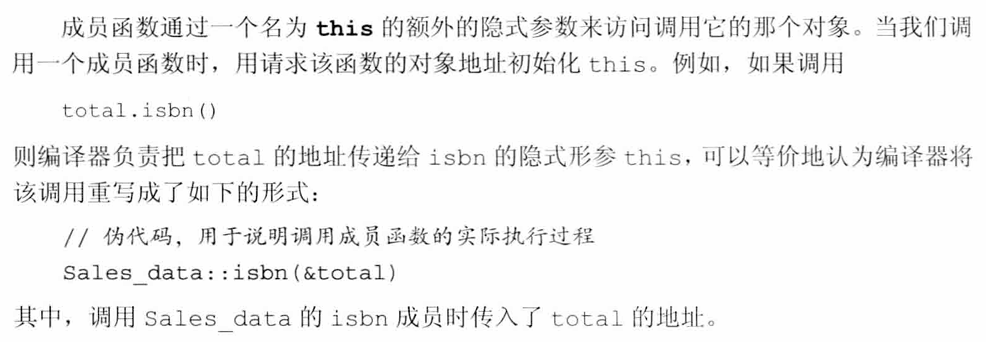
类的基本思想：数据抽象和封装。

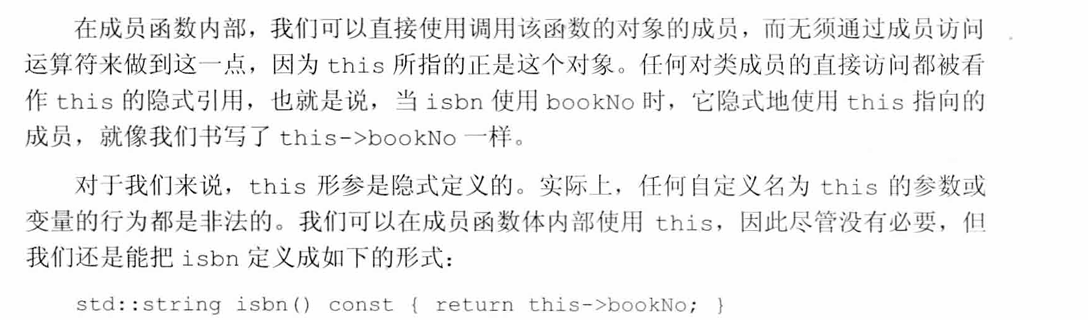
封装：将接口和实现分离。

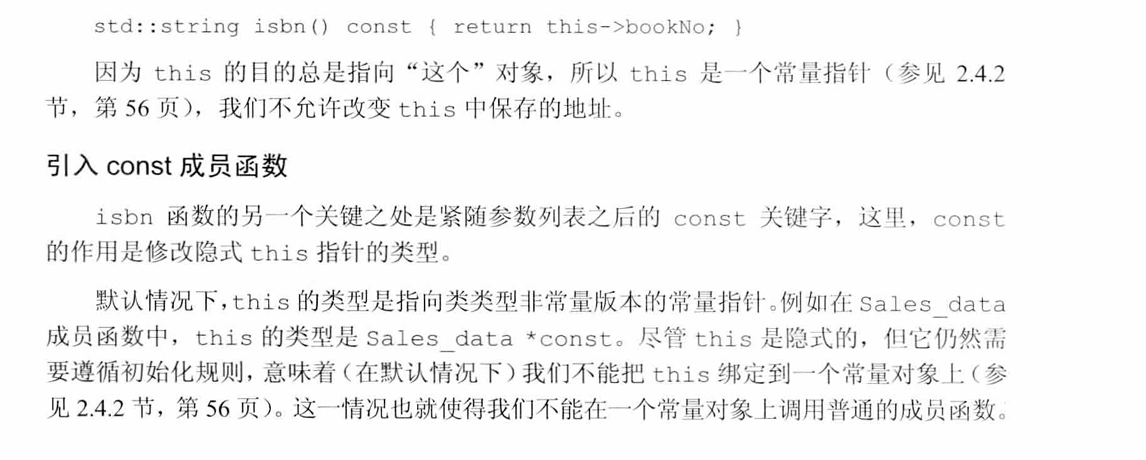
设计类的接口时，要考虑如何使类易于使用。当使用类时，不应该顾及类的实现机理。（体现封装性）

