or subtracting a pointer and an integer
3) Subtracting or comparing two pointers
4) Incrementing or decrementing the pointers
5) Assigning the value 0 to the pointer variable and comparing 0 with pointer.

Invalid operation for pointer

1) adding of two pointers
2) multiplying a pointer with a number
3) dividing a pointer with a number

1) **Assignment:** pointers with the assignment operators can be used if the following conditions are
   1) lefthand side operand is a pointer is a pointer and right hand operand is a null pointer constant.
   2) Incompatible type of pointer assignment
   3) Both the operands are pointers to compatibles.

Sample program:

```
main()
{
int i;
int *ip;
void *vp;
ip=&i;
vp=ip;
ip=vp;
if(ip!=&i)
printf("Compiler error");
else
printf("no compiler error");
}
```

2) Addition or Subtraction with integers

Addition: the operator + performs the addition operation between the pointer and number. For example p is a pointer, pointer p adds with value s means (p+s).

Example

```
main()
{
int a[]={10,12,6,7,2};
int i;
int sum=0;
int *p;
p=a;
for(i=0;i<5;i++)
{
sum=sum+*p;
p++;
}
printf("%d",sum);
getch();
}
```

3) Subtraction: the operator '-' performs the subtraction operation between the pointer and number. For example p is a pointer. Pointer p subtract value 5 mean (p-5).

Example:

```
main()
{
double a[2],*p,*q;
p=a;
p=p-1; printf("%d",(int)q-
(int)p);
}
```

4) Comparing pointers: C allows pointer to be compared with each other. If two pointer compare equal to each other.

```
main()
{
int a,*p,*q; p=&a;
q=&a; if(*p==*q)
printf("both are equal");

}
```

5) Assign the value 0 to the pointer and compare 0 with pointer: P declare as a pointer naturally it stores the address of another variable. Suppose 0 assigned to pointer, like p=0; and also pointer variable p compared with 0.

For example,

```
main()
{
int *p;
p=0;
if(p==0)
printf("P contains zero");
else
printf("p contains a non-zero value");
}
```

Pointers to pointers: Pointers to pointers concept supports the pointer of pointer, means pointer stores address of another variable. Pointer of pointer stores address of another pointer.

```
int a=5;
int *p;
int **q;
p=&a;
q=*p;
```

Hera, a is an integer variable it has value 5. P is integer pointer it has address of a. q is integer pointer of pointer it has address of p.

```
main()
{
int a=5;
int *p,**q;
p=&a;
q=&p;
printf("*p=%d",*p);
printf("**q=%d",**q);
}
```

Pointers to functions: Pointers are pointer i.e., variable which point to the address of a function. A running program is allocated a certain space in the main memory.

Declaration of a pointer to a function:

Syntax: *returntype(*function_pointer_name)(argument1,argument2,---);*

In the following example, a function pointer named fp is declared. It points to the functions that take one float, and two char and return as int.

*int (*fp)(float,char,char);*

Initialization of Function Pointers:

It is quite easy to assign the address of a function to a function pointer. It is optional to use the address operator & in front of the function's name. for example if add() and sub() are declared as follows.

```
int add(int,int);
int sub(int,int);
```

The names of the functions add and sub is pointers to those functions. These can be assigned to pointer variables.

```
fpointer=add;
fpointer=sub;
```

Calling of a function using a function pointer: in c there are two ways of calling a function using a function pointer. Use the name of the function pointer instead of the name of the function pointer instead of the name of the function or explicitly deference it.

```
Result=fpointer(4,5);
Result2=fpointer(6,2);
```

Example:

```
int (*fpointer)(int,int);
int add(int,int);
int sub(int,int);
main()
{
fpointer=add;
printf("%d",fpointer(4,5));
fpointer=sub;
printf("%d",fpointer(6,2));
}
int add(int a,int b)
{
return (a+b);
}
int sub(int a,int b)
{
return (a-b);
}
```

Passing a function to another function:

A function pointer can be passed as a function's calling argument.

Example:

```
double sum(double f(double,double),int m,int n)
{
int k;
double x;
double s=0.0;
printf("Enter the value of x");
scanf("%lf",&x);
for(k=m;k<=n;++k)
s+=f(k,x);
return s;
}
```

The following is an equivalent header to the function.

```
double sum(double (*f)(double),int m,int n)
{
same as above
}
```

Arrays of Pointers: an array of pointers can be declared very easily.

```
int *p[10];
```

This declares an array of 10 pointers each of which points to an integer. The first pointer is called p[0], second is p[1] and so on up to p[9].

Example:

```
main()
{
int *p[10];
int a=5,b=6,c=10;
p[0]=&a;
p[1]=&b;
p[2]=&c;
printf("Address of a %p",p[0]);
printf("Address of b %p",p[1]);
printf("Address of c %p",p[2]);
}
```

Pointers to an Array: we can declare a pointer to a simple integer value and make it point to the array asa is done normally.

```
int v[5]={1004,2201,3000,432,500};
int *p=v;
printf("%d",*p);
```

This piece of code displays the number, which the pointer p points to, that is the first number in the array namely 1004.

P++ this instruction increases the pointer so that it points to the next element of the array.

*printf(%d",*p);* this statement prints value 2201.

Pointers and Multidimensional Arrays:

It is usually best to allocate an array of pointers, and then initialize each pointer to a dynamically allocated row. Here is an example:

```
#include<stdio.h>
#include<stdio.h>
#define row 5
#define col 5
main()
{
int **arr,i,j;
arr=(int**)malloc(row*sizeof(int*));
if(!arr)
{
printf("out of memory\n");
exit(EXIT_FAILURE);
}
for(i=0;i<row;i++)
{
arr[i]=(int*)malloc(sizeof(int)*col);
if(!arr[i])
{
printf("Out of memory\n");
exit(EXIT_FAILURE);
}
}
printf("\nEnter the elements of the matrix row by row\n");
for(i=0;i<row;i++)
for(j=0;j<col;++j)
scanf("%d",&arr[i][j]);
printf("the matrix is as follows…\n");
for(i=0;i<row;++j)
{
printf("\n");
for(j=0;j<col;++j)
printf("%d\t",arr[i][j]);
}
}
```

With exit(), status is provided for the calling process as the exit status of the process.

