C++ OOP

面向对象编程，顾名思义，在编程中使用对象。面向对象编程旨在在编程中实现现实世界的实体，如继承、隐藏、多态性等。OOP 的主要目标是将数据和对它们进行作的函数绑定在一起，以便代码的其他部分除了该函数之外，其他任何部分都无法访问这些数据。

概述

Class 类

C++ 中面向对象编程的构建块是 **Class**。它是一种**用户定义的数据类型**，充当蓝图，表示一组共享一些常见属性和行为的对象。这些属性存储为数据成员，行为由成员函数表示。

例如，考虑`Animal`类。`Animal`类可以具有`name`,`age`,`species`等公共属性作为数据成员，以及`eat`,`sleep`,`makesound`等行为作为成员函数(member function)。每个单独的动物对象（猫，狗，大象）都可以以这个蓝图为基础进行创建，并且每个对象都有自己唯一的属性值（比如`name`,`age`），但是所有的对象都共享由类定义的相同行为（例如`eat`,`sleep`,`makesound`）。

使用关键字`class`定义一个类，代码实例如下：

```
class Animal {
public:
    string species;
    int age;
    int name;
    // Member functions
    void eat() {}// eat something
    void sleep() {}// sleep for few hrs
    void makeSound () {}// make sound;
};
```

这里的`public`是一个访问说明符，用于指定哪些函数可以供其他人使用。

Object 对象

Object是具有某些特征和行为的可识别实际实体。在C++中，它是类的实例。用上文为例子，`Animal`类只是一个概念或者类别，而不是一个实际的实体，但是一只名叫`ShaoFangwei`的猪是真实存在的动物。也就是说，类只是概念，而对象是属于该概念的实际实体。

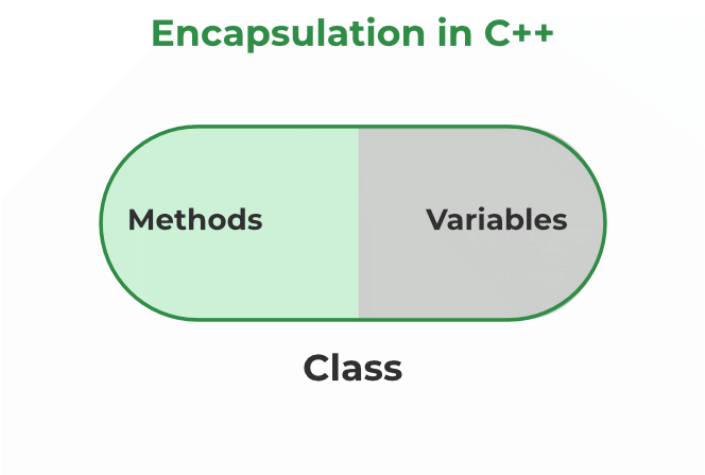
在定义类的时候，不会分配内存，但是当将其实例化（即创建对象）时，会分配内存。在上面的例子中，即使我们定义了`Animal`类，如果该类没有对象，它也不能被使用。创建一个类的对象的代码示例如下：

```
// 上文函数
int main(){
  Animal ShaoFangwei;
}
```

对象占用内存空间，并具有关联的地址，如 pascal 或 structure 或 union 中的记录。执行程序时，对象通过相互发送消息进行交互。每个对象都包含数据和用于作数据的代码。对象可以交互，而不必知道彼此的数据或代码的详细信息，只需知道接受的消息类型和对象返回的响应类型就足够了。

EncapSulation 封装

简单来说，封装就是把数据和信息包装在一个单元下。在面向对象编程中，封装被定义为将**数据和作用它们的函数**在一个类中绑定在一起。考虑一个`Animal`类的示例，数据成员`species`,`age`,`name`和成员函数`eat()`,`sleep()`等绑定在一起。它们可以通过访问说明符`protected`进行保护，从外部隐藏类的数据。



Abstraction 抽象化

抽象是C++中面向对象编程最基本和最重要的特性之一。抽象意味着只显示基本信息而忽略其他细节。数据抽象指指仅向外界提供有关数据的必要信息，隐藏背景细节或实现。在我们的例子中，当我们调用`makeSound()`方法时，我们不需要知道声音是内部如何产生的，只需要知道该方法会发出动物的声音。

Polymorphism 多态性

简单来说，多态性指的是实体在不同场景中表现不同的能力。例如，对于相同的`makeSound()`成员函数，输出将根据动物的类型而变化。多态性可以有三种类型：

- **Operator Overloading 运算符重载**：使运算符在不同实例中表现出不同行为的过程
- **Function Overloading 函数重载**：使用单个函数名称来执行不同类型的任务。
- **Function Overriding 函数覆盖**：使用继承更改从基类派生的函数的行为。

Inheritance 继承性

一个类从另一个类派生属性和特征的能力称为继承。

- **Sub Class 子类**：从另一个类继承属性的类称为Sub Class或者Derived Class（派生类）
- **Super Class 父类**：其属性由子类继承的类称为Base Class或者Super Class

继承支持 “可重用性” 的概念，即当我们想要创建一个新类并且已经有一个类包含我们想要的一些代码时，我们可以从现有类派生我们的新类。例如，`Dog`,`Cat`,`Cow`可以是`Animal`基类的派生类。

OOP的优势

- **模块化和可重用性**：OOP 通过类和对象促进模块化，从而允许代码可重用性。
- **数据封装**：OOP 将数据封装在对象中，从而提高数据的安全性和完整性。
- **继承**：OOP 支持继承，通过重用现有代码来减少冗余。
- **多态性**：OOP 允许多态性，通过函数覆盖实现灵活和动态的代码。
- **抽象化**：OOP 支持抽象，隐藏复杂的细节并仅公开基本功能

C++中的类和对象

C++ Class

类是用户定义的**数据类型**，它包含自己的数据成员和成员函数，可以通过创建该类的实例来访问和使用这些数据成员和成员函数。C++类就像对象的蓝图。

考虑 Cars 的 类别。可能有许多具有不同名称和品牌的汽车，但它们都会共享一些共同的属性，例如它们都有 **4 个轮子、限速、里程范围** 等。汽车还可以**加速、转弯、刹车**等。 所以在这里，Car是类，车轮、速度限制和里程是它的属性（数据成员）， 加速、转弯、刹车是它的方法（成员函数）。