实例使用点操作符调用函数实质上是这样，把实例的地址传入函数中

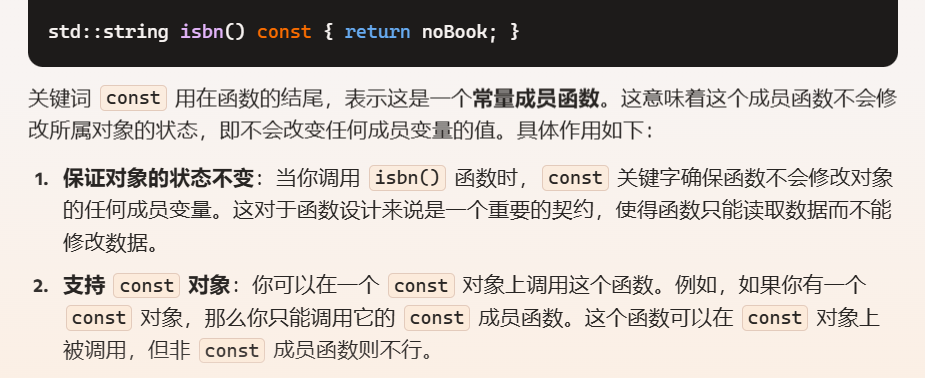


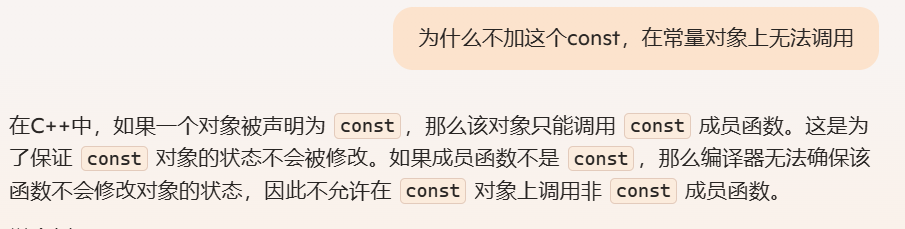
isbn方法内部调用成员函数自身的成员时，实际上有一个常量指针this时刻指向该实例自身，而this平时是隐藏起来的可以不写出来





ps：关于这里const的作用，读的时候第一时间没理解，AI的回答





概括一下，常量成员函数只会读取字段而不能修改字段

在编译类时，无需注意成员声明的次序，原因是编译器按照下面的顺序处理类：

1.编译声明

2.编译函数体

定义类相关的非成员函数

定义：类相关的非成员函数是指在类外部定义的函数，但通常用于对类对象进行操作。

ps：在这里有一个疑问，我尚不了解类外构造函数的应用场景以及有什么好处。

类的构造函数：

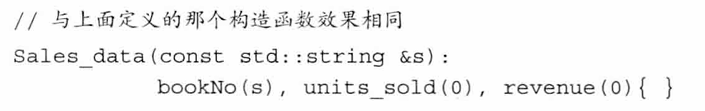
1.构造函数用来初始化类对象的数据成员。当类的对象被创建时，就会执行构造函数。

2.构造函数和类同名，且没有返回类型。

3.构造函数可以重载。

4.如果类没有任何构造函数则编译器自己会创建默认构造函数（适用范围很小，尽量不要如此），默认构造函数不用传入任何实参，当自定义显示构造函数后编译器不会创建默认构造函数（需要合成构造函数把当前手动创建的构造函数定为默认构造函数）

C++中，构造函数还可以用如下方式初始化变量，并且这样做更好，因为可以控制变量赋值的先后顺序



类的成员的访问控制：

跟在public说明符后的成员在整个程序内都可以被访问到,，public成员定义类的接口；private说明符后的成员只能被类的成员函数访问，private部分封装了类的实现细节。

