Directory

- 类 & 对象
- 继承
- 重载运算符和重载函数
- 多态
- 数据抽象
- 数据封装
- 接口（抽象类）

类 & 对象

类

类定义

- 定义类意味着定义了类的对象包括了什么，以及可以在这个对象上执行哪行操作

```
class Box
{
   public:
   // public 确定了类成员的访问属性，在类对象作用域内，公共成员在类的外部是可访问的。
      double length;   // 盒子的长度
      double breadth;  // 盒子的宽度
      double height;   // 盒子的高度
};
```

类访问修饰符

类成员的访问限制

- **public**
- **private**
- **protected**

```
class Base {

   public:

  // 公有成员
  //在程序中类的外部是可访问的。
  //可以不使用任何成员函数来设置和获取公有变量的值

   protected:
```

```
   // 受保护成员
   //与私有成员十分相似，但有一点不同，protected（受保护）成员在派生类（即子类）中是可访问的。

    private:

   // 私有成员
   //成员和类的默认访问修饰符是 private
   //在类的外部是不可访问的，甚至是不可查看的。
   //只有类和友元函数可以访问私有成员。

};
```

继承中的特点

有public, protected, private三种继承方式，它们相应地改变了基类成员的访问属性。

- 1.public 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：public, protected, private
- 2.protected 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：protected, protected, private
- 3.private 继承：基类 public 成员，protected 成员，private 成员的访问属性在派生类中分别变成：private, private, private

但无论哪种继承方式，上面两点都没有改变：

- 1.private 成员只能被本类成员（类内）和友元访问，不能被派生类访问；
- 2.protected 成员可以被派生类访问。

对象

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;   // 长度
      double breadth;  // 宽度
      double height;   // 高度
      // 成员函数声明
      double get(void);
      void set( double len, double bre, double hei );
};
// 成员函数定义
double Box::get(void)
{
```

```
    return length * breadth * height;
}

void Box::set( double len, double bre, double hei)
{
    length = len;
    breadth = bre;
    height = hei;
}
int main( )
{
   Box Box1;        // 声明 Box1, 类型为 Box
   double volume = 0.0;     // 用于存储体积

   // box 1 详述
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   Box1.set(16.0, 8.0, 12.0);
   volume = Box1.get();
   cout << "Box1 的体积: " << volume <<endl;
   return 0;
}
```

类成员函数

- 类的成员函数是指把定义和原型写在类定义内部的函数，就像类定义中的其他变量一样。
- 类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。
- 成员函数可以定义在
    - 类定义内部，在类定义中定义的成员函数把函数声明为内联的，即便没有使用 inline 标识符。
    - 单独使用范围解析运算符 :: 来定义。

```
class Box
{
   public:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
    //在类定义中定义
      double getVolume(void)
      {
         return length * breadth * height;
      }
};
//在类的外部使用范围解析运算符 :: 定义
//在 :: 运算符之前必须使用类名
double Box::getVolume(void)
{
    return length * breadth * height;
}
```

```
//调用成员函数是在对象上使用点运算符（.）
Box myBox;           // 创建一个对象
myBox.getVolume();   // 调用该对象的成员函数
```

构造函数和析构函数

构造函数

- 类的构造函数是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。
- 构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。
- 不带参数的构造函数

```
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // 这是构造函数
   private:
      double length;
};

// 成员函数定义，包括构造函数
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
```

- 带参数的构造函数

```
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // 这是构造函数
   private:
      double length;
};

// 成员函数定义，包括构造函数
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
```

```
    length = len;
}
```

- 使用初始化列表来初始化字段

```
C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

如:

```
Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```

析构函数

- 类的析构函数是类的一种特殊的成员函数，它会在每次删除所创建的对象时执行。
- 析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀，它不会返回任何值，也不能带有任何参数。
- 析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源。

```
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // 这是构造函数声明
      ~Line();  // 这是析构函数声明
   private:
      double length;
};

// 成员函数定义，包括构造函数
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}
```

友元函数

- 类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。
- 友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。
- 如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 friend
- 友元函数没有 this 指针

