Unit -1
Difference between OOP and POP

OOP:
Related with the real life objects and their properties. OOP Concepts:
1. Class and Objects
2. Data abstraction
3. Encapsulation
4. Polymorphism
5. Inheritance

POP:
Related with the conventional style. This approach is also known as the top-down approach. In this approach, a program is divided into functions that perform specific tasks. This approach is mainly used for medium-sized applications. Data is global, and all the functions can access global data. The basic drawback of the procedural programming approach is that data is not secured because data is global and can be accessed by any function. Program control flow is achieved through function calls and go to statements.

Difference between OOP and POP:

OOP	POP
Object oriented.	Structure oriented.
Program is divided into objects.	Program is divided into functions.
Bottom-up approach.	Top-down approach.
Inheritance property is used.	Inheritance is not allowed.
It uses access specifier.	It doesn't use access specifier.
Encapsulation is used to hide the data.	No data hiding.
Concept of virtual function.	No virtual function.

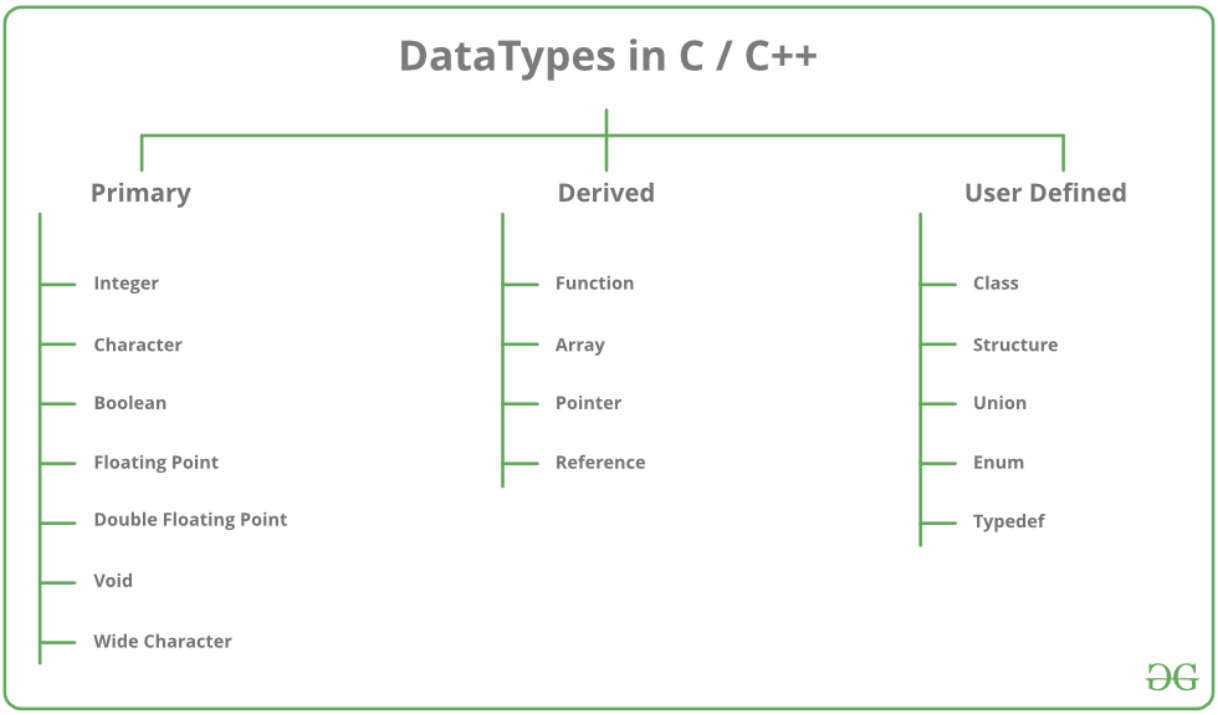
OOP	POP
C++, Java.	C, Pascal.

Standard Header Files And Their Uses:

1. **#include<stdio.h>:** It is used to perform input and output operations using functions **scanf()** and **printf()**.
2. **#include<iostream>:** It is used as a stream of Input and Output using cin and cout.
3. **#include<string.h>:** It is used to perform various functionalities related to string manupulation like strlen(), strcmp(), strcpy(), size(), etc.
4. **#include<math.h>:** It is used to perform mathematical operations like sqrt(), log2(), pow(), etc.
5. **#include<iomanip.h>:** It is used to access set() and setprecision() function to limit the decimal places in variables.
6. **#include<signal.h>:** It is used to perform signal handling functions like **signal()** and **raise()**.
7. **#include<stdarg.h>:**It is used to perform standard argument functions like **va_start()** and **va_arg()**. It is also used to indicate start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.
8. **#include<errno.h>:** It is used to perform error handling operations like **errno()**, **strerror()**, **perror()**, etc.
9. **#include<fstream.h>:** It is used to control the data to read from a file as an input and data to write into the file as an output.
10. **#include<time.h>:** It is used to perform functions related to date() and time() like setdate() and getdate(). It is also used to modify the system date and get the CPU time respectively.
11. **#include<float.h>:** It contains a set of various platform-dependent constants related to floating point values. These constants are proposed by ANSI C. They allow making programs more portable. Some examples of constants included in this header file are- e(exponent), b(base/radix), etc.
12. **#include<limits.h>:** It determines various properties of the various variable types. The macros defined in this header, limits the values of various variable types like **char**, **int**, and **long**. These limits specify that a variable cannot store any value beyond these limits, for example an unsigned character can store up to a maximum value of **255**.

C++ Data Types

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it 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. Every data type requires a different amount of memory.



Data types in C++ is mainly divided into three types:

1. **Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:
   - Integer
   - Character
   - Boolean
   - Floating Point
   - Double Floating Point
   - Valueless or Void
   - Wide Character
2. **Derived Data Types:** The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
   - Function
   - Array

- Pointer
- Reference

3. **Abstract or User-Defined Data Types**: These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
   - Class
   - Structure
   - Union
   - Enumeration
   - Typedef defined DataType

This article discusses **primitive data types** available in C++.

- **Integer**: Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.

- **Character**: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.

- **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. Keyword used for boolean data type is **bool**.

- **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.

- **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.

- **void**: Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.

- **Wide Character**: Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar_t**. It is generally 2 or 4 bytes long.

Concepts of String:

C++ provides following two types of string representations −

- The C-style character string.
- The string class type introduced with Standard 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**.

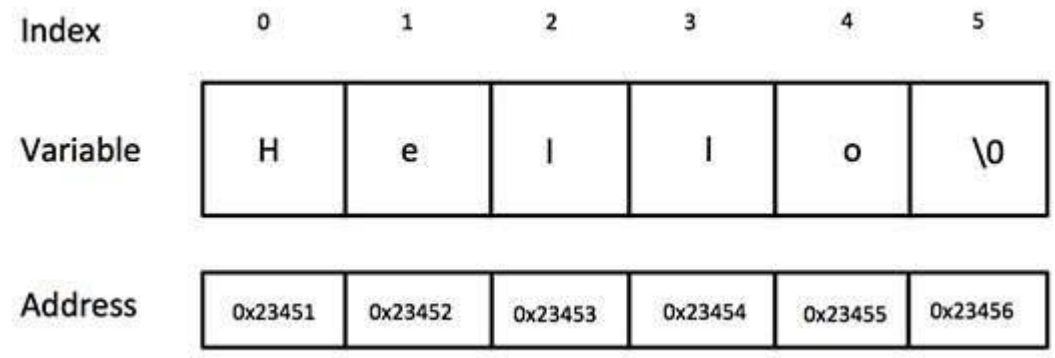
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
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 −

```
#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
```

C++ supports a wide range of functions that manipulate null-terminated strings −

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	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

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

```
#include <iostream>
#include <cstring>

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
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

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

using namespace std;

int main () {

   string str1 = "Hello";
   string str2 = "World";
   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
```

C++ Classes and Objects

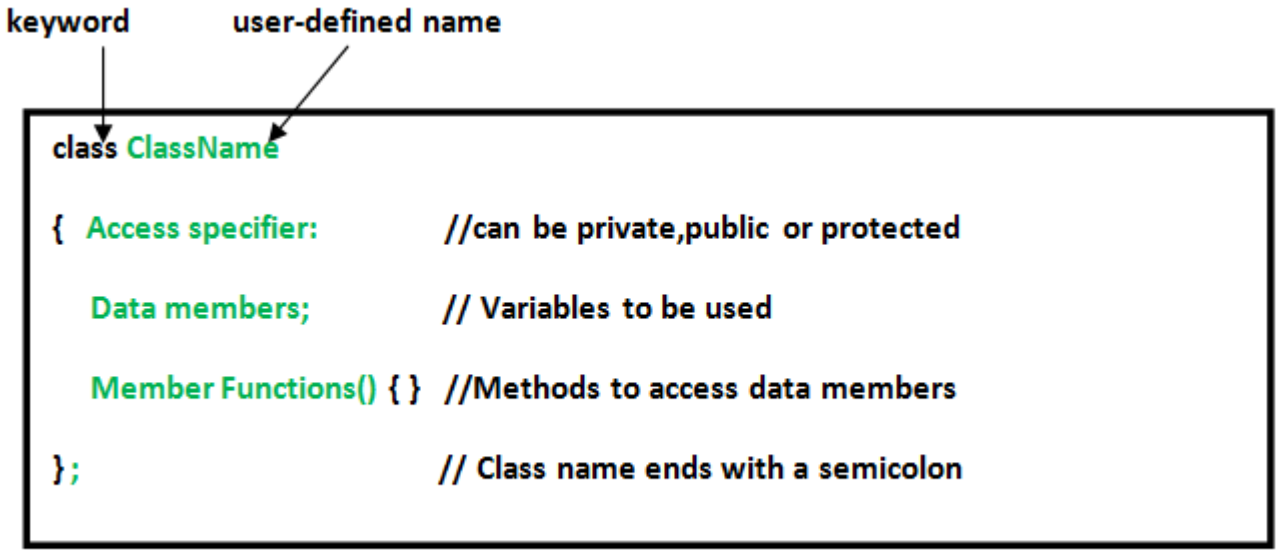
Class: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

```
ClassName ObjectName;
```

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.
This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

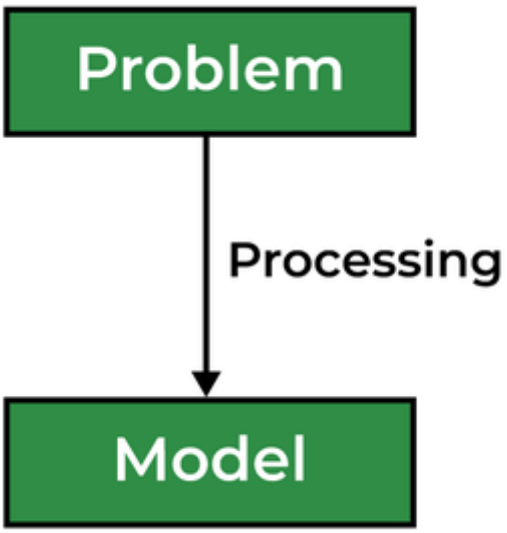
To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

Abstraction in C++

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.
Types of Abstraction:

1. **Data abstraction** – This type only shows the required information about the data and hides the unnecessary data.
2. **Control Abstraction** – This type only shows the required information about the implementation and hides unnecessary information.



Abstraction using Classes
We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files
One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Example:

```
// C++ Program to Demonstrate the
// working of Abstraction
#include <iostream>
using namespace std;

class implementAbstraction {
private:
    int a, b;
public:
    // method to set values of
    // private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};
```

```
int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Output

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

You can see in the above program we are not allowed to access the variables a and b directly, however, one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

Difference between Abstraction and Encapsulation

The following table highlights all the important differences between abstraction and encapsulation –

S.No	Abstraction	Encapsulation
1.	It is the process of gaining information.	It is a method that helps wrap up data into a single module.
2.	The problems in this technique are solved at the interface level.	Problems in encapsulation are solved at the implementation level.
3.	It helps hide the unwanted details/information.	It helps hide data using a single entity, or using a unit with the help of method that helps protect the information.

4.	It can be implemented using abstract classes and interfaces.	It can be implemented using access modifiers like public, private and protected.
5.	The complexities of the implementation are hidden using interface and abstract class.	The data is hidden using methods such as getters and setters.
6.	Abstraction can be performed using objects that are encapsulated within a single module.	Objects in encapsulation don't need to be in abstraction.

Access Modifiers in C++

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as **Data Hiding**. Consider a real-life example:

Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be 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 of these access modifiers in detail:
1. Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions 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.
Example:

```
// access modifier

#include<iostream>
using namespace std;

// class definition
class Circle
{
    public:
        double radius;

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

};

// main function
int main()
{
    Circle obj;
```

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

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is: 5.5

Area is: 94.985
```

In the above program, the data member *radius* is declared as public so it could be accessed outside the class and thus was allowed access from inside main().

2. Private: The class members declared as *private* can be accessed only by the member 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 the class.

Example:

- CPP

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

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
```

```
int 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();
    return 0;
}
```

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;
```

The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class. Yet an access to obj.radius is attempted, but radius being a private data member, we obtained the above compilation error.

However, we can access the private data members of a class indirectly using the public member functions of the class.

Example:

- CPP

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;
    // public member function
    public:
```

```
        void compute_area(double r)
        {   // member function can access private
            // data member radius
            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);
    return 0;
}
```

Output:

```
Radius is: 1.5

Area is: 7.065
```

3. Protected: The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of Inheritance.

Example:

```cpp
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

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

};

// sub class or derived class from public base 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
int main() {

    Child obj1;

    // member function of the derived class can
    // access the protected data members of the base class
```

```
    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

Output:

```
id_protected is: 81
```

Constructor & Destructor in CPP

Constructor in C++ is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

- Constructor is a member function of a class, whose name is same as the class name.
- Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically, when an object of the same class is created.
- Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.
- Constructor do not return value, hence they do not have a return type.

```
The prototype of the constructor looks like
    <class-name> (list-of-parameters);


Constructor can be defined inside the class declaration or outside the class
declaration


a.    Syntax for defining the constructor within the class


        <class-name>(list-of-parameters)
        {
                  //constructor definition
        }


b.    Syntax for defining the constructor outside the class


        <class-name>: :<class-name>(list-of-parameters)
        {
                //constructor definition
        }
```

- C++

```
// Example: defining the constructor within the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }



    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;  //constructor gets called automatically when we create the object of the
class
    s.display();
    return 0;

}
```

- C++

```
// Example: defining the constructor outside the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
```

```
    double fee;
    public:
    student();
    void display();

};

    student::student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }

int main()
{
    student s;
    s.display();
    return 0;
}
```

Characteristics of constructor

- The name of the constructor is same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.

