



A HANDBOOK ON ADVANCE C PROGRAMMING



Jv'n Ms. Nishu Sharma

JAYOTI VIDYAPEETH WOMEN'S UNIVERSITY, JAIPUR

UGC Approved Under 2(f) & 12(b) | NAAC Accredited | Recognized by Statutory Councils

Printed by :
JAYOTI PUBLICATION DESK

Published by :
Women University Press
Jayoti Vidyapeeth Women's University, Jaipur

Faculty of Education & Methodology

Title: A HANDBOOK ON ADVANCE C PROGRAMMING

Author Name: Ms. Nishu Sharma

Published By: Women University Press

Publisher's Address: Jayoti Vidyapeeth Women's University, Jaipur
Vedant Gyan Valley,
Village-Jharna, Mahala Jobner Link Road, NH-8
Jaipur Ajmer Express Way,
Jaipur-303122, Rajasthan (India)

Printer's Detail: Jayoti Publication Desk

Edition Detail:

ISBN: 978-93-90892-13-6

Copyright © - Jayoti Vidyapeeth Women's University, Jaipur

Table of Content:

S.No.	Content Name	Page number
1	CHAPTER 1:Pointers	1-3
2	CHAPTER 2:Arrays	4-12
3	CHAPTER 3:Strings	13-15
4	CHAPTER 4:Memory Allocation	16-23
5	CHAPTER 5:File	24-30
6	CHAPTER 6:Streams	31-41
7	CHAPTER 7:Graphics Programs	42-48

CHAPTER 1

Pointers

Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type or we can say that it hold the values of a certain data type such as int, float, char, double, short etc.

- Pointer Syntax : `data_type *var_name;` Example : `int *p;` `char *p;`
- Where, * is used to denote that “p” is pointer variable and not a normal variable.

Key points to remember about pointers in C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. `int *p = null;`
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

Before we discuss about **pointers in C**, lets take a simple example to understand what do we mean by the address of a variable.

A simple example to understand how to access the address of a variable without pointers?

In this program, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let's say the address assigned to variable num is 0x7fff5694dc58, which means

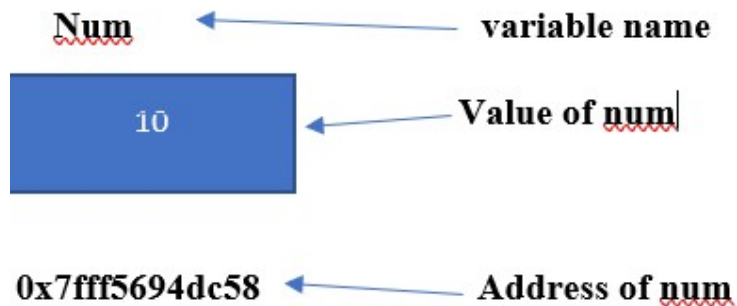
whatever value we would be assigning to num should be stored at the location: 0x7fff5694dc58.

So, take an example

```
#include <stdio.h>
int main()
{
    int num = 10;
    printf("Value of variable num is: %d", num);
    /* To print the address of a variable we use %p
    * format specifier and ampersand (&) sign just
    * before the variable name like &num.
    */
    printf("\nAddress of variable num is: %p", &num);
    return 0;
}
```

Output:

Value of variable num is: 10
Address of variable num is: 0x7fff5694dc58



A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

```
#include <stdio.h>
int main()
{
    //Variable declaration
    int num = 10;
```

```
//Pointer declaration
int *p;

//Assigning address of num to the pointer p
p = #

printf("Address of variable num is: %p", p);
return 0;
}
```

Output:

Address of variable num is: 0x7fff5694dc58

CHAPTER 2

Introduction To Arrays:

In C programming, one of the frequently problem is to handle similar types of data. For example: if the user wants to store marks of 500 students, this can be done by creating 500 variables individually but, this is rather tedious and impracticable. These types of problem can be handled in C programming using arrays.

An array in C Programing can be defined as number of memory locations, each of which can store the same data type and which can be references through the same variable name. It is a collective name given to a group of similar quantities. These similar quantities could be marks of 500 students, number of chairs in university, salaries of 300 employees or ages of 250 students. Thus we can say array is a sequence of data item of homogeneous values (same type). These values could be all integers, floats or characters etc.

We have two types of arrays:

1. One-dimensionalarrays.
2. Multidimensionalarrays.

One Dimensional Arrays:

A **one-dimensional array** is a structured collection of components (often called *array elements*) that can be accessed individually by specifying the position of a component with a single *index* value. Arrays must be declared before they can be used in the program. Here is the declaration syntax of one dimensional array:

```
data_typearray_name[array_size];
```

Here “data_type” is the type of the array we want to define, “array_name” is the name given to the array and “array_size” is the size of the array that we want to assign to the array. The array size is always mentioned inside the “[]”.

int	age	[5];
-----	-----	------

Here int is the data type

Age is the name of the array

[5] is the size of the array

The

following will be the result of the above declarations:

age[0]	age[1]	age[2]	age[3]	age[4]

Initializing Arrays

Initializing of array is very simple in c programming. The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name. Here is an example which declares and initializes an array of five elements of type int. Array can also be initialized after declaration. Look at the following code, which demonstrate the declaration and initialization of an array.

```
int age[5]={2,3,4,5,6};
```

It is not necessary to define the size of arrays during initialization

e.g. `int age[]={2,3,4,5,6};`

In this case, the compiler determines the size of array by calculating the

number of elements of an array. age[0] age[1]
 age[2] age[3]
 age[4]

2	3	4	5	6
---	---	---	---	---

Accessing array elements

In C programming, arrays can be accessed and treated like variables in

C. For example:

- `scanf("%d",&age[2]);`
//statement to insert value in the third element of array age[]

- `printf("%d",age[2]);`
//statement to print third element of an array.

Arrays can be accessed and updated using its index. An array of n elements, has indices ranging from 0 to n-1. An element can be updated simply by assigning

`A[i] = x;`

A great care must be taken in dealing with arrays. Unlike in Java, where array index out of bounds exception is thrown when indices go out of the 0..n-1 range, C arrays may not display any warnings if out of bounds indices are accessed. Instead, compiler may access the elements out of bounds, thus leading to critical run time errors.

