***Destructor:***

A destructor in C++ is a class member function that is implicitly invoked when the scope of a class object ends. Same as a constructor, which is also invoked implicitly when an object of a class is created. A destructor is used to free up the resources occupied by an object. The destructor function has the same name as that of a class, but a tilde (~) sign is used before writing the function's name.

***Virtual Destructor:***

Virtual Destructor in C++ is used to release or free the memory used by the child class (derived class) object when the child class object is being removed from the memory using the parent class's pointer object.

# Why We Use Virtual Destructor in C++?

As we know, a destructor is implicitly invoked when an object of a class goes out of scope, or the object's scope ends to free up the memory occupied by that object.

Due to early binding, when the object pointer of the parent class is deleted, which was pointing to the object of the derived class then, only the destructor of the parent class is invoked; it does not invoke the destructor of the child class, which leads to the problem of memory leak in our program.

So, When we use a Virtual destructor, i.e., a virtual keyword preceded by a tilde(~) sign and destructor name, inside the parent class, it makes sure that first the destructor of the child class should be invoked. And then, the destructor of the parent class is called so that it releases the memory occupied by both destructors.

***Note:*** *There is no concept of virtual constructors in C++.*

## Example 1: Delete Child Object Using Base Class Pointer Without a Virtual Destructor

```
#include <iostream>

using namespace std;

class Base
{
public:
  Base()
  {
        cout << "Constructor of Base class is called"<<endl;
  }
  ~Base()
```

```
  {
        cout << "Destructor of Base class is called"<<endl;
  }
};

//Inheriting features of the Base class into the Child class
class Child : public Base
{
public:
  Child()
  {
        cout << "Constructor of Child class is called"<<endl;
  }
  ~Child()
  {
        cout << "Destructor of Child class is called"<<endl;
  }
};

int main()
{
  Child *ch = new Child(); // Initializing the object of Child class.
  Base *b = ch;
  delete b;
  return 0;
}
```

OUTPUT:
Constructor of Base class is called
Constructor of Child class is called
Destructor of Base class is called

**Example Explanation:**

In the above program, we have used one parent class (Base) and a derived class (Child), inside which both constructors and destructors are defined in both classes.

When we delete an object of a derived class using a pointer of the parent class, it shows an undefined behavior because it does not have a virtual destructor. So, when we delete the object of the Child class to release the space occupied, it invokes the destructor of the base class, but the destructor of the child class is not invoked. As seen in the output, the destructor of the Child class is not invoked because the base class pointer can only remove the destructor of the base class, which causes the problem of memory leak in the program.

## Example 2: Delete Child Object Using a Base Class Pointer with a Virtual Destructor

//C++ program to demonstrate deletion

```
// of child class using base pointer
// using Virtual Destructor
#include <iostream>
using namespace std;

class Base
{
public:
  Base()
  {
        cout << "Constructor of Base class is called" << endl;
  }
  //Defining virtual destructor
  virtual ~Base()
  {
        // At last it will be printed
        cout << "Destructor of Base class is called" << endl;
  }
};

//Inheriting features of Base class in Child class
class Child : public Base
{
public:
  Child()
  {
        cout << "Constructor of Child class is called" << endl;
  }
  //It will be called before calling the Base class destructor
  ~Child()
  {
        cout << "Destructor of Child class is called" << endl;
  }
};

int main()
{
//   Base *b = new Child; //  object refers to the Base class.
  Child *ch = new Child();
  Base *b = ch;
  delete b;                 // Deleting the pointer object.
  return 0;
}
```

**Output:**

```
Constructor of Base class is called
Constructor of Child class is called
Destructor of Child class is called
Destructor of Base class is called
```

**Example Explanation:**

In the above program, a virtual destructor is used in the base class so that it calls the destructor of the child class before the base class's destructor is invoked. This releases the memory occupied and also resolves the problem of memory leaks.

# How Does Virtual Destructor Work?

Virtual destructor works in a way that ensures the correct order of invoking destructors when the object of the derived class goes out of scope or is deleted. If the order of invoking is incorrect, it will lead to a problem known as a memory leak.

Virtual destructor in C++ is mainly responsible for resolving the problem of memory leaks.

When we use a virtual destructor inside the base class, it will call the destructor of the child class, which will ensure that the child class's object should be deleted so there might be no memory leakage.

**Virtual Base Class:**

Virtual base classes in C++ are used to prevent multiple instances of a given class from 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 classes are inherited into another in a new class D as shown in figure below.