C++中struct和class的异同点：

异：

1.默认情况下，struct 的所有成员（包括数据成员和成员函数）都是 public 的；而class相反，都为private。

2.默认情况下，struct是公有继承，class为私有继承。

3.struct通常用于简单的数据结构，只包含数据成员而没有行为；class通常用于定义复杂的对象行为和状态，包含数据成员和成员函数。

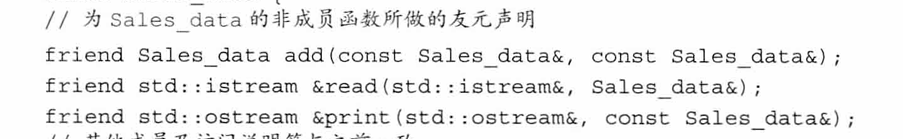
同：

struct和class在内存布局和对齐方式上没有本质区别（并不像C#中一个为值类型一个为引用类型，分别存储在栈上和堆上）

友元：

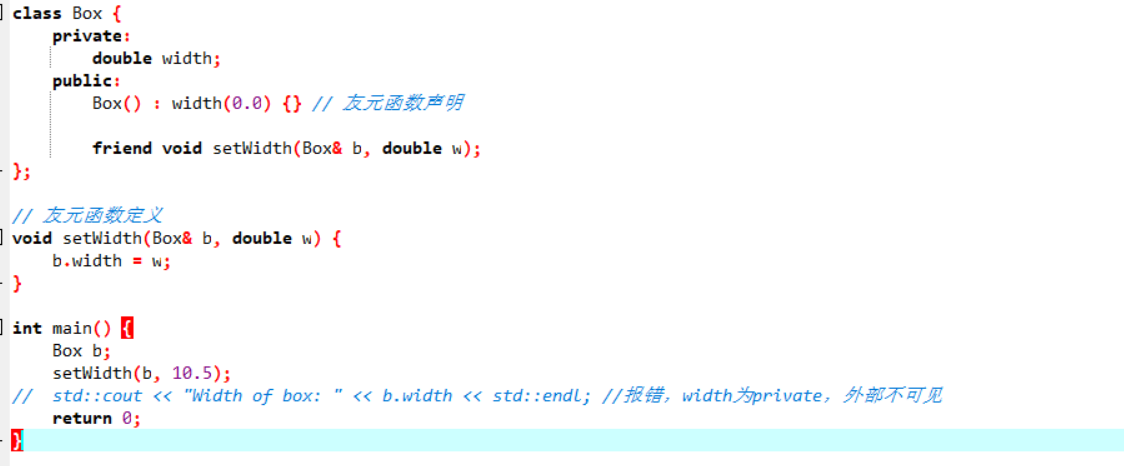
类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数成为它的友元。

如何声明友元，如果类要把一个函数作为它的友元，需要添加一条以friend关键字开始的函数声明语句



友元声明只能出现在类的内部，但可以声明在类内的任意位置，但是这里仅仅指定了访问的权限，还需要在外部实现友元函数

这是友元函数的一个例子，在下例中通过声明friend，然后在类外部定义友元函数，通过友元函数访问到了Box实例的private变量并赋值



可以在类中使用using或typedef为成员的类型定义别名，在类中为成员的类型定义别名时必须先定义后使用。

对于内联函数，可以不必要在声明和定义处都添加inline关键字，但最好只在类外部的定义处添加inline关键字。

ps：C++中为什么喜欢把函数定义在类外，有什么好处：

1.提高代码可读性和可维护性：将成员函数的声明和实现分开，可以使类的声明更加简洁易读，分离接口和实现，能提高代码的可维护性。

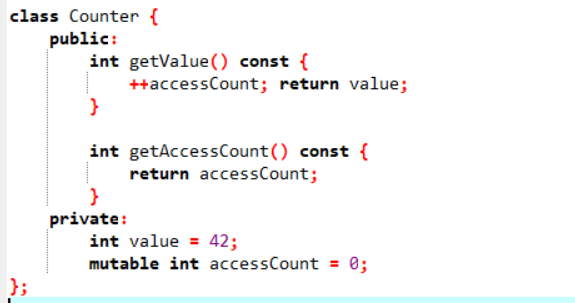
2.减少编译依赖：如果成员函数的实现放在头文件中，任何对头文件的修改都会导致包括该头文件的所有文件重新编译。而将实现放在源文件中，只有修改源文件时才需要重新编译，提高编译速度。

3.封装：在头文件中只暴露接口，在源文件中隐藏实现细节，可以保护类的实现不被外部代码直接依赖，从而提高封装性。这样，改变实现细节而不改变接口时，不会影响使用该类的代码。

