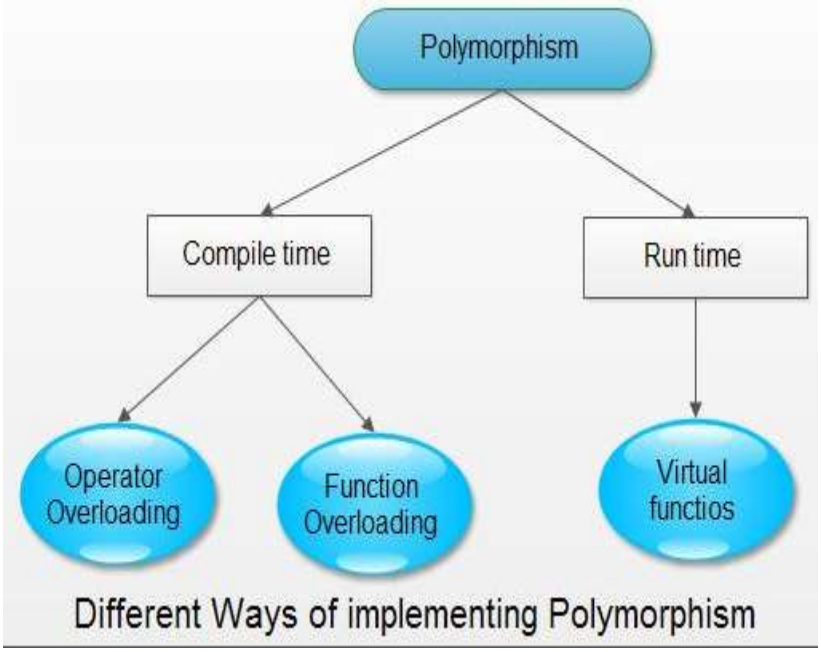
# Polymorphism and Virtual Functions in C++

- Polymorphism is the ability of objects to respond with different appropriate actions against the same message. It means taking more than one form.
- In C++ it is implemented in OOP as Function/Operator Overloading and Function Overriding.



Different Ways of implementing Polymorphism

Runtime polymorphism is also known as **Subtype Polymorphism/ Dynamic poly./ Late Binding**

Everyone understands runtime polymorphism when they say "polymorphism" in C++.

Compile time polymorphism is also known as **Parametric polymorphism/ Static poly/ Early binding.**

Another type is **Coercion** also known as (implicit or explicit) **casting**.

- Overloading : supplying more than one definition for a given function name in the same scope. The compiler is left to pick the appropriate version of the function or operator based on the arguments with which it is called.
- Overriding refers to the modifications made in the sub class to the inherited methods from the base class to change their behaviour.

For example, the + (plus) operator in C++:

In C++, that type of polymorphism is called *overloading*.

```
4 + 5        <-- integer addition
3.14 + 2.0   <-- floating point addition
s1 + "bar"   <-- string concatenation!
```

Now we will discuss different Implementation of Polymorphism in C++ one by one.

## 1. Different classes implement a common interface

Imagine that we need to display the original price and discounted price for a set of completely different products. For modelling these products in a reusable and extensible way we define an interface

```
public interface Product {
   //  Any class that implements this interface must  have methods with the following signatures
   Decimal getDiscount();
   Decimal getPrice();
   String getName();
}
```

Any class that implements this interface has to implement methods that match the method signatures that have been defined. But the implementation logic of the methods are different. This is very powerful and is a common use for polymorphism.

Now we create two categories of products, one for fruit and the other for finance.

```
/** Fruit Products Class **/

public with sharing class FruitProduct implements
                Product {

  private final Decimal discountedRate = 0.85;
  private Decimal price;
  private String name;
  ……..
  // Implemented methods of interface
 ……..
 }
```

```
/** Financial Products Class **/

public with sharing class FinancialProduct
                Implements Product {

  private final Decimal discountedRate = 0.45;
  private final Integer hiddenCost = 100;
  private Decimal price;
  private String name;
 ……..
// Implemented methods of interface
 ……..
}
```

```
public class ProductController {

  // This list can hold any product that implements the Product interface
  public List<Product> allProducts;

  public ProductController() {
    allProducts = new List<Product>();

    // Instantiate some fruit and financial products
    FruitProduct banana = new FruitProduct('Banana', 1);
    FruitProduct dragonFruit = new FruitProduct('Dragon Fruit', 2.50);

    FinancialProduct mortgage = new FinancialProduct('Mortgage', 100);
    FinancialProduct spreadBet = new FinancialProduct('Spread Bet', 5);
```

The list **allProducts** is of type Product which isn't a class but an interface. By doing this we are saying, "create a list that is able to contain any instantiated class that implements the Product interface" i.e. the list can hold any Financial or Fruit Product, or in fact any other class that might later be created that implements this interface.

```
    // And add them to the list
    allProducts.add(banana);        allProducts.add(dragonFruit);
    allProducts.add(mortgage);      allProducts.add(spreadBet);
  }
}
```

## 2. Subtype Polymorphism (Runtime Polymorphism)

Subtype polymorphism is what everyone understands when they say "polymorphism" in C++. It's the ability to use derived classes through base class pointers and references. Here is an example.

