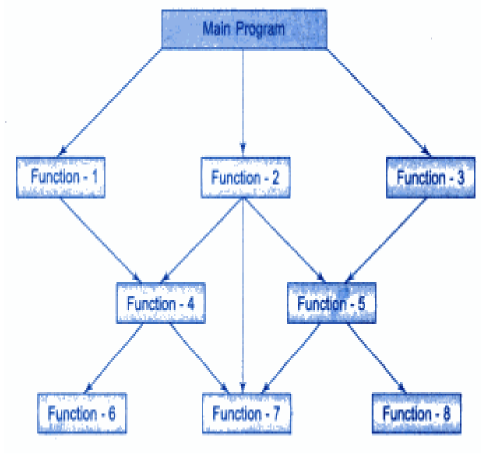
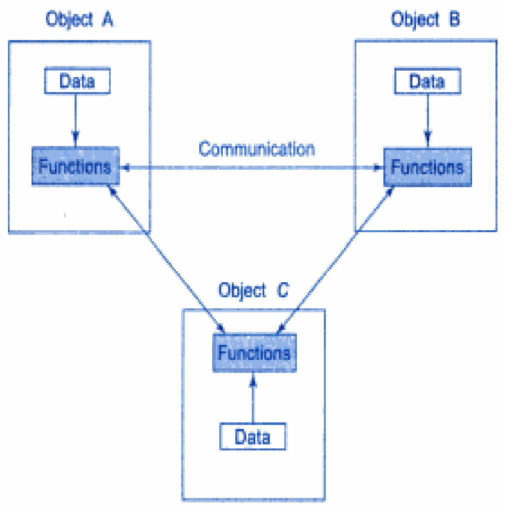
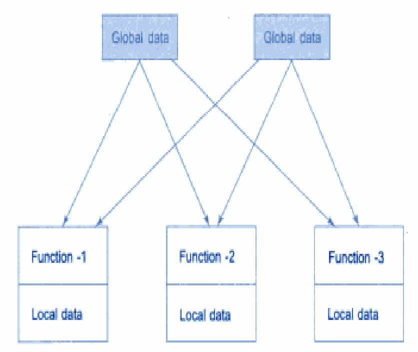
1.1 Procedure Oriented Programming (POP) Vs Object Oriented Programming (OOP)

Features	Procedure Oriented Programming	Object Oriented Programming
Structure	 Main Program Function - 1 Function - 2 Function - 3 Function - 4 Function - 5 Function - 6 Function - 7 Function - 8	 Object A Object B Data Data Functions Communication Functions Object C Functions Data
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	POP follows Top Down approach.	OOP follows Bottom Up approach.
Access Specifiers	POP does Not have any access specifiers.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.

	 Global data Global data Function -1 Local data Function -2 Local data Function -3 Local data	
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding S provides more security. o
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP is: C, VB, FORTRAN, and Pascal.	Example of OOP is: C++, JAVA, VB.NET, C#.NET.

1.2 Basic Concepts of Object Oriented Programming

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. **OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function**. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects.
Some of the features of object oriented programming are:

- ✓ Emphasis is on data rather than procedure.
- ✓ Programs are divided into what are known as objects.
- ✓ Data structures are designed such that they characterize the objects.
- ✓ Functions that operate on the data of an object are ties together in the data structure.
- ✓ Data is hidden and cannot be accessed by external function.
- ✓ Objects may communicate with each other through function.
- ✓ New data and functions can be easily added whenever necessary.
- ✓ Follows bottom up approach in program design.

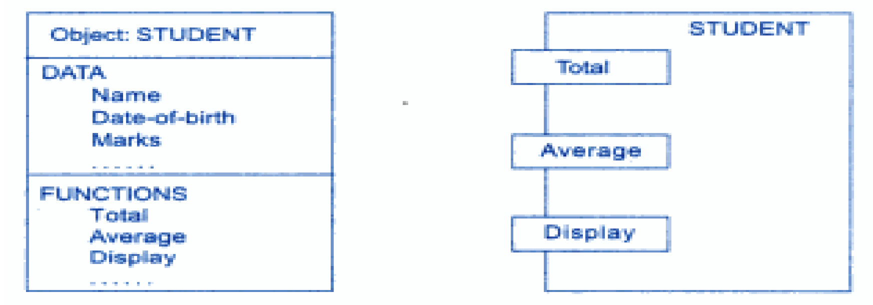
"Object Originated Programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and function that can be used as templates for creating copies of such modules on demand."

Basic Concepts of Object Oriented Programming

- ✓ Objects
- ✓ Classes
- ✓ Data abstraction and encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Dynamic binding
- ✓ Message passing

Objects

- Objects are the basic runtime entities in object oriented system. Objects are also called Instance of the class.
- An object represents a person, Bank-account, Vehicle, Book, products, Employee, etc…
- Programming problem is analyze in terms of object and nature of communication between them
- In program objects should choose and design in such a way that they match closely with real world object
- Objects occupies memory and they have associated address



(Two ways to represent object)

Classes

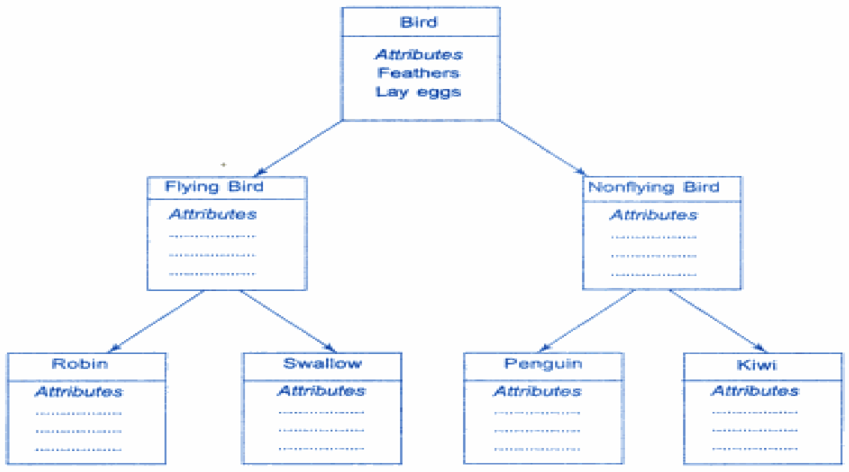
- The general meaning of class is category. Class is a pro-forma or blue-print that represents particular category.
- Class contains data and code (function) that operate on that data.
- It is user defined data type and object are variable of class.
- Once we define a class we can create any number of objects belonging to that class.
- So object is associated with data of type class thus, class is collection of similar type of objects.
- If fruit has been defines as a class, then the statement **Fruit Mango;** Will create an object mango belonging to the class fruit.

Data Abstraction and Encapsulation

- The wrapping up of data and function into a single unit (called class) is known as **encapsulation**.
- Data encapsulation is the most important feature of class with this concept data is not accessible to the outside class definition and only those functions which are wrapped in the class can access it for this to happen data members must be declare as **private**.
- **Public** members provides interface between the object, data and the program.
- It is also called **"Data hiding"** or **"Information hiding".**
- **Data abstraction** refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
- Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight, and cost and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.
- The attributes are some time called **data members** because they hold information. The functions that operate on these data are sometimes called **methods or member function**.

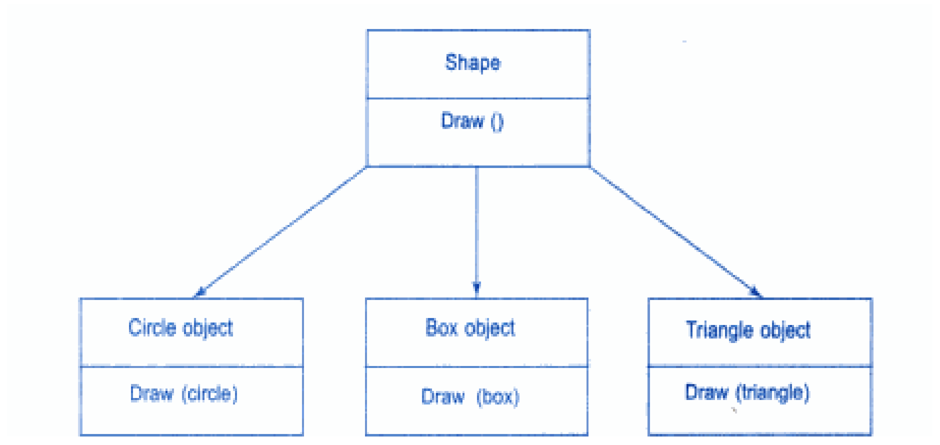
Inheritance

- Inheritance is the process by which the objects of one class acquire the properties of object of another class.
- It supports the concept of hierarchical – classification.
- In Oop, the concept of inheritance provides the idea of Reusability. It means that we can add additional features to an existing class without modifying it.
- It is possible by deriving a new class from existing class and new class will have combined features of both the class



➢ Polymorphism

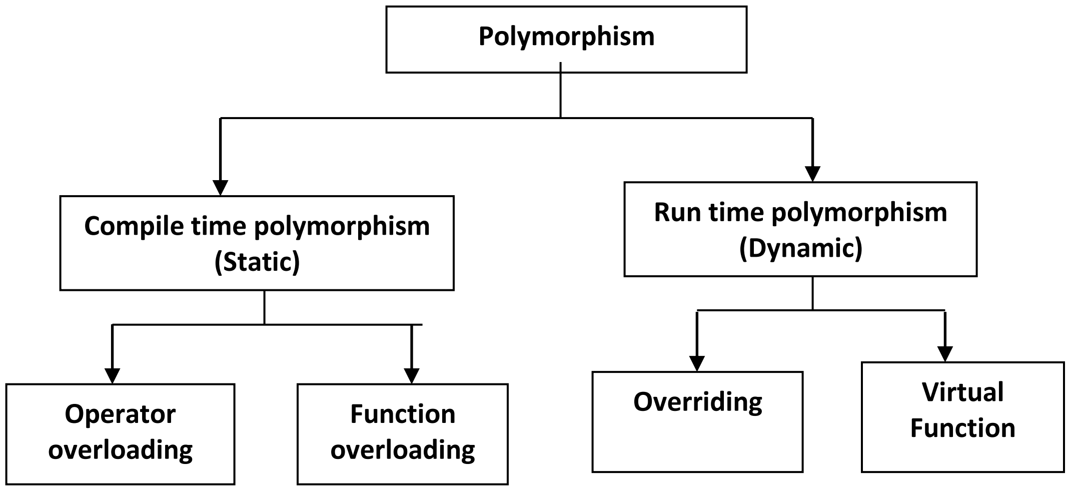
- Polymorphism simply means "One name And Multiple behavior".
- Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation.
- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ.
- Polymorphism is extensively used in implementing inheritance.



.

For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading**.

A single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as **function overloading**.



➢ **Dynamic Binding (Late binding)**

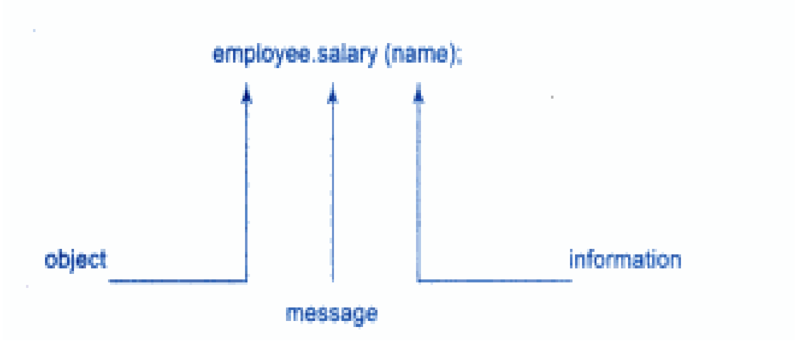
- Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions
- Dynamic binding also called **dynamic dispatch** is the process of linking procedure call to a specific sequence of code (method) at run-time. It means that the code to be executed for a specific procedure call is not known until run-time. Dynamic binding is also known as *late binding* or *run-time binding*

➢ **Message Passing**

- An object oriented program consists of set of objects that communicate with each other
- **Message Passing** is nothing but sending and receiving of information by the objects same as people exchange information. So this helps in building systems that simulate real life.
- Following are the basic steps in message passing.

  1. Creating classes that define objects and its behavior.
  2. Creating objects from class definitions
  3. Establishing communication among objects

- In OOPs, Message Passing involves specifying the name of objects, the name of the function, and the information to be sent
- Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.



1.3 Benefits of Object Oriented Programming

- Through inheritance, we can eliminate redundant (repeated) code and extend the use of existing classes which is not possible in procedure oriented approach.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch which happens procedure oriented approach. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of object to co-exist without any interference.

- It is possible to map objects in the problem domain to those in the computer program.
- It is easy to partition the work in a project based on objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

APPLICATION OF OBJECT ORIENTED PROGRAMMING

- System software
- Application software
- Real time systems
- Object oriented database
- Operating system
- Compiler
- Device driver

C VS. C++ :

C	**C++**
C is Procedural Language.	C++ is non Procedural i.e Object oriented Language.
No virtual Functions are present in C.	The concept of virtual Functions is used in C++.
In C, Polymorphism is not possible.	The concept of polymorphism is used in C++.Polymorphism is the most Important Feature of OOPS.
Operator overloading is not possible in C.	Operator overloading is one of the greatest Feature of C++.
Top down approach is used in Program Design.	Bottom up approach adopted in Program Design.
Multiple Declarations of global variables are allowed.	Multiple Declarations of global variables are not allowed.
In C scanf() Function used for Input. printf() Function used for output.	In C++ cin>> Function used for Input. cout<< Function used for output.

Mapping between Data and Function is difficult and complicated.	Mapping between Data and Function can be used using "Objects".
In C, we can call main() Function through other Functions.	In C++, we cannot call main() Function through other functions.
C requires all the variables to be defined at the starting of a scope.	C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
No inheritance is possible in C.	Inheritance is possible in C++.
In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	In C++, new and delete operators are used for Memory Allocating and Deallocating.
It supports built-in and primitive data types.	It support both built-in and user define data types.
In C, Exception Handling is not present.	In C++, Exception Handling is done with Try and Catch block.

1.4 Structure & Classes

- **Structure in C**

Structure provides a method to packing together data of different. Structure is user defined data type that serves to define its data properties. Once the structure is defined we can create variables of that type using declarations that are similar to built-in type declarations.

Example:

```
struct student
{
char name[20];
int roll_number;
float total_marks;
}
```

The keyword “struct” declares student as a new data type that can hold three of different data types. These fields are known as structure members, structure name can be used to create variables of type student as follows:

```
struct student A;    // C declaration
```

Member variables can be accessed as follows:

```
Strcpy(A.name,"John");
A.roll_number = 999;
A.total_marks = 595.5;
```

Structure can have arrays, Pointers or structures as members.

Extensions to Structures in C++

- ✓ C++ supports all the features of structures as defined in C.
- ✓ C++ provides facility to hide data (Using private) which is one of the main principal of OOP.
- ✓ In C++ a structure can have both variables and function as member.
- ✓ In C++ the structure names are stand alone and can be used like any other type name in following way :

```
student A;//          C++ declaration
```

"C++ incorporated all these extensions in another user-defined type Known as CLASS" There is a little syntactical difference in structure and Classes in C++ therefore; they can be used interchangeably with minor modification.

Note : The only difference between structure and classes in C++ is that ,by default, the member of a class are private, while by default the members of structure are public.

➢ Specifying a Class

A class in C++ combines related data and functions together. It makes a data type which is used for creating objects of this type.

Classes represent real world entities that have both data type properties (characteristics) and associated operations (behavior).

Generally a class specification has two parts:

- ✓ Class Declaration
- ✓ Class Function definition

Class Declaration

The syntax of a class definition is shown below:

```
Class  name_of _class
{
private :

variable declaration; // data member
Function declaration; // Member Function (Method)
protected:

Variable  declaration;
Function declaration;
public :

variable  declaration;
Function declaration;
};
```

Here, the keyword class specifies that we are using a new data type and is followed by the class name.

The body of the class has two keywords namely:

(i) private
(ii) (ii) public

In C++, the keywords private and public are called access specifiers. The data hiding concept in C++ is achieved by using the keyword private. Private data and functions can only be accessed from within the class itself. Public data and functions are accessible outside the class also.

The data declared under Private section are hidden and safe from accidental manipulation. Though the user can use the private data but not by accident.

The functions that operate on the data are generally public so that they can be accessed from outside the class but this is not a rule that we must follow.

- **Class Function definition**

In C++, the member functions can be coded in two ways:

(a) Inside class definition (IMPLICIT)
(b) Outside class definition using scope resolution operator (::) (EXPLICIT)

The code of the function is same in both the cases, but the function header is different as

Inside Class Definition:

When a member function is defined inside a class, we do not require to place a membership label along with the function name. We use only small functions inside the class definition and such functions are known as inline functions. In case of inline function the compiler inserts the code of the body of the function at the place where it is invoked (called) and in doing so the program execution is faster.

Outside Class Definition Using Scope Resolution Operator (::) :

- In this case the function's full name (qualified_name) is written as shown:

  Name_of_the_class :: function_name

- The syntax for a member function definition outside the class definition is

  return_type name_of_the_class::function_name (argument list)

  { body of function }

Here the operator:: known as scope resolution operator helps in defining the member function outside the class. Earlier the scope resolution operator (::) was use in situations where a global variable exists with the same name as a local variable and it identifies the global variable.

Example of Class :

```
Class student
{
private:

char reg_no;
char name[30];
int age;


public :

void init_data()                    //Inside Class Defenation
{
- - - - - //body of function - - - - -
}

void display_data();

};

void student :: display_data()          //Outside class Defination
{
----------//body of function -------
}

Student  ob;     //class variable (object) created
Ob.init_data();   //Access the member function
```

ob.display_data();//Access the member function

<Iostream.h> file

- iostream.h is the header file
- It is generally included at the beginning of all c++ programs that use input and output statements.
- This file contains declarations for cin and cout objects and << (insertion)and >>(extraction) operators

Output operator

- cout << "Hello cpp " ;
- Identifier cout is pre-defined object that represents standard output stream in c++.
- In this example , by default standard output device is monitor or screen
- It is also possible to redirect the output to other output device and it does not need format specifier
- The operator << is called "insertion" or "put to" operator.

Input operator

- cin >> number ;
- Identifier cin is pre-defined object that represents standard input stream in c++.
- In this example , by default standard input device is keyboard
- This input statement causes the program to wait for the user to type value for number and user entered number is placed in any variable with the help of cin object and extraction operator.
- The operator >> is called "extraction" or "get from" operator.

Local And Global Variable declaration

- In c++ if we have global variable with the same name as local variable then to access the global variable c++ provides a special operator known as scope resolution operator (::)
- Example:

```
Int x=10;
Void main()
{
Int x=3;
Cout << x;    // print local variable x value 3
Cout << ::x; // print global variable x value 10
getch();
}
```

References

- In c++ references are variable that contains link to specific value
- References variable permits us to pass parameters to the function by reference
- When we pass argument by reference the formal argument in call function becomes Elias to the actual argument in calling function
- It means that , when function is working with its own argument it is actually working on original data.
- A reference variable always created by & in front of any variable
- A reference variable points to the same address as the variable
- Reference variable does not utilize any additional space like pointer variable and still provides the functionality of pointer.

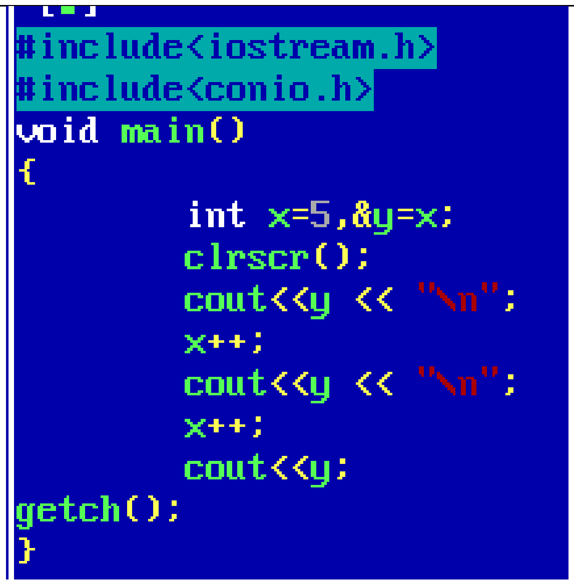
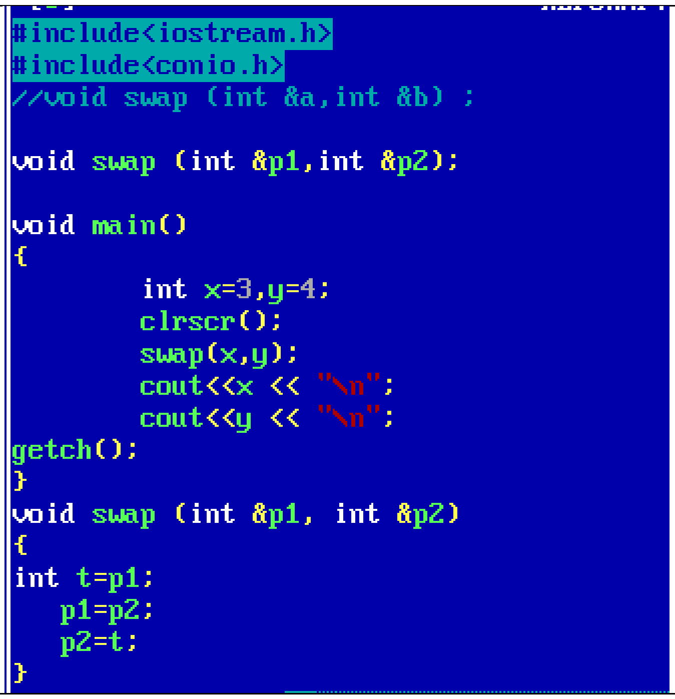
5

X

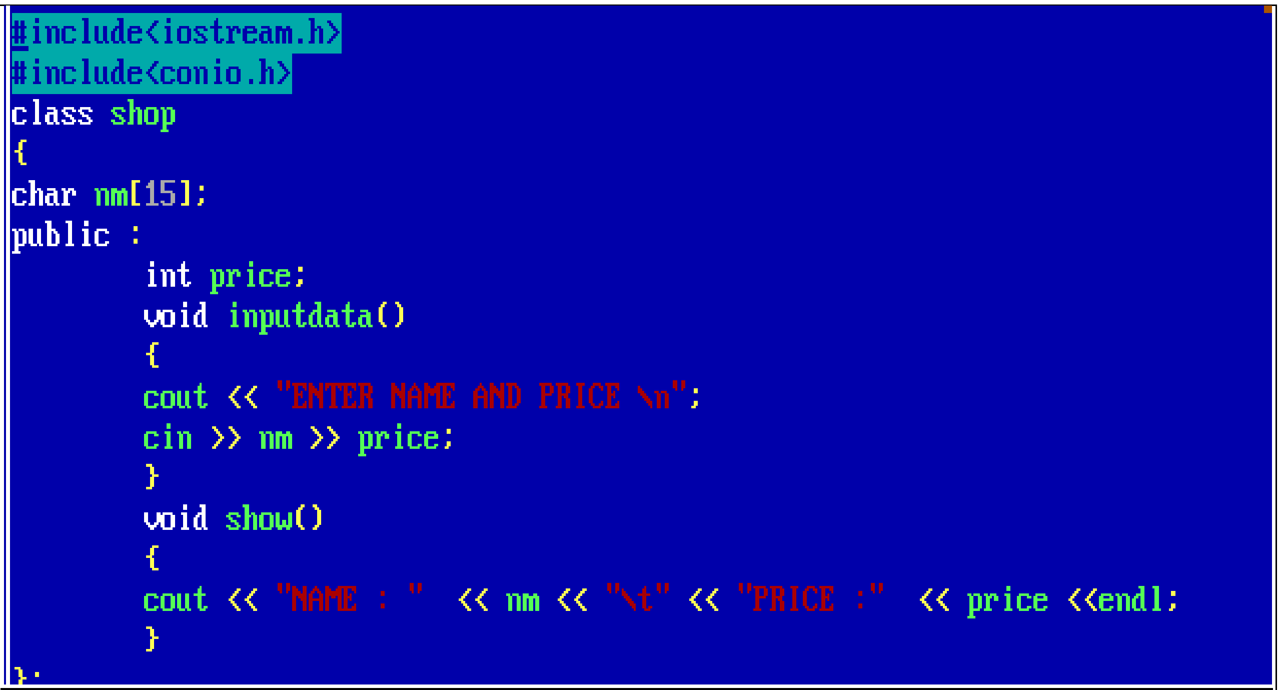
100

&y

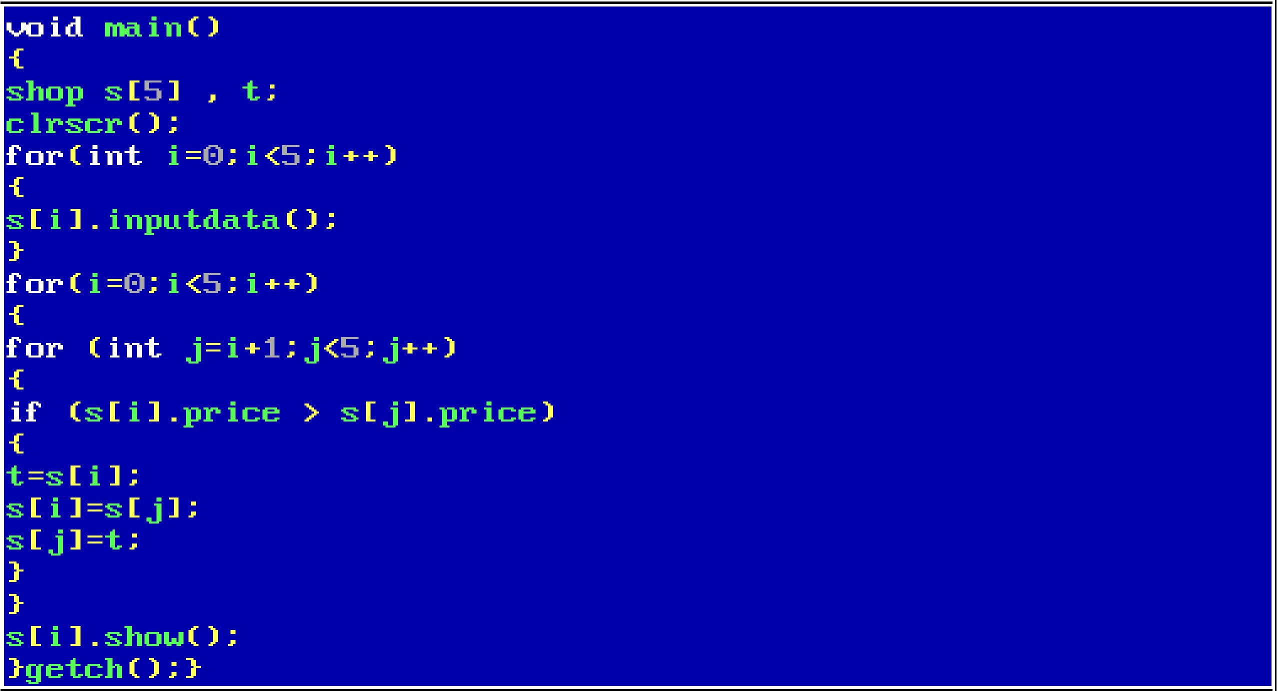
Reference of x

Program 1 :	Program 2 : (swapping of value)
 `#include<iostream.h>` `#include<conio.h>` `void main()` `{` `int x=5,&y=x;` `clrscr();` `cout<<y << "\n";` `x++;` `cout<<y << "\n";` `x++;` `cout<<y;` `getch();` `}`	 `#include<iostream.h>` `#include<conio.h>` `//void swap (int &a,int &b) ;` `void swap (int &p1,int &p2);` `void main()` `{` `int x=3,y=4;` `clrscr();` `swap(x,y);` `cout<<x << "\n";` `cout<<y << "\n";` `getch();` `}` `void swap (int &p1, int &p2)` `{` `int t=p1;` `p1=p2;` `p2=t;` `}`
Output : 5 6 7	Output : x=4 Y=3

Class with array (array of object) (sorting according to price in ascending order)

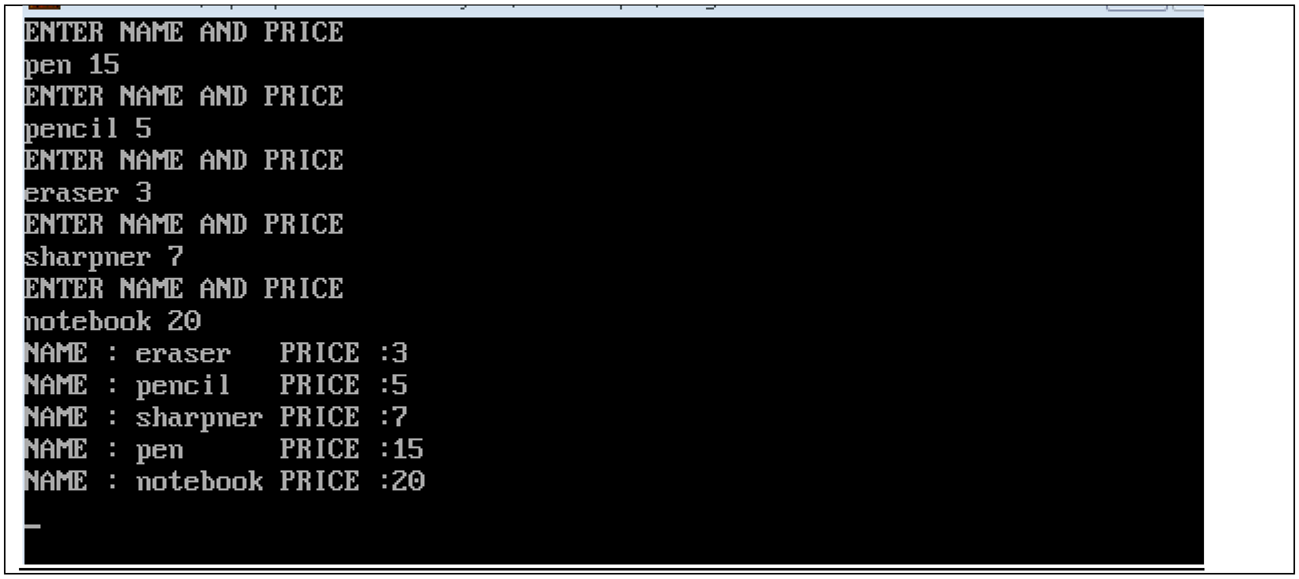


```
#include<iostream.h>
#include<conio.h>
class shop
{
char nm[15];
public :
        int price;
        void inputdata()
        {
        cout << "ENTER NAME AND PRICE \n";
        cin >> nm >> price;
        }
        void show()
        {
        cout << "NAME : "  << nm << "\t" << "PRICE :"  << price <<endl;
        }
};
```



```
void main()
{
shop s[5] , t;
clrscr();
for(int i=0;i<5;i++)
{
s[i].inputdata();
}
for(i=0;i<5;i++)
{
for (int j=i+1;j<5;j++)
{
if (s[i].price > s[j].price)
{
t=s[i];
s[i]=s[j];
s[j]=t;
}
}
s[i].show();
}getch();}
```

Output :



```
ENTER NAME AND PRICE
pen 15
ENTER NAME AND PRICE
pencil 5
ENTER NAME AND PRICE
eraser 3
ENTER NAME AND PRICE
sharpner 7
ENTER NAME AND PRICE
notebook 20
NAME : eraser   PRICE :3
NAME : pencil   PRICE :5
NAME : sharpner PRICE :7
NAME : pen      PRICE :15
NAME : notebook PRICE :20
_
```

