

OOP concepts and Interview questions

Advantages of OOP

1. Modularity for easier troubleshooting

Something has gone wrong, and you have no idea where to look. When working with object-oriented programming languages, you know exactly where to look. “Oh, the car object broke down? The problem must be in the Car class!” You don’t have to muck through anything else.

That’s the beauty of encapsulation. Objects are self-contained, and each bit of functionality does its own thing while leaving the other bits alone. Also, this modality allows an IT team to work on multiple objects simultaneously while minimizing the chance that one person might duplicate someone else’s functionality.

2. Reuse of code through inheritance

Suppose that in addition to your Car object, one colleague needs a RaceCar object, and another needs a Limousine object. Everyone builds their objects separately but discover commonalities between them. In fact, each object is really just a different kind of Car. This is where the inheritance technique saves time: Create one generic class (Car), and then define the subclasses (RaceCar and Limousine) that are to inherit the generic class’s traits.

Of course, Limousine and RaceCar still have their unique attributes and functions. If the RaceCar object needs a method to “fireAfterBurners” and the Limousine object requires a Chauffeur, each class could implement separate functions just for itself. However, because both classes inherit key aspects from the Car class, for example the “drive” or

“fillUpGas” methods, your inheriting classes can simply reuse existing code instead of writing these functions all over again.

What if you want to make a change to all Car objects, regardless of type? This is another advantage of the OO approach. Simply make a change to your Car class, and all car objects will simply inherit the new code.

3. Flexibility through polymorphism

Riffing on this example, you now need just a few drivers, or functions, like “driveCar,” “driveRaceCar” and “DriveLimousine.” RaceCarDrivers share some traits with LimousineDrivers, but other things, like RaceHelmets and BeverageSponsorships, are unique.

This is where object-oriented programming’s sweet polymorphism comes into play.

Because a single function can shape-shift to adapt to whichever class it’s in, you could create one function in the parent Car class called “drive” — not “driveCar” or “driveRaceCar,” but just “drive.” This one function would work with the RaceCarDriver, LimousineDriver, etc. In fact, you could even have “raceCar.drive(myRaceCarDriver)” or “limo.drive(myChauffeur).”

4. Effective problem solving

A language like C has an amazing legacy in programming history, but writing software in a top-down language is like playing Jenga while wearing mittens. The more complex it gets, the greater the chance it will collapse. Meanwhile, writing a functional-style program in a language like Haskell or ML can be a chore.

Object-oriented programming is often the most natural and pragmatic approach, once you get the hang of it. OOP languages allow you to break down your software into bite-sized problems that you then can solve — one object at a time.

This isn't to say that OOP is the One True Way. However, the advantages of object-oriented programming are many. When you need to solve complex programming challenges and want to add code tools to your skill set, OOP is your friend — and has much greater longevity and utility than Pac-Man or parachute pants.

Struct vs class

1. **Class** can create a subclass that will inherit parent's properties and methods, whereas **Structure** does not support the inheritance.
 2. A **class** has all members private by default. A **struct** is a **class** where members are public by default.
 3. **Classes** allow you to perform cleanup (garbage collector) before the object is deallocated because the garbage collector works on heap memory. Objects are usually deallocated when instance is no longer referenced by other code.
Structures can not be garbage collector so no efficient memory management.
 4. Size of the empty **class** is 1 Byte whereas Size of empty **structure** is 0 Bytes.
-

Const variables, objects and functions

If you make any variable constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while they are declared.

const data members of a class may be declared as const . Such a data member must be initialized by the constructor using an initialization list. Once initialized, a const data member may never be modified, not even in the constructor or destructor.

The const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes. So the constructor and destructor can modify the object, but other methods of the class can't.

Only const methods can be called for a const object. Objects that are not const can call either const or non-const methods.

Constructors and destructors can never be declared as const. They are always allowed to modify an object even if the object is const.

Static variables and functions

Static Variables in a class: As the variables declared as static are initialized only once as they are allocated space in separate static storage, so the static variables in a class are shared by the objects. There can not be multiple copies of the same static variables for different objects. Also because of this reason static variables can not be initialized using constructors.

Static functions in a class: Just like the static data members or static variables inside the class, static member functions also do not depend on the object of the class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator.

Class objects as static: Just like variables, objects also when declared as static have a scope till the lifetime of the program.

For further details:

<https://www.geeksforgeeks.org/static-keyword-cpp/#:~:text=Static%20variables%20in%20a%20class.static%20variables%20for%20different%20objects.>

Access specifiers

Access specifiers(or access modifiers) are used to set the accessibility of classes, methods and variables.

Types of modifiers

- Public
- Private
- Protected
- Internal
- Protected Internal

Public

Can be accessible outside the class (anywhere) in the same assembly or another assembly.

Private

Accessible only in the same class.

Protected


Accessible in the same class or in a derived class of that class.

Internal

Accessible anywhere in the code within the assembly. But not accessible in another assembly.

Protected Internal

Accessible anywhere in the code within the assembly. And accessible only in derived class in another assembly.

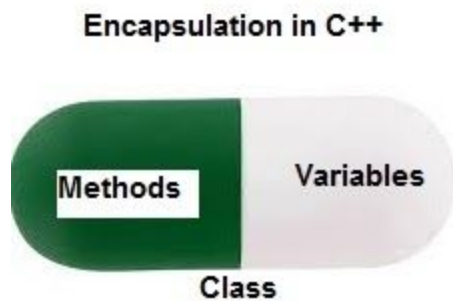


Encapsulation in C++

In normal terms **Encapsulation** is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. 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”.