Let us assume, we are working on a game (weapons specifically). We created the **Weapon** class and derived two classes **Bomb** and **Gun** to load features of respective weapons.

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

class Weapon
{
  public:
    void loadFeatures()   { cout << "Loading weapon features.\n"; }
};
class Bomb : public Weapon
{
  public:
    void loadFeatures()     { cout << "Loading bomb features.\n"; }
};
class Gun : public Weapon
{
  public:
    void loadFeatures()     { cout << "Loading gun features.\n"; }
};
int main()
{
  Weapon *w = new Weapon;     Bomb *b = new Bomb;     Gun *g = new Gun;
  w->loadFeatures();          b->loadFeatures();      g->loadFeatures();
  return 0;
}
```

**Output**

```
Loading weapon features.
Loading bomb features.
Loading gun features.
```

We defined three pointer objects w, b and g of classes Weapon, Bomb and Gun respectively. And, we called loadFeatures() member function of each objects. From output we can see it Works perfectly!

However, our game project started getting bigger and bigger. And, we decided to create a separate **Loader** class to load weapon features. This Loader class loads additional features of a weapon depending on which weapon is selected.

```
class Loader
{
```

The loadFeatures() loads the feature of a specific weapon.

```
  public:
    void loadFeatures(Weapon *weapon)   {     weapon->features();     }
};
```

**Let's try to implement our Loader class**

```
#include <iostream>
using namespace std;
class Weapon
{
   public:
     void features()      { cout << "Loading weapon features.\n"; }
};
class Bomb : public Weapon
{
   public:
     void features()      { cout << "Loading bomb features.\n"; }
};
class Gun : public Weapon
{
   public:
     void features()     { cout << "Loading gun features.\n"; }
};
class Loader
{
  public:
    void loadFeatures(Weapon *weapon)     {       weapon->features();     }
};
```

```
int main()
{
   Loader *l = new Loader;
   Weapon *w;
   Bomb b;
   Gun g;

   w = &b;
   l->loadFeatures(w);
   w = &g;
   l->loadFeatures(w);
   return 0;
}
```

**Output**

```
Loading weapon features.
Loading weapon features.
Loading weapon features.
```

- Our implementation seemed correct. However, weapon features was loaded 3 times. Why?
- Initially, the Weapon object w is pointing to the b object (of Bomb) class. And, we tried to load the features of Bomb object by passing it to loadFeatures() function using l object to pointer (of Loader class). Similarly, we tried to load the features of Gun object.
- However, the loadFeatures() function of the Loader class takes pointer to object of a Weapon class as an argument: void loadFeatures(Weapon *weapon)
- That's the reason weapon features are loaded 3 times. To solve this issue, we need to make function of base class (Weapon class) virtual using virtual keyword.

## Virtual function

When we want a Derived Class to override a member function of Base class, then we should make that member function in Base class virtual. So that if someone uses the Derived class object using Base class's pointer or reference to call the overridden member function, then function of derived class should be called.

```
class Weapon
{
```

```
  public:
    virtual void features()      { cout << "Loading weapon features.\n"; }
};
```

Example: Using Virtual Function to Solve the Problem

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

class Weapon
{
  public:
    virtual void features()      { cout << "Loading weapon features.\n"; }
};
//   Bomb and Gun class are same as before
……….
……..

int main()
{
  Loader *l = new Loader;
  Weapon *w;      Bomb b;      Gun g;

  w = &b;     l->loadFeatures(w);
  w = &g;     l->loadFeatures(w);
  return 0;
}
```

**Output**

```
Loading weapon features.
Loading bomb features.
Loading gun features.
```

Also, notice that, the `l->loadFeatures(w)` function calls the function of different classes depending upon what `l` object is pointing.

Using virtual function made our code not only clearer but flexible too.

If we want to add another weapon (let's say knife), we can easily add and load features of it. **How**?

```
class Knife : public Weapon
{
  public:
    void features()    { cout << "Loading knife features.\n"; }
};
```

And, in `main()` function.

```
Knife k;    w = &k;
l->loadFeatures(w);
```

It's worth noticing that we didn't changed anything in the `Loader` class to load features of knife.

