类和结构 Class和 Struct

类（class) 和 结构（Struct）

1. Class 中的变量不加访问说明符(public、private、protected) 的话是 private.
2. Struct 中的变量不加访问说明符(public、private、protected) 的话是 public.

类成员概述

- 例子

```
// TestRun.h

class TestRun
{
    // Start member list.

    //The class interface accessible to all callers.
public:
    // Use compiler-generated default constructor:
    TestRun() = default;
    // Don't generate a copy constructor:
    TestRun(const TestRun&) = delete;
    TestRun(std::string name);
    void DoSomething();
    int Calculate(int a, double d);
    virtual ~TestRun();
    enum class State { Active, Suspended };

    // Accessible to this class and derived classes only.
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();

    // Accessible to this class only.
private:
    // Default brace-initialization of instance members:
    State _state{ State::Suspended };
    std::string _testName{ "" };
    int _index{ 0 };

    // Non-const static member:
    static int _instances;
    // End member list.
};

// Define and initialize static member.
int TestRun::_instances{ 0 };
```

成员访问控制

成员访问控制

- 使用访问控制，可以将类的 **公共** 接口与 **专用** 实现详细信息和仅供派生类使用的 **受保护** 成员分离。

```
class Point
{
public:
    Point( int, int ) // Declare public constructor.;
    Point();// Declare public default constructor.
    int &x( int ); // Declare public accessor.
    int &y( int ); // Declare public accessor.

private:                 // Declare private state variables.
    int _x;
    int _y;

protected:      // Declare protected function for derived classes only.
    Point ToWindowCoords();
};
```

- 访问类型以及含义
    - private: 声明为的类成员只能 private 由类的**成员函数**和**友元 (类或函数)** 使用。
    - protected 声明为的类成员 protected 可由成员函数和友元 (类的类或函数) 使用。 此外，它们还可由**派生自该类的类**使用。
    - Public 声明为的类成员 public 可由**任何**函数使用。

派生类中的访问控制

- 两个因素控制基类的哪些成员可在派生类中访问；这些相同的因素控制对派生类中的继承成员的访问：
    - 派生类是否使用 public 访问说明符声明基类。
    - 基类中对成员的访问权限如何。

```
class B:private A{ };  //私有派生
class C:protected A{ };  //保护派生
class D:public A{ }; //公有派生
```

private	protected	public
无论派生访问权限如何，始终不可访问	如果使用私有派生，则在派生类中为私有	如果使用私有派生，则在派生类中为私有
	如果使用受保护派生，则在派生类中为受保护	如果使用受保护派生，则在派生类中为受保护

private	protected	public
	如果使用公有派生，则在派生类中为受保护	如果使用公有派生，则在派生类中为公有

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

- 例子

```
// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
    int PublicFunc(); // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc(); // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
        // error: ‘int BaseClass::PrivateFunc()’ is private within this
context
    }
};

class DerivedClass2 : private BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

int main()
{
```

```
    DerivedClass1 derived_class1;
    DerivedClass2 derived_class2;
    derived_class1.PublicFunc();
    derived_class2.PublicFunc(); // function is inaccessible
    // error: ‘int BaseClass::PrivateFunc()’ is private within this context
}
```

在 DerivedClass1 中，成员函数 PublicFunc 是公共成员，ProtectedFunc 是受保护成员，因为 BaseClass 是公共基类。 PrivateFunc 对于 BaseClass 是私有的，因此任何派生类都无法访问它。

在 DerivedClass2 中，函数 PublicFunc 和 ProtectedFunc 被视为私有成员，因为 BaseClass 是私有基类。 同样，PrivateFunc 对于 BaseClass 是私有的，因此任何派生类都无法访问它。

- 访问控制和静态成员
  - 将基类指定为时 private ，它只会影响非静态成员。 在派生类中，public 静态成员仍是可访问的。
- 对虚函数的访问
  - 应用于虚函数的访问控制由用于进行函数调用的类型确定。 重写函数的声明不会影响给定类型的访问控制。

友元

- 类可以允许其他类或者函数访问它的非公有成员， 方法是令其他类或者函数成为它的友元。只需要增加一条以friend关键字开始的函数声明语句。

```
class Sales_data{
// 为 Sales_data的非成员函数所做的友元声明
    friend Sales_data add(const Sales_data&, const Sales_data&);
    friend std::istream &read(std::istream&, Sales_data&);
    friend std::ostream &print(std::ostream&, Sales_data&);
//其他不变
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):bookNo(s),
units_sold(n), revenue(p*n){}
    Sales_data(const std::string &s):bookNo(s){}
    Sales_data(std::istream &s);
    std::string isbn() const {return bookNo;}
private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0;
};
```