Encapsulation also leads to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the sections like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and **access modifiers**. Look at the below program:

```
// c++ program to explain
// Encapsulation

#include<iostream>
using namespace std;

class Encapsulation
{
    private:
        // data hidden from outside world
        int x;

    public:
        // function to set value of
        // variable x
        void set(int a)
        {
            x =a;
        }

        // function to return value of
        // variable x
        int get()
        {
            return x;
        }
};

// main function
int main()
{
    Encapsulation obj;

    obj.set(5);

    cout<<obj.get();
    return 0;
}
```

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions `get()` and `set()` which are present inside the class. Thus we can say that here, the variable **x** and the functions `get()` and `set()` are binded together which is nothing but encapsulation.

Role of access specifiers in encapsulation

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public** access specifier

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 accelerators will increase the speed of the car or applying brakes will stop the car

but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

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

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the `pow()` method present in the `math.h` header file. Whenever we need to calculate 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 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 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:

```
#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 internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.



Encapsulation VS Abstraction

The most important difference between Abstraction and Encapsulation is that Abstraction solves the problem at design level while Encapsulation solves it implementation level.

Abstraction is about hiding unwanted details while giving out most essential details, while **Encapsulation** means hiding the code **and** data into a single unit e.g. class or method to protect inner workings of an object from the outside world.

Constructor and destructor behavior in classes

C++ constructor call order will be from top to down that is from base **class** to derived **class** and c++ destructor call order will be in reverse order.

Virtual destructor

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following a program results in undefined behavior.

```
// CPP program without virtual destructor
// causing undefined behavior
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following:

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
/ A program with virtual destructor
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Output:

```
Constructing base
Constructing derived
Destructing derived
```

Destructing base

Why & is it used in the copy constructor?

It is necessary to pass an object as reference and not by value because if you pass it by value its copy is constructed using the copy constructor. This means the copy constructor would call itself to make copy. This process will go on until the compiler runs out of memory.

Why is pointer not used in the copy constructor?

Passing by references ensures an actual object is passed to the copy constructor, whilst a pointer can have NULL value, and make the constructor fail. There are objects where a copy is definitely not "good", and there are other cases, like a big math class, where copy constructors are definitely a good thing.

What are three situations where a copy constructor may be called?

The following are the cases when a copy constructor is called.

- 1)When instantiating one object and initializing it with values from another object.
- 2)When passing an object by value.
- 3)When an object is returned from a function by value.

Copy constructor vs assignment operator

The Copy constructor and the assignment operators are used to initialize one object to another object. The main difference between them is that the copy constructor creates a separate memory block for the new object. But the assignment operator does not make new memory space. It uses the reference variable to point to the previous memory block.

Copy Constructor (Syntax)

```
classname (const classname &obj) {  
    // body of constructor  
}
```

Assignment Operator (Syntax)

```
classname Ob1, Ob2;  
Ob2 = Ob1;
```

Copy Constructor vs Assignment Operator

Copy Constructor	Assignment Operator
The Copy constructor is basically an overloaded constructor	Assignment operator is basically an operator.
This initializes the new object with an already existing object	This assigns the value of one object to another object both of which already exists.
Copy constructor is used when a new object is created with some existing object	This operator is used when we want to assign an existing object to a new object.
Both the objects uses separate memory locations.	One memory location is used but different reference variables are pointing to the same location.
If no copy constructor is defined in the class, the compiler provides one.	If the assignment operator is not overloaded then bitwise copy will be made

Inheritance

Inheritance is one of the key features of Object-oriented programming in C++. It allows the user to create a new class (derived class) from an existing class(base class). The

derived class inherits all the features from the base class and can have additional features of its own.

Why should inheritance be used?

Suppose, in your game, you want three characters - a maths teacher, a footballer and a businessman.

Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can teach maths, a footballer can play football and a businessman can run a business.

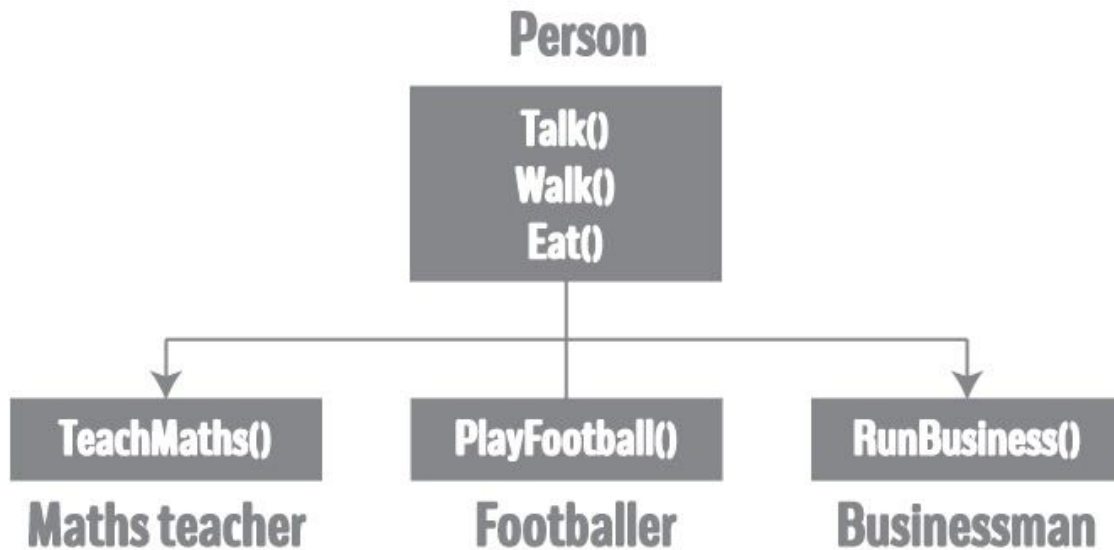
You can individually create three classes who can walk, talk and perform their special skill as shown in the figure below.