4.在类声明中定义的成员函数通常是内联的，而在类外部定义函数，可以避免内联，只有显式使用inline关键字时才会内联，从而控制可执行文件的大小、减小内存占用。

可变数据成员：mutable 关键字，可以在const成员函数中修改特定的数据成员，从而实现灵活的功能设计，而不破坏接口的const性质。

下面是例子：可以在const函数中xiu

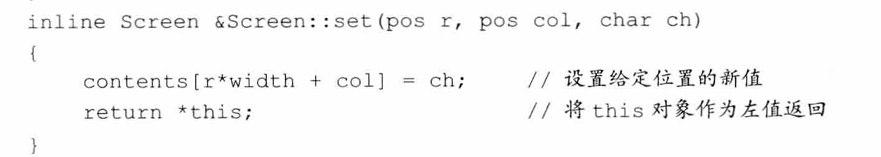


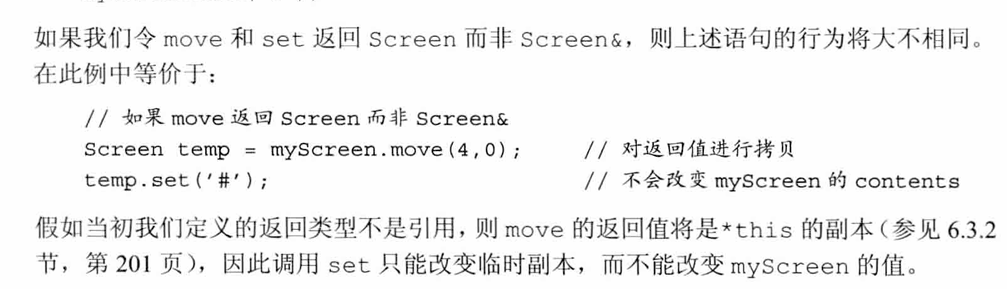
返回this指针的成员函数：

this 指针指向类本身，即 this 是类的地址，\*this 就是类本身。

可以定义返回类型为类对象的引用的函数。如果定义的返回类型不是引用，返回的则是\*this 的副本。

const 函数如果以引用的形式返回 this，返回一个常量引用。





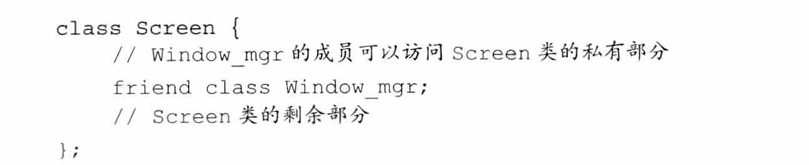
类类型

一个类的成员类型不能是它自己，但是类允许包含指向它自身类型的引用或指针。

友元类：

如何定义一个友元类：

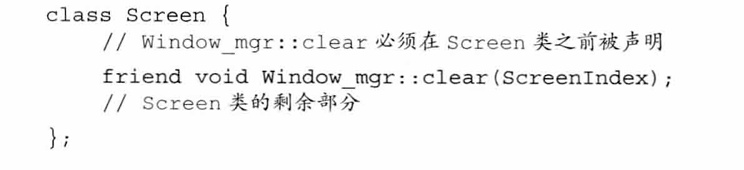
书上的例子中Screen类将Window\_mgr指定为了它的友元类，也就是Window中可以访问Screen全部的成员

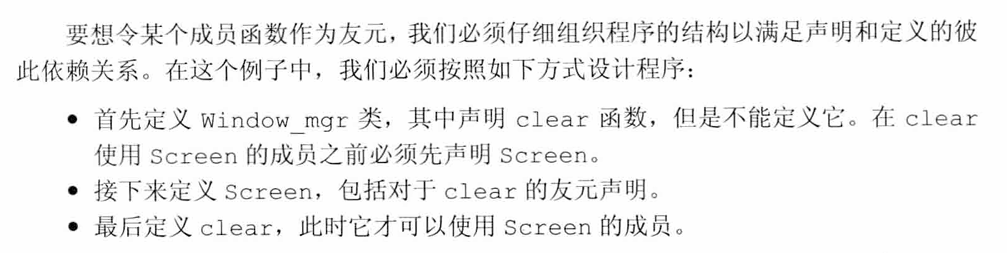


ps：友元在类之间不具有传递性，也就是说如果A中声明了友元类B，B中声明了友元类C，但C不可以访问A的成员。

友元函数：

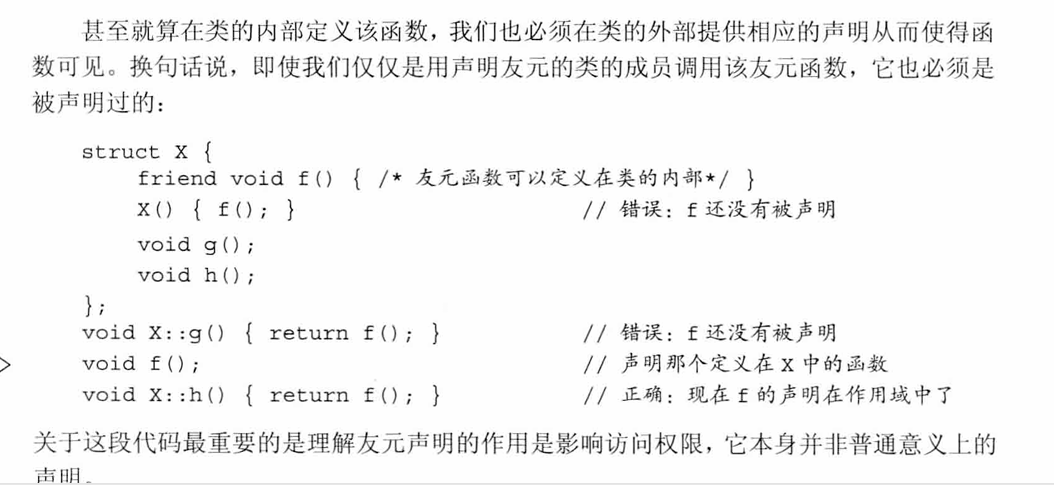
除了将整个类声明为友元以外，还可以单独声明友元为某个类中的某一个函数。这样就必须在函数名前加上类名,





函数重载和友元：如果有一组重载函数，如果要让这一组重载函数都为友元函数，需要逐一声明友元。如果只声明其中一个为友元，剩下的其他版本的函数都不是该类的友元函数，因为重载函数只是名字相同，实际上是不同的函数。