Types of constructor

- Default constructor
- Parameterized constructor
- Overloaded constructor
- Constructor with default value

- Copy constructor
- Inline constructor

Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

`<class-name> (list-of-parameters);`

Constructors can be defined inside or outside the class declaration:-

The syntax for defining the constructor within the class:

`<class-name> (list-of-parameters) { // constructor definition }`

The syntax for defining the constructor outside the class:

`<class-name>: :<class-name> (list-of-parameters){ // constructor definition}`

Example

- C++

```
// defining the constructor within the class

#include <iostream>
using namespace std;

class student {
    int rno;
    char name[10];
    double fee;

public:
    student()
    {
        cout << "Enter the RollNo:";
        cin >> rno;
        cout << "Enter the Name:";
        cin >> name;
        cout << "Enter the Fee:";
        cin >> fee;
    }

    void display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }
};

int main()
{
    student s; // constructor gets called automatically when
```

```
                // we create the object of the class
    s.display();

    return 0;
}
```

Output

```
Enter the RollNo:Enter the Name:Enter the Fee:

0        6.95303e-310
```

Example

- C++

```
// defining the constructor outside the class

#include <iostream>
using namespace std;
class student {
    int rno;
    char name[50];
    double fee;

public:
    student();
    void display();
};

student::student()
{
    cout << "Enter the RollNo:";
    cin >> rno;

    cout << "Enter the Name:";
    cin >> name;

    cout << "Enter the Fee:";
    cin >> fee;
}

void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
    student s;
    s.display();
```

```
    return 0;
}
```

Output:

```
Enter the RollNo: 30

Enter the Name: ram

Enter the Fee: 20000

30 ram 20000
```

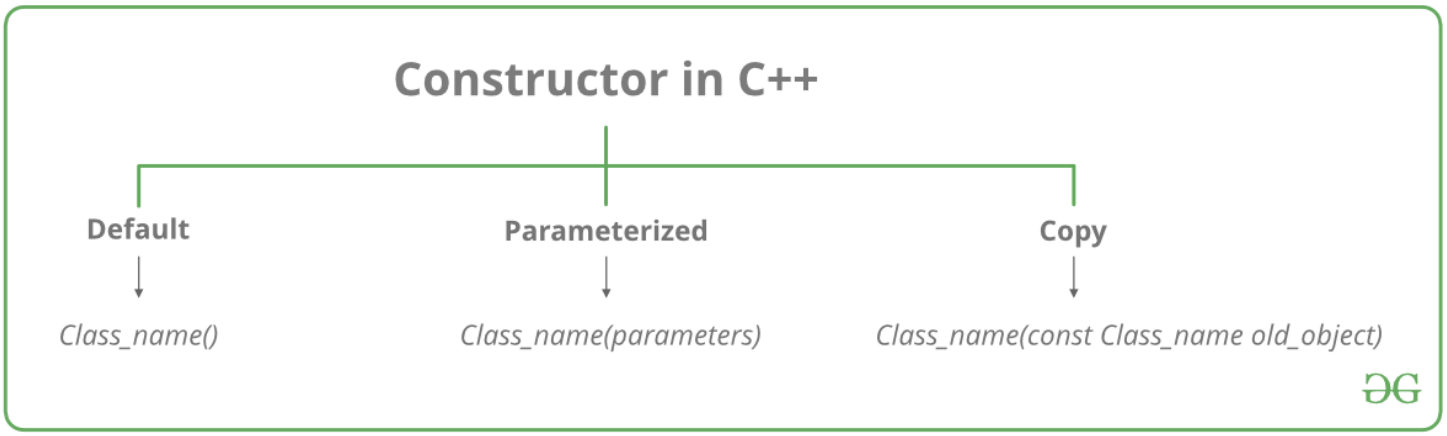
How constructors are different from a normal member function?

- C++

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

class Line {
  public:
     void setLength( double len );
     double getLength( void );
     Line( double len ); //This is the constructor
  private:
     double length;
};
//Member function definition including constructor
Line::Line( double len ) {
  cout<<"Object is being created , length ="<< len <<endl;
  length = len;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}
//Main function for the program
int main() {
  Line line(10.0);
  //get initially set length
  cout<<"Length of line :" << line.getLength() << endl;
  //set line length again
  line.setLength(6.0);
  cout<<"Length of line :" << line.getLength() << endl;

  return 0;
}
```

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Default Constructors don't have input argument however, Copy and Parameterized Constructors have input arguments
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).



Characteristics of the constructor:

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.
- Constructor cannot be inherited.
- Addresses of Constructor cannot be referred.
- Constructor make implicit calls to **new** and **delete** operators during memory allocation.

Types of Constructors

1. Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

- CPP

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
```

```
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 1;
}
```

Output

```
a: 10

b: 20
```

Note: *Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.*

- C++

```
// Example
#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student()                    //  Explicit Default constructor
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
```

```
    }

    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;
    s.display();
    return 0;
}
```

2. Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.
Note: *when the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as*
`Student s;`

`Will flash an error`

- CPP

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX() { return x; }
    int getY() { return y; }
};
```

```
int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();

    return 0;
}
```

Output

```
p1.x = 10, p1.y = 15
```

- C++

```
// Example

#include<iostream>
#include<string.h>
using namespace std;

class student
{
    int rno;
    char name[50];
    double fee;

      public:
    student(int,char[],double);
    void display();

};

student::student(int no,char n[],double f)
{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
```

```
{
    student s(1001,"Ram",10000);
    s.display();
    return 0;
}
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call


Example e(0, 50);           // Implicit call
```

- **Uses of Parameterized constructor:**
    1. It is used to initialize the various data elements of different objects with different values when they are created.
    2. It is used to overload constructors.
- **Can we have more than one constructor in a class?**
  Yes, It is called Constructor Overloading.

3. Copy Constructor:

A copy constructor is a member function that initializes an object using another object of the same class.
Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
          {
                    id=t.id;
          }
```

- CPP

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

class point {
private:
    double x, y;
```

```
public:
    // Non-default Constructor &
    // default Constructor
    point(double px, double py) { x = px, y = py; }
};

int main(void)
{

    // Define an array of size
    // 10 & of type point
    // This line will cause error
    point a[10];

    // Remove above line and program
    // will compile without error
    point b = point(5, 6);
}
```

Output:

```
Error: point (double px, double py): expects 2 arguments, 0 provided
```

- C++

```
// Implicit copy constructor

#include<iostream>
using namespace std;

class Sample
{          int id;
    public:
    void init(int x)
    {
        id=x;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;
```

```
    obj2.display();
    return 0;
}
```

Output

```
ID=10

ID=10
```

- C++

```
// Example: Explicit copy constructor

#include <iostream>
using namespace std;

class Sample
{
    int id;
    public:
    void init(int x)
    {
        id=x;
    }
    Sample(){}  //default constructor with empty body

    Sample(Sample &t)   //copy constructor
    {
        id=t.id;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};
int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;    copy constructor called
    obj2.display();
    return 0;
}
```

Output

```
ID=10

ID=10
```

- C++

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student(int,char[],double);
    student(student &t)       //copy constructor
    {
        rno=t.rno;
        strcpy(name,t.name);
        fee=t.fee;
    }
    void display();

};




    student::student(int no,char n[],double f)
    {
        rno=no;
        strcpy(name,n);
        fee=f;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }

int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s);   //copy constructor called
    manjeet.display();

    return 0;
}
```

- C++

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student(int,char[],double);
    student(student &t)       //copy constructor (member wise initialization)
    {
        rno=t.rno;
        strcpy(name,t.name);

    }
    void display();
    void disp()
    {
        cout<<endl<<rno<<"\t"<<name;
    }

};
    student::student(int no, char n[],double f)
    {
        rno=no;
        strcpy(name,n);
        fee=f;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }



int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s);   //copy constructor called
    manjeet.disp();

    return 0;
}
```

Destructor:

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope. Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

The syntax for defining the destructor within the class

```
~ <class-name>()
{
  }
```

The syntax for defining the destructor outside the class

```
<class-name>: : ~ <class-name>(){}
```

- C++

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

class Test {
public:
    Test() { cout << "\n Constructor executed"; }

    ~Test() { cout << "\n Destructor executed"; }
};
main()
{
    Test t;

    return 0;
}
```

Output

```
Constructor executed

Destructor executed
```

- C++

```
#include <iostream>
using namespace std;
class Test {
public:
```

```
    Test() { cout << "\n Constructor executed"; }

    ~Test() { cout << "\n Destructor executed"; }
};

main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

Output

```
Constructor executed
Constructor executed
Constructor executed
Constructor executed
Destructor executed
Destructor executed
Destructor executed
Destructor executed
```

- C++

```
#include <iostream>
using namespace std;
int count = 0;
class Test {
public:
    Test()
    {
        count++;
        cout << "\n No. of Object created:\t" << count;
    }

    ~Test()
    {
        cout << "\n No. of Object destroyed:\t" << count;
        --count;
    }
};

main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

Output

```
No. of Object created:    1
No. of Object created:    2
No. of Object created:    3
No. of Object created:    4
No. of Object destroyed:    4
No. of Object destroyed:    3
No. of Object destroyed:    2
No. of Object destroyed:    1
```

Characteristics of a destructor:-

1. Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
2. Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
3. Destructor cannot be declared as static and const;
4. Destructor should be declared in the public section of the program.
5. Destructor is called in the reverse order of its constructor invocation.

Containership

Whenever an object of a class is declared as a member of another class it is known as a container class. In the containership the object of one class is declared in another class.

We can create an object of one class into another and that object will be a member of the class. This type of relationship between classes is known as containership or has-a relationship as one class contain the object of another class. And the class which contains the object and members of another class in this kind of relationship is called a container class.

The object that is part of another object is called contained object, whereas object that contains another object as its part or attribute is called container object.

Difference between containership and inheritance:

Containership

When features of existing class are wanted inside your new class, but, not its interface

for eg.

1) Computer system has a hard disk

2) Car has an Engine, steering wheels.

Inheritance

When you want to force the new type to be the same type as the base class.

eg.

1) Computer system is an electronic device

2) Car is a vehicle

Example:

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

```
class A
{
int a;

public: void get();
};
class B
{
int b;

At;

public: void getdata();

void A::get()

cin>>a; cout<<a;

void B:: getdata()

cin>>b;

cout<<b; t.get();
};

int main()
{
clrscr();
B ab;

ab getdata();
getch();
return 0;
}
```

Unit 2 Data Encapsulation and inheritance

Introduction to Data Hiding

Data hiding is a technique of hiding internal object details, i.e., data members. It is an object-oriented programming technique. Data hiding ensures, or we can say guarantees to restrict the data access to class members. It maintains data integrity.

Data hiding means hiding the internal data within the class to prevent its direct access from outside the class.

If we talk about data encapsulation so, **Data encapsulation** hides the private methods and class data parts, whereas **Data hiding** only hides class data components. Both data hiding and data encapsulation are essential concepts of object-oriented programming. **Encapsulation** wraps up the complex data to present a simpler view to the user, whereas **Data hiding** restricts the data use to assure data security.

Data hiding also helps to reduce the system complexity to increase the robustness by limiting the interdependencies between software components. Data hiding is achieved by using the private access specifier.

Example : Data Hiding

```
#include<iostream>
using namespace std;
class Base{

    int num;  //by default private
    public:

    void read();
    void print();

};

void Base :: read(){
    cout<<"Enter any Integer value"<<endl; cin>>num;

}

void Base :: print(){
    cout<<"The value is "<<num<<endl;
}

int main(){
    Base obj;

    obj.read();
    obj.print();

    return 0;
}
```

Encapsulation :

Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.
Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,

- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.

This is what **Encapsulation** is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.
Two Important property of Encapsulation

1. **Data Protection:** Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class’s public methods, ensuring controlled and secure data manipulation.
2. **Information Hiding:** Encapsulation hides the internal implementation details of a class from external code. Only the public interface of the class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

For example if we give input , and output should be half of input

```
#include <iostream>
using namespace std;
class temp{
     int a;
  int b;
  public:
  int solve(int input){
    a=input;
    b=a/2;
    return b;
  }
};

int main() {
  int n;
  cin>>n;
```

```
  temp t1;
  int ans=t1.solve(n);
  cout<<ans<<endl;

}
```

Features of Encapsulation

Below are the features of encapsulation:

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Increase in the security of data
5. It helps to control the modification of our data members.

Data Hiding	Encapsulation
It is associated with data security.	It can be defined as the wrapping up of data into a single module.
It also helps conceal the complexities of the application.	This will hide the complicated and confidential information about the application.
It focuses on hiding/restricting the data usage.	It focuses on hiding the complexity of the system.
It is considered as a process and a technique.	It is considered as a sub–process in the bigger process of data hiding.
The data is always private and inaccessible.	The encapsulated data can be private or public, depending on the requirement.

Abstraction in C++

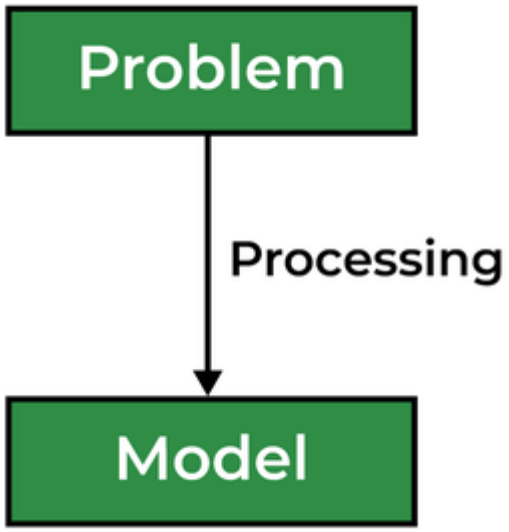
Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know

about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

Types of Abstraction:

1. **Data abstraction** – This type only shows the required information about the data and hides the unnecessary data.
2. **Control Abstraction** – This type only shows the required information about the implementation and hides unnecessary information.



Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be

marked as public. And these public members can access the private members as they are inside the class.