<u>**Local class**</u>

- It is the class which is defined inside the function or block
- Local classes can use global variables and static variable declare inside the function but cannot use automatic local variables
- Local classes cannot have static data members
- Member function must be define inside the class
- Enclosing function cannot access private members of a local class
- You can create and use object of local class only inside enclosing function
- <u>**Example:**</u>

```
void fun()
{
// Local class
class Test
{
/* ... */
};


Test t;  // Fine
Test *tp;  // Fine
}


void main()
{
Test t;  // Error
Test *tp;  // Error
getch();
}
```

Static keyword

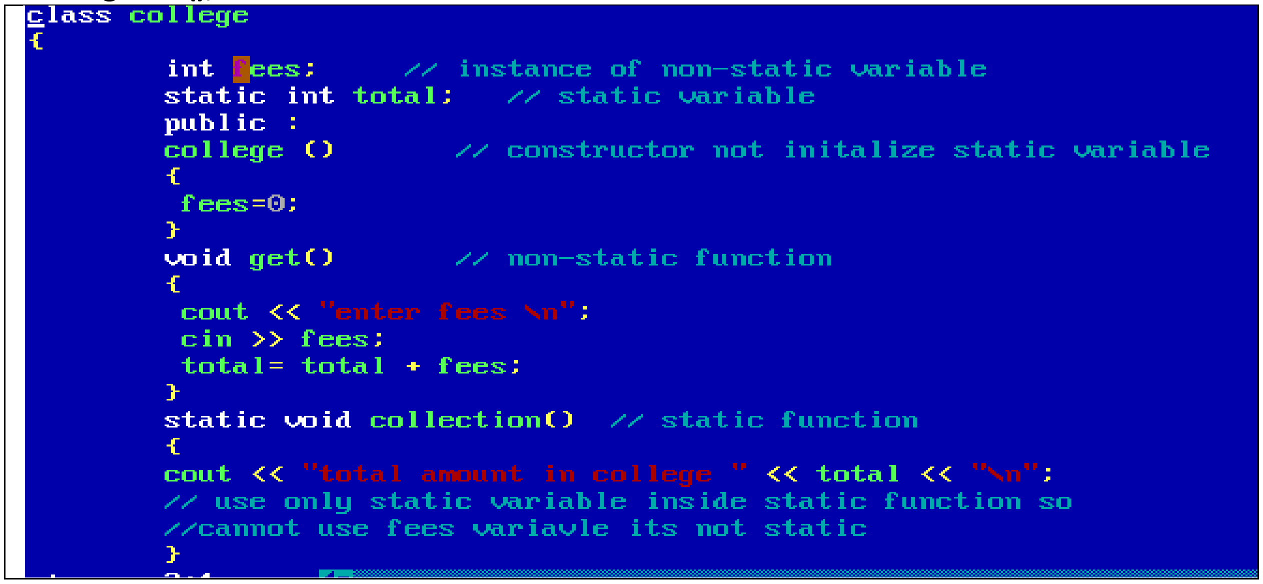
- In c++ “static” keyword can be used with variable as well as function
- We can access static variable and function using scope resolution operator (::)

1. Static data members:

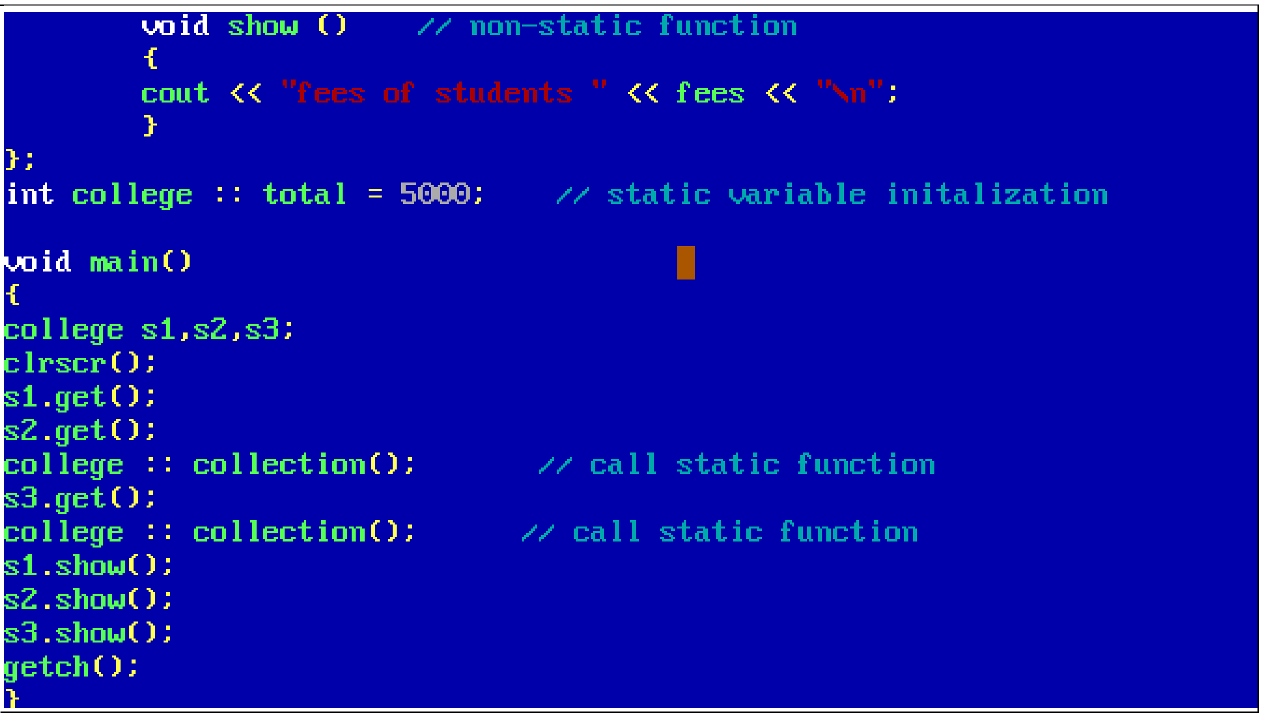
- Data members of class can be declared as static the main characteristic of static member variables are similar to characteristic of c programming static variable.
- **Characteristics:**
- It is initialize with 0 (zero) when the first time object of class is created
- Only one copy of that members is created for the entire class and it is shared by all the objects
- It is visible only within the class but its life time is entire program
- Static variables are normally used to maintain values common to entire class
- Data type and scope of each static member variable must define outside of the class definition its necessary because static data members are stored separately rather than as part of object like,
  Int test :: count;
- Static variables are associated with class rather than any object so they are called class variables

2. **Static member function:**

- Like static member variables we can also have static member function
- **Characteristics:**
- Static function can have access to only other static data members and static member function declare in the same class
- Static member function can be called using class name instead with the name of object like,
  test :: getcnt();

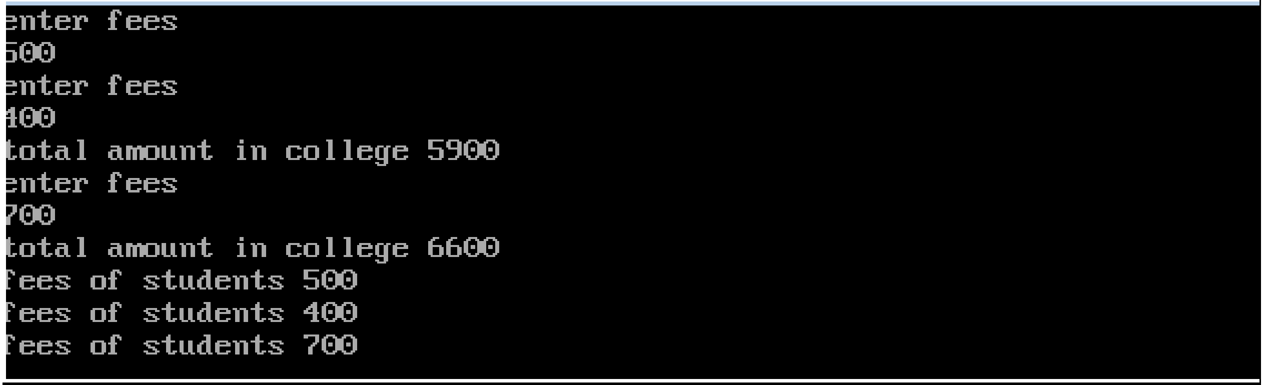


```
class college
{
        int fees;      // instance of non-static variable
        static int total;   // static variable
        public :
        college ()         // constructor not initalize static variable
        {
         fees=0;
        }
        void get()         // non-static function
        {
         cout << "enter fees \n";
         cin >> fees;
         total= total + fees;
        }
        static void collection()  // static function
        {
        cout << "total amount in college " << total << "\n";
        // use only static variable inside static function so
        //cannot use fees variavle its not static
        }
```



```
        void show ()    // non-static function
        {
        cout << "fees of students " << fees << "\n";
        }
};
int college :: total = 5000;    // static variable initalization

void main()
{
college s1,s2,s3;
clrscr();
s1.get();
s2.get();
college :: collection();        // call static function
s3.get();
college :: collection();       // call static function
s1.show();
s2.show();
s3.show();
getch();
}
```

Output :



```
enter fees
500
enter fees
400
total amount in college 5900
enter fees
700
total amount in college 6600
fees of students 500
fees of students 400
fees of students 700
```

1.5 Encapsulation and Data Hiding

Refer "**Data abstraction and encapsulation**" in**Basic Concepts of Object Oriented Programming.**

1.6 Constructors And Destructors

- A constructor (having the same name as that of the class) is a special member function which is automatically initializing the data-members (variables) of the class type with legal initial values.
- It called automatically when the object of class is created
- It's called constructor because it construct values of data members of the class

Declaration and Definition of a Constructor:

- It is defined like other member functions of the class, i.e., either inside the class definition or outside the class definition. For example, the following program illustrates the concept of a constructor:

```
//To demonstrate a constructor
Class rectangle
{
private :
float length, breadth;

public:

rectangle ()    //constructor definition
{
//displayed whenever an object is created

cout<< ”I am in the constructor”;
length=10.0;
breadth=20.5;

}

float area()
{
return (length*breadth);
}
};
void main()
{
clrscr();

rectangle rect; //object declared
cout<< ”\nThe area of the rectangle with default parameters is:”<< rect.area();

 getch();}
```

➢ **SPECIAL CHARACTERISTICS OF CONSTRUCTORS**

These have some special characteristics. These are given below:

- ✓ These are called automatically when the objects are created.
- ✓ All objects of the class having a constructor are initialized before some use.
- ✓ These should be declared in the public section for availability to all the functions.
- ✓ Return type (not even void) cannot be specified for constructors.
- ✓ These cannot be inherited, but a derived class can call the base class constructor.

- ✓ These cannot be static.
- ✓ Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.
- ✓ These can have default arguments as other C++ functions.
- ✓ A constructor can call member functions of its class.
- ✓ An object of a class with a constructor cannot be used as a member of a union.

➢ **Types of Constructors**

1. **Default constructor:**
    - Constructor does not accept any argument is called default constructor

2. **Parameterized constructor**
    - C++ allows us to initialize objects by passing argument to constructor function when objects are created
    - So, the constructor that can take argument is called parameterized constructor
    - It can be invoke implicitly or explicitly
    - Like function you can pass default argument in constructor also which known as **constructor with default argument**
    - All the rules of default argument passing apply to parameterized constructor with default argument
    - **Constructor overloading** means when we have more than one constructors in a same class
    - When you overload constructor and pass default argument at that time you need to take care that all the parameters are not given default values if default constructor exists otherwise, compiler will raise the error of ambiguity
    - To overcome error of ambiguity there are two alternatives
        1. Omit default constructor if all the argument are default
        2. Keep at least one argument mandatory if you want to define default constructor also
3. **Copy constructor**
    - Copy constructor is used to declare and initialize object from another object
    - Eg. Number obj1 (5) , obj2(obj1) ,obj3=obj1
    - Copy constructor takes a reference of an object of the same class as an argument.
    - We can pass argument by value to copy constructor

<u>Destructor:</u>

- The name of the class and destructor is same but it is prefixed with a ~ (tilde).
- It does not take any parameter nor does it return any value.
- Overloading a destructor is not possible and can be explicitly invoked.
- In other words, a class can have only one destructor.
- A destructor can be defined outside the class.

Declaration and Definition of a Destructor

The syntax for declaring a destructor is:

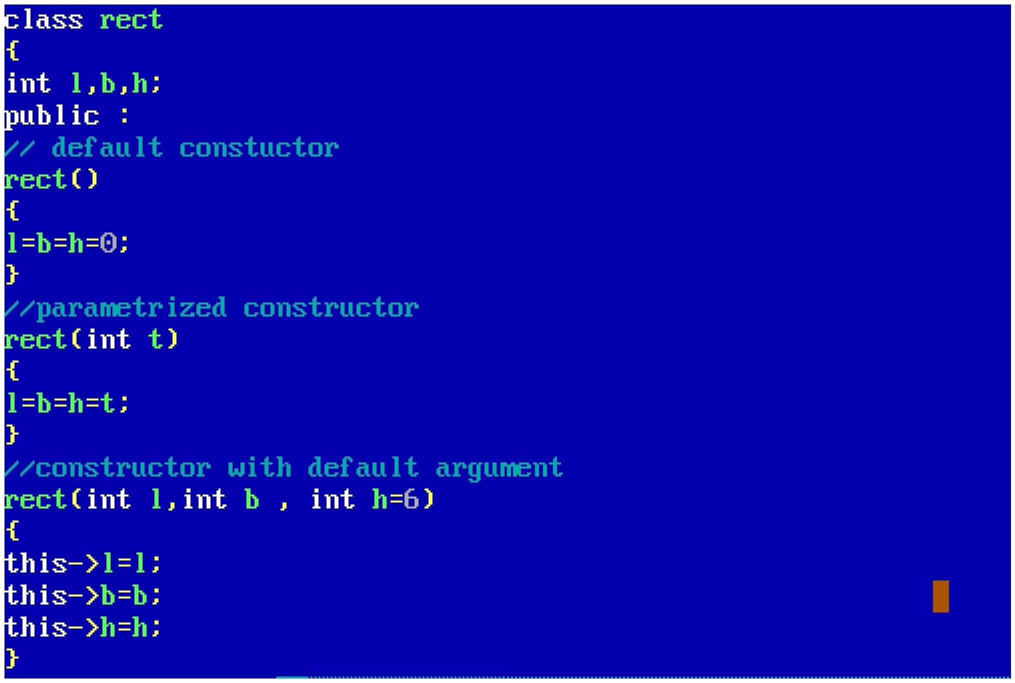
~name_of_the_class() { }

Special Characteristics of Destructors

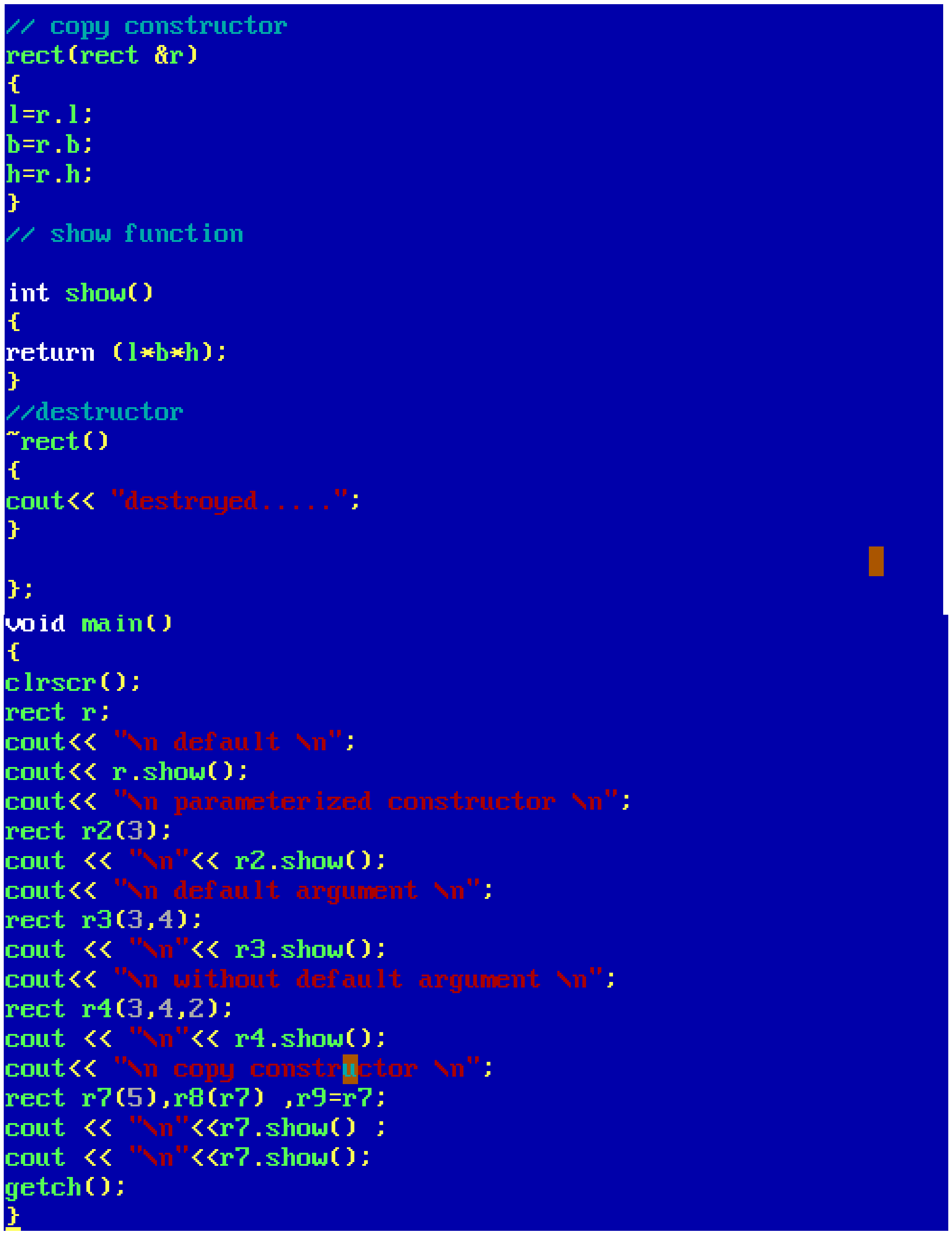
Some of the characteristics associated with destructors are:

- ✓ These are called automatically when the objects are destroyed.
- ✓ Destructor functions follow the usual access rules as other member functions.
- ✓ These de-initialize each object before the object goes out of scope.
- ✓ No argument and return type (even void) permitted with destructors.
- ✓ These cannot be inherited.
- ✓ Static destructors are not allowed.
- ✓ Address of a destructor cannot be taken.
- ✓ A destructor can call member functions of its class.
- ✓ An object of a class having a destructor cannot be a member of a union.

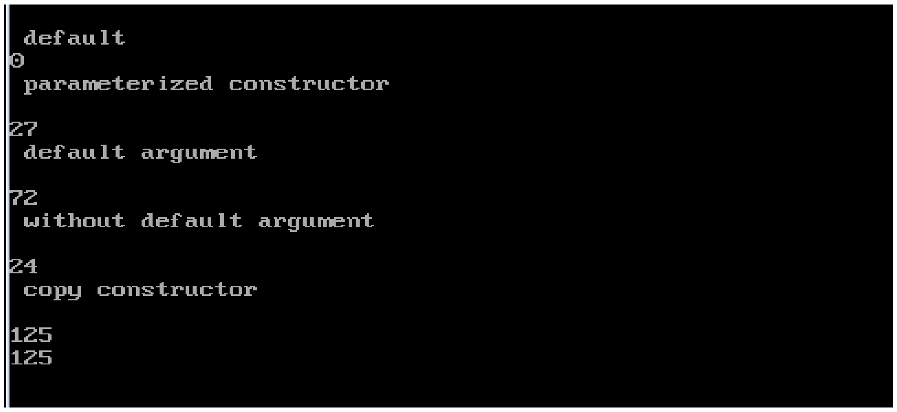
Note: The **this** pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class



```
class rect
{
int l,b,h;
public :
// default constuctor
rect()
{
l=b=h=0;
}
//parametrized constructor
rect(int t)
{
l=b=h=t;
}
//constructor with default argument
rect(int l,int b , int h=6)
{
this->l=l;
this->b=b;
this->h=h;
}
```



```
// copy constructor
rect(rect &r)
{
l=r.l;
b=r.b;
h=r.h;
}
// show function

int show()
{
return (l*b*h);
}
//destructor
~rect()
{
cout<< "destroyed.....";
}

};
void main()
{
clrscr();
rect r;
cout<< "\n default \n";
cout<< r.show();
cout<< "\n parameterized constructor \n";
rect r2(3);
cout << "\n"<< r2.show();
cout<< "\n default argument \n";
rect r3(3,4);
cout << "\n"<< r3.show();
cout<< "\n without default argument \n";
rect r4(3,4,2);
cout << "\n"<< r4.show();
cout<< "\n copy constructor \n";
rect r7(5),r8(r7) ,r9=r7;
cout << "\n"<<r7.show() ;
cout << "\n"<<r7.show();
getch();
}
```



```
 default
0
 parameterized constructor

27
 default argument

72
 without default argument

24
 copy constructor

125
125
```

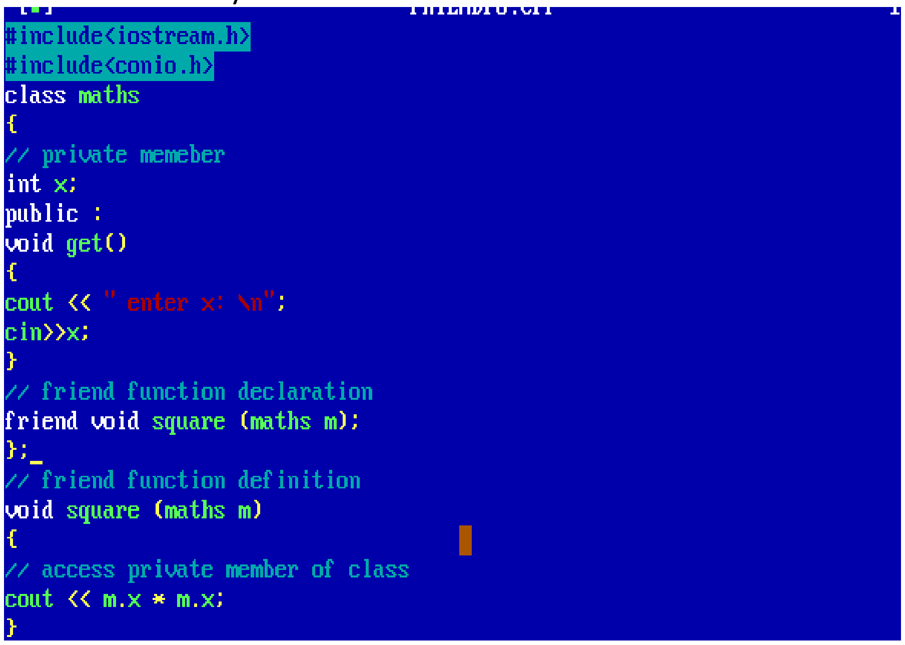
1.7 Friend Function

- Private members cannot be access outside of the class so; non-member functions cannot have access to private data of the class. But there can be a situation we would like to classes to share a particular function in such situation c++ allows common function to be made friendly with both classes.
- To make outside function friendly to the class we need to simply declare that function as a friend of the class
- For this function declaration should be presided by keyword **friend**
- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.
- A friend function can be declare as friend in any number of classes
- It has full access rights to private members of the class

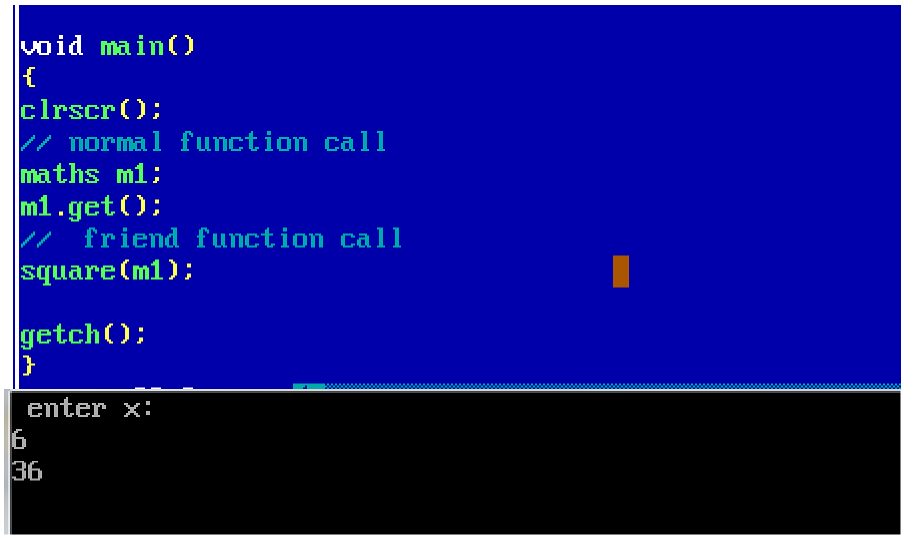
Characteristics of Friend Function

- ✓ A friend function is not in the scope of the class n which it has been declared as friend.
- ✓ It cannot be called using the object of that class.
- ✓ It can be invoked like a normal function without any object.
- ✓ Unlike member functions, it cannot use the member names directly.
- ✓ It can be declared in public or private part without affecting its meaning.
- ✓ Usually, it has objects as arguments.

- To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:



```
#include<iostream.h>
#include<conio.h>
class maths
{
// private memeber
int x;
public :
void get()
{
cout << " enter x: \n";
cin>>x;
}
// friend function declaration
friend void square (maths m);
};_
// friend function definition
void square (maths m)
{
// access private member of class
cout << m.x * m.x;
}
```



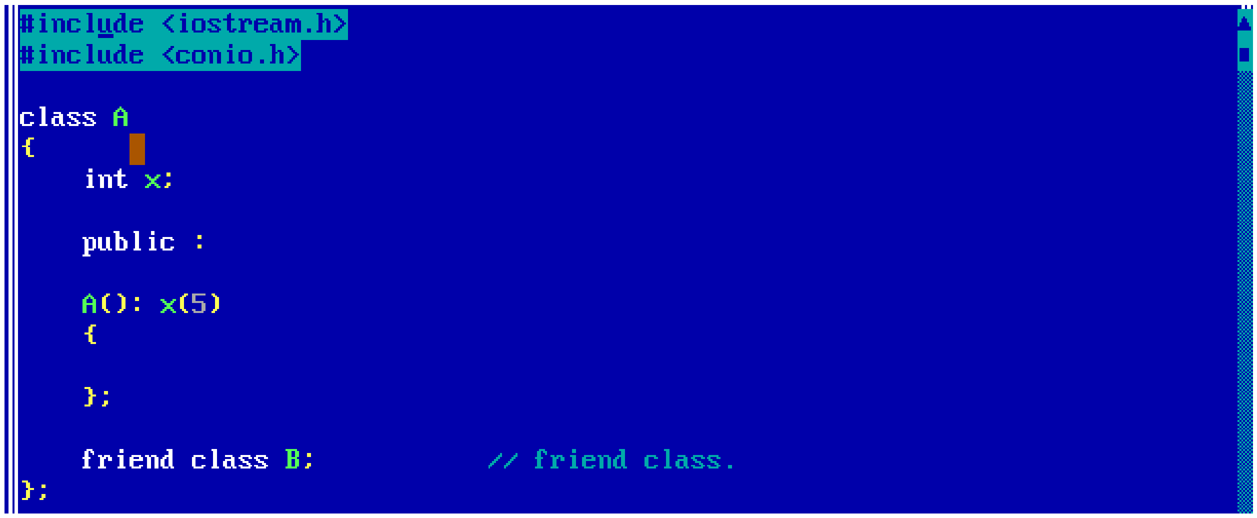
```
void main()
{
clrscr();
// normal function call
maths m1;
m1.get();
//  friend function call
square(m1);

getch();
}
```

```
 enter x:
6
36
```

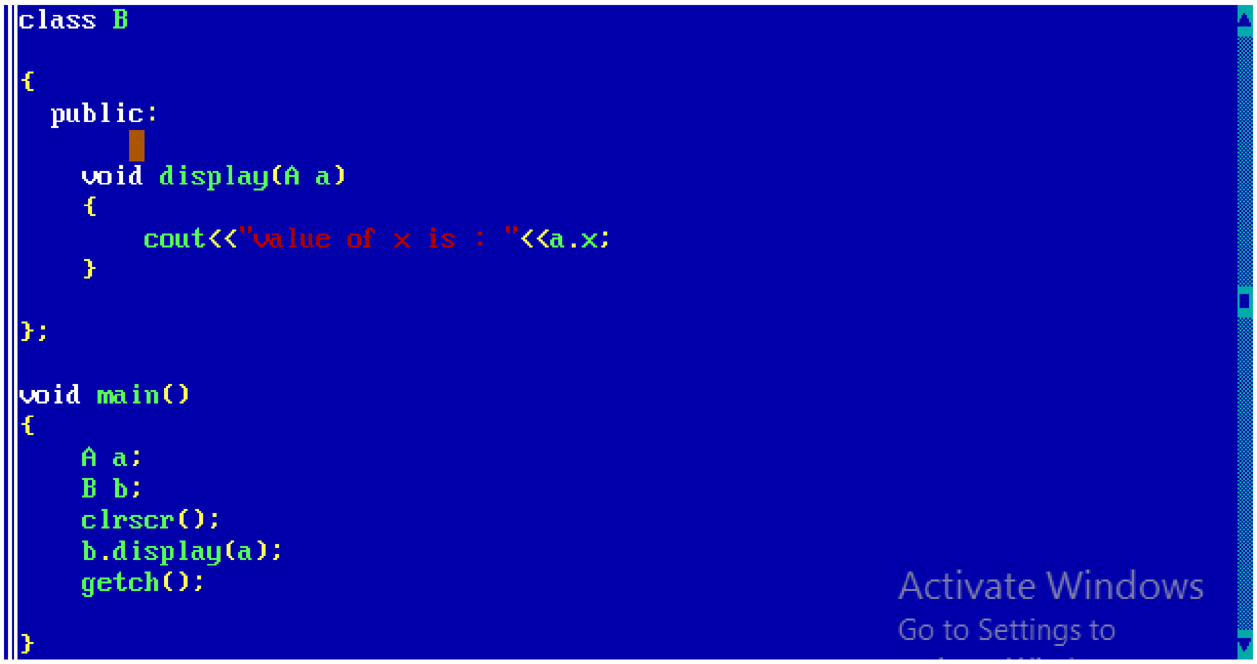
FRIEND CLASS:

As we know that a class cannot access the private members of other class. Similarly a class that doesn't inherit another class cannot access its protected members.

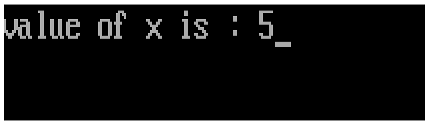
A **friend class** is a class that can access the private and protected members of a class in which it is declared as **friend**. This is needed when we want to allow a particular class to access the private and protected members of a class.