创建类

在使用类之前，必须定义该类。C++中的类是使用关键字class进行定义的，如下所示：

```
class className{
access_specifier:
  //data member
  //member method
}; // Remember this!!
```

- **Data Members 数据成员**：在类中定义的变量。
- **Member Functions 成员函数**：在类中声明的函数。

示例

```
class GfG {
public:
    int val;
    void show() {cout << "Value: " << val << endl;}
};
```

在示例中，GfG类是使用数据成员val和成员函数show()创建的。在这里，成员函数是在类内部定义的，但它们也可以只在类中声明，然后使用范围解析运算符(::)在外部定义。

C++ Objects

定义类时，仅定义对象的规范（属性和行为）。没有为类定义分配内存。要使用类中定义的data和access函数，我们需要创建其**对象** 。

对象是作为类的实例创建的实际实体。一个类的对象数量可以任意多。例如，在上面，我们讨论了`Cars`的类 。如果我们根据`Car`类的属性创建一辆实际的汽车，则我们制造的汽车是该类的对象。

创建对象

定义类后，我们可以像声明任何其他内置数据类型的变量一样创建其对象。

```
className objectName;
```

此语句创建一个`className`类的对象。

成员访问

只需使用分配的名称即可在类本身内部访问类的成员，而如果要在外部访问它们，需要加上`.`点运算符，例如：

```
obj.member1 // For data members
obj.member2(..) // For functions
```

其中`obj`是给定类的对象名称，`member1`是数据成员，`member2`是成员函数。

Access Modifiers 访问修饰符

在C++类中，我们可以使用访问修饰符控制对类成员的访问，它们是在类中指定的关键字，该访问说明符下的类的所有成员都将具有特定的访问级别，有3个访问修饰符，如下：

- **Public**：可以从类外部访问类的Public成员
- **Private**：只能在类本身内访问类的Private成员
- **Protected**：可以在类内访问，也可以在派生类中访问Protected成员

默认情况下，private修饰符将应用与每个成员。

Public Specifier

在 public 说明符下声明的所有类成员将对所有人可用。声明为`public`的数据成员和成员函数也可以由其他类和函数访问。可以使用直接成员访问运算符 （`.`） 对类的对象从程序中的任何位置访问类的公共成员。

```
class Circle {
public:
    double radius;
    double  compute_area() {return 3.14*radius*radius;}
};
int main() {
    Circle obj;
    obj.radius = 5.5;
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
} // Output: Radius is: 5.5  Area is: 94.985
```

在上面的程序中，数据成员`radius`被声明为`public`，因此可以在类外部访问它，因此允许从`main()`内部访问。与成员函数`compute_area()`相同。

Private Specifier

声明为private的类成员只能由类内的成员函数访问。不允许类外部的任何对象或函数直接访问它们。

那么如何访问private成员?

访问类的私有数据成员的标准方法是使用类的`public`成员函数。提供访问权限的函数称为`getter`方法，更新值的函数称为`setter`方法。

```
class Circle {
    double radius;
public:
    double getRadius() {return radius;}
    void setRadius(double val) {radius = val;}
    double  compute_area() {return 3.14*radius*radius;}
};
int main() {
    Circle obj;
    obj.setRadius(1.5);
    cout << "Radius is: " << obj.getRadius() << endl;
    cout << "Area is: " << obj.compute_area();
    return 0;
} // Output: Radius is: 5.5  Area is: 94.985
```

特殊成员函数

在C++类中，有一些特殊的成员函数对于管理对象和提供一些基本功能很重要。

Constructor 构造函数

构造函数是特殊的类成员，每次实例化该类的对象时，编译器都会调用这些成员，它们用于构造对象。**构造函数与类同名**！

```
class MyClass {
public:
    // Constructor
    MyClass() {cout << "Constructor called!";}
};

int main() {
    MyClass obj; // 也就是调用MyClass类创建一个名为obj的对象
    return 0;
}//输出为 Constructor called!
```

构造函数类型

1. **默认构造函数(Default Constructor)**

   如果程序员未定义默认构造函数，则编译器会自动生成默认构造函数。此构造函数不接受任何参数，并使用默认值初始化对象成员。它也称为零参数构造函数。但是，如果已显式定义构造函数，则编译器不会生成默认构造函数。

   ```
   1  class A {
   2  public:
   3  };
   4  int main() {
   5      A a; // 调用A类创建一个名为a的对象
   6      return 0;
   7  }
   ```

   在上面的程序中，类A不包含任何显式定义的构造函数。因此类A的对象是在没有任何参数的情况下构建的，因为该类将使用编译器生成的默认构造函数。

2. **参数化构造函数(Parameterized Constructor)**

参数化构造函数允许我们将参数传递给构造函数。通常，这些参数有助于初始化对象的成员。要创建参数化构造函数，只需像添加任何其他函数一样向其添加参数即可。定义构造函数的主体时，请使用参数初始化对象的成员。

```
class A {
public:
    int val;
    // Parameterized constructor
    A(int x) {val = x;}
};
int main() {
    A a(10);
    cout << a.val;
    return 0;
}
```

在此代码中，当我们创建具有参数`int val = 10`的对象a时，将调用参数化构造函数。如果定义了参数化构造函数，则还应定义非参数化构造函数，因为编译器不会创建默认构造函数。

参数化构造函数有两种写法，分别是成员初始化列表(Member Initializer List)和在函数体内赋值

```
Complex(int r, int i) :
real(r), imag(i) {} // 成员初始化列表
```

- 它在对象的构造阶段就直接**初始化成员变量**
- :后面的部分是初始化列表，用来给类的成员变量赋初始值

```
Complex(int r, int i){
  real = r;
  imag = i;
}
```

- 实际上，`real` 和 `imag` 先被调用默认构造函数初始化（如果是类类型），然后再赋值
- 对于内置类型（如 `int`）虽然不会调用构造函数，但语义上还是“先构造再赋值”

有些情况下必须使用初始化列表方法进行构造

1. **成员是常量(const)**

```
class MyClass {
    const int value;
public:
    MyClass(int v) : value(v) {} // 必须在这里初始化
};
```

2. **成员是引用(reference)**

```
class MyClass {
    int& ref;
public:
    MyClass(int& r) : ref(r) {} // 必须用初始化列表
};
```

3. **没有默认构造函数的类类型成员**

```
class A {
public:
    A(int x) {}
};
class B {
    A a;
public:
    B(int x) : a(x) {} // 必须用初始化列表
};
```

