# 类的基本思想是？

类的基本思想是 数据抽象（data abstraction）和封装（encapsulation）。

# 数据抽象（data abstraction） 是什么？

**数据抽象**：一种依赖于 接口（interface）和实现（implementation）分离的编程技术。

**接口**：类的接口包括用户可以执行的操作；

**实现**：类的实现包括 类的数据成员、负责接口实现的函数体、定义类需要的各种私有函数

# 封装（encapsulation）的作用是什么？

封装实现了 类的接口 和 实现的分离，封装后的类隐藏了它的实现细节，也就是说，类的用户 只能使用接口，但无法看到类的实现

# 什么是成员函数（member function）？

成员函数是 定义为类的一部分 的函数，有时也称为 **方法**（method）。

# 封装有什么好处

1 ) 确保用户代码不会无意间破坏封装对象的状态；

2 ) 被封装的类的具体实现细节可以随时改变 而不影响用户代码，因为用户使用的只是接口，只要接口不变，用户代码就无需改变；

3 ) 类改变后只需编译该类的源文件

# 成员函数的 定义、声明的位置 有什么要求？

成员函数必须在类的内部声明，但它的定义可以在类的内部，也可以在类的外部。

# 非成员函数的定义、声明的位置 什么要求？

非成员函数 一般也与类相关，但它们的定义和声明都在类的外部。

非成员函数的函数原型不需要有

# 成员函数 和 非成员函数的区别？

1 ) 声明的位置不同，成员函数声明在类的内部，而非成员函数声明在类的外部；

2 ) 调用方式不一样

# 成员函数定义在类内还是类外有什么不同吗？

成员函数 定义在类内部的话吗，那它将是隐式的inline函数。

# 成员函数中，紧跟着参数列表的const关键字修饰的是？

这说明该函数是常量成员函数，这个const修饰的是隐式this指针，因为this指针是隐式的（即不会出现的形参列表中），所以才把const写在了形参列表之后，用来说明this指针时 指向常量的指针（this指向的对象 是常量）。

# 什么是this指针？

this指针 是 **类的成员函数的** 一个 **隐式形参**，当我们调用成员函数时，会用 **请求该函数的对象的地址** 初始化 this，后面就通过这个this隐式参数来访问调用它的那个对象，比如：

1. Sales\_item total;
2. total.isbn()

其实就相当于：

1. //伪代码，用于说明调用成员函数的实际执行过程
2. Sales\_data::isbn(&total) // 把调用对象total的地址传进去，成员函数isbn()会用total的地址来初始化this指针

在成员函数内部，我们可以直接使用调用该函数的对象的成员，而无须通过成员访问运算符来做到这一点，因为 this 所指的正是这个对象。任何对类成员的直接访问都被看作是对 this 的隐式引用，也就是说，当 isbn 使用 bookNo 时，它隐式地使用 this 指向的成员，就像我们书写了 this->bookNo 一样。

对于下面的代码，

1. std::string isbn() **const**
2. {
3. **return** bookNo;   // 在编译器看来，它相当于 this->bookNo，因为在成员函数的内
4. // 部，任何对类成员的直接访问都被看作是对 this 的隐式引用。
5. }

其实相当于：

1. std::string isbn() **const**
2. {
3. **return** **this**->bookNo;   // 显式的使用了this指针，其实可以省略this。
4. }

# this指针是什么类型？

它是一个指针常量，是一个顶层const，即this指针自身是一个常量，因此我们不允许改变this指针中保存的地址。

对于类型Sales\_data类的成员函数来说， this指针 的类型是： Sales\_data \***const**

# this指针 对 sizeof（类对象）的影响？

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。

# this指针到底是不是register 变量？

看编译器，有的是，有的不是。

# 友元是否有this指针？

成员函数才有this指针，而友元并不是成员函数，因此它没有。

# 对于class A{public: int func(int p){}}, 在编译器看来,func的原型应该是怎样的？

在编译器看来，成员函数的第1个参数为this指针，所以func的原型应该是：

**int** func(A\* **const this**, **int** p); // 第一个参数应该是this指针

# this指针是什么时候创建的？

对于**class** A{**public**: **int** func(**int** p){}}, 在编译器看来成员函数的第1个参数为this指针，所以func的原型应该是：

**int** func(A\* **const register** **this**, **int** p); // 第一个参数应该是this指针，注意register 关键字，这意味着this是放在寄存器里面的

**由此可见，this在成员函数的开始前构造，在成员函数的结束后清除。**

this指针的生命周期同任何一个函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的this参数传递进去。如：

1. A a;
2. a.func(10);
3. // 此处，编译器将会编译成：
4. A::func(&a, 10);

换句话说，this指针其实是成员函数的隐式形参，既然它是形参，那就意味着它是进入函数时创建，离开函数时销毁。（虽然this是隐式形参，但它有可能存放在栈中，也可能则寄存器中，看编译器！）

# 如果我们知道一个对象this指针的位置，可以直接使用吗？

不能，因为this指针其实是成员函数的隐式形参，既然它是形参，那就意味着它是进入函数时创建，离开函数时销毁，所以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

因此，我们只有获得一个对象后，才能通过对象使用this指针。

# this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器。

# 在成员函数的声明中，形参列表后面还有一个const是什么意思？

这表明这个函数时const成员函数（const member function）

# 常量成员函数有何作用？

1 ) **确保成员函数不会修改调用它的对象**；

2 ) 如果不需要在成员函数中修改调用对象，那么声明为常量成员函数**可以提高 接受参数的灵活性**：

如果调用对象是一个const，那么它将不能调用非常量成员函数，this虽然是隐式的，但它还是得遵守初始化原则：不能用const对象来初始化 非cosnt对象，因此我们不能以常量对象来调用非常量成员函数，这相当于：

1. //伪代码
2. Sales\_data \* **const** **this**;  // this是顶层const
3. **const** Sales\_data total;   // total是底层const
4. **this** = &total             // 错误，不能用const对象来初始化 非cosnt对象

# 对于下面的类，bookNo成员定义在isbn()后面，那isbn()能否使用bookNo成员呢？为什么？

* 1. **struct** Sales\_data {
  2. // new members: operations on Sales\_data objects
  3. std::string isbn() **const** { **return** bookNo; }
  4. Sales\_data& combine(**const** Sales\_data&);
  5. **double** avg\_price() **const**;
  6. // data members are unchanged from § 2.6.1 (p. 72)
  7. std::string bookNo;
  8. unsigned units\_sold = 0;
  9. **double** revenue = 0.0;
  10. };
  11. // nonmember Sales\_data interface functions
  12. Sales\_data add(**const** Sales\_data&, **const** Sales\_data&);
  13. std::ostream &print(std::ostream&, **const** Sales\_data&);
  14. std::istream &read(std::istream&, Sales\_data&);

