```
}
inline Screen &Screen::set(pos r, pos col, char ch)
{
    contents[r*width + col] = ch;      // 设置给定位置的新值
    return *this;                      // 将 this 对象作为左值返回
}
```

和 move 操作一样，我们的 set 成员的返回值是调用 set 的对象的引用（参见 7.1.2 节，第 232 页）。返回引用的函数是左值的（参见 6.3.2 节，第 202 页），意味着这些函数返回的是对象本身而非对象的副本。如果我们把一系列这样的操作连接在一条表达式中的话：

```
// 把光标移动到一个指定的位置，然后设置该位置的字符值
myScreen.move(4,0).set('#');
```

这些操作将在同一个对象上执行。在上面的表达式中，我们首先移动 myScreen 内的光标，然后设置 myScreen 的 contents 成员。也就是说，上述语句等价于

```
myScreen.move(4,0);
myScreen.set('#');
```

如果我们令 move 和 set 返回 Screen 而非 Screen&，则上述语句的行为将大不相同。在此例中等价于：

```
// 如果 move 返回 Screen 而非 Screen&
Screen temp = myScreen.move(4,0);      // 对返回值进行拷贝
temp.set('#');                         // 不会改变 myScreen 的 contents
```

276

假如当初我们定义的返回类型不是引用，则 move 的返回值将是*this 的副本（参见 6.3.2 节，第 201 页），因此调用 set 只能改变临时副本，而不能改变 myScreen 的值。

从 const 成员函数返回*this

接下来，我们继续添加一个名为 diplay 的操作，它负责打印 Screen 的内容。我们希望这个函数能和 move 以及 set 出现在同一序列中，因此类似于 move 和 set，diplay 函数也应该返回执行它的对象的引用。

从逻辑上来说，显示一个 Screen 并不需要改变它的内容，因此我们令 diplay 为一个 const 成员，此时，this 将是一个指向 const 的指针而*this 是 const 对象。由此推断，display 的返回类型应该是 const Sales_data&。然而，如果真的令 diplay 返回一个 const 的引用，则我们将不能把 display 嵌入到一组动作的序列中去：

```
Screen myScreen;
// 如果 display 返回常量引用，则调用 set 将引发错误
myScreen.display(cout).set('*');
```

即使 myScreen 是个非常量对象，对 set 的调用也无法通过编译。问题在于 display 的 const 版本返回的是常量引用，而我们显然无权 set 一个常量对象。

一个 const 成员函数如果以引用的形式返回*this，那么它的返回类型将是常量引用。

基于 const 的重载

通过区分成员函数是否是 const 的，我们可以对其进行重载，其原因与我们之前根据指针参数是否指向 const（参见 6.4 节，第 208 页）而重载函数的原因差不多。具体说

来，因为非常量版本的函数对于常量对象是不可用的，所以我们只能在一个常量对象上调用 const 成员函数。另一方面，虽然可以在非常量对象上调用常量版本或非常量版本，但显然此时非常量版本是一个更好的匹配。

在下面的这个例子中，我们将定义一个名为 do_display 的私有成员，由它负责打印 Screen 的实际工作。所有的 display 操作都将调用这个函数，然后返回执行操作的对象：

```
class Screen {
public:
    // 根据对象是否是 const 重载了 display 函数
    Screen &display(std::ostream &os)
                { do_display(os); return *this; }
277 const Screen &display(std::ostream &os) const
                { do_display(os); return *this; }
private:
    // 该函数负责显示 Screen 的内容
    void do_display(std::ostream &os) const {os << contents;}
    // 其他成员与之前的版本一致
};
```

和我们之前所学的一样，当一个成员调用另外一个成员时，this 指针在其中隐式地传递。因此，当 display 调用 do_display 时，它的 this 指针隐式地传递给 do_display。而当 display 的非常量版本调用 do_display 时，它的 this 指针将隐式地从指向非常量的指针转换成指向常量的指针（参见 4.11.2 节，第 144 页）。

当 do_display 完成后，display 函数各自返回解引用 this 所得的对象。在非常量版本中，this 指向一个非常量对象，因此 display 返回一个普通的（非常量）引用；而 const 成员则返回一个常量引用。

当我们在某个对象上调用 display 时，该对象是否是 const 决定了应该调用 display 的哪个版本：

```
Screen myScreen(5,3);
const Screen blank(5, 3);
myScreen.set('#').display(cout);      // 调用非常量版本
blank.display(cout);                  // 调用常量版本
```

