Pointers

Introduction

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

```c
#include <stdio.h>


int main ()
{
   int  var1;
   char var2[10];


   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );


   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

The above example shows what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

What Are 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 you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk

that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable**(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Benefit of using pointers

- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It reduces length and the program execution time.
- It allows C to support dynamic memory management.

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

int main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Declaring a pointer variable

General syntax of pointer declaration is,

```
data-type *pointer_name;
```

Data type of pointer must be same as the variable, which the pointer is pointing. **void** type pointer works with all data types, but isn't used oftenly.

Initialization of Pointer variable

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

```
int a = 10 ;
int *ptr ;        //pointer declaration
ptr = &a ;        //pointer initialization
or,
int *ptr = &a ;      //initialization and declaration together
```

Pointer variable always points to same type of data.

```
float a;

int *ptr;

ptr = &a;    //ERROR, type mismatch
```

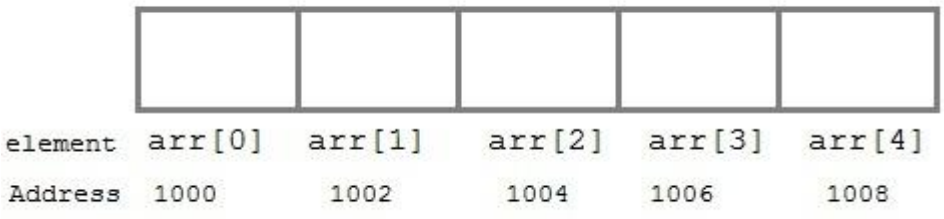
Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address which gives location of the first element is also allocated by the compiler.

Suppose we declare an array **arr**,

```
int arr[5]={ 1, 2, 3, 4, 5 };
```

Assuming that the base address of **arr** is 1000 and each integer requires two byte, the five element will be stored as follows



Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**. Therefore **arr** is containing the address of **arr[0]** i.e 1000.

```
arr is equal to &arr[0]   // by default
```

We can declare a pointer of type `int` to point to the array **arr**.

```
int *p;

p = arr;

or p = &arr[0];  //both the statements are equivalent.
```

Now we can access every element of array **arr** using **p++** to move from one element to another.

NOTE : You cannot decrement a pointer once incremented. `p--` won't work.

Pointer to Array

As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array. Lets have an example,

```
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a;  // same as int*p = &a[0]
for (i=0; i<5; i++)
{
 printf("%d", *p);
 p++;
}
```

In the above program, the pointer ***p** will print all the values stored in the array one by one. We can also use the Base address (**a** in above case) to act as pointer and print all the values.

Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

printf("%d", *a);⟶ **will print value of a[0] only**

a++;⟶ **Compile time error, we cannot change base address of the array.**

Pointer to Multidimensional Array

A multidimensional array is of form, `a[i][j]`. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In `a[i][j]`, **a** will give the base address of this array, even `a+0+0` will also give the base address, that is the address of **a[0][0]** element.

Here is the generalized form for using pointer with multidimensional arrays.

```
*(*(ptr + i) + j) is same as a[i][j]
```

Pointer and Character strings

Pointer can also be used to create strings. Pointer variables of **char** type are treated as string.

```
char *str = "Hello";
```

This creates a string and stores its address in the pointer variable **str**. The pointer **str** now points to the first character of the string "Hello". Another important thing to note that string created using **char** pointer can be assigned a value at **runtime**.

```
char *str;

str = "hello";   //thi is Legal
```

The content of the string can be printed using `printf()` and `puts()`.

```
printf("%s", str);

puts(str);
```

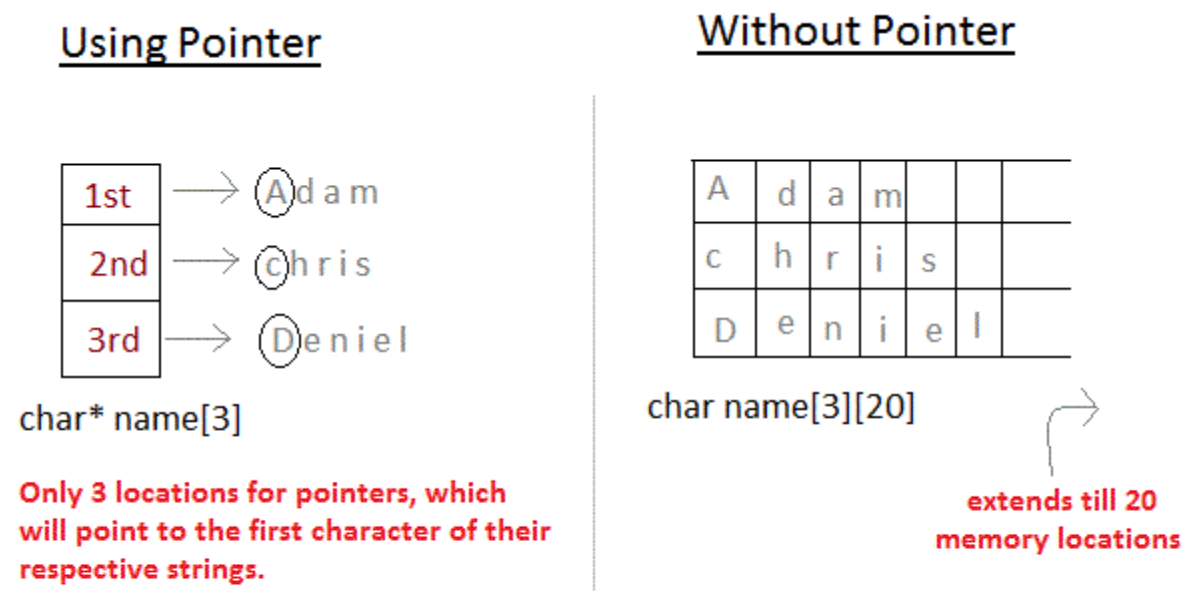
Notice that **str** is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator `*`.

Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3]={
                "Adam",
                "chris",
                "Deniel"
```

```
                };
//Now see same array without using pointer
char name[3][20]= {
                    "Adam",
                    "chris",
                    "Deniel"
                  };
```



In the second approach memory wastage is more, hence it is prefered to use pointer in such cases.