- Subtype polymorphism is also called **runtime polymorphism** for a good reason.
- The resolution of polymorphic function calls happens at runtime through an indirection via the **virtual table**. Another way of explaining this is that compiler does not locate the address of the function to be called at compile-time, instead when the program is run, the function is called by dereferencing the right pointer in the virtual table.

## How virtual functions works internally using vTable and vPointer?

The term binding refers to the connection between a function call and the actual code executed (function's definition i.e. function's address ) as a result of the call. In conventional programming function invoke is known when the program is compiling because each function is explicitly called by name. This is known as **static binding** or early binding because the compiler can figure out the function to be called before the program ever runs.

There are two types of Bindings,

- **Early Binding:** By Default C++ Compiler do the early binding for all function calls i.e. while linking when compiler sees a function call, then it binds that call with the particular function's address / definition.
- **Dynamic Binding or Run Time Binding:** The **virtual** keyword is a signal to the compiler that the member function qualified by the keyword may need to be called through a pointer. When pointer to a base class which is pointing to a derive class is used to call a virtual function, dynamic binding is used.

  Any call to that virtual function will not be linked to any function's address during compile time. Actual function's address to this call will be calculated at run time. To resolve the actual function's address or definition at run time, C++ compiler adds some additional data structure around virtual functions i.e. **vTable** **vPointers**

**What is vtable & vPointer and How C++ do run time binding using them:**

To implement virtual functions, C++ uses a special form of late binding known as the virtual table or vTable. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

- Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table.
- This table is simply a static array that the compiler creates at compile time.
- A virtual table contains one entry for each virtual function that can be called by objects of the class.
- Each entry in this vTable is simply a Function Pointer stores a pointer to each virtual function accessible by the class. Only the most specific function definition callable by the class is stored in the *vtable*. Entries in the *vtable* can point to either functions declared in the class itself or virtual functions inherited from a base class.

- Every time the compiler creates a *vtable* for a **base** class, it adds an extra argument to it: a hidden pointer to the corresponding virtual table, called the *vpointer* (***__vPtr)**.
- ***__vPtr** is set (automatically) when a **class instance** is created so that it points to the virtual table for that class. The compiler adds vPtr to every **object** of the Class which is Polymorphic (contains

atleast one virtual member function). *__vPtr is inherited by derived classes. Object contains a vPointer in first 4 bytes. This vPointer points to the vTable of that class.

- This vPointer will be used to find the actual function address from vTable at run time.

Lets look at an example,

```
class MessageCoverter
{
    int m_count;
    public:
      virtual std::string convert(std::string msg)
      {
          msg = "[START]" + msg + "[END]";
          return msg;
      }
      virtual std::string encrypt(std::string msg)
      {   return msg;   }
      void displayAboutInfo()
      {
         std::cout<<"MessageCoverter Class"<<std::endl;
      }
};
```

FunctionPointer *__vptr; // Compiler added automatically here

The class contains two virtual functions i.e. convert() and encrypt(), hence it has a vTable associated with it. vTable will look like this and points to the most derived function accessible by that class.

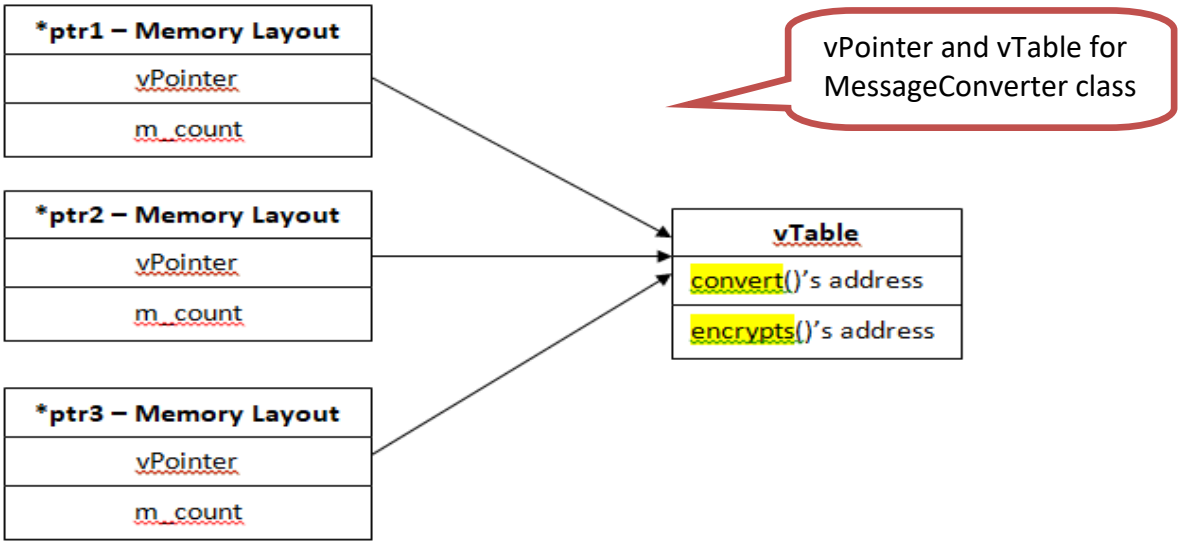
| vTable |
|---|
| convert()'s address |
| encrypts()'s address |