Example:

```
// C++ Program to Demonstrate the
// working of Abstraction
#include <iostream>
using namespace std;

class implementAbstraction {
private:
    int a, b;
public:
    // method to set values of
    // private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};
int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Output

```
a = 10
```

```
b = 20
```

You can see in the above program we are not allowed to access the variables a and b directly, however, one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

Difference between Abstraction and Encapsulation

The following table highlights all the important differences between abstraction and encapsulation –

S.No	Abstraction	Encapsulation
1.	It is the process of gaining information.	It is a method that helps wrap up data into a single module.
2.	The problems in this technique are solved at the interface level.	Problems in encapsulation are solved at the implementation level.
3.	It helps hide the unwanted details/information.	It helps hide data using a single entity, or using a unit with the help of method that helps protect the information.
4.	It can be implemented using abstract classes and interfaces.	It can be implemented using access modifiers like public, private and protected.
5.	The complexities of the implementation are hidden using interface and abstract class.	The data is hidden using methods such as getters and setters.
6.	Abstraction can be performed using objects that are encapsulated within a single module.	Objects in encapsulation don't need to be in abstraction.

Inheritance in C++

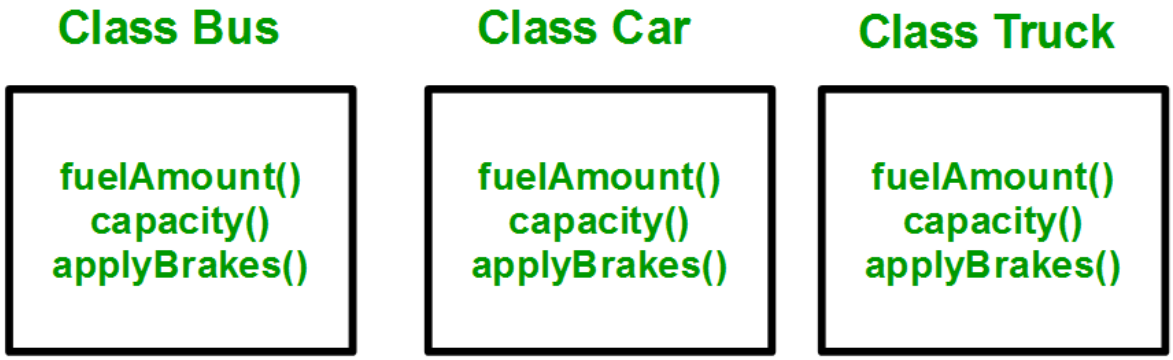
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.
Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called "derived class" or "child class" and the existing class is known as the "base class" or "parent class". The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

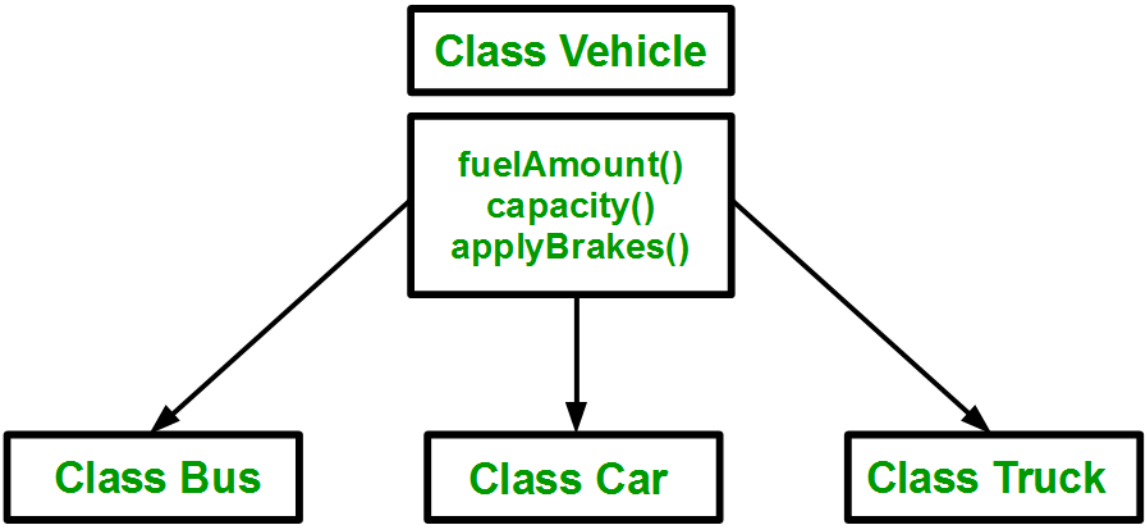
- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.
- **Why and when to use inheritance?**
- **Modes of Inheritance**
- **Types of Inheritance**

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).
Implementing inheritance in C++: For creating a sub-class that is inherited from the base class we have to follow the below syntax.
Derived Classes: A Derived class is defined as the class derived from the base class.
Syntax:

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
        //body
}
```

Where
class — keyword to create a new class
derived_class_name — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class
Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.
Example:

```
1. class ABC : private XYZ          //private derivation
        {          }
2. class ABC : public XYZ          //public derivation
        {          }
3. class ABC : protected XYZ          //protected derivation
        {          }
4. class ABC: XYZ                    //private derivation by default
{        }
```

Note:
o When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

o On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

```
// Example: define member function without argument within the class


#include<iostream>
using namespace std;
class Person
{
    int id;
    char name[100];

    public:
        void set_p()
        {
            cout<<"Enter the Id:";
            cin>>id;
            fflush(stdin);
            cout<<"Enter the Name:";
            cin.get(name,100);
        }

        void display_p()
        {
            cout<<endl<<id<<"\t"<<name<<"\t";
        }
};

class Student: private Person
{
    char course[50];
    int fee;
```

```
    public:
    void set_s()
        {
            set_p();
            cout<<"Enter the Course Name:";
            fflush(stdin);
            cin.getline(course,50);
            cout<<"Enter the Course Fee:";
            cin>>fee;
        }

        void display_s()
        {
            display_p();
            cout<<course<<"\t"<<fee<<endl;
        }
};

main()
{
    Student s;
    s.set_s();
    s.display_s();
    return 0;
}
```

Output:

```
Enter the Id: 101
Enter the Name: Dev
Enter the Course Name: GCS
Enter the Course Fee:70000

101      Dev     GCS     70000
```

```
// Example: define member function without argument outside the class
```

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

class Person
{
    int id;
    char name[100];

    public:
        void set_p();
        void display_p();
};

void Person::set_p()
{
    cout<<"Enter the Id:";
    cin>>id;
    fflush(stdin);
    cout<<"Enter the Name:";
    cin.get(name,100);
}

void Person::display_p()
{
    cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
    char course[50];
    int fee;

    public:
        void set_s();
```

```
        void display_s();
};

void Student::set_s()
{
    set_p();
    cout<<"Enter the Course Name:";
    fflush(stdin);
    cin.getline(course,50);
    cout<<"Enter the Course Fee:";
    cin>>fee;
}

void Student::display_s()
{
    display_p();
    cout<<"\t"<<course<<"\t"<<fee;
}

main()
{
    Student s;
    s.set_s();
    s.display_s();
    return 0;
}
```

Output

```
Enter the Id:Enter the Name:Enter the Course Name:Enter the Course Fee:
0    t    0
```

```
// Example: define member function with argument outside the class

#include<iostream>
```

```
#include<string.h>
using namespace std;

class Person
{
    int id;
    char name[100];

    public:
        void set_p(int,char[]);
        void display_p();
};

void Person::set_p(int id,char n[])
{
    this->id=id;
    strcpy(this->name,n);
}

void Person::display_p()
{
    cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
    char course[50];
    int fee;
    public:
    void set_s(int,char[],char[],int);
    void display_s();
};

void Student::set_s(int id,char n[],char c[],int f)
```

```
{
    set_p(id,n);
    strcpy(course,c);
    fee=f;
}


void Student::display_s()
{
    display_p();
    cout<<"t"<<course<<"\t"<<fee;
}

main()
{
    Student s;
    s.set_s(1001,"Ram","B.Tech",2000);
    s.display_s();
    return 0;
}
```

```
// C++ program to demonstrate implementation
// of Inheritance

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

// Base class
class Parent {
public:
    int id_p;
};
```

```
// Sub class inheriting from Base Class(Parent)
class Child : public Parent {
public:
    int id_c;
};


// main function
int main()
{
    Child obj1;


    // An object of class child has all data members
    // and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is: " << obj1.id_c << '\n';
    cout << "Parent id is: " << obj1.id_p << '\n';


    return 0;
}
```

Output

```
Child id is: 7
Parent id is: 91
```

Output:

```
Child id is: 7
Parent id is: 91
```

In the above program, the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.
Modes of Inheritance: There are 3 modes of inheritance.

1. **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

3. **Private Mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

```
// C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object.
class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is default for classes
{
    // x is private
```

```
    // y is private
    // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

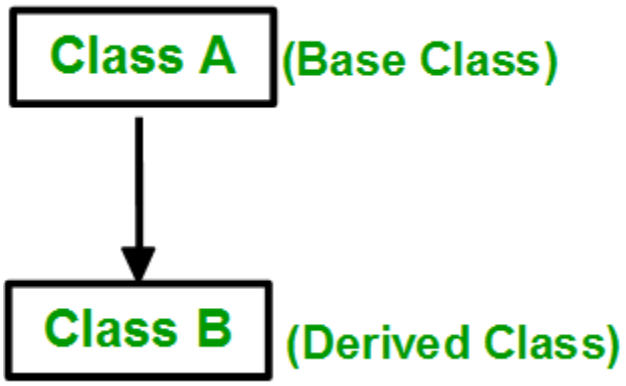
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Types of Inheritance in C++

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



Syntax:

```
class subclass_name : access_mode base_class
{
  // body of subclass
};
```

OR

```
class A
{
... .. ...
};

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

```
// C++ program to explain
// Single inheritance
#include<iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
    }
};

// sub class derived from a single base classes
class Car : public Vehicle {

};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
```

```
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
```

```
// Example:


#include<iostream>
using namespace std;

class A
{
    protected:
    int a;
    public:
        void set_A()
        {
            cout<<"Enter the Value of A=";
            cin>>a;

        }
        void disp_A()
        {
            cout<<endl<<"Value of A="<<a;
        }
};



class B: public A
{
    int b,p;
```

```
    public:
        void set_B()
        {
            set_A();
            cout<<"Enter the Value of B=";
            cin>>b;
        }

        void disp_B()
        {
            disp_A();
            cout<<endl<<"Value of B="<<b;
        }

        void cal_product()
        {
            p=a*b;
            cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
        }

};

main()
{

    B _b;
    _b.set_B();
    _b.cal_product();

    return 0;

}
```

Output:- Enter the Value of A= 3 3 Enter the Value of B= 5 5 Product of 3 * 5 = 15

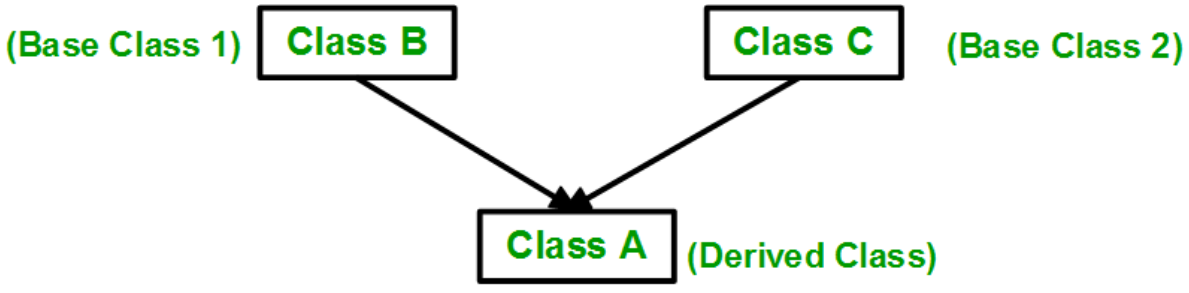
```
// Example:


#include<iostream>
using namespace std;
 class A
{
    protected:
    int a;
     public:
        void set_A(int x)
        {
             a=x;
        }

        void disp_A()
        {
            cout<<endl<<"Value of A="<<a;
        }
};

class B: public A
{
    int b,p;

    public:
        void set_B(int x,int y)
        {
            set_A(x);
            b=y;
        }

        void disp_B()
        {
            disp_A();
```

```
            cout<<endl<<"Value of B="<<b;
        }

        void cal_product()
        {
            p=a*b;
            cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
        }

};

main()
{
    B _b;
    _b.set_B(4,5);
    _b.cal_product();
    return 0;
}
```

Output

```
Product of 4 * 5 = 20
```

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
```

```
class B
{
... .. ...
};
class C
{
... .. ...
};
class A: public B, public C
{
... ... ...
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and the access mode for every base class must be specified.

- CPP

```cpp
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle\n";
    }
};
```

```
// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

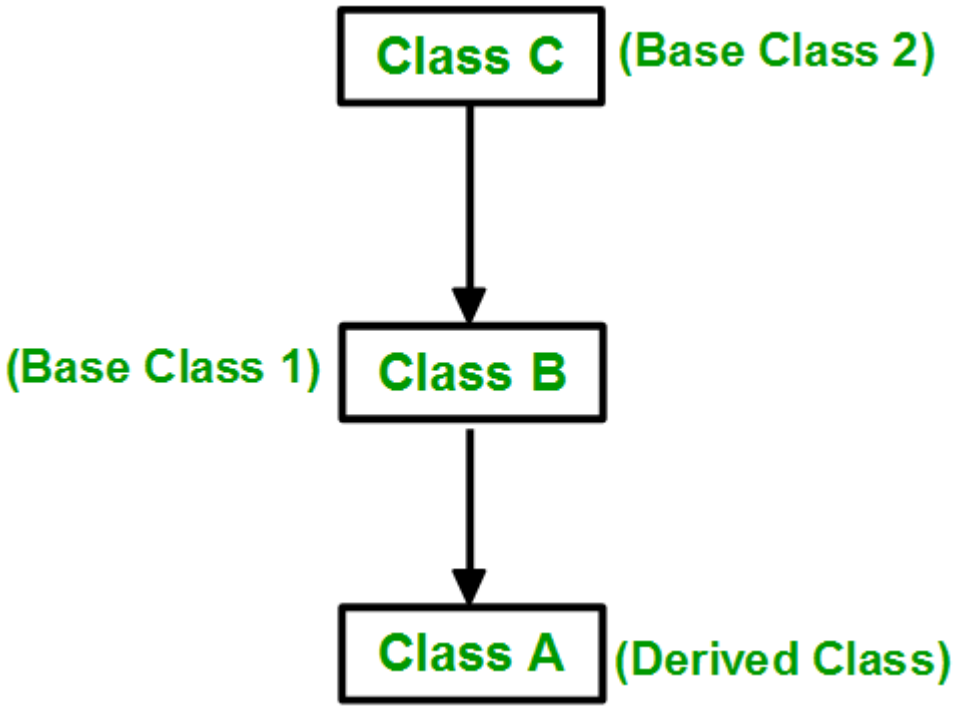
- C++

```
// Example:

#include<iostream>
using namespace std;

class A
{
              protected:
              int a;

              public:
                  void set_A()
                  {
                        cout<<"Enter the Value of A=";
                        cin>>a;
```

```
                }

                void disp_A()
                {
                      cout<<endl<<"Value of A="<<a;
                }
};

class B: public A
{
            protected:
                int b;

            public:
                   void set_B()
                {
                     cout<<"Enter the Value of B=";
                      cin>>b;
                  }


                void disp_B()
                  {
                     cout<<endl<<"Value of B="<<b;
                  }
};

class C: public B
{
      int c,p;

      public:
          void set_C()
```

```
          {
                cout<<"Enter the Value of C=";
                cin>>c;
           }

           void disp_C()
           {
                cout<<endl<<"Value of C="<<c;
           }

            void cal_product()
             {
                  p=a*b*c;
                 cout<<endl<<"Product of "<<a<<" * "<<b<<" * "<<c<<" = "<<p;
              }
};

main()
{

    C _c;
    _c.set_A();
    _c.set_B();
    _c.set_C();
    _c.disp_A();
    _c.disp_B();
    _c.disp_C();
    _c.cal_product();

    return 0;

}
```