建议：对于公共代码使用私有功能函数

有些读者可能会奇怪为什么我们要费力定义一个单独的 do_display 函数。毕竟，对 do_display 的调用并不比 do_display 函数内部所做的操作简单多少。为什么还要这么做呢？实际上我们是出于以下原因的：

- 一个基本的愿望是避免在多处使用同样的代码。
- 我们预期随着类的规模发展，display 函数有可能变得更加复杂，此时，把相应的操作写在一处而非两处的作用就比较明显了。
- 我们很可能在开发过程中给 do_display 函数添加某些调试信息，而这些信息将在代码的最终产品版本中去掉。显然，只在 do_display 一处添加或删除这些信息要更容易一些。

- 这个额外的函数调用不会增加任何开销。因为我们在类内部定义了 do_display，所以它隐式地被声明成内联函数。这样的话，调用 do_display 就不会带来任何额外的运行时开销。

在实践中，设计良好的 C++代码常常包含大量类似于 do_display 的小函数，通过调用这些函数，可以完成一组其他函数的“实际”工作。

7.3.2 节练习

练习 7.27：给你自己的 Screen 类添加 move、set 和 display 函数，通过执行下面的代码检验你的类是否正确。

```
Screen myScreen(5, 5, 'X');
myScreen.move(4,0).set('#').display(cout);
cout << "\n";
myScreen.display(cout);
cout << "\n";
```

练习 7.28：如果 move、set 和 display 函数的返回类型不是 Screen&而是 Screen，则在上一个练习中将会发生什么情况？

练习 7.29：修改你的 Screen 类，令 move、set 和 display 函数返回 Screen 并检查程序的运行结果，在上一个练习中你的推测正确吗？

练习 7.30：通过 this 指针使用成员的做法虽然合法，但是有点多余。讨论显式地使用指针访问成员的优缺点。

7.3.3 类类型

每个类定义了唯一的类型。对于两个类来说，即使它们的成员完全一样，这两个类也是两个不同的类型。例如：

278

```
struct First {
    int memi;
    int getMem();
};
struct Second {
    int memi;
    int getMem();
};
First obj1;
Second obj2 = obj1;          // 错误：obj1 和 obj2 的类型不同
```

即使两个类的成员列表完全一致，它们也是不同的类型。对于一个类来说，它的成员和其他任何类（或者任何其他作用域）的成员都不是一回事儿。

我们可以把类名作为类型的名字使用，从而直接指向类类型。或者，我们也可以把类名跟在关键字 class 或 struct 后面：

```
Sales_data item1;            // 默认初始化 Sales_data 类型的对象
class Sales_data item1;      // 一条等价的声明
```

上面这两种使用类类型的方式是等价的，其中第二种方式从C语言继承而来，并且在C++语言中也是合法的。

类的声明

就像可以把函数的声明和定义分离开来一样（参见6.1.2节，第186页），我们也能仅仅声明类而暂时不定义它：

```
class Screen;                    // Screen 类的声明
```

279 这种声明有时被称作**前向声明**（forward declaration），它向程序中引入了名字Screen并且指明Screen是一种类类型。对于类型Screen来说，在它声明之后定义之前是一个**不完全类型**（incomplete type），也就是说，此时我们已知Screen是一个类类型，但是不清楚它到底包含哪些成员。

不完全类型只能在非常有限的情景下使用：可以定义指向这种类型的指针或引用，也可以声明（但是不能定义）以不完全类型作为参数或者返回类型的函数。

对于一个类来说，在我们创建它的对象之前该类必须被定义过，而不能仅仅被声明。否则，编译器就无法了解这样的对象需要多少存储空间。类似的，类也必须首先被定义，然后才能用引用或者指针访问其成员。毕竟，如果类尚未定义，编译器也就不清楚该类到底有哪些成员。

在7.6节（第268页）中我们将描述一种例外的情况：直到类被定义之后数据成员才能被声明成这种类类型。换句话说，我们必须首先完成类的定义，然后编译器才能知道存储该数据成员需要多少空间。因为只有当类全部完成后类才算被定义，所以一个类的成员类型不能是该类自己。然而，一旦一个类的名字出现后，它就被认为是声明过了（但尚未定义），因此类允许包含指向它自身类型的引用或指针：

```
class Link_screen {
    Screen window;
    Link_screen *next;
    Link_screen *prev;
};
```

