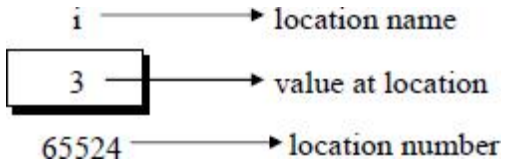
Unit Vth
An Introduction to Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

Pointer Notation Consider the
declaration, int i = 3 ;
This declaration tells the C compiler to:
(a) Reserve space in memory to hold the integer value.
(b) Associate the name **i** with this memory location.
(c) Store the value 3 at this location.

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



We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.
We can print this address number through the following program:

main( )
{
int i = 3 ;
printf ("\nAddress of i = %u", &i) ;
printf ("\nValue of i = %d", i) ;
}
	The output of the above program would be:
	Address of i = 65524
	Value of i = 3

Look at the first **printf( )** statement carefully. '&' used in this statement is C's 'address of' operator. The expression **&i** returns the address of the variable **i**, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer.

We have been using the '&' operator all the time in the **scanf( )** statement. The other pointer operator available in C is '*****', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program: main( )
{
int i = 3 ;
printf ("\nAddress of i = %u", &i) ;
printf ("\nValue of i = %d", i) ;
printf ("\nValue of i = %d", *(&i)) ;
}
	The output of the above program would be:
	Address of i = 65524
	Value of i = 3
	Value of i = 3

Note that printing the value of ***(&i)** is same as printing the value of **i**. The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying, j = &i ;

Here is a program another program. main( )
{
int i = 3 ;
int *j ; j = &i ;
printf ("\nAddress of i = %u", &i) ;
printf ("\nAddress of i = %u", j) ;
printf ("\nAddress of j = %u", &j) ;
printf ("\nValue of j = %u", j) ;
printf ("\nValue of i = %d", i) ;
printf ("\nValue of i = %d", *(&i)) ;
printf ("\nValue of i = %d", *j) ;
}
	The output of the above program would be:
	Address of i = 65524
	Address of i = 65524
	Address of j = 65522
	Value of j = 65524
	Value of i = 3
	Value of i = 3
	Value of i = 3

Look at the following declarations, int *alpha ;
char *ch ; float *s ;

Here, **alpha, ch** and **s** are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers.

Now we can put these two facts together and say—pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers.

The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value. Or in other words, the value at address stored in **ch** is going to be a **char**.

Consider the following example:	Here is the output of the program.

main( )
{
int i = 3, *x ;
float j = 1.5, *y ;
char k = 'c', *z ;
printf ("\nValue of i = %d", i) ;
printf ("\nValue of j = %f", j) ;
printf ("\nValue of k = %c", k) ;
x = &i ;
y = &j ;
z = &k ;
printf ("\nOriginal address in x = %u", x) ;
printf ("\nOriginal address in y = %u", y) ;
printf ("\nOriginal address in z = %u", z) ;
x++ ;
y++ ;
z++ ;
printf ("\nNew address in x = %u", x) ;
printf ("\nNew address in y = %u", y) ;
printf ("\nNew address in z = %u", z) ;
}
	Value of i = 3
	Value of j = 1.500000
	Value of k = c
	
	Original address in x = 65524
	Original address in y = 65520
	Original address in z = 65519
	
	New address in x = 65526
	New address in y = 65524
	New address in z = 65520

Observe the last three lines of the output. 65526 is original value in **x** plus 2, 65524 is original value in **y** plus 4, and 65520 is original value in **z** plus 1. This so happens because every time a pointer is incremented it points to the immediately next location of its type.

That is why, when the integer pointer **x** is incremented, it points to an address two locations after the current location, since an **int** is always 2 bytes long (under Windows/Linux since **int** is 4 bytes long, new value of **x** would be 65528).

Similarly, **y** points to an address 4 locations after the current location and **z** points 1 location after the current location. This is a very important result and can be effectively used while passing the entire array to a function.

The way a pointer can be incremented, it can be decremented as well, to point to earlier locations. Thus, the following **operations can be performed on a pointer**:

(a) Addition of a number to a pointer. For example,
int i = 4, *j, *k ; j = &i ; j = j + 1 ; j = j + 9 ; k = j + 3 ;

(b) Subtraction of a number from a pointer. For example,
j = &i ; j = j - 2 ; j = j - 5 ; k = j - 6 ;

(c) Subtraction of one pointer from another.