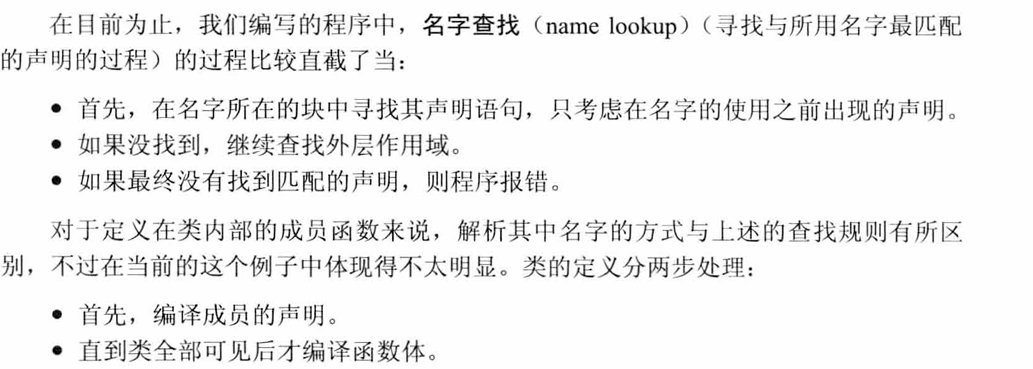
友元函数声明和作用域：



友元声明只是假定该函数在作用域内可见，而不算是真正的声明，真正的声明必须在定义友元函数的类的外部进行。

名字查找：名字查找是编译器在编译过程中解析标识符以确定其实际含义的过程。

对函数声明的名字查找步骤：



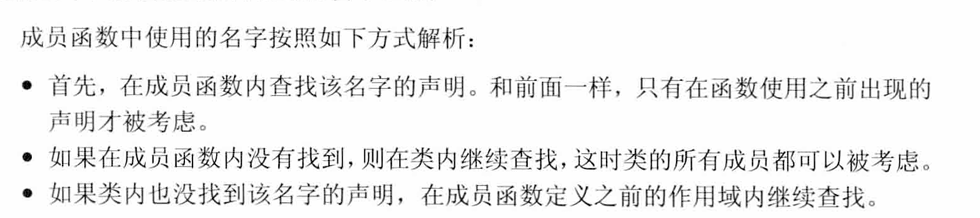
上面对类的两阶段处理只适用于成员函数中使用的名字，声明中使用的类型名都必须保证在使用前是可见的。如果某个成员的声明中使用了尚未出现的名字，编译器会在定义该类的作用域中继续寻找。

类型名对应的类型如果在类外已经定义过，不能在类内重定义。

ps：不建议使用其他成员的名字作为某个成员函数的参数。

ps：编译器在处理完类中的全部声明后才会处理成员函数的定义。如果同时被处理，会出现一个很不方便的事情，在函数中要被使用的成员，必须在前面出现过。这样会存在一个拓扑顺序。

成员函数定义的名字查找：



如果在类的作用域内没找到，会在类外围的作用域中继续查找，还没有查到在全局作用域查找，如果没找到就报错。

总结，（类型）名字查找的顺序：

1.局部作用域：首先在当前块或函数中查找。

2.类作用域：如果在局部作用域中未找到，并且当前函数属于一个类成员函数，则在类作用域中查找。

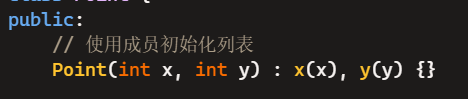
3.命名空间作用域：如果在类作用域中未找到，则在所属命名空间中查找。

4.全局作用域：最后在全局作用域中查找。

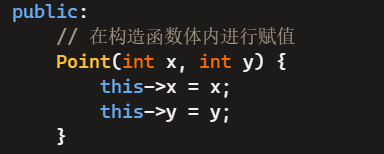
5.未找到报错：如果仍未找到，将报错。

关于构造函数：

1.成员初始化列表：



2.构造函数体内赋值：



建议使用第一种方式——成员初始化列表。

成员初始化列表的优点：

1.成员初始化列表可以避免额外的构造和赋值开销。成员初始化列表直接在成员变量的内存位置进行初始化，而在构造函数体内赋值则先调用成员变量的默认 构造函数，然后进行赋值操作。

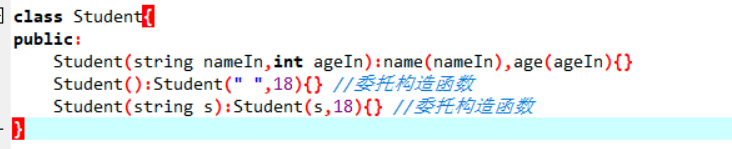
2.能够初始化常量成员和引用成员。常量成员和引用成员必须在初始化列表中初始化，因为第二种方式是赋值，而常量和引用必须初始化，因此只能使用成员初始化列表的方式。