7.3.3 节练习

练习7.31：定义一对类X和Y，其中X包含一个指向Y的指针，而Y包含一个类型为X的对象。

7.3.4 友元再探

我们的Sales_data类把三个普通的非成员函数定义成了友元（参见7.2.1节，第241页）。类还可以把其他的类定义成友元，也可以把其他类（之前已定义过的）的成员函数定义成友元。此外，友元函数能定义在类的内部，这样的函数是隐式内联的。

类之间的友元关系

举个友元类的例子，我们的Window_mgr类（参见7.3.1节，第245页）的某些成员可能需要访问它管理的Screen类的内部数据。例如，假设我们需要为Window_mgr添
280 加一个名为clear的成员，它负责把一个指定的Screen的内容都设为空白。为了完成这一任务，clear需要访问Screen的私有成员；而要想令这种访问合法，Screen需要

把 Window_mgr 指定成它的友元：

```
class Screen {
    // Window_mgr 的成员可以访问 Screen 类的私有部分
    friend class Window_mgr;
    // Screen 类的剩余部分
};
```

如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员。通过上面的声明，Window_mgr 被指定为 Screen 的友元，因此我们可以将 Window_mgr 的 clear 成员写成如下的形式：

```
class Window_mgr {
public:
    // 窗口中每个屏幕的编号
    using ScreenIndex = std::vector<Screen>::size_type;
    // 按照编号将指定的 Screen 重置为空白
    void clear(ScreenIndex);
private:
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};
void Window_mgr::clear(ScreenIndex i)
{
    // s 是一个 Screen 的引用，指向我们想清空的那个屏幕
    Screen &s = screens[i];
    // 将那个选定的 Screen 重置为空白
    s.contents = string(s.height * s.width, ' ');
}
```

一开始，首先把 s 定义成 screens vector 中第 i 个位置上的 Screen 的引用，随后利用 Screen 的 height 和 width 成员计算出一个新的 string 对象，并令其含有若干个空白字符，最后我们把这个含有很多空白的字符串赋给 contents 成员。

如果 clear 不是 Screen 的友元，上面的代码将无法通过编译，因为此时 clear 将不能访问 Screen 的 height、width 和 contents 成员。而当 Screen 将 Window_mgr 指定为其友元之后，Screen 的所有成员对于 Window_mgr 就都变成可见的了。

必须要注意的一点是，友元关系不存在传递性。也就是说，如果 Window_mgr 有它自己的友元，则这些友元并不能理所当然地具有访问 Screen 的特权。

每个类负责控制自己的友元类或友元函数。

令成员函数作为友元

除了令整个 Window_mgr 作为友元之外，Screen 还可以只为 clear 提供访问权限。当把一个成员函数声明成友元时，我们必须明确指出该成员函数属于哪个类：

```
class Screen {
    // Window_mgr::clear 必须在 Screen 类之前被声明
    friend void Window_mgr::clear(ScreenIndex);
    // Screen 类的剩余部分
};
```

281

要想令某个成员函数作为友元，我们必须仔细组织程序的结构以满足声明和定义的彼此依赖关系。在这个例子中，我们必须按照如下方式设计程序：

- 首先定义 Window_mgr 类，其中声明 clear 函数，但是不能定义它。在 clear 使用 Screen 的成员之前必须先声明 Screen。
- 接下来定义 Screen，包括对于 clear 的友元声明。
- 最后定义 clear，此时它才可以使用 Screen 的成员。

函数重载和友元

尽管重载函数的名字相同，但它们仍然是不同的函数。因此，如果一个类想把一组重载函数声明成它的友元，它需要对这组函数中的每一个分别声明：

```
// 重载的 storeOn 函数
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);
class Screen {
    // storeOn 的 ostream 版本能访问 Screen 对象的私有部分
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};
```

Screen 类把接受 ostream&的 storeOn 函数声明成它的友元，但是接受 BitMap&作为参数的版本仍然不能访问 Screen。

友元声明和作用域

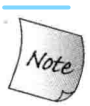
类和非成员函数的声明不是必须在它们的友元声明之前。当一个名字第一次出现在一个友元声明中时，我们隐式地假定该名字在当前作用域中是可见的。然而，友元本身不一定真的声明在当前作用域中（参见 7.2.1 节，第 241 页）。