- 类成员函数可以声明为其他类中的友元

```
// classes_as_friends1.cpp
// compile with: /c
class B;
```

```
class A {
public:
int Func1( B& b );

private:
int Func2( B& b );
};

class B {
private:
int _b;

// A::Func1 is a friend function to class B
// so A::Func1 has access to all members of B
friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

- 在前面的B类中，仅为函数 `A::Func1( B& )` 授予对类 B 的友元访问权限。

Private 关键字

- 在类成员列表前面时， private 关键字指定只能从类的成员函数和友元访问这些成员。
- 在基类的名称前面时， private 关键字指定基类的公共成员和受保护成员是派生类的私有成员。(private > protected > public)
- 类中成员的默认访问是 private 的。Struct或Union中成员的默认访问是public的。
- 例子

```
    #include<iostream>
#include<assert.h>
using namespace std;
class A{
public:
int a;
A(){
    a1 = 1;
    a2 = 2;
    a3 = 3;
    a = 4;
}
void fun(){
    cout << a << endl;    //正确
    cout << a1 << endl;   //正确
    cout << a2 << endl;   //正确
    cout << a3 << endl;   //正确
}
public:
int a1;
```

```
protected:
int a2;
private:
int a3;
};
class B : private A{
public:
int a;
B(int i){
    A();
    a = i;
}
void fun(){
    cout << a << endl;        //正确，public成员。
    cout << a1 << endl;        //正确，基类public成员,在派生类中变成了private,可以被派生类访问。
    cout << a2 << endl;        //正确，基类的protected成员，在派生类中变成了private,可以被派生类访问。
    cout << a3 << endl;        //错误，基类的private成员不能被派生类访问。
}
};
int main(){
B b(10);
cout << b.a << endl;        //正确。public成员
cout << b.a1 << endl;       //错误，private成员不能在类外访问。
cout << b.a2 << endl;       //错误, private成员不能在类外访问。
cout << b.a3 << endl;       //错误，private成员不能在类外访问。
system("pause");
return 0;
}
```

Protected 关键字

- 声明为 protected的类成员只能由以下方式访问:
  - 仅可以由类和友元函数，以及派生类可以访问。

```
// class A 与之前一样

class B : protected A{
public:
int a;
B(int i){
    A();
    a = i;
}
void fun(){
    cout << a << endl;        //正确，public成员。
    cout << a1 << endl;        //正确，基类的public成员，在派生类中变成了protected，可以被派生类访问。
    cout << a2 << endl;        //正确，基类的protected成员，在派生类中还是protected，可以被派生类访问。
```

```
    cout << a3 << endl;       //错误，基类的private成员不能被派生类访问。
}
};
int main(){
B b(10);
cout << b.a << endl;        //正确。public成员
cout << b.a1 << endl;       //错误，protected成员不能在类外访问。
cout << b.a2 << endl;       //错误，protected成员不能在类外访问。
cout << b.a3 << endl;       //错误，private成员不能在类外访问。
system("pause");
return 0;
}
```

Public 关键字

- 在类成员列表之前， public 关键字指定可以从任何函数访问这些成员
- 在基类的名称前面时， public 关键字指定基类的公共成员和受保护成员分别是派生类的公共和受保护成员。

```
class B : public A{
public:
int a;
B(int i){
    A();
    a = i;
}
void fun(){
    cout << a << endl;        //正确，public成员
    cout << a1 << endl;        //正确，基类的public成员，在派生类中仍是public成员。
    cout << a2 << endl;        //正确，基类的protected成员，在派生类中仍是protected可以被派生类访问。
    cout << a3 << endl;        //错误，基类的private成员不能被派生类访问。
}
};
int main(){
B b(10);
cout << b.a << endl;
cout << b.a1 << endl;    //正确
cout << b.a2 << endl;    //错误，类外不能访问protected成员
cout << b.a3 << endl;    //错误，类外不能访问private成员
system("pause");
return 0;
}
```

大括号初始化

- 对于简单的类，可以使用 uniform initialization来初始化类或者结构。
- 当类或结构没有构造函数时，将按在类中声明成员的顺序提供列表元素。