STATUS	INDICATES
EXIT_SUCCESS	Normal program terminals
EXIT_FAILURE	Abnormal program termination

Generic pointer (or) void pointer:

void pointer in c is known as generic pointer. Literal meaning of generic pointer is a pointer which can point type of data.

Example:

```
void *ptr;
```

Here ptr is generic pointer.

Important points about generic pointer in c?

We cannot dereference generic pointer.

```
#include<stdio.h>
#include <malloc.h>
int main(){
        void *ptr;
        printf("%d",*ptr);
return 0;
}
```

Output: Compiler error

2. We can find the size of generic pointer using sizeof operator.

```
#include <string.h>
#include<stdio.h>
int main(){
        void *ptr;
        printf("%d",sizeof(ptr));
return 0;
}
```

Output: 2

Explanation: Size of any type of near pointer in c is two byte.

3. Generic pointer can hold any type of pointers like char pointer, struct pointer, array of pointer etc without any typecasting.

Example:

```
#include<stdio.h>
int main(){
        char c='A';
        int i=4; void
        *p; char
        *q=&c; int
        *r=&i; p=q;

        printf("%c",*(char *)p);
        p=r; printf("%d",*(int
        *)p);
return 0;
}
```

Output: A4

4. Any type of pointer can hold generic pointer without any typecasting.

5. Generic pointers are used when we want to return such pointer which is applicable to all types of pointers. For example return type of malloc function is generic pointer because it can dynamically allocate the memory space to stores integer, float, structure etc. hence we type cast its return type to appropriate pointer type.

Examples:

1.
```
char *c;
c= (char *)malloc(sizeof(char));
```

2.
```
double *d;
d= (double *)malloc(sizeof(double));
```

3.
```
Struct student{
        char *name;
        int roll;
};
Struct student *stu;
Stu=(struct student *)malloc(sizeof(struct student));
```

NULL pointer in c programming

NULL pointer:

Literal meaning of NULL pointer is a pointer which is pointing to nothing. NULL pointer points the base address of segment.

Examples of NULL pointer:

```
1. int *ptr=(char *)0;
2. float *ptr=(float *)0;
3. char *ptr=(char *)0;
4. double *ptr=(double *)0;
5. char *ptr='\0';
6. int *ptr=NULL;
```

What is meaning of NULL?

NULL is macro constant which has been defined in the heard file stdio.h, alloc.h, mem.h, stddef.h and stdlib.h as

```
#define NULL 0
```

Examples:What will be output of following c program?

```
#include <stdio.h>
int main(){
        if(!NULL)
        printf("I know preprocessor");
        else
        printf("I don't know preprocessor");
 return 0;
}
```

Output: I know preprocessor

Explanation:

!NULL = !0 = 1

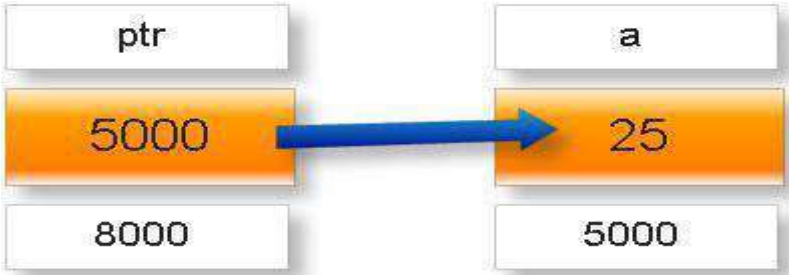
In if condition any non zero number mean true.

Dangling pointer problem in c programming

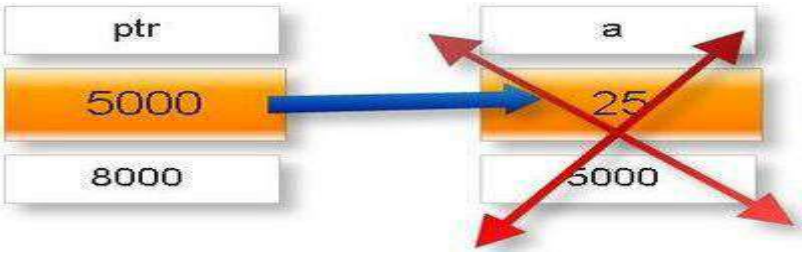
1. Dangling pointer:

If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as dangling pointer and this problem is known as dangling pointer problem.

Initially:



Later:



For example:

(q)What will be output of following c program?

```
#include<stdio.h>
int *call();
void main(){

        int *ptr;
        ptr=call();
        fflush(stdin);
        printf("%d",*ptr);
}
int * call(){
        int x=25;
        ++x;
        return &x;
}
```

Output: Garbage value

Note: In some compiler you may get warning message **returning address of local variable or temporary**

Explanation: variable x is local variable. Its scope and lifetime is within the function call hence after returning address of x variable x became dead and pointer is still pointing ptr is still pointing to that location.

Solution of this problem: Make the variable x is as static variable.
In other word we can say a pointer whose pointing object has been deleted is called dangling pointer.

```
#include<stdio.h>
int *call();
void main(){
        int *ptr;
        ptr=call();

        fflush(stdin);
        printf("%d",*ptr);
}
int * call(){
        static int x=25;
        ++x;
        return &x;
}
```

Output: 26

Character Pointers and Functions

Since text strings are represented in C by arrays of characters, and since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C.