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements. This is illustrated in the following program.

```
main( ) {
 int arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
 int *i, *j ;
```

```
i = &arr[1] ; j = &arr[5] ;
printf ( "%d %d", j - i, *j - *i ) ;
 }
```

Here **i** and **j** have been declared as integer pointers holding addresses of first and fifth element of the array respectively.

Suppose the array begins at location 65502, then the elements **arr[1]** and **arr[5]** would be present at locations 65504 and 65512 respectively, since each integer in the array occupies two bytes in memory. The expression **j - i** would print a value 4 and not 8. This is because **j** and **i** are pointing to locations that are 4 integers apart. What would be the result of the expression ***j - *i**? 36, since ***j** and ***i** return the values present at addresses contained in the pointers **j** and **i**.

(d) Comparison of two pointer variables

Pointer variables can be compared provided both variables point to objects of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The comparison can test for either equality or inequality. Moreover, a pointer variable can be compared with zero (usually expressed as NULL). The following program illustrates how the comparison is carried out.

```
main( )
{ int arr[ ] = { 10, 20, 36, 72, 45, 36 } ; int *j, *k ;
j = &arr [ 4 ] ; k = ( arr + 4 ) ;
if ( j == k )
printf ( "The two pointers point to the same location" ) ;
else
printf ( "The two pointers do not point to the same location" ) ;
 }
```

****A word of caution!** Do not attempt the following operations on pointers... they would never work out.

(a) Addition of two pointers

(b) Multiplication of a pointer with a constant

VOID POINTER

- Void pointer can hold any type of values in a c program.
- Void pointer variable store multiple type of values in to a single variable.
- Void pointer in C is a pointer which is not associate with any data types.
- It points to some data location in storage means points to the address of variables.
- It is also called general purpose pointer.
- Pointer arithmetic is not possible of void pointer due to its concrete size.

Void pointer Problem **main( )** **{** **int i=3,*x;** **float j=1.5,*y;** **char k='c',*z ;** **x = &i ;** **y = &j ;** **z = &k ;** **printf("\n** **= %d%f, %c",x,y,z);** **}**	**int a=10;** **float b=20.5;** **void *a** **char ch='a';** **int * a1** **Float *b1** **Char *c1** **a=&a** **a=&b** **a=&c**	**#include<stdlib.h>** **void main()** **{** **int a = 7; float b = 7.6;** **void *p;** **p = &a;** **printf("Integer variable is = %d",** ***((int*) p));** **p = &b;** **printf("\nFloat variable is = %f",** ***((float*) p));** **}**

Pointers and Arrays

- Array name is a constant pointer that points to the base address of the array[i.e the first element of the array].
- Elements of the array are stored in contiguous memory locations. They can be efficiently accessed by using pointers.
- Pointer variable can be assigned to an array.
- The address of each element is increased by one factor depending upon the type of data type. The factor depends on the type of pointer variable defined. If it is integer the factor is increased by 2
- Int a[];
- It display the error size is not define
- Int *a[]
- Not display any error but create array a dynamically.

void main()
{
int a[6]={10, 20, 30, 40, 50, 60};
int *p; int i;
p=a;
for(i=0;i<6;i++)
{
printf("%d", *p); //value of elements of array
*p++;}
	main( ) {
	int arr[] = { 10, 20, 30, 45, 67, 56, 74 };
	int *i, *j;
	i = &arr[1]; j = &arr[5];
	printf("%d %d", j-i, *j-*i);
	}
	Answer add(4),36

main(){
int arr[]={10,20,36,72,45,36};
int*j,*k;
j=&arr[4]; address of 45
k=(arr+4); address of 45
if(j==k)
printf("Same locations");
else printf("Different locations");
}
Ans : Same locations
	Sum of elements in the Array using pointer
	void main() {
	int a[10]; int i,sum=0; int *ptr;
	printf("Enter 10 elements:n");
	for(i=0;i<10;i++) { scanf("%d",&a[i]); }
	ptr = a;
	for(i=0;i<10;i++) {
	sum = sum + *ptr;
	ptr++; }
	printf("The sum of array elements is %d",sum); }

Different Parameter passing techniques (Difference between Call by value and call by reference)