Now suppose we created three different objects for this class,

```
MessageCoverter * ptr1 = new MessageCoverter();
MessageCoverter * ptr2 = new MessageCoverter();
MessageCoverter * ptr3 = new MessageCoverter();
```

In Every object first 4 bytes will a pointer that points to the **vTable** of that class. This pointer is called **vPointer**.

Memory layout of each of the object is as follows,



Because for virtual functions linking was not done at compile time. So, what happens when a call to virtual function is executed ,i.e. ptr1->convert("hello");

Steps are as follows,

- vpointer hidden in first 4 bytes of the object will be fetched
- vTable of this class is accessed through the fetched vPointer
- Now from the vTable corresponding function's address will be fetched
- Function will be executed from that function pointer

**What happens when inheritance comes into picture**

```
class Base
{
   public:
      virtual void function1() {};
      virtual void function2() {};
};
class D1: public Base
{
   public:
     // Say we have override this function
      void function1() {};
};
class D2: public Base
{
   public:
     // Say we have override this function
      void function2() {};
};
```

FunctionPointer *__vptr;

The compiler also adds a hidden pointer to the **most base class** that uses virtual functions. Although the compiler does this automatically, we'll put it in this example just to show where it's added.

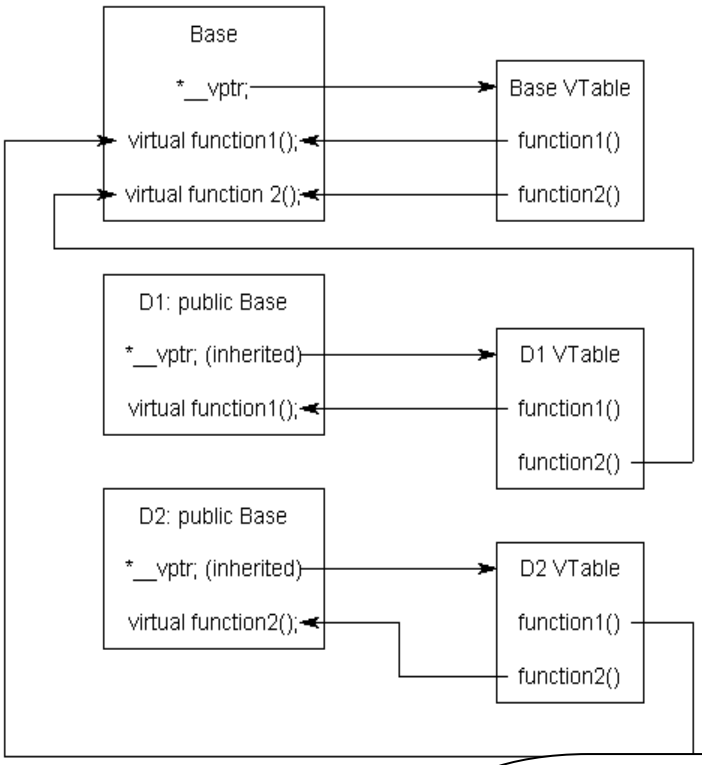
- Once a function is declared virtual in a class then for all its derived classes that function will remain virtual.
- Therefore funtion1() and function2() will be virtual in D1 and D2.
- As all classes Base, D1 and D2 contain virtual function, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.
- The compiler also adds a hidden pointer FunctionPointer *__vptr; to the most Base class that uses virtual functions.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you're only using a pointer or reference to a base class. This can be explained with the help of following pictorial representation.

- When a class object is created, *__vPtr is set to point to the virtual table for that class. For Base class object *__vPtr is set to point to the virtual table for Base. For objects of type D1 or D2 *__vPtr is set to point to the virtual table for D1 or D2 respectively.
- Each virtual table will have two entries (one for function1(), and one for function2()).

- Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

- An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn't overridden function2(), so the entry for function2 will point to Base::function2().
- D2's virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().




As D1 Class inherits Base class, and as we know Base class has a data member _vPtr, D1 class inherits _vPtr of Base class but new Virtual table will be created at compile time for D1 class. Hence _vPtr of D1 class holds the address of vtable of D1 class.

So consider what happens when we create an object of type D1:

```
int main()
{
   D1 d1;
}
```

Because d1 is a D1 object, d1 has its *__vPtr set to the D1 virtual table.

Now, let's set a base pointer to D1:

```
int main()
{
   D1 d1;
   Base *dPtr = &d1;
}
```

- Note that because dPtr is a base pointer, it only points to the Base portion of d1.
- However, also note that ***__vptr is in the Base portion of the class**, so dPtr has access to this pointer. Finally, note that dPtr->__vptr points to the D1 virtual table! Consequently, even though dPtr is of type Base, it still has access to D1's virtual table (through __vptr).

So what happens when we try to call dPtr->function1()?

```
int main()
{
   D1 d1;
   Base *dPtr = &d1;
   dPtr->function1();
```

- First, the program recognizes that function1() is a virtual function.
- Second, the program uses dPtr->__vptr to get to D1's virtual table.
- Third, it looks up which version of function1() to call in D1's virtual table. This has been set to D1::function1(). Therefore, dPtr->function1() resolves to D1::function1()!

}

Now, you might be saying, “But what if Base really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?”. The answer is no.

```
int main()
{
   Base b;
   Base *bPtr = &b;
   bPtr->function1();
}
```

In this case, when b is created, __vptr points to Base’s virtual table, not D1’s virtual table. Consequently, bPtr->__vptr will also be pointing to Base’s virtual table. Base’s virtual table entry for function1() points to Base::function1(). Thus, bPtr->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you’re only using a pointer or reference to a base class!

This is how everything works behind the curtain for every virtual function call.

## Why always make Base class's destructor as virtual if it has any virtual functions in it.

- If your derived class destructor is virtual then objects will be destructed in an order (firstly derived object then base ).
- If your derived class destructor is NOT virtual then only base class object will get deleted (because pointer is of base class "Base *myObj"). So there will be **memory leak** for derived object.
- Always make base classes' destructors virtual when they're meant to be manipulated polymorphic way ( i.e. using base class pointer to aces derived class).

- If you want to prevent the deletion of an instance through a base class pointer, you can make the base class destructor protected and nonvirtual; by doing so, the compiler won't let you call delete on a base class pointer.

A virtual constructor is not possible but virtual destructor is possible. Let us experiment....

```
class Base
{
public:
   Base(){
      cout << "Base Constructor Called\n";
   }
   ~Base(){
      cout << "Base Destructor called\n";
   }
};
```

```
class Derived1: public Base
{
public:
   Derived1(){
      cout << "Derived constructor called\n";
   }
   ~Derived1(){
      cout << "Derived destructor called\n";
   }
};
```

```
int main()
{
   Base *b = new Derived1();
   delete b;
}
```

Output :
Base Constructor Called
Derived constructor called
Base Destructor called

The construction of derived object follow the construction rule but when we delete the "b" pointer (base pointer) we have found that only the base destructor is call. But this must not be happened.

To do the appropriate thing we have to make the **base destructor virtual**. Now let see what happen in the following:

```
class Base
{
public:
   Base(){
      cout << "Base Constructor Called\n";
   }
   virtual ~Base(){
      cout << "Base Destructor called\n";
   }
};
```

```
class Derived1: public Base
{
public:
   Derived1(){
      cout << "Derived constructor called\n";
   }
   ~Derived1(){
      cout << "Derived destructor called\n";
   }
};
```

```
int main()
{
   Base *b = new Derived1();
   delete b;
}
```

The output changed as following:
Base Constructor Called
Derived constructor called
Derived destructor called
Base Destructor called

So the destruction of base pointer(which take an allocation on derived object!) follow the destruction rule i.e first the derived then the base. On the other hand for constructor there are nothing like virtual constructor.

Calling delete on pointer **b** follows the _vptr - which is pointing to Derived1's vtable. So it will calls it's own class's destructor and then calls the destructor of Base class - this as part of when derived object gets deleted it turn deletes it's embedded base object.