虽然，bookNo成员定义在isbn()后，但是isbn()还是可以使用bookNo成员。

因为编译器分两步处理类：

① 首先编译所有 类成员的声明；

② 然后才编译 函数体；

因此后声明的函数还是可以声明在他前面声明的成员。

# 在类外 定义成员函数需要注意什么？

① 函数名、形参列表 必须和类内匹配；

② 外部定义的成员 必须包含 它所属的类名（注意类名所在的位置）：

1. // 注意类名所在的位置，它在 返回值 和 函数名 之间
2. **double** Sales\_data::avg\_price() **const** {
3. **if** (units\_sold)
4. **return** revenue/units\_sold;
5. **else**
6. **return** 0;
7. }

# 如何设计一个返回 this对象 的函数？

函数的返回值的类型 必须和 return的对象 匹配：

1. // 解引用this之后得到的是Sales\_data对象，因此返回值类型应该是 Sales\_data的引用
2. Sales\_data& Sales\_data::combine(**const** Sales\_data &rhs)
3. {
4. units\_sold += rhs.units\_sold; // add the members of rhs into
5. revenue += rhs.revenue; // the members of ''this'' object
6. **return** \***this**;  // 解引用后得到的是 Sales\_data对象
7. }
8. total.combine.(trans); // 更新变量total的当前值

## 1 ) 对于调用 total.combine(trans) ，它的绑定情况是怎样的？

trans绑定到rhs形参，total绑定到this指针。

# 构造函数的作用是？

构造函数（constructor）是类用来 控制其对象的初始化过程 的函数。

# 构造函数有什么特点？

1 ) 构造函数的名字和类的名字相同；

2 ) 没有返回类型；

3 ) 一个类可以有多个构造函数，但是它们的实参列表必须不一样；

4 ) 不能声明为const（const成员函数）

# 一个类可以没有构造函数吗？

可以，如果类的设计者 没有定义 任何构造函数，那编译器会隐式的定义一个默认构造函数，这个隐式定义的函数也叫 合成的默认构造函数（synthesized default constructor）。

# 谁来控制类的默认初始化？

默认构造函数用来控制

# 默认构造函数什么时候被调用

若定义一个对象时没有提供初始化式，就使用默认构造函数，比如：

Sales\_data total; // 没有提供初值，此时调用默认构造函数初始化

# 默认构造函数有何特点？

一个函数是默认构造函数 **当且仅当** 调用它可以不需要传入任何参数，定义默认构造函数有两种方式，

① 没有任何参数；

② 定义的所有参数 都有默认值；

1. **class** testClass
2. {
3. **public**:
4. testClass();                     // 默认构造函数
5. testClass(**int** a, **char** b);        // 构造函数
6. testClass(**int** a=10,**char** b='c');  // 所有参数 都有默认值，所以是默认构造函数
8. **private**:
9. **int**  m\_a;
10. **char** m\_b;
11. };

# 合成的默认构造函数 什么情况下会被定义？

只有在类 **没有定义任何构造函数** 时，那编译器才会隐式的定义一个 合成的默认构造函数（synthesized default constructor）。

# 合成的默认构造函数 的初始化规则是怎样的？

如果存在类内的初始值，则用它来初始化成员；

如果没有类内的初始值，执行默认初始化。

* 1. **struct** Sales\_data {
  2. std::string isbn() **const** { **return** bookNo; }
  3. Sales\_data& combine(**const** Sales\_data&);
  4. **double** avg\_price() **const**;
  5. std::string bookNo;   // 无类内初始值，将执行默认初始化（空串）
  6. unsigned units\_sold = 0;  // 有类内初始值：0，将用来初始化units\_sold
  7. **double** revenue = 0.0;   // 有类内初始值：0.0，将用来初始化revenue
  8. };

# 什么样的类适使用 合用默认构造函数？

如果类包含的 所有内置类型、复核类型 的成员被赋予类内初始值时，这个类才适合使用合成的默认构造函数。

# 哪些类 不能依赖于编译器 合成的默认构造函数？为什么？

**1 ) 已经定义构造函数的类**

如果类内已经显式地定义了构造函数，则编译器不会再合成默认构造函数，也就是说：若不自力更生（自己定义一个默认构造函数），那这个该类将没有默认构造函数：

下面的类Sales\_data**没有**构造函数：

1. **struct** Sales\_data {
2. // 没有 默认构造函数
3. //① 自己定义了构造函数； ② 自己没有显示定义默认构造函数
4. Sales\_data(**const** std::string &s): bookNo(s) { }
5. // 下面省略了成员函数和数据成员
6. 略...
7. };

和上面的不一样，下面的类Sales\_data **有**构造函数：

1. **struct** Sales\_data {
2. // 虽然编译器不再生成默认构造函数，但自己显示的定义了默认构造函数
3. Sales\_data() = **default**;
4. Sales\_data(**const** std::string &s): bookNo(s) { }
5. // 下面省略了成员函数和数据成员
6. 略...
7. };

**2 ) 未给内置类型、复合类型 赋 类内初始值 的类**

其中，内置类型指的是：int等类型，复合类型指的是：数组、指针等类型

我们知道，在定义内置类型、复合类型变量的时候，如果没有指定初值，则变量将被默认初始化，此时变量被赋予了“默认值”，默认初始化的规则为：

(全局变量) 的变量被初始化为0；·

(局部变量) 将不被初始化，此时的值是未定义的（除局部static变量）

因此如果用编译器合成的默认构造函数，用户在创建类对象的时候就可能得到未定义的值，这样是很危险的。

**3 ) 有时，编译器不能为某些类合成默认的构造函数**。

如果类内包含一个其他类类型的成员，而且该成员的类型没有默认构造函数，则无法初始化。此时我们必须自定义默认构造函数，否则将没有可用的默认构造函数。

1. **class** NoDefault {
2. **public**:
3. // 已有构造函数 且未显示定义默认构造函数、也未用 =default要求编译器合成
4. // 所以类NoDefault 没有 默认构造函数
5. NoDefault(**const** std::string&);
6. **int** len = 0;
7. };
8. **struct** A {
9. NoDefault my\_mem;  // 注意！类NoDefault 没有 默认构造函数！
10. **double** balance = 0.0;
11. };
12. A a; // 错误: 类A的成员my\_mem的类型是类NoDefault，而它没有，默认构造函数，因此编译器不能为其合成默认构造函数