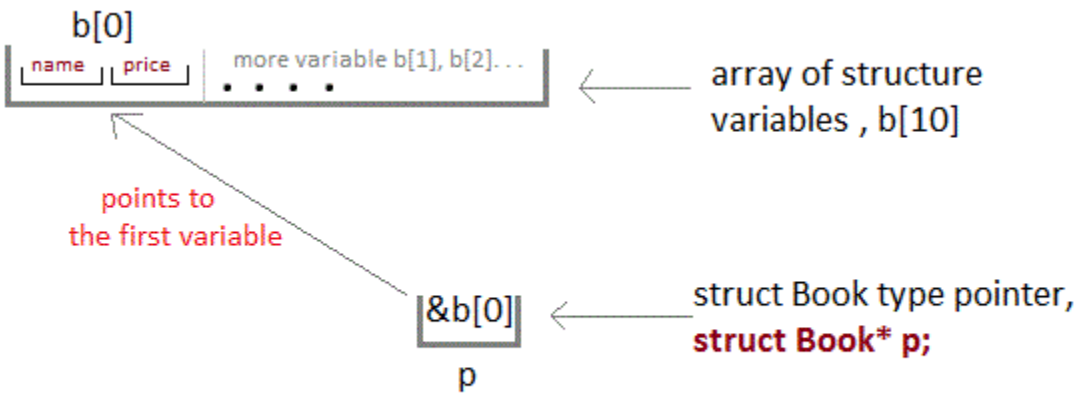
Pointer to Structure

Like we have array of integers, array of pointer etc, we can also have array of structure variables. And to make the use of array of structure variables efficient, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used with array of structure variables.

```
struct Book
{
 char name[10];
 int price;
}

int main()
{
 struct Book a;       //Single structure variable
 struct Book* ptr;    //Pointer of Structure type
 ptr = &a;

 struct Book b[10];     //Array of structure variables
 struct Book* p;        //Pointer of Structure type
 p = &b;
}
```



Accessing Structure Members with Pointer

To access members of structure with structure variable, we used the dot `.` operator. But when we have a pointer of structure type, we use arrow `->` to access structure members.

```
struct Book
{
 char name[10];
 int price;
}

int main()
{
 struct Book b;
 struct Book* ptr = &b;
 ptr->name = "Dan Brown";      //Accessing Structure Members
 ptr->price = 500;
}
```

Pointers can be accessed along with structures. A pointer variable of structure can be created as below:

```
struct name {

    member1;

    member2;

    .

    .

};

-------- Inside function -------

struct name *ptr;
```

Here, the pointer variable of type **struct name** is created.

Structure's member through pointer can be used in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

Consider an example to access structure's member through pointer.

```
#include <stdio.h>
struct name{
  int a;
  float b;
};
int main(){
   struct name *ptr,p;
   ptr=&p;          /* Referencing pointer to memory address of p */
   printf("Enter integer: ");
   scanf("%d",&(*ptr).a);
   printf("Enter number: ");
   scanf("%f",&(*ptr).b);
   printf("Displaying: ");
   printf("%d%f",(*ptr).a,(*ptr).b);
   return 0;
}
```

In this example, the pointer variable of type **struct name** is referenced to the address of *p*. Then, only the structure member through pointer can can accessed.

Structure pointer member can also be accessed using -> operator.

```
(*ptr).a is same as ptr->a

(*ptr).b is same as ptr->b
```

Accessing structure member through pointer using dynamic memory allocation

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.

Syntax to use malloc()

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

Example to use structure's member through pointer using malloc() function.

```
#include <stdio.h>
#include<stdlib.h>
struct name {
   int a;
   float b;
   char c[30];
};
int main(){
   struct name *ptr;
   int i,n;
   printf("Enter n: ");
   scanf("%d",&n);
   ptr=(struct name*)malloc(n*sizeof(struct name));
/* Above statement allocates the memory for n structures with pointer ptr pointing to base address
*/
   for(i=0;i<n;++i){
      printf("Enter string, integer and floating number  respectively:\n");
      scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);
   }
   printf("Displaying Infromation:\n");
   for(i=0;i<n;++i)
      printf("%s\t%d\t%.2f\n",(ptr+i)->c,(ptr+i)->a,(ptr+i)->b);
   return 0;
}
```

Output

```
Enter n: 2

Enter string, integer and floating number  respectively:

Programming

2

3.2

Enter string, integer and floating number  respectively:

Structure

6
```

```
2.3

Displaying Information

Programming      2      3.20

Structure      6      2.30
```

In C programming, file is a place on disk where a group of related data is stored.

Why files are needed?

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. In this tutorial, you will learn to handle standard I/O(High level file I/O functions) in C.