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

class A
{
    int x;

    public :

    A(): x(5)
    {

    };

    friend class B;           // friend class.
};
```

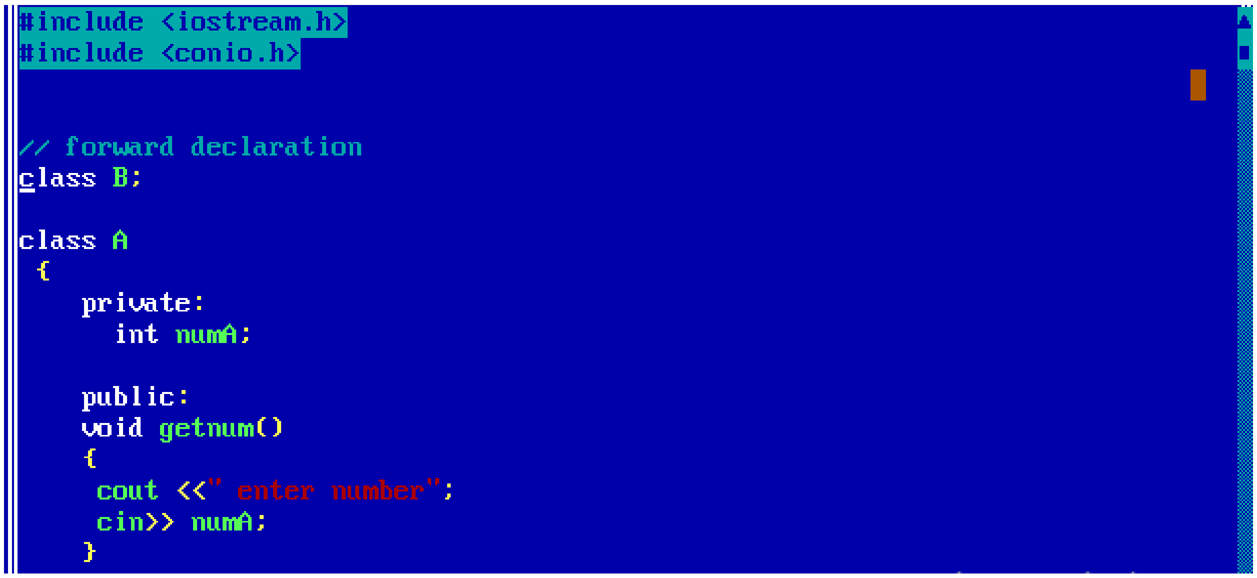


```
class B

{
  public:

    void display(A a)
    {
        cout<<"value of x is : "<<a.x;
    }

};

void main()
{
    A a;
    B b;
    clrscr();
    b.display(a);
    getch();

}
```

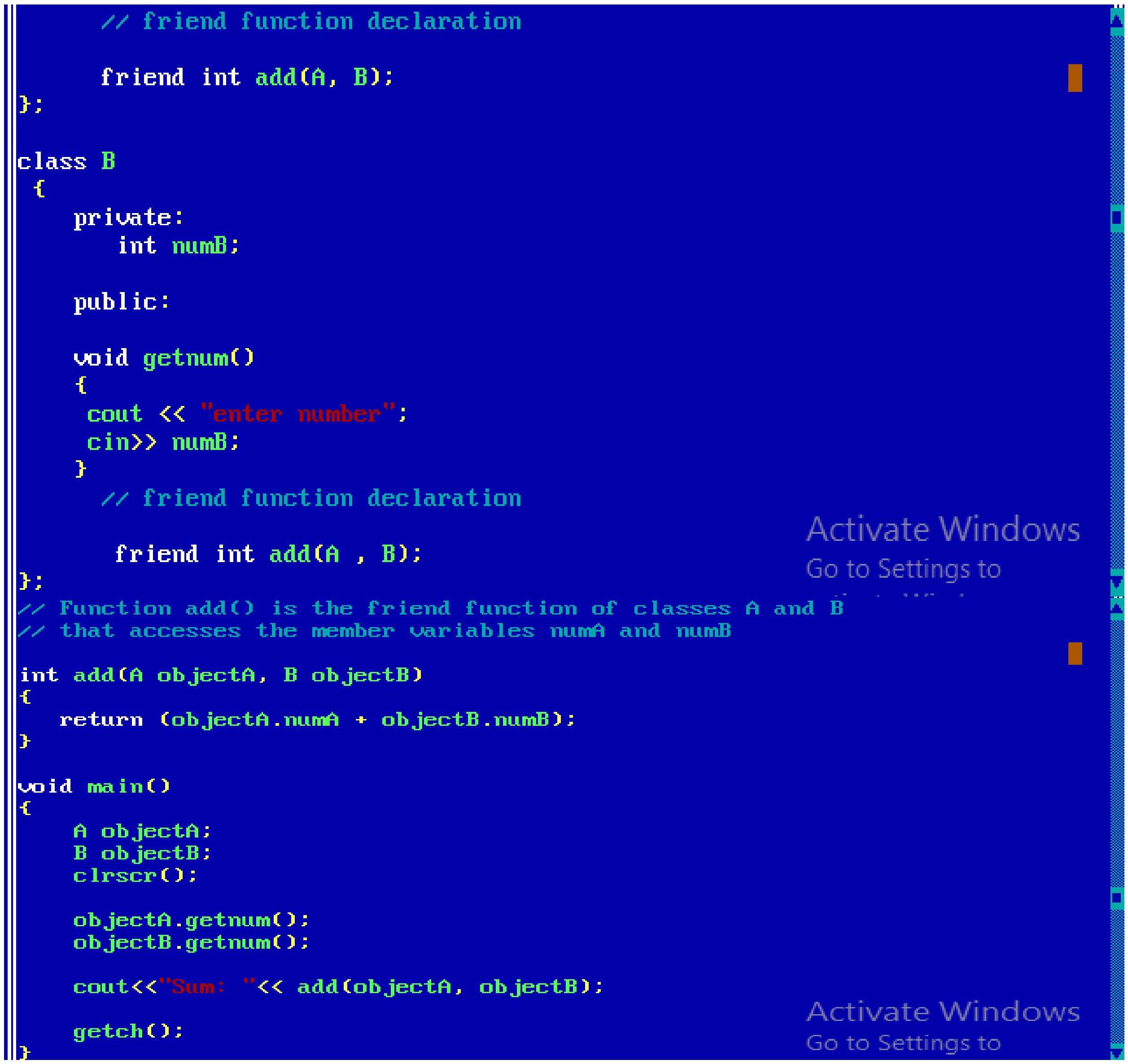


```
value of x is : 5_
```

FRIEND FUNCTION SHARING BOTH CLASS:



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


// forward declaration
class B;

class A
 {
    private:
      int numA;

    public:
    void getnum()
    {
     cout <<" enter number";
     cin>> numA;
    }
```



```
      // friend function declaration

      friend int add(A, B);
};

class B
 {
    private:
       int numB;

    public:

    void getnum()
    {
     cout << "enter number";
     cin>> numB;
    }
      // friend function declaration

       friend int add(A , B);
};
// Function add() is the friend function of classes A and B
// that accesses the member variables numA and numB

int add(A objectA, B objectB)
{
   return (objectA.numA + objectB.numB);
}

void main()
{
    A objectA;
    B objectB;
    clrscr();

    objectA.getnum();
    objectB.getnum();

    cout<<"Sum: "<< add(objectA, objectB);

    getch();
}
```

1.8 Inline Function

- The main objective of using function in program is to save memory space when function is likely to be called many times
- Whenever function is called it takes lot of extra time in executing a series of instruction for task such as, jumping to the function, returning to the function call, saving register, pushing argument in to stack ,etc
- When function is small it takes a lot of time for all these procedure to happen
- One solution of these is , Macros but, major drawback of macros is that they are actually not a function so, usual error checking does not occur during compilation
- In c++ solution for these is **INLINE FUNCTION** ,these are the functions designed to speed up program execution
- **An inline function is a function that is expanded in line when it is invoked, means the compiler replaces the function call with corresponding function code**

The syntex of inline function is as follows:

```
inline function_name()
{
body of the function
}
```

For example,

```
//function definition min()

inline void min (int x, int y)
{
cout<< (x < Y? x : y);
 }

Void main()
{
int num1, num2;
cout<<"\n enter two integers\n;
cin>>num1>>num2;
min (num1,num2);

//function code inserted here ------------------ ------------------

}
```

- An inline function definition must be defined before being invoked as shown in the above example. Here min ( ) being inline will not be called during execution, but its code would be inserted into main ( ) as shown and then it would be compiled.

- **Inline sends Request to compiler not command**

- If the size of the inline function is large then heavy memory penalty makes it not so useful and in that case normal function use is more useful.

The inline function does not work for the following situations:

- ✓ For functions returning values and having a loop or a switch or a goto statement.
- ✓ For functions that do not return value and having a return statement.
- ✓ For functions having static variable(s).
- ✓ If the inline functions are recursive (i.e. a function defined in terms of itself).

The benefits of inline functions are as follows:

- ✓ Better than a macro.
- ✓ Function call overheads are eliminated.
- ✓ Program becomes more readable.
- ✓ Program executes more efficiently.

1.9 Dynamic Object Creation & destruction (new and delete)

- Dynamic objects are used when the object to be created is not predictable enough
- .This is usually when we cannot determine at compile time
    - ✓ Object Identities
    - ✓ Object Quantities
    - ✓ Object Lifetimes
- Therefore when creating Dynamic Objects they cannot be given unique names, so we have to give them some other identities In this case we use pointers.

Creating Dynamic Objects Dynamic

- Objects use dynamic memory allocation In C++ a pointer can be directed to an area of dynamically allocated memory at runtime this can then be made to point to a newly created object.
- To do this we use the new operator in the following way

  Point2D *point1;

  point1 = **new** Point2D();

- First we create a pointer to the Point2D Class using the *, Then we use the **new** operator to construct a new instance to the class Using Dynamic Objects

- To call a user defined constructor we use the following syntax:

  Point2D *point1 = new Point2D(100.0,100.0);

- To send a message (i.e. use a method) we use the -> 'pointed to' delimiter as follows cout <<GetY();
- Apart from these distinctions dynamic Objects behave in the same as static

Destroying a Dynamic Object

- To create an Object we use a constructor, conversely to destroy an object we use a destructor.
- As there is a default constructor which the programmer does not define, there is also a default destructor which the programmer does not define.
- However it is also possible to define a default destructor this will either be called by the programmer or by the program when the object falls out of scope

- To destroy a dynamic Object the destructor must be called. This is done by using the **delete** operator as follows

  Point2D *point = new Point2D(100.0,12.0);

  // now do some thing with the object

  delete point; // now we destroy the object

Operator New and Delete

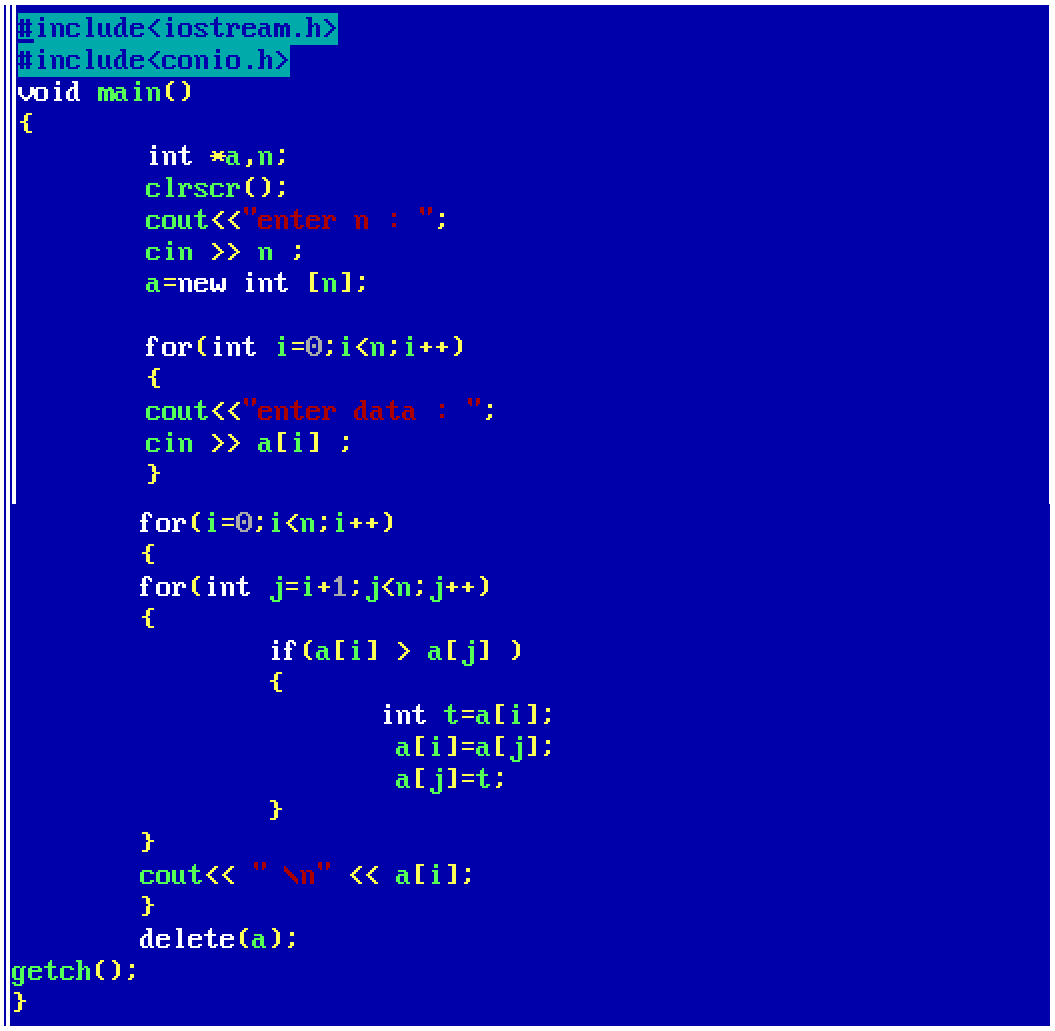
- **New:** it is used for allocating the memory at run time which generally allocated as pointer. It helps in dynamic initialization.
- **Delete:** it is used for de-allocating the memory allocated with new operator. it can only remove the variables which are allocated by new operator.

Example 1: (with class)

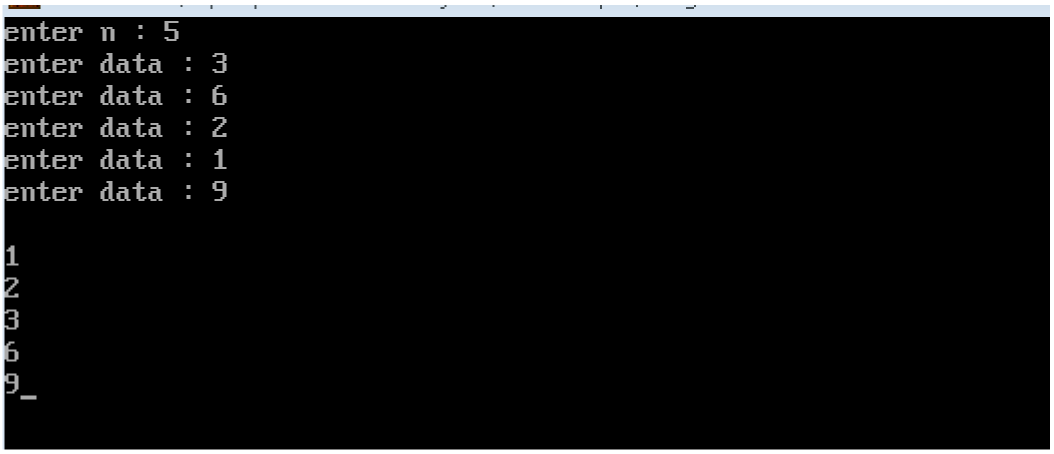
```
#include<iostream.h>
#include<conio.h>

class colour
{
int r,g,b;

public:

colour(int m,int n,int o)
{
r= m;
g= n;
b= o;
}

void print()
{
cout<< "RBG : "<< r << b << g;
}

};
```

```
void main()
{

colour *current;
clrscr();

current = new colour(1,0,0);
current-> print();
delete current;

current = new colour(1,1,1);
current-> print();
delete current;

getch();
}
```

Example 2: (without class) sorting of data in ascending order



```
#include<iostream.h>
#include<conio.h>
void main()
{
        int *a,n;
        clrscr();
        cout<<"enter n : ";
        cin >> n ;
        a=new int [n];

        for(int i=0;i<n;i++)
        {
        cout<<"enter data : ";
        cin >> a[i] ;
        }

        for(i=0;i<n;i++)
        {
        for(int j=i+1;j<n;j++)
        {
                if(a[i] > a[j] )
                {
                        int t=a[i];
                         a[i]=a[j];
                         a[j]=t;
                }
        }
        cout<< " \n" << a[i];
        }
        delete(a);
getch();
}
```



```
enter n : 5
enter data : 3
enter data : 6
enter data : 2
enter data : 1
enter data : 9

1
2
3
6
9_
```

C++ Data Types

Data types specify the type of data that a variable can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared as every data type requires a different amount of memory.

C++ supports a wide variety of data types, and the programmer can select the data type appropriate to the needs of the application.

Example:

```
#include <iostream>

using namespace std;

int main() {

        // Creating a variable to store integer
  int var = 10;

        cout << var;
  return 0;
}
```

Output

```
10
```

Explanation: In the above code, we needed to store the value **10** in our program, so we created a variable **var**. But before **var**, we have used the keyword '**int**'. This keyword is used to define that the variable **var** will store data of type **integer**.

Classification of Datatypes

In C++, different data types are classified into the following categories:

S. No.	Type	Description	Data Types
1	**Basic Data Types**	Built-in or primitive data types that are used to store simple values.	**int, float, double, char, bool, void**
2	**Derived Data Types**	Data types derived from basic types.	**array, pointer, reference, function**
3	**User Defined Data Types**	Custom data types created by the programmer according to their need.	**class, struct, union, typedef, using**

Let's see how to use some primitive data types in C++ program.

1. Character Data Type (char)

The **character data type** is used to store a single character. The keyword used to define a character is **char**. Its size is 1 byte, and it stores characters enclosed in single quotes (' '). It can generally store upto 256 characters according to their ASCII codes.

Example:
```
#include <iostream>
using namespace std;

int main() {

   // Character variable
   char c = 'A';
   cout << c;

   return 0;
}
```

Output
```
A
```

2. Integer Data Type (int)

Integer data type denotes that the given variable can store the integer numbers. The keyword used to define integers is **int.** Its size is **4-bytes** (for 64-bit) systems and can store numbers for binary, octal, decimal and hexadecimal base systems in the range from **-2,147,483,648 to 2,147,483,647.**

Example:
```
#include <iostream>
using namespace std;

int main() {

   // Creating an integer variable
   int x = 25;
   cout <<  “value of x is :” << x << endl;

   // Using hexadecimal base value
   x = 0x15;
   cout << x;

   return 0;
}
```

Output
```
25
21
```

To know more about different base values in C++, refer to the article - Literals in C++

3. Boolean Data Type (bool)

The **boolean data type** is used to store logical values: **true(1)** or **false(0)**. The keyword used to define a boolean variable is **bool**. Its size is 1 byte.

Example:
```
#include <iostream>
using namespace std;
```

```
int main() {

   // Creating a boolean variable
   bool isTrue = true;
   cout << isTrue;
   return 0;
}
```

Output
```
1
```

4. Floating Point Data Type (float)

Floating-point data type is used to store numbers with decimal points. The keyword used to define floating-point numbers is **float**. Its size is 4 bytes (on 64-bit systems) and can store values in the range from **1.2E-38 to 3.4e+38.**

Example:
```
#include <iostream>
using namespace std;

int main() {

   // Floating point variable with a decimal value
   float f = 36.5;
   cout << f;

   return 0;
}
```

Output
```
36.5
```

5. Double Data Type (double)

The **double data type** is used to store decimal numbers with higher precision. The keyword used to define double-precision floating-point numbers is **double**. Its size is 8 bytes (on 64-bit systems) and can store the values in the range from **1.7e-308 to 1.7e+308**

Example:
```
#include <iostream>
using namespace std;

int main() {

   // double precision floating point variable
   double pi = 3.1415926535;
   cout << pi;

   return 0;
}
```

Output
```
3.14159
```

6. Void Data Type (void)

The **void data type** represents the absence of value. We cannot create a variable of void type. It is used for pointer and functions that do not return any value using the keyword **void**.

Type Safety in C++

C++ is a **strongly typed language**. It means that all variables' data type should be specified at the declaration, and it does not change throughout the program. Moreover, we can only assign the values that are of the same type as that of the variable.

Example: If we try to assign **floating point** value to a boolean variable, it may result in data corruption, runtime errors, or undefined behaviour.

```
#include <iostream>
using namespace std;

int main() {

   // Assigning float value to isTrue
   bool a = 10.248f;
   cout << isTrue;
   return 0;
}
```

Output
```
1
```

As we see, the floating-point value is not stored in the bool variable **a.** It just stores 1. This type checking is not only done for fundamental types, but for all data types to ensure valid operations and no data corruptions.

Data Type Conversion

Type conversion refers to the process of changing one data type into another compatible one without losing its original meaning. It's an important concept for handling different data types in C++.

```
#include <iostream>
using namespace std;

int main() {
   int n = 3;
   char  c = 'C';

   // Convert char data type into integer
   cout << (int)c << endl;
   int sum = n + c;
   cout << sum;
   return 0;
}
```

Output
```
67
70
```

Size of Data Types in C++

Earlier, we mentioned that the size of the data types is according to the 64-bit systems. Does it mean that the size of C++ data types is different for different computers?
Actually, it is partially true. The size of C++ data types can vary across different systems, depending on the architecture of the computer (e.g., 32-bit vs. 64-bit systems) and the compiler being used. But if the architecture of the computer is same, then the size across different computers remains same.
We can find the size of the data type using **sizeof** operator. According to this type, the range of values that a variable of given data types can store are decided.

Example:

```
#include <iostream>
using namespace std;

int main() {

    // Printing the size of each data type
    cout << "Size of int: " << sizeof(int) << " bytes" << endl;
    cout << "Size of char: " << sizeof(char) << " byte" << endl;
    cout << "Size of float: " << sizeof(float) << " bytes" << endl;
    cout << "Size of double: " << sizeof(double) << " bytes";

    return 0;
}
```

Output

```
Size of int: 4 bytes
Size of char: 1 byte
Size of float: 4 bytes
Size of double: 8 bytes
```

Derived Data Types in C++

The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. They are generally the data types that are created from the primitive data types and provide some additional functionality.
In C++, there are four different derived data types:
Table of Content

Functions

A **function** is a block of code or program segment that is defined to perform a specific well-defined task. It is generally defined to save the user from writing the same lines of code again and again for the same input. All the lines of code are put together inside a single function and this can be called anywhere required.

Let's take a look at example that demonstrates the use of function in C++:

```
#include <iostream>
using namespace std;

// max here is a function which is a derived type
int max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}


// main function is also a derived type
int main() {
    int a = 10, b = 20;

    // Calling above function to
    // find max of 'a' and 'b'
    int m = max(a, b);

    cout << "m is " << m;
    return 0;
}
```

Output

```
m is 20
```

Explanation: This above program demonstrates the use of function derived types It defines a function called **max** this function returns the maximum of two integers provided as input. In the main function, **max** function is called to find the maximum of variables **a** and **b** and store it in m and finally print **m**(max number).

Arrays

An **array** is a collection of items stored at continuous memory locations. The idea of array is to represent many variables using a single name. The below example demonstrates the use of array in C++.

```
#include <iostream>
using namespace std;

int main() {

    // Array Derived Type
    int arr[5] = {1, 2, 3, 4, 5};
    arr[0] = 5;
    arr[2] = -10;
    arr[3] = arr[0];

    // Printing the data
    for (int i = 0; i < 5; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output

```
5 2 -10 5 5
```

Explanation: This above program shows the use of array-derived type. It creates an integer array **arr** and assigns values using indices. Then it prints all the array elements.

Pointers

Pointers are symbolic representation of addresses. They can be said as the variables that can store the address of another variable as its value. The below example demonstrates the use of pointer in C++.

```
#include <iostream>
using namespace std;

int main() {
   int var = 20;

   // Pointers Derived Type
   // declare pointer variable

   int* ptr;

   // note that data type of ptr
   // and var must be same

   ptr = &var;

   // assign the address of a variable
   // to a pointer

   cout << "Value at ptr = " << ptr << endl;
   cout << "Value at var = " << var << endl;
   cout << "Value at *ptr = " << *ptr;

   return 0;
}
```

Output

```
Value at ptr = 0x7fffa5bef2cc
Value at var = 20
Value at *ptr = 20
```

Explanation: This above program demonstrates the use of pointers as a derived type. It declares pointer variable **ptr** and assigning the address of a variable **var** to it. It then prints the values of the pointer, the variable, and the dereferenced pointer, showcasing the basics of pointer usage in C++.

References

When a variable is declared as **reference**, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting ‘&’ in the declaration.
The below example demonstrates the use of reference in C++.

```
#include <iostream>
using namespace std;
int main() {
   int x = 10;

   // Reference Derived Type
   // ref is a reference to x.
   int& ref = x;

   // Value of x is now changed to 20
   ref = 20;
   cout << "x = " << x << endl;

   // Value of x is now changed to 30
   x = 30;

   cout << "ref = " << ref << endl;

   return 0;
}
```

Output

```
x = 20
ref = 30
```

Explanation: The above program demonstrates the use of reference-derived type. A reference **ref** to an integer variable **x**. is created. If the value of **ref** is changed the value **x**, is also modified and vice versa.

User Defined Data Types in C++

User defined data types are those data types that are defined by the user himself. In C++, these data types allow programmers to extend the basic data types provided and create new types that are more suited to their specific needs. C++ supports 5 user-defined data types:

- Class
- Structure
- Union
- Enumeration
- Typedef

1. Class

A **Class** is the building block of C++'s Object-Oriented programming paradigm. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Example

```
#include <bits/stdc++.h>
using namespace std;

class abc {

   // Access specifier
public:

   // Data Member
   string name;

   // Member Function
   void printname() {
       cout << name;
   }
};

int main() {

   // Declare an object of class geeks
   abc a;

   // Accessing data member
   a.name = "khyati";

   // Accessing member function
   a.printname();

   return 0;
}
```

Output
```
khyati
```

Explanation: The above program defines a class named **abc** with a **name** attribute and a function **printname**() to print the name. In the main function, it creates an object named **a**, sets the geekname as "**khyati**", and calls the **printname**() function to display it.

2. Structure

A **Structure** is a user-defined data type like class. A structure creates a data type that can be used to group items of possibly different types into a single type.

Example
```
#include <iostream>
using namespace std;

// Declaring structure
struct A {
    int i;
    char c;
};

int main() {

    // Create an instance of structure
    A a;

    // Initialize structure members
    a.i = 65;
    a.c = 'A';
    cout << a.c << ": " << a.i;

    return 0;
}
```

Output

```
A: 65
```

Explanation: The above demonstrates program demonstrates the use of structures by defining a structure named **A** having **i** and **c** members. It then creates an instance if structure in the main function, sets the members' values, and prints them.
Structures in C++ are different from structures in C and resembles classes.

3. Union

Like structures , **union** is also user-defined data type used to group data of different type into a single type. But in union, all members share the same memory location.

Example

```
#include <iostream>
using namespace std;

// Declaration of union is same as the structures
union A {
    int i;
    char c;
};

int main() {
    // A union variable t
    A a;

    // Assigning value to c, i will also
    // assigned the same
    a.c = 'A';

    cout << "a.i: " << a.i << endl;
    cout << "a.c: " << a.c;

    return 0;
}
```

Output

```
a.i: 65
a.c: A
```

Explanation: The above program demonstrates the use of unions. Union named A with members **i** and **c** is defined that shares the same memory space. It is shown that when we only assign **c** some value, the **i** also stores the same value.

4. Enumeration

Enumeration (or enum) is a user-defined data type in C++ mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Example
```
#include <iostream>
using namespace std;

// Declaring enum
enum Week { Mon, Tue, Wed, Thur, Fri, Sat, Sun };

int main() {

   // Creating enum variable
   enum Week day;

   // Assigning value to the variabe
   day = Wed;

   cout << day;
   return 0;
}
```

Output
```
2
```

5. Typedef and Using

C++ allows you to define explicitly new data type names by using the keywords **typedef** or **using**. They do not create a new data class, rather, defines a name for an existing type. This can increase the portability (the ability of a program to be used across different types of machines; i.e., mini, mainframe, micro, etc; without many changes to the code) of a program as only the typedef statements would have to be changed.

Example
```
#include <iostream>
using namespace std;

// Using typedef to define a new name for existing type
typedef float f;

// Using 'using' to define a new name for existing type
using integer = int;

int main() {
   // Declaring variables using new type names
   f x = 3.14;
   integer y = 42;

   cout << "Float Value: " << x << endl;
   cout << "Integer Value: " << y;

   return 0;
}
```

Output
```
Float Value: 3.14
Integer Value: 42
```

String:

C++ Strings

A string is a collection of characters. There are two types of strings commonly used in C++ :

- Strings that are objects of string class (The Standard C++ Library String Class)
- C-strings (C-style Strings)

C-strings

In C programming, the collection of characters is stored in the form of arrays. This is also supported in C++ programming. Hence, it's called C-strings.
C-strings are arrays of type char terminated with a null character, that is, \0 (ASCII value of null character is **0**).

How to define a C-string?

```
char str[] = "C++";
```

In the above code, str is a string and it holds **4** characters.
Although "C++" has three characters, the null character \0 is added to the end of the string automatically.

Alternative ways of defining a string

```
char str[4] = "C++";

char str[] = {'C','+','+','\0'};

char str[4] = {'C','+','+','\0'};
```

Like arrays, it is not necessary to use all the space allocated for the string. For example:

```
char str[100] = "C++";
```

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

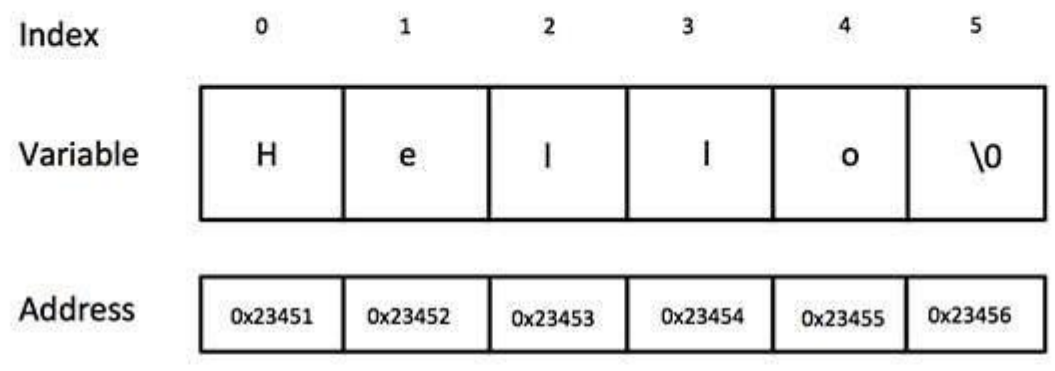
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

Example

```
#include <iostream>
using namespace std;
int main () {
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

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

```
Greeting message: Hello
```

Character Arrays

We can declare strings using the C-type arrays in the format of characters. This is done using the following syntax −

Syntax

```
char variable_name[len_value] ;
```

Here, len_value is the length of the character array.

Example of Creating String using Character Array

In the following examples, we are declaring a character array, and assigning values to it.

```
#include <iostream>
using namespace std;

int main() {
   char s[5]={'h','e','l','l','o'};
   cout<<s<<endl;
   return 0;
}
```

C Style String Functions

C++ supports a wide range of functions that manipulate null-terminated strings. These functions are defined in <string.h> header file.