Example of array in C programming

```
/* C program to find the sum marks of n students using arrays */  
  
#include <stdio.h>  
  
int main(){  
  
    int i,n;  
  
    int marks[n];  
    int sum=0;  
  
    printf("Enter number of students: ");  
    scanf("%d",&n);  
  
    for(i=0;i<n;i++){  
        printf("Enter marks of student%d: ",i+1);  
        scanf("%d",&marks[i]); //saving the marks in array  
        sum+=marks[i];  
    }  
  
    printf("Sum of marks = %d",sum);
```

Output :

```
Enter number of students: (input by user)3
Enter marks of student1: (input by user) 10
Enter marks of student2: (input by user) 29
Enter marks of student3: (input by user) 11
```

```
Sum of marks = 50
```

Important thing to remember in C arrays

Suppose, you declared the array of 10 students. For example: `students[10]`. You can use array members from `student[0]` to `student[9]`. But, what if you want to use element `student[10]`, `student[100]` etc. In this case the compiler may not show error using these elements but, may cause fatal error during program execution.

Multidimensional Arrays:

C programming language allows the user to create arrays of arrays known as multidimensional arrays. To access a particular element from the array we have to use two subscripts one for row number and other for column number. The notation is of the form `array[i][j]` where `i` stands for row subscripts and `j` stands for column subscripts. The array holds `i*j` elements. Suppose there is a multidimensional array `array[i][j][k][m]`. Then this array can hold `i*j*k*m` numbers of data. In the same way, the array of any dimension can be initialized in C programming. For example :

```
int smarks[3][4];
```

Here, `smarks` is an array of two dimension, which is an example of multidimensional array. This array has 3 rows and 4 columns. For better understanding of multidimensional arrays, array elements of above example can be as below:

	Col 1	Col 2	Col 3	Col 4
Row 1	<code>smakrs[0][0]</code>]	<code>smarks[0][1]</code>]	<code>smarks[0][2]</code>]	<code>smarks[0][3]</code>]
Row 2	<code>smarks[1][0]</code>]	<code>smarks[1][1]</code>]	<code>smarks[1][2]</code>]	<code>smarks[1][3]</code>]
Row 3	<code>smarks[2][0]</code>]	<code>smarks[2][1]</code>]	<code>smarks[2][2]</code>]	<code>smarks[2][3]</code>]

Make sure that you remember to put each subscript in its own, correct pair of brackets. All three examples below are wrong.

```

inta2[5,7];           /* XXX WRONG*/

a2[i,j]=0;           /* XXX WRONG*/

a2[j][i] =0;         /* XXX WRONG */ would do anything remotely
                     like what youwanted

```

Initialization of Multidimensional Arrays

In C, multidimensional arrays can be initialized in different number of ways.

```
int smarks[2][3]={ {1,2,3}, {-1,-2,-3}};
```

Arrays and Functions

In C, Arrays can be passed to functions using the array name. Array name is a const pointer to the array. both one-dimensional and multi-dimensional array can be passed to function as argument. Individual element is passed to function using pass by value. Original Array elements remain unchanged, as the actual element is never passed to function. Thus function body cannot modify original value in this case. If we have declared an array 'array [5]' then instead of passing individual elements, a for loop is useful in this case to pass all 5 elements of the array.

Passing One-dimensional Array In Function

Passing entire one-dimensional array to a function

While passing arrays to the argument, the name of the array is passed as an argument (i.e, starting address of memory area is passed as argument).

C program to pass an array containing age of person to a function. This function will return average age and display the average age in main function.

```
#include
<stdio.h> float
average(float
a[]); int main(){

    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};

    avg=average(c); /* Only name of array is passed as
argument. */ printf("Average age=%.2f",avg);

    return 0;
}

float
average(float
a[]){ int i;

    float avg,
    sum=0.0;
```

Output

Average age=27.08

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example to pass two-dimensional arrays to function

C Pointer and Arrays

In C Programming pointers and arrays are very closely related to each other in terms of functionality. Both pointers as well as arrays uses consecutive memory locations to store the data with one key difference in accessing the data. Pointer uses address to access the data stored in a memory location whereas array uses index value to access the data stored in a memory location. Fortunately, data or strings initialized using the pointer variable can be accessed using array and vice versa.

Program - Pointers To Initialize, Arrays To Access

pointer-array.c

```
#include <stdio.h>
int main()
{
    char *ptr="pointer array", i;
    for(i=0; i<13; i++)
        printf("%c",ptr[i]);
    return 0;
}
```

- pointer array

Note:

In the above example program *ptr is a char pointer variable which is initialized by a string then the same pointer variable is used as an array to access the value get stored by a pointer variable *ptr.

C Program - Arrays To Initialize, Pointers To Access

```
array-pointer-access.c
#include <stdio.h>
int main()
{
    char arr[13] = "Pointer array", i, *ptr;
    ptr = arr;
    for(i =0; i<13; i++)
        printf("%c",*ptr++);
    return 0;
}
```

- pointer array

Note:

In the above example program arr[13] is a char array variable which is initialized by a set of characters then the address of first element in an array is assigned to the pointer variable which is then post incremented to print all the characters initialized to the array variable arr[13].

Ways To Expressions Pointer Array Elements

There are four different syntaxes are used to express the Pointer and array elements. Two syntaxes for array and two syntaxes for pointer.

- a[i]
- i[a]

- `*(a + i)`
- `*(i + a)`

C Program - Access Pointer Array Elements

```
pointer-array-access.c
#include <stdio.h>
int main()
{
    int i, a[3] = {10, 11, 12};
    for(i = 0; i < 3; i++)
    {
        printf("%d \t", a[i]);
        printf("%d \t", i[a]);
        printf("%d \t", *(a + i));
        printf("%d \n", *(i + a));
    }
    return 0;
}
```

OUTPUT

```
10    10    10    10
11    11    11    11
12    12    12    12
```

Note:

In the above example program array can express their elements either `a[i]` or `i[a]`. If array can express their elements in two ways then pointers too can express their elements in two ways (`*(a + i)` or `*(i + a)`) as pointers and arrays are all a same family.

Pointers And Two Dimensional Arrays

The general form of two dimensional array looks `arr[i][j]`, `i` represents the number of rows in an array whereas `j` represents the number of column in an array. The base address of an array is `arr[0][0]`.

C Program - Pointers And Two Dimensional Arrays

```
pointers-2d-array.c
#include <stdio.h>
int main()
{
    int *iptr;
    int i, a[2][2] = {10, 11, 12, 13};
    iptr = &a[0][0];
```

```
for(i = 0; i < 4; i++)
{
if(i == 2)
{
printf("\n");
}
printf("%d ", *(iptr+i));
}
return 0;
}
```

output

10 11

12 13

Note:

In the above example program two dimensional array is declared and initialized. Though it is a two dimensional array, the elements get stored in the consecutive memory locations. Thus with pointers capability we access all the elements which we initialized to the two dimensional array earlier.