**That is why we must always make Base class's destructor as virtual if it has any virtual functions in it.**

## override Keyword in C++ (since C++11)

- Function overriding is redefinition of base class function in its derived class with same signature i.e return type and parameters.
- But there may be situations when a programmer makes a mistake while overriding that function. So, to keep track of such an error, C++11 has come up with the **keyword override**. It will make the compiler to check the base class to see if there is a virtual function with this exact signature. And if there is not, the compiler will show an error.

```
// A CPP program without override keyword. Here
programmer makes a mistake and it is not caught.

#include <iostream>
using namespace std;

class Base {
public:
   // user wants to override this in  derived class
   virtual void func() {
      cout << "I am in base" << endl;
   }
};

class derived : public Base {
public:
   // did a silly mistake - an argument "int a"
   void func(int a) {
      cout << "I am in derived class" << endl;
   }
};
// Driver code
int main()
{
   Base b;
   derived d;
   cout << "Compiled successfully" << endl;
   return 0;
}
```

Overridden function signature differ, but program compiled successfully.

Output:

```
Compiled successfully
```

```
// A CPP program that uses override keyword so
// that any difference in function signature is
// caught during compilation.
#include <iostream>
using namespace std;

class Base {
public:
    // user wants to override this in derived class
   virtual void func()   {
      cout << "I am in base" << endl;
   }
};

class derived : public Base {
public:
   // did a silly mistake - an argument "int a"
   void func(int a) override   {
      cout << "I am in derived class" << endl;
   }
};

int main()
{
   Base b;
   derived d;
   cout << "Compiled successfully" << endl;
   return 0;
}
```

Output:

```
prog.cpp:17:7: error: 'void derived::func(int)'
marked 'override', but does not override
```

| | |
|---|---|
| | void func(int a) override<br>^ |

## What do the override and final keywords do? Why are they useful?

- These keywords give explicit control over virtual function overriding. They are useful because they let the programmer explicitly declare intent in a way the language can enforce at compile time.
- Writing override declares the intent to override a base class virtual function.
- Writing final makes a virtual function no longer override able in further-derived classes, or a class no longer permitted to have further-derived classes.
- If you write override but in derive class overridden function mismatch you get a compile-time error.
- If you write final and a further-derived class tries to implicitly or explicitly override the function anyway, you get a compile-time error.
- Of the two, by far the more commonly useful is override; uses for final are rarer.

**Note** that use of final specifier in C++ 11 is same as in Java but Java uses final before the class name while final specifier is used after the class name in C++ 11.
Same way Java uses final keyword in the beginning of method definition (Before the return type of method) but C++ 11 uses final specifier after the function name.

```
class Test
{
public:
   virtual void fun() final
   {}
};
```

**Use of final specifier in C++ 11:**

Sometimes you don't want to allow derived class to override the base class' virtual function. C++ 11 allows built-in facility to prevent overriding

```
class Base {
public:
   virtual void myfun() final   {    cout << "myfun() in Base";   }
};
class Derived : public Base {
   void myfun()    {    cout << "myfun() in Derived\n";   }
};

int main()
{
   Derived d;     Base &b = d;     b.myfun();      return 0;
}
```

Output:

```
prog.cpp:14:10: error: virtual function 'virtual void Derived::myfun()'
    void myfun()
         ^
prog.cpp:7:18: error: overriding final function 'virtual void Base::myfun()'
    virtual void myfun() final
```

## Some more Examples of Runtime Polymorphism

- **Example -2**

Say there are 2 types of employees :
- a *generic employee* (class **Employee**)
- a *manager* (class **Manager**)

The declaration of generic Employee is given. To help demonstrate polymorphism in C++, we'll focus on the methods that *calculate an employee's pay*.

Let's first define the generic Employee:

```
class Employee {
  public:
    Employee(string theName, float thePayRate);
    string getName() const;
    float getPayRate() const;
    float pay(float hoursWorked) const;
    protected:
          string name;
          float payRate;
};
```

If a member function does not alter any data in the class, you should declare the as **const** function.

Compiler generates an error message if the definition of the function includes any code that inadvertently assigns a value to any variable in that class

Definitions for each of the methods follow:

```
Employee::Employee(string theName, float thePayRate)
{
    name = theName;    payRate = thePayRate;
}
string Employee::getName() const  {   return name;  }

float Employee::getPayRate() const    {    return payRate;    }

float Employee::pay(float hoursWorked) const          {    return hoursWorked * payRate;   }
```

Now define the **Manager** class inheriting from Employee

```
#include "employee.h"
class Manager : public Employee {
     public:
          Manager(string theName,
                float thePayRate, bool isSalaried);
          bool getSalaried() const;
          float pay(float hoursWorked) const;
      protected:
          bool salaried;
};
```