Sr.No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strrev(s1);** Returns a s1 string in reverse order

Example
Following example makes use of few of the above-mentioned functions −

```
#include <iostream>
#include <string>
using namespace std;
int main () {

   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

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

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

C++ strcmp()

The strcmp() function in C++ compares two null-terminating strings (C-strings). The comparison is done lexicographically. It is defined in the cstring header file.

Example

```
#include <cstring>
#include <iostream>
using namespace std;

int main() {
  char str1[] = "Megadeth";
  char str2[] = "Metallica";

  // compare str1 and str2 lexicographically
  int result = strcmp(str1, str2);
  cout << result;
  return 0;
}

// Output: -1
```

strcmp() Return Value

The strcmp() function returns:

- a **positive value** if the first differing character in lhs is greater than the corresponding character in rhs.
- a **negative value** if the first differing character in lhs is less than the corresponding character in rhs.
- **0** if lhs and rhs are equal.

The String Class in C++

We can declare a String variable using the 'string' keyword. This is included in the <string.h> header file. The syntax of declaring a string is explained as follows −

Syntax

```
string variable_name = [value];
```

Here, [value] is an optional and can be used to assign value during the declaration.

Example
In the following examples, we are declaring a string variable, assigning a value to it.

```
#include <iostream>
using namespace std;

int main() {
   string s="a merry tale";
   cout<<s;
   return 0;
}
```

Output

```
a merry tale
```

Example of String Class

```
#include <iostream>
#include <string>

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
char str1[6]={};
   string str3;
   int  len ;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;
   return 0;
}
```

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

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

Traversing a String

Using looping statements

We can traverse a string using for loops, while loops and do while loops using a pointer to the first and the last index in the string.

```
#include <iostream>
using namespace std;

int main() {
  string s="Hey, I am at TP.";

  for(int i=0;i<s.length();i++){
    cout<<s[i]<<" ";
  }
  cout<<endl;
  }

  return 0;
}
```

Output

```
H e y ,   I   a m   a t   T P .
```

Example 1: C++ String to Read a Word
C++ program to display a string entered by user.

```
#include <iostream>
using namespace std;

int main()
{
   char str[100];

   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;

   cout << "\nEnter another string: ";
   cin >> str;
   cout << "You entered: " << str << endl;

   return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Notice that, in the second example, only "Programming" is displayed instead of "Programming is fun".

This is because the extraction operator >> works as scanf() in C and considers a space " " as a terminating character.

Example 2: C++ String to read a line of text
C++ program to read and display an entire line entered by the user.

```
#include <iostream>
using namespace std;
int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin.get(str, 100);

   cout << "You entered: " << str << endl;
   return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

To read the text containing blank space, cin.get function can be used. This function takes two arguments.

The first argument is the name of the string (address of the first element of the string), and the second argument is the maximum size of the array.
In the above program, str is the name of the string and 100 is the maximum size of the array.

<u>pointer to a character array</u>
(often used to represent C-style strings) is a char* type variable that stores the memory address of the first character in the array.

Declaration and Initialization:

Directly pointing to the first element.

```
   char myCharArray[] = "Hello";
   char* ptr = myCharArray; // ptr now points to 'H'
```

The name of a character array, when used without an index, decays into a pointer to its first element. Using the address-of operator.

```
   char myCharArray[] = "World";
   char* ptr = &myCharArray[0]; // ptr now points to 'W'
```

Pointing to a string literal.

```
    char* ptr = "C++"; // ptr points to the first character of the string literal "C++"
```
Note that string literals are typically stored in read-only memory, so attempting to modify the characters through ptr in this case would lead to undefined behavior.

Accessing Characters:

Once a char* points to a character array, you can access individual characters using: Dereference operator (*).

```
    char firstChar = *ptr; // Gets the character pointed to by ptr
```
Pointer arithmetic and dereference.

```
    char secondChar = *(ptr + 1); // Gets the second character in the array
```

Array-like indexing.
C++
```
    char thirdChar = ptr[2]; // Gets the third character in the array (equivalent to *(ptr + 2))
```

Traversing the Array:
You can iterate through the characters of a C-style string using pointer arithmetic:

```
char myString[] = "Example";
char* currentPtr = myString;

while (*currentPtr != '\0') { // Iterate until the null terminator is found
    // Process *currentPtr
    currentPtr++; // Move to the next character
}
```

Type Conversion in C++

The conversion of one data type into another in the C++ programming language.

Type conversion is the process that converts the predefined data type of one variable into an appropriate data type.

The main idea behind type conversion is to convert two different data type variables into a single data type to solve mathematical and logical expressions easily without any data loss.

For example, we are adding two numbers, where one variable is of int type and another of float type; we need to convert or typecast the int variable into a float to make them both float data types to add them.

Type conversion can be done in two ways in C++,

1) **Implicit type conversion**
2) **Explicit type conversion**.

Those conversions are done by the compiler itself, called the **implicit type or automatic type conversion.**

The conversion, which is done by the user or requires user interferences called the **explicit or user define type conversion**

Implicit Type Conversion

The implicit type conversion is the type of conversion done **automatically by the compiler without any human effort.**

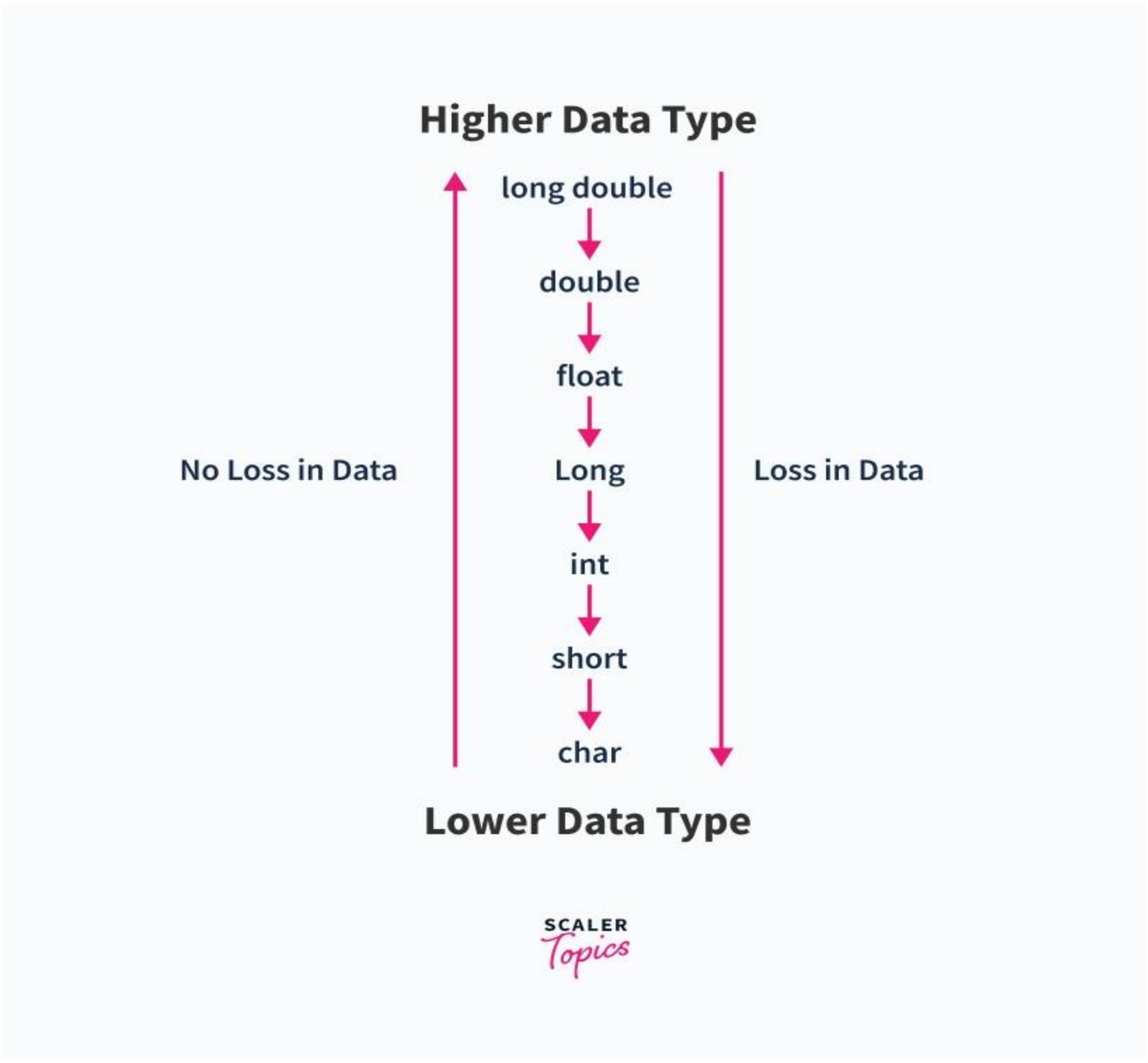
It means an implicit conversion automatically converts one data type into another type based on some predefined rules of the C++ compiler. Hence, it is also known as the **automatic type conversion**.

For example:

```
int x = 20;
short int y = 5;
int z = x + y;
```

In the above example, there are two different data type variables, x, and y, where x is an int type, and the y is of short int data type. And the resultant variable z is also an integer type that stores x and y variables. But the C++ compiler automatically converts the lower rank data type (short int) value into higher type (int) before resulting the sum of two numbers. Thus, it avoids the data loss, overflow, or sign loss in implicit type conversion of C++.

Order of the typecast in implicit conversion



The following is the correct order of data types from lower rank to higher rank:

1. **bool** -> **char** -> **short int** -> **int** -> unsigned **int** -> **long int** -> unsigned **long int** -> **long long int** -> **float** -> **double** -> **long double**

Program to convert int to float type using implicit type conversion

Let's create a program to convert smaller rank data types into higher types using implicit type conversion.

Program1.cpp

```
#include <iostream>
using namespace std;
int main ()
{
// assign the integer value
int num1 = 25;
// declare a float variable
float num2;
// convert int value into float variable using implicit conversion
num2 = num1;
cout << " The value of num1 is: " << num1 << endl;
cout << " The value of num2 is: " << num2 << endl;
return 0;
}
```

Output

```
The value of num1 is: 25
The value of num2 is: 25
```

Program to convert double to int data type using implicit type conversion

Let's create a program to convert the higher data type into lower type using implicit type conversion.

Program2.cpp

```
#include <iostream>
using namespace std;
int main()
{
int num; // declare int type variable
double num2 = 15.25; // declare and assign the double variable

// use implicit type conversion to assign a double value to int variable
num = num2;
cout << " The value of the int variable is: " << num << endl;
cout << " The value of the double variable is: " << num2 << endl;
return 0;
}
```

Output

```
The value of the int variable is: 15
 The value of the double variable is: 15.25
```

In the above program, we have declared num as an integer type and num2 as the double data type variable and then assigned num2 as 15.25. After this, we assign num2 value to num variable using the assignment operator. So, a C++ compiler automatically converts the double data value to the integer type before assigning it to the num variable and print the truncate value as 15.

Explicit type conversion

Conversions that require **user intervention** to change the data type of one variable to another, is called the **explicit type conversion**.

In other words, an explicit conversion allows the programmer to manually changes or typecasts the data type from one variable to another type. Hence, it is also known as **typecasting**.

Generally, we force the explicit type conversion to convert data from one type to another because it does not follow the implicit conversion rule.

The explicit type conversion is divided into two ways:

1. Explicit conversion using the **cast operator**
2. Explicit conversion using the **assignment operator**

Program to convert float value into int type using the cast operator

Cast operator: In C++ language, a cast operator is a unary operator who forcefully converts one type into another type.

Let's consider an example to convert the float data type into int type using the cast operator of the explicit conversion in C++ language.

Program3.cpp

```
#include <iostream>
using namespace std;
int main ()
{
float f2 = 6.7;
// use cast operator to convert data from one type to another
int x = static_cast <int> (f2);
cout << " The value of x is: " << x;
return 0;
}
```

Output

```
The value of x is: 6
```

Program to convert one data type into another using the assignment operator

Let's consider an example to convert the data type of one variable into another using the assignment operator in the C++ program.

Program4.cpp

```
#include <iostream>
using namespace std;
int main ()
{
// declare a float variable
float num2;
// initialize an int variable
int num1 = 25;

// convert data type from int to float
num2 = (float) num1;
cout << " The value of int num1 is: " << num1 << endl;
cout << " The value of float num2 is: " << num2 << endl;
return 0;
}
```

Output

```
The value of int num1 is: 25
The value of float num2 is: 25.0
```

2.1 Introduction to OOP properties

- ➔ The main purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.
- ➔ The core of the pure object-oriented programming is to create an object in code that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects.
- ➔ For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, break, and so on.
- ➔ There are a few principle concepts that form the foundation of object-oriented programming:

  - ➢ **Object**
  - ➢ **Class**
  - ➢ **Abstraction**
  - ➢ **Encapsulation**
  - ➢ **Inheritance**
  - ➢ **Polymorphism**

NOTE: refer all the properties describe in UNIT-1

2.2 Abstraction

- ➔ Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
- ➔ Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.
- ➔ Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen
- ➔ Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.
- ➔ Now, if we talk in terms of C++ Programming, C++ classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

- ➔ For example, your program can make a call to the sort() function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

- ➔ In C++, we use classes to define our own abstract data types (ADT). You can use the coutobject of class ostream to stream data to standard output like this:

```
int main()

{

cout << "Hello C++" <<endl;

return0;

}
```

- ➔ Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout are free to change.

Access Labels Enforce Abstraction:

- ➔ In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:
- ➔ Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- ➔ Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.
- ➔ There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:

- ➔ Data abstraction provides two important advantages:

  - Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
  - The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

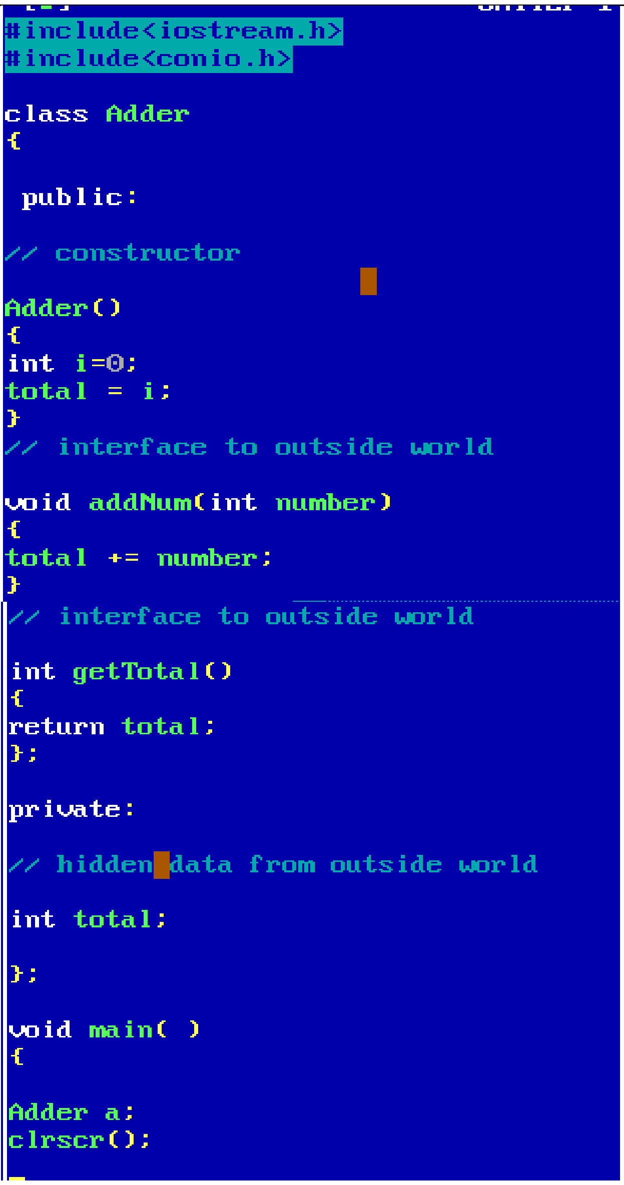
- ➔ By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have.

➔ If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example:

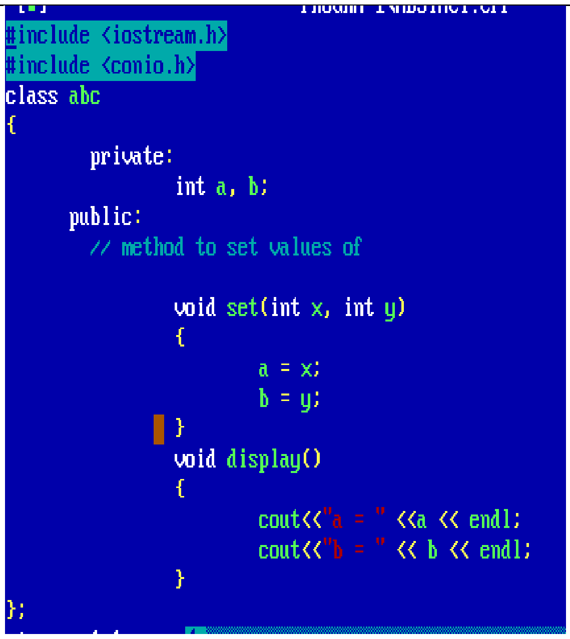
Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

Example 1 :

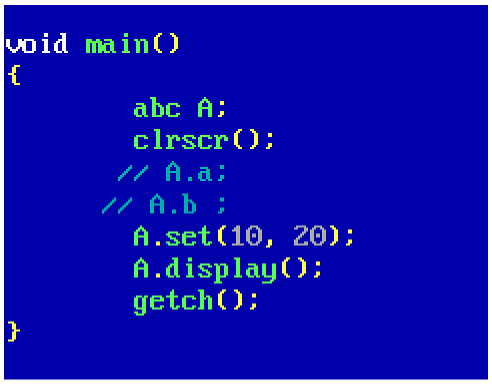


```
#include<iostream.h>
#include<conio.h>

class Adder
{

 public:

// constructor

Adder()
{
int i=0;
total = i;
}
// interface to outside world

void addNum(int number)
{
total += number;
}
// interface to outside world

int getTotal()
{
return total;
};

private:

// hidden data from outside world

int total;

};

void main( )
{

Adder a;
clrscr();
```

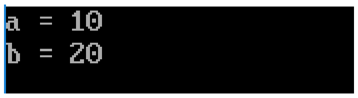
Example 2 :



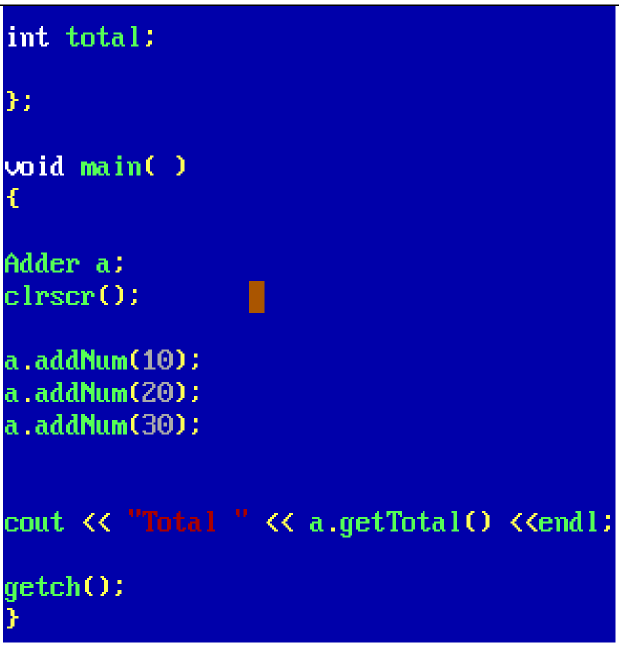
```
#include <iostream.h>
#include <conio.h>
class abc
{
        private:
                int a, b;
      public:
        // method to set values of

                void set(int x, int y)
                {
                        a = x;
                        b = y;
                }
                void display()
                {
                        cout<<"a = " <<a << endl;
                        cout<<"b = " << b << endl;
                }
};
```



```
void main()
{
        abc A;
        clrscr();
       // A.a;
      // A.b ;
        A.set(10, 20);
        A.display();
        getch();
}
```



```
a = 10
b = 20
```



```
int total;

};

void main( )
{

Adder a;
clrscr();

a.addNum(10);
a.addNum(20);
a.addNum(30);


cout << "Total " << a.getTotal() <<endl;

getch();
}
```

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

result: Total 60

Above class adds numbers together, and returns the sum. The public members addNumand getTotal are the interfaces to the outside world and a user needs to know them to use the class. The private member total is something that the user doesn't need to know about, but is needed for the class to operate properly.

Access Modifiers/Specifier in C++

- ➔ Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding. Consider a real-life example:
- ➔ The Indian secret informatic system having 10 senior members have some top secret regarding national security. So we can think that 10 people as class data members or member functions who can directly access secret information from each other but anyone can't access this information other than these 10 members i.e. outside people can't access information directly without having any privileges. This is what data hiding is.
- ➔ Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Let us now look at each one these access modifiers in details:

I. **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
// class definition

class Circle
{
   public:
      double radius;

      double  compute_area()
      {
         return 3.14*radius*radius;
      }

};

// main function

void main()
{
   Circle obj;

   // accessing public datamember outside class
   obj.radius = 5.5;

   cout << "Radius is: " << obj.radius << "\n";
   cout << "Area is: " << obj.compute_area();
   getch();
}
```

Output:

```
Radius is: 5.5

Area is: 94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

II. **Private**: The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

```
class Circle
{
  // private data member

  private:
    double radius;

  // public member function

  public:
    double  compute_area()
    {   // member function can access private
      // data member radius
      return 3.14*radius*radius;
    }

};

// main function

void main()
{
  // creating object of the class
  Circle obj;

  // trying to access private data member
  // directly outside the class
  obj.radius = 1.5;

  cout << "Area is:" << obj.compute_area();
  getch();
}
```

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.

Output:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
     double radius;
          ^
31:9: error: within this context
   obj.radius = 1.5;
```

III. **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any sub class(derived class) of that class.

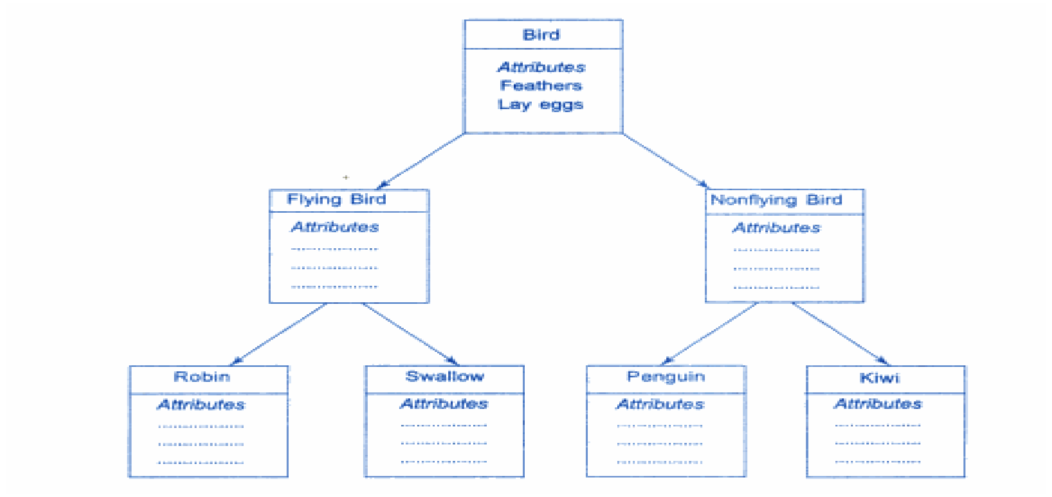
```
// base class

class Parent
{
  // protected data members
  protected:
  int id_protected;

};

// sub class or derived class

class Child : public Parent
{

  public:
  void setId(int id)
  {

    // Child class is able to access the inherited
    // protected data members of base class

    id_protected = id;

  }

  void displayId()
  {
    cout << "id_protected is: " << id_protected << endl;
  }
};

// main function

void main() {

  Child obj1;
    // member function of the derived class can
  // access the protected data members of the base class

  obj1.setId(81);
  obj1.displayId();
 getch();
}
```

Output:

```
id_protected is: 81
```

2.3 Inheritance

- ➔ One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- ➔ When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.
- ➔ Inheritance is the process by which the objects of one class acquire the properties of object of another class.
- ➔ It supports the concept of hierarchical – classification.
- ➔ In Oop, the concept of inheritance provides the idea of Reusability. It means that we can add additional features to an existing class without modifying it.
- ➔ It is possible by deriving a new class from existing class and new class will have combine features of both the class



Base& Derived Classes:

- ➔ A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

Class derived-class: access-specifier base-class

- ➔ Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

- ➔ **Base class / super class / parent class:** The class whose properties are inherited by sub class is called Base Class or Super class or Parent class.

- ➔ **Derive class / sub class / child class** :The class that inherits properties from another class is called Sub class or Derived Class or Child class

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

Rules for inheritance:

1. Private variables/methods will not be accessed in child class.
2. Child inherits parent = protected;[parent protected &public]=protected
3. Child inherits parent = private;[parent protected &public]=private
4. Child inherits parent = public;[parent protected & public]=[protected &public]

We can summarize the different access types according to who can access them in the following way:

Access	Public	protected	private
Same class	Yes	Yes	yes
Derived classes	Yes	Yes	no
Outside classes	Yes	No	no

Variable or function scope in base class	**Base class**	**Base class object**	**Child class**	**Child class object**
public	Accessasible	Accessasible	Accessasible	Accessasible
Private	Accessasible	Not Accessasible	Not Accessasible	Not Accessasible
Protected (class to class)	Accessasible	Not Accessasible	Accessasible	Not Accessasible

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Inheritance according to access specifier:

➔ When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

➔ We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

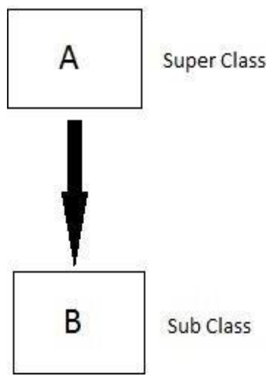
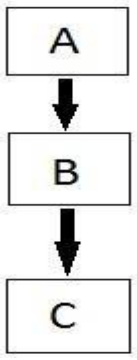
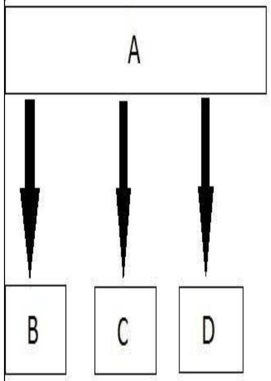
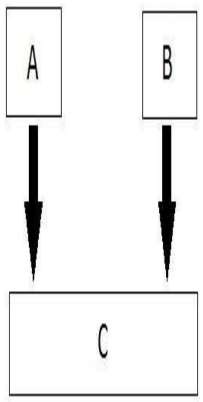
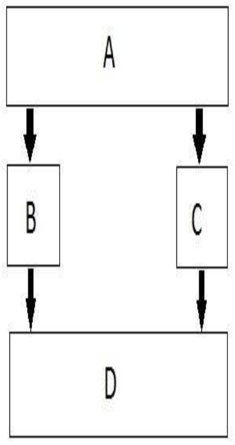
- **Public Inheritance**: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

- **Protected Inheritance**: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

- **Private Inheritance**: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Access specifier in base class	**Access specifier when inherited in public scope**	**Access specifier when inherited in private scope**	**Access specifier when inherited in protected scope**
public	Public	Private	Protected
Private	Not accessible	Not accessible	Not accessible
Protected	protected	Private	protected

2.3.1 Types of Inheritance

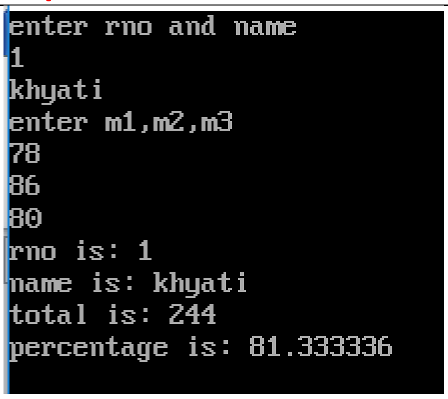
<u>**Singal**</u>	<u>**Multilevel**</u>	<u>**Hierarchical**</u>	<u>**Multiple**</u>	<u>**Hybrid**</u>
				
A derive class with only one class is known as single inheritance	A mechanism of deriving one class from another derive class is called multilevel inheritance	One class inherited by more than one classes is known as hierarchical inheritance	A derive class with more than one base class is known as multiple inheritance	Combination of more than one type of inheritance is known as hybrid inheritance

ClassA
{
//class body
};
 	
classB:publicA
{
//class body
};
	Class A
	{
	//class body
	};
	classB:publicA
	{
	//class body
	};
	
	ClassC:public B
		ClassA
		{
		//class body
		};
		classB:publicA
		{
		//class body
		};
		classC:publicA
		{
		//class body };
		classD:publicA
			ClassA
			{
			//class body
			};
			
			class B
			{
			//class body
			};
			
			Class C:public A,
				ClassA
				{
				//class body
				};
				classB: virtual publicA
				{
				//class body
				};

	{
	//class body
	};
		{
		//class body
		};
			public B
			{
			//class body
			};
			//derived from 2 base classes A and B
				classC:virtual publicA
				{
				//class body
				};
				class D:publicB, publicC
				{
				//class body
				};

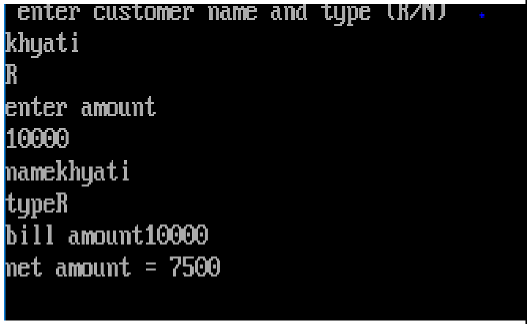
Examples :

Single Inheritance	**Output**

class stud
{
int rno;
char nm[15];
public:
void get()
{
cout << "enter rno and name";
cin >> rno >> nm;
}
 	
void display()
	
	enter rno and name
	1
	khyati
	enter m1,m2,m3
	78
	86
	80
	rno is: 1
	name is: khyati
	total is: 244
	percentage is: 81.333336

```
        {
            cout << "rno is: " << rno << endl;
            cout << "name is: " << nm << endl;
        }
};
//exam class is derived from base class stud.
class exam : public stud
{
        int m1,m2,m3;
   public:


   void get()
   {
   stud :: get();
   cout<< "enter m1,m2,m3"<<endl;
   cin >> m1 >>m2 >>m3;
   }
       void display()
        {
        stud :: display();


        int tot = m1+m2+m3;
        float per=tot/3.0;



        cout << "total is: " << tot << endl;
        cout << "percentage is: " << per << endl;
        }
  };
  void main()
{
        exam e1;
        clrscr();
        e1.get();
        e1.display();
  getch();
}
```

Multilevel inheritance	**output**

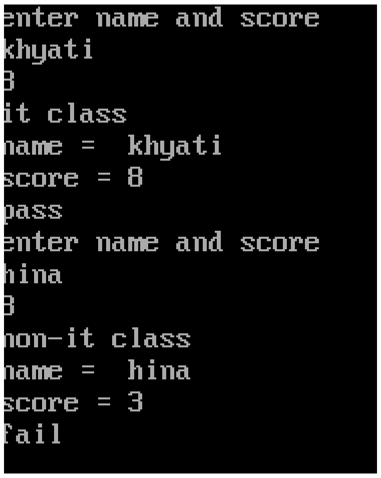
class cust
{
protected:
char nm[15],type;
public:
void get()
{
cout << " enter customer name and type (R/N)";
cin >> nm >> type;
}
	

```
 enter customer name and type (R/N)
khyati
R
enter amount
10000
namekhyati
typeR
bill amount10000
net amount = 7500
```

```
void show()
{
cout << "name" << nm << endl;
cout << "type" << type << endl;
}
};
class bill : public cust
{
protected :
        int amt;
public :
void get()
{
cust :: get();
cout << "enter amount";
cin >> amt;
}
void show()
{
cust ::show();
cout << "bill amount" << amt << endl;
}
};
class payment : public bill
{
private:
        float dis,namt;
public :
void show()
{
bill :: show();
if ( type=='R')
{
dis= 0.25 * amt;
}
else
{
dis= 0.10 * amt;
}
namt= amt-dis;
cout << "net amount = " << namt <<endl;
}
};
void main()
{
payment p1;
clrscr();
p1.get();
p1.show();
getch();
}
```