C does not provide any operators for processing an entire string of characters as a unit.We've said this sort of thing before, and it's a general statement which is true of all arrays. Make sure you understand that in the lines

```
        char *pmessage;
pmessage = "now is the time";
pmessage = "hello, world";
```

all we're doing is assigning two pointers, not copying two entire strings.

We also need to understand the two different ways that string literals like "now is the time" are used in C. In the definition

```
char amessage[] = "now is the time";
```

The string literal is used as the initializer for the array amessage. amessage is here an array of 16 characters, which we may later overwrite with other characters if we wish. The string literal merely sets the initial contents of the array. In the definition

```
char *pmessage = "now is the time";
```

on the other hand, the string literal is used to create a little block of characters somewhere in memory which the pointer pmessage is initialized to point to. We may reassign pmessage to point somewhere else, but as long as it points to the string literal, we can't modify the characters it points to.

As an example of what we can and can't do, given the lines

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

we could say

```
amessage[0] = 'N';
```

to make amessage say "Now is the time". But if we tried to do

```
pmessage[0] = 'N';
```

The first function is strcpy(s,t), which copies the string t to the string s. It would be nice just to say s=t but this copies the pointer, not the characters.

This is a restatement of what we said above, and a reminder of why we'll need a function, strcpy, to copy whole strings.

Once again, these code fragments are being written in a rather compressed way. To make it easier to see what's going on, here are alternate versions of strcpy, which don't bury the assignment in the loop test. First we'll use array notation:

```
void strcpy(char s[], char t[])
{
        int i;
        for(i = 0; t[i] != '\0'; i++)
                s[i] = t[i];
        s[i] = '\0';
}
```

Note that we have to manually append the '\0' to s after the loop. Note that in doing so we depend upon i retaining its final value after the loop, but this is guaranteed in C, as we learned in Chapter 3.

Here is a similar function, using pointer notation:

```
void strcpy(char *s, char *t)
{
        while(*t != '\0')
        *s++ = *t++;
        *s = '\0';
}
```

Again, we have to manually append the '\0'. Yet another option might be to use a do/while loop.

Dynamic memory allocation

Dynamic memory allocation is the practice of assigning memory locations to variables during execution of the program by explicit request of the programmer. dynamic allocation is a unique feature to C

Finally, the dynamically allocated memory area, the area is secured in a location different from the usual definition of a variable. Used in the dynamic memory allocation area is called the heap..

The following functions are used in C for purpose of memory management.

1.malloc() 2.calloc() 3.realloc() 4.free()

malloc()

The **malloc**() function dynamically allocates memory when required. This function allocates 'size' byte of memory and returns a pointer to the first byte or NULL if there is some kind of error.

Format is as follows.

void * malloc (size_t size);

The return type is of type void *, also receive the address of any type. The fact is used as follows.

double * p = (double *) malloc (sizeof (double));

The size of the area using the sizeof operator like this. The return value is of type void *, variable in the receiving side can be the pointer of any type in the host language C .In C, a pointer type to void * type from another, so that the cast automatically, there is no need to explicitly cast originally.

With this feature, you get a pointer to an allocated block of memory. Its structure is:

code:

pointer = (type) malloc (size in bytes);

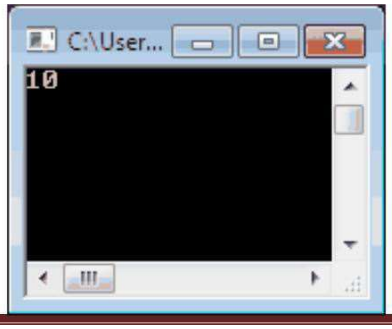
An example:

code:

```
int * p;
p = (int *) malloc (sizeof (int));
* p = 5;
```

The following example illustrates the use of malloc() function.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int a,*ptr;
a=10;
```



```
ptr=(int*)malloc(a*sizeof(int));
ptr=a;
printf("%d",ptr);
free(ptr);
getch();
}
```

calloc() function

The calloc function is used to allocate storage to a variable while the program is running. This library function is invoked by writing calloc(num,size).This function takes two arguments that specify the number of elements to be reserved, and the size of each element in bytes and it allocates memory block equivalent to num * size . The important difference between malloc and calloc function is that calloc initializes all bytes in the allocation block to zero and the allocated memory may/may not be contiguous. For example, an int array of 10 elements can be allocated as follows.

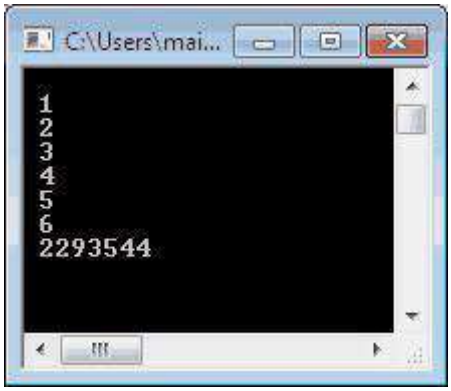
```
int * array = (int *) calloc (10, sizeof (int));
```

Note that this function can also malloc, written as follows.

```
int * array = (int *) malloc (sizeof (int) * 10);
        ptr = malloc(10 * sizeof(int));        is just like this:
        ptr = calloc(10, sizeof(int));
```

The following example illustrates the use of

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int *ptr,a[6]={1,2,3,4,5,6};
int i;
ptr=(int*)calloc(a[6]*sizeof(int),2);
for(i=0;i<7;i++)
{
printf("\n %d",*ptr+a[i]);
}
free(ptr);
getch();
}
```



realloc()

With the function realloc, you can change the size of the allocated area once. Has the following form.

void * realloc (void * ptr, size_t size);

The first argument specifies the address of an area that is currently allocated to the size in bytes of the modified second argument. Change the size, the return value is returned in re-allocated address space. Otherwise it returns NULL.