```
#include <stdio.h>
#include <string>
using namespace std;
class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};

int main()
{
    class_d d1{};
    class_d d2{ 4.5 };
    class_d d3{ 4.5, "string" };
    class_d d4{ 4.5, "string", 'c' };

    class_d d5{ "string", 'c' }; // compiler error
    class_d d6{ "string", 'c', 2.0 }; // compiler error
}
```

构造函数

- 构造函数具有与类相同的名称，没有返回值。 你可以根据需要定义任意多个重载构造函数。构造函数通常是 public的，以便类外的代码可以调用构造函数。但是也可以声明为 protected 或者 private。
- 默认构造函数 通常没有参数，但它们可以包含具有默认值的参数。

```
class Box {
public:
    Box() { /*perform any required default initialization steps*/}

    // All params have default values
    Box (int w = 1, int l = 1, int h = 1): m_width(w), m_height(h),
m_length(l){}
...
}
```

- 默认构造函数是一种 特殊的成员函数。 如果未在类中声明任何构造函数，则编译器将提供隐式的 inline 默认构造函数。次方法需要在类定义中初始化成员。

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

class Box {
public:
    int Volume() {return m_width * m_height * m_length;}
private:
```

```
    int m_width { 0 };
    int m_height { 0 };
    int m_length { 0 };
};

int main() {
    Box box1; // Invoke compiler-generated constructor
    cout << "box1.Volume: " << box1.Volume() << endl; // Outputs 0
}
```

- 可以通过将默认构造函数定义为 已删除，阻止编译器生成隐式默认构造函数：

```
// Default constructor
Box() = delete;
```

复制构造函数

- 复制构造函数 通过从同一类型的对象复制成员值来初始化对象。

```
Box(Box& other); // Avoid if possible--allows modification of other.
Box(const Box& other);
Box(volatile Box& other);
Box(volatile const Box& other);

// Additional parameters OK if they have default values
Box(Box& other, int i = 42, string label = "Box");
```

移动构造函数

- 移动构造函数 是一种特殊的成员函数，它将现有对象的数据的所有权转移到新的变量，而不复制原始数据。它采用右值引用作为其第一个参数，任何其他参数都必须具有默认值。

```
Box(Box&& other);
```

显式默认的和删除的构造函数

- 可以显式 默认 复制构造函数、默认构造函数、移动构造函数、复制赋值运算符、移动赋值运算符和析构函数。

```
class Box
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
```

```
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

显式构造函数

- 如果类具有带一个参数的构造函数，或是如果除了一个参数之外的所有参数都具有默认值，则参数类型可以隐式转换为类类型。

  例如

  ```
  Box(int size): m_width(size), m_length(size), m_height(size){}
  ```

  则可以这样初始化Box对象

  ```
  Box b = 42;
  ```

To be continued

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。
- 声明析构函数：
  - 不接受自变量。
  - 不要将值返回 (或 void) 。
  - 不能声明为 const 、volatile 或 static 。但是，可以调用它们来销毁作为、或声明的对象 const volatile static 。
  - 可以声明为 virtual 。通过使用虚拟析构函数，无需知道对象的类型即可销毁对象 - 使用虚函数机制调用该对象的正确析构函数。 请注意，析构函数也可以声明为抽象类的纯虚函数。
- 例子：

```
#include <iostream>

using namespace std;

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;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
Line line;

// 设置长度
line.setLength(6.0);
cout << "Length of line : " << line.getLength() <<endl;

return 0;
}
```

代码输出：

```
Object is being created
Length of line : 6
Object is being deleted
```

成员函数

成员函数概述

- 成员函数是静态或非静态的。
- 静态成员函数的行为与其他成员函数的行为不同，因为静态成员函数没有隐式 this 自变量。 非静态成员函数具有 this 指针。
- 可以在类声明的内部或外部定义成员函数（无论是静态的还是非静态的）。
- 在类的外部声明成员函数，需要通过 :: 域运算符限定函数名的作用域。

```
//在类外部声明成员函数
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}
```

virtual speficier 虚拟说明符

- 见虚函数

override speficier 重写说明符

- 重写(Override)是指子类重新定义的父类中有相同的名称或者参数的虚函数，主要在继承关系中出现。语法如下：