High level file I/O functions can be categorized as:

1. Text file
2. Binary file

File Operations

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

Working with file

While working with file, you need to declare a pointer of type file. This declaration is needed for communication between file and program.

```
FILE *ptr;
```

Opening a file

Opening a file is performed using library function fopen(). The syntax for opening a file in standard I/O is:

```
ptr=fopen("fileopen","mode")


For Example:

fopen("E:\\cprogram\program.txt","w");


/* ---------------------------------------------------------- */
 E:\\cprogram\program.txt is the location to create file.
 "w" represents the mode for writing.
/* ---------------------------------------------------------- */
```

Here, the program.txt file is opened for writing mode.

Opening Modes in Standard I/O		
File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns

Opening Modes in Standard I/O		
File Mode	Meaning of Mode	During Inexistence of file
		NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

Closing a File

The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose().

```
fclose(ptr); //ptr is the file pointer associated with file to be closed.
```

The Functions fprintf() and fscanf() functions.

The functions `fprintf()` and `fscanf()` are the file version of `printf()` and `fscanf()`. The only difference while using `fprintf()` and `fscanf()` is that, the first argument is a pointer to the structure FILE

Writing to a file

```
#include <stdio.h>
int main()
{
  int n;
  FILE *fptr;
  fptr=fopen("C:\\program.txt","w");
  if(fptr==NULL){
    printf("Error!");
    exit(1);
  }
  printf("Enter n: ");
  scanf("%d",&n);
  fprintf(fptr,"%d",n);
  fclose(fptr);
  return 0;
}
```

This program takes the number from user and stores in file. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open that file, you can see the integer you entered.

Similarly, `fscanf()` can be used to read data from file.

Reading from file

```
#include <stdio.h>
int 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);
  return 0;
}
```

If you have run program above to write in file successfully, you can get the integer back entered in that program using this program.

Other functions like `fgetchar()`, `fputc()` etc. can be used in similar way.

Binary Files

Depending upon the way file is opened for processing, a file is classified into text file and binary file.

If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used.

Working of binary files is similar to text files with few differences in opening modes, reading from file and writing to file.

Opening modes of binary files

Opening modes of binary files are `rb`, `rb+`, `wb`, `wb+`,`ab` and `ab+`. The only difference between opening modes of text and binary files is that, b is appended to indicate that, it is binary file.

Reading and writing of a binary file.

Functions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Function fwrite() takes four arguments, address of data to be written in disk, size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data,size_data,numbers_data,pointer_to_file);
```

Function `fread()` also take 4 arguments similar to `fwrite()` function as above.

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the beginning point

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 feigned 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.

fseek()

```
int fseek(FILE *stream, long offset, int whence);
```

The fseek() function is used to set the file position indicator for the stream to a new position. This function accepts three arguments. The first argument is the FILE stream pointer returned by the fopen() function. The second argument 'offset' tells the amount of bytes to seek. The third argument 'whence' tells from where the seek of 'offset' number of bytes is to be done. The available values for whence are SEEK_SET, SEEK_CUR, or SEEK_END. These three values (in order) depict the start of the file, the current position and the end of the file.

Upon success, this function returns 0, otherwise it returns -1.

ftell()

The C library function **long int ftell(FILE *stream)** returns the current file position of the given stream.

Declaration

Following is the declaration for ftell() function.

```
long int ftell(FILE *stream)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

Example

The following example shows the usage of ftell() function.

```c
#include <stdio.h>

int main ()
{
  FILE *fp;
  int len;

  fp = fopen("file.txt", "r");
  if( fp == NULL )
  {
    perror ("Error opening file");
    return(-1);
  }
  fseek(fp, 0, SEEK_END);

  len = ftell(fp);
  fclose(fp);

  printf("Total size of file.txt = %d bytes\n", len);

  return(0);
}
```

Program to copy contents of one file to another

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

void main() {
  FILE *fp1, *fp2;
  char a;
  clrscr();

  fp1 = fopen("test.txt", "r");
  if (fp1 == NULL) {
    puts("cannot open this file");
    exit(1);
```

```
  }

  fp2 = fopen("test1.txt", "w");
  if (fp2 == NULL) {
    puts("Not able to open this file");
    fclose(fp1);
    exit(1);
  }

  do {
    a = fgetc(fp1);
    fputc(a, fp2);
  } while (a != EOF);

  fcloseall();
  getch();
}
```

MODULE 4

POINTERS

Q. Differentiate pointer to an array and array of pointers with example. (7 marks) [Jun 2013, '12]
(The answer will carry 2.5 marks for pointer to an array, 1 mark for its example, 2.5 marks for array of pointers, 1 mark for its example)

Ans :

Pointer to an array

An array name is a constant pointer to the first element of the array. Therefore, in the declaration −

double balance[50];

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p as the address of the first element of balance −

*double *p;*

double balance[10];

p = balance; */ Here you can see pointer p to an array balance */*

Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on.

```
#include <stdio.h>
int main () {
   /* an array with 4 elements */
   double balance[4] = {1000.0, 2.0, 3.4, 17.0};
   double *p;
   int i;
   p = balance;
    /* output each array element's value */
   printf( "Array values using pointer\n");
   for ( i = 0; i < 4; i++ ) {
      printf("*(p + %d) : %f\n",  i, *(p + i) );
   }
   return 0;
}
```

The above code produces the following result −

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
```

Array of pointers

Just like array of integers or characters, there can be array of pointers too.

An array of pointers can be declared as :

*<type> *<name>[<number-of-elements];*

For example :

char *ptr[3];

The above line declares an array of three character pointers.