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



Syntax:-

```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

- CPP

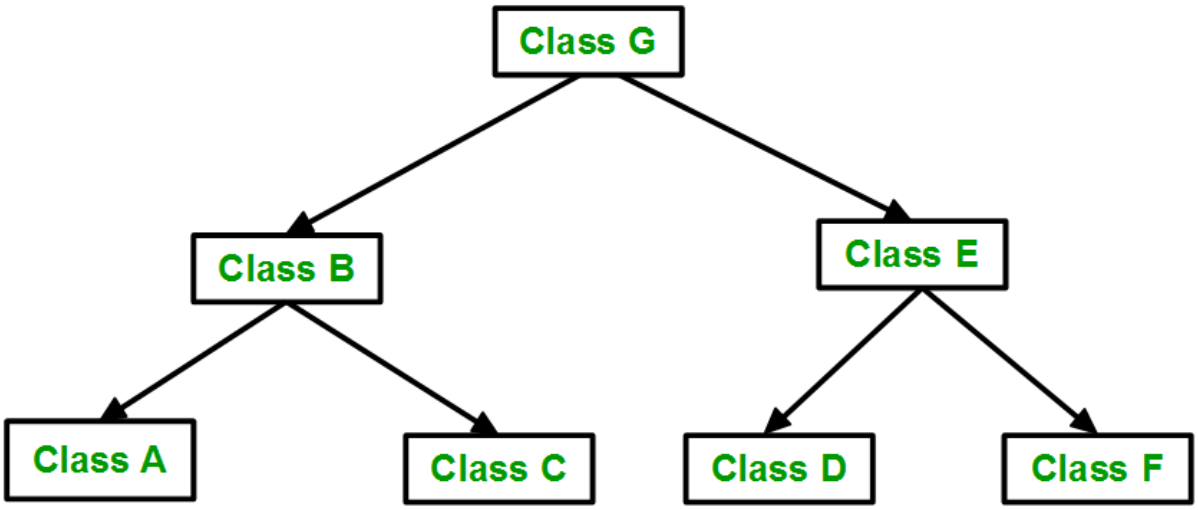
```cpp
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
```

```
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles\n";
    }
};
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
    Car() { cout << "Car has 4 Wheels\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Output

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:-

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

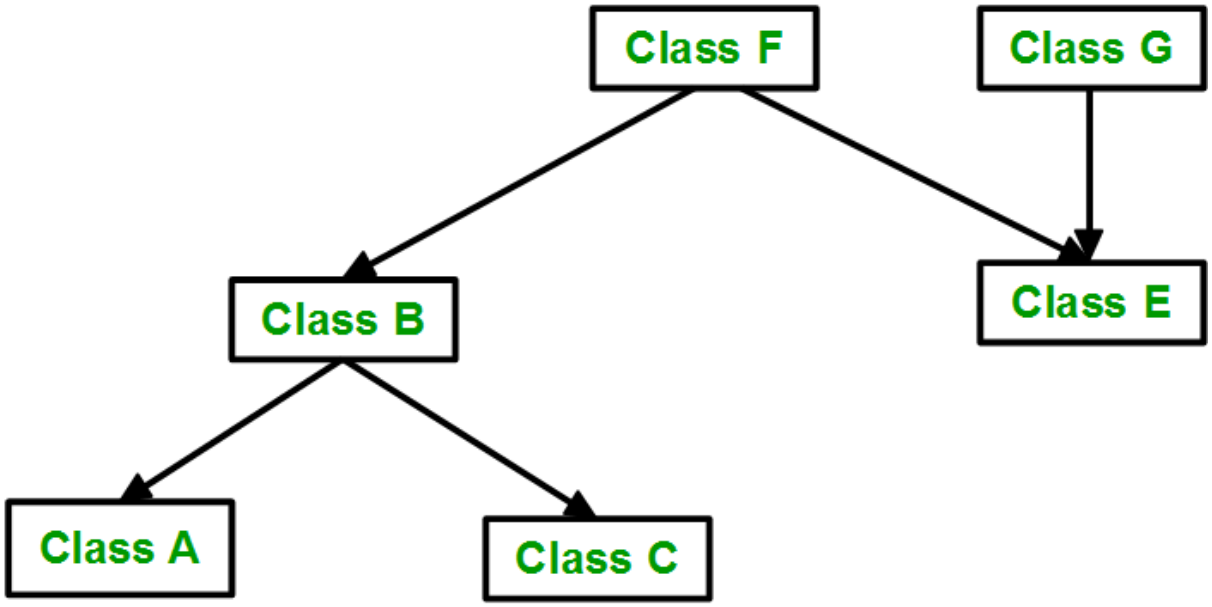
```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
```

```
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output

```
This is a Vehicle
This is a Vehicle
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritances:



- CPP

```cpp
// C++ program for Hybrid Inheritance


#include <iostream>
using namespace std;


// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};


// base class
class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }
};


// first sub class
class Car : public Vehicle {
};


// second sub class
```

```
class Bus : public Vehicle, public Fare {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Bus obj2;
    return 0;
}
```

Output

```
This is a Vehicle
Fare of Vehicle
```

- C++

```
// Example:

#include <iostream>
using namespace std;

class A
{
    protected:
    int a;
    public:
    void get_a()
    {
       cout << "Enter the value of 'a' : ";
       cin>>a;
    }
};
```

```
class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
       cout << "Enter the value of 'b' : ";
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        cout << "Enter the value of c is : ";
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         cout << "Multiplication of a,b,c is : " <<a*b*c;
```

```
    }
};



int main()
{
    D d;
    d.mul();
    return 0;
}
```

6. A special case of hybrid inheritance: Multipath inheritance: A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.
Example:

- CPP

```
// C++ program demonstrating ambiguity in Multipath
// Inheritance

#include <iostream>
using namespace std;

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
```

```
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

int main()
{
    ClassD obj;

    // obj.a = 10;                  // Statement 1, Error
    // obj.a = 100;                 // Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a from ClassB  : " << obj.ClassB::a;
    cout << "\n a from ClassC  : " << obj.ClassC::a;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Output

```
 a from ClassB  : 10
 a from ClassC  : 100
 b : 20
 c : 30
 d : 40
```

Output:

```
a from ClassB : 10

a from ClassC : 100

b : 20

c : 30

d : 40
```

In the above example, both ClassB and ClassC inherit ClassA, they both have a single copy of ClassA. However Class-D inherits both ClassB and ClassC, therefore Class-D has two copies of ClassA, one from ClassB and another from ClassC.
If we need to access the data member of ClassA through the object of Class-D, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bcoz compiler can't differentiate between two copies of ClassA in Class-D.

There are 2 Ways to Avoid this Ambiguity:

1) Avoiding ambiguity using the scope resolution operator: Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

- CPP

```
obj.ClassB::a = 10;       // Statement 3

obj.ClassC::a = 100;      // Statement 4
```

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

- CPP

```
#include<iostream>


class ClassA

{

  public:

    int a;

};


class ClassB : virtual public ClassA

{

  public:

    int b;

};
```

```
class ClassC : virtual public ClassA
{
  public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
  public:
    int d;
};

int main()
{
    ClassD obj;

    obj.a = 10;       // Statement 3
    obj.a = 100;      // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Output:

```
a : 100
b : 20
c : 30
d : 40
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

Enumeration data type in CPP

Enumeration (Enumerated type) is a user-defined data type that can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.

If we assign a float value to a character value, then the compiler generates an error. In the same way, if we try to assign any other value to the enumerated data types, the compiler generates an error. Enumerator types of values are also known as enumerators. It is also assigned by zero the same as the array. It can also be used with switch statements.

Syntax:

```
enum enumerated-type-name
{    value1, value2, value3…..valueN
};
```

For Example: If a gender variable is created with the value male or female. If any other value is assigned other than male or female then it is not appropriate. In this situation, one can declare the enumerated type in which only male and female values are assigned.

The **enum** keyword is used to declare enumerated types after that enumerated type name was written then under curly brackets possible values are defined. After defining Enumerated type variables are created. Enumerators can be created in two types:-

1. It can be declared during declaring enumerated types, just add the name 006Ff the variable before the semicolon. or,
2. Besides this, we can create enumerated type variables as the same as the normal variables.

```
enumerated-type-name variable-name = value;
```

By default, the starting code value of the first element of the enum is 0 (as in the case of the array). But it can be changed explicitly.

Example:

```
enum enumerated-type-name{value1=1, value2, value3};
```

Also, The consecutive values of the enum will have the next set of code value(s).

Example:

```
//first_enum is the enumerated-type-name
enum first_enum{value1=1, value2=10, value3};

In this case,
first_enum e;
e=value3;
cout<<e;
Output:
11
```

```
int main()
```

```
{
    // Defining enum Gender
    enum Gender { Male, Female };

    // Creating Gender type variable
    Gender gender = Male;
    switch (gender) {
    case Male:
        cout << "Gender is Male";
        break;
    case Female:
        cout << "Gender is Female";
        break;
    default:
        cout << "Value can be Male or Female";
    }
    return 0;
}
```

Output:

```
Gender is Male
```

```
enum year {
    Jan,
    Feb,
    Mar,
    Apr,
    May,
    Jun,
    Jul,
    Aug,
    Sep,
    Oct,
    Nov,
    Dec
};

// Driver Code
int main()
{
    int i;
    // Traversing the year enum
    for (i = Jan; i <= Dec; i++)
        cout << i << " ";
```

```
    return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Example 1: Enumeration Type

```
#include <iostream>
using namespace std;
enum week { Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday };
int main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    return 0;
}
```

Output

```
Day 4
```

Example2: Changing Default Value of Enums

```
#include <iostream>
using namespace std;
enum seasons { spring = 34, summer = 4, autumn = 9, winter =
32};
int main() {
    seasons s;
    s = summer;
    cout << "Summer = " << s << endl;
    return 0;
}
```

Dynamic Memory Allocation Operators

There are two types of memory allocation.
1) **Static memory allocation** - allocated by the compiler. Exact size and type of memory must be known as compile time.
2) **Dynamic memory allocation**-memory allocated during run time. To dynamically allocate memory in C++, we use the new operator and to de-allocate dynamic memory, we use the delete operator.

How is memory allocated/deallocated in C++?
C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete**, that perform the task of allocating and freeing the memory in a better and easier way.
new:

- Dynamic memory allocation means creating memory at runtime. For example, when

we declare an array, we must provide size of array in our source code to allocate memory at compile time. But if we need to allocate memory at runtime we must use new operator followed by data

If we need to allocate memory for more than one element, we must provide to number of elements required in square bracket[ ]. It will return the address of t byte of memory.

Syntax of new operator:-

ptr = new data-type;

//allocte memory for one element

ptr = new data-type [size];

//allocte memory for fixed number of element

delete:

• Delete operator is used to deallocate the memory created by new operator time.
at run- Once the memory is no longer needed it should by freed so that the memory becomes available again for other request of dynamic memory.

//deallocte memory for one element

Syntax of delete operator:-

delete ptr;

delete[] ptr;

//deallocte memory for array

Example:

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

void main()
{
int size,i;
int *ptr;
cout<<"\n\tEnter of Array: ";
cin>>size;
ptr = new int[size];

//Creating memory at run-time and return byte of address to ptr.
for(i=0;i<5;i++)
{
cout<<"\nEnter any number: ";
cin>>ptr[i];
```

```
for(i=0;i<5;i++)
{
//Output arrray to console.

cout<<ptr[i]<<","
delete[] ptr;
}}
//deallocating all the memory created by new operator

}
```

Output:

Enter size of Array: 5

Enter any number: 78

Enter any number: 45

Enter any number: 12

Enter any number: 89 Enter any number: 56

78, 45, 12, 89, 56,

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++

```
1. class class_name
2. {
3.     friend data_type function_name(argument/s);          // syntax of friend function.
4. };
```

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;
 }
```

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.

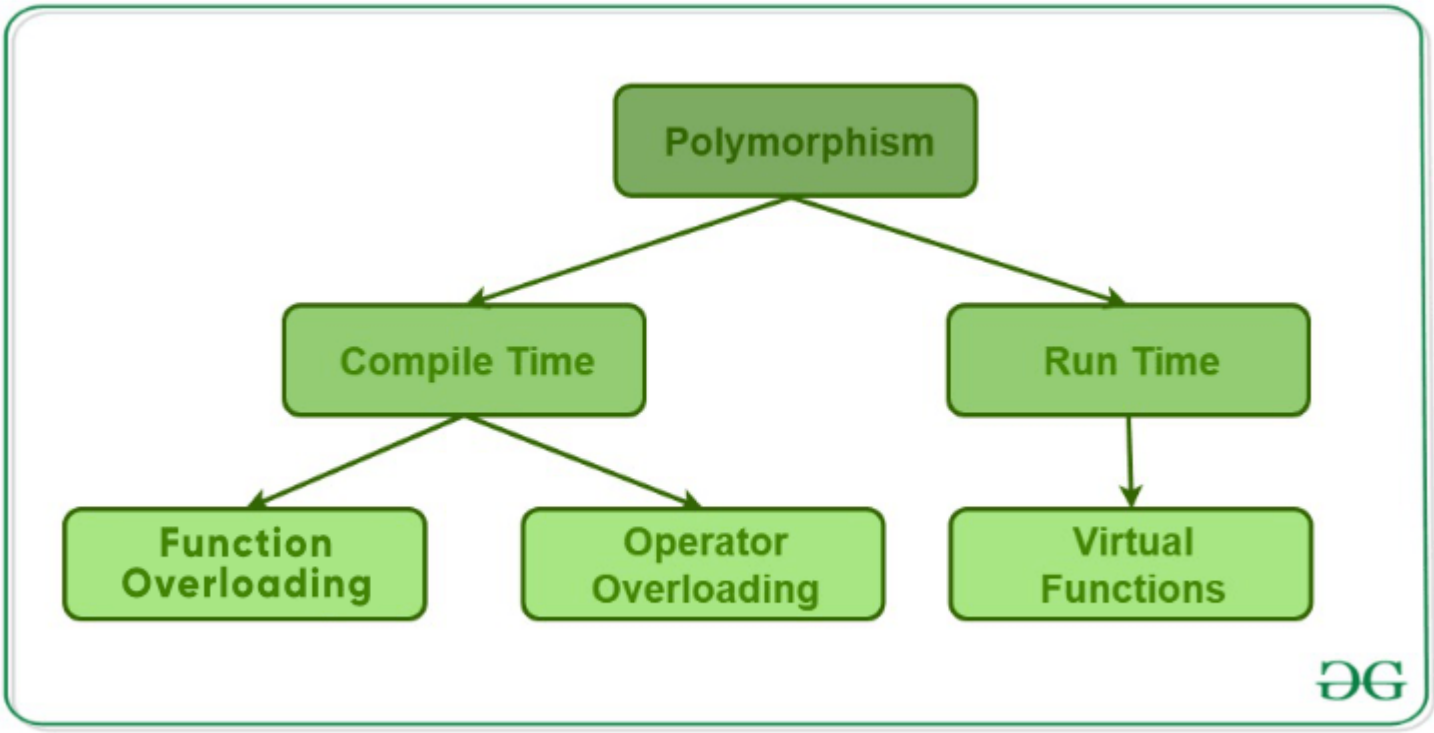
A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.