如果你不显式初始化 `a`，编译器会尝试调用 `A()`，但如果没有默认构造函数就会报错。

3. **复制构造函数(Copy Constructor)**

复制构造函数是一个成员函数，它使用同一类的另一个对象初始化对象。复制构造函数将对同一类的对象的引用作为参数。

```
class A {
public:
    int val;
    // Parameterized constructor
    A(int x) {val = x;}
    // Copy constructor
    A(A& a) {val = a.val;}
};
int main() {
    A a1(20); // 创建一个int val = 20的a1对象
```

```
    // Creating another object from a1
    A a2(a1); // 创建一个a2对象作为a1对象的副本
    cout << a2.val;
    return 0;
} // Output: 20
```

在此代码中，复制构造函数用于创建新对象a2作为对象a1的副本。当类A的对象作为构造参数传递时，它会自动调用。与默认构造函数一样，如果显式复制构造函数定义不存在，则 C++ 编译器也会提供隐式复制构造函数。

> *注意：与默认构造函数不同，在默认构造函数中，存在任何类型的显式构造函数都会导致删除隐式默认构造函数，如果不存在显式复制构造函数或显式移动构造函数，则隐式复制构造函数将始终由编译器创建。*

4. **移动构造函数(Move Constructor)**

   移动构造函数就像一个复制构造函数，它从已经存在的对象中构造对象，但它不是复制新内存中的对象，而是利用移动语义将已创建对象的所有权转移到新对象，而无需创建额外的副本。它可以被视为从其他对象窃取资源。move 构造函数使用`std::move()`来转移资源的所有权，当临时对象被传递或按`value`返回时，将调用它。在此不做详细展开。

构造函数的特征

- 构造函数的名称与其类名相同。
- 构造函数大多声明为类的`public`成员，尽管它们可以声明为`private`。
- 构造函数不返回值，因此它们没有返回类型。
- 当我们创建类的对象时，会自动调用构造函数。
- 可以在单个类中声明多个构造函数
- 如果有多个构造函数，则将调用具有匹配函数签名的构造函数。

Destructors 析构函数

析构函数是另一个特殊的成员函数，当对象的范围结束的时候，编译器会调用这个函数。它会释放类的对象以前使用的所有内存，因此不会发生内存泄露。**析构函数也和类同名，但是前缀为~**。析构函数自动存在于每个 C++ 类中，但我们也可以使用以下语法重新定义它们。

```
~MyClass() {cout << "Destructor called!";}
```

就像类的任何其他成员函数一样，我们也可以在类外部定义析构函数，但我们仍然需要在 class 中声明析构函数：

```
className {
public:
     ~className();
}
class-name :: ~className() {
    // Desctructor Body
}
```

示例

- 下面的代码演示了 在创建和销毁对象时分别自动执行构造函数和析构函数。

```
class Test {
public:
    Test() {cout << "Constructor Called"<< endl;}
    ~Test() { cout << "Destructor Called"<< endl;}
};
main() {
    Test t;
    return 0;
} // Output: Constructor Called \n Destructor Called
```

- 下面的代码演示了调用构造函数和析构函数的次数。

```
int Count = 0;
class Test {
public:
    Test(){
        // Number of times constructor is called
        Count++;
        cout << "No. of Object created: "<< Count << endl;
    }
    ~Test() {
        // It will print count in decending order
        cout << "No. of Object destroyed: " << Count<< endl;
        Count--;
    }
};
int main() {
    Test t, t1, t2, t3; // 按照顺序调用Test类创建t,t1,t2,t3对象
    return 0;
```

```
18 }
```

Output:

```
1 No. of Object created: 1
2 No. of Object created: 2
3 No. of Object created: 3
4 No. of Object created: 4
5 No. of Object destroyed: 4
6 No. of Object destroyed: 3
7 No. of Object destroyed: 2
8 No. of Object destroyed: 1
```

对象的销毁顺序与其创建顺序相反。在这种情况下，t3 是第一个被销毁的，而 t 是最后一个。

什么时候需要编写自定义的析构函数?

如果我们不在 class 中编写自己的析构函数，编译器会为我们创建一个默认析构函数。默认的析构函数工作正常，除非我们在类中有动态分配的内存或指针。当一个类包含指向类中分配的内存的指针时，我们应该编写一个析构函数，以便在类实例被销毁之前释放内存。

```
class MyClass {
private:
    // Pointer to dynamically allocated memory
    int* data;
public:
    MyClass(int value) {
        data = new int; // 给 data 指针分配一个新的、在堆上的整数内存地址。
        *data = value; // 把传入的 value 值存到这块刚申请的内存中。
        cout << *data << endl;
    }
    // User-defined destructor: Free the dynamically allocated memory
    ~MyClass() {
        delete data;
        cout << "Destructor: Memory deallocated";
    }
};
int main() {
    MyClass obj1(10);
    return 0;
} // Output: 10   Destructor: Memory deallocated
```

静态成员

类的成员可以声明为static，类的静态成员不与类的对象相关联，而是与类本身相关联。这些成员的主要特征是它们可以通过类直接访问，而无需创建任何对象。数据成员和成员函数都可以是静态的。

静态数据成员

由类的所有对象共享**静态数据成员**，这意味着该成员在类的所有对象中只存在一个副本，我们既可以使用类名调用，也可以使用对象名调用。

```
class GfG {
public:
    static int val;
};
int GfG::val = 22;
int main() {
    cout << GfG::val << endl; // 不需要创建实例就可以访问了
} // 输出为22
```

静态成员函数

静态成员函数与类本身相关联，而不是与任何特定的对象相关联。**它们只能访问静态数据成员，不能访问实例数据成员。**静态成员函数使用类名（而不是对象名）调用。

我们在类内部定义了成员函数，但我们也可以在类外部定义成员函数。要在类定义之外定义成员函数，我们可以使用(::)运算符，例如：

```
class GfG {
public:
    static void printHello();
};
void GfG::printHello() {cout << "Hello World";}
int main() {
    GfG::printHello();
    return 0;
} // Output: Hello World
```

访问静态成员

我们可以在不创建类实例的情况下访问静态数据成员。但是我们需要先初始化它。有两种方法可以访问静态数据成员：

1. 使用Class Name和Scope Resolution Operator访问静态成员

```
Class_Name::var_name
Class_Name::func_name
```