In each of the classes, you would be copying the same code for walk and talk for each character.

If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes.

It'd be a lot easier if we had a Person class with basic features like talk, walk, eat, sleep, and add special skills to those features as per our characters. This is done using inheritance.



Using inheritance, now you don't implement the same code for walk and talk for each class. You just need to inherit them.

So, for Maths teacher (derived class), you inherit all features of a Person (base class) and add a new feature TeachMaths. Likewise, for a footballer, you inherit all the features of a Person and add a new feature PlayFootball and so on.

This makes your code cleaner, understandable and extendable.

It is important to remember: When working with inheritance, each derived class should satisfy the condition whether it "is a" base class or not. In the

example above, Maths teacher is a Person, Footballer is a Person. You cannot have: Businessman is a Business.

Implementation of Inheritance in C++ Programming

```
class Person
{
    ... ..
};

class MathsTeacher : public Person
{
    ... ..
};

class Footballer : public Person
{
    .... ..
};
```

In the above example, class `Person` is a base class and classes `MathsTeacher` and `Footballer` are the derived from *Person*.

The derived class appears with the declaration of a class followed by a colon, the keyword `public` and the name of base class from which it is derived.

Since, `MathsTeacher` and `Footballer` are derived from `Person`, all data member and member function of `Person` can be accessible from them.

Example: Inheritance in C++ Programming

Create game characters using the concept of inheritance.

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

```

class Person
{
    public:
        string profession;
        int age;

        Person(): profession("unemployed"), age(16) { }
        void display()
        {
            cout << "My profession is: " << profession << endl;
            cout << "My age is: " << age << endl;
            walk();
            talk();
        }
        void walk() { cout << "I can walk." << endl; }
        void talk() { cout << "I can talk." << endl; }
};

```

// MathsTeacher class is derived from base class Person.

```

class MathsTeacher : public Person
{
    public:
        void teachMaths() { cout << "I can teach Maths." << endl; }
};

```

// Footballer class is derived from base class Person.

```

class Footballer : public Person
{
    public:
        void playFootball() { cout << "I can play Football." << endl; }
};

```

```

int main()
{
    MathsTeacher teacher;
    teacher.profession = "Teacher";
    teacher.age = 23;
    teacher.display();
    teacher.teachMaths();

    Footballer footballer;
    footballer.profession = "Footballer";
    footballer.age = 19;
    footballer.display();
}

```

```
        footballer.playFootball();  
  
        return 0;  
    }
```

output

```
My profession is: Teacher  
My age is: 23  
I can walk.  
I can talk.  
I can teach Maths.  
My profession is: Footballer  
My age is: 19  
I can walk.  
I can talk.  
I can play Football.
```

In this program, `Person` is a base class, while `MathsTeacher` and `Footballer` are derived from `Person`.

`Person` class has two data members - `profession` and `age`. It also has two member functions - `walk()` and `talk()`.

Both `MathsTeacher` and `Footballer` can access all data members and member functions of `Person`.

However, `MathsTeacher` and `Footballer` have their own member functions as well: `teachMaths()` and `playFootball()` respectively. These functions are only accessed by their own class.

In the `main()` function, a new `MathsTeacher` object `teacher` is created.

Since, it has access to `Person`'s data members, `profession` and `age` of `teacher` is set. This data is displayed using the `display()` function defined in the `Person`

class. Also, the `teachMaths()` function is called, defined in the `MathsTeacher` class.

Likewise, a new `Footballer` object `footballer` is also created. It has access to `Person`'s data members as well, which is displayed by invoking the `display()` function. The `playFootball()` function only accessible by the `footballer` is called then after.

Access specifiers in Inheritance

When creating a derived class from a base class, you can use different access specifiers to inherit the data members of the base class.

These can be public, protected or private.

In the above example, the base class `Person` has been inherited `public`-ly by `MathsTeacher` and `Footballer`.

Public, private and protected inheritance

You can declare a derived `class` from a base class with different access control, i.e., public `inheritance`, protected inheritance or private inheritance.

```
#include <iostream>

using namespace std;
```

```
class base

{

.....

};

class derived : access_specifier base

{

.....

};
```

Note: Either `public`, `protected` or `private` keyword is used in place of `access_specifier` term used in the above code.

Example of public, protected and private inheritance in C++

```
class base

{

    public:

        int x;

    protected:

        int y;
```

```
private:

    int z;

};

class publicDerived: public base

{

    // x is public

    // y is protected

    // z is not accessible from publicDerived

};

class protectedDerived: protected base

{

    // x is protected

    // y is protected

    // z is not accessible from protectedDerived

};

class privateDerived: private base
```



```
{  
  
    // x is private  
  
    // y is private  
  
    // z is not accessible from privateDerived  
  
}
```

In the above example, we observe the following things:

- `base` has three member variables: `x`, `y` and `z` which are `public`, `protected` and `private` member respectively.
- `publicDerived` inherits variables `x` and `y` as `public` and `protected`. `z` is not inherited as it is a `private` member variable of `base`.
- `protectedDerived` inherits variables `x` and `y`. Both variables become `protected`. `z` is not inherited

If we derive a class `derivedFromProtectedDerived` from `protectedDerived`, variables `x` and `y` are also inherited to the derived class.

- `privateDerived` inherits variables `x` and `y`. Both variables become `private`. `z` is not inherited

If we derive a class `derivedFromPrivateDerived` from `privateDerived`, variables `x` and `y` are not inherited because they are `private` variables of `privateDerived`.