# 在effective c++中指出：惟有默认构造函数”被需要“的时候编译器才会合成默认构造函数，怎么理解 “ 被需要 ”？

# Sales\_data() = default;中后面的=default 是什么意思？

1 ) 该构造函数不接受任何实参，所以它是一个默认构造函数

2 ) 在参数列表后写上 “=default”，意思是 要编译器生成一个默认构造函数，这个函数和之前的 合成默认构造函数 的作用完全相同

# 为什么需要 “ =default ”？

**背景**：

C++ 的类有四类特殊的成员函数，分别为：默认构造函数，析构函数，拷贝构造函数以及拷贝赋值函数（TODO：书上说（13.1.5）是默认构造函数或拷贝控制成员，这样一来就有五个，等看到13章再回来）。如果程序没有显式地为一个类定义某个特殊成员函数，而又需要用到该特殊成员函数时，编译器会隐式地为这个类生成一个默认的特殊成员函数。

下面的代码虽然没有定义构造函数，但编译器隐式生成默认构造函数，所以它能编译通过：

1. **class** X {
2. **private**:
3. **int** a;
4. };
5. X x;   //可以编译通过，编译器隐式生成默认构造函数

但下面的代码会出现编译错误，因为如果类自己显式的定义了构造函数，编译器是不会为类生成默认构造函数的：

1. **class** X{
2. **public**:
3. X(**int** i){
4. a = i;
5. }
6. **private**:
7. **int** a;
8. };
10. X x; // 错误，类自己显式的定义了构造函数，编译器是不会为类生成默认构造函数，因此默认构造函数 X::X()不存在

为了解决上面的问题，我们需要自定义默认构造函数，如下代码可正常运行：

1. **class** X{
2. **public**:
3. X(){};  // 手动定义默认构造函数
4. X(**int** i){
5. a = i;
6. }
7. **private**:
8. **int** a;
9. };
11. X x; //正确，自己定义了默认构造函数 X::X()

但是手动编写存在两个问题：

1. 程序员工作量变大

2. 没有编译器自动生成的默认特殊构造函数效率高。

为了解决上述问题，可以在函数后面加上 “ =default ”，把工作交给编译器，相当于 主动让编译器为你生成合成默认构造函数（析构函数，拷贝构造函数以及拷贝赋值函数 同理）

# 哪些函数可以使用 “ =default ” ？

只能对具有合成版本的成员函数使用“ =default ”（即，默认构造函数或拷贝控制成员），TODO：

* 1. **class** Sales\_data {
  2. **public**:
  3. // copy control; use defaults
  4. Sales\_data() = **default**;
  5. Sales\_data(**const** Sales\_data&) = **default**;
  6. Sales\_data& operator=(**const** Sales\_data &);
  7. ~Sales\_data() = **default**;
  8. // other members as before
  9. };
  10. Sales\_data& Sales\_data::operator=(**const** Sales\_data&) = **default**;

# “ =default ”的使用规则？

1) “ =default ” 仅用于类的特殊成员函数，且该特殊成员函数**没有默认参数**。例如：

1. **class** X {
2. **public**:
3. **int** f() = **default**:  // 错误， f()非特殊成员函数
4. X(**int**) = **default**;  // 错误， 非特殊成员函数
5. X(**int** i = 1) = **default**;  // 错误， 含有默认参数
6. }

2) 可以在类内使用=default，也可以在类外：

可以在类内使用=default，这样合成的函数将隐式的声明为内联（就像其它定义都在类内的成员函数一样）；

也可以在类外使用=default，这样就相当于定义在类外，即合成的函数就不是内联的了。

* 1. **class** Sales\_data {
  2. **public**:
  3. Sales\_data() = **default**;  // 隐式声明为内联函数
  4. Sales\_data(**const** Sales\_data&) = **default**;  // 隐式声明为内联函数
  5. Sales\_data& operator=(**const** Sales\_data &);  // 隐式声明为内联函数
  6. ~Sales\_data() = **default**;  // 隐式声明为内联函数
  7. // other members as before
  8. };
  9. Sales\_data& Sales\_data::operator=(**const** Sales\_data&) = **default**; //非内联函数

# 什么是构造函数初始值列表？

构造函数初始化列表 以一个冒号开始，接着是以逗号分隔的数据成员列表，最后跟着一对空花括号，如下面的第二和第三个构造函数：

* 1. **struct** Sales\_data {
  2. // constructors added
  3. Sales\_data() = **default**;
  4. Sales\_data(**const** std::string &s): bookNo(s) { }
  5. Sales\_data(**const** std::string &s, unsigned n, **double** p):  bookNo(s), units\_sold(n), revenue(p\*n) { }
  6. Sales\_data(std::istream &);
  7. // other members as before
  8. std::string isbn() **const** { **return** bookNo; }
  9. Sales\_data& combine(**const** Sales\_data&);
  10. **double** avg\_price() **const**;
  11. std::string bookNo;
  12. unsigned units\_sold = 0;
  13. **double** revenue = 0.0;
  14. };

## Sales\_data(const std::string &s): bookNo(s) { } 只提供了bookNo的值，剩下的units\_sold和revenue怎么初始化？

当某个成员被初始化列表忽略时，它将以 合成默认构造函数相同的方式初始化，这上面的代码中，units\_sold和revenue都有类内初始值，所以他们将 使用类内初始值初始化，因此这个构造函数相当于：

1. // 等价于：Sales\_data(const std::string &s): bookNo(s) { }
2. Sales\_data(**const** std::string &s):bookNo(s), units\_sold(0), revenue(0.0){ }

# 在构造函数中，初始化发生在什么时候？

在构造函数中，成员的初始化发生在 **函数体执行前**，且按照成员在类中出现的顺序进行初始化。

# 使用构造函数初始化列表中初始化 和 在构造函数内部赋值 有区别吗？

有的，初始化列表是 **初始化**，而在构造函数内部是 **赋值，**对于内置类型来说初始化和赋值结果都一样，但是对于 引用 和 const成员，在内部赋值显然是不可以的：构造函数

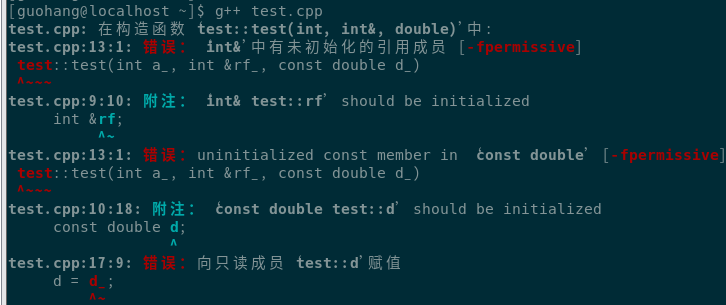
# 哪些成员不能在构造函数内部赋值？

因为在构造函数内部进行的是 **赋值操作**，而有些类型是**不能赋值，只能初始化**的：引用成员变量、const成员变量，因此它们必须在 初始化列表中进行初始化；