2. 通过Objects访问静态成员

```
object_name.var_name
```

this 指针

在C++中，**this指针**只指向类的当前实例的指针。它存在于每一个非静态成员函数中，用来表示调用这个函数的那个对象本身。**this是const指针，不能被用户自定义**

```
class A {
  public:
    int a;
    A(int a) {this->a = a;} // 这里的this->a是成员变量，右边的a是构造函数参数
    void display() {cout << "Value: " << this->a;}
};

int main() {
    A o(10);
    o.display();
    return 0;
} // Output: Value: 10
```

要理解this指针，了解对象如何看待类的函数和数据成员是很重要的。

1. 每个对象都会获得自己的数据成员副本。
2. All-access与代码段中存在的函数定义相同。

这意味着每个对象都有自己的数据成员副本，并且所有对象共享成员函数的单个副本。那么现在的问题是，如果每个成员函数只有一个副本存在并被多个对象使用，那么如何访问和更新正确的数据成员？编译器提供隐式指针以及函数名称 'this'。'this' 指针作为隐藏参数传递给所有非静态成员函数调用，并可用作所有非静态函数主体中的局部变量。

'this' 指针在静态成员函数中不可用，因为可以在没有任何对象（使用类名）的情况下调用静态成员函数。对于类 X，此指针的类型为 'X* '。此外，如果 X 的成员函数声明为 const，则此指针的类型为 'const X *' 。C++ 通过调用以下代码让对象销毁自身：

```
delete this;
```

以下是使用‘this’指针的情况：

1. 当局部变量的名称与成员的名称相同时

```
class Test{
private:
   int x;
public:
   void setX (int x){this->x = x;} // 这里x可以是成员函数的参数，也可以是类中自己的数据成员，所以需要用this指针进行区分
   void print() { cout << "x = " << x << endl; } // 由于构造函数中的参数仅仅存在于成员函数内部，因此这里的x就是数据成员
};
int main(){
   Test obj;
   obj.setX(20);
   obj.print();
   return 0;
} // Output: x = 20
```

2. 返回对调用对象的引用

```
class Test{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; } // 这个是构造函数
  Test &setX(int a) { x = a; return *this; } // 把对象中的x的值变成a的值之后，返回对于整个对象的引用
  Test &setY(int b) { y = b; return *this; } // 把对象中的y的值变成b的值之后，返回对于整个对象的引用
  void print() { cout << "x = " << x << " y = " << y << endl; }
};
int main(){
```

```
    Test obj1(5, 5);
    obj1.setX(10).setY(20); // 前半部分obj1.setX(10)返回了一个对象，然后.setY(20)又对其进行操作，所以这是没问题的
    obj1.print();
    return 0;
} // Output: x = 10 y = 20
```

3. 作为参数传递给其他函数或方法，有时候你需要把当前对象传给另一个函数或类：

```
void registerObject(MyClass* obj);
void MyClass::init() {
    registerObject(this); // 把当前对象的指针传过去
}
```

结构和类的区别

在 C++ 中，结构的工作方式与类相同，但只有两个细微的差异。其中最重要的是**隐藏实现细节**。默认情况下，结构不会对在代码中使用它的人隐藏其实现详细信息，而类默认情况下会隐藏其所有实现详细信息，因此默认情况下会阻止程序员访问它们。下表总结了所有基本差异。

Difference	Struct	Class
Accessibility	默认情况下，struct的成员是public的。	默认情况下，class的成员是private的。
Keyword	它是使用**struct**关键字声明的。	它是使用 **class** 关键字声明的。
Functionality	它通常用于对不同数据类型进行分组。	它通常用于数据抽象和继承。

Syntax	class class_name {
	data_member;
	member_function;
	};
		struct structure_name {
		structure_member1;
		structure_member2;
		};

Class	Object
用于创建对象的蓝图或模版	具有实际值的类的实例
在创建对象之前，不会为类分配内存	内存是在创建对象时分配的
描述结构和行为的概念实体	从类创建的真实实体
定义该类型所有对象通用的属性和函数	存储特定数据并使用类函数函数对其作用
表示一般概念或类型	表示类的特定实例

C++中的封装

封装定义为将数据和信息包装在单个单元中。在面向对象编程中，**封装**被定义为将数据和作用它们的函数绑定在一起。

考虑一个真实的封装示例，在公司中，有不同的部门，如账户部门、财务部门、销售部门等。现在财务部门处理所有财务交易并保留与财务相关的所有数据的记录。所有财务流程均由此部门完成。同样，销售部门处理所有与销售相关的活动并保留所有销售的记录。

主要优点：

- **模块化**：封装通过将代码组织成处理特定任务的独立对象来促进模块化，使程序更具结构性和易于管理。
- **代码维护**：封装有助于更轻松地维护和更新代码。
- **提高安全性**：通过隐藏数据并限制对数据的未授权访问，封装提高了程序的安全性。
- **解耦**：它有助于减少系统组件之间的依赖，使它们能够独立运行，从而使系统更易于维护和扩展。
- **促进其他面向对象特征的实现**：它还使得实现其他面向对象特征（如多态、继承、抽象等）成为可能。

封装本质上是将数据与其操作方法捆绑在一起，以控制访问并防止意外修改。另一方面，抽象是关于接口设计和简化，呈现高级视图的同时隐藏复杂的实现细节。

封装也与数据隐藏不同。它是使用面向对象编程语言中的访问修饰符来促进数据隐藏的技术，但不是数据隐藏本身。

C++中的抽象

抽象是 C++ 面向对象编程中最基本和最重要的特性之一。抽象意味着只显示必要的信息，忽略细节。数据抽象指的是只向外界提供数据必要的信息，忽略不必要或实现的细节。考虑一个现实生活中的例子：一个人开车。这个人只知道踩油门会提高车速，踩刹车会停车，但他不知道踩油门时车速是如何实际增加的，他不知道汽车的内部机制或汽车中油门、刹车等的实现。