Consider the following example :

```
#include<stdio.h>
int main(void)
{
   char *p1 = "Himanshu";
   char *p2 = "Arora";
   char *p3 = "India";

   char *arr[3];              */ declaring an array that can hold pointer values */

   arr[0] = p1;               */ assigning a pointer to 0th index of the array */
   arr[1] = p2;
   arr[2] = p3;

  printf("\n p1 = [%s] \n",p1);
  printf("\n p2 = [%s] \n",p2);
  printf("\n p3 = [%s] \n",p3);

  printf("\n arr[0] = [%s] \n",arr[0]);
  printf("\n arr[1] = [%s] \n",arr[1]);
  printf("\n arr[2] = [%s] \n",arr[2]);

  return 0;
}
```

In the above code, we took three pointers pointing to three strings. Then we declared an array that can contain three pointers. We assigned the pointers ‘p1′, ‘p2′ and ‘p3′ to the 0,1 and 2 index of array.

Let’s see the output :

p1 = [Himanshu]

p2 = [Arora]

p3 = [India]

arr[0] = [Himanshu]

arr[1] = [Arora]

arr[2] = [India]

So we see that array now holds the address of strings.

Q. What are pointers to functions? Explain with examples. (7 marks) [Jun 2013,15, ‘12]
(What is pointer to function 2 marks, concept explained with example 5 marks)

AND

Q. Write a note on function pointers in C. (5 marks) [Jun 2013, ‘06]

Ans:

Pointer to function means a pointer which point to the address of a function. Just like pointer to characters, integers etc, we can have pointers to functions.

A function pointer can be declared as:

<return type of function> (<name of pointer>) (type of function arguments)*

For example :

*int (*fptr)(int, int)*

The above line declares a function pointer ‘fptr’ that can point to a function whose return type is ‘int’ and takes two integers as arguments.

Consider the following example :

```
#include<stdio.h>

int func (int a, int b)

{

   printf("\n a = %d\n",a);

   printf("\n b = %d\n",b);

   return 0;
```

```
}
int main(void)
{
   int(*fptr)(int,int); // Function pointer
   fptr = func; // Assign address to function pointer
   func(2,3);        // Normal function call
   fptr(2,3);        // Calling the function using function pointer
   return 0;
}
```

In the above example, we defined a function 'func' that takes two integers as inputs and returns an integer. In the main() function, we declare a function pointer 'fptr' and then assign value to it. Note that, name of the function can be treated as starting address of the function so we can assign the address of function to function pointer using function's name.

The output of the above program is:

a = 2

b = 3

a = 2

b = 3

So from the output we see that calling the function through function pointer produces the same output as calling the function from its name.

The pointer to function concept also has the advantage of passing a function to another function.

Q. Explain the importance of pointers in C Language. (5 marks) [Jun 2013, '12]

AND

Q. What are the advantages of using pointers? (5 marks) [Jun 2013, '06]

Ans:

- It provides a way of accessing a variable without referring to the variable directly by using address of variable.
- It can be used in passing arguments to function using call by reference method.
- Return more than one value from a function.
- Pass arrays and strings more conveniently from one function to another.
- It makes array manipulation easier.
- Used to create complex data structures like linked list and binary trees.
- Communicate information about free memory.

Q. What is a pointer? How it is initialized? Explain (5 marks) [Jun 2012, '06]

Ans:

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 you can use it to store any variable address. The general form of a pointer variable declaration is:

*type *var-name;*

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk (*) is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Pointer initialization is done as follows:

```
int  var = 20;   /* actual variable declaration */
int  *ip;        /* pointer variable declaration */
ip = &var;  /* store address of var in pointer variable*/ [This is the pointer initialization]
```

The few important operations done very frequently using pointers include:

(a) we define a pointer variable

(b) assign the address of a variable to a pointer and

(c) finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>
int main ()
{
  int  var = 20;   /* actual variable declaration */
  int  *ip;        /* pointer variable declaration */

  ip = &var;  /* store address of var in pointer variable*/ [This is the pointer initialization]

  printf("Address of var variable: %x\n", &var  );

  /* address stored in pointer variable */
  printf("Address stored in ip variable: %x\n", ip );
```

```
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of *ip variable: 20

Q. Write a program that uses a function pointer as a function argument. (7 marks) [Jun 2012, '06]

AND

Q. With the help of an example, describe how can we pass functions to other functions as arguments. (7 marks) [June 2011, '06]

Ans:

Write an algorithm for the below program:

Program

```
#include<stdio.h>
int func (int a, int b)
{
        int c;
        c=a+b;
        return c;
}

int compute(int x, int y)
{
        int z;
        z=x+y;
        return z;
}

int main(void)
{
```

```
    int(*fptr)(int,int); // Function pointer
    int sum;
    fptr = func; // Assign address to function pointer
    sum=compute(fptr(2,3),5);    // function pointer fptr coming as a function argument in compute() function call
    printf("Sum is: %d\n",sum);
    return 0;
}
```

Output

Sum is 10

The above program demonstrates the use of function pointer as a function argument. The function pointer in the above program is *fptr. It points to the function func(). The function pointer is used as an argument in the compute() function.

Q. Describe typical applications of pointers in developing programs. (8 marks) [June 2012, '06]

Ans:

Pointer Applications in C Programming

Pointer is used for different purposes. Pointer is low level construct in programming which is used to perform high level task. Some of the pointer applications are listed below

A. Passing Parameter by Reference

First pointer application is to pass the variables to function using pass by reference scheme.

```
void interchange(int *num1,int *num2)
{
    int temp;
    temp  = *num1;
    *num1 = *num2;
    *num2 = *num1;
}
```

Pointer can be used to simulate passing parameter by reference. Pointer is used to pass parameter to function. In this scheme we are able to modify value at direct memory location.