某些类类型成员：该类没有默认构造函数，不能进行默认初始化，它们也必须在初始化列表中进行初始化。

1. #include <iostream>
3. **using** **namespace** std;
5. **class** test{
6. **public**:
7. test(**int** , **int** &, **const** **double**);
8. **int** a;
9. **int** &rf;
10. **const** **double** d;
11. };
13. test::test(**int** a\_, **int** &rf\_, **const** **double** d\_)
14. {
15. a = a\_;
16. rf = rf\_;
17. d = d\_;
18. }
20. **int** main()
21. {
22. **int** rf = 2;
23. test(1, rf, 3.0);
25. **return** 0;
26. }

编译运行，结果为：



# 初始化列表的成员初始化顺序 是怎样的？

成员变量**在类中的声明次序** 就是 其在初始化列表中的初始化顺序，与其在初始化列表中的先后次序无关，所以要特别注意，保证两者顺序一致才能真正保证其效率和准确性。

# 可以用一个成员 来初始化另一个成员吗？

可以但是要确保该成员比被初始化的成员后初始化。

# 哪些函数可能拥有编译器合成版本的函数？

# 为什么需要 访问说明符（access specifiers）？

为了数据封装。

# 有哪些访问说明符？

**public：**

公有成员变量、公有成员函数，它可以在被的外部被访问，public成员一般定义类的接口。

**private：**

私有成员变量、私有成员函数，它们在类的外部是不可访问的，甚至是不可查看的。只有类的成员函数和友元函数可以访问私有成员，类的使用者不能访问私有成员（非成员函数也不能访问private成员，除非该非成员函数声明为该类的友元）。private封装了（隐藏了）类的实现细节。

**protected ：**

保护成员变量或函数 与私有成员十分相似，**但有一点不同**，保护成员在派生类（即子类）中是可访问的。

# 访问说明符的有效范围

从访问说明符A出现开始，直到下一个访问说明符B出现，在这期间是访问说明符A的有效范围。

# 同一个访问说明符在一个类里可以出现多次吗？

可以出现多次。

# 使用class和struct有何区别

它们**唯一的区别**是：**默认访问权限不一样**

class：在第一个访问说明符出现之前，成员都是private的

strut：在第一个访问说明符出现之前，成员都是public的

# 为什么需要友元

有一些类虽然不是类的成员（比如说 非成员函数），但也是类的一部分，为了让类的非成员函数可以访问 类的**非公有**成员（包括private和public）。

# 什么可以成为友元？

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；

# 关于友元

TODO：

# 如何声明一个友元函数？

1 ) **在类的内部**，在函数声明前面**加一个friend关键字**即可（建议在类定义的开始或结束的位置集中声明友元）；

2 ) 在外面**再次声明**友元函数

1. **class** Sales\_data {
2. // friend declarations for nonmember Sales\_data operations added
3. **friend** Sales\_data add(**const** Sales\_data&, **const** Sales\_data&);
4. **friend** std::istream &read(std::istream&, Sales\_data&);
5. **friend** std::ostream &print(std::ostream&, **const** Sales\_data&);
6. // other members and access specifiers as before
7. **public**:
8. /\*
9. 公有成员
10. \*/
11. **private**:
12. /\*
13. 私有成员
14. \*/
15. };
16. // declarations for nonmember parts of the Sales\_data interface
17. Sales\_data add(**const** Sales\_data&, **const** Sales\_data&);
18. std::istream &read(std::istream&, Sales\_data&);
19. std::ostream &print(std::ostream&, **const** Sales\_data&);

# 友元的声明在类内出现的位置有限制吗？

没有，友元必须声明在类的内部，但在类内部的具体位置没有要求，因为友元并不是类的成员，这意味着**它不受它所在区域的访问控制级别的约束（声明在private区域也是可以的）**，但建议在类定义的开始或结束的位置集中声明友元。

# 什么是 友元类？

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。

# 如何将成员函数设为内联函数？

1 ) 将成员函数定义在类的内部，这样该成员函数将隐式的被声明为inline函数；

2 ) 在类的内部显式声明为inline，但在定义的时候不加inline：

1. **class** Screen {
2. **public**:
3. /\*
4. 其它成员
5. \*/
6. **inline** **char** get(pos ht, pos wd) **const**; // 类内声明为inline
7. **private**:
8. /\*
9. 其它成员
10. \*/
11. };
12. // 虽然定义的时候没有inline说明符，但已经在类的内部声明为inline了；
13. **char** Screen::get(pos r, pos c) **const**
14. {
15. pos row = r \* width;      // compute row location
16. **return** contents[row + c]; // return character at the given column
17. }

3 ) 在类的内部**没有**显式声明为inline，但在类外部定义的时候加上了inline运算符：

1. **class** Screen {
2. **public**:
3. /\*
4. 其它成员
5. \*/
6. Screen &move(pos r, pos c); // 类内声明没有inline，但在类外定义的时候加上了
7. **private**:
8. /\*
9. 其它成员
10. \*/
11. };
12. // 虽然类内没有声明为inline，但在定义的时候加上了inline
13. **inline** Screen &Screen::move(pos r, pos c)
14. {
15. pos row = r \* width; // compute the row location
16. cursor = row + c;    // move cursor to the column within that row
17. **return** \***this**;        // return this object as an lvalue
18. }

# 什么是 可变数据成员？

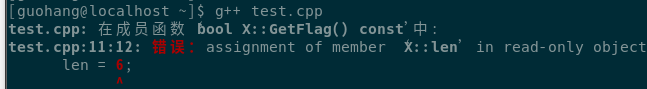
可变数据成员（mutable data member）永远不会是const，即使它是一个const对象的成员，也就是说 可变数据成员 可被 const成员函数修改。

# 为什么需要 可变数据成员？

我们知道，const成员函数的const是修饰this指针的，这就意味着const成员函数不能修改 调用该成员函数的对象，但有的时候，我们又希望在const成员函数中修改某一个成员变量，此时就可以将该将该成员变量声明为 可变数据成员