```
function-declaration override;
```

- 基本条件
  - 子类重写父类中的virtual函数
  - 重写函数和被重写函数的函数名和参数必须一致(具体实现一般不同)
  - 重写函数和被重写函数的返回值相同，要么都返回指针，要么都返回引用
  - 重写函数和被重写函数都是virtual函数(其中重载函数可以带virtual，也可以不带)
  - 静态方法不能被重写，也就是static和virtual不能同时使用
  - C++ 11中新增了final关键字，`final`修饰的虚函数不能被被重写。

```
//例子
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override;
    // compiler error: DerivedClass::funcB() does not override
BaseClass::funcB() const

    virtual void funcC( double = 0.0 ) override;
    // compiler error:DerivedClass::funcC(double) does not override
BaseClass::funcC(int)
    //参数类型不一样

    void funcD() override;
    // compiler error: DerivedClass::funcD() does not override the non-
virtual BaseClass::funcD()

};
```

- 与重定义（redefining）的区别
  - redefining是子类重定义父类中的非虚函数，屏蔽了父类的同名函数(相当于创建了一个新的函数，跟父类无关)。
  - redefining的条件
    - 子类和父类函数的名称相同，参数也相同，父类中的函数不是virtual，父类的函数将被隐藏。
    - 子类和父类的函数名称相同，但参数不同，此时不管父类函数是不是virtual函数，都将被隐藏。

Final 关键字

- 用途1：阻止子类的继承。

```
class TaskManager {/*..*/} final;
class PrioritizedTaskManager: public TaskManager {
};  //compilation error: base class TaskManager is final
```

- 用途2：阻止一个虚函数的重载。

```
class A
{
pulic:
virtual void func() const;
};
class  B: A
{
pulic:
void func() const override final; //OK
};
class C: B
{
pulic:
void func()const; //error, B::func is final
};
```

const 关键字

- const 成员变量的用法和普通 const 变量的用法相似，只需要在声明时加上 const 关键字。初始化 const 成员变量只有一种方法。
- const 成员函数可以使用类中的所有成员变量，但是不能修改它们的值。
- 我们通常将 get 函数设置为常成员函数。读取成员变量的函数的名字通常以get开头

```
class Student{
public:
    Student(char *name, int age, float score);
    void show();
    //声明常成员函数
    char *getname() const;
    int getage() const;
    float getscore() const;
private:
    char *m_name;
    int m_age;
    float m_score;
};

Student::Student(char *name, int age, float score): m_name(name),
m_age(age), m_score(score){ }
void Student::show(){
    cout<<m_name<<"的年龄是"<<m_age<<"，成绩是"<<m_score<<endl;
}
//定义常成员函数，只能读不能修改成员变量
char * Student::getname() const{
    return m_name;
}
int Student::getage() const{
    return m_age;
}
float Student::getscore() const{
```

```
    return m_score;
}
```

继承

单个继承 Single Inheritance

- 派生类只有一个基类。例如 Book 既是派生类（来自 PrintedDocument），又是基类（PaperbackBook 派生自 Book）

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

PrintedDocument 被视为 Book 的“直接基”类；它是 PaperbackBook 的“间接基”类。 差异在于，直接基类出现在类声明的基础列表中，而间接基类不是这样的。

多个基类 Multiple Base Classes

- 一个类可以从多个基类派生。 在多重继承模型中 (其中的类派生自多个基类)，则使用 " 基本列表 " 语法元素指定基类。 `CollectionOfBook`派生自`Collection`和`Book`

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

虚拟基类 Virtual base classes

- 由于一个类可能多次成为派生类的间接基类，因此 C++ 提供了一种优化这种基类的工作方式的方法。用虚拟基类可以节省内存。

LunchQueue 和 CashierQueue 分别继承自 Queue类， LunchCashierQueue 继承自 LunchQueue和 CashierQueue。
只有一个Queue类在物理内存中。而不是两个。

```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

显式重写

- 如果在两个或更多个 接口 中声明了同一虚函数，并且如果类派生自这些接口，则可以显式重写每个虚函数。
    - 假设有两个具有相同纯虚方法的类。

```
class InterfaceA
{
    public: virtual void doSomething() = 0;
};
class InterfaceB
{
    public: virtual void doSomething() = 0;
};
```

    - 想在派生类中覆盖每个虚拟功能，需要

```
class ConcreteClass : public InterfaceA, public InterfaceB
{
public:
    void InterfaceA::doSomething() override {
        InterfaceA_doSomething();
    }

    void InterfaceB::doSomething() override {
        InterfaceB_doSomething();
    }

private:
    void InterfaceA_doSomething();
    void InterfaceB_doSomething();
};
```