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

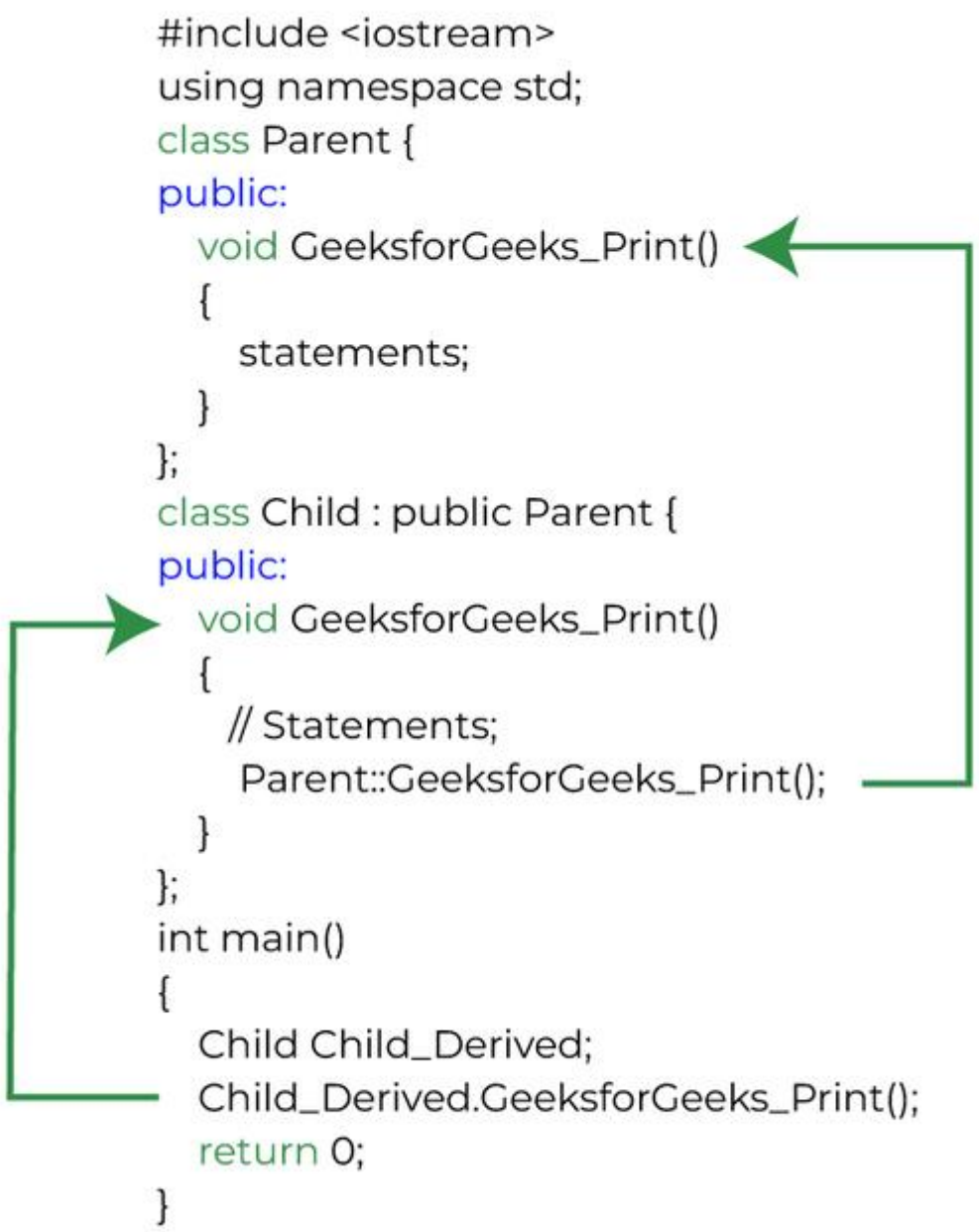
1. Call Overridden Function From Derived Class

```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
```



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

2. Call Overridden Function Using Pointer

```cpp
// 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
```

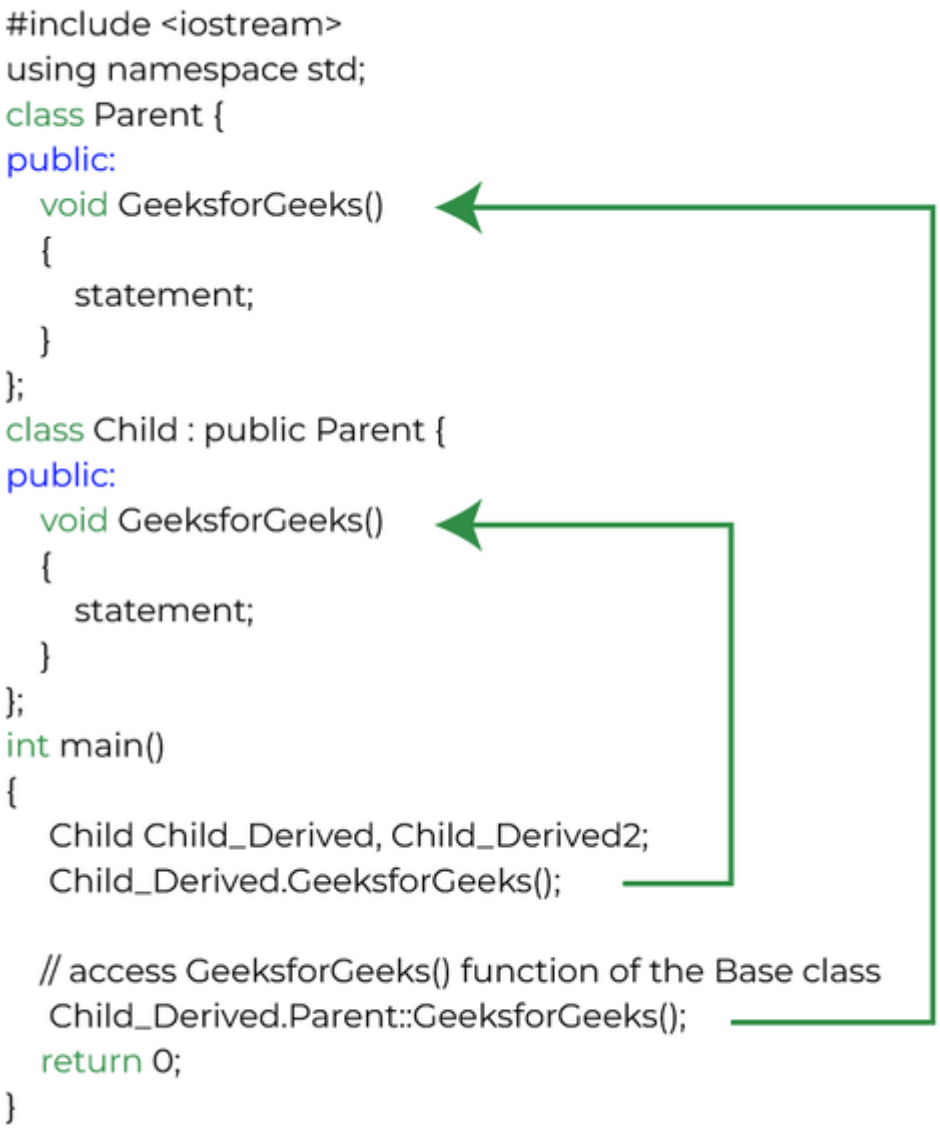
3. Access of Overridden Function to the Base Class

```
// 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
```



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.

```
// 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
```

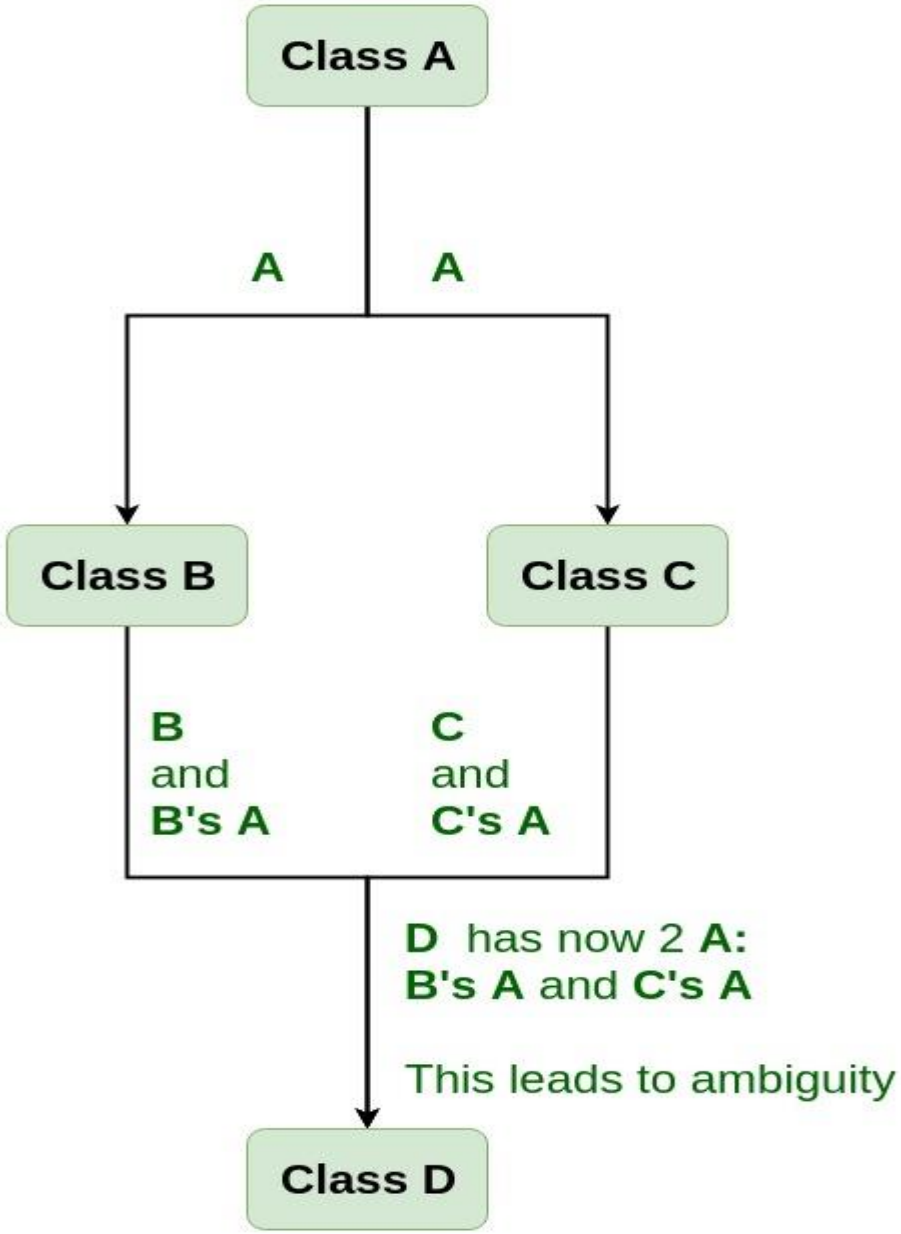
```
    cout << d.color;
}
```

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.

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

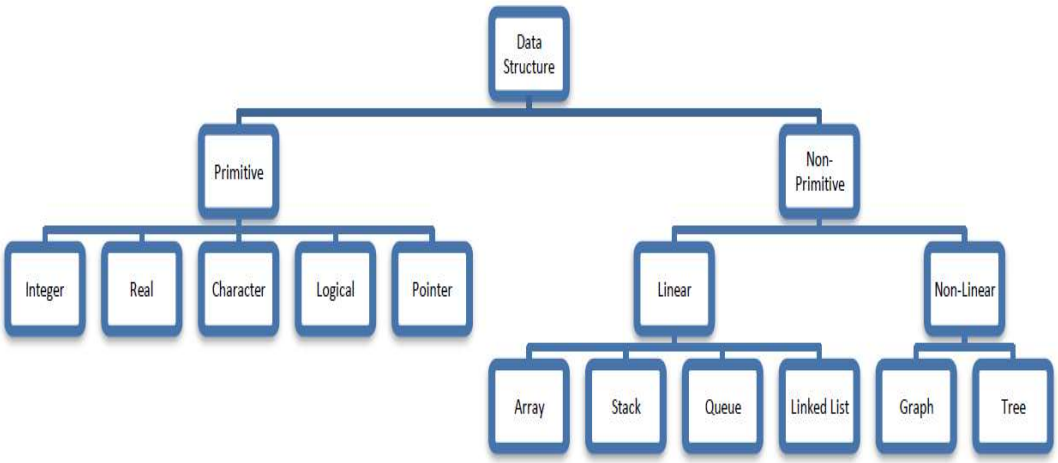
The non-primitive data structures can further be classified in two categories.

- Linear

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

- 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

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

- 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

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

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

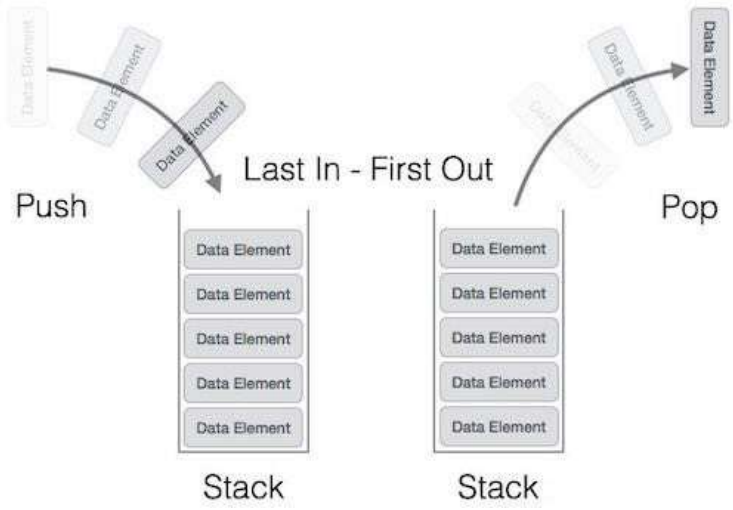
References:
Data Structures by R. D. Morena,Priti Tailor, Vaishali Dindoliwala
An Introduction to Data Structures with applications, Trembley – Tata McGraw Hill.