Hierarchical inheritance	**Output**

```
class stud
{
char nm[15];
protected:
int score ;
public:
void get()
{
cout << "enter name and score";
cin >> nm >> score;
}

void show()
{
cout << "name =  " << nm << endl;
cout << "score = " << score << endl;
}
};
class it : public stud
{
public :
void show()
{
cout << "it class"<< endl;
stud :: show();
if ( score >=7)
```

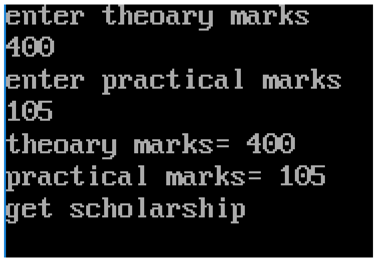


```
enter name and score
khyati
8
it class
name =  khyati
score = 8
pass
enter name and score
hina
3
non-it class
name =  hina
score = 3
fail
```

```
        cout << "pass"<<endl;
else
        cout << "fail"<< endl;
}
};
class nonit : public stud
{
public :
void show()
{
 cout << "non-it class"<< endl;
stud :: show();
if ( score >=5)
        cout << "pass";
else
        cout << "fail";
}
};
void main()
{
it i;
 clrscr();
 i.get();
 i.show();

nonit n;
```

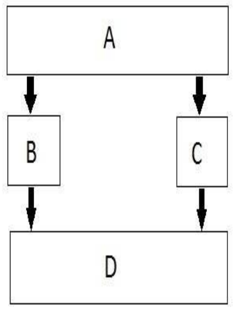
n.get() ;
n.show() ;
getch();
}	

Multiple inheritance	**output**

class t
{
protected:
int tmarks;
public :
void get ()
{
cout << "enter theoary marks";
cin >> tmarks;
}
void show()
{
cout << "theoary marks= " << tmarks << endl ;
}
};
class p
{
protected:
int pmarks;
public :
void get ()
	
	enter theoary marks
	400
	enter practical marks
	105
	theoary marks= 400
	practical marks= 105
	get scholarship

```
{
cout << "enter practical marks";
cin >> pmarks;
}
void show()
{
cout << "practical marks= " << pmarks << endl ;
}
};
class scholar : public t , public p
{
public:
void get()
{
t :: get();
p :: get();
}
void show()
{
t::show();
p:: show();
if( t::tmarks >= 300  && p::pmarks>= 100)
{
cout << "get scholarship" << endl ;
}
else
{
cout << "sorry"<< endl ;
}
}
};
void main()
{
scholar s;
s.get();
s.show();
getch();
}
```

❖ **Virtual Base Class:**



D=D+C+A
D=D+B+A

D= D+C+B+2A
(Two copy of class A we get so error of ambiguity arise)

- ➔ In situation, where we require use of multiple, hierarchical and multilevel inheritance as shown in above fig.
- ➔ In this case , class D have two direct base classes (b,c) and these two base classes have a common base class A. in this situation class D inherits member of class A via two seprate paths.
- ➔ So, class A is also called **indirect base class** of class D
- ➔ All the public and protected member of class A are inherited in class D two times
- ➔ It means class D has duplicate set of members inherited from class A
- ➔ And because of this problem ambiguity arises
- ➔ This duplication of inherited members can be avoided by making **common base class** as **virtual base class** while declaring declaring direct or intermediate base class as shown below:
- ➔ Class A
  Class B : public virtual A
  Class C : public virtual A

Class D : public B , public C

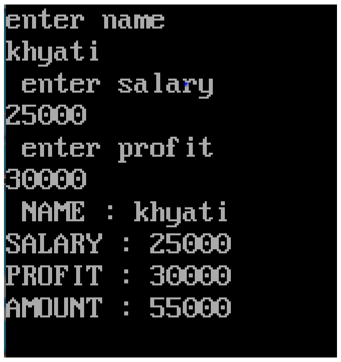
➔ When a class is made **virtual base class** c++ compiler takes care that only one copy of that class is inherited regardless of number of inherited paths exist between virtual base class and derive class

NOTE : refer the example of hybrid inheritance

Hybrid inheritance

```
class person
{
char nm[15] ;
public :
void get ()
{
cout << "enter name";
cin >> nm;
}
void show()
{
cout << " NAME : " << nm << endl;
}
};
class job : public virtual person
{
protected:
long sal;
public :
```

Output



```
enter name
khyati
 enter salary
25000
 enter profit
30000
 NAME : khyati
SALARY : 25000
PROFIT : 30000
AMOUNT : 55000
```

```
void get()
{
cout << " enter salary " ;
cin >> sal;
}
void show()
{
cout << "SALARY : " << sal << endl;
}
};
class bussiness : public virtual person
{
protected:
long profit;
public :
void get()
{
cout << " enter profit " ;
cin >> profit;
}
void show()
{
cout << "PROFIT : " << profit << endl;
}
};
```

```
class netearning : public job , public bussiness
{
long amt;
public:
void get()
{
person :: get();
job :: get() ;
bussiness :: get();
}
void show()
{
person :: show();
job:: show();
bussiness :: show();
amt= sal+profit;
cout << "AMOUNT : "<< amt << endl;
}
};
void main()
{
netearning n;
clrscr();
n.get();
n.show();
```

getch(); }	

2.3.2 Constructor and destructor call during inheritance

→ The default constructor and destructor of base class are always called when a new object of derived class is created or destroyed

- **Constructor and Inheritance**
  - ⇨ The compiler automatically calls a base class constructor before executing the derived class constructor.
  - ⇨ The compiler's default action is to call the default constructor in the base class.
  - ⇨ If you want to specify which of several base class constructors should be called during the creation of a derived class object.
  - ⇨ In these cases, you must explicitly specify which base class constructor should be called by the compiler.
  - ⇨ This is done by specifying the arguments to the selected base class constructor in the definition of the derived class constructor.
  - ⇨ When both base class and derived class contain constructor at that time base class constructor is executed first then the statements in derive class constructor are executed
  - ⇨ Without declaration of constructor in base class if you called derived class constructor it will raise an error

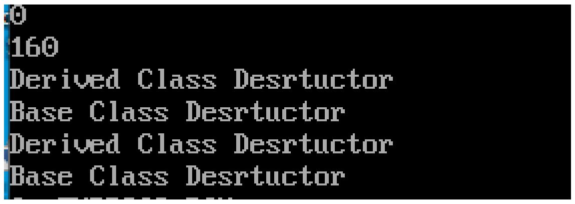
- **Destructor and Inheritance**
  - ⇨ In inheritance, destructors are executed in reverse order of constructor execution.
  - ⇨ The destructors are executed when an object goes out of scope.
  - ⇨ When both base class and derived class contain destructor at that time derive class destructor is executed first then the base class destructor is executed

Example 1

```
class Rectangle

{

public:

float length; float width;

Rectangle () //Default constructor

{

length = 0;

width = 0;

}
```

```
Rectangle (float len, float wid) // parameterized constructor
{
length = len; width = wid;
}
~Rectangle() //destructor
{
cout<< "Base Class Desrtuctor";
}
float area()
{
return length * width ;
}
};
class Box : public Rectangle
{
public:
float height;
Box () //default constructor
{
height = 0;
}
Box (float len, float wid, float ht) : Rectangle(len, wid) //calling constructor of parent class
{
height = ht;
}
~Box() //destructor
{
```

```
cout<<"Derived Class Desrtuctor";
}
float volume()
{
return area() * height;
}
};
void main ()
{
Box bx;
clrscr();
Box cx(4,8,5);
cout << bx.volume() << endl;
cout << cx.volume() << endl;
getch();
}
```



```
0
160
Derived Class Desrtuctor
Base Class Desrtuctor
Derived Class Desrtuctor
Base Class Desrtuctor
```

C++ this Pointer

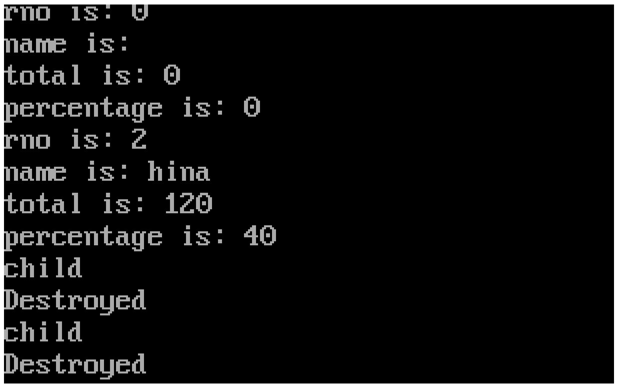
In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

Example 2

```
#include < string.h>

class stud
{

          int rno;
          char nm[15];

     public:


          stud()
          {
          rno=0;
          strcpy(nm," ");
          }

          stud(int rno,char nm[])
          {
          this->rno=rno;
          strcpy(this->nm,nm);
          }

      ~stud()
          {
          cout << "Destroyed";
          }

          void display()
          {
              cout << "rno is: " << rno << endl;
              cout << "name is: " << nm << endl;
          }
};

//exam class is derived from base class stud.

class exam : public stud
{
          int m1,m2,m3;
```

```
  public:



  exam()
  {
  m1=m2=m3=0;
  }

 exam (int rno,char nm[],int m1,int m2,int m3) : stud (rno,nm)
  {
  this->m1=m1;
  this->m2=m2;
  this->m3=m3;
 }

  ~exam()
  {
  cout << "child";
  }

        void display()
        {
                stud :: display();

                int tot = m1+m2+m3;
                float per=tot/3.0;

                cout << "total is: " << tot << endl;
                cout << "percentage is: " << per << endl;
        }
};


void main()
{
        clrscr();
        exam e;
        e.display();

        exam e2(2,"hina",30,40,50) ;
        e2.display();

        getch();
}
```


```
rno is: 0
name is:
total is: 0
percentage is: 0
rno is: 2
name is: hina
total is: 120
percentage is: 40
child
Destroyed
child
Destroyed
```

2.3.3 Abstract Class

Abstract Class

- ➔ Abstract Class is a class which contains at least one Pure Virtual function in it.
- ➔ Abstract classes are used to provide an Interface for its sub classes.
- ➔ Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class:

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.

2. Abstract class can have normal functions and variables along with a pure virtual function.

3. Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.

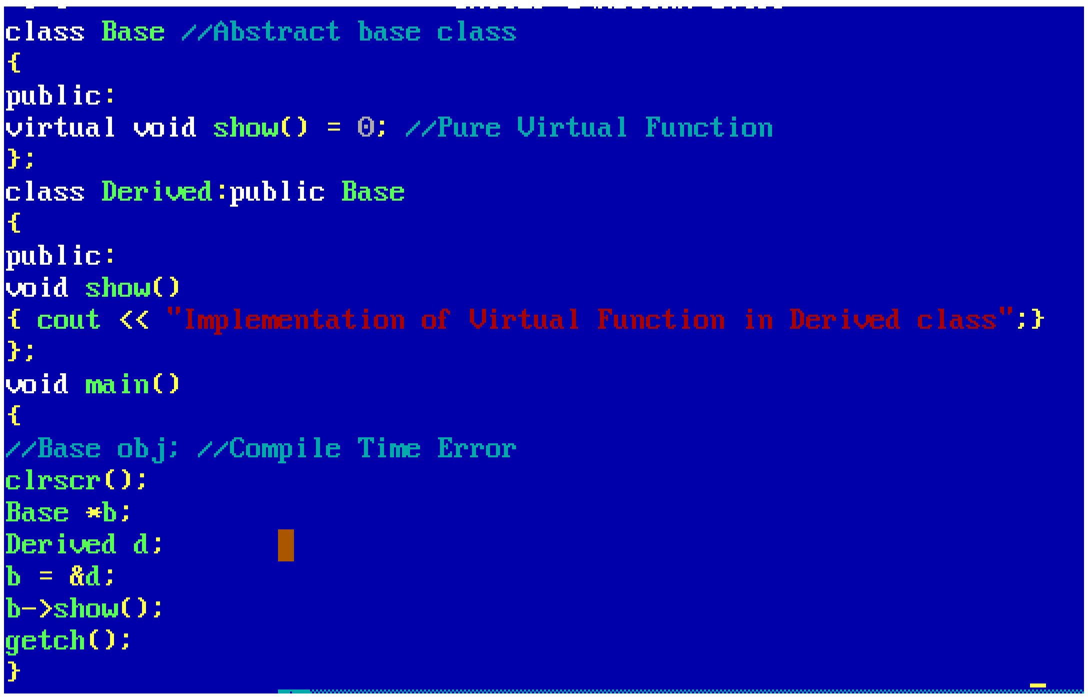
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

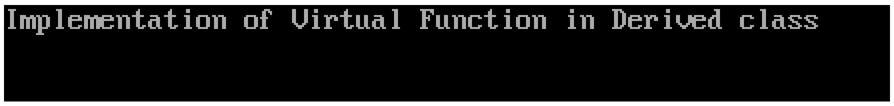
Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with "= 0" Here is the syntax for a pure virtual function,

virtual void f() = 0;

Example of Abstract Class:



```
class Base //Abstract base class
{
public:
virtual void show() = 0; //Pure Virtual Function
};
class Derived:public Base
{
public:
void show()
{ cout << "Implementation of Virtual Function in Derived class";}
};
void main()
{
//Base obj; //Compile Time Error
clrscr();
Base *b;
Derived d;
b = &d;
b->show();
getch();
}
```

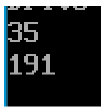


```
Implementation of Virtual Function in Derived class
```

Overloading and overriding

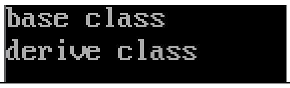
Function overloading	Function Overriding
It is possible only in single class	It is possible only in inheritance
It means two function with the same name but different signature means different number of argument, different data type or different data type for different argument	It means two function with the same name and same signature but in inherited class
It is possible for function, constructor and operator	It is possible only for function
Example:	**Example:**

```
class Addition
{
public:
   int sum(int num1,int num2)
{
    return num1+num2;
   }

   int sum(int num1,int num2, int num3)
 {
      return num1+num2+num3;
   }
};

void main()
 {
   Addition obj;
   cout<<obj.sum(20, 15)<<endl;
   cout<<obj.sum(81, 100, 10) << endl;
   getch();
}
```



```
35
191
```

```
class A
{
public:
void show()
{
cout << "base class" << endl;
}
};
class B : public A
{
public:
void show()
{
A::show();
cout << "derive class" << endl;
}
};
void main()
 {
   B b;
   b.show();
   getch();}
```



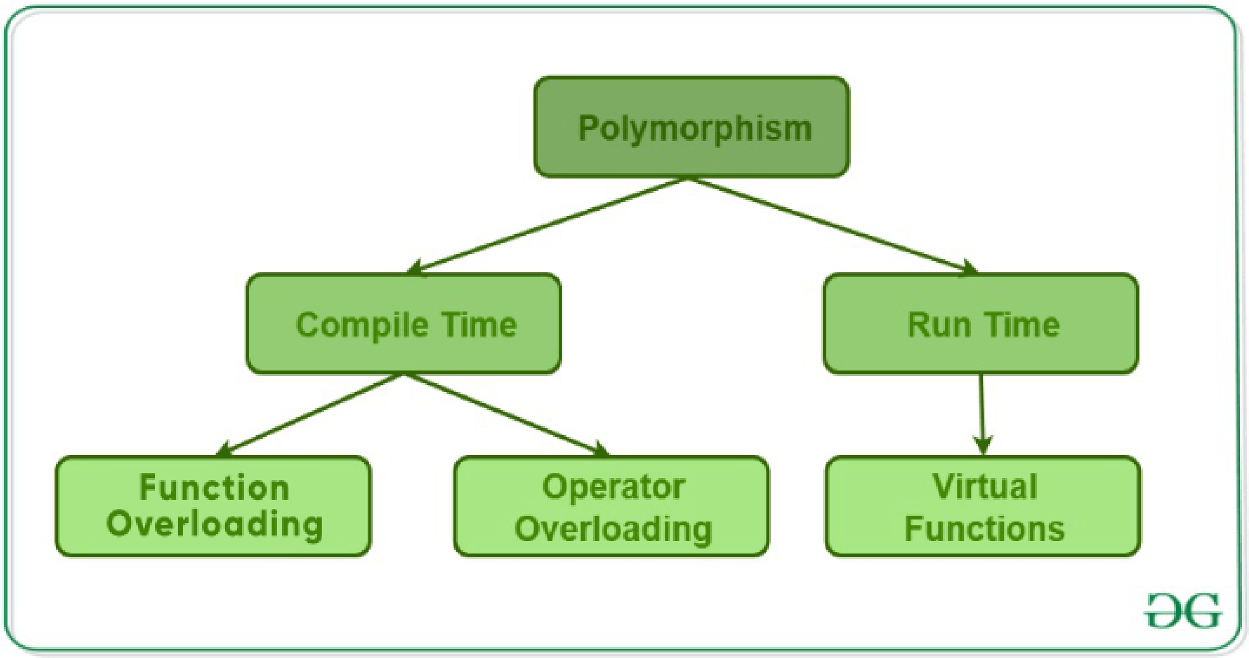
```
base class
derive class
```

Polymorphism

- ➔ Polymorphism simply means “One name And Multiple behavior”.
- ➔ Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation.
- ➔ Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ.
- ➔ Polymorphism is extensively used in implementing inheritance.
- ➔ For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading**.
- ➔ A single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as **function overloading**.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



1. compile time /Static Polymorphism/ early binding

- ➔ Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called *early binding*. The term *early binding* stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.
- ➔ C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called **function overloading and operator overloading** respectively.
- ➔ An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- ➔ When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

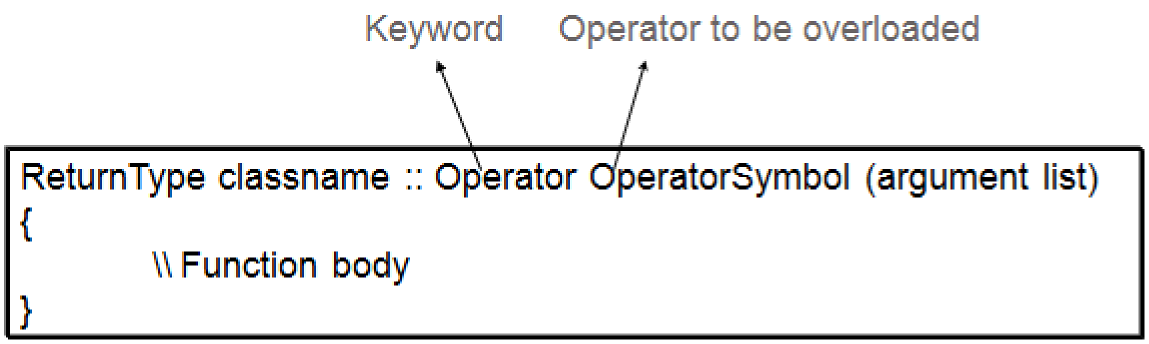
C++ Operators Overloading

- ➔ Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- ➔ In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- ➔ For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.
- ➔ Operator overloading is used to overload or redefines most of the operators available in C++.
- ➔ It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.
- ➔ The **advantage** of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(.*)
- ternary operator(?:)
- () , []

Syntax of Operator Overloading



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. **Member Function**

return_type operator op(argument_list)
{
// body of the function. **Inside class declaration**
}

2. **Non-Member Function**

return_type class_name : : operator op(argument_list)
{
// body of the function. **Outside class declaration**
}

3. Friend Function

- Friend function using operator overloading offers better flexibility to the class.
- These functions are not a member of the class and they do not have 'this' pointer.
- When you overload a unary operator, you have to pass one argument.
- When you overload a binary operator, you have to pass two arguments.
- Friend function can access private members of a class directly.

```
friend return-type operator operator-symbol (Variable 1, Varibale2)
{
   //Statements;
}
```

Where the **return type** is the type of value returned by the function. **class_name** is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

<u>Rules for Operator Overloading</u>

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.
- We cannot change the basic meaning of an operator
- Overloaded operator follows syntax rules of original operator they cannot be overridden

<u>What is the difference between operator functions and normal functions?</u>

Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

EXAMPLE 1 :

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++ and --.

```
 class Test
{
   private:
      int num;

   public:
       Test()
             {
              num =8;
             }


  /*  void operator ++()
       {
              num = num+2;
       } */

             void  operator --()
       {
              num = num-2;
       }

             void Print()
              {
                cout<<"The Count is: "<<num<<endl;
              }

      void operator ++();
};
```

//outside class declaration

```
       void Test:: operator ++()
             {
             num = num+2;
}
```

```
void main()
{
   Test tt;

   ++tt;  // calling of a function "operator ++()"

Test tt1;

   --tt1;  // calling of a function “operator --()"

tt.Print();

tt1.Print();

getch();

}
```

Output:

```
The Count is: 10
The Count is: 6
```

EXAMPLE 2 :

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators + and -.

```
class A

{

   int x;

     public:

   A(int i)

   {

      x=i;

   }
   void operator+(A a)
{
   int m = a.x + x;
   cout << "\n";
   cout<<"The result of the addition of two objects is : "<<m;
}
   void operator-(A a)
   {
   int m = x  - a.x ;
   cout << "\n";
   cout<<"The result of the summation of two objects is : "<<m;
   }
  };

void main()
{
          clrscr();
          A a1(5);
          A a2(4);
          a1 + a2; // calling the operator + function  object1 operator object2
          a1 - a2;
          getch();
}
```

Output:

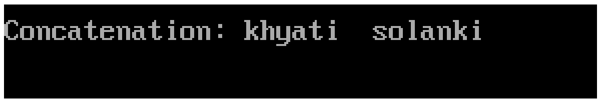
```
The result of the addition of two objects is : 9
The result of the summation of two objects is : 1
```

EXAMPLE 3 :

Let's see a simple example of overloading the unary operators with string.

// program to overload the unary operators + with string

```
class AddString
 {

public:

    char s1[25], s2[25];

    // Parametrized Constructor
    AddString(char str1[], char str2[])
    {
     // AddString a1(str1, str2);
                // Initialize the string to class object
                strcpy(this->s1, str1);
                strcpy(this->s2, str2);
    }

    // Overload Operator+ to concat the string
    void operator+()
    {
                cout << "\nConcatenation: " << strcat(s1, s2);
    }
};
void main()
{
clrscr();
    // Declaring two strings

  char str1[] = "khyati";
  char str2[] = "  solanki";
  // Declaring and initializing the class
    // with above two strings

    AddString a1(str1, str2);

    // Call operator function

    +a1;
getch();
}
```



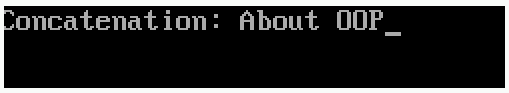
```
Concatenation: khyati  solanki
```

EXAMPLE 4 :

Let's see a simple example of overloading the binary operators with string.

// program to overload the binary operators + with string

```
class AddString
 {

public:

    char str[100];

    // default constructor

     AddString(){}

    // Parametrized constructor to
    // initialize class Variable

    AddString(char str[])
    {
            strcpy(this->str, str);
    }

    // Overload Operator+ to concatenate the strings

    AddString operator+(AddString& S2)
    {

            // Object to return the copy
            // of concatenation
            AddString S3;

            // Use strcat() to concat two specified string
            strcat(this->str, S2.str);
            // Copy the string to string to be return
            strcpy(S3.str, this->str);

            // return the object
            return S3;
   }
};


void main()
{
clrscr();
   // Declaring two strings

   char str1[] = "About ";
   char str2[] = "OOP";

   // Declaring and initializing the class
   // with above two strings

   AddString a1(str1);
   AddString a2(str2);

   AddString a3;

   // Call the operator function

   a3 = a1 + a2;

   cout << "Concatenation: " << a3.str;

   getch();
}
```



```
Concatenation: About OOP_
```

Overloading I/O operator

- ➔ Overloaded to perform input/output for user defined datatypes. Left Operand will be types ostream& and istream&
- ➔ Function overloading this operator must be a Non-Member function because left operand is not an Object of the class.

- ➔ It must be a friend function to access private data members.
- ➔ You have seen above that << operator is overloaded with ostream class object cout to print primitive type value output to the screen. Similarly you can overload << operator in your class to print user-defined type to screen.

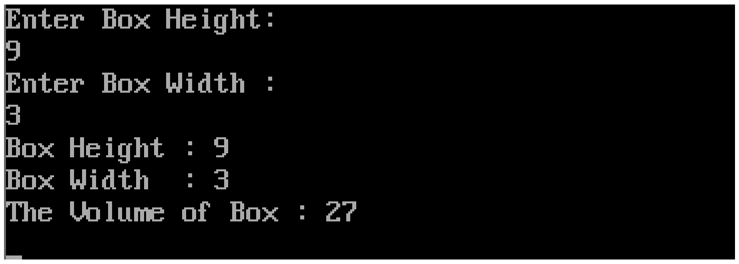
NOTE: When the operator does not modify its operands, the best way to overload the operator is via friend function.

EXAMPLE 5 :

Let's see a simple example of overloading the I/O operators.

```
// program to overload the I/O operators >> and <<

class Box

{

   double height;

   double width;

   double vol ;

   public :

         friend istream & operator >> (istream &din, Box &b);    // cin >>

         friend ostream & operator << (ostream &dout, Box &b);   // cout <<

 };

 istream & operator >> (istream &din, Box &b)

 {       cout << "Enter Box Height: " << endl;

         din >>b.height;

         cout << "Enter Box Width : " << endl;

         din >> b.width;

         return din ; }
```

```
ostream & operator << (ostream &dout, Box &b)

{

            dout << "Box Height : " << b.height << endl ;

            dout << "Box Width  : " << b.width << endl ;

            b.vol = b.height * b.width ;

            dout << "The Volume of Box : " << b.vol << endl;

            return dout ;

 }

 void main()

 {

      Box b1;

     clrscr();

     cin >> b1;

     cout << b1;

getch();

}
```



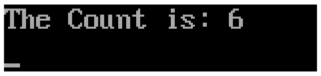
```
Enter Box Height:
9
Enter Box Width :
3
Box Height : 9
Box Width  : 3
The Volume of Box : 27
_
```

<u>EXAMPLE 6 :</u>

Let's see a simple example of overloading the unary operators with friend function.

```
// program to overload the unary operators - using friend function

class Test

{

  private:

     int num;

  public:

      Test()

           {

            num =8;

           }

           friend void  operator --(Test &t);

           void Print()

            {

              cout<<"The Count is: "<<num<<endl;

            }

};

    void  operator --(Test &t)

      {

             t.num = t.num-2;

     }
```

```
void main()
{
   Test tt ;
   clrscr();
   --tt;
   tt.Print();
    getch();
}
```



```
The Count is: 6
_
```

<u>EXAMPLE 7 :</u>

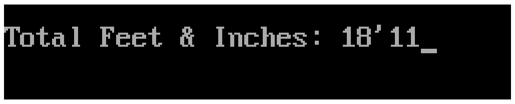
Let's see a simple example of overloading the binary operators with friend function.

// program to overload the binary operators +using friend function

```
class Distance
 {
public:
   int feet, inch;
     // No Parameter Constructor
   Distance()
   {
      this->feet = 0;
      this->inch = 0;
   }
```

```
     // Constructor to initialize the object's value
    // Parametrized Constructor
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
     // Declaring friend function using friend keyword
    friend Distance operator+(Distance&d1, Distance&d2);
};
 // Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
    // Create an object to return
    Distance d3;
    // Perform addition of feet and inches
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.inch + d2.inch;
    // Return the resulting object
    return d3;
}
```

```
void main()
{
   clrscr();
   // Declaring and Initializing first object
   Distance d1(8, 9);
   // Declaring and Initializing second object
   Distance d2(10, 2);
   // Declaring third object
   Distance d3;
   // Use overloaded operator
   d3 = d1 + d2;
   // Display the result
   cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
         getch();
}
```



```
Total Feet & Inches: 18'11_
```

Function Overloading

- ➔ If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.
- ➔ Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Ways to overload a function

- ✓ By changing number of Arguments.
- ✓ By having different types of argument.

Number of Arguments different

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return sum of two and three integers.

```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}
```

Here sum() function is overloaded, to have two and three arguments. Which sum() function will be called, depends on the number of arguments.

```
int main()
{
sum (10,20); // sum() with 2 parameter will be called

sum(10,20,30); //sum() with 3 parameter will be called

}
```

Different Data type of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

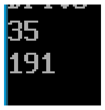
```
int sum(int x,int y)
{
 cout<< x+y;
}

double sum(double x,double y)
{
 cout << x+y;
}

int main()
{
 sum (10,20);
 sum(10.5,20.5
 );
}
```

<u>Example:</u>

```
class Addition

{

public:

   int sum(int num1,int num2)

{

    return num1+num2;

   }

   int sum(int num1,int num2, int num3)

 {

      return num1+num2+num3;

   }

};
```

```
void main()
 {
   Addition obj;
   cout<<obj.sum(20, 15)<<endl;
   cout<<obj.sum(81, 100, 10) << endl;
   getch();
}
```

Output:



```
35
191
```

Static Vs Dynamic Polymorphism

1. Static polymorphism is considered more efficient and dynamic polymorphism more flexible.
2. Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.
3. As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.

C++ Polymorphism

The word “polymorphism” means having many forms.

In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

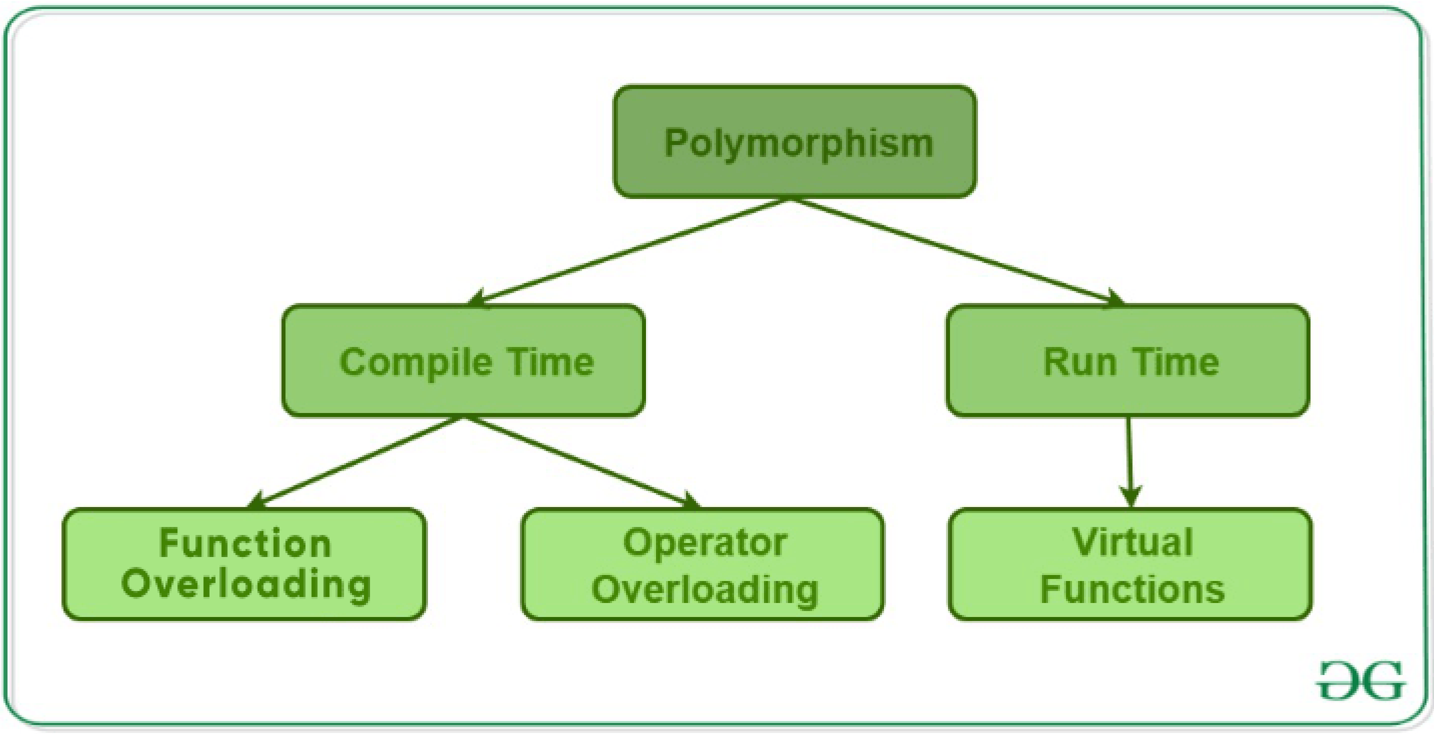
A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.