B. Accessing Array element

```
int main()
{
int a[5] = {1,2,3,4,5};
```

```
int *ptr;
ptr = a;

for(i=0;i<5;i++) {
 printf("%d",*(ptr+i));
}
return(0);
}
```

We can access array using pointer. We can store base address of array in pointer.

```
ptr = a;
```

Now we can access each and individual location using pointer.

```
for(i=0;i<5;i++) {
    printf("%d",*(ptr+i));
}
```

C. Dynamic Memory Allocation:

Another pointer application is to allocate memory dynamically.

We can use pointer to allocate memory dynamically. Malloc and calloc function is used to allocate memory dynamically.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  char *str;

  str = (char *) malloc(15);
  strcpy(str, "mahesh");

  printf("String = %s,  Address = %u\n", str, str);
  free(str);
    return(0);
}
```

Consider above example where we have used malloc() function to allocate memory dynamically.

D. Reducing size of parameter

```
struct student {
  char name[10];
```

```
  int rollno;
};
```

Suppose we want to pass the above structure to the function then we can pass structure to the function using pointer in order to save memory.

Suppose we pass actual structure then we need to allocate (10 + 4 = 14 Bytes(*)) of memory. If we pass pointer then we will require 4 bytes(*) of memory.

E. Some other pointer applications :

- Passing Strings to function
- Provides effective way of implementing the different data structures such as tree,graph,linked list

Q. What are the advantages of using pointers? Write the statements for declaring 1-D and 2-D arrays using pointers? (10 marks) [June 2011, '06]

(Advantages of pointers will carry 5 marks, declaring 1-D array 2.5 marks, declaring 2-D array 2.5 marks with example)

Ans:

Advantages of using pointers

- It provides a way of accessing a variable without referring to the variable directly by using address of variable.
- It can be used in passing arguments to function using call by reference method.
- Return more than one value from a function.
- Pass arrays and strings more conveniently from one function to another.
- It makes array manipulation easier.
- Used to create complex data structures like linked list and binary trees.
- Communicate information about free memory.

Declaring 1-D and 2-D Arrays using Pointers

1-D Array

```
#include <stdio.h>

#include <stdlib.h>

int main(){

   int n,i,*ptr,sum=0;

   printf("Enter number of elements: ");

   scanf("%d",&n);

   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc

   if(ptr==NULL)

   {
```

```
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

2-D Array

```
#include <stdio.h>
#include <stdlib.h>
double **read_matrix(int rows, int cols);
void print_matrix(int rows, int cols, double **mat);

double **read_matrix(int rows, int cols){
    double **mat = (double **) malloc(sizeof(double *)*rows);
    int i=0,j=0;
    for(i=0; i<rows; i++)
    /* Allocate array, store pointer  */
        mat[i] = (double *) malloc(sizeof(double)*cols);
       for(i=0; i<rows; i++){
           for(j=0; j<cols; j++){
               scanf("%lf",&mat[i][j]);
           }
```

```
    }
   return mat;
}
void print_matrix(int rows, int cols, double **mat){
   int i=0,j=0;
 for(i=0; i<rows; i++){    /* Iterate of each row */
      for(j=0; j<cols; j++){  /* In each row, go over each col element  */
         printf("%lf ",mat[i][j]); /* Print each row element */
      }
      printf("\n");
   }
}
int main(){
 double **matrix;
 int rows, cols;
 /* First matrix */
 printf("Matrix 1\n");
 printf("Enter # of rows and cols: ");
 scanf("%d%d",&rows,&cols);
 printf("Matrix, enter %d reals: \n",rows*cols);
 matrix = read_matrix(rows,cols);
 printf("Your Matrix\n");  /* Print the entered data */
 print_matrix(rows,cols,matrix);
 return 0;
}
```

Q. Write a note on Pointer Arithmetic. (5 marks) [June 2011, '06]

A pointer in C is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

ptr++

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array –

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  /* let us have array address in pointer */
  ptr = var;

  for ( i = 0; i < MAX; i++) {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below –

```
#include <stdio.h>
const int MAX = 3;
int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = &var[MAX-1];

  for ( i = MAX; i > 0; i--) {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the previous location */
    ptr--;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have address of the first element in pointer */
  ptr = var;
  i = 0;

  while ( ptr <= &var[MAX - 1] ) {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* point to the previous location */
    ptr++;
    i++;
  }
     return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Q. Distinguish pointer to an array and array of pointers with examples. (5 marks) [June 2010, '06]

Ans:

Pointer to an Array

An array name is a constant pointer to the first element of the array. Therefore, in the declaration −

double balance[50];

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p as the address of the first element of balance −

double *p;

double balance[10];

p = balance;

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on.

Example:

```
#include <stdio.h>

int main () {
  /* an array with 3 elements */
  double balance[3] = {1000.0, 2.0, 3.4};
  double *p;
  int i;
  p = balance;
   /* output each array element's value */
  printf( "Array values using pointer\n");
  for ( i = 0; i < 3; i++ ) {
     printf("*(p + %d) : %f\n",  i, *(p + i) );
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
```

Array of Pointers

An array which can store pointers to an int or char or any other data type is called an array of pointers. Following is the declaration of an array of pointers to an integer −

```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows −

```
#include <stdio.h>

 const int MAX = 3;

 int main () {

   int  var[] = {10, 100, 200};

   int i, *ptr[MAX];

    for ( i = 0; i < MAX; i++) {

      ptr[i] = &var[i]; /* assign the address of integer. */

   }

      for ( i = 0; i < MAX; i++) {

      printf("Value of var[%d] = %d\n", i, *ptr[i] );

   }

      return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200
```