1. #include <iostream>
3. **using** **namespace** std;
5. **class** X
6. {
7. **public**:
8. **bool** GetFlag() **const**  // const成员函数
9. {
10. m\_accessCount++;   // 正确，m\_accessCount是mutable的；
11. len = 6;   // 错误，len不是mutable的；
12. cout<<"len:"<<len<<endl;
13. **return** m\_flag;
14. }
15. **private**:
16. **bool** m\_flag;
17. **int** len = 0;
18. **mutable** **int** m\_accessCount;
19. };
21. **int** main()
22. {
23. X x;
24. x.GetFlag();
25. }

上面的代码直接编译不过，错误为：



# 如果一个类包含一个 类数据成员，该怎么给它赋初值比较合适？

用“=”或花括号给该类成员一个初始值：

1. **class** Window\_mgr {
2. **private**:
3. // 初始值为 Screen(24, 80, ' ')
4. std::vector<Screen> screens{Screen(24, 80, ' ') };
5. };

# 从 const成员函数 返回 \*this 需要注意什么？

因为const成员函数的const是修饰 this指向的对象的，所以返回的\*this 是一个常量引用，若改变 const成员函数 返回的\*this 的话讲引发错误（修改const对象）：

# 为什么要在 类内 对const成员函数 重载？

1 ) 我们知道，常量对象 不能调用 非常量版本的函数，因为不能用常量实参去初始化非常量的形参，所以我们需要重载const成员函数，一个接收常量对象，一个接收非常量对象，即使它们完成的工作一样。

2 ) 如果需要返回\*this的话，const成员函数返回的是常量对象，这可能造成不方便，需要用一个非常量版本的函数 来返回 非常量对象；

1. **class** Screen {
2. **public**:
3. // 根据对象是否const 重载了display函数
4. Screen &display(std::ostream &os)
5. { do\_display(os); **return** \***this**; }
6. // const成员函数返回的是 const对象
7. **const** Screen &display(std::ostream &os) **const**
8. { do\_display(os); **return** \***this**; }
9. **private**:
10. // function to do the work of displaying a Screen
11. **void** do\_display(std::ostream &os) **const** {os << contents;}
12. // other members as before
13. };

## 上述代码为什么要费力单独定义一个do\_display函数呢？

这样代码比较统一，在项目里，同样的功能最好用同样的代码来完成，这样后面修改也比较简单；

# 如何新建一个类对象？

加不加class都可以：

1. Sales\_data item1;  // 不加class
2. **class** Sales\_data item1; // 也可以加上class，这等价于上面的变量声明

# 在类被 声明之后，定义之前，可以新建该类的对象吗？

不能，对于一个类来说，在外面创建它的对象之前，该类必须被定义，而不能仅仅是被声明。

## 为什么呢？

因为没有定义的话，编译器就不知道这个类占了多少空间，自然不能用来新建对象。

## 有什么折中的办法吗？

在类被 声明之后，定义之前，我们可以新建指向该类的指针，这样也能完成任务。

# 想在类A中修改类B的私有成员，应该怎么做？

1 ) 将 类A 声明为类B的友元类；

2 ) 将 类A中需要修改类B的函数 声明为类B的友元函数；

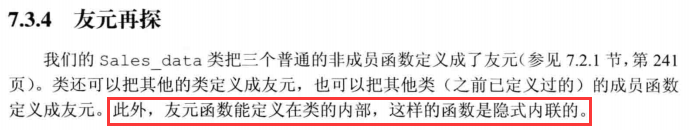
# 如何将 一组重载函数 都 声明为类的友元？

虽然重载函数的函数名相同，但实际上它们是不同的函数，要想将一组重载函数 都 声明为类的友元，需要为这组函数中的每一个函数分别声明为friend。

对于下面的代码，只有第一个才是Screen类的友元：

1. **extern** std::ostream& storeOn(std::ostream &, Screen &); //它是Screen类的友元
2. **extern** BitMap& storeOn(BitMap &, Screen &);  //它不是Screen类的友元
3. **class** Screen {
4. // ostream version of storeOn may access the private parts of Screen objects
5. **friend** std::ostream& storeOn(std::ostream &, Screen &);
6. // . . .
7. };

# 友元函数可以定义在 类的内部吗？



# 友元的声明 和 普通意义上的声明 有何区别？

友元声明的作用 **仅仅是** 影响 友元函数 对类的访问权限，它本身意义并不是普通意义上的声明（想要在类内使用该友元函数，需要在外面声明它，好让它对成员函数可见）。

也就是说如果类内的成员函数想使用该友元函数，则必须在外面声明该函数，要不然对于类内的函数来说该友元函数是不可见的（即使该友元函数定义在类内）。

下面的代码是错的，因为函数 X()、g()在函数f()被声明之前使用了它：

1. **struct** X {
2. **friend** **void** f()  // 友元的声明，对于后面的X()来说，它是不可见的
3. {
4. /\* 定义在内部的友元函数 \*/
5. }
6. X() { f(); } // 错误: 函数f()还未被声明
7. **void** g();
8. **void** h();
9. };
10. **void** X::g() { **return** f(); }  // 错误: 函数f()还未被声明
11. **void** f();  //此处为f()的声明
12. **void** X::h() { **return** f(); }  // 正确: 函数f()在上一句声明了

想要修改上面的代码很容易，把 **void** f() 放到**struct** X前面，其它不变即可。

# 定义在类外部的函数 的返回值前面 需要加类名吗？

**因为类名只能保证它后面的部分（参数列表、函数体）的作用域在类的作用域，例如对于下面的代码：**

1. **void** Window\_mgr::clear(ScreenIndex i)
2. {
3. Screen &s = screens[i];
4. s.contents = string(s.height \* s.width, ' ');
5. }

Window\_mgr::只能保证函数名clear、以及它的参数列表、它的函数体在类Window\_mgr的作用域内，而前面的返回类型 **void** ，并不受Window\_mgr::的作用，因为返回类型 **void**在他前面。也就是说，**如果返回值类型是类自己定义的**，那就需要，如果不是那就不需要，只要保证编译器能找到就行。

对于下面的例子：

因为类型ScreenIndex 是类Window\_mgr内部定义的，因此需要加上类名，告诉编译器，这里的类型ScreenIndex 是取自类Window\_mgr：

1. **class** Window\_mgr {
2. **public**:
3. // 自己定义了类型 ScreenIndex
4. **using** ScreenIndex = std::vector<Screen>::size\_type;
5. ScreenIndex addScreen(**const** Screen&);
6. // other members as before
7. };
8. // 下面有两个  类名(Window\_mgr::)
9. // 第一个的作用：告诉编译器 类型ScreenIndex  是取自类Window\_mgr
10. // 第二个的作用：告诉编译器 addScreen()函数是类Window\_mgr的成员
11. Window\_mgr::ScreenIndex
12. Window\_mgr::addScreen(**const** Screen &s)
13. {
14. screens.push\_back(s);
15. **return** screens.size() - 1;
16. }