So the same person exhibits different behaviour in different situations. This is called polymorphism.

Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**



Types of Polymorphism

1. Compile-Time Polymorphism

In compile-time polymorphism, a function is called at the time of program compilation. We call this type of polymorphism as early binding or Static binding.

This type of polymorphism is achieved by function overloading **or** operator overloading.

A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as **Function Overloading**.
Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**.
In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.

There are certain Rules of Function Overloading that should be followed while overloading a function.
Below is the C++ program to show function overloading or compile-time polymorphism:

- C++

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Temp {
public:
    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name and
    // 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y
             << endl;
    }
};

// Driver code
int main()
{
    Temp obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

Output

```
value of x is 7
value of x is 9.132

value of x and y is 85, 64
```

Explanation: In the above example, a single function named function **func()** acts differently in three different situations, which is a property of polymorphism.

B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.

Operator overloading is used to overload or redefines most of the operators available in C++.

It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function. We define the function inside the class whose objects/variables we want the overloaded operator to work with.

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

- `returnType` is the return type of the function.
- `operator` is a keyword.
- `symbol` is the operator we want to overload. Like: `+`, `<`, `-`, `++`, etc.
- `arguments` is the arguments passed to the function.

Unary Operators and Binary Operator overloading

Unary operators:

- Operators which work on a single operand are called unary operators.
- Examples: Increment operators(++), Decrement operators(−−),unary minus operator(-), Logical not operator(!) etc…

Binary operators:

- Operators which works on Two operands are called binary operator.

Implementing Operator overloading:

- Member function: It is in the scope of the class in which it is declared.
- Friend function: It is a non-member function of a class with permission to access both private and protected members.

Operator that can be overloaded are as follows:

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operator that cannot be overloaded are as follows:

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Below is the C++ program to demonstrate operator overloading:

- CPP

/ program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()        {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
```

```
{

    int x;
      public:
      A(){}
    A(int i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

```
The result of the addition of two objects is : 9
```

2. Runtime Polymorphism

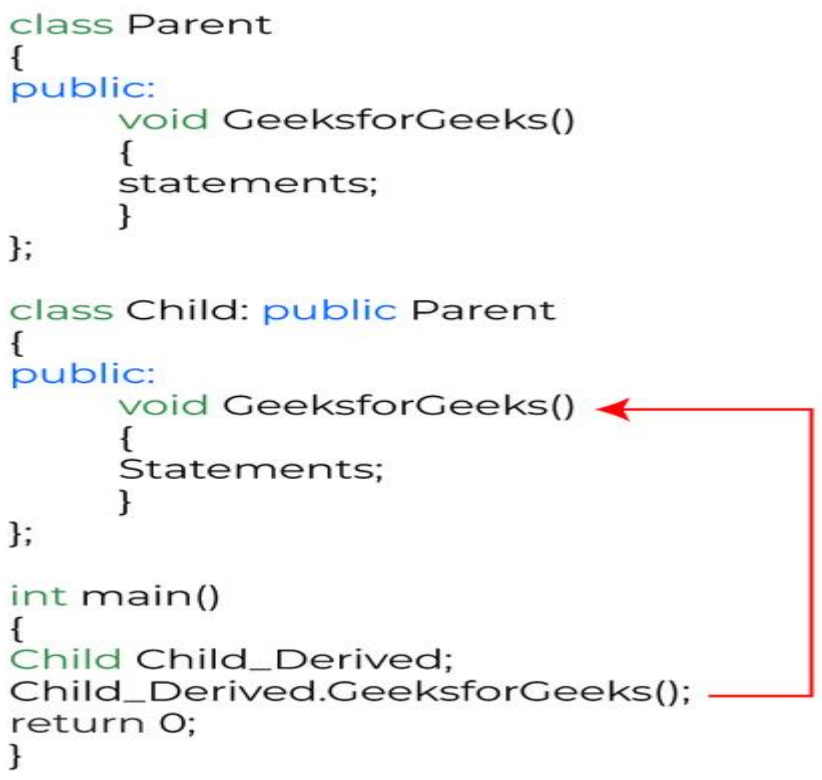
This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.
Syntax:

```
class Parent{
  access_modifier:
    // overridden function
    return_type name_of_the_function(){}
  };
}
  class child : public Parent {
    access_modifier:
      // overriding function
      return_type name_of_the_function(){}
    };
  }
```

Example:



```
class Parent
{
public:
        void GeeksforGeeks()
        {
        statements;
        }
};

class Child: public Parent
{
public:
        void GeeksforGeeks()
        {
        Statements;
        }
};

int main()
{
Child Child_Derived;
Child_Derived.GeeksforGeeks();
return 0;
}
```

Function overriding Explanation

```cpp
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks_Print()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void GeeksforGeeks_Print()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Child Child_Derived;
    Child_Derived.GeeksforGeeks_Print();
    return 0;
}
```

Output

```
Derived Function
```

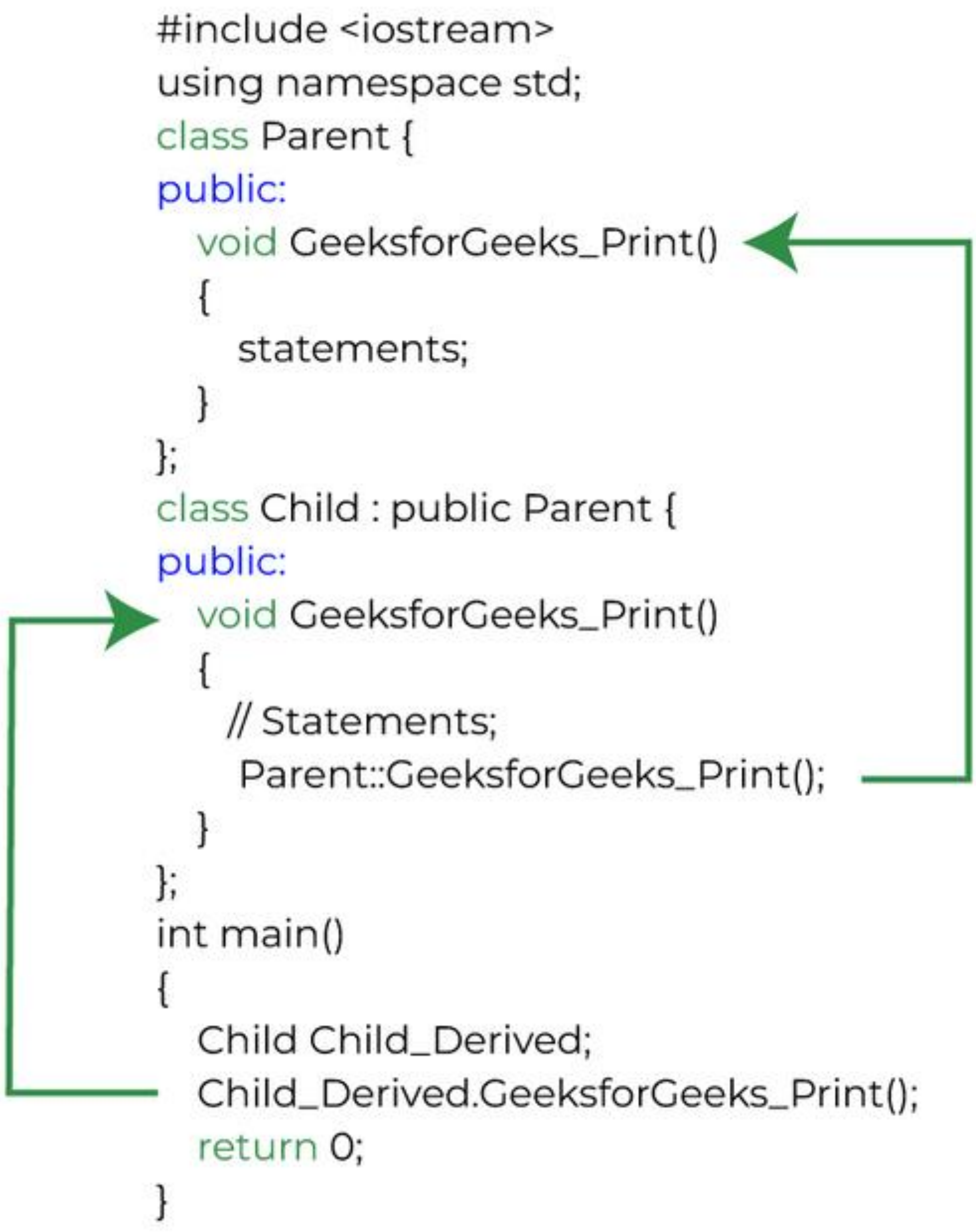
Variations in Function Overriding

1. Call Overridden Function From Derived Class

```cpp
// C++ program to demonstrate function overriding
// by calling the overridden function
// of a member function from the child class

#include <iostream>
using namespace std;

class Parent {
public:
    void Print()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void Print()
    {
        cout << "Derived Function" << endl;

        // call of overridden function
        Parent::Print();
    }
};

int main()
{
    Child Child_Derived;
    Child_Derived.Print();
    return 0;
}
```

Output

```
Derived Function

Base Function
```



```
#include <iostream>
using namespace std;
class Parent {
public:
   void GeeksforGeeks_Print()
   {
      statements;
   }
};
class Child : public Parent {
public:
   void GeeksforGeeks_Print()
   {
     // Statements;
      Parent::GeeksforGeeks_Print();
   }
};
int main()
{
   Child Child_Derived;
   Child_Derived.GeeksforGeeks_Print();
   return 0;
}
```

*The output of **Call Overridden Function From Derived Class***

2. Call Overridden Function Using Pointer

```
// C++ program to access overridden function using pointer
// of Base type that points to an object of Derived class
#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
```

```
public:
    void GeeksforGeeks()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Child Child_Derived;

    // pointer of Parent type that points to derived1
    Parent* ptr = &Child_Derived;

    // call function of Base class using ptr
    ptr->GeeksforGeeks();

    return 0;
}
```

Output

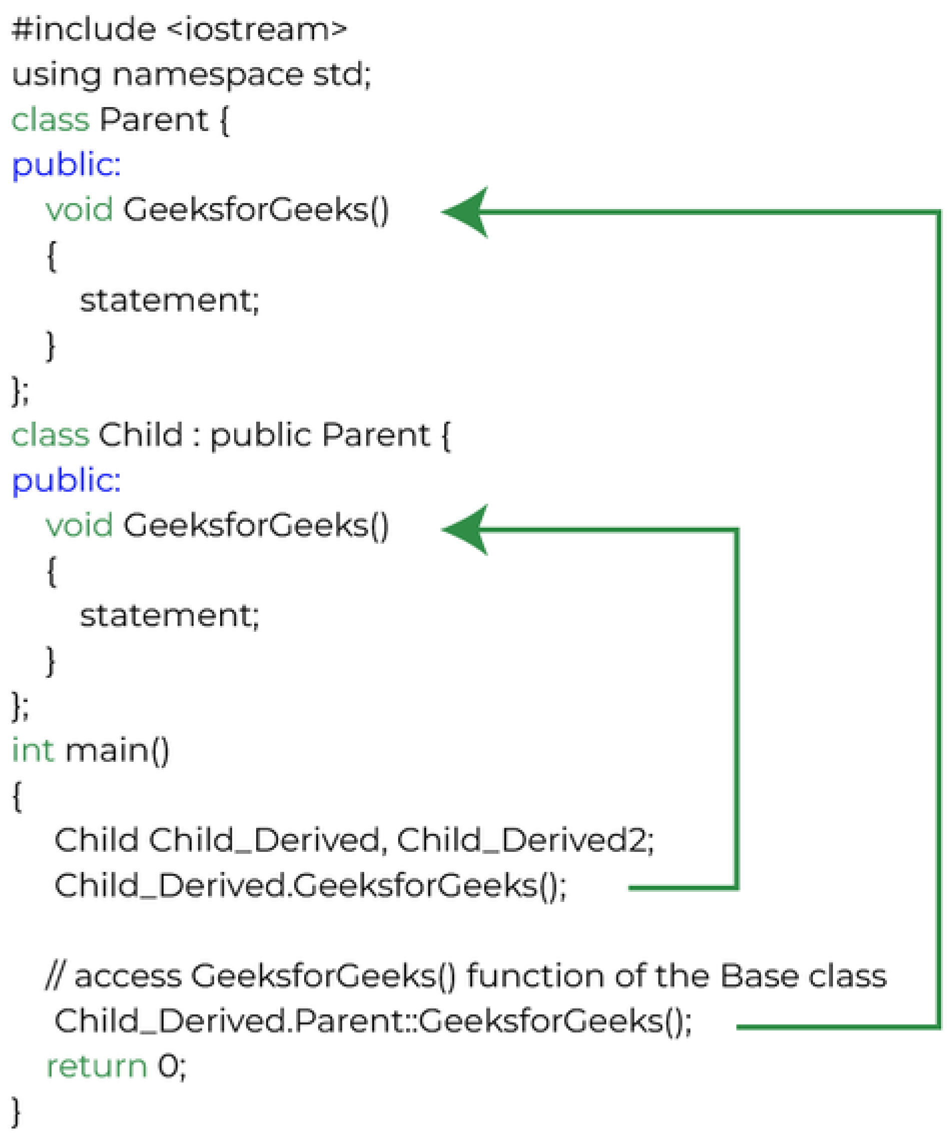
```
Base Function
```

3. Access of Overridden Function to the Base Class

```
// C++ program to access overridden function
// in main() using the scope resolution operator ::

#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
     Child Child_Derived;
     Child_Derived.GeeksforGeeks();

    // access GeeksforGeeks() function of the Base class
     Child_Derived.Parent::GeeksforGeeks();
    return 0;
}
```

Output

```
Derived Function

Base Function
```



```
#include <iostream>
using namespace std;
class Parent {
public:
   void GeeksforGeeks()
   {
      statement;
   }
};
class Child : public Parent {
public:
   void GeeksforGeeks()
   {
      statement;
   }
};
int main()
{
    Child Child_Derived, Child_Derived2;
    Child_Derived.GeeksforGeeks();

   // access GeeksforGeeks() function of the Base class
    Child_Derived.Parent::GeeksforGeeks();
   return 0;
}
```

Access of Overridden Function to the Base Class

4. Access to Overridden Function

```cpp
// C++ Program Demonstrating
// Accessing of Overridden Function
#include <iostream>
using namespace std;

// defining of the Parent class
class Parent

{

public:
    // defining the overridden function
    void GeeksforGeeks_Print()
    {
        cout << "I am the Parent class function" << endl;
    }
};

// defining of the derived class
class Child : public Parent

{
public:
    // defining of the overriding function
    void GeeksforGeeks_Print()
    {
        cout << "I am the Child class function" << endl;
    }
};

int main()
{
    // create instances of the derived class
    Child GFG1, GFG2;

    // call the overriding function
    GFG1.GeeksforGeeks_Print();

    // call the overridden function of the Base class
    GFG2.Parent::GeeksforGeeks_Print();
    return 0;
}
```

Output

```
I am the Child class function

I am the Parent class function
```

Function Overloading Vs Function Overriding

Function Overloading	Function Overriding
It falls under Compile-Time polymorphism	It falls under Runtime Polymorphism
A function can be overloaded multiple times as it is resolved at Compile time	A function cannot be overridden multiple times as it is resolved at Run time
Can be executed without inheritance	Cannot be executed without inheritance
They are in the same scope	They are of different scopes.

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of the derived class.

```cpp
// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

//  base class declaration.
class Animal {
public:
    string color = "Black";
};

// inheriting Animal class.
class Dog : public Animal {
public:
    string color = "Grey";
};

// Driver code
int main(void)
{
    Animal d = Dog(); // accessing the field by reference
                      // variable which refers to derived
    cout << d.color;
}
```

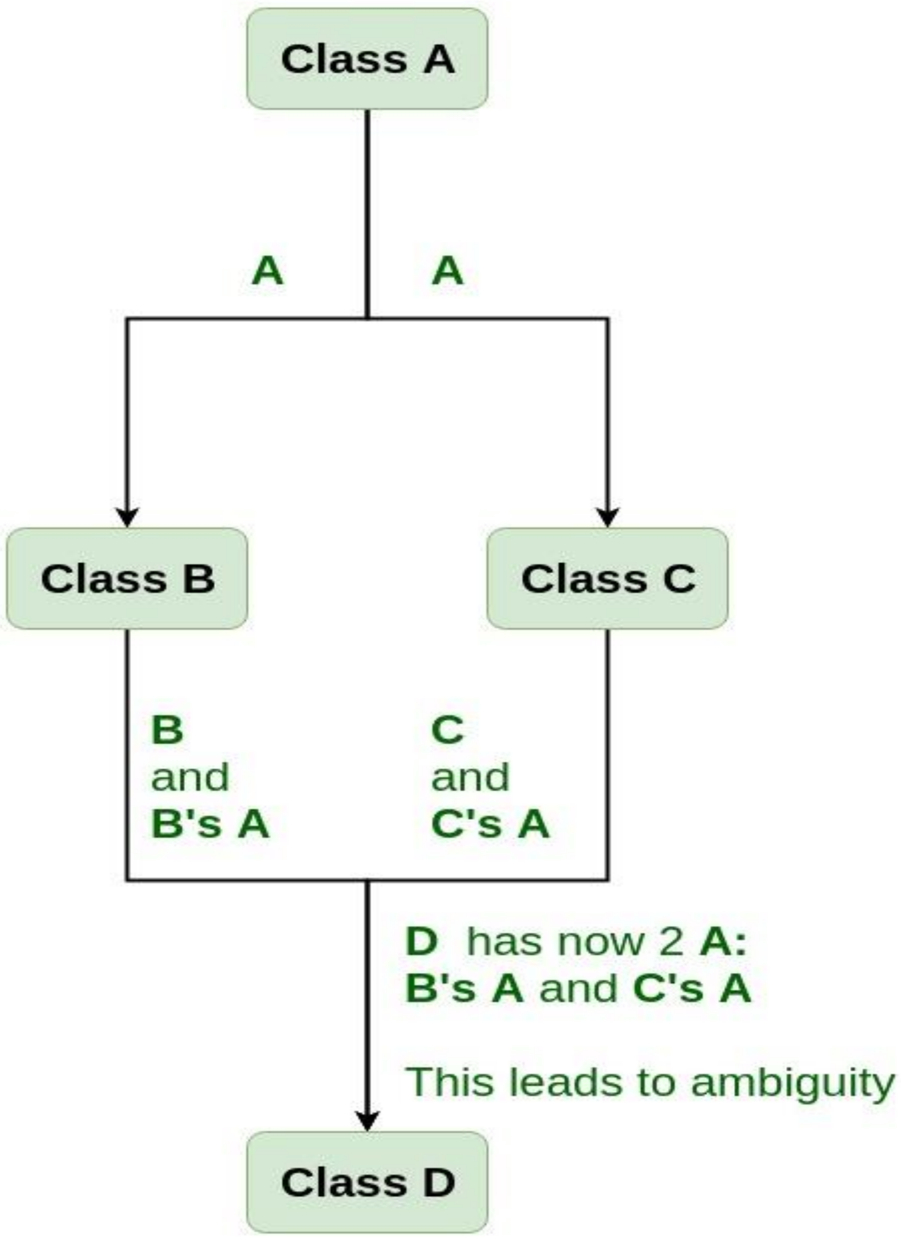
Output

```
Black
```

Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes: Consider the situation where we have one class **A** . This class **A** is inherited by two other classes **B** and **C.** Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, **ambiguity arises as to which data/function member would be called?** One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Example: To show the need of Virtual Base Class in C++

```
#include <iostream>
using namespace std;

class A {
public:
    void show()
    {
        cout << "Hello form A \n";
    }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

Compile Errors:

```
prog.cpp: In function 'int main()':
prog.cpp:29:9: error: request for member 'show' is
ambiguous
  object.show();
         ^
prog.cpp:8:8: note: candidates are: void A::show()
   void show()
        ^
prog.cpp:8:8: note:                 void A::show()
```

How to resolve this issue?

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

Note:

virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example 1

```
#include <iostream>
using namespace std;

class A {
public:
    int a;
    A() // constructor
    {
        a = 10;
    }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;

    return 0;
}
```

Output:

```
a = 10
```

Explanation :
The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

Example 2:

```
#include <iostream>
using namespace std;

class A {
public:
    void show()
    {
        cout << "Hello from A \n";
    }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

Output:
```
Hello from A
```

B. Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

Below is the C++ program to demonstrate virtual function:

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
        std::cout << "Value of y is : " <<y<< std::endl;
    }
};
int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
```

Output:

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
 class A
{
   int x=5;
    public:
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
```

```
    a->display();   //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
```

```
    void show()
    {
        std::cout << "Derived class is derived from the base class." << std::en
dl;
    }
};
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

Output:

```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

Compile time polymorphism	**Run time polymorphism**
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
    friend data_type function_name(argument/s);            // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

```
#include <iostream>
using namespace std;
class B;          // forward declarartion.
class A
{
    int x;
    public:
    void setdata(int i)
    {
        x=i;
    }
    friend void min(A,B);         // friend function.
};
```

```
class B
{
    int y;
    public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);                    // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
    std::cout << a.x << std::endl;
    else
    std::cout << b.y << std::endl;
}
   int main()
{
   A a;
   B b;
   a.setdata(10);
   b.setdata(20);
   min(a,b);
    return 0;
 }
```

Unit 4. Data Structure

4.1 Introduction of Data Structure and application areas.
4.2 Recursion concepts
4.3 Difference among Linear and Non-Linear Data Structure
4.4 Stack
- Concepts of Stack(LIFO)
- Pop, Push and Display(Peep)
- Application areas of Stack
(Infix to postfix, Infix to prefix)

Data Structure:

Data Structure is a representation of the logical relationship between or individual elements of data.

- Data may be organized in many ways.
- The logical or mathematical model of particular organization of data is called data structure.
  i.e. Data structure is collection of organized data and operations allowed on it.

The choice of particular data structure model depends on two considerations.

- It must be rich enough in structure to mirror the actual relationships of the data in the real world.
- The structure should be simple enough that we can effectively process the data when necessary.
- Data structure is a collection of data elements whose organization is characteristics by assessing operations that are used to store and retrieve the individual data element.

Type of Data Structures

Data structure can be organized/classified into two categories.

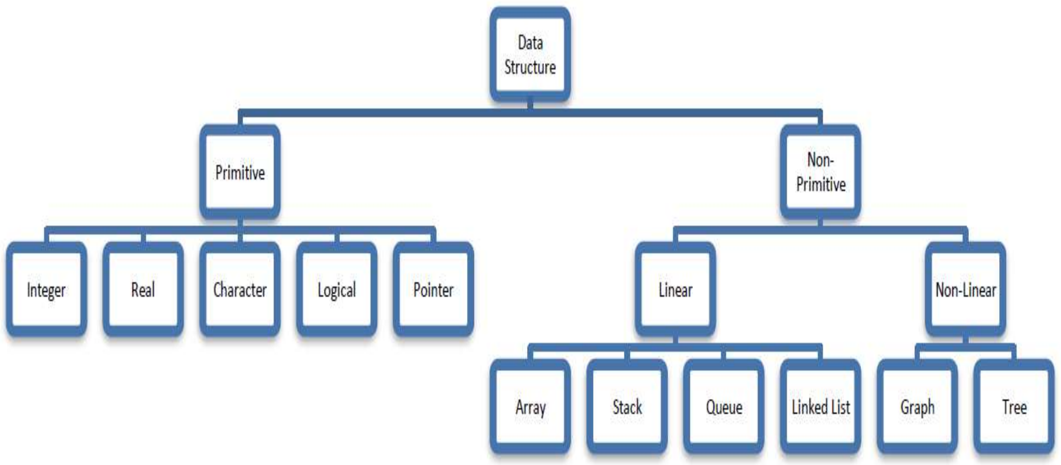
- Primitive
- Non-primitive

The non-primitive data structures can further be classified in two categories.

- Linear

- Non-Linear

The following chart shows various classifications of data structure.



Primitive and Non primitive

Primitive:

Primitive data structure includes data at their most primitive level within a computer. i.e. The data structure that typically are directly operated upon by machine level instructions. Their structure cannot be modified.
e.g. Integer, Real, Character, Pointer, Logical

Non-primitive:
Non-primitive data structures are further classified into two types.

- Linear
- Non-linear data structure.

In a **linear data structure,** the data items are arranged in linear sequence. The linear data structure exhibits the property of adjacency. Various linear data structures are:

- array
- stack

- queue
- strings
- linked list

Non-linear data structure:

The data items are not stored in a sequential format. The non-linear data structure exhibits either hierarchical or parent child relationship. In this data structure insertion and deletion cannot be done in linear fashion. The different non-linear data structures are

- Tree
- Graph

Homogenous and Non-Homogenous Data structures

The data structure can also be classified into homogenous and non-homogenous.

Homogenous:
In this type of data structure all the elements are of same type.
e.g. array

Non-homogenous:
In this type of data structure all the elements are not of same type.
e. g. records

Static or Dynamic data structure

Static Data Structure

These Data Structures are ones whose sizes and structure associated memory location are fixed at compile time. E.g. array
Int a[10];

Dynamic Data Structure

These data structure which expands or shrinks as required during the program execution and their associated memory location change.

E.g. Linked list

Recursion:

A function that call itself or function calls to a second function which eventually

calls original function is called recursive function.
Any recursive procedure must satisfy two conditions:

a. There must be a stopping condition also called base criteria. E.g. In factorial function if argument value is 1 the function does not call itself.
b. Each time the procedure call itself must be nearer to the stopping condition (based criteria) for example in factorial function each time of function is called to itself argument is decremented by one which goes near to the stopping condition. The problem must be in recursive form.

There are mainly two types of recursive function

1. Primitive recursive function:
   The function that directly calls itself is call primitive recursive function or recursively defined function for example factorial function N!=N*(N-1)*(N-2)*…*3*2*1
   Calculating 5!
   5! =5*4!
   =5*4*3!
   =5*4*3*2!
   =5*4*3*2*1! (1!=Base criteria hit)
   We have to postpone the calculation of 5! till 4! Is found. The calculation of 4! Postpone till 3! is found. The calculation of 3! is postpone until we get the value of 2! and calculation of 2! is postpone under the until the calculation of 1!. This delay can be done using stack.
2. Non primitive recursive function:
   The function that indirectly calls itself is called non primitive recursive function or recursive use of function. E.g. Ackermann's function.
   A(m,n)=n+1, if m==0;
   A(m,n)=A(m-1,1), if n==0 but m!=0;
   A(m,n)= A(m-1, A(m,n-1)), if m!=0 and n!=0;

Advantages of recursion

- The main advantage is usually simplicity.
- Through recursion we can solve problems in easy way while it is iterative solution is huge and complex for example tower of Hanoi.
- You can reduce size of the code when you use a recursive call.

Disadvantage of recursion

- As the functions are called recursively, the system has to keep track return addresses and system has to keep track of parameters and variables of each recursive a call to the function. So recursive algorithm may require large amount of memory if the depth of the recursive is very large which is not required to be done in the case of non-recursive function call.

- Recursion cause system's unavoidable function call overhead so transformation from recursion to iteration can improve both speed and space requirement

Difference between linear and non-linear data structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next items	Every item is attached with many other items
Data is arranged in linear sequence	Data is not arranged in sequence
Data items can traverse in a single Run	Data cannot be traversed in a single run
Implementation is easy	Implementation is difficult
e.g array, stacks, linked list, queue	e.g. tree, graph

Stack

STACK:

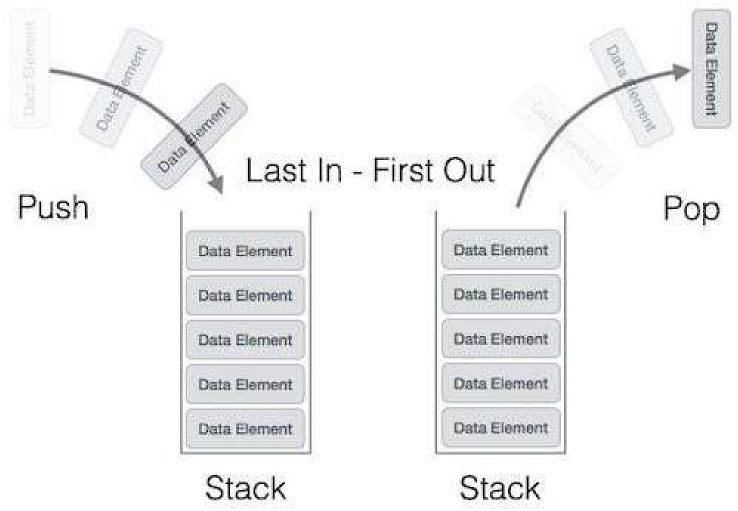
- Stack is a non-primitive linear data structure
- It is a data structure in which elements can be inserted and deletedfrom only one end.
- It is also considered as an ADT (abstract data type).
- The end from which insertion and deletion is done is known as "TOP"(top of stack).
- Since insertion and deletion are done from the single end, its elements are removed in reverse order from which they wereinserted.
- That means the last item added will be the first to be removed.
- Because of this characteristic it is known as LIFO (Last In First Out) list.
- It is also called Push Down list or Pile.
- Real life example or stack are:
    - Box containing books
    - Tray holder in cafeteria
    - Plates kept on one another

- Neatly folded

clothesImplementing:

Static Stack = Array

Dynamic stack= Linked list



Stack Operations:

Push : is inserting element in stack

Pop : Deleting element from stack

Change: for changing value of specific element from the top of stack

Peep : getting the value of specified elementfrom

the top of stack

An Algorithm to Insert an element into stack

The task (objective)

- Check for overflow
- Increment TOP
- Insert the element

PUSH(S,TOP,Item)

S: An array the represents the stack

Top: indicates the position of the top most element in the

stackItem: the value to be inserted

Max: Maximum number of elements

allowedStep 1: [Check for stack overflow]

```
            If TOP >=Max
                  Write ("stack Overflow")
                  Exit
            [End of if ]
Step 2:
 [Increment TOP]
            TOP=TOP+1
Step 3: [Insert Element]
            S[TOP]=Item
Step 4: [Finished]
            Exit/Return
```

An Algorithm to delete an element from stack

POP (S, TOP)

S: An array the represents the stack

TOP: indicates the position of the top most element in the stack
[Deletes 'item' from the 'stack', top is the number of elements currently in'stack'.]

```
Step 1 : [Check for
      Underflow ]if TOP
      = 0 then
```

```
            write (Stack Underflow on POP")
            Exit
            [End if structure]
   Step 2 : [Decrement
   Pointer]
            TOP:=TOP-1
   Step 3 : [Return top element of
            stack]return (S[TOP+1])
```

An Algorithm to Change the value of Ith element from top of the stack

The main tasks in this algorithm are:

- Check for underflow
- Update the desired

elementChange (S,TOP,ITEM,I)

S: Array representing stack

TOP: indicates position of top element in the stack.