Q. Write a C program to multiply 2 matrices using pointer notation. (15 marks) [June 2010, '06]

Ans:

Algorithm

Write an algorithm for the below program

Program

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

```
int main(void)
{
int a[10][10],b[10][10],c[10][10],r1=0,c1=0,i=0,j=0,r2=0,c2=0,k=0;
  int *pt,*pt1,*pt2;
printf("Enter size of 1st 2d array : ");
scanf("%d %d",&r1,&c1);
  for(i=0;i<r1;i++)
{
     for(j=0;j<c1;j++)
        {
        printf("Enter element no. %d %d :",i,j);
scanf("%d",&a[i][j]);
}
}
printf("Enter size of 2nd 2d array : ");
  scanf("%d %d",&r2,&c2);
for(i=0;i<r2;i++)
{
for(j=0;j<c2;j++)
        {
printf("Enter element no. %d %d :",i,j);
scanf("%d",&b[i][j]);
     }
}
if(c1!=r2)
     {
printf("Multiplication cannot be done\n");
exit (0);
}
pt=&a[0][0];
pt1=&b[0][0];
pt2=&c[0][0];
for(i=0;i<r1;i++)
     {
     for(k=0;k<c2;k++)
        {
```

```
        *(pt2+(i*10+k))=0;
        for(j=0;j<c1;j++)
{
*(pt2+(i*10+k))+=*(pt+(i*10+j))**(pt1+(j*10+k));
        } }}
for(i=0;i<r1;i++)
     {
for(j=0;j<c2;j++)
        {
        printf("%d ",c[i][j]);
     }
printf("\n");
}
return 0;
}
```

Output

Enter size of 1st 2D array: 2 2

Enter element no 0 0 :1

Enter element no 0 1 :1

Enter element no 1 0 :1

Enter element no 1 1 :1

Enter size of 2nd 2D array: 2 2

Enter element no 0 0 :1

Enter element no 0 1 :1

Enter element no 1 0 :1

Enter element no 1 1 :1

2 2

2 2

Q. What is a dynamic array? How is it created? (5 marks) [May 2015, '12]

Ans:

A dynamic array, growable array, resizable array, dynamic table, mutable array, or array list is a random access, variable-size list data structure that allows elements to be added or removed.

Features in C that enable you to implement your own dynamic array:

Memory management functions:

malloc( ) (we will use calloc() for arrays)

free( )

Pointer arithmetic:

if p points to a[0] then

*(p + i) is an alias for a[i]

Steps to create a dynamic array (of any data type):

Define a reference variable (say p) of the desired data type

Example: suppose we want to create a dynamic array of double variables

```
double*  p;       // reference variable to a double variable
```

Allocate memory cells for the array elements and make p point to the first array element:

The array elements can now be accessed as:

*(p+0) first array element

*(p+1) second array element

*(p+2) third array element

...

*(p+i) ith array element

The calloc() function: memory allocation function for arrays

The malloc() function only alocate memory cells for one variable:

malloc(nBytes) will allocates a block of memory cells of at least nBytes bytes that us suitably aligned for any usage.

To allocate memory cells for N consecutive variable (= array), you must use this function:

calloc(nElems, nBytes)

Example:

```
calloc( 10, 8 )
    // Allocate space for 10 elements of size 8 (= array of 10 double)
```

The calloc() function allocates space for an array of nElems elements of size nBytes.

Note:

The allocated space will also be initialized to zeros.

C program that illustrates a dynamic array of double variables:

```
int main(int argc, char *argv[])
{
   int i;

   double* p;    // We uses this reference variable to access
                 // dynamically created array elements
```

```
    p = calloc(10, sizeof(double) );  // Make double array of 10 elements

    for ( i = 0; i < 10; i++ )
        *(p + i) = i;           // put value i in array element i

    for ( i = 0; i < 10; i++ )
        printf("*(p + %d) = %lf\n", i, *(p+i) );


    free(p);     // Un-reserve the first array

    putchar('\n');

    p = calloc(4, sizeof(double) );  // Make a NEW double array of 4 elements

    // ***** Notice that the array size has CHANGED !!! ****

    for ( i = 0; i < 4; i++ )
        *(p + i) = i*i;           // put value i*i in array element i

    for ( i = 0; i < 4; i++ )
        printf("*(p + %d) = %lf\n", i, *(p+i) );

    free(p);     // Un-reserve the second array
  }
```

Output:

```
  *(p + 0) = 0.000000
  *(p + 1) = 1.000000
  *(p + 2) = 2.000000
  *(p + 3) = 3.000000
  *(p + 4) = 4.000000
  *(p + 5) = 5.000000
  *(p + 6) = 6.000000
  *(p + 7) = 7.000000
  *(p + 8) = 8.000000
```

*(p + 9) = 9.000000

*(p + 0) = 0.000000

*(p + 1) = 1.000000

*(p + 2) = 4.000000

*(p + 3) = 9.000000

DYNAMIC MEMORY ALLOCATION

Q. What is dynamic memory allocation? Give library function used for that with syntax. (5 marks)
[June 2014, '12]

Ans:

C dynamic memory allocation refers to performing manual memory management for dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely malloc, realloc, calloc and free.

The exact size of array is unknown until the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

- **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()** : dellocate 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 with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

free(ptr);

This statement causes the space in memory pointer by ptr to be deallocated.

COMMAND LINE ARGUMENTS

1. What are Command Line Arguments? (5 marks)

Ans:

Command line arguments are the values that are passed to the program while executing the C program. The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program.