# 编译器怎么处理 类的成员函数的 声明 和 定义？

第一步：编译所有成员函数的声明；

第二步：直到整个类的声明都处理完后，处理函数体（即定义）

## 为什么编译器要这么做呢？

这样可以简化类代码的组织方式，因为编译器先处理所有的声明，这就意味着，先声明的成员 也可以使用 后声明的成员。

## 有例外情况吗？

有的，类型别名 在使用前必须被声明，如果某个成员使用了类中还没出现的名字，则编译器会在外层作用域查找

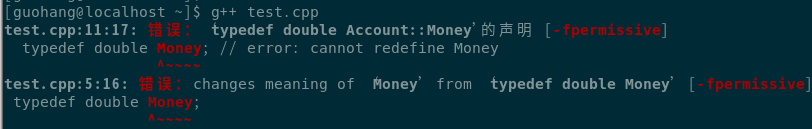
1. **typedef** **double** Money;
2. string bal;
3. **class** Account {
4. **public**:
5. Money balance() { **return** bal; }  // 用的是类外的 类型别名Money
6. **private**:
7. Money bal;  // 用的是类外的 类型别名Money
8. // ...
9. };

# 在类内 定义类型别名 需要注意什么？

假设在类内定义了类型别名A，且在类的外层也有某个类型别名B和类型别名A 同名，若在类内使用了类型别名B，则类型别名A的声明将报错，错误原因是重复定义了相同的类型别名。

下面的代码编译时将报错：

1. #include <iostream>
3. **using** **namespace** std;
5. **typedef** **double** Money;
7. **class** Account {
8. **public**:
9. Money balance() { **return** bal; } // 用的是类外的 类型别名Money
10. **private**:
11. // 前面的成员函数blance()使用了外层的类型别名Money，则报错
12. **typedef** **double** Money; // 错误: 重复定义了  类型别名Money
13. Money bal;
14. };
16. **int** main()
17. {
18. Account A;
19. }



编译报错的原因：

成员函数balance()使用了外层的类型别名Money，所以后面的的声明 **typedef** **double** Money重复定义了类型别名Money

将代码修改成下面之后，成功编译，因为将类型别名的声明移到类的最前面来了，它将隐藏 外面那个 同名的别名：

1. #include <iostream>
3. **using** **namespace** std;
5. **typedef** **double** Money;
7. **class** Account {
8. **public**:
9. // 将类型别名的声明移到类的最前面来了，它将隐藏 外面那个 同名的别名
10. **typedef** **double** Money;
11. Money balance() { **return** bal; }
12. **private**:
13. Money bal;
14. // ...
15. };
17. **int** main()
18. {
19. Account A;
20. }

# 在类内 定义类型别名 建议怎么做？

把类型别名的 定义写在类的开始处，这样可以保证所有使用该类型的成员都可以使用它。

# 如果在类外有一个变量名为 height，类内也有一个名为height的成员，如何在类内部使用 类外那个height呢？

用作用域运算符来请求外部的那个height，代码如下：

1. **int** height; // 全局变量
2. **class** Screen {
3. **public**:
4. **typedef** std::string::size\_type pos;
5. **void** dummy\_fcn(pos height) {
6. cursor = width \* height; // which height? the parameter
7. }
8. **private**:
9. pos cursor = 0;
10. pos height = 0, width = 0;
11. };
12. **void** Screen::dummy\_fcn(pos height) {
13. cursor = width \* **this**->height;  // 成员变量height
14. cursor = width \* Screen::height;  // 成员变量height
15. cursor = width \* ::height; // 全局变量height
16. }

# 初始化 类数据成员 有哪些方法？

随着构造函数开始执行，初始化就完成了，也就是说初始化类的数据成员的**唯一机会**就是通过 构造函数初始化列表。

# 使用 构造函数初始值 效率会更高吗？

是的，构造函数初始化列表 是直接初始化成员，而在构造函数内部赋值是先初始化，然后赋值，相比之下多了一个赋值的操作，效率确实不如构造函数初始化列表。

# 下面的代码有什么问题？

* 1. **class** X {
  2. **int** i;
  3. **int** j;
  4. **public**:
  5. // undefined: i is initialized before j
  6. X(**int** val): j(val), i(j) { }
  7. };

因为构造函数初始化成员的顺序并不是按初始值列表出现的顺序来进行的，而是根据数据成员在类内声明的顺序来初始化的，所以在上述的代码中，成员i比j先初始化，因此在构造函数 X(**int** val): j(val), i(j) { }中，用j来初始化i，因此这段代码存在的问题是：用未初始化的变量j来初始化i。

# 什么是 委托构造函数（delegating constructor）？

顾名思义，委托构造函数 是 把自己的 一些（甚至全部）工作（初始化数据成员） “委托”给 其它构造函数，

# 如何编写 委托构造函数？

和其它构造函数一样， 委托构造函数 也有 成员初始值列表 和 函数体，例如：

Sales\_data(std::istream &is): Sales\_data()

                 { read(is, \***this**); }

成员初始值列表 ：Sales\_data()

函数体 ：{ read(is, \***this**); }

1. **class** Sales\_data {
2. **public**:
3. // 第一个构造函数：它是 非委托构造函数 使用对应的实参初始化成员：
4. Sales\_data(std::string s, unsigned cnt, **double** price):
5. bookNo(s), units\_sold(cnt), revenue(cnt\*price) {}
6. // 下面都是 委托构造函数
7. // 第二个构造函数：它是 默认构造函数，委托给了第一个构造函数
8. Sales\_data(): Sales\_data("", 0, 0) {}
9. // 第三个构造函数：委托给了第一个构造函数
10. Sales\_data(std::string s): Sales\_data(s, 0,0) {}
11. // 第四个构造函数：先委托给了 默认构造函数，然后 默认构造函数 又委托给了 第一个构造函数；
12. // 受委托的 托构造函数执行完之后，开始执行函数体  read(is, \*this)
13. Sales\_data(std::istream &is): Sales\_data()
14. { read(is, \***this**); }
15. // other members as before
16. };

# 默认构造函数 什么时候会被调用？

当对象被 默认初始化 或 值初始化 的时候，会自动执行默认构造函数

# 下面的代码存在什么问题？

* 1. **class** NoDefault {
  2. **public**:
  3. NoDefault(**const** std::string&);
  4. **int** len = 0;
  5. };
  6. **struct** A {
  7. NoDefault my\_mem;
  8. **double** balance = 0.0;
  9. };
  10. A a;
  11. **struct** B {
  12. B() {}
  13. NoDefault b\_member;
  14. };