I: Position of the element to be changed from TOP of the

stackITEM: New value of Ith element from top of the stack.

```
Step-1 : [Check for stack
    Underflow]If TOP -I +1<=0
    then
        Write ("Stack Underflow on change")
        Exit
Step-2: [Change Ith element from top of
    stack]S[TOP-I+1]:=ITEM
Step-3 : [Finished]
    Exit
```

References:
Data Structures by R. D. Morena,Priti Tailor, Vaishali Dindoliwala
An Introduction to Data Structures with applications, Trembley – Tata McGraw Hill.

Applications of stack

Stacks are frequently used to indicate the order of the processing of data

1. Converting infix to un-parenthesized (Polish-Prefix/Reverse Polish Postfix notation)
   a. Covert from infix to prefix
   b. Convert from infix to postfix
   c. Evaluate prefix
   d. Evaluate postfix
2. Recursion

Algorithm to convert Infix to Suffix Expression/Postfix/Reverse polish notation

Algorithm to convert Infix to Suffix Expression/Postfix/Reverse polish notation POSTFIX((Q,P)

STACK (S) : Stack to store operations

Q: Array Containing expression in infix notation(input) P : is Equivalent postfix notation (output)

Step-1: Push "(" on to stack S and add ")" to the end of Q

Step-2 : Scan Q from left to right and repeat step-3 to 6 for each element of Q until the stack is empty

Step-3: If an operand is encountered then add it to "P"

Step-4 : if "(" is encountered push it into stack

Step-5 : If an operator $\otimes$ is encountered then

a. Repeatedly pop from the stack and added to P each operator on the top of stack which has the same or higher precedence than operator $\otimes$
b. Add $\otimes$ to stack

Step 6: If ")" is encountered then

a. Repeatedly pop from the stack and add to P all the elements from the stack till "(" in stack is encountered

b. Remove "(" from stack

Step 7 [finish]

Algorithm to convert Infix to Prefix /Polish notation

Prefix(Q,P)

Q->array representing infix notation
P -> array for prefix

Stack-> array representing stack

Step1 Push ")" on to stack and add "(" to the beginning of Q

Step-2: Scan Q from right to left and repeat step-3 to 6 for each element of Q until the stack is empty

Step3: If an operand is encountered add to P Step-4: If

")" is found push it into stack

Step-5 : if operator $\otimes$ is encountered then

a. Repeatedly pop from stack and add to P each operator on the top stack which has the higher precedence then operator $\otimes$
b. Add $\otimes$ to stack

Step: 6 if "(" is encountered then

a. Repeatedly pop from stack and add to P each operator till the matching ")" is encountered from the stack
b. Remove ")" form the stack

Step-7 Reverse expression P

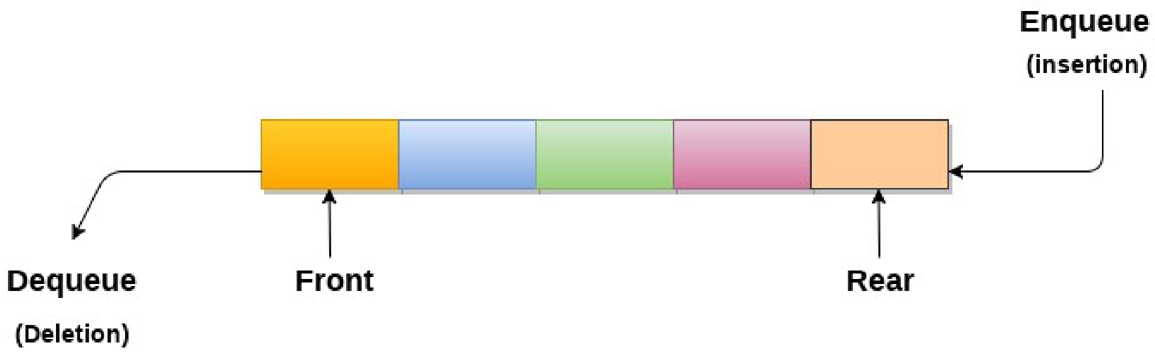
Step-8 [Finished]
Exit

Queue

A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

Queue is referred to be as First In First Out list.

For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

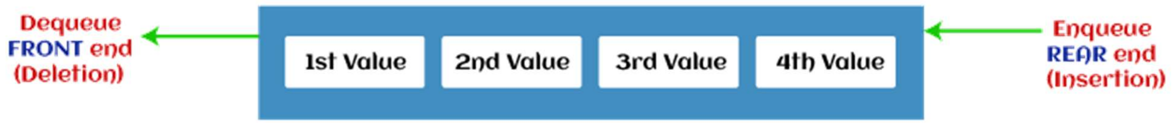
Types of Queue
There are four different types of queue that are listed as follows -

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

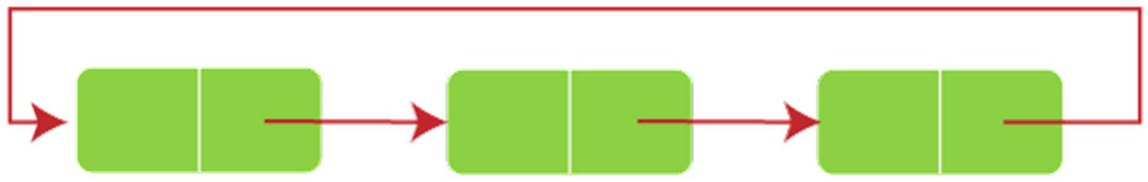
```
Step 1: IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
```

[END OF IF]

Step 2: EXIT

Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



Circular Queue

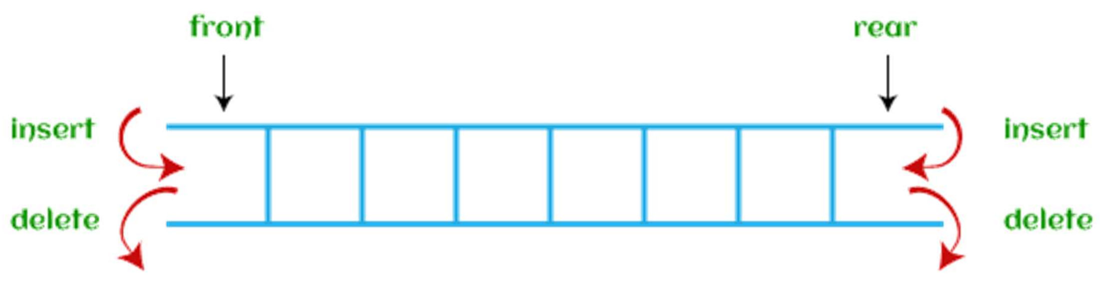
The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

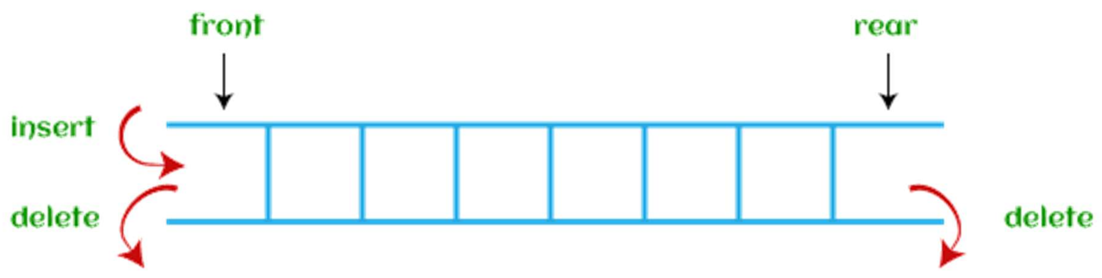
The representation of the deque is shown in the below image -



double ended queue

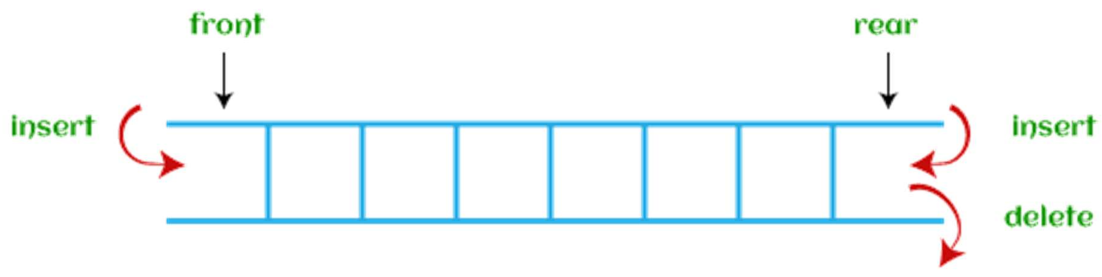
There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue.

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Now, let's see the ways to implement the queue.

Ways to implement the queue

There are two ways of implementing the Queue:

- **Implementation using array:** The sequential allocation in a Queue can be implemented using an array. For more details, click on the below link: https://www.javatpoint.com/array-representation-of-queue
- **Implementation using Linked list:** The linked list allocation in a Queue can be implemented using a linked list. For more details, click on the below link: https://www.javatpoint.com/linked-list-implementation-of-queue

Compare dynamic allocation and static allocation.

Static Memory Allocation	Dynamic Memory Allocation
Variables get allocated Permanently	Variable get allocation only if your program unit gets active
Allocation is done before program Execution	Allocation is done during program Execution

Faster execution compare to Dynamic	Faster execution compare to Dynamic
Less efficient	More efficient
There is no memory reusability	There is, memory reusability and memory can be freed when not Required
Memory can be wasted if allocated memory is not used e.g. int a[10]; if we store only 5 elements and 5 elements are wasted	Memory is not wasted

1. What do you mean by linear data structure?

   A data structure is said to be linear if its elements form a sequence or alinear list e.g. Array, Linked list, stack, queue

2. Operations that can be performed on data structure.

   The operations that can be performed on data structure are :

   - Traversal - Visit every part of data structure
   - Search – Traversal through the data structure for a given element
   - Insertion – Adding new element
   - Deletion – Removing element form ds
   - Sorting – Rearranging the elements
   - Merging – Combining two similar ds in one

3. What is the difference between int *p & int **p?

   int *p is a statement that declares p as pointer to integer. i.e. p is apointer type which can point to any variable of integer data type.

   Int **p is a statement that declares p as pointer to pointer. “p” points toa pointer that is pointing to variable of integer data type.

4. What do you mean by non-linear data structure? List out various advantages and disadvantages of non-linear data structure.

   A data structure in which a data item is connected to several other data items is known as non-linear data structure.

   Advantages:

   - Uses memory efficiently that free contiguous memory is not needed.
   - The length of the data items/number of data items is not necessary to be known prior to allocation

   Disadvantages:

   - Overhead of the link to the next data item.

   Non-linear data structure are:

   - Tree
   - Graph

Deque (or double-ended queue)

What is a queue?

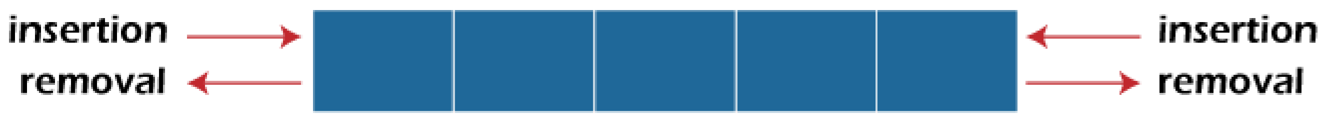
A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the **rear end** or the **tail,** whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the person enters last in the queue gets the ticket at last.

What is a Deque (or double-ended queue)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -



Representation of deque

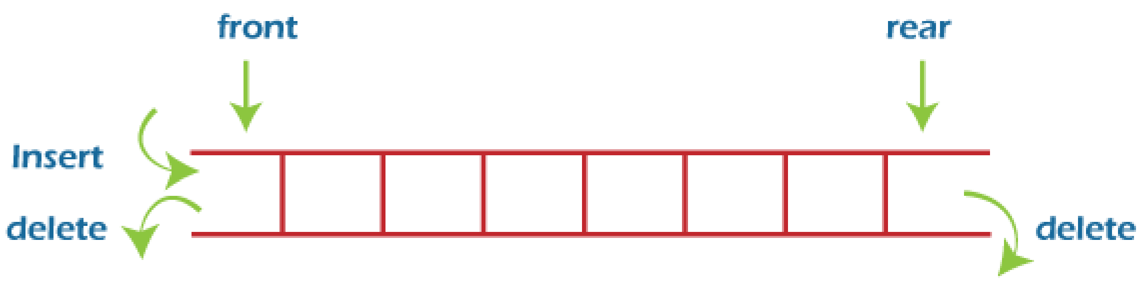
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

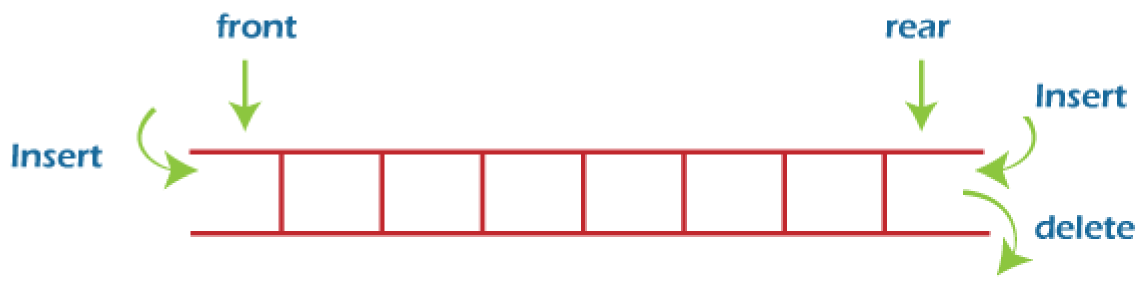
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

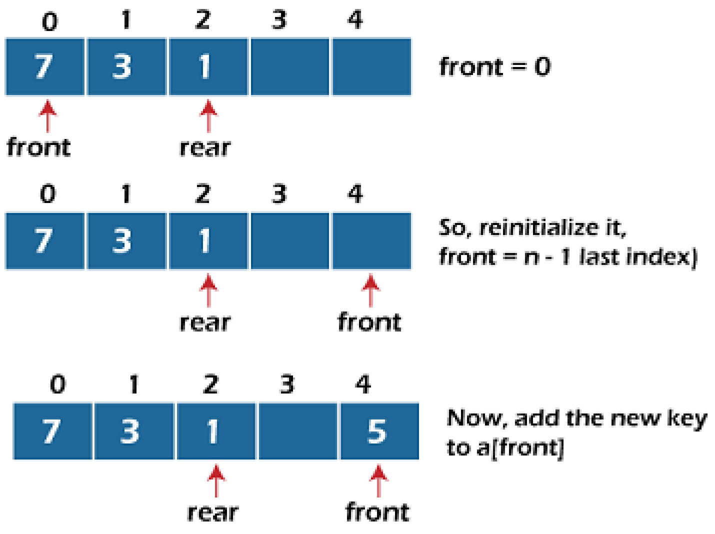
- Get the front item from the deque

- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

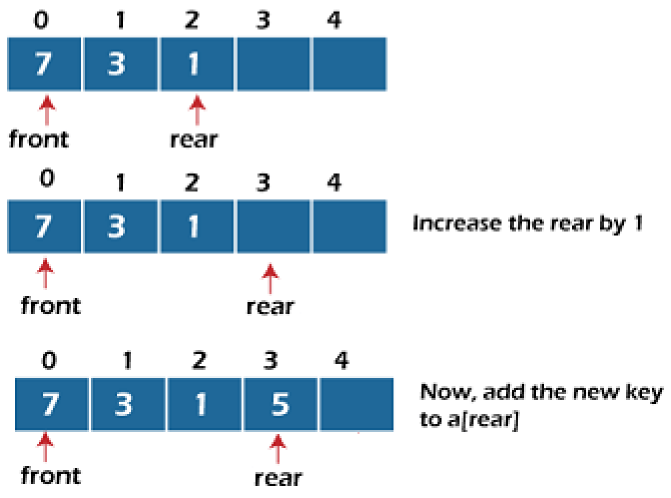
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

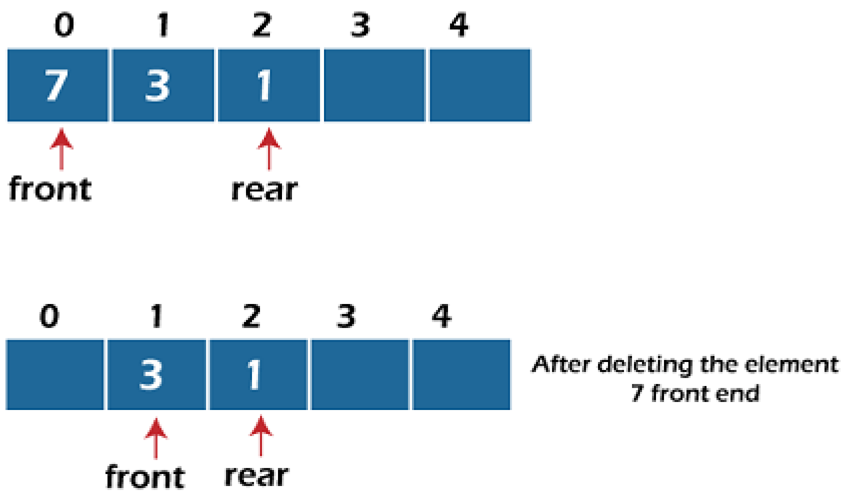
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

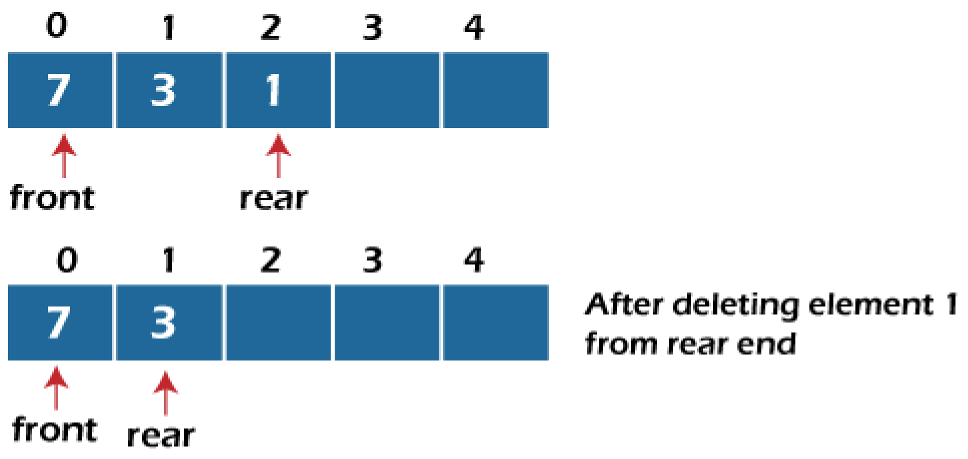
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

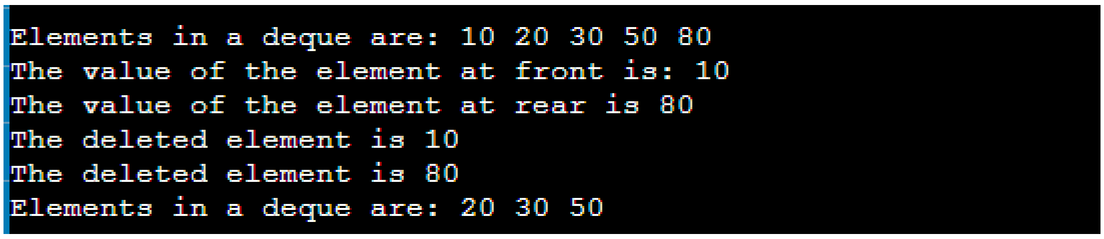
Implementation of deque

Now, let's see the implementation of deque in C programming language.

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
//  insert_front function will insert the value from the front
void insert_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }

}

// display function prints all the value of deque.
void display()
{
    int i=f;
    printf("\nElements in a deque are: ");

    while(i!=r)
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
     printf("%d",deque[r]);
}

// getfront function retrieves the first value of the deque.
void getfront()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the element at front is: %d", deque[f]);
    }

}

// getrear function retrieves the last value of the deque.
void getrear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the element at rear is %d", deque[r]);
    }

}

// delete_front() function deletes the element from the front
void delete_front()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=-1;
        r=-1;

    }
     else if(f==(size-1))
     {
         printf("\nThe deleted element is %d", deque[f]);
         f=0;
     }
     else
     {
          printf("\nThe deleted element is %d", deque[f]);
          f=f+1;
     }
}

// delete_rear() function deletes the element from the rear
void delete_rear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;

    }
     else if(r==0)
     {
         printf("\nThe deleted element is %d", deque[r]);
         r=size-1;
     }
     else
     {
          printf("\nThe deleted element is %d", deque[r]);
          r=r-1;
     }
}

int main()
{
    insert_front(20);
    insert_front(10);
    insert_rear(30);
    insert_rear(50);
    insert_rear(80);
    display();  // Calling the display function to retrieve the values of deque
    getfront();  // Retrieve the value at front-end
    getrear();  // Retrieve the value at rear-end
```

```
        delete_front();
        delete_rear();
        display(); // calling display function to retrieve values after deletion
        return 0;
    }
```

Output:



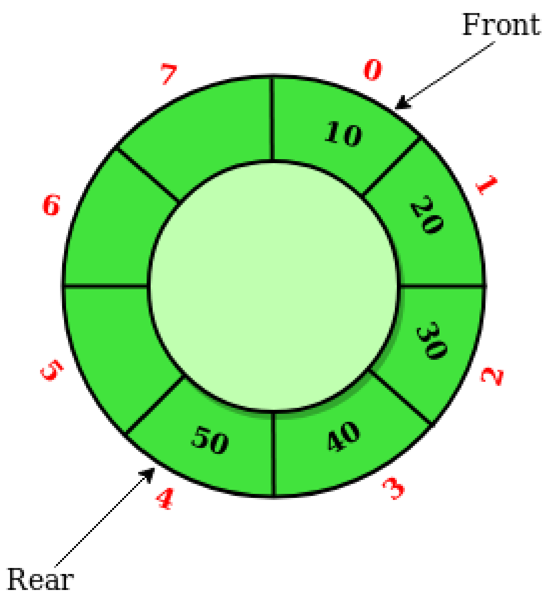
```
Elements in a deque are: 10 20 30 50 80
The value of the element at front is: 10
The value of the element at rear is 80
The deleted element is 10
The deleted element is 80
Elements in a deque are: 20 30 50
```

Introduction to Circular Queue

What is a Circular Queue?

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

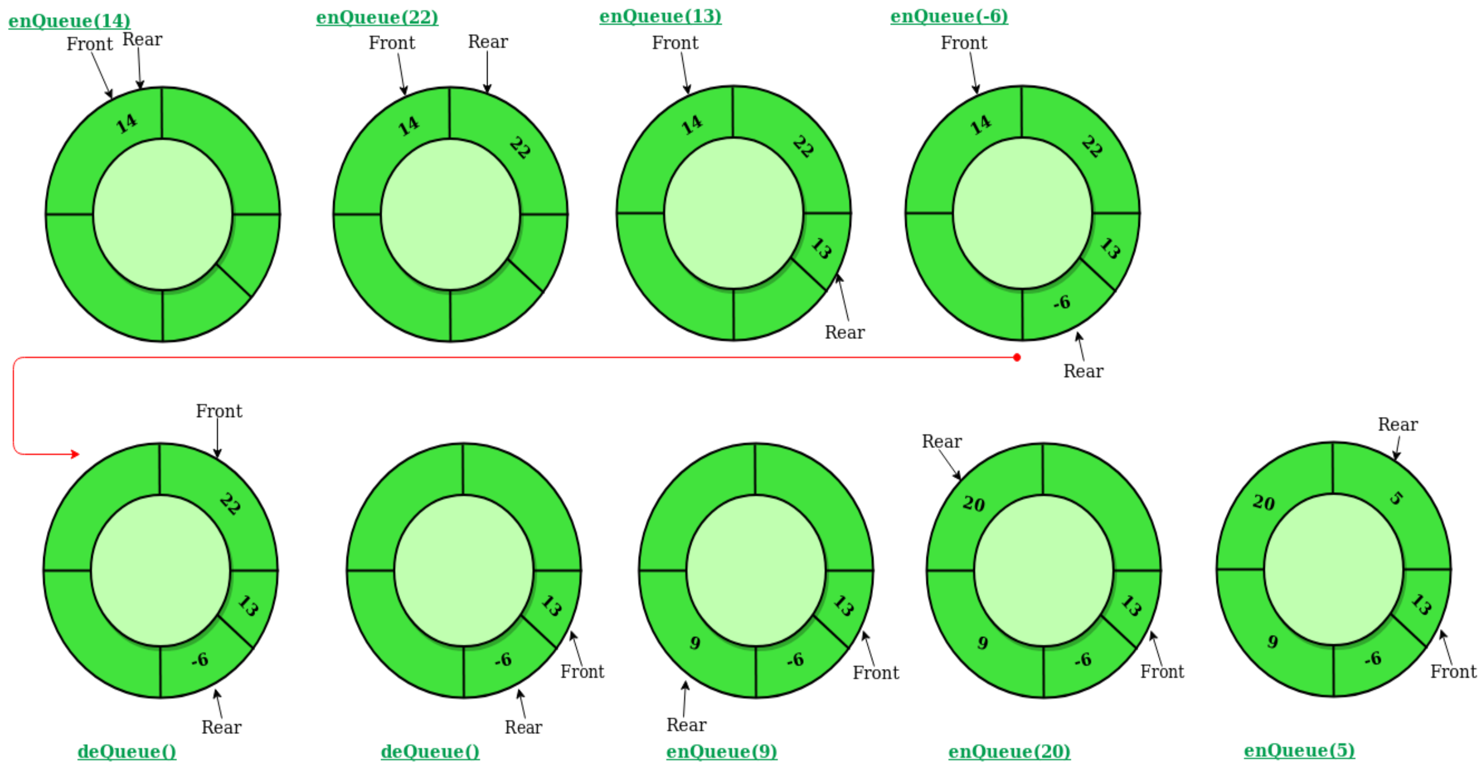
Operations on Circular Queue:

- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
    - Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
    - If it is full then display Queue is full.
        - If the queue is not full then, insert an element at the end of the queue.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
    - Check whether the queue is Empty.
    - If it is empty then display Queue is empty.

- If the queue is not empty, then get the last element and remove it from the queue.

Illustration of Circular Queue Operations:

Follow the below image for a better understanding of the enqueue and dequeue operations.



How to Implement a Circular Queue?

A circular queue can be implemented using two data structures:

- Array

Here we have shown the implementation of a circular queue using an array data structure.

Implement Circular Queue using Array:

1. Initialize an array queue of size **n**, where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.
3. **Enqueue:** To enqueue an element **x** into the queue, do the following:
    - Increment rear by 1.
        - If **rear** is equal to n, set **rear** to 0.
    - If **front** is -1, set **front** to 0.
    - Set queue[rear] to x.
4. **Dequeue:** To dequeue an element from the queue, do the following:
    - Check if the queue is empty by checking if **front** is -1.
        - If it is, return an error message indicating that the queue is empty.

- Set **x** to queue[front].
- If **front** is equal to **rear**, set **front** and **rear** to -1.
- Otherwise, increment **front** by 1 and if **front** is equal to n, set **front** to 0.
- Return x.

Complexity Analysis of Circular Queue Operations:

- **Time Complexity:**
  - Enqueue: O(1) because no loop is involved for a single enqueue.
  - Dequeue: O(1) because no loop is involved for one dequeue operation.
- **Auxiliary Space:** O(N) as the queue is of size N.

Applications of Circular Queue:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)
```

Circular Queue Operations

The circular queue work as follows:

- two pointers `FRONT` and `REAR`
- `FRONT` track the first element of the queue
- `REAR` track the last elements of the queue
- initially, set value of `FRONT` and `REAR` to -1

1. Enqueue Operation

- check if the queue is full
- for the first element, set value of `FRONT` to 0
- circularly increase the `REAR` index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by `REAR`

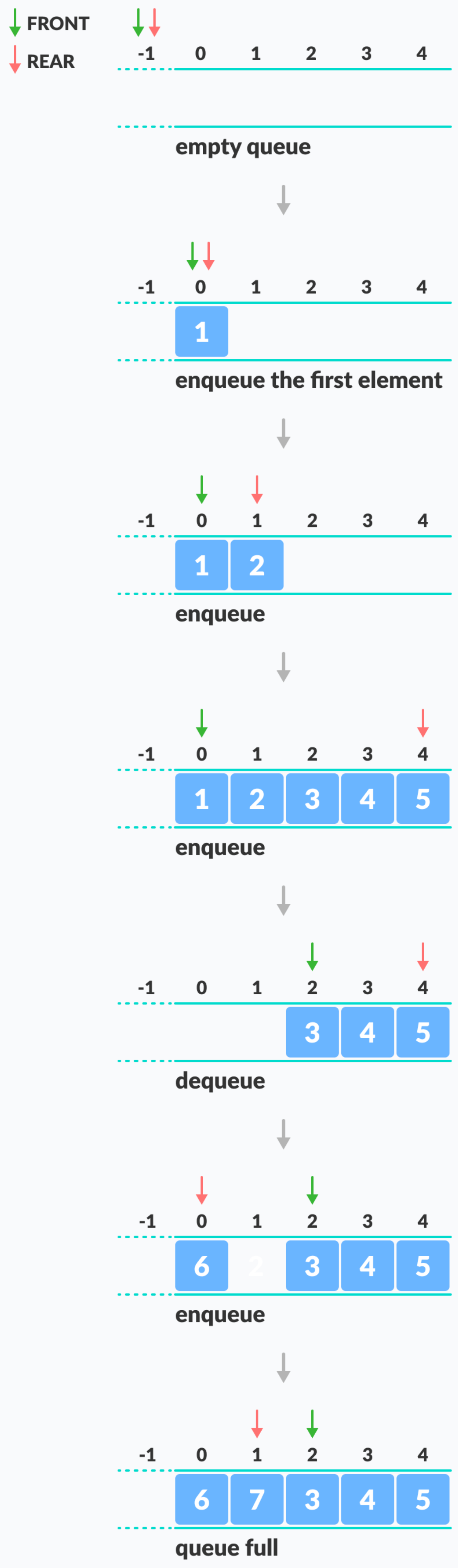
2. Dequeue Operation

- check if the queue is empty
- return the value pointed by `FRONT`
- circularly increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1

However, the check for full queue has a new additional case:

- Case 1: `FRONT` = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`

The second case happens when `REAR` starts from 0 due to circular increment and when its value is just 1 less than `FRONT`, the queue is full.