CHAPTER 3

Strings

Strings are sequences of characters. C does not have a built-in string type; instead, strings are created out of character arrays. In fact, strings are just character arrays with a few restrictions. One of these restrictions is that the special character '\0' (NUL) is used to indicate the end of a string.

For example:

```
char name[4];
int main()
{
    name[0] = 'S';
    name[1] = 'a';
    name[2] = 'm';
    name[3] = '\0';
    return (0);
}
```

Strings and Pointers

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string.

Example 1: Strings and Pointers

```
#include <stdio.h>

int main(void)
{
    char name[] = "Harry Potter";

    printf("%c", *name); // Output: H
    printf("%c", *(name+1)); // Output: a
    printf("%c", *(name+7)); // Output: o

    char *namePtr;

    namePtr = name;
    printf("%c", *namePtr); // Output: H
    printf("%c", *(namePtr+1)); // Output: a
    printf("%c", *(namePtr+7)); // Output: o
}
```


Pointer Within Structure in C Programming :

1. Structure may contain the **Pointer variable as member**.
2. Pointers are used to store the address of memory location.
3. They can be **de-referenced** by '*' operator.

Example :

```
struct Sample
{
    int *ptr; //Stores address of integer Variable
    char *name; //Stores address of Character String
}s1;
```

4. **s1** is structure variable which is used to access the “**structure members**”.

```
s1.ptr = &num;
s1.name = "Pritesh"
```

5. Here **num** is any variable but it's address is stored in the Structure member ptr (**Pointer to Integer**)
6. Similarly Starting address of the String “Pritesh” is stored in structure variable name(**Pointer to Character array**)
7. Whenever we need to print the content of variable **num** , we are dereferencing the pointer variable num.

```
printf("Content of Num : %d ",*s1.ptr);
printf("Name : %s",s1.name);
```

Live Example : Pointer Within Structure

```
#include<stdio.h>

struct Student
{
    int *ptr; //Stores address of integer Variable
    char *name; //Stores address of Character String
}s1;

int main()
{
    int roll = 20;
    s1.ptr = &roll;
    s1.name = "Maheshwari";

    printf("\nRoll Number of Student : %d",*s1.ptr);
    printf("\nName of Student : %s",s1.name);

    return(0); }
```

Output :

```
Roll Number of Student : 20  
Name of Student      : Maheshwari
```

Some Important Observations :

```
printf("\nRoll Number of Student : %d",*s1.ptr);
```

We have stored the address of variable 'roll' in a pointer member of structure thus we can access value of pointer member directly using de-reference operator.

```
printf("\nName of Student      : %s",s1.name);
```

Similarly we have stored the base address of string to pointer variable 'name'. In order to de-reference a string we never use de-reference operator.

CHAPTER 4

MEMORY ALLOCATION

Static memory allocation is an allocation technique which allocates a fixed amount of memory during compile time and the operating system internally uses a data structure known as Stack to manage this.

Background

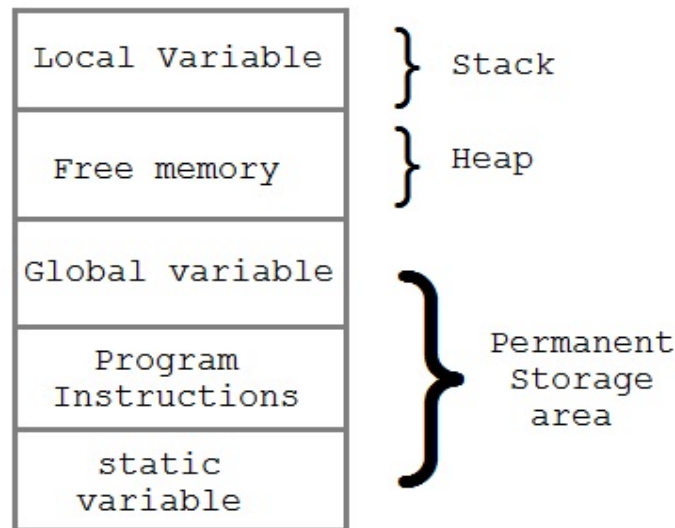
Memory is central to any computing system and its architecture determines the performance of any process. While building system, one of the fundamental tasks is to allocate memory.

There are two types of memory allocated to a program:

- **Stack memory**
- **Heap memory**

Stack memory is allocated during compilation time execution. This is known as static memory allocation.

Whereas, **heap memory** is allocated at run-time compilation. This is known as dynamic memory allocation.



Static memory allocation

In Static Memory Allocation the memory for your data is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and variables qualified with static defined inside functions. This memory allocation is fixed and cannot be changed, i.e. increased or decreased after allocation. So, exact memory requirements must be known in advance.

Key features:

- Variables get allocated permanently
- Allocation is done before program execution
- It uses the data structure called stack for implementing static allocation
- Less efficient
- There is no memory reusability

Example

- All the variables in the program below are statically allocated.

```
void play
{
    int a;
}
int main()
{
    int b;
    int c[10];
    return 1;
}
C
Copy
```

Advantages of Static memory allocation

- Simplicity of usage.
- Efficient execution time.
- Need not worry about memory allocation/re-allocation/freeing of memory
- Variables remain permanently allocated.

Disadvantages of Static memory allocation

- Main disadvantage is wastage of memory.
- Memory can't be freed when it is no longer needed.

Dynamic Memory Allocation (DMA)

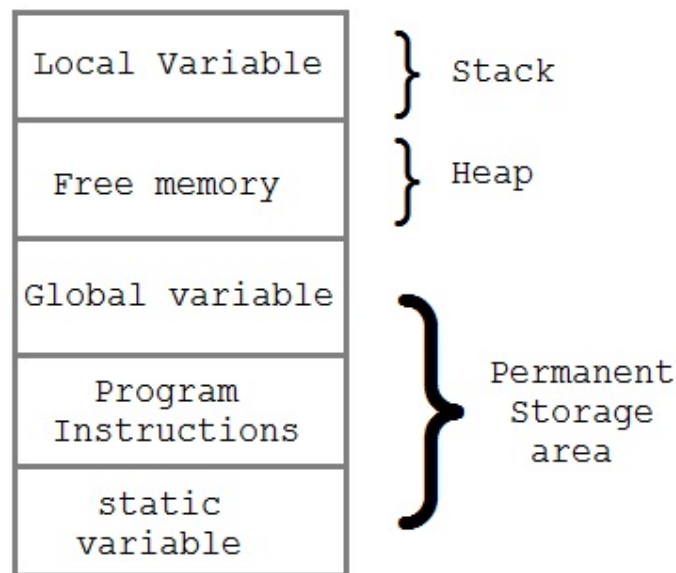
The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as **memory management functions** are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h** header file.