3.保持成员变量的正确性，防止成员变量间的依赖关系产生的未定义行为。成员初始化列表可以确保这些变量按照定义顺序初始化，而如果在构造函数体内进行赋值，可能导致未定义行为。

ps：构造函数也可以使用默认初始值，如果一个构造函数为所有参数提供了默认实参，那么相当于定义了默认构造函数。

委托构造函数：

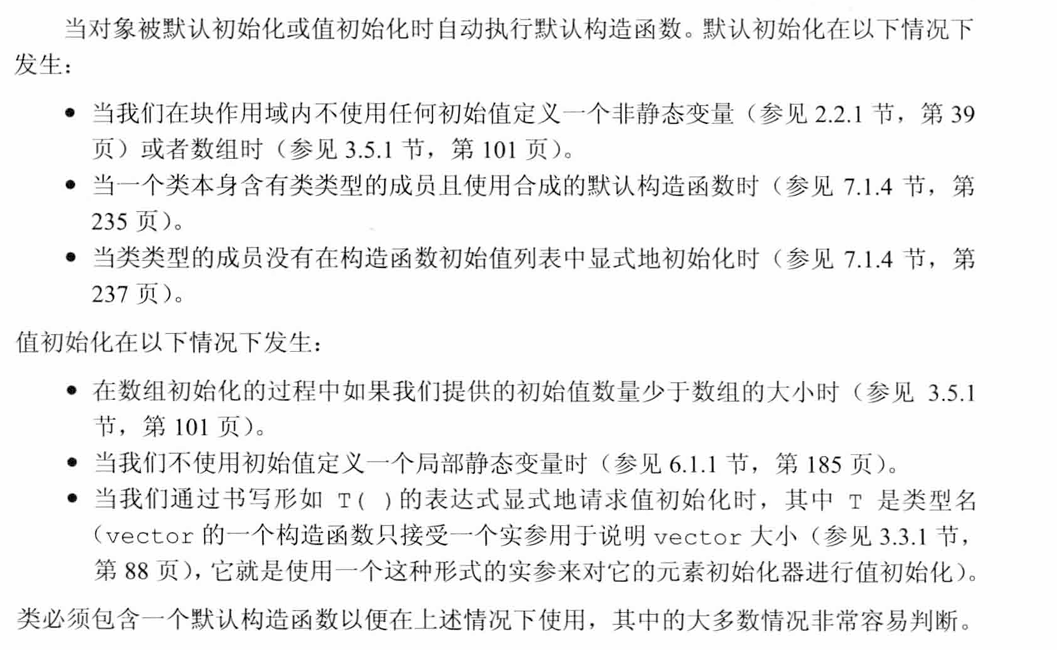
委托构造函数通过其他构造函数来执行自己的初始化过程。



上例的两个委托构造函数都通过第一个构造函数进行初始化。

默认初始化和值初始化发生时机：

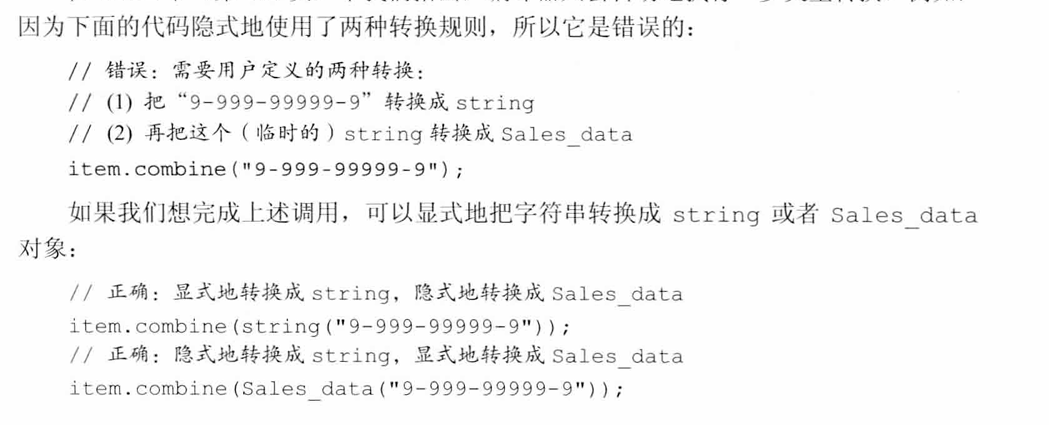
在实际中即使定义了构造函数，最好也保留默认构造函数



隐式的类类型转换：隐式的类类型转换是指编译器在需要时自动将一个对象从一种类型转换为另一种类型的过程。这种转换通常发生在函数调用、赋值操作或返回值处理中。隐式类类型转换通过定义构造函数或类型转换运算符来实现。