- 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)

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

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

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

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  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 
b. Add  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

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

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 -

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

- 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

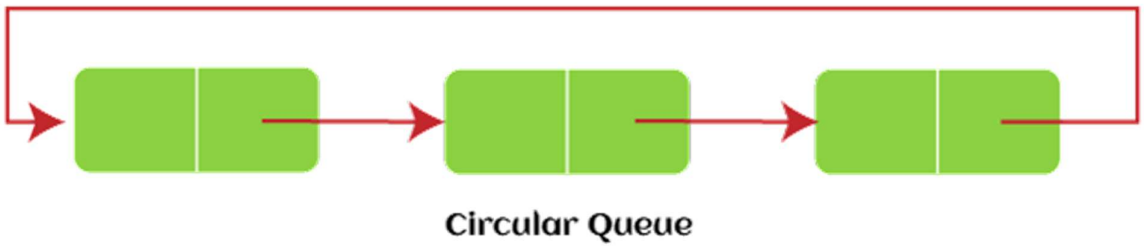
References:
Data Structures by R. D. Morena,Priti Tailor, Vaishali Dindoliwala
An Introduction to Data Structures with applications, Trembley – Tata McGraw Hill.

[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.

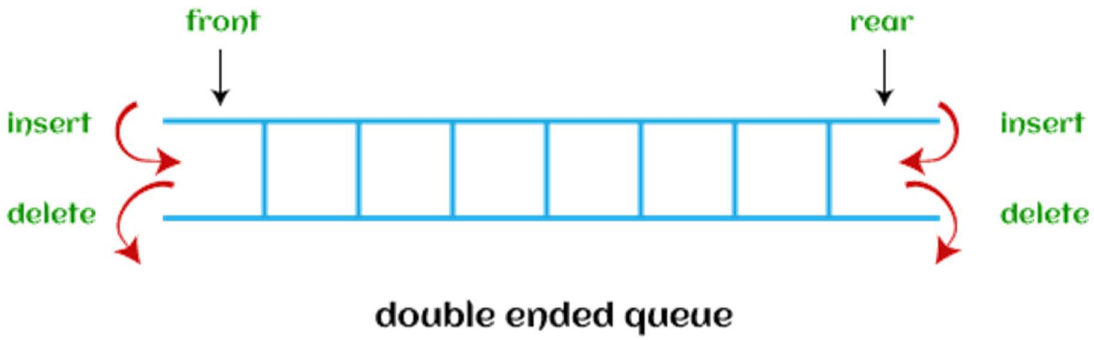
Deque (or, Double Ended Queue)

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.

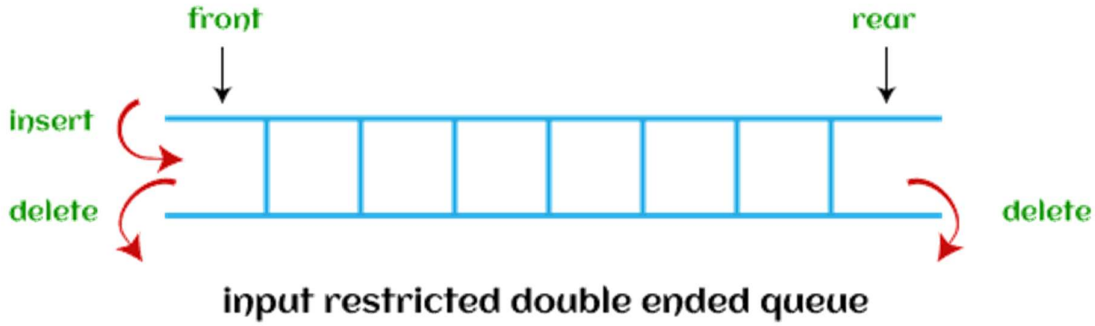
The representation of the deque is shown in the below image -

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

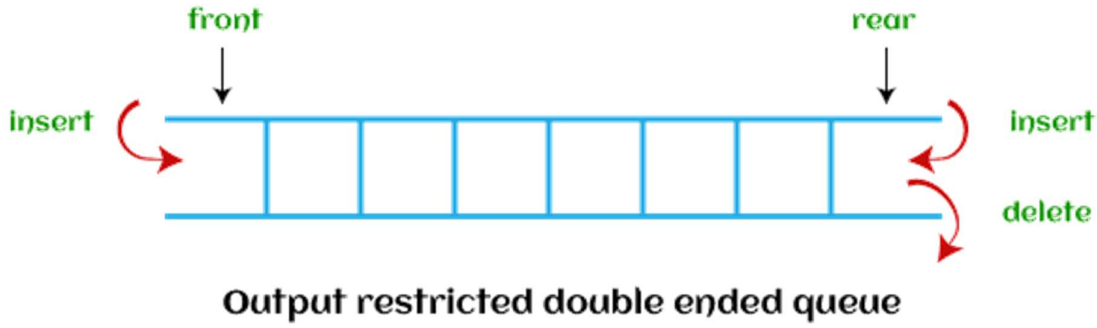


There are two types of deque that are discussed as follows -

- **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.



- **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.



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

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

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

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

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.

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

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

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

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

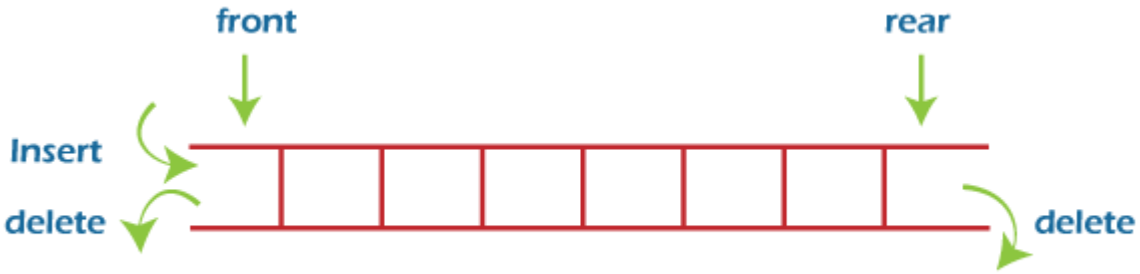
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

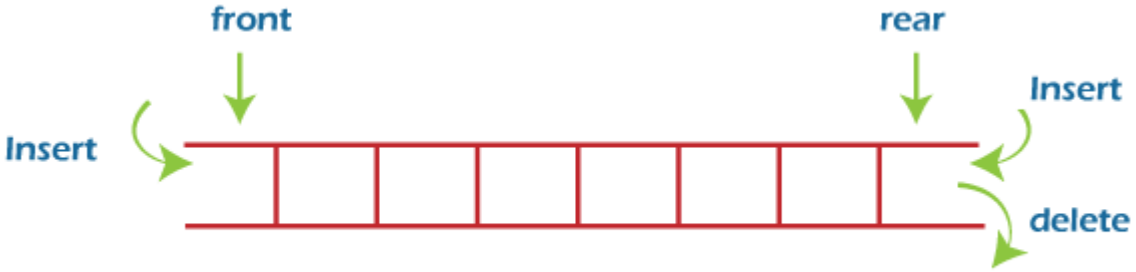
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



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 -

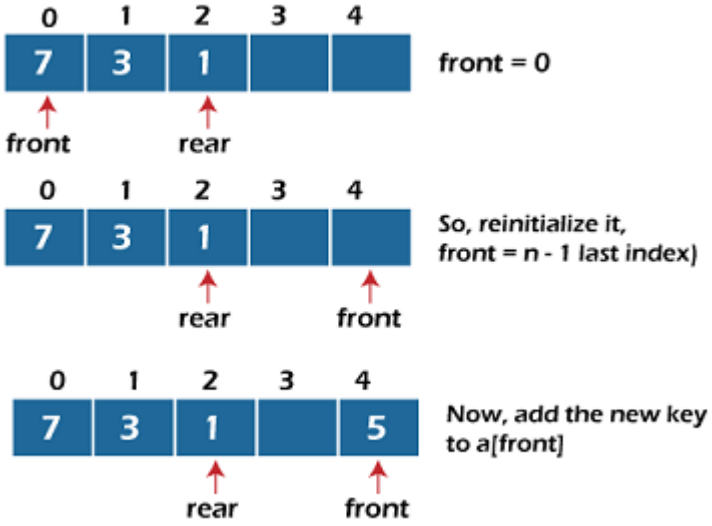
- Get the front item from the deque

- 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 -

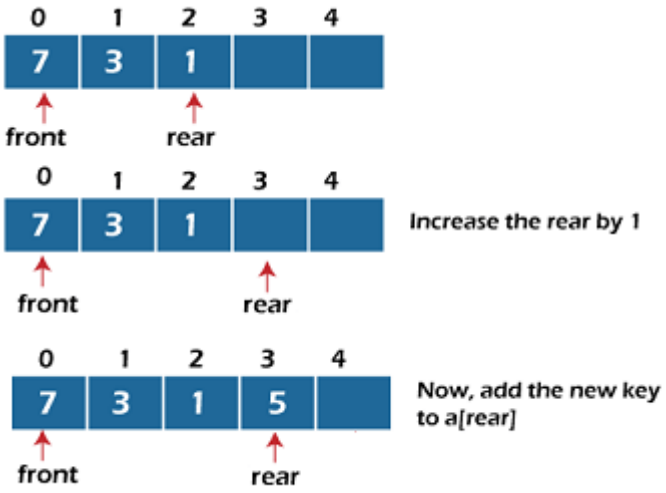
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



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

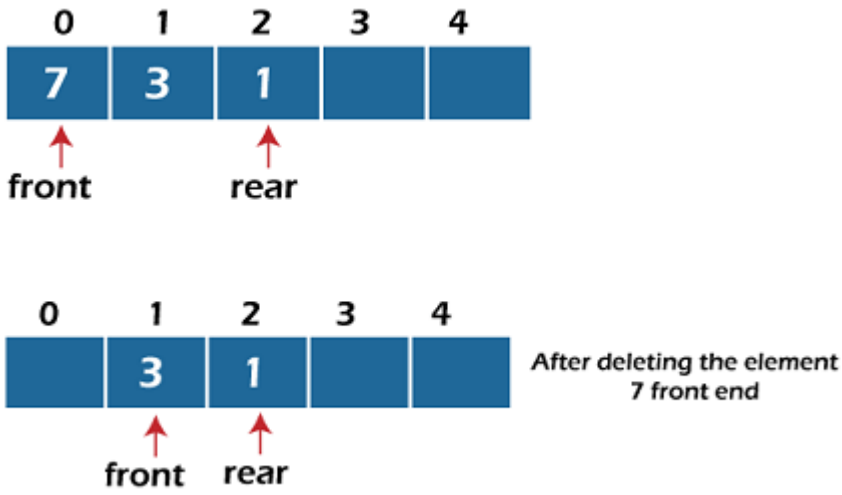
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

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

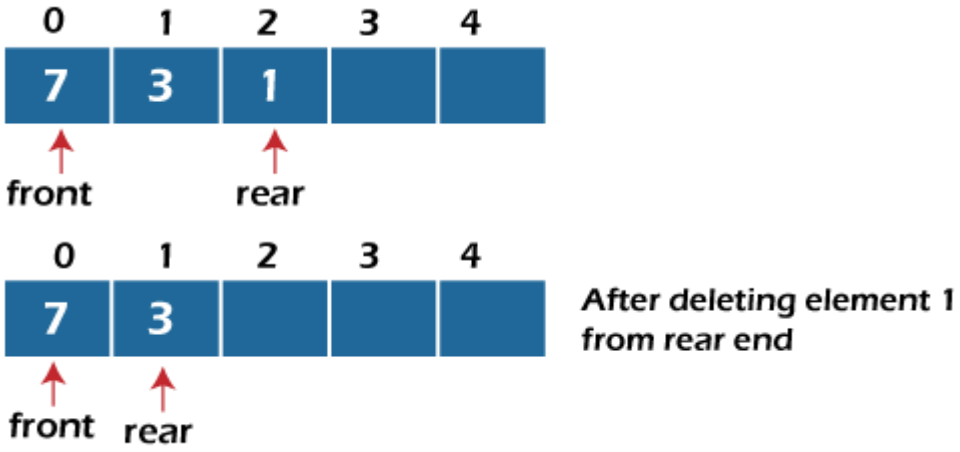
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

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
```

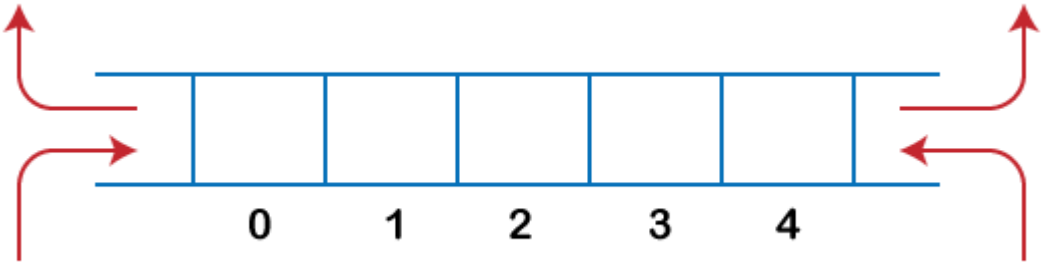
```
    delete_front();
    delete_rear();
    display(); // calling display function to retrieve values after deletion
    return 0;
}
```

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
```

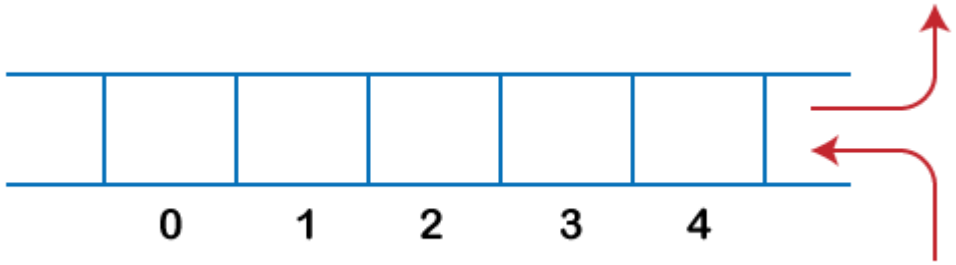
Dequeue in CPP

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**.

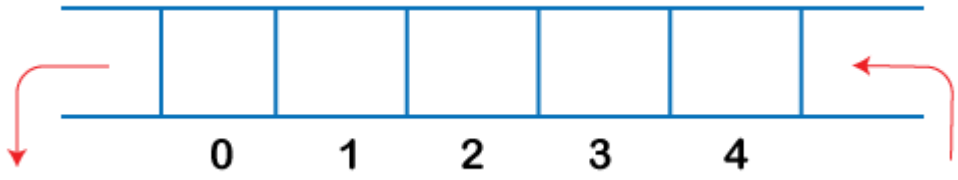


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

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.