Function	Description
malloc()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Memory Allocation Process

Global variables, static variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**.

The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



Allocating block of Memory

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of the given size and returns a **pointer** of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void* malloc(byte-size)
```

Time for an Example: malloc()

```
int *x;  
x = (int*)malloc(50 * sizeof(int)); //memory space allocated to variable x  
free(x); //releases the memory allocated to variable x
```

calloc() is another memory allocation function that is used for allocating memory at runtime. calloc function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void *calloc(number of items, element-size)
```

Time for an Example: calloc()

```
struct employee  
{  
    char *name;  
    int salary;  
};  
typedef struct employee emp;  
emp *e1;  
e1 = (emp*)calloc(30,sizeof(emp));
```

realloc() changes memory size that is already allocated dynamically to a variable.

Syntax:

```
void* realloc(pointer, new-size)
```

Time for an Example: realloc()

```
int *x;  
x = (int*)malloc(50 * sizeof(int));  
x = (int*)realloc(x,100); //allocated a new memory to variable x
```

Difference between malloc() and calloc()

calloc()	malloc()
calloc() initializes the allocated memory with 0 value.	malloc() initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax :	Syntax :
(cast_type *)calloc(blocks , size_of_block);	(cast_type *)malloc(Size_in_bytes);

sizeof operator

The sizeof operator is the most common operator in C. It is a compile-time unary operator and used to compute the size of its operand. It returns the size of a variable. It can be applied to any data type, float type, pointer type variables.

When sizeof() is used with the data types, it simply returns the amount of memory allocated to that data type. The output can be different on different machines like a 32-bit system can show different output while a 64-bit system can show different of same data types.

Here is an example in C language,

Example

```
#include <stdio.h>
int main() {
    int a = 16;
    printf("Size of variable a : %d\n",sizeof(a));
    printf("Size of int data type : %d\n",sizeof(int));
    printf("Size of char data type : %d\n",sizeof(char));
    printf("Size of float data type : %d\n",sizeof(float));
    printf("Size of double data type : %d\n",sizeof(double));
    return 0;
}
```

Output

```
Size of variable a : 4
Size of int data type : 4
Size of char data type : 1
Size of float data type : 4
Size of double data type : 8
```

When the sizeof() is used with an expression, it returns the size of the expression. Here is an example.

Example

```
#include <stdio.h>
int main() {
    char a = 'S';
    double b = 4.65;
    printf("Size of variable a : %d\n",sizeof(a));
    printf("Size of an expression : %d\n",sizeof(a+b));
    int s = (int)(a+b);
    printf("Size of explicitly converted expression : %d\n",sizeof(s));
    return 0;
}
```

Output

Size of variable a : 1
Size of an expression : 8
Size of explicitly converted expression : 4

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

int main()
{
    int i,j,k, n ;
    int* Array;

    printf("Enter the number of elements of Array : ");
    scanf("%d", &n );
    Array= (int*) calloc(n, sizeof(int));
    if( Array== (int*)NULL)
    {
        printf("Error. Out of memory.\n");
        exit (0);
    }
    printf("Address of allocated memory= %u\n" , Array);
    printf("Enter the values of %d array elements:", n);
    for (j =0; j<n; j++)
```



```

scanf("%d",&Array[j]);

printf("Address of Ist member= %u\n", Array);

printf("Address of 2nd member= %u\n", Array+1);

printf("Size of Array= %u\n", n* sizeof(int) );

for ( i = 0 ; i<n; i++)

printf("Array[%d] = %d\n", i, *(Array+i));

free(Array);

printf("After the action of free() the array elements are:\n");

for (k =0;k<n; k++)

printf("Array[%d] = %d\n", k, *(Array+i));

return 0;

}

```

```

Enter the number of elements of Array : 4
Address of allocated memory= 17416208
Enter the values of 4 array elements:10 20 30 40
Address of Ist member= 17416208
Address of 2nd member= 17416212
Size of Array= 16
Array[0] = 10
Array[1] = 20
Array[2] = 30
Array[3] = 40
After the action of free() the array elements are:
Array[0] = 0
Array[1] = 0
Array[2] = 0
Array[3] = 0

...Program finished with exit code 0
Press ENTER to exit console.

```

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}

```

Output will be:

```

Enter number of elements: 5
Enter elements:
12
31
46
64
54
m = 207

```

Su

CHAPTER 5

FILE:

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready-made structure.

In C language, we use a structure pointer of file type to declare a file.

FILE *fp;

FILE structure in C Programming

1. C Provides smart way to **manipulate data using streams**
2. In **stdio.h header** file structure is defined
3. FILE structure provides us the necessary information about a **FILE or stream which performs input and output operations**

Structure :

```
typedef struct
{
    short level ;
    short token ;
    short bsize ;
    char fd ;
    unsigned flags ;
    unsigned char hold ;
    unsigned char *buffer ;
    unsigned char *curp ;
    unsigned istemp;
}FILE ;
```

Members of Structure :

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 an 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
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 begining point

1. What is File Handling in C?

A file is nothing but a source of storing information permanently in the form of a sequence of bytes on a disk. The contents of a file are not volatile like the C compiler memory. The various operations available like creating a file, opening a file, reading a file or manipulating data inside a file is referred to as file handling.

Member	Use / Function
Level	Fill / Empty level of Buffer
Token	Validity Checking
Bsize	Buffer size
Fd	File descriptor using which file can be identified
Flags	File status flag
Hold	ungetc character if no buffer space available
Buffer	Data transfer buffer
Curp	Current active pointer
Istemp	Temp. File indicator

2. Need for File Handling in C

There is a time when the output generated by compiling and running the program does not serve the purpose. If we want to check the output of the program several times, it becomes a tedious task to compile and run the same program multiple times. This is where file handling comes into play. Here are some of the following reasons behind the popularity of file handling:

- **Reusability:** It helps in preserving the data or information generated after running the program.
- **Large storage capacity:** Using files, you need not worry about the problem of storing data in bulk.

- **Saves time:** There are certain programs that require a lot of input from the user. You can easily access any part of the code with the help of certain commands.
- **Portability:** You can easily transfer the contents of a file from one computer system to another without having to worry about the loss of data.