Accessibility in Public Inheritance

Accessibility	private variables	protected variables	public variables
Accessible from own class?	yes	yes	yes
Accessible from derived class?	no	yes	yes
Accessible from 2nd derived class?	no	yes	yes

Accessibility in Protected Inheritance

Accessibility	private variables	protected variables	public variables
---------------	-------------------	---------------------	------------------

Accessible from own class?	yes	yes	yes
Accessible from derived class?	no	yes	yes (inherited as protected variables)
Accessible from 2nd derived class?	no	yes	yes

Accessibility in Private Inheritance

Accessi bility	privat e varia bles	protected variables	public variables
Accessi ble from	yes	yes	yes

own class?			
Accessible from derived class?	no	yes (inherited as private variables)	yes (inherited as private variables)
Accessible from 2nd derived class?	no	no	no

Diamond problem (multiple inheritance)

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class.

The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

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

class A
{
public:
    A() { cout << "A's constructor called" << endl; }
};

class B
{
public:
    B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A // Note the order
{
public:
    C() { cout << "C's constructor called" << endl; }
};

int main()
{
    C c;
    return 0;
}
```

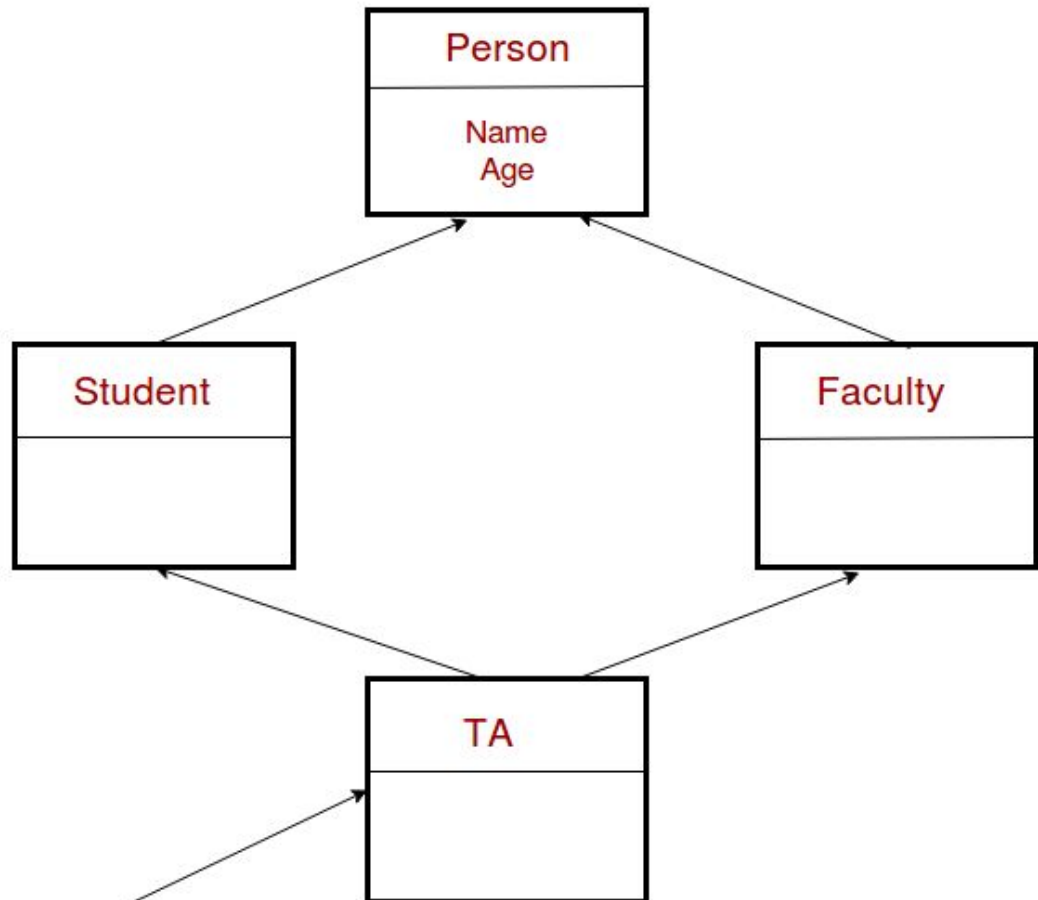
Output:

```
B's constructor called
A's constructor called
C's constructor called
```

The destructors are called in reverse order of constructors.

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



Name and Age needed only once

For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
    // Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" <<
endl; }
};

class Faculty : public Person {
    // data members of Faculty
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : public Person {
    // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
```

```

    }
};

int main() {
    TA ta1(30);
}
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called

```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. *The solution to this problem is 'virtual' keyword.* We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.


```

#include<iostream>
using namespace std;
class Person {
public:
    Person(int x)  { cout << "Person::Person(int ) called" <<
endl;  }
    Person()      { cout << "Person::Person() called" << endl;  }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x)    {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x) , Faculty(x)    {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main()  {
    TA ta1(30);
}

```

Output:

```

Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called

```

```
TA::TA(int ) called
```

Shallow vs deep copy

A shallow copy of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied, but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disastrous consequences.

Deep copies need ...

If an object has pointers to dynamically allocated memory, and the dynamically allocated memory needs to be copied when the original object is copied, then a deep copy is required.