The pay() method is overridden. If the manager is salaried, payRate is the fixed rate

Definitions for the *additional* or *overridden* methods follow:

```
Manager::Manager(string theName,
    float thePayRate,  bool isSalaried)
          : Employee(theName, thePayRate)
{
    salaried = isSalaried;
}
bool Manager::getSalaried() const
{     return salaried;    }

float Manager::pay(float hoursWorked) const
{
```

| | |
|---|---|
| for the pay period; otherwise, it represents an hourly rate, just like it does for a regular employee. | if (salaried)<br>return payRate;<br>else<br>return Employee::pay(hoursWorked);<br>} |

**Using Employee and Manager objects** : These Employee and Manager classes can be used as follows:

| | |
|---|---|
| #include "employee.h"<br>#include "manager.h"<br><br>...<br>// Print out name and pay (based on 40 hours work).<br><br>Employee empl("John Burke", 25.0);<br><br>cout << "Name: " << empl.getName() << endl;<br>cout << "Pay: " << empl.pay(40.0) << endl;<br><br>Manager mgr("Jan Kovacs", 1200.0, true);<br>cout << "Name: " << mgr.getName() << endl;<br>cout << "Pay: " << mgr.pay(40.0) << endl;<br>cout << "Salaried: " << mgr.getSalaried() << endl; | A Manager has all the methods inherited from Employee, like getName(), new versions for those it overrode, like pay(), plus ones it added, like getSalaried().<br><br>**Note:** Everything that was protected in the base class remains protected in the derived class. And, those things that were private in the base class are not directly accessible in the derived class.<br><br>Here we access the method by creating the object and calling its method and the overridden method is not declared as **virtual.** |

**Pointer to a base class** : A base class pointer can point to either an object of the base class or of any publicly-derived class:

```
Employee *emplP;

if (condition1) {
  emplP = new Employee(...);
} else if (condition2) {
  emplP = new Manager(...);
}
```

This allows us, for example, to write one set of code to deal with any kind of employee:

```
cout << "Name: " << emplP->getName();
cout << "Pay rate: " << emplP->getPayRate();
```

**Note:** Typically, one just needs to write *different code* only to assign the pointer to the right kind of object, but not to call methods (as above).

**Calling methods with base class pointers:** Calling methods which are not overridden using the base class pointer such as

```
cout << "Name: " << emplP->getName();
cout << "Pay rate: " << emplP->getPayRate();
```

will do the same things (return the name field or payRate field) whether the pointer points to an Employee or Manager. That's because both classes use the exact same version of those methods--the one defined in Employee.

What will happen when a overridden method is called?

```
Employee *emplP;

if (condition1) {
  emplP = new Employee(...);
} else if (condition2) {
  emplP = new Manager(...);
}
```

In the following code we create two objects separately and then call the pay methods of respective objects as shown.

```
Employee empl;     Manager mgr;
cout << "Pay: " << empl.pay(40.0);  // calls  Employee::pay()
cout << "Pay: " << mgr.pay(40.0);   // calls   Manager::pay()
```

Now say we set the base class pointer with **empl** and **mgr** objects respectively and then call the pay method (overridden method). We will found that in both cases it will **call the pay method of Employee object** and it will give wrong result.

```
emplP = &empl;          // make point to an Employee
cout << "Pay: " << emplP->pay(40.0);
                        // calls Employee::pay()
emplP = &mgr;           // make point to a Manager
cout << "Pay: " << emplP->pay(40.0);
                        // calls Employee::pay()
```

**The reason behind calling base class method in both cases** : By default the type of the pointer (i.e., Employee), **not the type of the object** it points to determines which version will be called.

The reason for the incorrect output is that the call of the function pay() 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**.

To get the intended behavior we have to make the pay() method **virtual**.

```
class Employee {
public:
  ...
  virtual float pay(float hoursWorked) const;
  ...
};
```

**Note:** Once a method is declared as virtual, it is virtual in all derived classes too.

**Virtual methods with references:**

The same behavior that virtual methods exhibit with ***pointers* to objects extends to *references***.
For example, suppose we wanted a function to print the pay for any kind of employee.

If the pay() method is declared virtual, the function can be written much simpler:

```
void PrintPay(const Employee &empl, float hoursWorked)
{
      cout << "Pay: " << empl.pay(hoursWorked) << endl;
}
```

If a new class that inherits (publicly) from Employee is later added, then PrintPay() works for it too (without modification). Function PrintPay() can be used as:

```
Employee empl;      PrintPay(empl, 40.0);
Manager mgr;        PrintPay(mgr, 40.0);
```

**Calling virtual methods within other methods:**

The polymorphic behavior of virtual methods extends to calling them within another method.