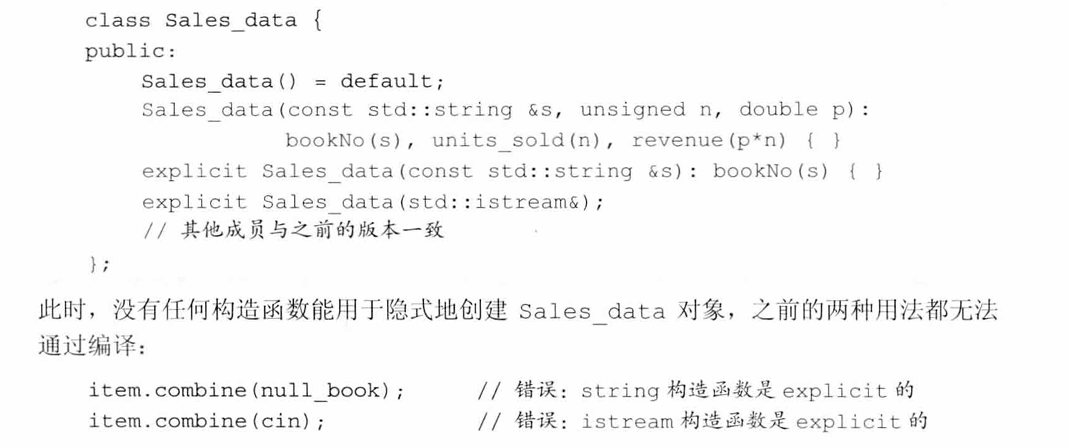
ps：如果类有一个单参数构造函数，它可以被用作隐式转换构造函数。编译器会使用这个构造函数将源类型的对象转换为目标类型的对象。

编译器只会自动地执行一步类型转换。



这个被隐式转换成Sales\_data类的变量是一个临时变量，在执行完combine函数后就会被丢弃。

explicit-抑制构造函数定义的隐式转换：

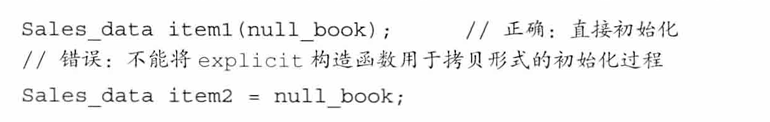


ps：

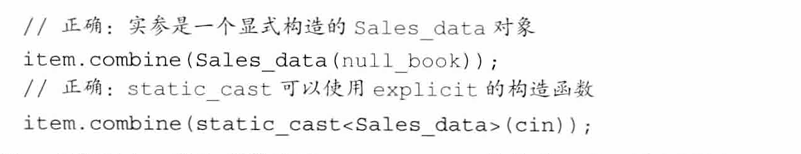
1.explicit关键字和friend一样，在有多个重载的函数时只能作用于其中一个。