A class that requires deep copies generally needs:

- A [constructor](#) to either make an initial allocation or set the pointer to NULL.
 - A [destructor](#) to delete the dynamically allocated memory.
 - A [copy constructor](#) to make a copy of the dynamically allocated memory.
 - An [overloaded assignment operator](#) to make a copy of the dynamically allocated memory.
-

Friend function and classes

One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmers to write long and complex codes. So, there is a mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

Friend Function in C++

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

The compiler knows a given function is a friend function by the use of the keyword friend.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Declaration of friend function in C++

```
class class_name
{
    ... ..
    friend return_type function_name(argument/s);
    ... ..
}
```

Now, you can define the friend function as a normal function to access the data of the class. No friend keyword is used in the definition.

```
class className
{
    ... ..
    friend return_type functionName(argument/s);
    ... ..
}

return_type functionName(argument/s)
{
    ... ..
    // Private and protected data of className can be accessed from
    // this function because it is a friend function of className.
    ... ..
}
```

Example 1: Working of friend Function

```
/* C++ program to demonstrate the working of friend function.*/
#include <iostream>
```

```

using namespace std;

class Distance
{
    private:
        int meter;
    public:
        Distance(): meter(0) { }
        //friend function
        friend int addFive(Distance);
};

// friend function definition
int addFive(Distance d)
{
    //accessing private data from non-member function
    d.meter += 5;
    return d.meter;
}

int main()
{
    Distance D;
    cout<<"Distance: "<< addFive(D);
    return 0;
}

```

Output

Distance: 5

Here, friend function addFive() is declared inside Distance class. So, the private data meter can be accessed from this function.

Though this example gives you an idea about the concept of a friend function, it doesn't show any meaningful use.

A more meaningful use would be when you need to operate on objects of two different classes. That's when the friend function can be very helpful.

You can definitely operate on two objects of different classes without using the friend function but the program will be long, complex and hard to understand.

Example 2: Addition of members of two different classes using friend Function

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

// forward declaration
class B;
class A {
    private:
        int numA;
    public:
        A(): numA(12) { }
        // friend function declaration
        friend int add(A, B);
};

class B {
    private:
        int numB;
    public:
        B(): numB(1) { }
        // friend function declaration
        friend int add(A , B);
};

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

int main()
{
    A objectA;
    B objectB;
    cout<<"Sum: "<< add(objectA, objectB);
    return 0;
}
```

Output

Sum: 13

In this program, classes A and B have declared add() as a friend function. Thus, this function can access private data of both class.

Here, add() function adds the private data numA and numB of two objects objectA and objectB, and returns it to the main function.

To make this program work properly, a forward declaration of a class class B should be made as shown in the above example.

This is because class B is referenced within the class A using code: friend int add(A , B);.

Friend Class in C++

Similarly, like a friend function, a class can also be made a friend of another class using keyword friend. For example:

```
... ..  
class B;  
class A  
{  
    // class B is a friend class of class A  
    friend class B;  
    ... ..  
}  
  
class B  
{  
    ... ..  
}
```

When a class is made a friend class, all the member functions of that class becomes friend functions.

In this program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.

Remember, friend relation in C++ is only granted, not taken.



Operator overloading

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows –

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below

```
#include <iostream>
```

```
using namespace std;
```

```
class Box {
public:
    double getVolume(void) {
        return length * breadth * height;
    }
    void setLength( double len ) {
        length = len;
    }
    void setBreadth( double bre ) {
        breadth = bre;
    }
    void setHeight( double hei ) {
        height = hei;
    }
}
```

```
// Overload + operator to add two Box objects.
```

```
Box operator+(const Box& b) {
    Box box;
```

```

    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
}

```

```
private:
```

```

    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box

```

```
};
```

```
// Main function for the program
```

```
int main() {
```

```

    Box Box1;          // Declare Box1 of type Box
    Box Box2;          // Declare Box2 of type Box
    Box Box3;          // Declare Box3 of type Box
    double volume = 0.0; // Store the volume of a box here

```

```
// box 1 specification
```

```

Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

```

```
// box 2 specification
```

```

Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

```

```
// volume of box 1
```

```

volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume << endl;

```

```
// volume of box 2
```

```

volume = Box2.getVolume();
cout << "Volume of Box2 : " << volume << endl;

```

```
// Add two object as follows:
```

```
Box3 = Box1 + Box2;
```

```
// volume of box 3
```

```

volume = Box3.getVolume();
cout << "Volume of Box3 : " << volume << endl;

```



```
    return 0;
}
```

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

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

Following is the list of operators, which can not be overloaded –

::	.*	.	?:
----	----	---	----

Stream extraction and insertion operator overloading

In C++, stream insertion operator "<<" is used for output and extraction operator ">>" is used for input.

We must know the following things before we start overloading these operators.

- 1) cout is an object of ostream class and cin is an object istream class
- 2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friends.

Why must these operators be overloaded as global?

In operator overloading, if an operator is overloaded as a member, then it must be a member of the object on the left side of the operator. For example, consider the statement "ob1 + ob2" (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload '+' in class of 'ob1' or make '+' a global function.

The operators '<<' and '>>' are called like 'cout << ob1' and 'cin >> ob1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

Following is complete C++ program to demonstrate overloading of <> operators.

```

#include <iostream>
using namespace std;

class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {   real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Complex
&c);
    friend istream & operator >> (istream &in, Complex &c);
};

```

```

ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

```

```

istream & operator >> (istream &in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary Part ";
    in >> c.imag;
    return in;
}

```

```

int main()
{
    Complex c1;
    cin >> c1;
    cout << "The complex object is ";
    cout << c1;
    return 0;
}

```

Output:

```
Enter Real Part 10
Enter Imaginary Part 20
The complex object is 10+i20
```

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function print() is being used to print different data types –

[Live Demo](#)

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

class printData {
public:
    void print(int i) {
        cout << "Printing int: " << i << endl;
    }
    void print(double f) {
        cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
        cout << "Printing character: " << c << endl;
    }
};

int main(void) {
    printData pd;

    // Call print to print integer
    pd.print(5);

    // Call print to print float
```

```

pd.print(500.263);

// Call print to print character
pd.print("Hello C++");

return 0;
}

```

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

```

Printing int: 5
Printing float: 500.263
Printing character: Hello C++

```

Function overloading

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function `myfuncn(int a, float b)` is (int, float) which is different from the function `myfuncn(float a, int b)` parameter list (float, int). Function overloading is a **compile-time polymorphism**.

Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.

```

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

```

The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have different **type**, **number** or **sequence** of parameters.

For example:

These two functions have different parameter **type**:

```
sum(int num1, int num2)
sum(double num1, double num2)
```

These two have different **number** of parameters:

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

These two have different **sequence** of parameters:

```
sum(int num1, double num2)
sum(double num1, int num2)
```

All of the above three cases are valid case of overloading. We can have any number of functions, just remember that the parameter list should be different.

For example:

```
int sum(int, int)
double sum(int, int)
```

This is not allowed as the parameter list is same. Even though they have different return types, its not valid.

Function overloading Example

Lets take an example to understand function overloading in C++.

```
#include <iostream>
using namespace std;
class Addition {
public:
    int sum(int num1,int num2) {
        return num1+num2;
    }
    int sum(int num1,int num2, int num3) {
        return num1+num2+num3;
    }
};
int main(void) {
    Addition obj;
    cout<<obj.sum(20, 15)<<endl;
    cout<<obj.sum(81, 100, 10);
    return 0;
}
```

}

Output:

35

191

Function overloading Example 2

As I mentioned in the beginning of this guide that functions having different return types and same parameter list cannot be overloaded. However if the functions have different parameter lists then they can have same or different return types to be eligible for overloading. In short the return type of a function does not play any role in function overloading. All that matters is the parameter list of function.

```
#include <iostream>
using namespace std;
class DemoClass {
public:
    int demoFunction(int i) {
        return i;
    }
    double demoFunction(double d) {
        return d;
    }
};
int main(void) {
    DemoClass obj;
    cout<<obj.demoFunction(100)<<endl;
    cout<<obj.demoFunction(5005.516);
    return 0;
}
```

Output:

100

5006.52

Advantages of Function overloading

The main advantage of function overloading is to improve the **code readability** and allows **code reusability**. In the example 1, we have seen how we were able to have more than one function for the same task (addition) with different parameters, this allowed us to add two integer numbers as well as three

integer numbers, if we wanted we could have some more functions with same name and four or five arguments.

Imagine if we didn't have function overloading, we either have the limitation to add only two integers or we had to write different name functions for the same task addition, this would reduce the code readability and reusability.

Function overriding

Function overriding is a feature that allows us to have the same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

Function Overriding Example

To override a function you must have the same signature in child class. By signature I mean the data type and sequence of parameters. Here we don't have any parameter in the parent function so we didn't use any parameter in the child function.

```
#include <iostream>
using namespace std;
class BaseClass {
public:
    void disp() {
        cout<<"Function of Parent Class";
    }
};
class DerivedClass: public BaseClass{
public:
    void disp() {
```



```

        cout<<"Function of Child Class";
    }
};

int main() {
    DerivedClass obj = DerivedClass();
    obj.disp();
    return 0;
}

```

Output:

```
Function of Child Class
```

Note: In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.

How to call overridden function from the child class

As we have seen above that when we make the call to function (involved in overriding), the child class function (overriding function) gets called. What if you want to call the overridden function by using the object of child class. You can do that by creating the child class object in such a way that the reference of parent class points to it. Lets take an example to understand it.

```

#include <iostream>

using namespace std;

class BaseClass {
public:
    void disp() {
        cout<<"Function of Parent Class";
    }
};

class DerivedClass: public BaseClass{
public:
    void disp() {

```

```

        cout<<"Function of Child Class";
    }
};

int main() {
    /* Reference of base class pointing to
    * the object of child class.
    */
    BaseClass obj = DerivedClass();
    obj.disp();
    return 0;
}

```

Output:

Function of Parent Class

If you want to call the Overridden function from overriding function then you can do it like this:

```
parent_class_name::function_name
```

To do this in the above example, we can write following statement in the disp() function of child class:

```
BaseClass::disp();
```

Function overloading vs Function overriding

1. **Inheritance:** Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.

2. **Function Signature:** Overloaded functions must differ in function signature i.e. either number of parameters or type of parameters should differ. In overriding, function signatures must be the same.
3. **Scope of functions:** Overridden functions are in different scopes; whereas overloaded functions are in the same scope.
4. **Behavior of functions:** Overriding is needed when a derived class function has to do some added or different job than the base class function.
Overloading is used to have same name functions which behave differently depending upon parameters passed to them.

Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes –

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

class Shape {
protected:
    int width, height;

public:
    Shape( int a = 0, int b = 0) {
        width = a;
        height = b;
    }
}
```

```

    int area() {
        cout << "Parent class area :" << endl;
        return 0;
    }
};

class Rectangle: public Shape {
public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
        cout << "Rectangle class area :" << endl;
        return (width * height);
    }
};

class Triangle: public Shape {
public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
        cout << "Triangle class area :" << endl;
        return (width * height / 2);
    }
};

// Main function for the program
int main() {
    Shape *shape;
    Rectangle rec(10,7);
    Triangle tri(10,5);

    // store the address of Rectangle
    shape = &rec;

    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;

    // call triangle area.

```

```

    shape->area();

    return 0;
}

```

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

```

Parent class area :
Parent class area :

```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this –

```

class Shape {
protected:
    int width, height;

public:
    Shape( int a = 0, int b = 0) {
        width = a;
        height = b;
    }
    virtual int area() {
        cout << "Parent class area :" <<endl;
        return 0;
    }
};

```

After this slight modification, when the previous example code is compiled and executed, it produces the following result –

```

Rectangle class area
Triangle class area

```

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

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.

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.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     int x=5;
6.     public:
7.     void display()
8.     {
9.         std::cout << "Value of x is : " << x<<std::endl;
10.    }
11.};
12.class B: public A
13.{
14.    int y = 10;
15.    public:
16.    void display()
17.    {
18.        std::cout << "Value of y is : " <<y<< std::endl;
19.    }
20.};
21.int main()
22.{
23.    A *a;
24.    B b;
25.    a = &b;
26.    a->display();
```

```

27.   return 0;
28. }

```

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.

virtual function Example

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

```

1. #include <iostream>
2. {
3.   public:
4.   virtual void display()
5.   {
6.     cout << "Base class is invoked"<<endl;
7.   }
8. };
9. class B:public A
10. {
11.   public:
12.   void display()
13.   {
14.     cout << "Derived Class is invoked"<<endl;
15.   }
16. };
17. int main()
18. {
19.   A* a;   //pointer of base class
20.   B b;    //object of derived class

```



```
21. a = &b;  
22. a->display(); //Late Binding occurs  
23. }
```

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:

1. **virtual void** display() = 0;

Pure virtual function Example

Let's see a simple example:

1. **#include <iostream>**
2. **using namespace** std;
3. **class** Base
4. {

```
5.     public:
6.     virtual void show() = 0;
7. };
8. class Derived : public Base
9. {
10.    public:
11.    void show()
12.    {
13.        std::cout << "Derived class is derived from the base class." << std::endl;
14.    }
15.};
16.int main()
17.{
18.    Base *bptr;
19.    //Base b;
20.    Derived d;
21.    bptr = &d;
22.    bptr->show();
23.    return 0;
24.}
```

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.

Static binding vs dynamic binding or late vs early binding

Binding generally refers to a mapping of one thing to another. In the context of compiled languages, binding is the link between a function call and the function definition. When a function is called in C++, the program control binds to the memory address where that function is defined.

There are two types of binding in C++: static (or early) binding and dynamic (or late) binding. This post provides an overview of the differences between the between static and dynamic binding in C++.

1. The static binding happens at the compile-time and dynamic binding happens at the runtime. Hence, they are also called early and late binding respectively.
2. In static binding, the function definition and the function call are linked during the compile-time whereas in dynamic binding the function calls are not resolved until runtime. So they are not bound until runtime.
3. Static binding happens when all information needed to call a function is available at the compile-time. Dynamic binding happens when all information needed for a function call cannot be determined at compile-time.
4. Static binding can be achieved using the normal function calls, function overloading and operator overloading while dynamic binding can be achieved using the virtual functions.
5. Since all information needed to call a function is available before runtime, static binding results in faster execution of a program. Unlike static binding, a function call is not resolved until runtime for later binding and this results in somewhat slower execution of code.

6. The major advantage of dynamic binding is that it is flexible since a single function can handle different type of objects at runtime. This significantly reduces the size of the codebase and also makes the source code more readable.

Example of Static Binding in C++:

Consider below code where `sum()` function is overloaded to accept two and three integer arguments. Even though there are two functions with the same name inside `ComputeSum` class, the function call `sum()` binds to the correct function depending on the parameters passed to those functions. This binding is done statically during compile time.

```

1 // C++ program to illustrate the concept of static binding
2 #include <iostream>
3 using namespace std;
4
5 class ComputeSum
6 {
7     public:
8
9     int sum(int x, int y)
10    {
11        return x + y;
12    }
13
14    int sum(int x, int y, int z)
15    {
16        return x + y + z;
17    }
18 };
19
20 int main()
21 {
22     ComputeSum obj;
23     cout << "Sum is " << obj.sum(10, 20) << '\n';
24     cout << "Sum is " << obj.sum(10, 20, 30) << '\n';
25
26     return 0;
27 }

```

[Download](#) [Run Code](#)

Output:

```

Sum is 30
Sum is 60

```

Example of Dynamic Binding in C++:

Consider below code where we have a base class B and a derived class D. Base class B has a virtual function `f()` which is overridden by function in the derived class D. i.e. `D::f()` overrides `B::f()`.

Now consider lines 30-34 where the decision as to which class's function will be invoked depends on the dynamic type of the object pointed to by basePtr. This information can only be available at the runtime and hence `f()` is subject to dynamic binding.

```

1  // C++ program to illustrate the concept of dynamic binding
2  #include <iostream>
3  using namespace std;
4
5  class B
6  {
7      public:
8
9      // Virtual function
10     virtual void f()
11     {
12         cout << "Base class function called.\n";
13     }
14 };
15
16 class D: public B
17 {
18     public:
19     void f()
20     {
21         cout << "Derived class function called.\n";
22     }
23 };
24
25 int main()
26 {
27     B base;
28     D derived;
29
30     B *basePtr = &base;
31     basePtr->f();
32
33     basePtr = &derived;
34     basePtr->f();
35
36     return 0;
37 }

```

[Download](#) [Run Code](#)

Output:

```

Base class function called.
Derived class function called.

```

<https://www.techiedelight.com/difference-between-static-dynamic-binding-cpp/#:~:text=In%20static%20binding%2C%20the%20function,available%20at%20the%20compile%2Dtime.>

Virtual table

<https://pabloariasal.github.io/2017/06/10/understanding-virtual-tables/>

Templates

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works similarly to a normal [function](#), with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

How to declare a function template?