The address of the source address changed, but the same could possibly be different, even if the different areas of the old style, because it is automatically released in the function realloc, for the older areas it is not necessary to call the free function. #include<alloc.h>

```
#include<string.h>
main()
{
char *str;
clrscr();
str=(char*)malloc(6);
str=("india");
printf("str=%s",str);
str=(char*)realloc(str,10);
 strcpy(str,"hindustan");
printf("\nnow str=%s",str);
free(str);
getch();
}
```

free() function

Next, how to release the reserved area, which will use the function free. Has also been declared in stdlib.h, which has the following form. void free (void * ptr);

The argument specifies the address of a dynamically allocated area. You can then free up the space. Specify an address outside the area should not be dynamically allocated. Also, if you specify a NULL, the free function is guaranteed to do nothing at all

So, an example program that uses functions malloc and free functions in practice.

```
# include<stdlib.h>
# include<conio.h>
int main (void)
{
int * p;
p = (int *) malloc (sizeof (int)); /* partitioning of type int */
if (p == NULL) * / failed to reserve area * /
{
printf ("Failed to allocate space for% d bytes", sizeof (int));
return 1;
}
* P = 150;
printf ("% d \ n", * p);
free (p); * / free area * /
return 0;
}
```

COMMAND LINE ARGUMENTS

It is possible to pass arguments to C programs when they are executed. The brackets which follow main are used for this purpose. *argc* refers to the number of arguments passed, and *argv[]* is a pointer array which points to each argument which is passed to main. A simple example follows, which checks to see if a single argument is supplied on the command line when the program is invoked.

```
#include <stdio.h>

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

Note that **argv[0]* is the name of the program invoked, which means that **argv[1]* is a pointer to the first argument supplied, and **argv[n]* is the last argument. If no arguments are supplied, *argc* will be one. Thus for n arguments, *argc* will be equal to n + 1. The program is called by the command line,

```
myprog  argument1
```

Example : commandline.c

```
#include<stdio.h>
main(int argc,int *argv[])
{
int i;
printf("No of arguments %d",argc);
for(i=0;i<argc;i++)
{
printf("\n %s",argv[i]);
}
}
```

```
C:\CSE>filename arg1 arg2 …….. argn
C:\CSE>commandline hello how are u
No of arguments 5
commandline
hello
how
are
u
```

DERIVED DATATYPES:

C provides facilities to construct user defined datatypes. A user defined datatypes may also be called a derived datatype. Some of the datatypes are…

1. structures
2. unions
3. typedef
4. arrays
5. enumerators…etc.,

STRUCTURE

→ Def. of structure:

Structure is the user defined datatype that can hold a heterogeneous datatypes. Structures are constructing with "struct" keyword and end with semicolon…

→ Declaration of structures and structure variables:

Structures are declared by using struct keyword followed by structure name followed by body of the structure. The variables or members of the structure are declared with in the body.

Syntax:

```
struct structname
  {
   datatype member1;
    .
     .
     .
   datatype member n;
   };
   struct structurename <struct variable1>….<struct variable n>;
```

In the above syntax structurename is the name of the structure. The structure variables are the list of the variable names separated by comas. Variables declared with in the braces called members or variables.

Example:

```
struct country
 {
   char name[30];
    int population;
    char language[15];
  } India,japan,England;
```

In the above example country is the structure name and structure members are name, population, language and structure variable are India, Japan and England.

→ Initialisation of structure:

This consists of assigning some constants to the constants to the members of the structure. Default values for integer and float datatype members are zero. For char member default value '\0' (null).

Syntax:

```
struct struct name
{
datatype member1;
.
.
.
datatype member n;
}<structvariable>={constant1,constant2…}
```

Example:

```
#include<stdio.h>
struct item
{
        int count;
        float avgweight;
        int date,month,year;
} batch={2000,25.3,07,11,2010};
main()
{
printf("\n count=%d",batch.count);
printf("\n averageweight=%f",batch.avgweight);
printf("\nmfg-date=%d%d%d",batch.date,batch.month,batch.year);
}
```

→ Accessing the member of a structure:

The members of the structure can be accessed by using "." Operator(dot operator).

Syntax:

<structurevariable>.<membername>;

The .(dot) operator selects a particular member from a structure.

Example:

```
struct list
    {
      int a;
      float b;
    } s,s1;
```

Accesing the a,b from the structure list is

s.a, s.b scanf("%d%f",&s.a,&s.b);

s1.a, s1.b scanf("%d%f",&s1.a,&s1.b);

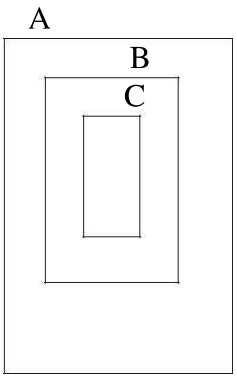
same as printing values

printf("%d%f",s.a,s.b);

printf("%d%f",s1.a,s1.b);

→ Nested structures:

A structure can be placed with in another structure. In other words structures can contain other structures as members. A structure with in a structure means nested structures.



Outermost structure (named A)
Inner structure (named B)
Innermost structure (named C)

Example:

```
#include<stdio.h>
struct b
{
int x;
struct b
{
int y;
struct c
{
int  z
}c1;
}b1;
}a1;
main()
{
        a1.x=10;
        a1.b1.y=20;
        a1.b1.c1.z=30;
```