*int main(int argc, char *argv[])*

argv[0] holds the name of the program itself and argv[1] is a pointer to the first command line argument supplied, and *argv[n] is the last argument.

Example:

When the following is executed

./cmdline Hello.out hello world

The values are as given below.

argc = 3

argv[1] = hello

argv[2] = world

2. Using command line arguments, concatenate two strings Hello and World. (6 marks)

Ans:

```
#include <stdio.h>
#include <string.h>
int main ( int argc , char *argv [ ] )
{
int i ;
```

```
char concat [ 50 ] ;
// Displaying the value of the argc
printf ( " argc  = %d \n" , argc) ;
// loop to display all the arguments passed
// after the program name on separate lines
for ( i =1; i < argc ; i++)
printf ( " argv[%d ]  = %s  \n", i,argv[ i ]);
// concatenate
printf( "concatenated string = %s \n", strcat ( argv [ 1 ] , argv [ 2 ] ) ) ;
}
```

OUTPUT

```
./ cmdline Hello.out hello world
argc = 3
argv [ 1 ] = hello
argv [ 2 ] = world
concatenated string = helloworld
```

3. Using command line arguments, find the largest of three numbers. (6 marks)

Ans:

```
#include <stdio.h>
int main ( char argc , char *argv [ ] )
{
int a , b , c ;
i f ( argc == 4)
{
a = atoi ( argv [ 1 ] ) ;
b = atoi ( argv [ 2 ] ) ;
c = atoi ( argv [ 3 ] );

if ( a > b && a > c )
printf ( " Largest  = %d\n" , a ) ;
else if ( b > c && b > a )
printf ( "Largest  = %d\n" , b ) ;
else
printf ( "Largest  = %d\n" , c ) ;
}
else
printf ("usage: %s  <a> <b> <c>\n" , argv [ 0 ] ) ;
```

```
return 0 ;
}

// Usage <filename> <int1> <int2> <int3>
```

OUTPUT

```
./ cmd Largest.out 3 4 5
Largest = 5
```

4. Write a program to receive a file name and a line of text as command line arguments and write the text to the file. (10 marks)

Ans:

```
#include <stdio.h>
void main(int argc, char *argv[])
{
        FILE *fp;
        int i;
        char word[20];
        fp=fopen(argv[1], "w");  /* open file with name argv[1]  */
        printf("\n No. of  arguments in command line = %d\n",argc);
        for(i=2;i<argc;i++)
        fprintf(fp,"%s",argv[i]); /* write to file argv[1]  */
        fclose(fp);

        /* writing the contents of the file to the screen   */
        printf("Contents of %s file\n\n", argv[1]);
        fp=fopen(argv[1],"r");
        for(i=2;i<argc;i++)
        {
                fscanf(fp,"%s",word);
                printf("%s ",word);
        }
        fclose(fp);
}
```

OUTPUT

```
C:\>File_write TEXT.txt I AM A GOOD PERSON
No. of arguments in command line = 7
Contents of TEXT.txt file
```

I AM A GOOD PERSON

(In the above example we assume that the file TEXT.txt is located in the C drive. If not, exact path is to be given in the argument.)

5. How can we give command line arguments in C? Explain with an example. (5 marks)

Ans:

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

```
#include <stdio.h>

int main( int argc, char *argv[] )  //arguments supplied in the command prompt window will be accepted here

{

  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");

  }

}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing

The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2                //This is how you pass Command line arguments to a C program

Too many arguments supplied.
```

FILES

1. **Write the syntax for fgetc() and fgets(). Give and example for each.** (5 marks)

Ans:

Syntax:

*int fgetc(FILE *stream)*

The C library function int fgetc() gets the next character from the specified stream and advances the position indicator for the stream.

```
#include <stdio.h>
int main ( )
{
FILE *fp ;
int c , n = 0 ;
fp = fopen("file.txt","r");
if(fp!=NULL)
{
do
{
c=fgetc(fp);
if(feof(fp))
{
break;
}
printf("%c",c);
}while(1);
fclose(fp);
}
return(0);
}
```

Syntax:

*char *fgets(char *str, int n, FILE *stream)*

The C library function fgets() reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

```
#include<stdio.h>
int main()
{
FILE *fp ;
char str[60];
fp = fopen ("file.txt","r");
if(fp!=NULL)
{
if(fgets(str,60,fp)!=NULL)
15
{
```

```
puts(str);
}
fclose(fp);
}
return(0);
}
```

2. Explain the different types of files and four file handling functions with its prototype. (5 marks)

Ans:

There are two different types of data files, called stream-oriented (or standard) data files, and system-oriented (or low-level) data files.

1. Stream-oriented data files are general text or data files. They are easier to work with and are therefore more commonly used. Stream-oriented data files can be subdivided into two categories.
   a) In the first category are text files, consisting of consecutive characters. These characters can be interpreted as individual data items, or as components of strings or numbers.
   b) The second category of stream-oriented data files, often referred to as unformatted data files (Binary file), organizes data into blocks containing contiguous bytes of information. These blocks represent more complex data structures, such as arrays and structures.
2. System-oriented data files are more closely related to the computer's operating system than stream-oriented data files.