3. Different Types of Files in C

When talking about files with reference to file handling, we generally refer to it as data files. There are basically 2 distinct types of data files available in the C programming language:

3.1 Text files

These are the simplest files a user can create when dealing with file handling in C. It is created with a **.txt** extension using any simple text editor. Generally, we use notepads to create text files. A text file stores information in the form of ASCII characters internally, but when you open the text file, you would find the content of the text readable to humans.

Hence, it is safe to say that text files are simple to use and access. But, along with advantages comes disadvantages as well. Since it is easily readable, it does not provide any security of information. Moreover, it consumes large storage space. Hence, there is a different type of file available called binary files which helps us solve this problem.

3.2 Binary files

A binary file stores information in the form of the binary number system (0's and 1's) and hence occupies less storage space. In simple words, it stores information in the same way as the information is held in computer memory. Therefore, it proves to be much easier to access.

It is created with a **.bin** extension. It overcomes the drawback offered by text files. Since it is not readable to humans, the information is more secure. Hence, it is safe to say that binary files prove to be the best way to store information in a data file.

4. C File Handling Operations

The C programming offers various operations associated with file handling. They are:

- Creating a new file: **fopen()**
- Opening an existing file in your system: **fopen()**
- Closing a file: **fclose()**
- Reading characters from a line: **getc()**
- Writing characters in a file: **putc()**
- Reading a set of data from a file: **fscanf()**
- Writing a set of data in a file: **fprintf()**
- Reading an integral value from a file: **getw()**
- Writing an integral value in a file: **putw()**
- Setting a desired position in the file: **fseek()**
- Getting the current position in the file: **ftell()**
- Setting the position at the beginning point: **rewind()**

Key takeaway: It is important to note that a file-type pointer needs to be declared when working with files. It establishes communication between the file and the program.

This is how it is done:

FILE *fpointer;

Out of these various operations, there are some basic operations used in the C programming language which we will discuss in detail one by one:

5. Opening a Text File in C

We use the `fopen()` function to create or open a file as mentioned earlier. It is pretty obvious that creating or opening a file is the first step in file handling. Once the file has been created, it can be opened, modified or deleted.

The basic syntax of opening a file is:

1. `*fpointer = FILE *fopen(const char *file_name, const char *mode);` Here,

`*fpointer` is the pointer to the file that establishes a connection between the file and the program.

`*file_name` is the name of the file.

`*mode` is the mode in which we want to open our file.

The following modes in which the file can be opened are:

MODE ELUCIDATION

R	We use it to open a text file in reading mode
W	We use it to open or create a text file in writing mode
A	We use it to open a text file in append mode
r+	We use it to open a text file in both reading and writing mode
w+	We use it to open a text file in both reading and writing mode
a+	We use it to open a text file in both reading and writing mode
Rb	We use it to open a binary file in reading mode
Wb	We use it to open or create a binary file in writing mode
ab	We use it to open a binary file in append mode
rb+	We use it to open a binary file in both reading and writing mode
wb+	We use it to open a binary file in both reading and writing mode
ab+	We use it to open a binary file in both reading and writing mode

6. Closing a Text File in C

We use the **`fclose()`** function to close a file that is already open. It is pretty obvious that the file needs to be opened so that the operation to close a file can be performed.

1. `int fclose(FILE *fpinter);`

Here, the `fpinter` is the pointer associated with closing the file. This function is responsible for returning the value 0 if the file is closed successfully. Otherwise, EOF (end of file) in case of any error while closing the file.

7. Reading and Writing a Text File in C

After discussing how to open and close a file, it is important to note that there are 3 types of streams (sequence of bytes) in a file:

- Input
- Output
- Input / Output

The input/output operations in a file help you read and write in a file.

TYPES OF FILES

There are 2 kinds of files in which data can be stored in 2 ways either in characters coded in their ASCII character set or in binary format. They are

1. Text Files.
2. Binary Files

TEXT FILES

A Text file contains only the text information like alphabets ,digits and special symbols. The ASCII code of these characters are stored in these files.It uses only 7 bits allowing 8 bit to be zero.

1.w(write):

This mode opens new file on the disk for writing.If the file exist,disk for writing.If the file exist, then it will be over written without then it will be over written without any confirmation.

SYNTAX:

```
fp=fopen("data.txt","w");  
"data.txt" is filename  
"w" is writemode.
```

2. r (read)

This mode opens an preexisting file for reading.If the file doesn't Exist then the compiler returns a NULL to the file pointer

SYNTAX:

```
fp=fopen("data.txt","r");
```

3. w+(read and write)

This mode searches for a file if it is found contents are destroyed If the file doesn't found a new file is created.

SYNTAX:

```
fp=fopen("data.txt","w+");
```

4.a(append)

This mode opens a preexisting file for appending the data.

SYNTAX:

```
fp=fopen("data.txt","a");
```

5.a+(append+read)

the end of the file.

SYNTAX:

```
fp=fopen("data.txt","a+");
```

6.r+ (read +write)

This mode is used for both Reading and writing

BINARY FILES

A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.

The only difference between the text file and binary file is the data contain in text file can be recognized by the word processor while binary file data can't be recognized by a word processor.

l.wb(write)

this opens a binary file in write mode.

SYNTAX:

```
fp=fopen("data.dat","wb");
```


2.rb(read)

this opens a binary file in read mode

SYNTAX:

```
fp=fopen("data.dat","rb");
```

3.ab(append)

this opens a binary file in a Append mode i.e. data can be added at the end of file.

SYNTAX:

```
fp=fopen("data.dat","ab");
```

4.r+b(read+write)

this mode opens preexisting File in read and write mode.

SYNTAX:

```
fp=fopen("data.dat","r+b");
```

5.w+b(write+read)

this mode creates a new file for reading and writing in Binary mode.

SYNTAX:

```
fp=fopen("data.dat","w+b");
```

6.a+b(append+write)

this mode opens a file in append mode i.e. data can be written at the end of file.

SYNTAX:

```
fp=fopen("data.dat","a+b");
```

CHAPTER 6

STREAMS

A Stream refers to the characters read or written to a program. The streams are designed to allow the user to access the files efficiently .A stream is a file or physical device like keyboard, printer and monitor.

The FILE object contains all information about stream like current position, pointer to any buffer, error and EOF(end of file).