2.explicit关键字只能作用于构造函数。

explicit函数只能用于直接初始化：



explicit关键字虽然能抑制隐式转换，但不能阻止显式转换：



聚合类：

聚合类满足以下条件：

1.所有成员都是public

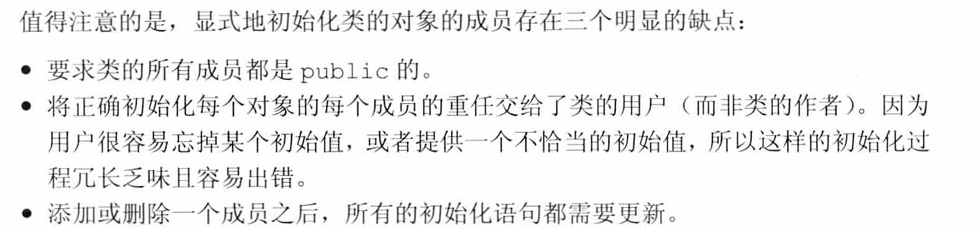
2.没有定义任何构造函数

3.没有类内初始值

4.没有基类，也没有虚函数

ps：聚合类可以像结构体一样使用花括号进行列表初始化。如果花括号内元素数量少于类成员数量，靠后的成员将被值初始化。

聚合类的缺陷：其中后两点缺陷确实很大



字面值常量类：

定义：数据成员都是字面值类型的聚合类

如果不是聚合类，满足以下条件也算字面值常量类：

1.数据成员都必须是字面值类型

2.至少含一个constexpr构造函数

3.如果一个数据成员有类内初始值，则初始值必须是常量表达式（如果成员是类，则初始值必须使用成员自己的 constexpr 构造函数）

4.类必须使用析构函数的默认定义

constexpr构造函数：

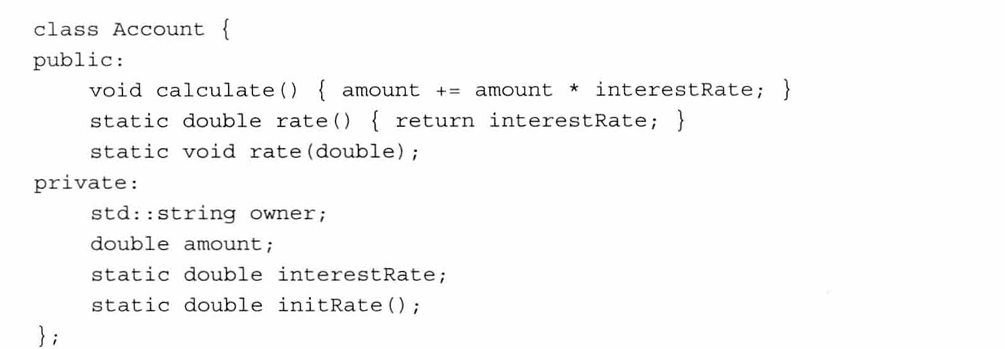
一个字面值常量类至少要提供一个constexpr构造函数。constexpr构造函数的函数体一般情况下是空的。

类的静态成员：

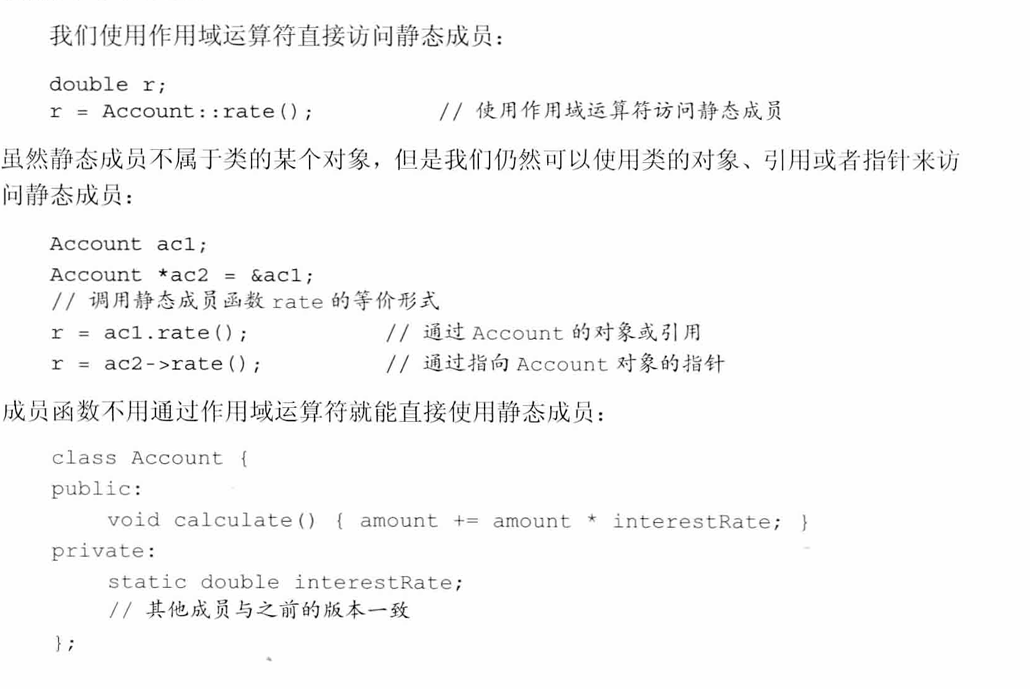
类的静态成员与类本身直接关联，而不是与类的对象保持关联。（因为不和对象绑定，因此没有this指针，且存在于对象之外）

静态数据成员可以是常量、引用、指针、类等。

书上声明静态成员的例子：

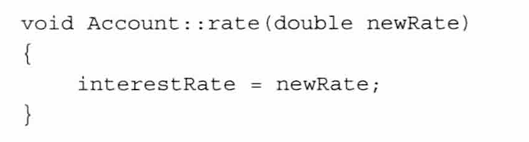


类的静态成员使用方式：



静态成员的定义方式：

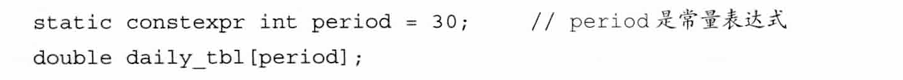
在类外部定义静态成员时，不能加static关键字，这个关键字只能在类内声明时添加。



ps：静态成员只能定义一次，类似于全局变量，定义后在程序的生命周期内一直存在。

静态成员内类初始化：

通常情况下类的静态成员不应该在类的内部初始化，但是可以为静态变量提供const整数类型和类内初始值，不过要求静态成员必须是字面值常量的constexpr，初始值必须是常量表达式。



静态成员可以用但普通成员不能用的场景：

1.静态数据成员可以是不完整类型

2.静态成员可以作为默认实参

ps：什么是不完整类型

不完整类型是指尚未完全定义的类型，它的内存布局或具体内容在使用时尚未明确。最常见的不完整类型包括未定义的类和前向声明的结构体。