甚至就算在类的内部定义该函数，我们也必须在类的外部提供相应的声明从而使得函数可见。换句话说，即使我们仅仅是用声明友元的类的成员调用该友元函数，它也必须是被声明过的：

```
struct X {
    friend void f() { /* 友元函数可以定义在类的内部*/ }
    X() { f(); }                          // 错误：f 还没有被声明
    void g();
    void h();
};
void X::g() { return f(); }               // 错误：f 还没有被声明
void f();                                 // 声明那个定义在 X 中的函数
void X::h() { return f(); }               // 正确：现在 f 的声明在作用域中了
```

282

关于这段代码最重要的是理解友元声明的作用是影响访问权限，它本身并非普通意义上的声明。



请注意，有的编译器并不强制执行上述关于友元的限定规则（参见 7.2.1 节，第 241 页）。

7.3.4 节练习

练习 7.32：定义你自己的 Screen 和 Window_mgr，其中 clear 是 Window_mgr 的成员，是 Screen 的友元。

7.4 类的作用域

每个类都会定义它自己的作用域。在类的作用域之外，普通的数据和函数成员只能由对象、引用或者指针使用成员访问运算符（参见 4.6 节，第 133 页）来访问。对于类类型成员则使用作用域运算符访问。不论哪种情况，跟在运算符之后的名字都必须是对应类的成员：

```
Screen::pos ht = 24, wd = 80;          // 使用 Screen 定义的 pos 类型
Screen scr(ht, wd, ' ');
Screen *p = &scr;
char c = scr.get();                    // 访问 scr 对象的 get 成员
c = p->get();                          // 访问 p 所指对象的 get 成员
```

作用域和定义在类外部的成员

一个类就是一个作用域的事实能够很好地解释为什么当我们在类的外部定义成员函数时必须同时提供类名和函数名（参见 7.1.2 节，第 230 页）。在类的外部，成员的名字被隐藏起来了。

一旦遇到了类名，定义的剩余部分就在类的作用域之内了，这里的剩余部分包括参数列表和函数体。结果就是，我们可以直接使用类的其他成员而无须再次授权了。

例如，我们回顾一下 Window_mgr 类的 clear 成员（参见 7.3.4 节，第 251 页），该函数的参数用到了 Window_mgr 类定义的一种类型：

```
void Window_mgr::clear(ScreenIndex i)
{
    Screen &s = screens[i];
    s.contents = string(s.height * s.width, ' ');
}
```

因为编译器在处理参数列表之前已经明确了我们当前正位于 Window_mgr 类的作用域中，所以不必再专门说明 ScreenIndex 是 Window_mgr 类定义的。出于同样的原因，编译器也能知道函数体中用到的 screens 也是在 Window_mgr 类中定义的。

283

另一方面，函数的返回类型通常出现在函数名之前。因此当成员函数定义在类的外部时，返回类型中使用的名字都位于类的作用域之外。这时，返回类型必须指明它是哪个类的成员。例如，我们可能向 Window_mgr 类添加一个新的名为 addScreen 的函数，它负责向显示器添加一个新的屏幕。这个成员的返回类型将是 ScreenIndex，用户可以通过它定位到指定的 Screen：

```
class Window_mgr {
public:
    // 向窗口添加一个 Screen，返回它的编号
    ScreenIndex addScreen(const Screen&);
    // 其他成员与之前的版本一致
};
// 首先处理返回类型，之后我们才进入 Window_mgr 的作用域
```

```
Window_mgr::ScreenIndex
Window_mgr::addScreen(const Screen &s)
{
    screens.push_back(s);
    return screens.size() - 1;
}
```

因为返回类型出现在类名之前，所以事实上它是位于 Window_mgr 类的作用域之外的。在这种情况下，要想使用 ScreenIndex 作为返回类型，我们必须明确指定哪个类定义了它。

7.4 节练习

练习 7.33：如果我们给 Screen 添加一个如下所示的 size 成员将发生什么情况？如果出现了问题，请尝试修改它。

```
pos Screen::size() const
{
    return height * width;
}
```