```
        printf("x value %d",a1.x);
        printf("y value is %d",a1.b1.y);
        printf("z value is %d:,a1.b1.c1.z);
        getch();
}
```

→ Arrays of structures:

This means that the structure variable would be an array of objects. Each of which contains the number elements declared within the structure construct.

Syntax:

```
struct <structname>
{
                <datatype member1>;
<datatype member2>;
.
.
.
<datatype member n>;
}<structure variable[index];
```

(or)

```
struct <structname><structurevariable>[index];
```

Here the term index specifies the number of array objects.

Example:

```
#include<stdio.h>
struct Prasad
{
int a;
float b;char c;
};
main()
{
struct Prasad p[3];
int i;
clrscr();
printf("enter details\n");
for(i=0;i<3;i++)
scanf("%d%f%c",&p[i].a,&p[i].b,&p[i].c);
printf("entered details are \n");
for(i=0;i<3;i++)
printf("%d%f%c",p[i].a,p[i].b,p[i].c);
getch();
}
```

→ Initialising arrays of structures:

Example:

```
struct student
{
char name[30];
int rno,age;
};
main()
{
int i;
struct student
s[3]={{“prassad”,1201,30},{“raju”,530”,29},{“rani”,430,30}};
printf(“details fro students”);
for(i=0;i<3;i++)
printf(“\n %s\t %d\t%d”,s[i].name,s[i].rno,s[i].age);
getch();
}
```

→ Size of the structure:

Size of the structure specifies the structure occupied how many bytes using “sizeof” keyword..

Example:

```
struct student
{
char name[20];
int a[10];
float b[20];
}s;
main()
{
clrscr();
printf(“the size of the structure is %d”,sizeof(s));
getch();
}
```

→ Structures and pointers:

A pointer to a structure is not itself a structure, but merely a variable that holds the address of a structure this pointer variable takes four bytes of memory.

Syntax:

```
struct <structname>
{
<datatype member1>;
<datatype member2>;
.
.
<datatype membern>;
}*ptr;
```

Or

```
struct <structname> *ptr;
```

The pointer *ptr can be assigned to any other pointer of the same type and can be used to access the members of its structure using pointer ,access the structure members in two ways…

One is…

(*ptr).member1;

The bracket is needed to avoid confusion about '*' and '.' Operators.

Second is…

ptr→member1;

This is less confusing and better way to access a member in a structure through its pointer. The → operator, an arrow made out of a minus sign and a greater than symbol.

Example:

```
#include<stdio.h>
#include<conio.h>
struct abc
{
int a;
float b;
}*p;
main()
{
printf("initialization for  members/n");
p→a=10;
p→b=20.12;
printf("a=%d,b=%f",p→a,p→b);
printf("read the values in");
scanf("%d%f",&p→a,p→b);
printf("\n a=%d,b=%f",p→a,p→b);
}
```

→ Structures and functions:

An entire structure can be passed on a function arguments just like any other variables.when a structure is passed as an argument,each member of the structure is copied.

Syntax:

struct<structurename><functioname>(struct structrename structurevariable);

Example:

```
#include<stdio.h>
struct A
{
char ch;
int i;
float f;
};
void show(struct A);
main()
{
struct A x;
printf("in entre ch,i and f");
scanf("%c%d%f",&x.ch,&x.i,&x.f);
show(x);
}
void show(A b)
{
printf("%c%d%f",b.ch,b.i,b.f);
}
```

→ typedef:

The typedef keyword allows the programmer to create a new datatype name for an existing datatype. No new datatype is produced but an alternative name is given to a known datatype

Syntax:

typedef <existing datatype><new datatype,…>;

typedef statement does not occupy storage, it simply defines a new type.

```
typedef int id;
typedef float wt;
typedef char lw;
```

id is the new datatype name given to datatype int, while wt is the new datatype name given to the datatype float,a nd lw is the new datatype name given to datatype char…

```
id x,y,z;
wt p,m;
lw a,b;
```

x,y and z are variable names that are declared to hold int datatype.

Example:

```
main()
{
typedef int Prasad;
Prasad a,b,c;
a=10;b=20;
c=a+b;
printf("%d",c);
}
```

In structures

```
#include<stdio.h>
typedef struct point
{
int x;
int y;
}data;
main()
{
data lt,rt;
printf("enter x and y coordinates of left and right");
scanf("%d%d%d%d",&lt.x,&lt.y,&rt.x,&rt.y);
printf("left:x=%d,y=%d.right:x=%d,y=%d",lt.x,lt.y,rt.x,rt.y);
getch();
}
```

→ Union:

Union is the user defined datatype that can hold heterogeneous datatypes. Unions are construct with "union" keyword. Unions are end with semicolon.

Syntax:

```
union unionname
{
Member1;
Member2;
.
.
.
Member n;
}variable1,variable2…..variable n;
```

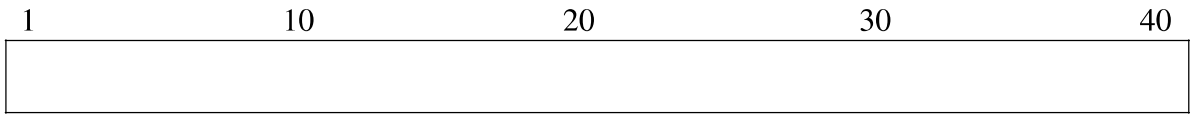
Union also has a union name,members and variable names. The union variables are declared with following syntax.

Union unionname variable1,variable2….variable n;

Example:

```
union mixed
  {
char ch[10];
 float marks[10];
  int no[10];
  }all;
```

Union members share the same storage



-------char----------
10bytes
----------int 20 bytes----------

----------float 40 bytes----------

→ Accessing and initializing members of union:

Syntax:

```
union unionname
{
 Member1;
  Member2;
    .
    .
    .
  Member n;
  }variable1,variable2….variable n;
```

For accessing the member of union using the following syntax.
unionvariable.member of union;
For initialization of union member is
unionvariable.member=constant;

Example:

```
#include<stdio.h>
#include<conio.h>
union exp
{
int i;
char c;
}var;
main()
{
```

```
var.i=65;
var.c=var.i;
printf("\n var.i=%d",var.i);
printf("\n var.c=%c",var.c);
getch();
}
```

→ Enumeration types:

Enumeration datatypes are data items whose values may be any number of a symbolically declared set of values the symbolically declared members are integer constants. The keyword enum is used to declare an enumeration type. Simply "enumerators are named integers"

The general form of the enumerator is

```
enum <enumerator name>
   {
    Member1;
     Member2;
       .
       .
       .
      Member n;
  }var1,var2,…var n;
```

The enum enumerator name specifies the userdefined type the member are integer constants, by default ,the first member, that is member1 is given the value0.

Example:

```
#include<stdio.h>
enum days
{
Mon,tue,wed,thur,fri,sat,sun
};
main()
{
enum days start,end;
start=tue;
end=sat;
printf("start=%d,end=%d",start,end);
start=64;
printf("\n start now is equal to %d",start);
getch();
}
```

Example:

```
#include<stdio.h>
enum rank
{
Ramu,ramesh,Prasad=10,vinaya,rani
}v;
main()
{
v=ramesh;
printf("ramesh rank is %d\n",r);
v=Prasad;
printf("prasad rank is %d\n",r);
v=rani;
printf("\n rani rank is %d",r);
getch();
}
```

→ Self referential structure:

A self referential structure is one which contains a pointer to it own type. This type of structure is also known as linked list. For example..

```
struct node
{
 int data;
 struct node *nextptr;
 };
```

Defines a datatype, struct node. A structured of type struct node has two members-integer member data and pointer member nextptr. Member nextptr points to a structure of type struct node-a structure of the same type as one being declared here, hence the term "self referential structure" member nextptr is refered to as link i.e nextptr can be used to "tie" a structure of type struct node to another structure of the sametype.

Example:

```
#include<stdio.h>
main()
{
struct stinfo
{
char name[20];
int age;
char city[20];
struct stinfo *next;
};
struct stinfo s1={"rani",25,"kkd"};
struct stinfo s2={"raju",27,"rjy"};
struct stinfo s3={"sita",29,"amp"};
s1.next=&s2;
```

```
s2.next=&s3;
s3.next=NULL;
printf("student name=%s,age=%d,city=%s",s1.name,s2.age,s3.city);
printf("student1 stored at %x \n",s1.next);
printf("student2 name=%s,age=%d,city=%s",s2.name,s2.age,s2.city);
printf("student2 stored at %x\n",s2.next);
printf("student3 name=%s,age=%d,city=%s",s3.name,s3.age,s3.city);
printf("student3 stored at %x\n",s3.next);
}
```

The next three statement in the program

```
s1.next=&s2;
s2.next=&s3;
s3.next=NULL;
```

Setup the linked list , with the next member of s1 pointing to s2 and the next member of s2 pointing to s3 and next member of s3 pointing to NULL.

→ Bitfields:

There are two ways to manipulate bits in c. one of the ways consists of using bitwise operators . the other way consists of using bit fields in which the definition and the access method are based on structure.

```
struct bitfield_tag
{
  unsigned int member1:bit_width1;
  unsigned int member2:bit_width2;
    .
    .
    .
  unsigned int member n:bit_width n;
};
```

Each bitfield for example 'unsigned int member1:bit_width1' is an integer that has a specified bit width. By this technique the exact number of bits required by the bit field is specified.

Exaample:

```
struct test
    {
      unsigned tx:2;
      unsigned rx:2;
      unsigned chk_sum:3;
      unsigned p:1;
    } status_byte;
```

Structure variable name status_byte containing four unsigned bitfields. The value assigned to a field must not be greater than its maximum storage value.

The assignment of the structure member is

```
Chk_sum=6;
```

Sets the bits in the field chk_sum on 110.

→ Difference between structure and union:

Structure	Union
1. structure is the user defined datatype . 2. structure are constructed with struct keyword. 3. structures members are stored in individual storage. 4. occupied more memory. 5. member accessing is difficult.	1. union is the user defined datatype. 2. union are construct with union keyword. 3. members share the same memory. 4. occupied less memory. 5. member accessing is easy.

→ Difference between array and pointers:

Array	Pointer
1. Array allocates space automatically. 2. It cannot be resized. 3. It cannot be reassigned. 4. sizeof(arrayname) gives the number of bytes occupied by array.	1. It is explicitly assigned to point to an allocated space. 2. It can be resized using realloc(). 3. It can be reassigned. 4. sizeof(p) returns the number of bytes used to store the point variable.

→ Difference between array and structure:

Array	Structure
1. Group of homogeneous elements. 2. Array is derived datatype. 3. An array behavior like a built in datatype. 4. Elements differ with index. 5. sizeof(arrayname) gives the number of bytes occupied by the array.	1. Group of heterogeneous elements. 2. Structure is a user defined datatype. 3. Structure behaves like a built in different datatypes. 4. members differ with datatype and member name. 5. sizeof(structurevariable) returns the number of bytes used to store the structure variable.

File management in C

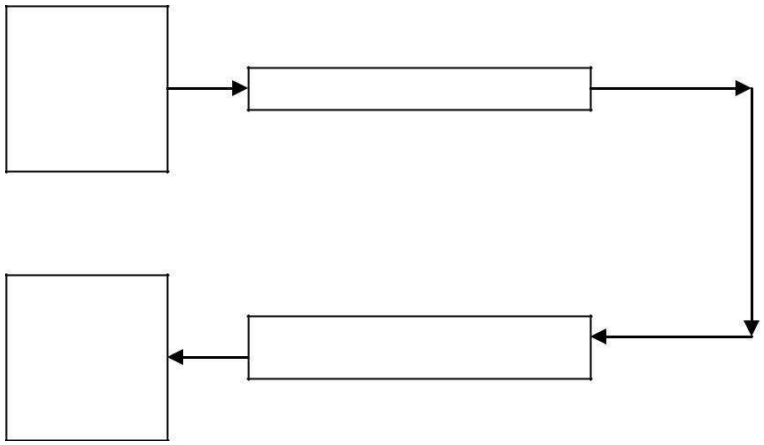
FILE:

A **file** is an external collection of related data treated as a unit. The primary purpose of a file is to keep a record of data. files are stored in auxiliary or secondary storage devices .

Buffer is a temporary storage area that holds data while they are being transferred to or from memory.

Streams

A stream can be associated with a physical device, such as a terminal, or with a file stored in auxiliary memory.



Text and Binary Streams

C uses two types of streams: text and binary. A **text stream** consists of characters divided into lines with each line terminated by a newline(\n).A **binary stream** consists of data values such as integer, real or complex.

Creating a Stream

We create a stream when we declare it. The declaration uses the FILE type as shown in the following example the FILE type is a structure that contains the information needed for reading and writing a file.