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

1. naming a file
2. opening a file
3. reading data from a file
4. writing data into a file
5. closing a file

Write a program to read character by character from the user and write them into a file using fputc() and read the same data from the file using fgetc() and display that on the screen.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    Char ch;
    Clrscr();
    fp=fopen("a.txt", "w+");
    printf("Enter value into file\n");
    while((ch=getchar())!=EOF)
    {
        fputc(ch,fp);
    }
    printf("Reading data from file\n");
    rewind(fp); // Here rewind() will move file pointer to the beginning of the file
    while((ch=fgetc(fp))!=EOF)
    {
        printf("%c",ch);
    }
    fclose(fp);
    getch();
}
```

OUTPUT:

Enter value into file

Hii hello how r u I am good

Reading data from a file

Hii hello how r u I am good

- Information is kept in files.
- Files reside on secondary storage.
- When this information is to be used, it has to be accessed and brought into primary main memory.
- Information in files could be accessed in many ways.
- It is usually dependent on an application.
- There are three file access methods:
 1. Sequential Access
 2. Direct Access
 3. Indexed Sequential Access

Sequential Access:

- A simple access method, information in a file is accessed sequentially one record after another.
- To process with record all the 1-1 records previous to 1 must be accessed.
- Sequential access is based on the tape model that is inherently a sequential access device.
- Sequential access is best suited where most of the records in a file are to be processed. For example: Transaction files.

Direct Access:

- Sometimes it is not necessary to process every record in a file.
- It may not be necessary to process records in the order in which they are present.
- Information present in a record of a file is to be accessed only if some key value in that record is known. In all such cases, direct access is used.
- Direct access is based on the disk that is a direct access device and allows random access of any file block. Since a file is a collection of physical blocks, any block and hence the records in that block are accessed.
- For example, Master files. Databases are often of this type since they allow query processing that involves immediate access to large amounts of information. All reservation systems fall into this category. Not all operating systems support direct access files. Usually files are to be defined as sequential or direct at the time of creation and accessed accordingly later.
- Sequential access of a direct access file is possible but direct access of a sequential file is not.

Indexed Sequential Access:

- This access method is a slight modification of the direct access method.
- It is in fact a combination of both the sequential access as well as direct access.
- The main concept is to access a file direct first and then sequentially from that point onwards. This access method involves maintaining an index.
- The index is a pointer to a block. To access a record in a file, a direct access of the index is made.
- The information obtained from this access is used to access the file. For example, the direct access to a file will give the block address and within the block the record is accessed sequentially.
- Sometimes indexes may be big. So hierarchies of indexes are built in which one direct access of an index leads to info to access another index directly and so on till the actual file is accessed sequentially for the particular record.
- The main advantage in this type of access is that both direct and sequential access of files is possible.

Print f & Scanf

printf and scanf are two standard C programming language functions for input and output. Both are functions in the stdio library which means `#include <stdio.h>` is required at the top of your file.

scanf

Function arguments:

`int scanf(const char * format, [var1], [var2],...)`

scanf requires format placeholders within the format string to denote what data type we would like the input to be read as by the computer.

Here are a table of some of the most commonly used placeholders:

PLACEHOLDER	DATATYPE
%d or %i	integer
%u	unsigned int
%f	floating-point
%lf	doublefloating-point
%c	character
%s	String

Example 1:

```
int number; scanf("%d", &number);
```

In this example we have declared an integer called number.

The “%d” in scanf allows the function to recognise user input as being of an integer data type, which matches the data type of our variable number. The ampersand (&) allows us to pass the address of variable number which is the place in memory where we store the information that scanf read.

Example 2:

```
char word[256]; scanf("%s",  
word);
```

In this example we have declared a string called word.

The “%s” in scanf allows the function to recognise user input as being of string data type, which matches the data type of our variable word.

Since we’re using a string here we require the header file <string.h>, but unlike the example of the integer before, we do not require the ampersand (&) since string is an **array** of characters. An array is already a pointer to a character data type in memory as we have already learned before. Using the ampersand, the above example would be equivalent to:

```
for(int i=0;i<256;i+=1){  
    scanf("%s", &word[i]);  
}
```

Some extra uses of scanf

In a past assignment we were required to read strings one at a time without the \n so that we could print them all on one line.

Square brackets in scanf allow us to define which characters to read from input.

Example 1:

```
scanf("%[JASON]s", string);
```

In this example the program uses scanf to ask user for a string. [JASON] determines the characters the program will only take the characters J, A, S, O and N.

So if the input were “SON”, variable S would become “SON”. Whereas if the input were “SANE”, variable S would become “SAN”.

Example 2:

```
scanf("%[^\n]s", string);
```

This example demonstrates part of the aforementioned assignment. Here we have [^\n]. The not operator (^) is used on the character \n, causes scanf to read everything but the character \n – which is automatically added when you press return after entering input.

scanf also allows us to format our input in various ways. We can ignore preceding white spaces or determine the number of integers we would like to read from the user’s input.

Example 3:

```
scanf(" %s", string);
```

In this example white space preceding %s, means that any leading whitespaces before the string will be ignored.

So if the input was " whoops", our variable string would only take "whoops".

Example 4:

```
scanf("%2d", input);
```

In this example we have a number located between the % and d, which in this case is 2. The number determines the number of integers our variable input (of integer type) will read.

So if the input was "3222", our variable would only read "32".

printf

Function arguments:

int printf(const char * format, ...) printf like scanf, also requires format

placeholders.

Here are a table of some of the most commonly used placeholders:

PLACEHOLDER	DATATYPE
%d	integer
%u	unsigned int
%f	floating-point
%lf	double floating-point
%c	character
%s	String

Example 1:

```
printf("hello");
```

In this example, the only argument we need in here is the string we're trying to print.

This will print to stdout the string input.

Example 2:

```
int number = 9; printf("%d", number);
```

In this example we have declared an integer called number, which takes the value 9. The "%d" in printf allows the function to recognise the variable number as being of integer data type.

Our output would simply be 9.

Example 3:

```
char letter = 'a'; int number = 9;  
printf("%c\n%d", letter, number);
```

In this example we demonstrate how to print multiple variables.

Here, %c corresponds to our character variable called letter, and %d corresponds to our integer variable called number. We have included \n--- a newline character – which will print a newline to separate our two variables.

Our output will look like this: a9

Example 4:

```
int x = 10; printf("%5d",x);
```

In this example we have declared an integer x with a value of 10. The number located between % and d represents the indent value. The example above will print: ' 10'.

Example 5:

```
int* pointer = 5; printf("%p",pointer);
```

In this example we have declared a pointer to an integer in memory and assigned a value of 5 to it. We use "%p" to print to stdout the memory address where pointer points to. This is also the same address that the integer 5 is stored at. Memory addresses that print to stdout are actually

just hex numbers. What will print out is different for every computer and different every time that this function is used.

Special & Important printfss printf

Example 1:

```
int n = sprintf(storageString,"string"); printf("%s",storage);
```

This function is used along with strings. There are at least 2 parameters. The first parameter is where the string will be stored; the second parameter is the formatted string that needs to be printed. The formatted string that is defined in the same way that printf is with its respective placeholders. The possible multi-parameter that are needed in the function are defined the same way as printf. On success, this function returns an integer indicating how long the formatted string is. It is also followed by using printf to print the string to stdout.

fprintf

Example 1:

```
int n = fprintf(storageFile,"string");
```

This function is used along with file reading and writing. Similar to sprintf, it takes in at least 2 parameters and returns an integer indicating how long the formatted string is. One of the big differences between sprintf and fprintf is that the first parameter is not a string where the formatted string will be stored, but a file where the formatted string will be printed. Instead of printing to stdout, the formatted string will appear in the file, similar to the >out.txt option in shell commandline.

HAPTER 7

GRAPHICS PROGRAMS

To create a text file that contains a list of the contents in a folder, use one of the following methods.

How to Create a File List at a Command Prompt

1. Click Start, point to Programs, and then click MS-DOS Prompt (or Command Prompt in Windows NT).
2. At a command prompt, locate the drive that contains the folder whose contents you want to list. For example, if you want to create a text file that contains a list of the contents of a folder on drive C, type the following command at a command prompt, and then press ENTER:

```
c:
```

3. At a command prompt, locate the folder whose contents you want to list. For example, if you want to create a text file that contains a list of the contents in the Windows folder on drive C, type the following commands at a command prompt, and press ENTER after you type each command:

```
cd\  
cd windows
```

4. Type the following command at a command prompt, and then press ENTER, where **filename** is the name of the text file that you are creating:

```
dir>filename.txt
```

For example, if you want to create a file named Windowsfolderlist.txt, type the following command at a command prompt, and then press ENTER:

```
dir> windowsfolderlist.txt
```

NOTE: The text file that you create is located in the folder that you are in when you follow these steps. In the earlier example, the Windowsfolderlist.txt file is located in the Windows folder.

5. Use a text editor, such as Notepad, to view or print this file.

NOTE: You cannot export or print a list of the files that are contained in a folder in Windows Explorer.

In this program we read how to draw a pixel on graphics by using putpixel

```
/*C graphics program to draw a line.*/
```

```
#include <graphics.h>
```

```

#include <conio.h>

Int main()
{
    int gd = DETECT, gm;

    //init graphics
    initgraph(&gd, &gm, "C:/TURBOC3/BGI");

    /*

    if you are using turboc2 use below line to init graphics:
    initgraph(&gd, &gm, "C:/TC/BGI");

    */

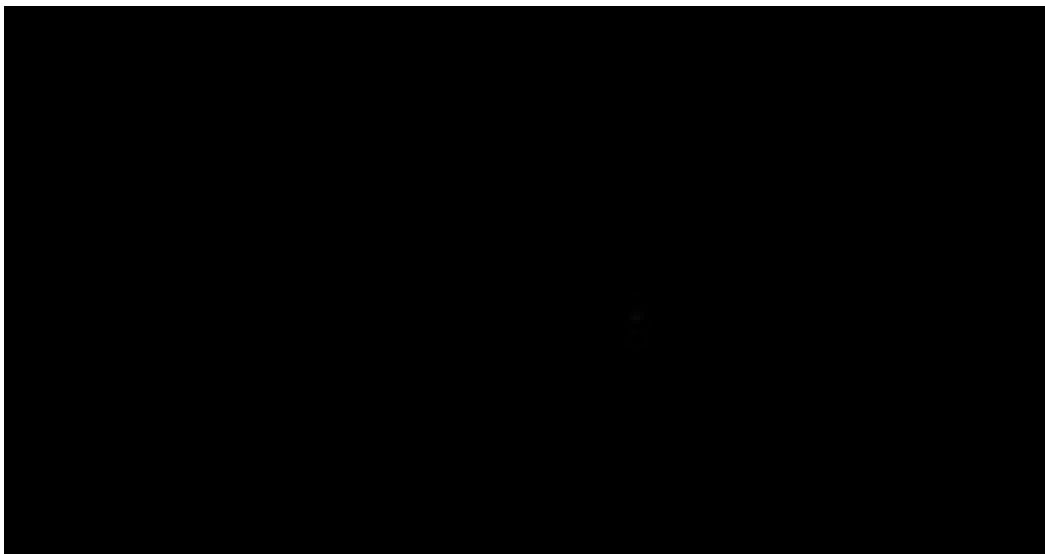
    //draw a pixel
    putpixel(100,100,RED); //will draw a red pixel

    getch();
    closegraph();

    return 0;
}

```

Output:



WAP for making a triangle by using c programming

```
#include<stdio.h>

#include<conio.h>

#include<graphics.h>

void main()

{

    int gd = DETECT, gm;

    initgraph(&gd, &gm, "c:\\tc\\bgi");

    line(300, 100, 200, 200); // (from_x,from_y,to_x,to_y)

    line(300, 100, 400, 200);

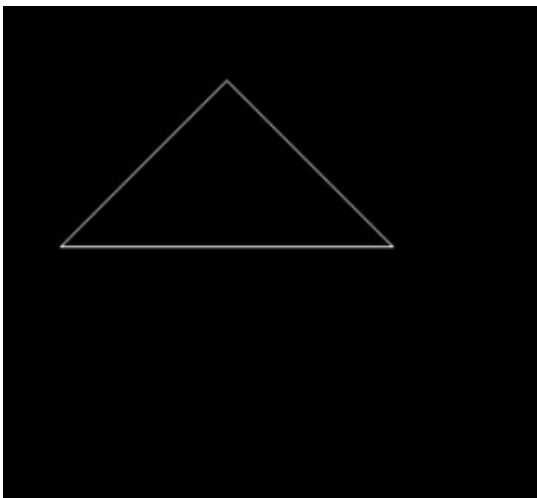
    line(200, 200, 400, 200);

    getch();

    closegraph();

}
```

Output



how to draw a line using programming using line() function of graphics.h header file?

In this example we will draw two horizontal lines using line() function of graphics.h.

line() in c programming:

line() is a library function of graphics.c in c programming language which is used to draw a line from two coordinates.

For example if you want to draw a line from point(x1,y1) to point(x2,y2) you have to use line() function like line(x1,y1,x2,y2);

Syntax (Declaration of line() function in C)