抽象的类型：

- **数据抽象**：这种类型只显示关于数据所需的信息，忽略不必要的细节。

- **控制抽象**：这种类型只显示实现所需的必要信息，并忽略不必要细节。

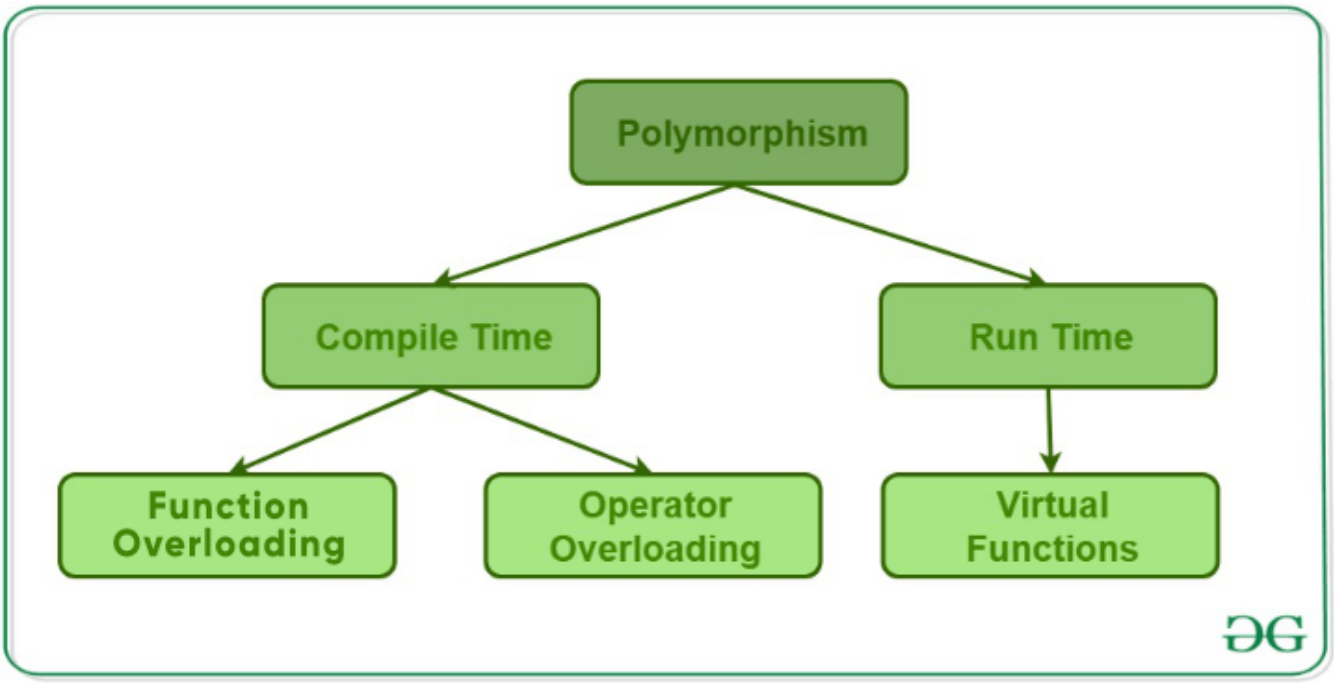
主要优点：

- 帮助用户避免编写底层代码。
- 避免代码重复并提高可重用性。
- 可以独立地更改类的内部实现，而不会影响用户。
- 有助于提高应用程序或程序的安全性，因为只向用户提供重要细节。
- 它减少了代码的复杂性和冗余，从而提高了可读性。
- 新功能或更改可以添加到系统中，对现有代码的影响最小。

抽象(Abstraction)	封装(Encapsulation)
抽象是获取信息的过程或方法	封装是包含信息的过程或方法
在抽象中，问题是在设计或接口部分解决的。	而在封装中，问题是在实现部分解决的。
抽象是隐藏不必要信息的方法。	封装是一种将数据隐藏在单个实体或单元中的方法，以及一种保护信息免受外部访问的方法。
我们可以使用抽象类和接口来实现抽象。	封装可以通过访问修饰符（即 private、protected 和 public）来实现。
在抽象中，通过抽象类和接口隐藏实现复杂性。	而在封装中，通过 getter 和 setter 方法隐藏数据。
用于实现抽象的对象是封装的。	而封装的对象则不必抽象。

C++中的多态

多态意味着有多种形式。现实生活中的多态例子是一个人同时可以具有不同的特征。一个男人同时可以是父亲、丈夫和员工。因此，同一个人在不同的情境中表现出不同的行为。在C++中，多态概念可以应用于函数和运算符。同一个函数在不同的情境中可以有不同的作用。同样，运算符在不同的上下文中也有不同的表现。



编译时多态(Compile-Time Polymorphism)

编译时多态也称为早期绑定(early binding)和静态多态(static polymorphism)，在编译时多态中，编译器根据上下文确定函数或运算符将如何工作。这种多态是通过函数重载或运算符重载实现的。

函数重载(Function Overloading)

函数重载是面向编程中的一个特性，两个或多个函数可以具有相同的名称，但对于不同的参数表现不同。这样的函数被称为重载函数。因此，这被称为函数重载。函数可以通过改变参数的数量或改变参数的类型来进行重载。

```
class Geeks {
public:
    void add(int a, int b) {cout << "Integer Sum = " << a + b<< endl;}
    void add(double a, double b) {cout << "Float Sum = " << a + b<< endl ;}
};
int main() {
    Geeks gfg;
    gfg.add(10, 2);
    gfg.add(5.3, 6.2);
    return 0;
} // Output: Integer Sum = 12 \n Float Sum = 11.5
```

在上述示例中，定义了两个同名的add()函数，但参数类型不同：一个用于整数，一个用于浮点数。编译时根据传入参数的类型调用正确的add()函数，允许使用同一个函数名表示不同的数据类型。

运算符重载(Operator Overloading)

C++ 能够为特定数据类型提供运算符的特殊含义，这种能力称为运算符重载。例如，我们可以使用加法运算符（+）来连接两个字符串，或者对两个整数求和。<< 和 >> 运算符是二元移位运算符，但它们也用于输入和输出流。

```
class Complex {
public:
    int real, imag;
    Complex(int r, int i) :
    real(r), imag(i) {} // 这里用的是成员初始化列表
    // Overloading the '+' operator
    Complex operator+(const Complex& obj) { // 注意这里的语法
        return Complex(real + obj.real,imag + obj.imag);
    }
  void print(){cout << real << " + i" << imag;}
};
int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    cout << c3.real << " + i" << c3.imag;
    return 0;
}
```

这里涉及到构造函数的两种不同写法：成员初始化列表(Member Initializer List)和赋值操作

运行时多态(Run-Time Polymorphism)

运行时多态也称为后期绑定(late binding)和动态多态(dynamic polymorphism)，运行时多态中的函数调用在运行时解析，而编译时多态中，编译器在编译时确定绑定哪个函数调用。运行时多态通过使用**虚函数的函数重写**来实现。