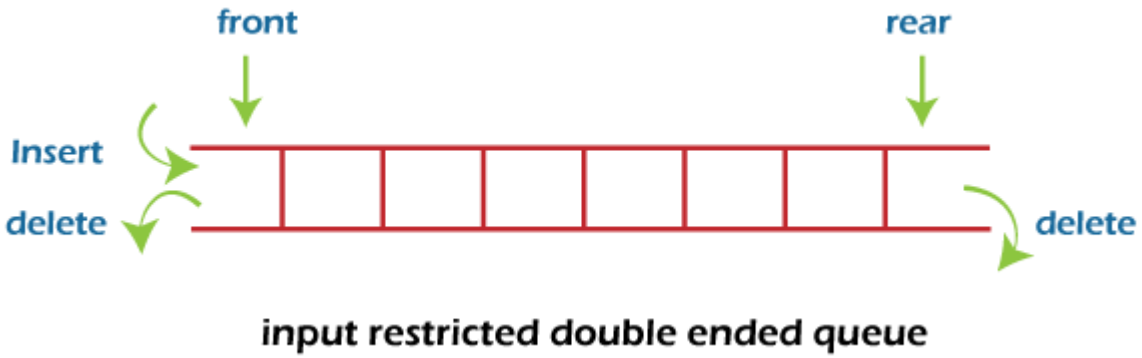


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.



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:

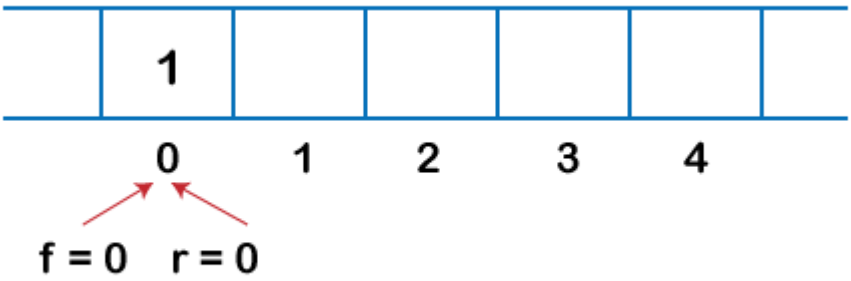
- **Insert at front**
- **Insert at rear**
- **Delete at front**
- **Delete from rear**

Implementation of Deque using a circular array

The following are the steps to perform the operations on the Deque:

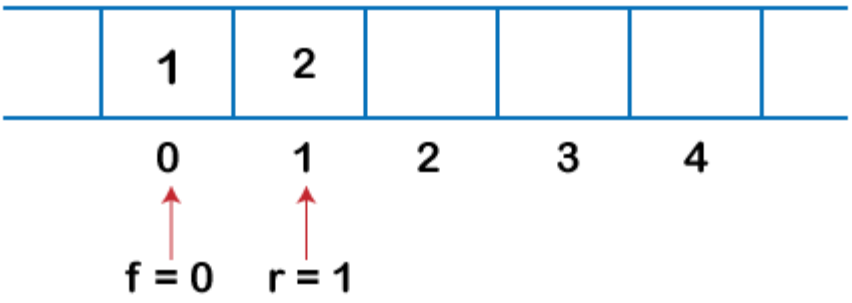
Enqueue operation

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.**

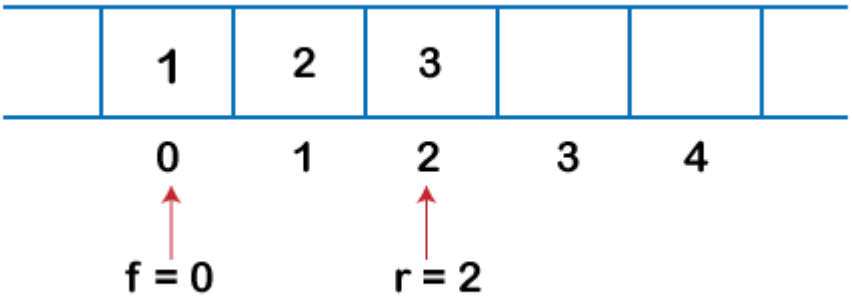


3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the

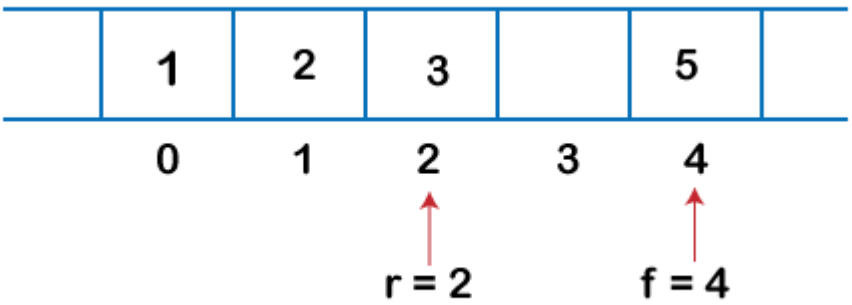
second element, and the front is pointing to the first element.



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.

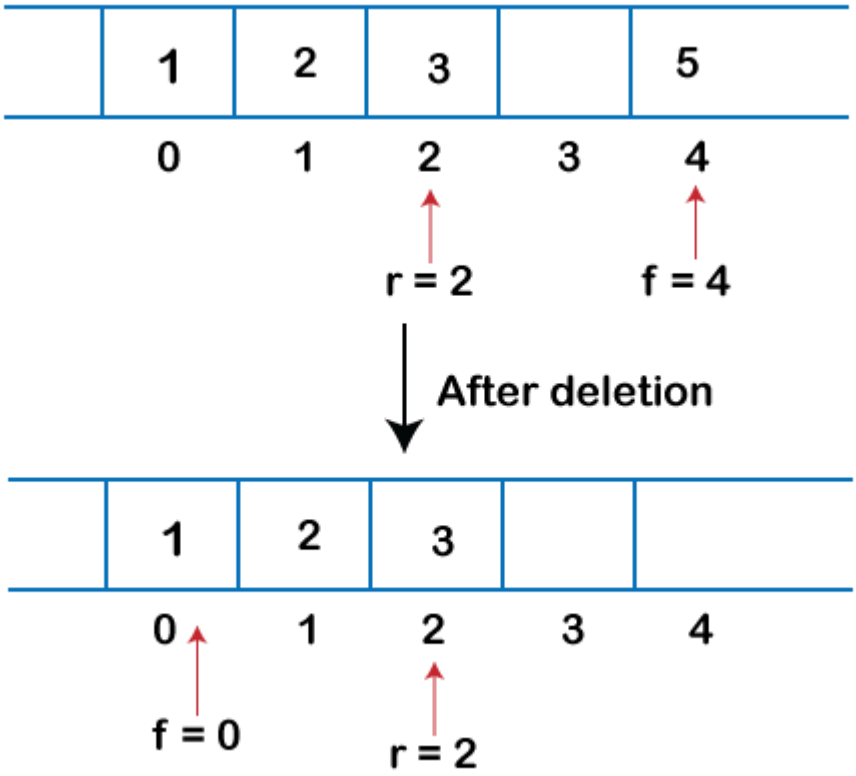


5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



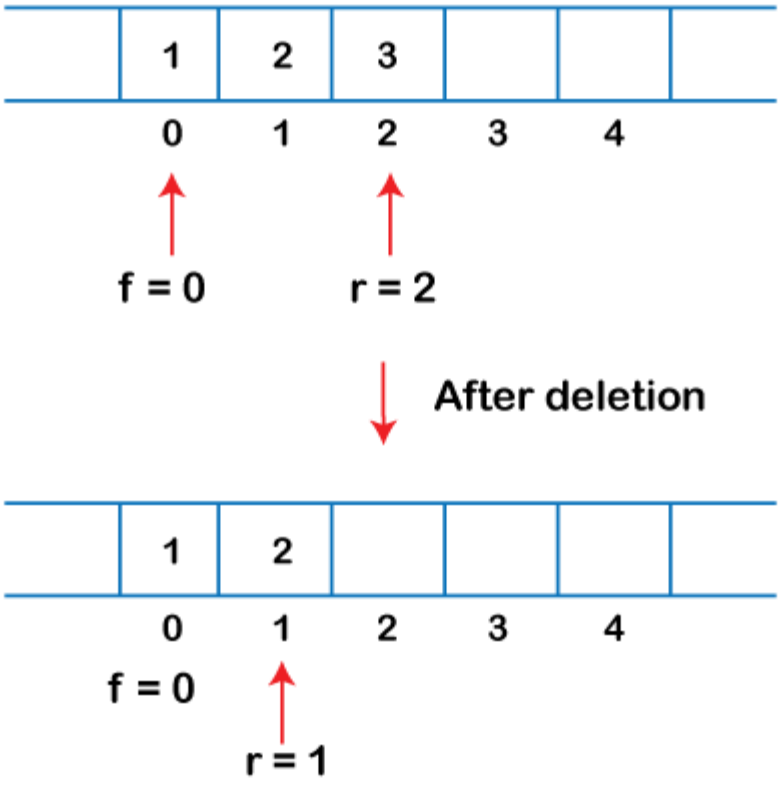
Dequeue Operation

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.

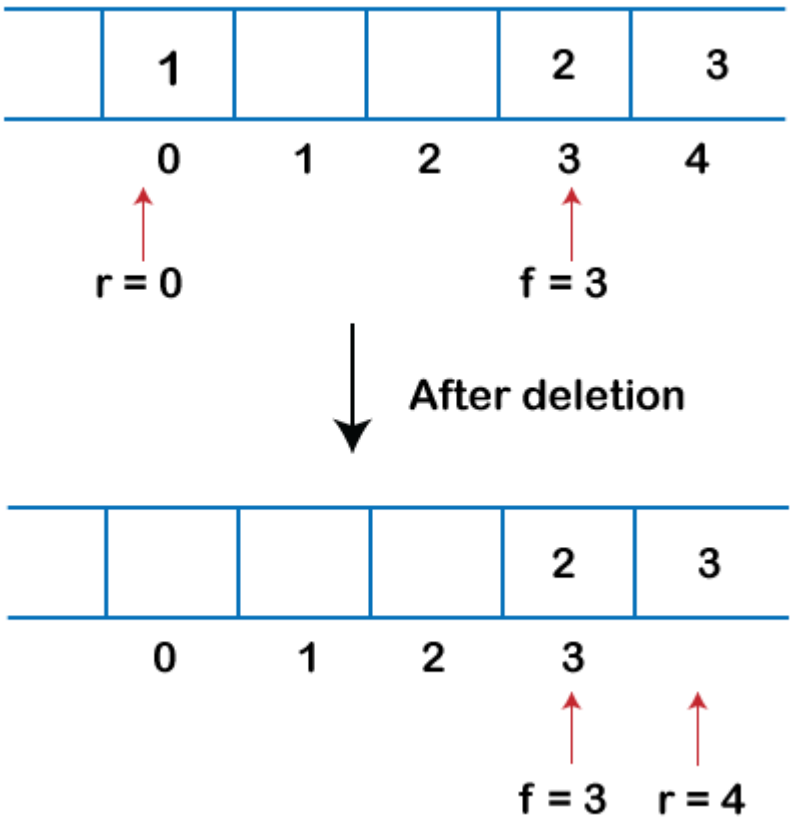


2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the

below figure:

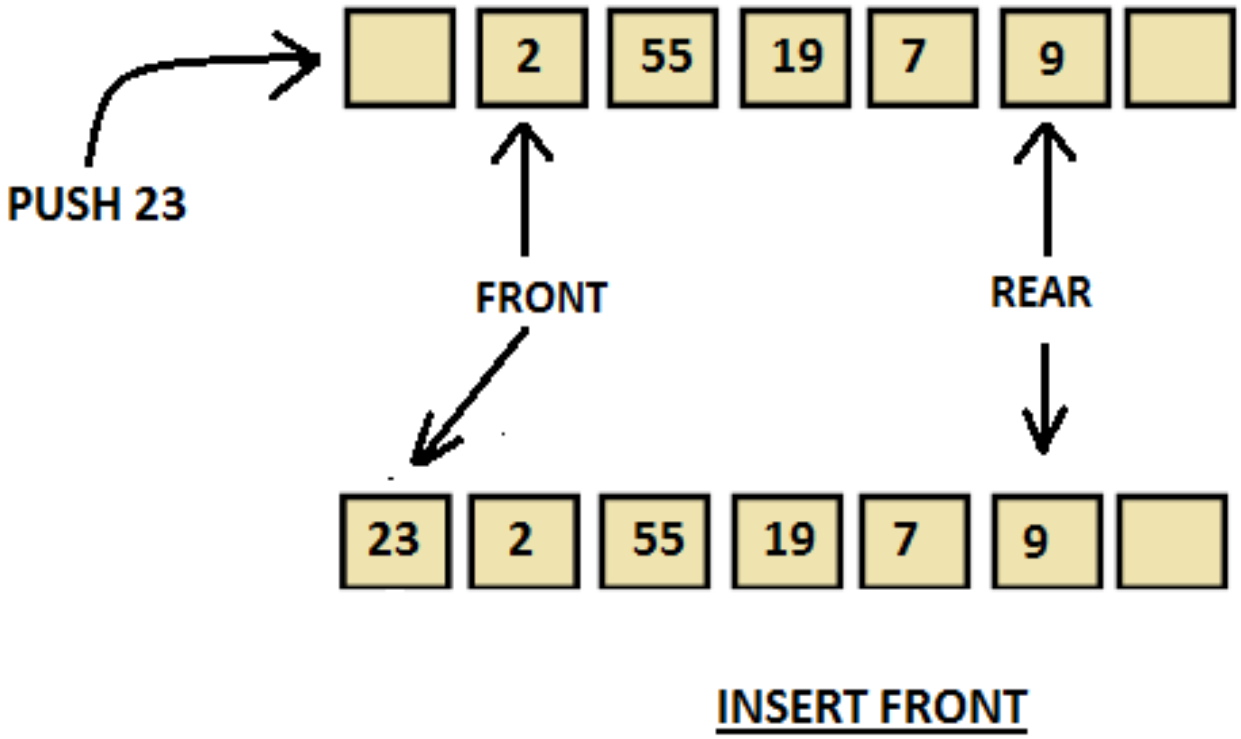


3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



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
```

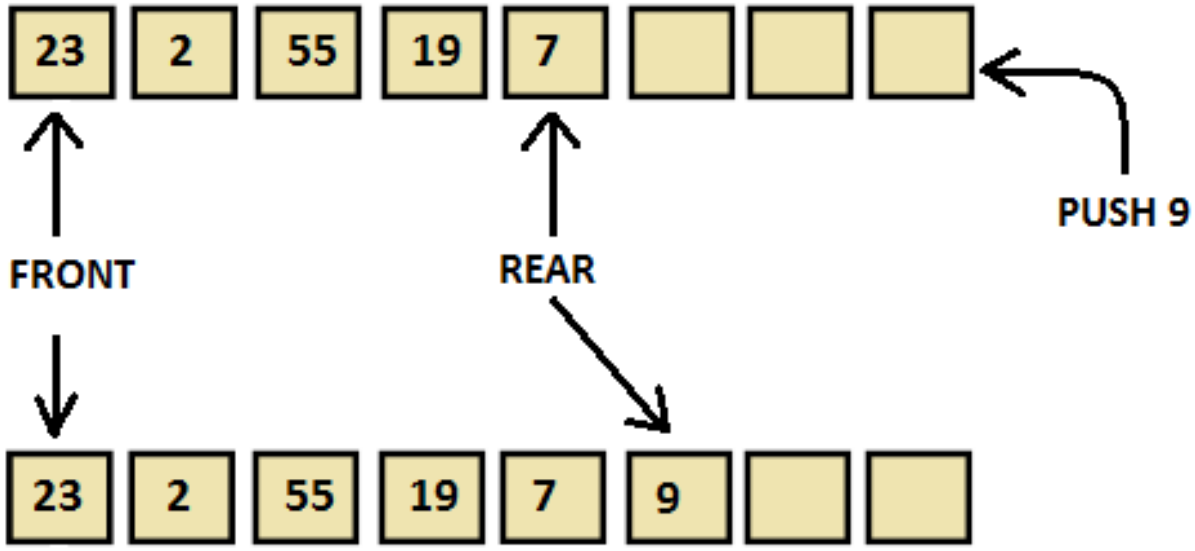


INSERT FRONT

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
```