For example, suppose the pay() method has been declared virtual in Employee. And, we add a printPay() method to the Employee class:

```
void Employee::printPay(float hoursWorked) const
{
      cout << "Pay: " << pay(hoursWorked) << endl;
}
```

As Manager class is inherited from Employee, the same version of printPay() is available for Manager as the function is not overloaded or being overridden.

Now say we execute following :

```
Manager mgr;          mgr.printPay(40.0);
```

Now the question is which version of pay() will be called within printPay() for a Manager?

The Manager version of pay() gets called inside of printPay() even though printPay() was only defined in Employee! Why? Remember that:

```
void Employee::printPay(float hoursWorked)
                                        const
{
  ... pay(hoursWorked) ...
}

is really shorthand for:
```

```
void Employee::printPay(float hoursWorked)
                                        const
{
  ... this->pay(hoursWorked) ...
}
```

and we know that virtual functions behave polymorphically with pointers!

**Example -3**

In this example a class CRectangle and CTraingle is derived from the base class CPolygon. A new method area() is added in both derived class.

```
#include <iostream>
using namespace std;
class CPolygon {
      protected:
```

```
        int width, height;
    public:
        void set_values (int a, int b)
                { width=a; height=b; }
};
class CRectangle: public CPolygon {
    public:
        int area ()  { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
        int area ()  { return (width * height / 2); }
};
```

```
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
20
10
```

- In function main, we create two base class pointers and the assign references to rect and trgl to these pointers.
- The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon.

  For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

- In order to use area() with the pointers to class CPolygon we need to declare it in the class CPolygon and appropriately overridden by CRectangle and CTriangle as shown below.

```
// virtual members
#include <iostream>
using namespace std;
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
        { width=a; height=b; }
        virtual int area () { return (0); }
};
class CRectangle: public CPolygon {
    public:
        int area ()  { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
        int area ()  { return (width * height / 2); }
```

A **pure virtual function** is that the base class cannot implement. It is indicated by adding the **= 0** initializer following the declaration of the function. Example :

```
class shape
{
   public :
      virtual double compute_area (void)
                    const  = 0
};
```

Compiler will do two error checking for pure virtual function :

- Does not allow creation of instance of a class containing pure virtual function. Thus, if you write **shape s1;** a error will be thrown.
- Checks to be sure the pure virtual functions of a base class are implemented by one of the derived classes.

```
};
int main () {
        CRectangle rect;
        CTriangle trgl;
        CPolygon poly;
        CPolygon * ppoly1 = &rect;
        CPolygon * ppoly2 = &trgl;
        CPolygon * ppoly3 = &poly;

        ppoly1->set_values (4,5);
        ppoly2->set_values (4,5);   ppoly3->set_values (4,5);
        cout << ppoly1->area() << endl;      cout << ppoly2->area() << endl;
        cout << ppoly3->area() << endl;
        return 0;
}
20
10
0
```

The member function area() has been declared as **virtual** in the base class because it is later redefined in each derived class.

If we run the program we will get correct result 20, 10, 0
If we remove this virtual keyword from the declaration of area() within CPolygon, the result will be 0 for the three polygons instead of 20, 10 and 0.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a ***polymorphic class***.

## Another Example

With **play( )** defined as **virtual** in the base class, you can add as many new types as you want without changing the **tune( )** function. In a well-designed OOP program, most or all of your functions will follow the model of **tune( )** and communicate only with the base-class interface. The functions that manipulate the base-class interface (**tune()**) will not need to be changed at all to accommodate the new classes. Here's the instrument example with more virtual functions and a number of new classes, all of which work correctly with the old, unchanged **tune( )** function:

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat };

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
```

```
  // Assume this will modify the object:
  virtual void adjust(int) {}
};
class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};
class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};
class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};
class Brass : public Wind {
public:
  void play(note) const { cout << "Brass::play" << endl;    }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const { cout << "Woodwind::play" << endl;     }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}
```

```
// New function:
void f(Instrument& i) {
i.adjust(1); }

// Upcasting during array
initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

You can see that another inheritance level has been added beneath **Wind**, but the **virtual** mechanism works correctly no matter how many levels there are. The **adjust( )** function is *not* overridden for **Brass** and **Woodwind**. When this happens, the "closest" definition in the inheritance hierarchy is automatically used – the compiler guarantees there's always *some* definition for a virtual function, so you'll never end up with a call that doesn't bind to a function body.The array **A[ ]** contains pointers to the base class **Instrument**, so upcasting occurs during the process of array initialization.