Four file handling functions

1. fopen( ) function
   - to create a new file or to open an existing file. This call will initialize an object of the type FILE, which contains all the information necessary to control the stream.
   - Its prototype is FILE *fopen(const char * filename, const char * mode)
2. fclose( ) function
   - To close a file, use the fclose( ) function. The fclose() function returns zero on success, or EOF if there is an error in closing the file.
   - The prototype of this function is int fclose(FILE *fp)
3. fgetc() function
   - fgetc() function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns EOF.
   - The prototype of this function is int fgetc(FILE * fp)
4. fputc() function
   - fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error.
   - The prototype of this function is int fputc(int c, FILE *fp)

3. Write a program to write N numbers to a file. (6 marks)

Ans:

```
#include <stdio.h>

int main ( )

{
```

```
int length , loop , num[10];
// create stream
FILE  *fptr;
// open file in write mode
fptr=fopen("sample.txt " , "w" ) ;
// check if the file is opened successfully
if (fptr!=NULL)
{
// read a number
printf("Enter a number :  ");
scanf ("%d", &length);
// read numbers from keyboard and write them to file
for ( loop = 0 ; loop < length ; loop++)
{
scanf ( "%d" , &num[ loop ]) ;
fprintf(fpt r , "%d" ,num[ loop ] ) ;
}
// close the file after use
fclose(fptr);
return ;
}
else
{
printf("Error ! ") ;
return;
}
}
```

Output

```
Enter a number : 5
12
23
34
45
56
```

Contents of sample.txt

```
1223344556
```

4. Write a program to open a file called text.dat and delete the same file. (6 marks)

Ans:

```
#include <stdio.h>
int main ( )
{
int status ;
char * file_name = "sample.txt" ;
// create stream
FILE *fptr;
// open file in write mode
fptr=fopen ( file_name , "w" ) ;
// check if the file is opened successfully
if (fptr==NULL){
printf ( " Error ! " ) ;
return ;
}
printf ( "%s   file created successfully.\ n" , file_name ) ;
status = remove ( file_name ) ;
if ( status == 0 )
printf ( "%s file deleted successfully.\n" , file_name ) ;
else
{
printf ( "Unable to delete the file\n" ) ;
perror ( " Error " ) ;
}
return 0 ;
}
```

Output

sample.txt file created successfully.

sample.txt file deleted successfully.

5. Write a program to count number of characters and number of lines in a file. (10 marks)

(Algorithm will carry 2 marks, Program 5 marks and Output 1 mark)

Ans:

Algorithm:

1) Start

2) Declare file pointer, 2 variables nol=0 and noc=0 which is used to store number of lines and number of characters respectively.
3) Open the file in read mode using fopen()
4) Read each character in the file using getc() in a while loop and every time a character is read we check for :
   if(c==EOF)
   break;
   noc++;
   if(c=='\n')
   nol++;
5) Close the file using fclose()
6) Finally print the values of noc and nol+1 to disaply the result.
7) Stop

Program

```
#include<stdio.h>

#include<conio.h>

void main()

{

 FILE *fp;

 char a[20];

 int nol=0,noc=0;

 char c;

 printf("Enter the name of File:\n");

 gets(a);

 if((fp=fopen(a,"r"))==NULL)

 {

  printf("File dosen't exist.");

 }

 else

 {

  while(1)

  {

   c=fgetc(fp);

   if(c==EOF)

      break;
```

```
  noc++;
  if(c=='\n')
  nol++;
 }
}
fclose(fp);
printf("Number of characters = %d\n",noc);
printf("Number of lines = %d\n",nol+1);
}
```

OUTPUT

Enter the name of the file:

Hello.txt

Number of characters = 11

Number of lines = 2

Hello.txt (Assuming the contents as follows)

Hello
World

6. Write a program to Copy contents of one file to another. (10 marks)

Ans:

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

void main() {
  FILE *fp1, *fp2;
  char filename[100], a;

  printf("Enter the name of the file to be opened for reading");
  scanf ( "%s " , filename ) ;
// Open one file for reading
  fp1 = fopen(filename, "r");
  if (fp1 == NULL) {
    puts("cannot open this file");
    exit(1);
  }
  printf("Enter the name of the file to be opened for writing");
  scanf ( "%s " , filename ) ;
```

```
// Open one file for writing
 fp2 = fopen(filename, "w");
   if (fp2 == NULL) {
      puts("Not able to open this file");
      fclose(fp1);
      exit(1);
   }
 //contents read and copied
   do {
      a = fgetc(fp1);
      fputc(a, fp2);
   } while (a != EOF);
printf ("\ nContents copied to %s ", filename);
    fclose(fp1);
    fclose(fp2);
   return 0;
}
```

Output

Enter the filename to open for reading

output.txt

Enter the filename to open for writing

sample.txt

Contents copied to sample.txt

7. **Write a program that will receive a filename and a line of text as command line argument and write the text to the file.** (8 marks) [June 2012, '06]

Ans:

```
#include <stdio.h>
void main(int argc, char *argv[])
{
        FILE *fp;
        int i;
        char word[20];
        fp=fopen(argv[1], "w");  /* open file with name argv[1]  */
        printf("\n No. of  arguments in command line = %d\n",argc);
        for(i=2;i<argc;i++)
```

```
        fprintf(fp,"%s",argv[i]); /* write to file argv[1]  */
        fclose(fp);

        /* writing the contents of the file to the screen   */
        printf("Contents of %s file\n\n", argv[1]);
        fp=fopen(argv[1],"r");
        for(i=2;i<argc;i++)
        {
                fscanf(fp,"%s",word);
                printf("%s ",word);
        }
        fclose(fp);
}
```

OUTPUT

C:\>File_write TEXT.txt I AM A GOOD PERSON

No. of arguments in command line = 7

Contents of TEXT.txt file

I AM A GOOD PERSON

(In the above example we assume that the file TEXT.txt is located in the C drive. If not, exact path is to be given in the argument.)