```
FILE *spData;
```

In this example ,spData is a pointer to the stream.

Opening a File

When the file is opened , the stream and the file are associated with each other , and the FILE type is filled with the pertinent file information. the open function returns the address of the file type ,which is stored in the stream pointer variable, spData .

Using the Stream Name

After we create the stream ,we can use the stream pointer in all functions that need to access the corresponding file for input and output.

Closing the Stream

When the File processing is complete, we close the file. Closing the association between the stream name and file name . After the close, the stream is no longer available and any attempt to use it results in an error

The File Modes

Your program must open a file before it can access it. This is done using the fopen function, which returns the required file pointer. If the file cannot be opened for any reason then the value NULL will be returned. You will usually use fopen as follows
if ((output_file = fopen("output_file", "w")) == NULL) fprintf(stderr, "Cannot open %s\n", "output_file");
fopen takes two arguments, both are strings, the first is the name of the file to be opened, the second is an access character, which is usually one of r, a or w etc. Files may be opened in a number of modes, as shown in the following table.

File Modes	
r	Open a text file for reading.
w	Create a text file for writing. If the file exists, it is overwritten.
a	Open a text file in append mode. Text is added to the end of the file.
rb	Open a binary file for reading.
wb	Create a binary file for writing. If the file exists, it is overwritten.
ab	ary file in append mode. Data is added to the end of the file.
r+	Open a text file for reading and writing.
w+	Create a text file for reading and writing. If the file exists, it is overwritten.
a+	Open a text file for reading and writing at the end.
r+b or rb+	Open binary file for reading and writing.
w+b orwb+	Create a binary file for reading and writing. If the file exists, it is overwritten.
a+b or ab+	Open a text file for reading and writing at the end.

The update modes are used with fseek, fsetpos and rewind functions. The fopen function returns a file pointer, or NULL if an error occurs.
The following example opens a file, tarun.txt in read-only mode. It is good programming practice to test the file exists.

```
if ((in = fopen("tarun.txt", "r")) == NULL)
{
puts("Unable to open the file");
return 0;
}
```

File operation functions in C

File operation functions in C, Defining and opening a file, Closing a file, The getw and putw functions, The fprintf & fscanf functions

C supports a number of functions that have the ability to perform basic file operations, which include:

1. Naming a file
2. Opening a file
3. Reading from a file
4. Writing data into a file
5. Closing a file

- Real life situations involve large volume of data and in such cases, the console oriented I/O operations create two major problems
- It becomes burden and time consuming to handle large volumes of data through terminals.
- The entire data is lost when either the program is terminated or computer is turned off.

File operation functions in C:

Function Name	Operation
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use
getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the begining of the file

Defining and opening a file:

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the fielname, data structure, purpose.

The general format of the function used for opening a file is

```
FILE *fp;
fp=fopen("filename","mode");
```

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening the file. The mode does this job.

R open the file for read only.
W open the file for writing only.
A open the file for appending data to it.

Consider the following statements:

```
FILE *p1, *p2;
p1=fopen("data","r");
p2=fopen("results","w");
```

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

Closing a file:
The input output library supports the function to close a file; it is in the following format.
fclose(file_pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer. Observe the following program.

```
….
FILE *p1 *p2;
p1=fopen ("Input","w");
p2=fopen ("Output","r");
….
…
fclose(p1);
fclose(p2)
```

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode.

```
c=getc(fp2).
```

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include< stdio.h >
main()
{
file *f1;
printf("Data input output");
f1=fopen("Input","w"); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key board*/
putc(c,f1); /*write a character to input*/
fclose(f1); /*close the file input*/
printf("nData outputn");
f1=fopen("INPUT","r"); /*Reopen the file input*/
while((c=getc(f1))!=EOF)
printf("%c",c);
fclose(f1);
}
```

Reading & Writing Files

The getw and putw functions:

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp);
getw(fp);
/*Example program for using getw and putw functions*/
#include< stdio.h >
main()
{
FILE *f1,*f2,*f3;
int number I;
printf("Contents of the data filenn");
f1=fopen("DATA","W");
for(I=1;I< 30;I++)
{
scanf("%d",&number);
if(number==-1)
break;
putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopen("EVEN","w");
while((number=getw(f1))!=EOF)/* Read from data file*/
```