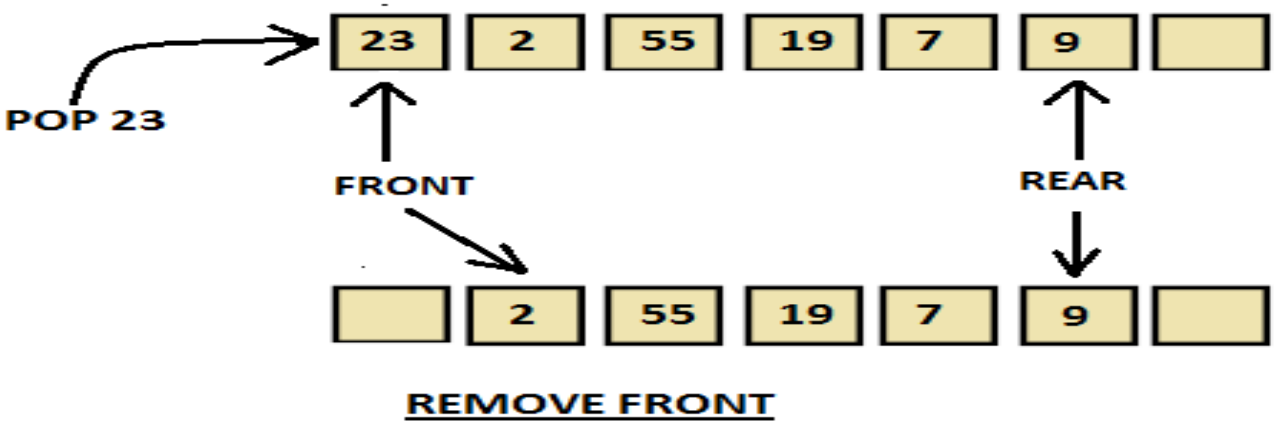


INSERT BACK

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

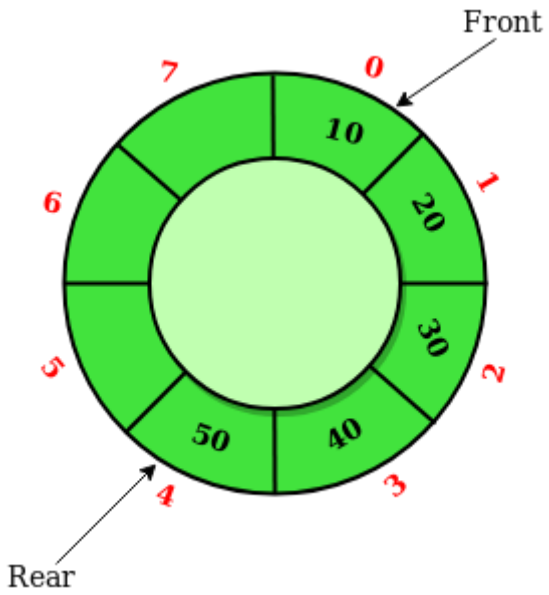
- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

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.

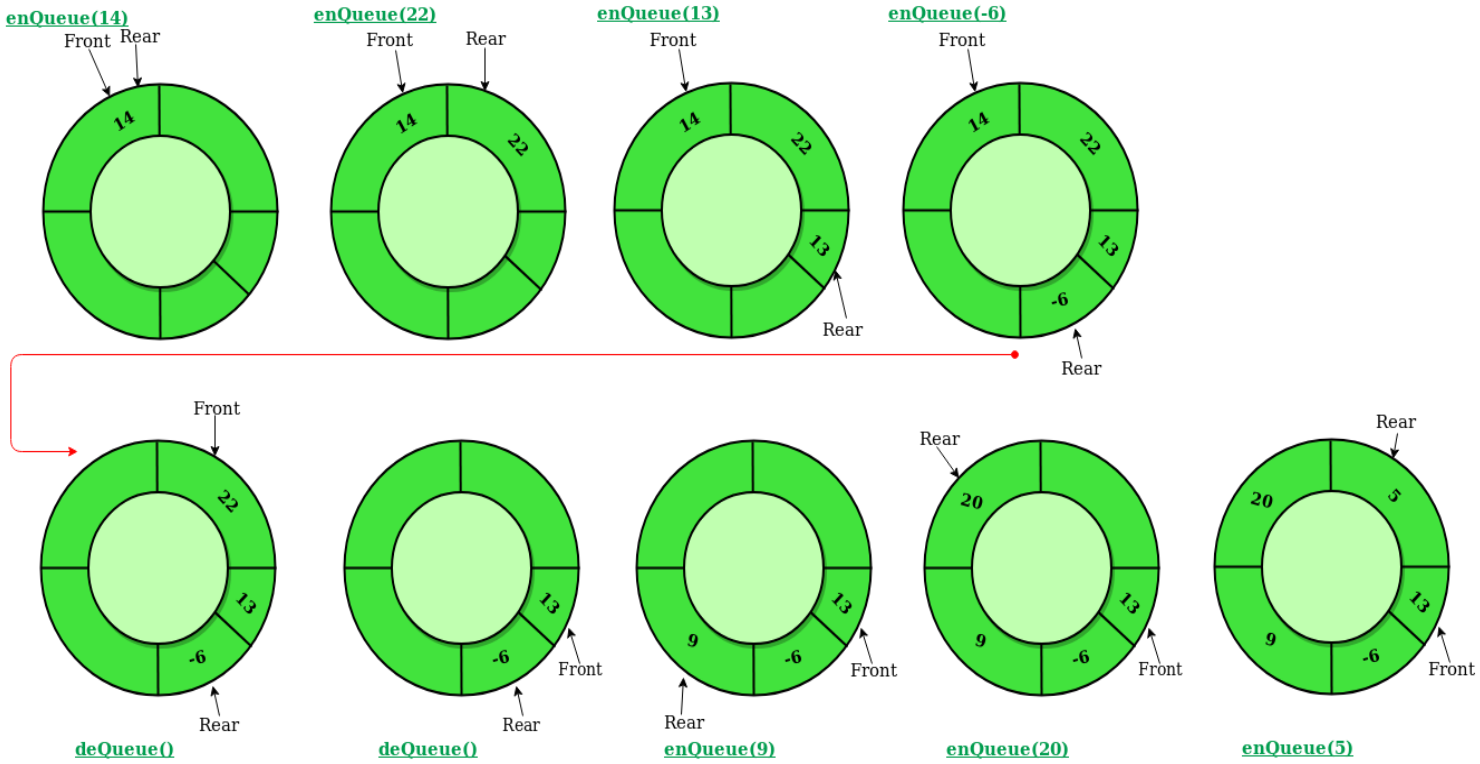
Operations on Circular Queue:

- **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`

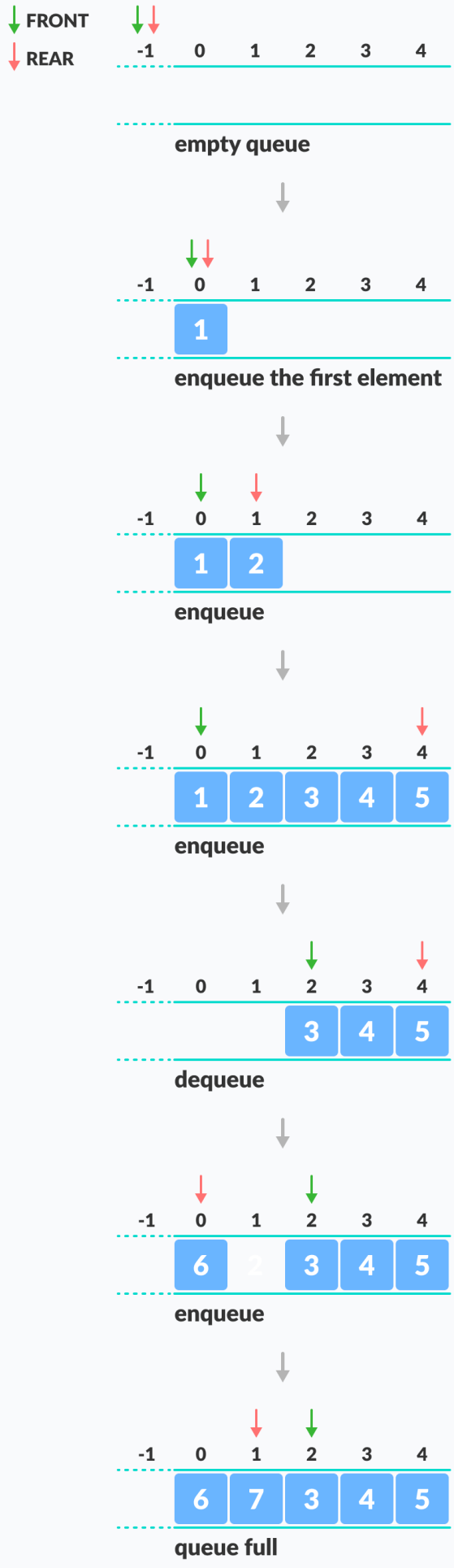
2. Dequeue Operation

- 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;
}
```

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.**

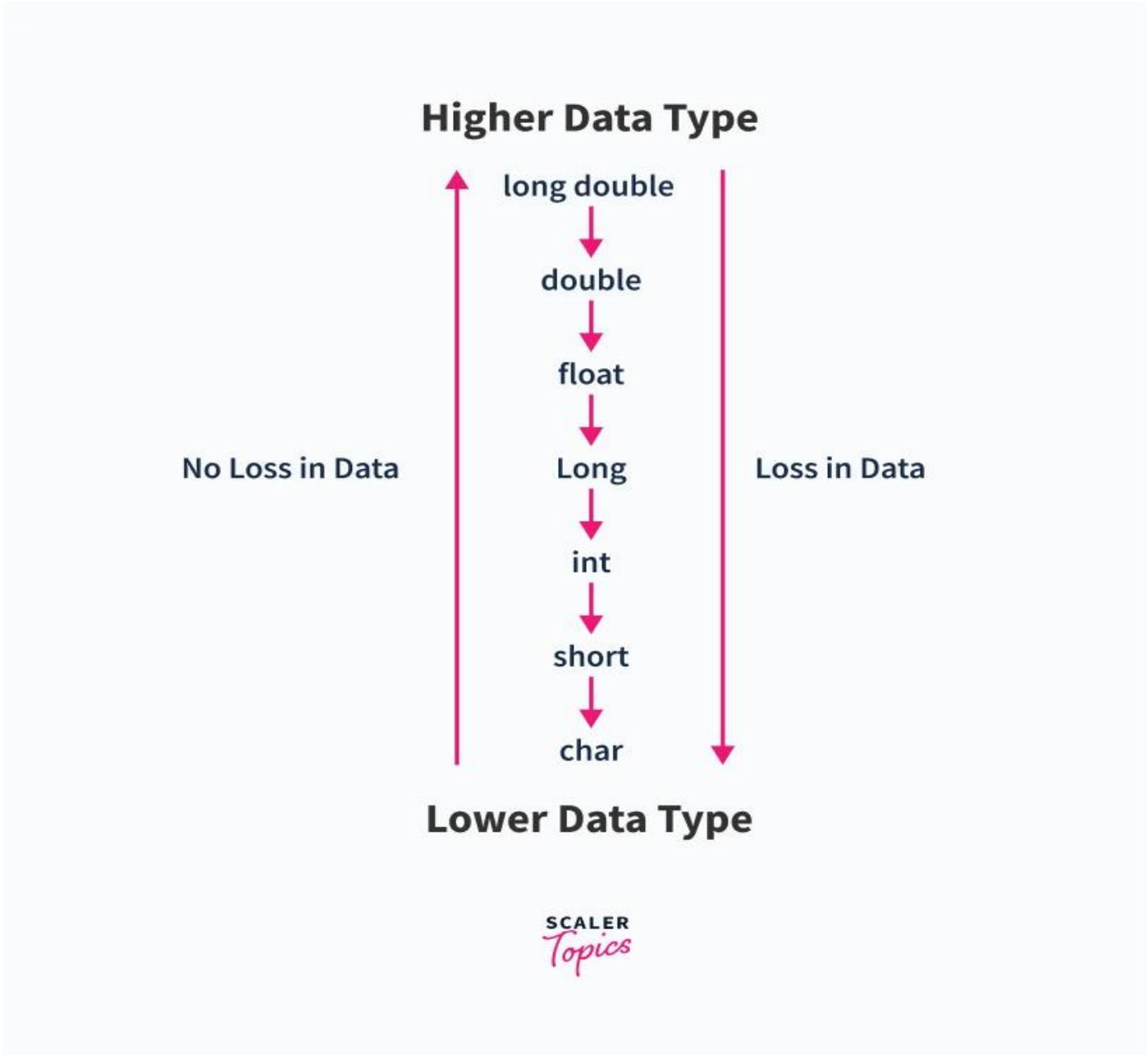
It means an implicit conversion automatically converts one data type into another type based on some predefined rules of the C++ compiler. Hence, it is also known as the **automatic type conversion**.

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
```