```
//友元函数声明
class Box
{
   double width;
public:
   double length;
   friend void printWidth( Box box );
   void setWidth( double wid );
};
//友元类声明
friend class ClassTwo;//声明类 ClassTwo 的所有成员函数作为类 ClassOne 的友元
```

this指针

- 每一个对象都能通过 this 指针来访问自己的地址。
- this 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。
- 友元函数没有 this 指针，因为友元不是类的成员。只有成员函数才有 this 指针。

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

class Box
{
   public:
      Box(double l=2.0, double b=2.0, double h=2.0)// 构造函数定义
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      int compare(Box box)
      {
         return this->Volume() > box.Volume();
```

```
        }
    private:
        double length;     // Length of a box
        double breadth;    // Breadth of a box
        double height;     // Height of a box
};

int main(void)
{
    Box Box1(3.3, 1.2, 1.5);    // Declare box1
    Box Box2(8.5, 6.0, 2.0);    // Declare box2

    if(Box1.compare(Box2))
        cout << "Box2 is smaller than Box1" <<endl;
    else
        cout << "Box2 is equal to or larger than Box1" <<endl;

    return 0;
}
```

拷贝构造函数

- 拷贝构造函数通常用于：
    - 通过使用另一个同类型的对象来初始化新创建的对象。
    - 复制对象把它作为参数传递给函数。
    - 复制对象，并从函数返回这个对象。

内联函数

- 内联函数是通常与类一起使用。如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。
- 对内联函数进行任何修改，都需要重新编译函数的所有客户端，因为编译器需要重新更换一次所有的代码，否则将会继续使用旧的函数。
- 使用 `inline` 关键字定义为内联函数，在调用函数之前需要对函数进行定义。如果已定义的函数多于一行，编译器会忽略 inline 限定符。
- 在类定义中的定义的函数都是内联函数，即使没有使用 inline 说明符。

指向类的指针

- 一个指向 C++ 类的指针与指向结构的指针类似，访问指向类的指针的成员，需要使用成员访问运算符 ->，就像访问指向结构的指针一样。

```
class Box
{
    public:
```

```
        // 构造函数定义
        Box(double l=2.0, double b=2.0, double h=2.0)
        {
            cout <<"Constructor called." << endl;
            length = l;
            breadth = b;
            height = h;
        }
        double Volume()
        {
            return length * breadth * height;
        }
    private:
        double length;     // Length of a box
        double breadth;    // Breadth of a box
        double height;     // Height of a box
};

int main(void)
{
    Box Box1(3.3, 1.2, 1.5);    // Declare box1
    Box Box2(8.5, 6.0, 2.0);    // Declare box2
    Box *ptrBox;                // Declare pointer to a class.

    // 保存第一个对象的地址
    ptrBox = &Box1;

    // 现在尝试使用成员访问运算符来访问成员
    cout << "Volume of Box1: " << ptrBox->Volume() << endl;

    // 保存第二个对象的地址
    ptrBox = &Box2;

    // 现在尝试使用成员访问运算符来访问成员
    cout << "Volume of Box2: " << ptrBox->Volume() << endl;

    return 0;
}
```

类的静态成员

- 可以使用 static 关键字来把类成员定义为静态的。
- 当声明类的成员为静态时，意味着无论创建多少个类的对象，静态成员都只有一个副本。
- 静态成员在类的所有对象中是共享的。
- 如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。
- 不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 :: 来重新声明静态变量从而对它进行初始化

```
class Box
{
```

```
    public:
       static int objectCount;
       // 构造函数定义
       Box(double l=2.0, double b=2.0, double h=2.0)
       {
          cout <<"Constructor called." << endl;
          length = l;
          breadth = b;
          height = h;
          // 每次创建对象时增加 1
          objectCount++;
       }
       double Volume()
       {
          return length * breadth * height;
       }
    private:
       double length;     // 长度
       double breadth;    // 宽度
       double height;     // 高度
};

// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // 声明 box1
   Box Box2(8.5, 6.0, 2.0);    // 声明 box2

   // 输出对象的总数
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

静态成员函数

- 如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，静态函数只要使用类名加范围解析运算符 :: 就可以访问。
- 静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。
- 静态成员函数有一个类范围，他们不能访问类的 this 指针。可以使用静态成员函数来判断类的某些对象是否已被创建。
- 静态成员函数与普通成员函数的区别：
    - 静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。
    - 普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针。

继承

继承允许我们依据另一个类来定义一个类，这个已有的类称为基类，新建的类称为派生类。

```
// 基类
class Animal {
    // eat() 函数
    // sleep() 函数
};