- WE can pass two types of parameter passing techniques
  Call by value
  Call by reference (Address)

- In call by value variable is access on the value but in call by reference variable is access based on the address.
- In call by value value is copied from one variable to another variable but in call by reference Address is copied from one variable to another variable,
- In Call by value variable value is passed in to a function call whereas in call by reference, variable's address is passed into the function call as the actual parameter.
- A program without pointer is call by value whereas a program with pointer is call by reference.
- In call by value stack data structure is used but in call by reference heap data structure is used.

```
Call by value
Int a,b;
a = 10 ;
b = a;
Printf("%d%d",a,b);
a++;
printf("%d,d",a,b);
```

```
Call by Reference
Int a, *b;
a = 10 ;
b = &a;
Printf("%d%d",a,b);
a++;
printf("%d%d",a,b);
```

- In Call by value if we change the value of one variable then another variable value is not automatically change. but in call by reference if we change the value of base variable then another variable value is automatically change
- In above prog if we change a++ after printf
- Then in call by value a=11 and b=10
- But in call by reference a=11 and b=11

Call By Value

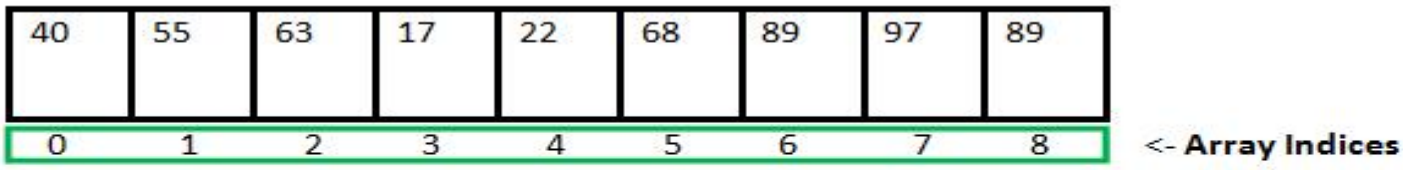
```
#include <stdio.h>
void swap(int , int);
void  main()
{ int a = 10; int b = 20;
 printf("Before swapped a = %d, b = %d\n",a,b);
swap(a,b);
}
void swap (int a, int b)
{ int temp;
 temp = a;
a=b;
b=temp;
printf("After swapped a = %d, b = %d\n",a,b);
}
```

Call By Reference

```
#include <stdio.h>
void swap(int *, int *);
Void  main()
{ int a = 10; int b = 20;
printf("Before swapped a = %d, b = %d\n",a,b);
swap(&a,&b);
}
void swap (int *a, int *b)
{
int  temp;
temp = *a; *a=*b; *b=temp;
 printf("After swapped a = %d, b = %d\n",*a,*b);
 }
```

Dynamic Memory allocation

- The process of allocating memory during program execution is called dynamic memory allocation.
- Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array.
- An array is collection of items stored at continuous memory locations



Array Length = 9
First Index = 0
Last Index = 8

- As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,
- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

DMA-

Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.

Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

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

malloc()

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

Syntax of malloc()

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

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

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

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

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

Syntax of calloc()

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

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

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

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

free()
Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

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

Here, ptr is reallocated with size of newsize.

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

Write a C program to find the sum of 'n' numbers using dynamically allocating memory

```
Void  main()
{
int i, count;  *arr[] , sum = 0;
printf("Enter the total number of elements you want to enter : ");
scanf("%d",&count);
arr = (int *)malloc(count * sizeof(int));
for(i = 0;i<count;i++ )  {
scanf("%d",arr+i);
}
for(i = 0;i<count;i++ ) {
sum += *(arr+i);
}
printf("sum is %d \n",sum);
free(arr);  }
```

Notion of linked list (no implementation)

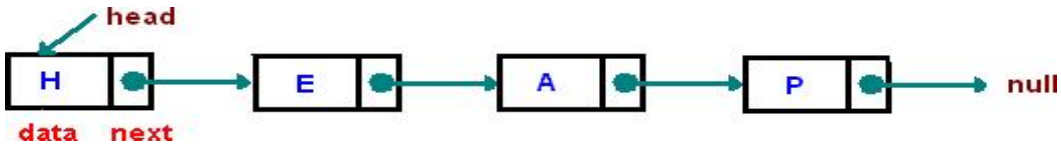
Introduction

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

Following are the limitation of array .