```
{
if(number%2==0)
putw(number,f3);/*Write to even file*/
else
putw(number,f2);/*write to odd file*/
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
printf("nnContents of the odd filenn");
while(number=getw(f2))!=EOF)
printf("%d%d",number);
printf("nnContents of the even file");
while(number=getw(f3))!=EOF)
printf("%d",number);
fclose(f2);
fclose(f3);
}
```

The fprintf & fscanf functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,"control string", list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

```
fprintf(f1,%s%d%f",name,age,7.5);
```

Here name is an array variable of type char and age is an int variable

The general format of fscanf is

```
fscanf(fp,"controlstring",list);
```

This statement would cause the reading of items in the control string.

Example:

```
#include <stdio.h>
int main()
{
FILE *fp;
file = fopen("file.txt","w");
fprintf(fp,"%s","This is just an example :)");
fclose(fp);
return 0;
}
```

Variables

Variables defined in the stdio.h header include:

Name	Notes
stdin	a pointer to a FILE which refers to the standard input stream, usually a keyboard.
stdout	a pointer to a FILE which refers to the standard output stream, usually a display terminal.
stderr	a pointer to a FILE which refers to the standard error stream, often a display terminal.

fflush

Defined in header <stdio.h>

int fflush(FILE *stream);

Causes the output file stream to be synchronized with the actual contents of the file. If the given stream is of the input type, then the behavior of the function is undefined.

Parameters

stream	the file stream to synchronize

Return value

Returns zero on success. Otherwise EOF is returned and the error indicator of the file stream is set.

Text Files in C

A file is for storing permanent data. C provides file operations in stdio.h. A file is viewed as a stream of characters. Files must be opened before being accessed, and characters can be read one at a time, in order, from the file.

There is a current position in the file's character stream. The current position starts out at the first character in the file, and moves one character over as a result of a character read (or write) to the file; to read the 10th character you need to first read the first 9 characters (or you need to explicitly move the current position in the file to the 10th character).

There are special hidden chars (just like there are in the stdin input stream), '\n', '\t', etc. In a file there is another special hidden char, EOF, marking the end of the file.

Using text files in C

1 DECLARE a FILE * variable

```
FILE *infile;
FILE *outfile;
```

2 OPEN the file: associate the variable with an actual file using fopen you can open a file in read, "r", write, "w", or append, "a" mode

```
infile = fopen("input.txt", "r");
if (infile == NULL) {
  exit(1);
        Error("Unable to open file.");
}
outfile = fopen("/home/newhall/output.txt", "w");
if (outfile == NULL) {
        Error("Unable to open file.");
}
```

3 USE I/O operations to read, write, or move the current position in the file

```
int ch;
ch = getc(infile);
putc(ch, outfile);
```

4 CLOSE the file: use fclose to close the file after you are done with it

```
fclose(infile);
fclose(outfile);
```

You can also move the current file position in a file:

```
   void rewind(FILE *f);
  rewind(infile);
    fseek(FILE *f, long offset, int whence);
  fseek(f, 0, SEEK_SET);
  fseek(f, 3, SEEK_CUR);
  fseek(f, -3, SEEK_END);
```

Sequential and Random Access File Handling in C

A file handling in C tutorial detailing the use of sequential and random access files in C, along with examples of using the fseek, ftell and rewind functions. In computer programming, the two main types of file handling are:

- Sequential;
- Random access.

Sequential files are generally used in cases where the program processes the data in a sequential fashion – i.e. counting words in a text file – although in some cases, random access can be affected by moving backwards and forwards over a sequential file.

True random access file handling, however, only accesses the file at the point at which the data should be read or written, rather than having to process it sequentially. A hybrid approach is also possible whereby a part of the file is used for sequential access to locate something in the random access portion of the file, in much the same way that a File Allocation Table (FAT) works.

The three main functions that this article will deal with are:

- rewind() – return the file pointer to the beginning;
- fseek() – position the file pointer;
- ftell() – return the current offset of the file pointer.

Each of these functions operates on the C file pointer, which is just the offset from the start of the file, and can be positioned at will. All read/write operations take place at the current position of the file pointer.

The rewind() Function

The rewind() function can be used in sequential or random access C file programming, and simply tells the file system to position the file pointer at the start of the file. Any error flags will also be cleared, and no value is returned.

While useful, the companion function, fseek(), can also be used to reposition the file pointer at will, including the same behavior as rewind().

Using fseek() and ftell() to Process Files

The fseek() function is most useful in random access files where either the record (or block) size is known, or there is an allocation system that denotes the start and end positions of records in an index portion of the file. The fseek() function takes three parameters:

- FILE * f – the file pointer;
- long offset – the position offset;
- int origin – the point from which the offset is applied.

The origin parameter can be one of three values:

- SEEK_SET – from the start;
- SEEK_CUR – from the current position;
- SEEK_END – from the end of the file.

So, the equivalent of rewind() would be:

fseek(f, 0, SEEK_SET);

By a similar token, if the programmer wanted to append a record to the end of the file, the pointer could be repositioned thus:

fseek(f, 0, SEEK_END);

Since fseek() returns an error code (0 for no error) the stdio library also provides a function that can be called to find out the current offset within the file:

long offset = ftell(FILE * f)

This enables the programmer to create a simple file marker (before updating a record for example), by storing the file position in a variable, and then supplying it to a call to fseek:

```
long file_marker = ftell(f);
// … file processing functions
fseek( f, file_marker, SEEK_SET);
```

Of course, if the programmer knows the size of each record or block, arithmetic can be used. For example, to rewind to the start of the current record, a function call such as the following would suffice:

fseek(f, 0 – record_size, SEEK_CURR);

With these three functions, the C programmer can manipulate both sequential and random access files, but should always remember that positioning the file pointer is absolute. In other words, if fseek is used to position the pointer in a read/write file, then writing will overwrite existing data, permanently.