//派生类
class Dog : public Animal {
    // bark() 函数
};
```

基类 & 派生类

- 定义一个派生类，使用类派生列表来指定基类。
- 类派生列表以一个或多个基类命名
- 派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private。
- 一个派生类继承了所有的基类方法，但下列情况除外：
    - 基类的构造函数、析构函数和拷贝构造函数。
    - 基类的重载运算符。
    - 基类的友元函数。

多继承

多继承即一个子类可以有多个父类，它继承了多个父类的特性

```
class Shape // 基类 Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};
```

```
class PaintCost // 基类 PaintCost
{
   public:
      int getCost(int area)
      {
         return area * 70;
      }
};

// 派生类
class Rectangle: public Shape, public PaintCost
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // 输出对象的面积
   cout << "Total area: " << Rect.getArea() << endl;

   // 输出总花费
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0;
}
```

重载运算符和重载函数

- 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为函数重载和运算符重载
- 重载声明是指一个与之前已经在该作用域内声明过的函数或方法具有相同名称的声明，但是它们的参数列表和定义（实现）不相同。

函数重载

- 可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。
- 不能仅通过返回类型的不同来重载函数

运算符重载

- 可以重定义或重载大部分 C++ 内置的运算符来使用自定义类型的运算符

```
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;
}
// 把两个对象相加，得到 Box3
Box3 = Box1 + Box2;
```

C++ 多态

- 当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。
- 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

虚函数

- 使用关键字 virtual 声明虚函数。
- 在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。
- 我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或后期绑定。

纯虚函数

- 想要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是您在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。
- `=0` 告诉编译器，函数没有主体，上面的虚函数是纯虚函数

```
    // pure virtual function
    virtual int area() = 0;
```

C++ 数据抽象

- 数据抽象是指，只向外界提供关键信息，并隐藏其后台的实现细节，即只表现必要的信息而不呈现细节。
- 数据抽象是一种依赖于接口和实现分离的编程（设计）技术。

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

class Adder{
   public:
      // 构造函数
      Adder(int i = 0)
      {
        total = i;
      }
      // 对外的接口
      void addNum(int number)
      {
          total += number;
      }
      // 对外的接口
      int getTotal()
      {
          return total;
      };
   private:
      // 对外隐藏的数据
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

数据封装

- 两个基本要素
  - 程序语句（代码）：这是程序中执行动作的部分，它们被称为函数。
  - 程序数据：数据是程序的信息，会受到程序函数的影响。
- 封装是面向对象编程中的把数据和操作数据的函数绑定在一起的一个概念，这样能避免受到外界的干扰和误用，从而确保了安全。数据封装引申出了另一个重要的 OOP 概念，即**数据隐藏**。
- **数据封装**是一种把数据和操作数据的函数捆绑在一起的机制，**数据抽象**是一种仅向用户暴露接口而把具体的实现细节隐藏起来的机制。
- 通常情况下，我们都会设置类成员状态为私有（private）

C++ 接口（抽象类）

- 如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "= 0" 来指定的

```
class Box
{
   public:
      // 纯虚函数
      virtual double getVolume() = 0;
   private:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
};
```

- 设计抽象类（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为接口使用。如果试图实例化一个抽象类的对象，会导致编译错误。
- 因此，如果一个 ABC 的子类需要被实例化，则必须实现每个虚函数，这也意味着 C++ 支持使用 ABC 声明接口。如果没有在派生类中重写纯虚函数，就尝试实例化该类的对象，会导致编译错误。
- 可用于实例化对象的类被称为具体类。

```
#include <iostream>

using namespace std;

// 基类
class Shape
{
public:
   // 提供接口框架的纯虚函数
   virtual int getArea() = 0;
   void setWidth(int w)
   {
      width = w;
   }
   void setHeight(int h)
   {
      height = h;
   }
protected:
   int width;
   int height;
};

// 派生类
class Rectangle: public Shape
{
public:
   int getArea()
```

```
    {
        return (width * height);
    }
};
class Triangle: public Shape
{
public:
    int getArea()
    {
        return (width * height)/2;
    }
};

int main(void)
{
    Rectangle Rect;
    Triangle  Tri;

    Rect.setWidth(5);
    Rect.setHeight(7);
    // 输出对象的面积
    cout << "Total Rectangle area: " << Rect.getArea() << endl;

    Tri.setWidth(5);
    Tri.setHeight(7);
    // 输出对象的面积
    cout << "Total Triangle area: " << Tri.getArea() << endl;

    return 0;
}
```