- We can not change the size of array.
- We can not insert element on any position in pre allocated array.
- We can not delete any data from entered array.
- We can not delete any position from the array.
- The size of the arrays is fixed.
- linked list has dynamic size whereas for array it is fixed size
- insertion and deletion is easy in linked list
- Linked List and Arrays are both used for storage of elements, but both are different techniques. In an array, elements are one after the another(successive memory allocation). But in linked list, memory is not contiguous.

A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

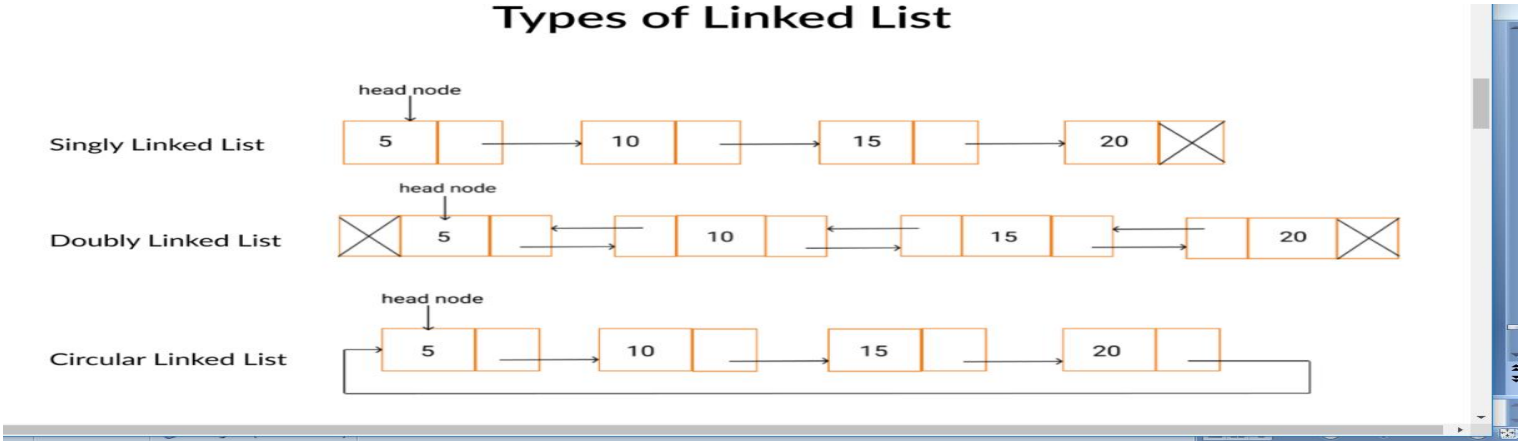
Types of Linked Lists

•Types of Linked List
•Following are the various types of linked list.
•**Singly Linked List** − one node hold the address of next node. Item navigation is forward only.
•**Doubly Linked List** − one node hold the address of previous node and next node both.

Items can be navigated forward and backward.

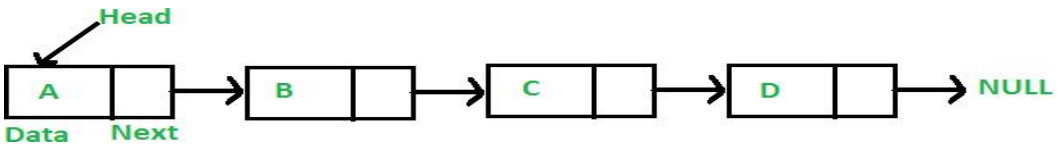
•**Circular Linked List** − Last node hold the address of 1st node not null. So list move in a circular way.

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



Applications of linked list data structure

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Applications of linked list in computer science –

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation: We use linked list of free blocks.
4. Performing arithmetic operations on long integers
5. Manipulation of polynomials by storing constants in the node of linked list
6. representing sparse matrices

File Handling-

- In C Programming when we store data .The data is stored in variables ,array .
- data is stored temporarily in nature.

After the execution of the program we can insert the new data in the variables and Arrays. Such type of data is normally stored in the main memory (RAM) of the computer and it will be lost when the program is terminated

- So to store the data permanently we can use the secondary storage device such as hard disk and magnetic tapes.
- For data storage in secondary storage device we used Files.
- In C Programming Files are used to stored data permanently in the hard disk. And we can insert the data and retrieve the data from a file to in our C Program.