函数重载(Function Overriding)

函数重载发生在派生类，当你定义了基类的一个或多个成员函数，那个基函数被称为被重载函数。基类函数必须声明为虚函数，以便编写运行时多态代码。

```
class Base {
public:
    // Virtual function
    virtual void display() {cout << "Base class function";}
};

class Derived : public Base {
public:
    // Overriding the base class function
    void display() override {cout << "Derived class function";}
};
int main() {
    // Creating a pointer of type Base
    Base* basePtr;
    // Creating an object of Derived class
    Derived derivedObj;
    // Pointing base class pointer to derived class object
    basePtr = &derivedObj;
    // Calling the display function using base class pointer
    basePtr->display();
    return 0;
} // Output: Derived class function
```

在上述示例中，基类 Base 中定义了一个虚拟函数 display()，并在派生类 Derived 中重写了它。Base 类的指针 basePtr 指向一个 Derived 类的对象。当使用 basePtr 调用 display() 函数时，会调用派生类版本的 display() 函数，并打印 "Derived class function."

函数重载总述

函数是一组语句的块，通过接受一些输入并产生特定的输出，共同执行一个特定的任务。在 C++ 中，函数重载是指在其派生类中以相同的签名重新定义基类函数，即返回类型和参数。它可以分为编译时和运行时多态两种类型。函数重载是一种多态类型，在这种多态中，我们重新定义了类从其基类继承的成员函数。函数签名保持不变，但函数的工作方式会改变以满足派生类的需求。因此，当我们使用其名称调用父对象中的函数时，会执行父类中的函数。但是，当我们使用子对象调用该函数时，会执行子类版本。

函数重载有两种类型：Compile-Time Function Overloading和Run-Time Function Overriding

Compile-Time Function Overloading

在编译时函数重载中，函数调用和定义在程序编译时绑定，它也被称为早期绑定或静态绑定。

```
class Parent {
    access_modifier :
        return_type name_of_the_function() {}
};
class child : public Parent {
    access_modifier :
        return_type name_of_the_function() {}
};
```

编译时函数重载示例：

```
class Parent {
public:
    void GeeksforGeeks_Print(){cout << "Base Function" << endl;}
};
class Child : public Parent {
public:
    void GeeksforGeeks_Print(){cout << "Derived Function" << endl;}
};
int main(){
    Child Child_Derived;
    Child_Derived.GeeksforGeeks_Print();
    return 0;
} // Output: Derived Function
```

不能被重载的函数：在 C++中，有一些特定的情况下函数不能被重载。因为函数重载主要基于函数名及其参数的类型或数量。在某些情况下，C++编译器无法区分函数，因此不能重载。

- 具有不同访问修饰符但名称和参数类型列表相同的成员函数声明不能被重载。
- 具有相同参数但传递方法不同（即按值传递和按引用传递）的函数。

函数不能根据返回类型重载，这是因为返回类型不包括在函数调用中，因此编译器无法区分它们，导致出现歧义问题。因此，我们只能通过改变参数的数量或类型来实现函数重载，而不能仅通过改变返回类型来实现。

函数重载和const关键字：

```
class Test {
protected:
    int x;
public:
    Test(int i): x(i){}
    void fun() const {cout << "fun() const called " << endl;}
    void fun() { cout << "fun() called " << endl; }
};
int main()
{
    Test t1(10);
    const Test t2(20);
    t1.fun();
    t2.fun();
    return 0;
} // Output: fun() called   fun() const called
```

这两个方法 `void fun() const` 和 `void fun()` 的函数名相同，不同之处在于一个是const而另一个不是。此外，我们会发现`const void fun()`在const对象上被调用，而`void fun()`在非const对象上被调用。C++ 允许基于const类型重载成员方法。

```
void fun(const int i) { cout << "fun(const int) called "; }
void fun(int i) { cout << "fun(int ) called " ; }
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

上面这个程序会报错！

```
void fun(char *a) { cout << "non-const fun() " << a;}
void fun(const char *a) { cout << "const fun() " << a; }
int main()
{
const char *ptr = "GeeksforGeeks";
fun(ptr);
return 0;
}
```

这个函数可以正常运行

C++ 仅当const参数是指针或引用时，才允许根据参数的const属性进行函数重载。这就是为什么程序 1 编译失败，而程序 2 运行正常。在程序 1 中，参数i是按值传递的，所以fun()中的i是main()中i的一个副本。因此，fun()不能修改main()中的i。所以，无论i是作为const参数接收还是作为普通参数接收，都没有关系。当我们通过引用或指针传递时，我们可以修改所引用或指向的值，因此我们可以有两个版本的函数，一个可以修改所引用或指向的值，另一个则不能。

```
void fun(const int &i) { cout << "fun(const int &) called "; }
void fun(int &i) { cout << "fun(int &) called " ; }
int main()
{
    const int i = 10;
    fun(i);
    return 0;
} // Output: fun(const int &) called
```

Run-Time Function Overriding

函数重写也可以在运行时执行，这意味着函数调用将在运行时绑定到其定义（也称为后期绑定或动态绑定）。这可以通过虚函数来实现。

```
class Base {
public:
    virtual func(){...} // original definition
};
class Derived : public Base {
public:
    func() override{...} // new definition
};
```

这里，override 关键字告诉编译器，给定的被覆盖的函数应该在父类中声明为虚拟的。这是一种双重检查，即使函数不是虚拟的，程序在编译的时候也不会报错。但是，那将是编译时多态，我们不会得到函数覆盖的预期行为。

虚函数

虚函数是在基类中使用 virtual 关键字声明的成员函数，并在派生类中重新定义（覆盖）。它告诉编译器执行后期绑定，即编译器在运行时将对象与正确的调用函数匹配并执行它。这种技术属于运行时多态。

注意：在C++中，调用虚函数意味着；如果我们调用一个成员函数，那么它可能会根据调用它的对象类型执行不同的函数。

核心思想是，虚函数是根据指向的对象实例的类型来调用的，而不是根据指针或引用的类型。换句话说，虚函数是在运行时解析的。

是否为虚函数的区别，主要体现在使用指针调用重载函数的时候

- 如果基类中没有使用虚函数，则`ptr->ChildFunction`会自动调用基类中的函数，而不是子类中的函数
- 如果基类中使用了虚函数，则`ptr->ChildFunction`会自动调用子类中的函数，而不是基类中的函数。

Examples

1. 从派生类中调用重载的函数

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