# What is Virtual Class?

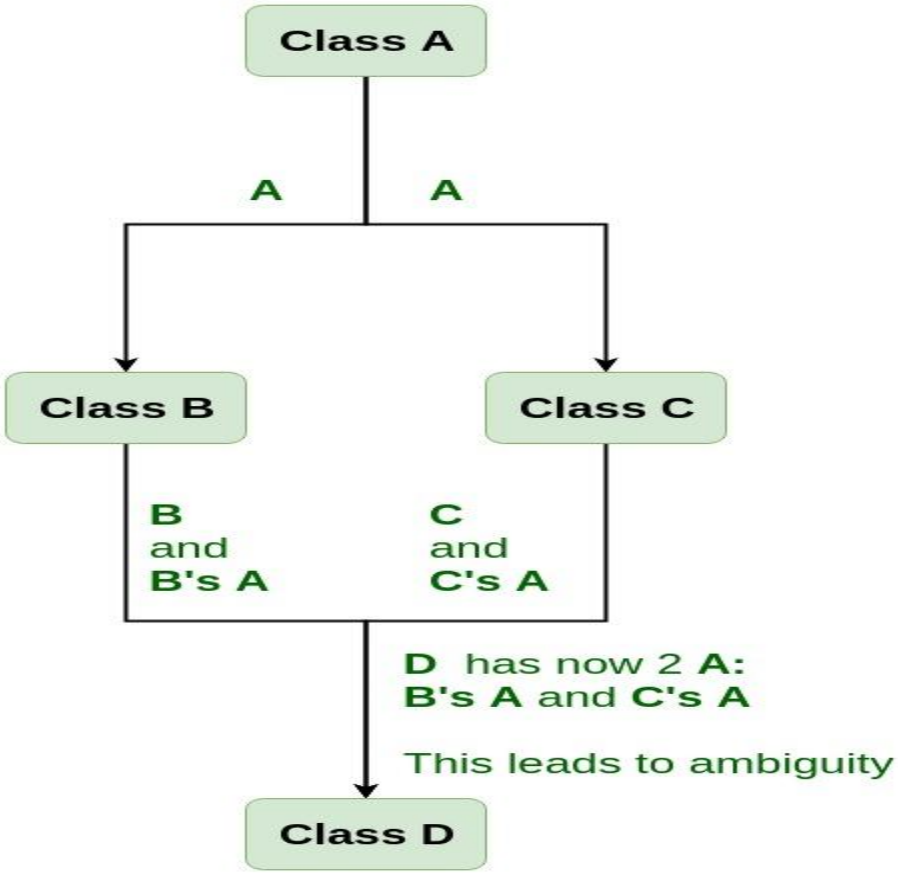
Virtual Class is defined by writing a keyword “virtual” in the derived classes, allowing only one copy of data to be copied to Class B and Class C (referring to the above example). It prevents multiple instances of a class appearing as a parent class in the inheritance hierarchy when multiple inheritances are used.

# Need for Virtual Base Class in C++

To prevent the error and let the compiler work efficiently, we've to use a virtual base class when multiple inheritances occur. It saves space and avoids ambiguity.

When a class is specified as a virtual base class, it prevents duplication of its data members. Only one copy of its data members is shared by all the base classes that use the virtual base class.

If a virtual base class is not used, all the derived classes will get duplicated data members. In this case, the compiler cannot decide which one to execute.




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 the compiler and it displays errors.

**Let us look at an example without a virtual base class in C++ and see what the output looks like:**

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

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

class B: public A {
};

class C: public A {
};

class D: public B, public C {
};

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

*OUTPUT:*

```
error: non-static member 'display' found in multiple base-class subobjects of type 'A':
        class D -> class B -> class A
        class D -> class C -> class A
        object.display();
        ^
note: member found by ambiguous name lookup
        void display()
        ^
1 error generated.
```

In the above example, we create a Class A and then two of its derived classes, Class B and Class C. Class A has a method that prints out a statement. All the derived classes must have inherited data members from Class A.

Next, we declare Class D, which inherits class B and class C. Since Classes B and C are child classes of A and then D is the child class of B and C, Class D inherits data members of Class A from both B and C. Hence, duplication occurs, and the compiler doesn't know what to execute and throws an error.

**Output After removing line object.display():**

Constructor A
Constructor A

**Explanation:** If we remove the line object.display() in main, the program will compile successfully, and the above output will be printed. It means two objects of class A were created, one from B and the other from C. That's why the call is ambiguous.

But this situation is avoided if the virtual base class is used.

# How to Declare Virtual Base Class in C++?

**Syntax**

If Class A is considered as the base class and Class B and Class C are considered as the derived classes of A.

**Note:** The word "virtual" can be written before or after the word "public".

```
class B: virtual public A {
  // statement 1
};
class C: public virtual A {
  // statement 2
};
```

**Let us see an example:**

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

class A {
  public:
    A() // Constructor
    {
        cout << "Constructor A\n";
    }
};
```

```
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.

  return 0;
}
```

**Output:**

```
Constructor A
```

In this case, we are using a virtual base class in C++, so only one copy of data from Class A was inherited to Class D; hence, the compiler will be able to print the output.

When we mention the base class as virtual, we avoid the situation of duplication and let the derived classes get only one copy of the data.

### There are a few details that one needs to remember.

1. Virtual base classes are always created before non-virtual base classes. This ensures all bases are created before their derived classes.
2. Note that classes B and C still have calls to class A, but they are simply ignored when creating an object of class D. If we are creating an object of class B or C, then the constructor of A will be called.
3. If a class inherits one or more classes with virtual parents, the most derived class is responsible for constructing the virtual base class. Here, class D is responsible for creating class A object.

```
#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

## ***Friend Function and class:***

A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.

We can declare a friend class in C++ by using the **friend** keyword.

**Syntax:**

```
friend class class_name;    // declared in the base class
```

# Friend Function

Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are the non-member functions that can access and manipulate the private and protected members of the class for they are declared as friends.

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the friend keyword inside the body of the class.

**Syntax:**

```
friend return_type function_name (arguments);    // for a global function

        or

friend return_type class_name::function_name (arguments);    // for a member function of another class
```

# Advantages of Friend Functions

- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

# Disadvantages of Friend Functions

- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- Friend functions cannot do any run-time polymorphism in their members.

# Important Points About Friend Functions and Classes

1. Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming.
2. Friendship is **not mutual**. If class A is a friend of B, then B doesn't become a friend of A automatically.
3. Friendship is not inherited.
4. The concept of friends is not in Java.

## Features of Friend Functions

- A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to **access** the **private and protected data** of a class.
- A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword "**friend**" inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword "friend" is placed only in the function declaration of the friend function and **not** in the **function definition or call.**

- A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.
- A friend function can be declared in any section of the class i.e. public or private or protected.