```
line(int x1,int y1, int x2,int y2);
```

C Code Snippet - Graphics program to Draw a line using line() and lineto()

Let's consider the following example:

```
/*C graphics program to draw a line.*/
```

```
#include <graphics.h>
```

```
#include <conio.h>
```

```
main()
```

```
{ int gd = DETECT, gm;
```

```
    //init graphics
```

```
initgraph(&gd, &gm, "C:/TURBOC3/BGI");
```

```
    /*
```

```
    if you are using turboc2 use below line to init graphics:
```

```
initgraph(&gd, &gm, "C:/TC/BGI");
```

```
    */
```

```
    //draw a line
```

```
    /*
```

```
    line() function description
```

```
    parameter left to right
```

```
    x1: 100
```

```

y1: 100

x2: 200

y2: 100

*/

line(100,100,200,100); //will draw a horizontal line

line(10,10,200,10); //will draw another horizontal line

getch();

closegraph();

return 0;

```

} Output



Draw a line from current point to point(x,y) using lineto() function

lineto() is a library function of graphics.h and it can be used to draw a line from current point to specified point.

For example, if you want to draw a line from current point to point(x,y), you have to use lineto() function like lineto(x,y)

Syntax (Declaration of line() function in C)

```
lineto(int x, int y);
```

In this program we will draw a line from current point to point (100,200).

```
#include <graphics.h>
```

```

#include <conio.h>

main()
{
    int gd = DETECT, gm;

    //init graphics
    initgraph(&gd, &gm, "C:/TURBOC3/BGI");

    /*
    if you are using turboc2 use below line to init graphics:
    initgraph(&gd, &gm, "C:/TC/BGI");
    */

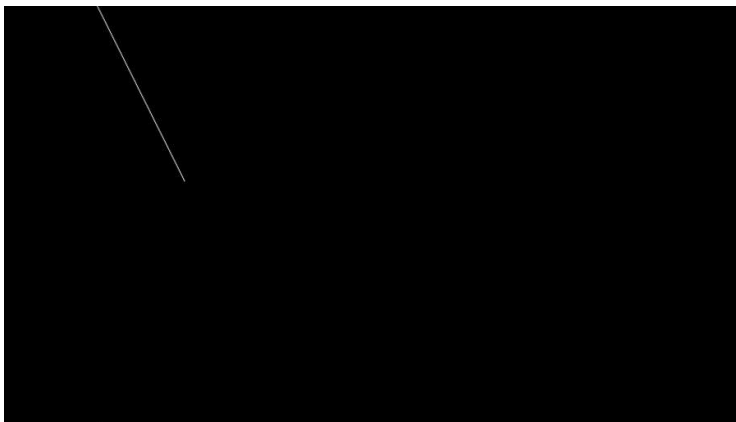
    //draw a line
    lineto(100,200); //will dra a line to point(100,200);

    getch();
    closegraph();

    return 0;
}

```

Output



A video adapter is a board that plugs into a personal computer to give it display capabilities. The display capabilities of a computer, however, depend on both the logical circuitry

(provided in the video adapter) and the display monitor. A monochrome monitor, for example, cannot display colors no matter how powerful the video adapter.

Many different types of video adapters are available for PCs. Most conform to one of the video standards defined by IBM or VESA.

Each adapter offers several different video modes. The two basic categories of video modes are *text* and *graphics*. In text mode, a monitor can display only ASCII characters. In graphics mode, a monitor can display any bit-mapped image. Within the text and graphics modes, some monitors also offer a choice of resolutions. At lower resolutions a monitor can display more colors.

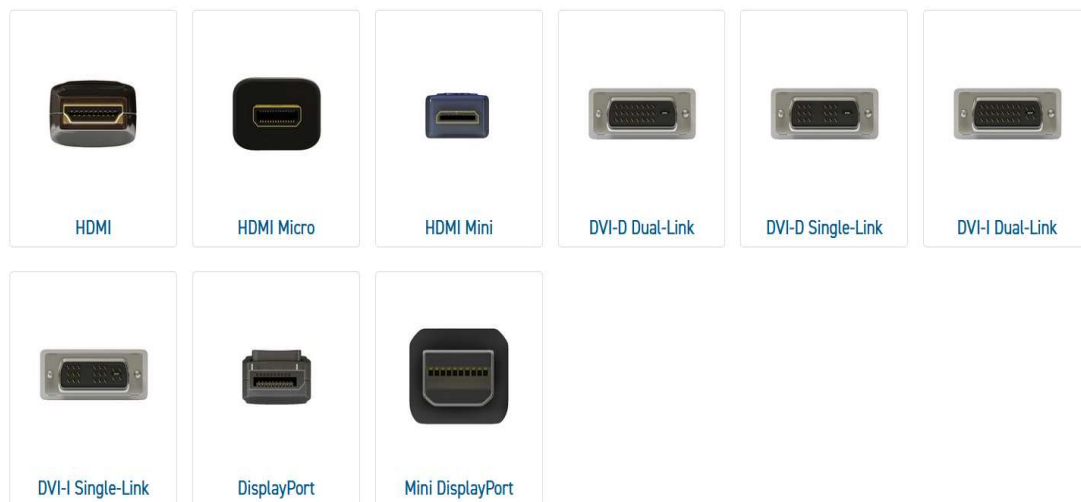
Modern video adapters contain memory, so that the computer's RAM is not used for storing displays. In addition, most adapters have their own graphics coprocessor for performing graphics calculations. These adapters are often called *graphics accelerators*.

Video adapters are also called *video cards*, *video boards*, *video display boards*, *graphics cards* and *graphics adapters*.

Types of Video Adapters

Different video adapters are available that are used for different resolutions and reasons. From component cables to HDMI cords, there are many ways to transport video signals to your television, monitor or projector. Some adapters provide a better image than others, depending on your needs.

 Select the digital video connector that you want to learn more about:





Contact Us:

University Campus Address:

Jayoti Vidyapeeth Women's University

Vadaant Gyan Valley, Village-Jharna, Mahala Jobner Link Road,
Jaipur Ajmer Express Way, NH-8, Jaipur- 303122, Rajasthan (INDIA)

(Only Speed Post is Received at University Campus Address, No. any Courier Facility is available at Campus Address)

Pages : 48
Book Price : ₹ 150/-



Year & Month of Publication- 3/3/2021