```
    Child Child_Derived;
    Child_Derived.GeeksforGeeks_Print();
    return 0;
} // Output: Derived Function  Base Function
```

2. 使用指针调用重载函数

```
class Parent {
public:
    void GeeksforGeeks(){cout << "Base Function" << endl;}
};
class Child : public Parent {
public:
    void GeeksforGeeks(){cout << "Derived Function" << endl;}
};
int main(){
    Child Child_Derived;
    Parent* ptr = &Child_Derived;
    ptr->GeeksforGeeks();
    return 0;
} // Output: Base Function
// 因为这里没有使用虚函数进行定义，所以在使用指针进行调用的时候默认调用基类函数
```

3. 使用子类对象调用重载函数

```
class Parent {
public:
    void GeeksforGeeks(){cout << "Base Function" << endl;}
};
class Child : public Parent {
public:
    void GeeksforGeeks(){cout << "Derived Function" << endl;}
};
int main(){
     Child Child_Derived;
     Child_Derived.GeeksforGeeks();
     Child_Derived.Parent::GeeksforGeeks();
    return 0;
} // Output: Derived Function. Base Function
```

Function Overloading	Function Overriding
编译时多态	运行时多态或者编译时多态
函数可以多次重载	不能多次重载
无需继承即可执行	必须要先继承，然后执行
位于同一个作用域中	位于不同的作用域中

C++中的继承

一个类从另一个类中派生属性和特征的能力称为继承。其基本语法如下：

```
class DerivedClass : mode_of_inheritance BaseClass {
    // Body of the Derived Class
};
```

继承方式控制基类成员在派生类中的访问级别。在C++中，有三种继承方式：

- **Public Inheritance Mode**：基类的public成员在派生类中仍然是public的，基类的protected成员在派生类中仍然是protected的。
- **Protected Inheritance Mode**：基类的public和protected成员在派生类中都将变为protected。
- **Private Inheritance Mode(默认)**：基类的public和protected成员在派生类中都会变成private。

访问基类成员

基类的成员可以通过直接使用它们的名称在派生类中访问

```
class Base {
public:
    int n;
    void printN() {cout << n << endl;}
};
// Inheriting Base class publicly
class Derived : public Base {
    public:
    void func () {n = 22;}
};
```

```
int main(){
    Derived d;
    d.func();
    d.printN();
    return 0;
} // Output: 22
```

如果基类以public方式继承，那么基类的public成员可以通过派生类的对象来访问，如上所示。

```
// Base class that is to be inherited
class Parent {
public:
    int id_p;
    Parent(int x = 22) : id_p(x) {} // 构造函数
    void printID_p() {cout << "Base ID: " << id_p << endl;}
}
// Derived publicly inheriting from Base Class
class Child : public Parent {
public:
    int id_c;
    Child(int x = 22) : id_c(x) {} // 构造函数
    void printID_c() {cout << "Child ID: " << id_c << endl;}
};
int main() {
    Child obj1;

    // An object of class child has all data members and member functions of class
parent so we try accessing the parents method and data from the child class object.
    obj1.id_p = 7;
    obj1.printID_p();
    // finally accessing the child class methods and data too
    obj1.id_c = 91;
    obj1.printID_c();
    return 0;
}
```

在上述程序中，Child类是从Parent类中以public方式继承的，所以Parent类的公有数据成员也会被Child类继承。

派生类的访问权限：

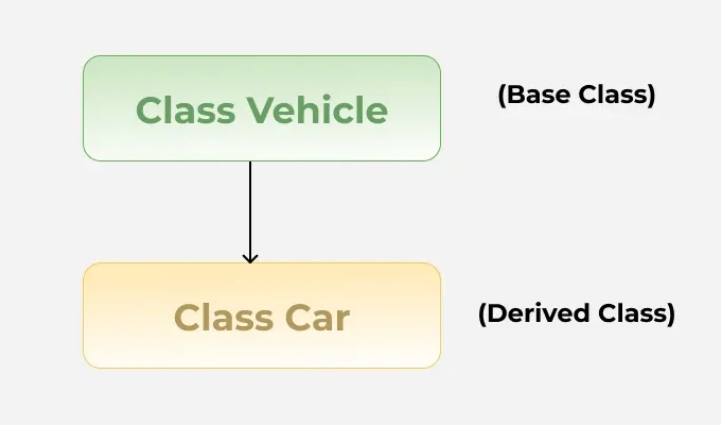
Accessibility	Public Members	Protected Members	Private Members
Base Class	Yes	Yes	Yes
Derived Class	Yes	Yes	No

继承类型

继承可以根据派生类和基类之间的关系进行分类，在C++中，我们有5种继承方式：

1. Single Inheritance

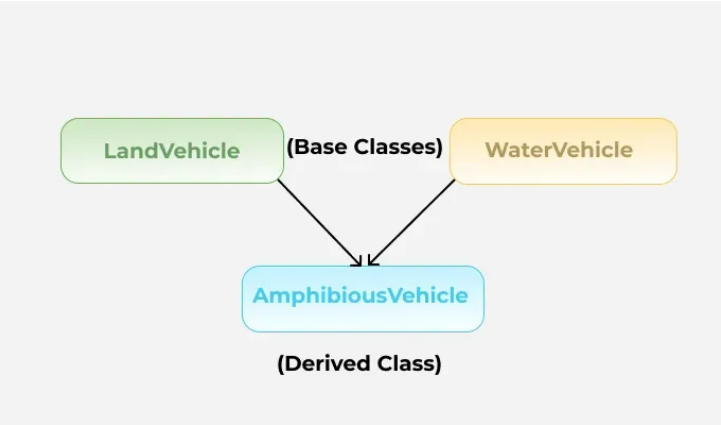
在单继承中，一个类只允许从一个类继承。即一个基类只能被一个派生类继承。