① 类NoDefault 是没有默认构造函数的，因为类NoDefault 已经有一个构造函数了，而它没有要求编译器给它生成（使用 =default），因此编译器不会帮它合成默认构造函数；

② 如果类内包含一个其他类类型的成员，而且该成员的类型没有默认构造函数，则无法初始化。此时我们必须自定义默认构造函数，否则将没有可用的默认构造函数。

而类A有一个成员是类NoDefault ，而它没有 默认构造函数，因此编译器不能为其合成默认构造函数，具体看下面的注释：

1. **class** NoDefault {
2. **public**:
3. // 已有构造函数 且未显示定义默认构造函数、也未用 =default要求编译器合成
4. // 所以类NoDefault 没有 默认构造函数
5. NoDefault(**const** std::string&);
6. **int** len = 0;
7. };
8. **struct** A {
9. NoDefault my\_mem;  // 注意！类NoDefault 没有 默认构造函数！
10. **double** balance = 0.0;
11. };
12. A a; // 错误: 类A的成员my\_mem的类型是类NoDefault，而它没有 默认构造函数，因此编译器不能为其 合成默认构造函数
13. **struct** B {
14. B() {} // 错误:  b\_member没有初始值
15. NoDefault b\_member;
16. };

# 什么是 转换构造函数（converting constructor）？

如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制，有时我们把这种构造函数称为转换构造函数（converting constructor）

# 如何定义 隐式的 类类型转换？(什么样的函数可以进行隐式的类类型转换？)

如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制，有时我们把这种构造函数称为转换构造函数（converting constructor）

能通过一个实参调用 的构造函数 定义了一条 从构造函数的参数类型 向类类型 隐式转换的规则。

# 下面有哪些是 转换构造函数？

* 1. **class** Sales\_data {
  2. **public**:
  3. // constructors
  4. Sales\_data() = **default**;
  5. Sales\_data(**const** std::string &s): bookNo(s) { }
  6. Sales\_data(**const** std::string &s, unsigned n, **double** p):
  7. bookNo(s), units\_sold(n), revenue(p\*n) { }
  8. Sales\_data(std::istream &);
  10. // operations on Sales\_data objects
  11. std::string isbn() **const** { **return** bookNo; }
  12. Sales\_data& combine(**const** Sales\_data&);
  13. **double** avg\_price() **const**;
  14. **private**:
  15. std::string bookNo;
  16. unsigned units\_sold = 0;
  17. **double** revenue = 0.0;
  18. };

下面这两个是，因为它们都是 只接受一个参数的构造函数：

① Sales\_data(**const** std::string &s): bookNo(s) { }

② Sales\_data(std::istream &);

## 对于上面的类定义，编译器在执行下面的语句的时候 发生了什么？

1. string null\_book = "9-999-99999-9";
2. item.combine(null\_book);  // 为什么传string对象进去，编译器没有报错？

我们来看一下combine()函数的函数原型：

Sales\_data& combine(**const** Sales\_data&);

它接受一个Sales\_data参数，并返回一个Sales\_data的引用，但下面的语句在调用combine()函数的时候用的是一个string 类型：

item.combine(null\_book);

然而编译器却没有报错，这是因为编译器将 string 类型转换成了一个Sales\_data类型的对象：

① 调用构造函数Sales\_data(**const** std::string &s): bookNo(s) { }创建一个临时变量tmp；

② 临时变量tmp的units\_sold 成员、revenue成员 等于 0 ， bookNo成员则等于变量null\_book；

③ 新生成的这个tmp变量被传给了combine()函数；

## 那下面的代码会发生什么呢？

1. item.combine("9-999-99999-9");

编译器只会执行一步类型转换，而上面的代码执行了两步，分别是：

(1)  将"9-999-99999-9" 转换为string 类型；

(2)  将string类型转换为 Sales\_data类型；

因此这条语句将报错。

## 下面的代码呢？

1. item.combine(string("9-999-99999-9"));
2. item.combine(Sales\_data("9-999-99999-9"));
3. item.combine(cin);

1 ) item.combine(string("9-999-99999-9"))：

① "9-999-99999-9"**显式** 地转换为string；

② string隐式的转换为Sales\_data；

也就是说只执行了一步 **隐式** 类型转换，因此是正确的；

**2 )** item.combine(Sales\_data("9-999-99999-9"))：

① "9-999-99999-9" **隐式** 地转换为string；（注意，上一句是显示的转换为string）

② string **显式** 的转换为Sales\_data；

注意上面发的是隐式还是显式类型转换。

**3 )** item.combine(cin)：

隐式的将cin转换成Sales\_data，这个转换

# 如何抑制 构造函数定义的 的隐式转换？

通过将 构造函数 声明为explicit加以阻止。

1. **class** Sales\_data {
2. **public**:
3. Sales\_data() = **default**;
4. Sales\_data(**const** std::string &s, unsigned n, **double** p):
5. bookNo(s), units\_sold(n), revenue(p\*n) { }
6. **explicit** Sales\_data(**const** std::string &s): bookNo(s) { }
7. **explicit** Sales\_data(std::istream&);
8. // remaining members as before
9. };

# 编写和使用 explicit构造函数 时要注意什么？

**1 )** 关键字explicit只对于 一个实参的构造函数 有效，因为多个实参的构造函数不能用于隐式转换，所以没必要将这些构造函数指定为explicit；

**2 )** 只能在类内声明构造函数的时候加explicit关键字，在类外定义的时候不应该加explicit：

1. **class** Sales\_data {
2. **public**:
3. Sales\_data() = **default**;
4. **explicit** Sales\_data(**const** std::string &s): bookNo(s) { }
5. **explicit** Sales\_data(std::istream&);
6. // remaining members as before
7. };
8. // 错误，类外定义的生活不应该加 explicit关键字
9. **explicit** Sales\_data::Sales\_data(istream& is)  {
10. read(is, \***this**);
11. }

**3 )** explicit构造函数 只能用于显示初始化，不能用于拷贝形式的初始化

1. string null\_book = "9-999-99999-9";
2. // 错误:
3. Sales\_data item2 = null\_book;