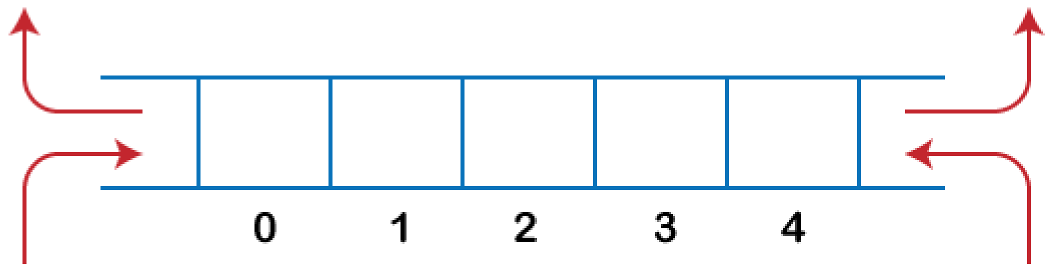
Enque and Deque Operations

// Circular Queue implementation in C++

```
#include <iostream>
#define SIZE 5 /* Size of Circular Queue */

using namespace std;

class Queue {
   private:
  int items[SIZE], front, rear;

   public:
  Queue() {
    front = -1;
    rear = -1;
  }
  // Check if the queue is full
  bool isFull() {
    if (front == 0 && rear == SIZE - 1) {
      return true;
    }
    if (front == rear + 1) {
      return true;
    }
    return false;
  }
  // Check if the queue is empty
  bool isEmpty() {
    if (front == -1)
      return true;
    else
      return false;
  }
  // Adding an element
  void enQueue(int element) {
    if (isFull()) {
      cout << "Queue is full";
    } else {
      if (front == -1) front = 0;
      rear = (rear + 1) % SIZE;
      items[rear] = element;
      cout << endl
         << "Inserted " << element << endl;
    }
  }
  // Removing an element
  int deQueue() {
    int element;
    if (isEmpty()) {
      cout << "Queue is empty" << endl;
      return (-1);
    } else {
      element = items[front];
      if (front == rear) {
        front = -1;
        rear = -1;
      }
      // Q has only one element,
      // so we reset the queue after deleting it.
      else {
        front = (front + 1) % SIZE;
      }
      return (element);
    }
  }

  void display() {
    // Function to display status of Circular Queue
    int i;
    if (isEmpty()) {
      cout << endl
         << "Empty Queue" << endl;
    } else {
      cout << "Front -> " << front;
      cout << endl
         << "Items -> ";
      for (i = front; i != rear; i = (i + 1) % SIZE)
        cout << items[i];
      cout << items[i];
      cout << endl
         << "Rear -> " << rear;
    }
  }
};

int main() {
  Queue q;

  // Fails because front = -1
  q.deQueue();

  q.enQueue(1);
  q.enQueue(2);
  q.enQueue(3);
  q.enQueue(4);
  q.enQueue(5);

  // Fails to enqueue because front == 0 && rear == SIZE - 1
  q.enQueue(6);

  q.display();

  int elem = q.deQueue();

  if (elem != -1)
    cout << endl
       << "Deleted Element is " << elem;
  q.display();


  q.enQueue(7);


  q.display();


  // Fails to enqueue because front == rear + 1
  q.enQueue(8);


  return 0;
}
```

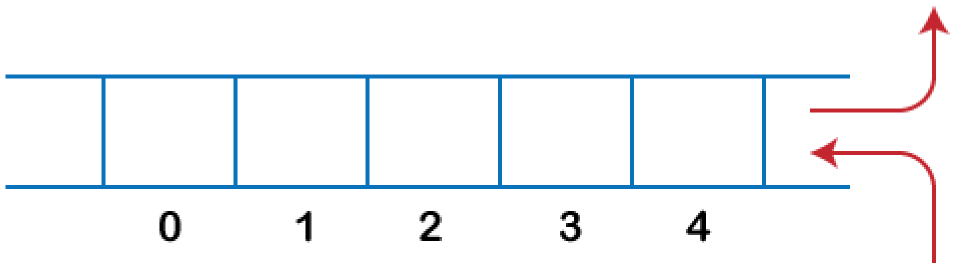
Dequeue in CPP

The dequeue stands for **Double Ended ;**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

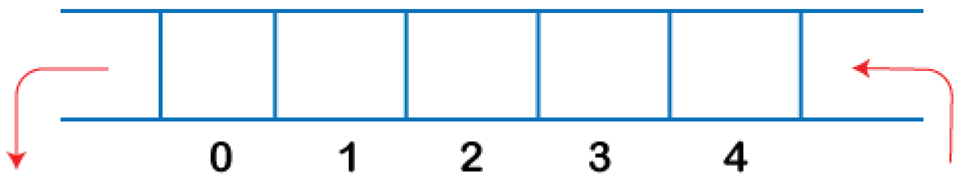


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

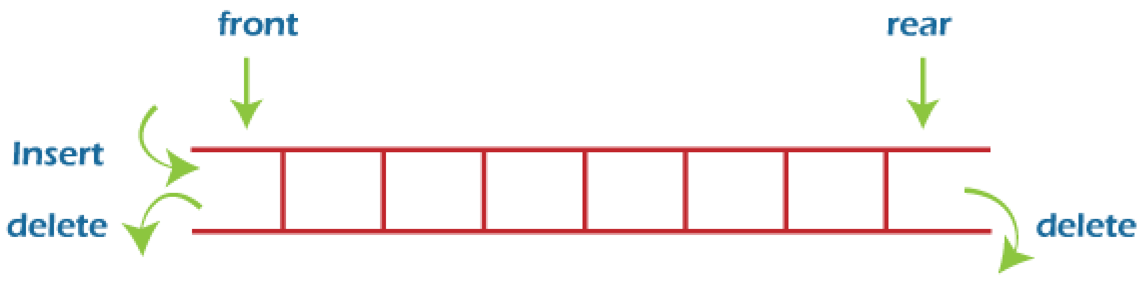


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.



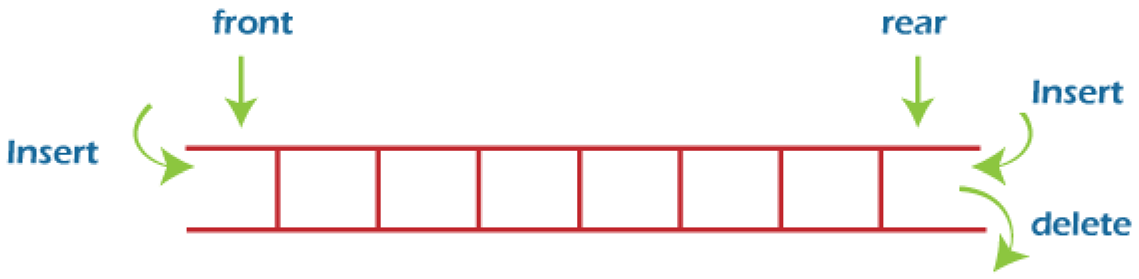
There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



input restricted double ended queue

2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Output restricted double ended queue

Operations on Deque

The following are the operations applied on deque:

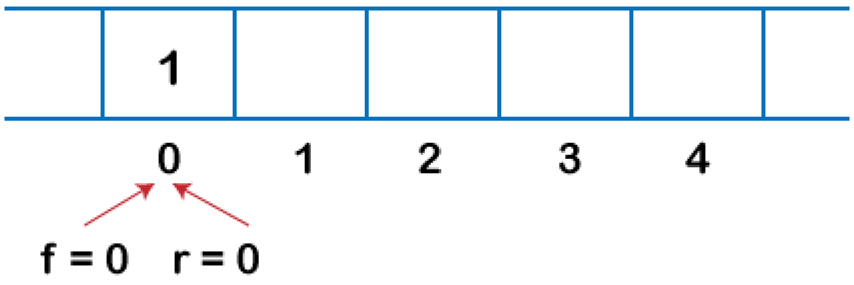
- **Insert at front**
- **Insert at rear**
- **Delete at front**
- **Delete from rear**

Implementation of Deque using a circular array

The following are the steps to perform the operations on the Deque:

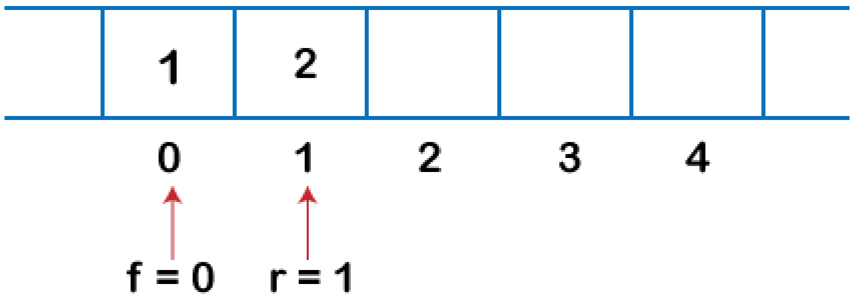
Enqueue operation

1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**

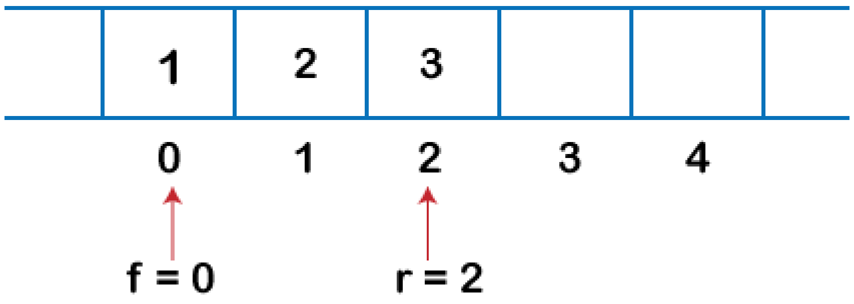


3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the

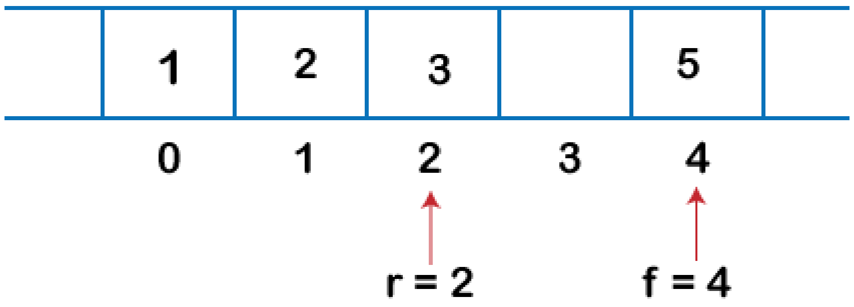
second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.

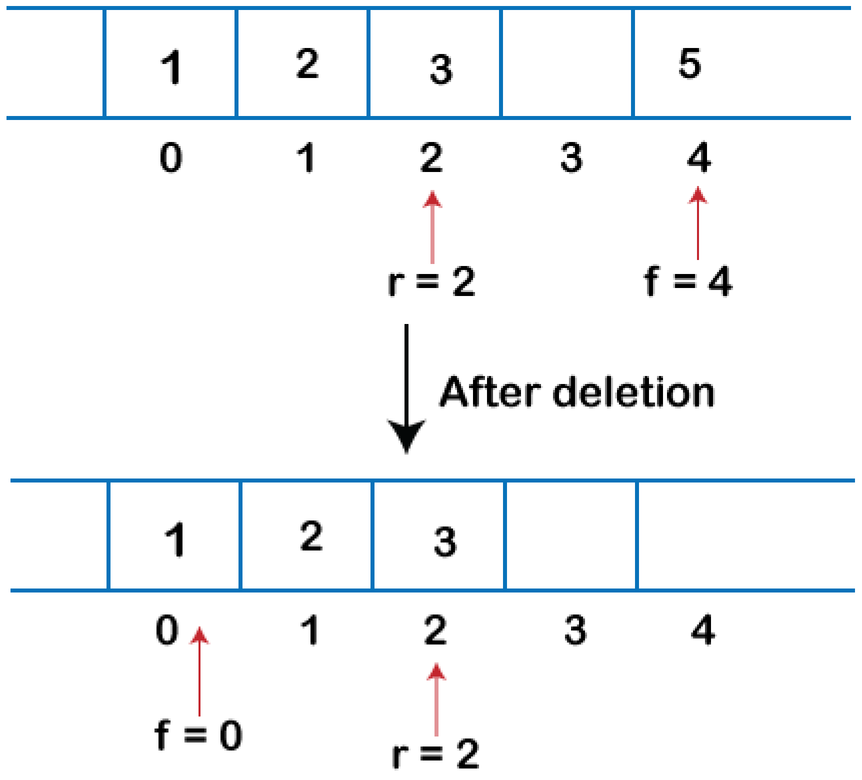


5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



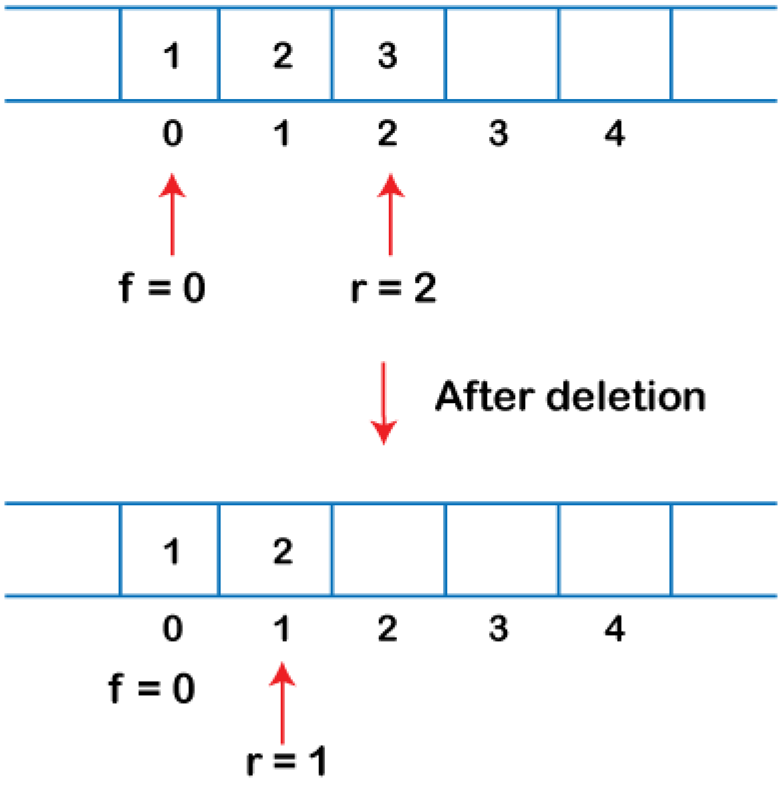
Dequeue Operation

1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last element, then front is set to 0 in case of delete operation.

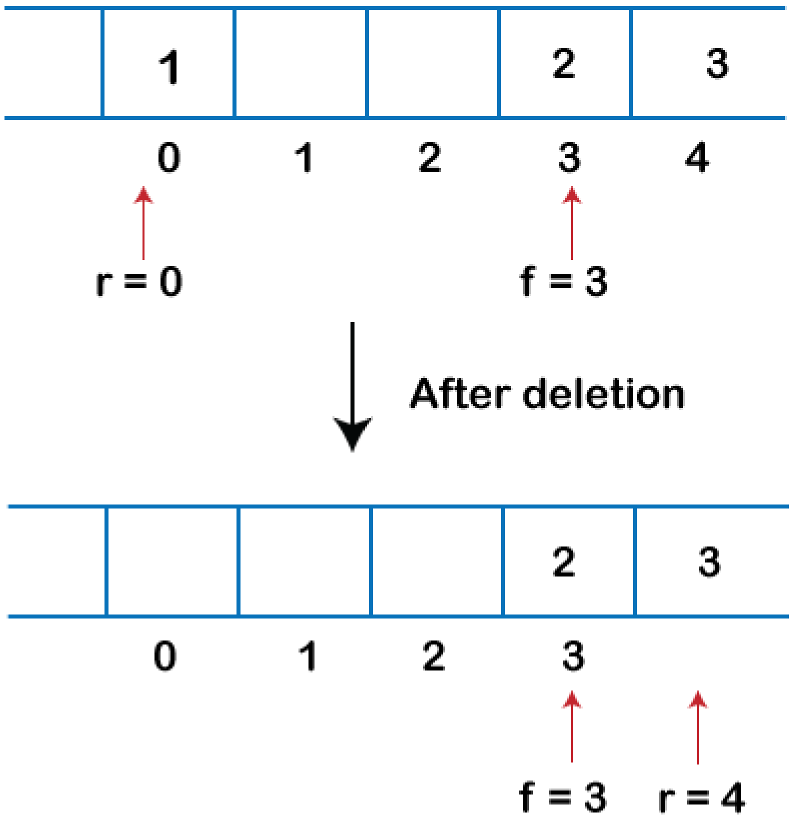


2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the

below figure:

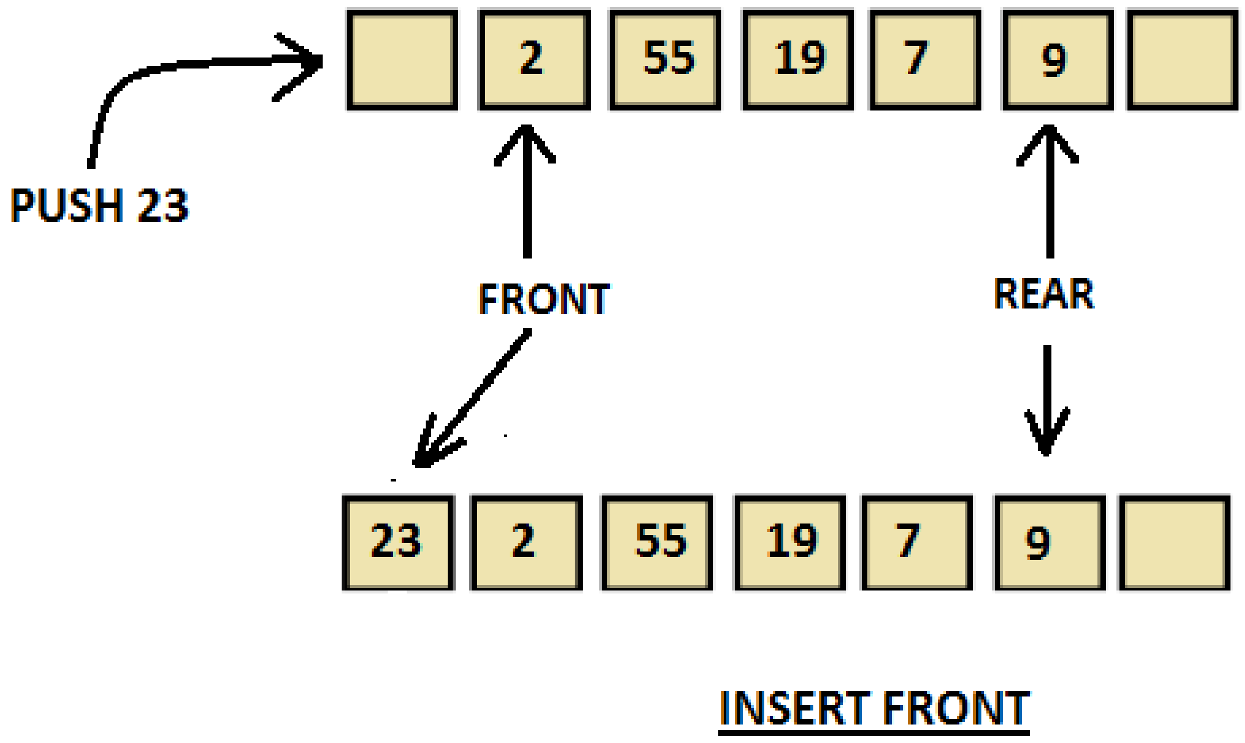


3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



Algorithm for Insertion at front end

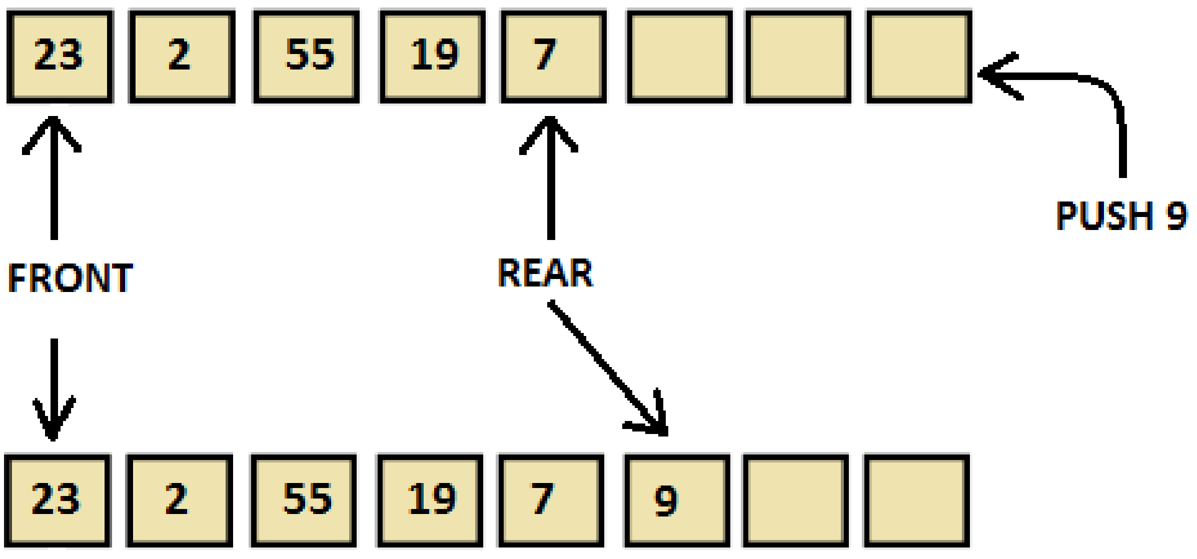
```
Step-1 :  [Check for the front position]
              if(f=0 && r=r-1 || f=r+1)
                      Print("Cannot add item at the front overflow”);
                      return;
Step-2 :  If f=-1 set f=r=0
           Else if f=0 set f=n-1
           Else f=f-1
Step-3  : [Insert at front]
           q[f]=data;
Step-4  : Return
```



Algorithm for Insertion at rear end

```
Step-1: [Check for overflow]
              if(rear==MAX)
```

```
                    Print("Queue is Overflow");
                    return;
Step-2:  [Insert Element]
                else
                    rear=rear+1;
                    q[rear]=no;
                [Set rear and front pointer]
                if rear=0
                    rear=1;
                if front=0
                    front=1;
Step-3: return
```

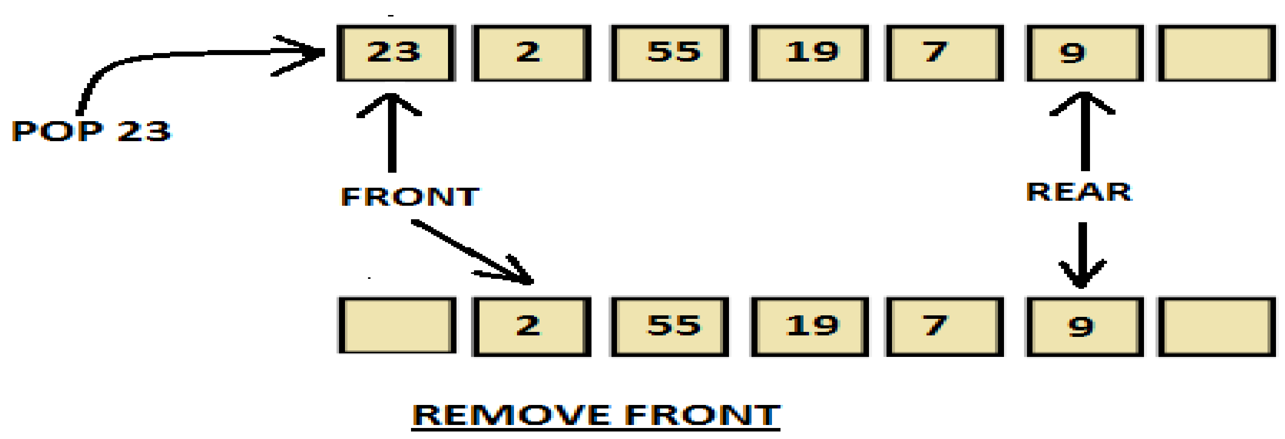


INSERT BACK

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
                if front=0
                    print(" Queue is Underflow");
                    return;
Step-2 [Perform deletion]
```

```
                else
                        no=q[front];
                        print("Deleted element is",no);
                [Set front and rear pointer]
                if front=rear
                        front=0;
                        rear=0;
                else
                        front=front+1;
Step-3 : Return
```



Algorithm for Deletion from rear end

```
Step-1 : [Check for the rear pointer]
                if rear=0
                        print("Cannot delete value at rear end");
                        return;
Step-2:  [ perform deletion]
                else
```

```
                  no=q[rear];
                  [Check for the front and rear pointer]
             if front= rear
                  front=0;
                  rear=0;
             else
                  rear=rear-1;
                  print("Deleted element is",no);
Step-3 : Return
```

Algorithm for input restricted dequeue.

```
Step1 [check for under flow condition]
          if front = -1 & rear = -1, then
          output underflow & exit
Step2 [delete element at the front end ]
          if [front] >= 0 then
          item =Q[front]
Step3 [check queue for empty]
          if front = rear, then
          rear = -1
          front = -1
        else
          front = front+1
Step4 [delete element at the rear end ]
          if rear > = 0
        item = Q[rear]
Step5 [check queue for empty ]
          if front = rear, then
```

```
                  front = -1
                  rear = -1
         else
                  rear = rear – 1
Step6   exit
```

Algorithm for output restricted dequeue

```
 Step1 [check for  overflow condition]
          If front  = 0 & rear >= size -1
           Output over flow & exit
Step2 :  [check front pointer value]
                  If front >0, then
                   Front = front -1
Step3:  [insert  element at the front end ]
                  Q[front] = value
Step4 :  [ check rear pointer value]
                  If rear < size-1
                  Rear  = rear+1
Step5:  [ insert element at the rear end]
                  Q[rear] = value
Step6 : exit
```

Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Question Bank

UNIT – 1

Short Questions:-

1) What are the Characteristics of OOPs?
2) Define Class and Object
3) Explain limitation of inline function.
4) List out the advantages of inline function.
5) What are the advantages of cin and cout compared to printf and scanf?
6) What is the use of scope resolution operator?
7) What is the use of 'this' pointer?
8) What is containership?
9) What is the use of 'new' operator?
10) What is reference variable?
11) What is application of scope resolution operator (::) in C++?
12) State the difference between ios::in and ios::out.
13) Differentiate between int *p and int **p.

Long Questions:-

1) What is header file? Explain various header file used in C++.
2) Write a short note on String functions with appropriate example.
3) Explain different data types in OOPs.
4) State the difference between C and C++.
5) Write a note on Procedural and Object oriented Programming.
6) What is object oriented programming ? Write the difference between OOP and POP.
7) Compare call by value and call by reference with appropriate example.
8) Explain array of objects with example.
9) Explain memory management operator. Discuss advantages of new over malloc.
10) Explain default argument and function prototyping with an example

UNIT – 2

Short Questions:-

1) What is Visibility modifier?
2) What is the use of ‘Protected’ over ‘Private’ modifier?
3) What is meant by Encapsulation?
4) Explain data hiding concept in classes.
5) What are the types of inheritance?
6) Differentiate between copy and parameterized constructor.
7) What is constructor with default argument?
8) What is the use of destructor? How is it created?
9) What is Object? Give one example of it.
10) What is an abstract class? What is use of it?
11) What is the meaning of "IS-A" and "HAS-A" relationship
12) What is Containership?

Long Questions:-

1) Explain Access modifier with example.
2) What is constructor? Explain parameterized constructor with an example.
3) What is constructor and destroyed in C++? How are they defined and when are they used? Illustrate with an example for each.
4) What is destructor? How it is written in C++? When it is called? Differentiate between constructor and destructor with proper example.
5) Explain Inheritance and types of inheritance with examples.
6) What is the purpose of inheritance? Explain multilevel inheritance with example.
7) Explain data abstraction and encapsulation.
8) What is containership? How does it differ from Inheritance? Give example.
9) How to remove ambiguity occurred in the case of hybrid inheritance?
10) Write a program to create a class student stores the roll_no, class test stores the mark obtained in two subjects and class result contains the total marks obtained the test. The class result can inherit the details of the marks obtained in the test and the roll_no of stusent class.
11) Create an abstract class “Shape” which stores data members like length, breadth and radius, Create two classes “Circle” and “Rectangle” which stores data members like area respectively. Write a function to calculate area and display it.

UNIT – 3

Short Questions:-

1) What is polymorphism?
2) Explain use of late binding.
3) What is static binding?
4) What do you mean by function overloading?
5) What is function overriding?
6) Explain pure virtual function.
7) Which function in C++ is defined to do nothing? Explain it.
8) List the operators which cannot be overloaded .
9) What are the rules for the Unary operator overloading using friend?

Long Questions:-

1) Write a note on Polymorphism.
2) What is dynamic binding? Explain it with proper example.
3) What is polymorphism? Explain different types of polymorphism in brief.
4) How runtime polymorphism is achieved in C++? Explain with example.
5) Explain type conversion with example.
6) What is type conversion? Explain explicit type conversion with example.
7) Explain operator overloading with example.
8) Differentiate between overloading and overriding. Explain the concept of overriding with example.
9) What is friend function? Why we need to write friend function? Explain with example. Discuss its advantages.
10) Explain pure virtual function? When it is necessary? Explain rules of pure virtual function
11) Explain virtual base class. How it is differ from virtual function. Explain with proper example.
12) Write a program to overload >> and << operators
13) Write a program to overload == and += operator (String object)
14) Write + operator to concate two string. Use = operator to copy one string to another string.

UNIT – 4

Short Questions:-

1) What is Data structure? List various data structures.
2) Differentiate Linear and non linear data structure.
3) What is stack? How it is differ from array?
4) Write application of stack.
5) What is Recursion? Define recursive function.
6) List out applications of stack.
7) What is Top pointer in stack?
8) Evaluate Postfix expression : (i) 2, 3, 4, *, + (ii) 3, 4, *, 2, 5, *, + (iii) 5, 6, 2, +, *, 12, 4, /, - (iv) 3, 2, 4, - , +
9) Convert from Infix to Postfix : (i) A+ (B * C – D / E * G) + H

Long Questions:-

1) What is linear data structure? Discuss difference between FIFO and LIFO concept.
2) What is stack? Write algorithm for push and pop operation of Stack.
3) What is stack? Write a program to perform Push, Pop, Display operation.
4) Explain Tower of Hanoi as application of stack.
5) What is recursion? Write an algorithm to find factorial number.
6) Write an algorithm to convert infix expression to prefix.
7) Write an algorithm to convert infix expression to postfix.
8) Convert following expression into postfix
   (i) (A + B) * (C – D / E) * G + H
   (ii) (A + B / C) * (D + E) / F
   (iii) A / (B – C + D) * E + F ^ G

UNIT – 5

Short Questions:-

1) What is Queue? List out types of queues.
2) Define D-queue with an example.
3) What are the advantages of circular queue?
4) What are the properties of circular queue?
5) Discuss the real world example of Queue.
6) What will be the position of front and rear if circular queue is empty?
7) Write an overflow and underflow conditions of input restricted Dqueue.
8) Write an overflow and underflow conditions of output restricted Dqueue.

Long Questions:-

1) What is Queue? List out types of queue. Write an algorithm to insert and delete elements in a queue.
2) What is D-queue? Write an algorithm to insert and delete elements from output restricted D-queue?
3) What is D-queue? Explain difference between input restricted and output restricted D-queue?
4) Write an algorithm to perform insert and delete operations on circular queue.
5) Define circular queue. Discuss advantages of circular queue over simple queue. Write algorithm to insert and delete element in circular queue.