```
class Vehicle {
public:
    Vehicle() {cout << "This is a Vehicle"<< endl;}
};
class Car : public Vehicle {
public:
    Car() {cout << "This Vehicle is Car"<< endl;}
};
int main() {
    // Creating object of sub class will invoke the constructor of base classes
    Car obj;
    return 0;
} // Output: This is a Vehicle   This Vehicle is Car
```

2. Multiple Inheritance

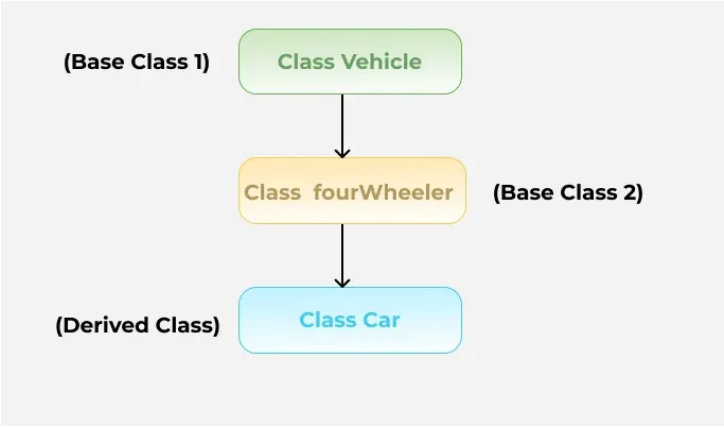
多重继承是 C++的一个特性，其中类可以从多个类中继承。也就是说，一个子类可以继承多个基类。




```
class LandVehicle {
public:
    LandVehicle() {cout << "This is a LandVehicle"<< endl;}
};
class WaterVehicle {
public:
    WaterVehicle() {cout << "This is a WaterVehicle"<< endl;}
};
class AmphibiousVehicle : public WaterVehicle, public LandVehicle {
  public:
    AmphibiousVehicle() {cout << "This is an AmphibiousVehicle"<< endl;}
};
int main() {
    // Creating object of sub class will invoke the constructor of base classes.
    AmphibiousVehicle obj;
    return 0;
} // Output: This is a WaterVehicle   This is a LandVehicle   This is an
AmphibiousVehicle
```

3. Multilevel Inheritance

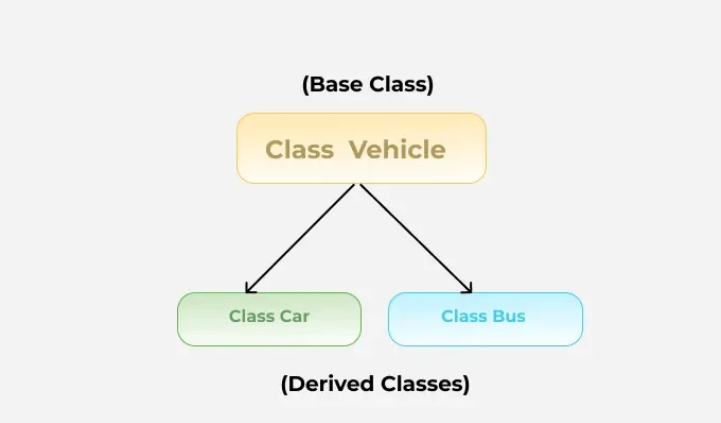
在多级继承中，一个派生类是从另一个派生类创建的，并且这个派生类可以是从基类或任何其他派生类派生的。可以有任意数量的级别。例如，一个车辆可以是一个四轮车，而一个四轮车辆可以是一辆车。




```
class Vehicle {
public:
    Vehicle() {cout << "This is a Vehicle"<< endl;}
};
class fourWheeler : public Vehicle {
public:
    fourWheeler() {cout << "4 Wheeler Vehicles"<< endl;}
};
class Car : public fourWheeler {
public:
    Car() {cout << "This 4 Wheeler Vehical is a Car";}
};
int main() {
    // Creating object of sub class will invoke the constructor of base classes.
    Car obj;
    return 0;
} // Output: This is a Vehicle   4 Wheeler Vehicles   This 4 Wheeler Vehical is a Car
```

4. Hierarchical Inheritance

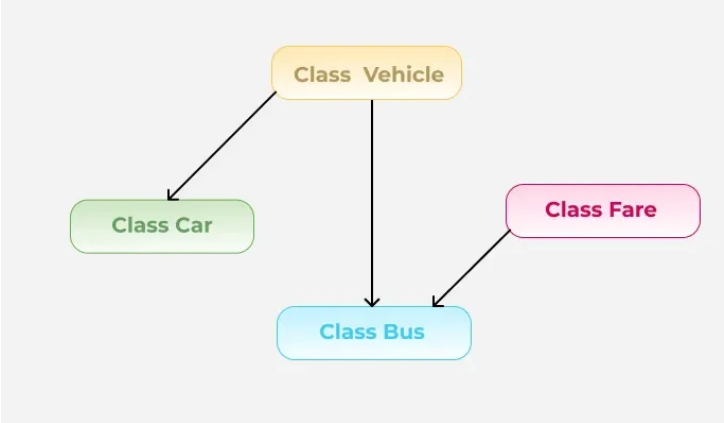
在层次继承中，一个基类可以派生出多个子类。也就是说，一个基类可以派生出多个派生类。例如，汽车和公共汽车都是交通工具。




```
class Vehicle {
public:
    Vehicle() {cout << "This is a Vehicle"<< endl;}
};
class Car : public Vehicle {
public:
    Car() {cout << "This Vehicle is Car"<< endl;}
};
class Bus : public Vehicle {
public:
    Bus() {cout << "This Vehicle is Bus"<< endl;}
};
int main() {
    // Creating object of sub class will invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
} // Output: This is a Vehicle
  // This Vehicle is Car
  // This is a Vehicle
  // This Vehicle is Bus
```

Hybrid Inheritance

混合继承是通过组合多种继承类型来实现的。例如：将层次继承和多重继承结合起来将在C++中创建混合继承。混合继承没有特定的语法。我们只需组合上述几种继承类型中的任意两种。下面的图片显示了层次继承和多重继承的一种组合：



多态和继承的区别

Inheritance	Polymorphism
继承是指创建一个新类（派生类），该类从已经存在的类（基类）中继承特性。	多态性则可以定义为多种形式。
它基本上应用于类	它基本上应用于函数或方法
继承支持可重用性，并在面向对象编程中减少代码长度。	多态允许对象在编译时（重载）以及运行时（覆盖）决定要实现的功能形式。
继承可以是单继承、混合继承、多重继承、层次继承和多级继承。	多态可以是编译时多态（重载）以及运行时多态（覆盖）。
类“bike”可以从“two-wheel vehicles”的类继承，这又可以成为“vehicles”的子类。	自行车类可以有函数 set_color()，该方法根据您输入的颜色名称更改自行车的颜色。
用于模式设计	用于模式设计