错误的原因：

因为null\_book是一个string类型的变量，而将null\_book赋给Sales\_data类型的item2 ，这意味着它将调用构造函数

**explicit** Sales\_data(**const** std::string &s): bookNo(s) { }

来进行类型转换，而这个构造函数是**explicit** 的，所以会报错

# 什么是 explicit构造函数？

在只有一个实参的构造函数前面加explicit关键字，用来阻止通过该构造函数来进行隐式类型转换。

# 下面的代码 正确吗？

* 1. **class** Sales\_data {
  2. **public**:
  3. Sales\_data() = **default**;
  4. Sales\_data(**const** std::string &s, unsigned n, **double** p):
  5. bookNo(s), units\_sold(n), revenue(p\*n) { }
  6. **explicit** Sales\_data(**const** std::string &s): bookNo(s) { }
  7. **explicit** Sales\_data(std::istream&);
  8. // remaining members as before
  9. };
  10. item.combine(**static\_cast**<Sales\_data>(cin));

正确，虽然构造函数  **explicit** Sales\_data(std::istream&); 是**explicit** 的，这意味不能通过它进行隐式类型转换，而通过它进行显示类型转换是可以的。

# 什么是显示构造函数？

explicit构造函数 也称为 显示构造函数。

# 标准库中有哪些 显示构造函数

The string constructor that takes a single parameter of type const char\*  
is not explicit.

The vector constructor that takes a size is explicit.

# 什么是 聚合类（Aggregate Classes）

1 ) 所有成员都是public的（直接用struct即可）

2 ) 没有定义任何构造函数；

3 ) 没有类内初始值；

4 ) 没有基类，也没有virtual函数

**这感觉有点像C语言的struct**

**下面这个就是聚合类：**

1. **struct** Data {
2. **int** ival;
3. string s;
4. };
5. // val1.ival = 0; val1.s = string("Anna")
6. Data val1 = { 0, "Anna" };

**聚合类的初始化：**

1 ) 初始值的顺序 要和 声明的顺序一致：

1. // error: can't use "Anna" to initialize ival, or 1024 to initialize s
2. Data val2 = { "Anna", 1024 };

2 ) 若初始值列表中的元素个数少于类的成员数量，则剩下的会被值初始化：

1. //  s 被值初始化了
2. Data val2 = { 1024 };

# 什么是字面值常量类？

TODO：

# 什么是 类的静态成员？

不同于其它成员，**类的静态成员**与 类本身 直接相关，而不是与类的各个对象相关。

# 为什么需要 类的静态成员？

有的时候我们会需要一些成员与 类本身 直接相关，而不是与类的各个对象相关，比如对于一个银行账户类来说，可能需要一个成员来表示基准利率：

① 因为每个人的基准利率都是一样的，从效率上来说，我们没必要为每一个对象都存储相同的基础利率

② 而且万一基准利率有调整，我们希望能做到一改全改；

此时使用静态成员就能做到。

# 如何声明静态成员？

在 类内 将你想声明为静态的成员、函数前面加上static关键字即可：

1. **class** Account {
2. **public**:
3. **void** calculate() { amount += amount \* interestRate; }
4. **static** **double** rate() { **return** interestRate; }
5. **static** **void** rate(**double**);
6. **private**:
7. std::string owner;
8. **double** amount;
9. **static** **double** interestRate;
10. **static** **double** initRate();
11. };

## 上面声明的类 的对象 有几个成员？

它的对象包含两个数据成员amount、owner，因为interestRate是静态的，他不和对象关联，而是与类本身关联。

## 如何定义 上面的类的 静态成员函数？

和其它成员函数一样，成员函数能在类内定义也能字类外定义，但在外面定义静态成员函数的时候，不能加static关键字：

类外定义的时候要带上 类名：

1. //错误：在外面定义 静态成员函数的时候不能出现 static关键字。
2. **static** **void** Account::rate(**double** newRate){ interestRate = newRate; }
3. // 正确
4. **void** Account::rate(**double** newRate){ interestRate = newRate; }

## 如何定义 上面的类的 静态成员变量？

类的静态成员变量有两种定义方式：

**① 在类外定义**：

因为静态成员属于整个类，而不属于某个对象，如果在类内初始化，会导致每个对象都包含该静态成员，这是矛盾的，因此

1. // 注意，这是定义，不是赋值！
2. // 和静态成员函数一样，类外定义的时候不需要加 static
3. **double** Account::interestRate = initRate();

**② 提供类内初始值：**

还有一种方法是提供一个类内初始值，这个时候**要求静态成员变量的类型是 字面值常量类型constexptr**：

1. **class** Account {
2. **public**:
3. **static** **double** rate() { **return** interestRate; }
4. **static** **void** rate(**double**);
5. **private**:
6. **static** constexpr **int** period = 30;// period 是车辆表达式！
7. **double** daily\_tbl[period];
8. };

即使一个静态成员在类内部被初始化了，通常情况下也应该在类的外部定义一下该成员，而且若提供了类内初始值，则在类外定义的时候不能再指定一个初始值了：

constexpr **int** Account::period;

# 静态成员函数 和 普通的成员函数有何区别？

因为静态成员函数不和任何对象绑定在一起，因此静态成员函数没有this指针。这意味着静态成员函数不能声明为 const成员函数，因此它们的区别总结如下：

① 静态成员函数没有 this指针；

② 静态成员函数不能声明为const成员函数，因为它没有this指针。

# 静态成员变量 的生命周期？

静态成员变量和其它staici变量一样，存在于程序的整个生命周期。

# 类的 普通数据成员 可以做默认实参吗？

普通成员不能作为默认实参，因为普通数据成员的本身属于类对象的一部分，这么做的结果是无法真正提供一个对象以便从中获取成员的值，最终引发错误。

# 哪些场景下，静态成员能使用，而普通成员不能？

静态成员独立于任何对象。因此，在某些非静态数据成员可能非法的场合，静态成员却可以正常地使用:

① 静态数据成员可以是不完全类型。特别的，静态数据成员的类型可以就是它所属的类类型。而非静态数据成员则受到限制，只能声明它所属的指针或引用：

1. **class** Bar {
2. **public**:
3. // ...
4. **private**:
5. **static** Bar mem1;  // 正确: 静态成员可以是不完全类型
6. Bar \*mem2;  // 正确: 普通成员可以是不完全类型的指针
7. Bar mem3;  // 错误: 普通成员不能是可以是不完全类型
8. };

② 可以用 静态数据成员 作为默认实参，而普通成员不能作为默认实参，因为普通数据成员的本身属于类对象的一部分，这么做的结果是无法真正提供一个对象以便从中获取成员的值，最终引发错误：

1. **class** Screen {
2. **public**:
3. // bkground refers to the static member
4. // declared later in the class definition
5. Screen& clear(**char** = bkground);
6. **private**:
7. **static** **const** **char** bkground;
8. };