A File is a collection of data stored on a secondary storage device like hard disk. File operation is to combine all the input data into a file and then to operate through the C program.

Various operations like insertion, deletion, opening closing etc can be done upon a file. When the program is terminated, the entire data is lost in C programming.

If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created these information can be accessed using few commands.

There are large numbers of functions to handle file I/O in C language. High level file I/O functions can be categorized as: Text file & Binary file

- **Text file** : A text file stored text information like alphabets,numbers,special symbols etc.The ASCII code of a text character is stored in the text file. For storage the data first ASCII data is converted into binary form than its stored in the secondary memory.
- **Binary file:** A Binary file stored the information in the binary form. In the same format as they stored in to the memory. The binary file eliminates the need of data conversion from text to binary format for storage purpose

File Modes-

A file can be open in several modes for these operations. The various modes are:

r **:** Open a text file for reading
w : truncate to zero length or create a text file for writing
a : append; open or create text file for writing at end-of-file
rb : open binary file for reading
wb truncate to zero length or create a binary file for writing
r+ open text file for update (reading and writing)
w+ truncate to zero length or create a text file for update a+ append; open or create text file for update

File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Creating a new file & Opening an existing file

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

- fp is declared as a pointer to the data type FILE.
- filename is a string - specifies the name of the file.
- fopen returns a pointer to the file which is used in all subsequent file operations.
- mode is a string which specifies the purpose of opening the file:

Q1. Write a program to open a file using fopen().

Ans:

```
#include<stdio.h> void main()
{
fopen() file *fp;
fp=fopen("student.DAT", "r");
if(fp==NULL)
{
printf("The file could not be open"); exit(0);
}
```

Write the content in a file Example

```
#include <stdio.h>
void main()
{
    FILE  *fp;
    char ch ;
    fp = fopen( "prog.txt", "w" );
While(1)
            {
Printf("Enter any character");
getc(          ch);
putc( ch,fp );
    }
    fclose( fp );
}
```

Q.3. Write a program to read data from file and close using fclose function. Ans:

```
#include <stdio.h>
Void  main()
{
int n
FILE *fptr;
if ((fptr=fopen("C:\\program.txt","r"))==NULL){ printf("Error! opening file");
exit(1); // Program exits if file pointer returns NULL.
}
fscanf(fptr,"%d",&n);
printf("Value of n=%d",n);
fclose(fptr);
```

Count characters in a file

```
void main()
        {
        FILE  *fp;
        char ch , nc, nlines;
        char filename[40] ;
        nlines = 0 ;   nc = 0;
        printf("Enter file name: ");
        gets( filename );
        fp = fopen( filename, "r" );
        if ( fp == NULL )
        {
printf("Cannot open %s for reading \n", filename );
                exit(1);      /* terminate program */
        } ch = getc( fp ) ;
while (  ch != EOF )
        {
if ( ch  ==  '\n'  )
nlines++ ;
                nc++ ;
                ch = getc ( fp );
        }
        fclose( fp );
if ( nc != 0 ) {
printf("There are %d characters in %s \n", nc, file);
printf("There are %d lines \n", nlines );
        }
        else
printf("File: %s is empty \n", filename );
```

}	

Write a C program to Copy the content from one file in another file

```
#include <stdio.h>
void main()
{
FILE *fp1, *fp2 ;
char ch ;
fp1 = fopen( “prog.txt”, “r” ); /open for read /
fp2 = fopen( “result.txt”, “w” ) ;/*open for write */
if ( fp1 == NULL ) /* check does file
exist etc */ {
printf(“Cannot open prog.c for reading
\n” );
exit(1); /* terminate program */
} else if ( fp2 == NULL ) {
printf(“Cannot open prog.old for writing
\n”);
exit(1); /* terminate program */ }
else /* both files O.K. */
{
ch = getc(fp1) ;
while ( ch != EOF)
{
putc( ch, fp2); /* copy to result.txt */
ch = getc( fp1 ) ;
}
fclose ( fp1 ); /* Now close files
*/
fclose ( fp2 ); }
```

Q2. Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.