A function template starts with the keyword `template` followed by template parameter/s inside `< >` which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
    . . . . .
}
```

In the above code, `T` is a template argument that accepts different data types (int, float), and `class` is a keyword.

You can also use keyword `typename` instead of `class` in the above example.

When, an argument of a data type is passed to `someFunction()`, compiler generates a new version of `someFunction()` for the given data type.

Example 1: Function Template to find the largest number

Program to display largest among two numbers using function templates.

```
// If two characters are passed to function template, character with  
larger ASCII value is displayed.
```

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

// template function
template <class T>
T Large(T n1, T n2)
{
    return (n1 > n2) ? n1 : n2;
}

int main()
{
    int i1, i2;
    float f1, f2;
    char c1, c2;

    cout << "Enter two integers:\n";
    cin >> i1 >> i2;
    cout << Large(i1, i2) << " is larger." << endl;

    cout << "\nEnter two floating-point numbers:\n";
    cin >> f1 >> f2;
    cout << Large(f1, f2) << " is larger." << endl;

    cout << "\nEnter two characters:\n";
    cin >> c1 >> c2;
    cout << Large(c1, c2) << " has larger ASCII value.";

    return 0;
}
```

Output

```
Enter two integers:
```

```
5
```

```
10
```

```
10 is larger.
```

```
Enter two floating-point numbers:
```

```
12.4
```

```
10.2
```



```
12.4 is larger.
```

```
Enter two characters:
```

```
z
```

```
Z
```

```
z has larger ASCII value.
```

In the above program, a function template `Large()` is defined that accepts two arguments `n1` and `n2` of data type `T`. `T` signifies that argument can be of any data type.

`Large()` function returns the largest among the two arguments using a simple [conditional operation](#).

Inside the `main()` function, variables of three different data types: `int`, `float` and `char` are declared. The variables are then passed to the `Large()` function template as normal functions.

During run-time, when an integer is passed to the template function, compiler knows it has to generate a `Large()` function to accept the `int` arguments and does so.

Similarly, when floating-point data and `char` data are passed, it knows the argument data types and generates the `Large()` function accordingly.

This way, using only a single function template replaced three identical normal functions and made your code maintainable.

Example 2: Swap Data Using Function Templates

Program to swap data using function templates.

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

template <typename T>
void Swap(T &n1, T &n2)
{
    T temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}

int main()
{
    int i1 = 1, i2 = 2;
    float f1 = 1.1, f2 = 2.2;
    char c1 = 'a', c2 = 'b';

    cout << "Before passing data to function template.\n";
    cout << "i1 = " << i1 << "\ni2 = " << i2;
    cout << "\nf1 = " << f1 << "\nf2 = " << f2;
    cout << "\nc1 = " << c1 << "\nc2 = " << c2;

    Swap(i1, i2);
    Swap(f1, f2);
    Swap(c1, c2);

    cout << "\n\nAfter passing data to function template.\n";
    cout << "i1 = " << i1 << "\ni2 = " << i2;
    cout << "\nf1 = " << f1 << "\nf2 = " << f2;
    cout << "\nc1 = " << c1 << "\nc2 = " << c2;

    return 0;
}
```

Output

```
Before passing data to function template.
i1 = 1
i2 = 2
```

```
f1 = 1.1  
f2 = 2.2  
c1 = a  
c2 = b
```

After passing data to function template.

```
i1 = 2  
i2 = 1  
f1 = 2.2  
f2 = 1.1  
c1 = b  
c2 = a
```

In this program, instead of calling a function by passing a value, a [call by reference](#) is issued.

The `swap()` function template takes two arguments and swaps them by reference.

Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change in one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template <class T>
class className
{
    ... ..
public:
    T var;
    T someOperation(T arg);
    ... ..
};
```

In the above declaration, `T` is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable `var` and a member function

`someOperation()` are both of type `T`.

How to create a class template object?

To create a class template object, you need to define the data type inside a `<` when creation.

```
className<dataType> classObject;
```

For example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Example 3: Simple calculator using Class template

Program to add, subtract, multiply and divide two numbers using class template

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

template <class T>
class Calculator
{
private:
    T num1, num2;

public:
    Calculator(T n1, T n2)
    {
        num1 = n1;
        num2 = n2;
    }

    void displayResult()
    {
        cout << "Numbers are: " << num1 << " and " << num2 << "." <<
endl;
        cout << "Addition is: " << add() << endl;
        cout << "Subtraction is: " << subtract() << endl;
        cout << "Product is: " << multiply() << endl;
        cout << "Division is: " << divide() << endl;
    }

    T add() { return num1 + num2; }

    T subtract() { return num1 - num2; }
```

```
T multiply() { return num1 * num2; }

T divide() { return num1 / num2; }
};

int main()
{
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);

    cout << "Int results:" << endl;
    intCalc.displayResult();

    cout << endl << "Float results:" << endl;
    floatCalc.displayResult();

    return 0;
}
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

In the above program, a class template `Calculator` is declared.

The class contains two private members of type `T: num1 & num2`, and a constructor to initialize the members.

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user. Likewise, a function `displayResult()` to display the final output to the screen.

In the `main()` function, two different `Calculator` objects `intCalc` and `floatCalc` are created for data types: `int` and `float` respectively. The values are initialized using the constructor.

Notice we use `<int>` and `<float>` while creating the objects. These tell the compiler the data type used for the class creation.

This creates a class definition each for `int` and `float`, which are then used accordingly.

Then, `displayResult()` of both objects is called which performs the `Calculator` operations and displays the output.