Ans:

```
#include <stdio.h>
int main()
{
char name[50]; int marks, i,n;
printf(“Enter number of students”);
scanf(“%d”, &n);
FILE *fptr; fptr=(fopen(“C:\\student.txt”,”a”));
if (fptr==NULL){
printf("Error!"); exit(1);
}
for(i=0;i<n;++i)
{ printf("For student%d\nEnter name: ",i+1);
scanf("%s",name);

printf(“Enter marks”);
scanf(“%d”, &marks);
fprintf(fptr, “\nName: %s\nMarks=%d\n”,
name, marks);
}
fclose(fptr);
}
```

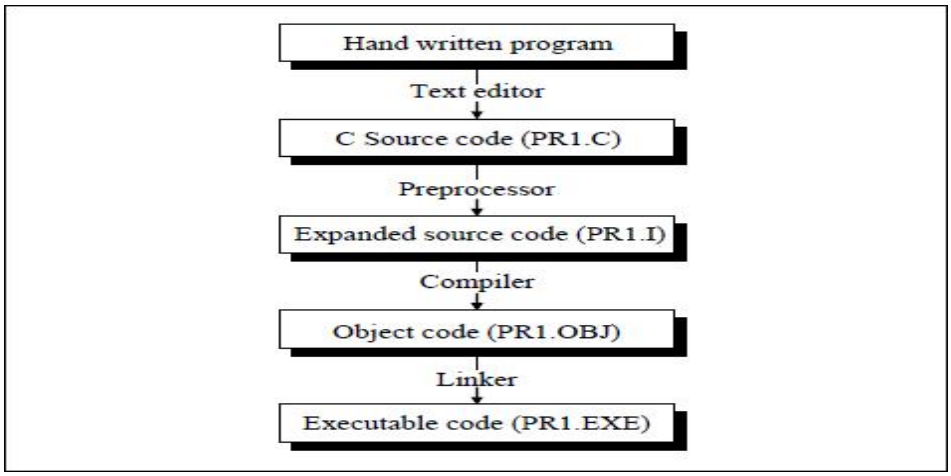
IIIIII----------

C Preprocessor Directives

- The preprocessor is a program that processes the source program before it is compiled.
- It is used for fast execution.
- These directives are executive before the C Program passes through the compiler.
- The C preprocessor directly substitutes the value or expression of a statement in source program so that the source now becomes an expended source program.

There are several steps involved from the stage of writing a C program to the stage of getting it executed.

Note that if the source code is stored in a file PR1.C then the expanded source code gets stored in a file PR1.I. When this expanded source code is compiled the object code gets stored in PR1.OBJ. When this object code is linked with the object code of library functions the resultant executable code gets stored in PR1.EXE.



- The #include and #define statements are preprocessor directives. They must start in the first column and no space is required between the sharp sign (#) and the directive.

- The directive is terminated not by a semicolon but by the end of the line on which it appears
- When you issue the command to compile a C program, the program is run automatically through the preprocessor.
- The preprocessor is a program that modifies the C source program according to directives supplied in the program.
- An original source program usually is stored in a file. The preprocessor does not modify this program file but creates a new file that contains the processed version of the program
- This new file is then submitted to the compiler

We would learn the following preprocessor directives here:

(a) Macro expansion
b) File inclusion
(c) Conditional Compilation

Let us understand these features of preprocessor one by one.

Macro Expansion

- Macro is a preprocessor directives . Its replacing an Identifier of a C Program by a constant or a symbolic constant .
- The #define is a macro definition statement
- #define Variable/Constant/Expression
- #define NUM 10
- #define Num (34.2-21.2)
  #define max(x,y) x>y?x:y

Have a look at the following program.

```
#define UPPER 25
main( )
{
int i ;
for ( i = 1 ; i <= UPPER ; i++ )
printf ( "\n%d", i ) ;
}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement,

#define UPPER 25

This statement is called 'macro definition' or more commonly, just a 'macro'. What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

Remember that a macro definition is never to be terminated by a semicolon.

Macros with Arguments

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
float r1 = 6.25, r2 = 2.5, a ;
a = AREA ( r1 ) ;
printf ( "\nArea of circle = %f", a ) ;
a = AREA ( r2 ) ;
printf ( "\nArea of circle = %f", a ) ;
```

}
Here's the output of the program...
Area of circle = 122.656250
Area of circle = 19.625000

In this program wherever the preprocessor finds the phrase **AREA(x)** it expands it into the statement **(3.14 * x * x)**. However, that's not all that it does. The **x** in the macro template **AREA(x)** is an argument that matches the **x** in the macro expansion **(3.14 * x * x)**. The statement **AREA(r1)** in the program causes the variable **r1** to be substituted for **x**. Thus the statement **AREA(r1)** is equivalent to:

Be careful not to leave a blank between the macro template and its argument while defining the macro. For example, there should be no blank between **AREA** and **(x)** in the definition, #define AREA(x) (3.14 * x* x)

Macros versus Functions

when is it best to use macros with arguments and when is it better to use a function? Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.

If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program.

But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.
Moral of the story is—if the macro is simple and sweet like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a function.

File Inclusion

The second preprocessor directive we'll explore in this chapter is file inclusion. This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:
#include "filename"
and it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program.

- For design and use user defined header file
- This is a process of inserting the external files containing functions or macro definitions into a C program.
- This process of eliminates the job of rewriting the functions or macro definitions.
- An external file containing functions can be include in a given C program using the #include directive**.**
- #include <stdio.h>

- #include <math.h>
- #include "Grade.h"
- Here stdio and math are the pre defined heafer file whereas Grade.h is a user defined header file.
-

File save cube.h	#include <stdio.h>
	#include "cube.h"
	void main()
	{
	int num,cu;
	printf("Enter any Number");
	scanf("%d",&num);
	cb= cube(num);
	printf("The cube is:%d",cb);
	}

int cube (int x)
{
return (x*x*x);
}	

Conditional Compilation

- Complete code is not compiled .
- code is compiled based on the basis of some conditions
- .if condition true then true block code is compiled and if condition is false then false block code is compiled.
- C preprocessors provide a conditional compilation directive which is used to select alternate segment of code in a C program depending upon the condition
  - Suppose there are two different version of a program and they are more alike then are different. It will be redundant if you maintain both versions. We can overcome this problem by simply including both versions in a single program. Then it would be possible to select a particular version depending on the requirements.
  - The #ifdef , #else , #elif ,#endif directives are used for conditional compilation of the program.

```
#ifdef macroname
statement 1 ;
statement 2 ;
statement 3 ;
#endif
```

If **macroname** has been **#define**d, the block of code will be processed as usual; otherwise not.

#*if* and #*elif* Directives

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expression is nonzero, then subsequent lines upto a **#else**, **#elif** or **#endif** are compiled, otherwise they are skipped. A simple example of **#if** directive is shown below:

```
main( )
{
#if TEST <= 5
statement 1 ;
statement 2 ;
statement 3 ;
```

```
#else
statement 4 ;
statement 5 ;
statement 6 ;
#endif
}
```

If the expression, **TEST <= 5** evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled.

.

Command Line Arguments in C –

So far, we have seen that no arguments were passed in the main function. But the C programming language gives the programmer the provision to add parameters or arguments inside the main function to reduce the length of the code. These arguments are called Command Line Arguments in C.
In this lecture , we will discuss:

- Components of Command Line Arguments
- C program to understand command line arguments

What are Command Line Arguments in C?

Command line arguments are simply arguments that are specified after the name of the program in the system's command line, and these argument values are passed on to your program during program execution.

Components of Command Line Arguments

There are 2 components of Command Line Argument in C:

1. **argc:** It refers to "argument count". It is the first parameter that we use to store the number of

command line arguments. It is important to note that the value of argc should be greater than or equal to 0.

2. **agrv:** It refers to “argument vector”. It is basically an array of character pointer which we use to list all the command line arguments.

In order to implement command line arguments, generally, 2 parameters are passed into the main function:

1. Number of command line arguments
2. The list of command line arguments

The basic **syntax** is:

```
int main( int argc, char *argv[] )
{
// BODY OF THE MAIN FUNCTION
}
```

Program to add two number by using the command line argument

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char * argv[]) {
  int i, sum = 0;
  if (argc != 3) {
     printf("You have forgot to specify two numbers.");
     exit(1);
  }
printf("The sum is : ");
sum= (argv[1])+(argv[2]);
printf("%d", sum);
return 0;
}
$./a.out 5 9
```

Here is a code in C that illustrates the use of command line arguments.