7.4.1 名字查找与类的作用域

在目前为止，我们编写的程序中，**名字查找**（name lookup）（寻找与所用名字最匹配的声明的过程）的过程比较直截了当：

- 首先，在名字所在的块中寻找其声明语句，只考虑在名字的使用之前出现的声明。
- 如果没找到，继续查找外层作用域。
- 如果最终没有找到匹配的声明，则程序报错。

284

对于定义在类内部的成员函数来说，解析其中名字的方式与上述的查找规则有所区别，不过在当前的这个例子中体现得不太明显。类的定义分两步处理：

- 首先，编译成员的声明。
- 直到类全部可见后才编译函数体。

编译器处理完类中的全部声明后才会处理成员函数的定义。

按照这种两阶段的方式处理类可以简化类代码的组织方式。因为成员函数体直到整个类可见后才会被处理，所以它能使用类中定义的任何名字。相反，如果函数的定义和成员的声明被同时处理，那么我们将不得不在成员函数中只使用那些已经出现的名字。

用于类成员声明的名字查找

这种两阶段的处理方式只适用于成员函数中使用的名字。声明中使用的名字，包括返回类型或者参数列表中使用的名字，都必须在使用前确保可见。如果某个成员的声明使用了类中尚未出现的名字，则编译器将会在定义该类的作用域中继续查找。例如：

```
typedef double Money;
string bal;
class Account {
public:
```

```
    Money balance() { return bal; }
private:
    Money bal;
    // ...
};
```

当编译器看到 balance 函数的声明语句时，它将在 Account 类的范围内寻找对 Money 的声明。编译器只考虑 Account 中在使用 Money 前出现的声明，因为没找到匹配的成员，所以编译器会接着到 Account 的外层作用域中查找。在这个例子中，编译器会找到 Money 的 typedef 语句，该类型被用作 balance 函数的返回类型以及数据成员 bal 的类型。另一方面，balance 函数体在整个类可见后才被处理，因此，该函数的 return 语句返回名为 bal 的成员，而非外层作用域的 string 对象。

类型名要特殊处理

一般来说，内层作用域可以重新定义外层作用域中的名字，即使该名字已经在内层作用域中使用过。然而在类中，如果成员使用了外层作用域中的某个名字，而该名字代表一种类型，则类不能在之后重新定义该名字：

285

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; }   // 使用外层作用域的 Money
private:
    typedef double Money;             // 错误：不能重新定义 Money
    Money bal;
    // ...
};
```

需要特别注意的是，即使 Account 中定义的 Money 类型与外层作用域一致，上述代码仍然是错误的。

尽管重新定义类型名字是一种错误的行为，但是编译器并不为此负责。一些编译器仍将顺利通过这样的代码，而忽略代码有错的事实。

类型名的定义通常出现在类的开始处，这样就能确保所有使用该类型的成员都出现在类名的定义之后。

成员定义中的普通块作用域的名字查找

成员函数中使用的名字按照如下方式解析：

- 首先，在成员函数内查找该名字的声明。和前面一样，只有在函数使用之前出现的声明才被考虑。
- 如果在成员函数内没有找到，则在类内继续查找，这时类的所有成员都可以被考虑。
- 如果类内也没找到该名字的声明，在成员函数定义之前的作用域内继续查找。

一般来说，不建议使用其他成员的名字作为某个成员函数的参数。不过为了更好地解释名字的解析过程，我们不妨在 dummy_fcn 函数中暂时违反一下这个约定：

```
// 注意：这段代码仅为了说明而用，不是一段很好的代码
// 通常情况下不建议为参数和成员使用同样的名字
int height;                    // 定义了一个名字，稍后将在 Screen 中使用
```

```
class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height) {
        cursor = width * height;      // 哪个 height?是那个参数
    }
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};
```

286

当编译器处理 dummy_fcn 中的乘法表达式时，它首先在函数作用域内查找表达式中用到的名字。函数的参数位于函数作用域内，因此 dummy_fcn 函数体内用到的名字 height 指的是参数声明。

在此例中，height 参数隐藏了同名的成员。如果想绕开上面的查找规则，应该将代码变为：

```
// 不建议的写法：成员函数中的名字不应该隐藏同名的成员
void Screen::dummy_fcn(pos height) {
    cursor = width * this->height;        // 成员 height
    // 另外一种表示该成员的方式
    cursor = width * Screen::height;      // 成员 height
}
```

尽管类的成员被隐藏了，但我们仍然可以通过加上类的名字或显式地使用 this 指针来强制访问成员。

其实最好的确保我们使用 height 成员的方法是给参数起个其他名字：

```
// 建议的写法：不要把成员名字作为参数或其他局部变量使用
void Screen::dummy_fcn(pos ht) {
    cursor = width * height;          // 成员 height
}
```

在此例中，当编译器查找名字 height 时，显然在 dummy_fcn 函数内部是找不到的。编译器接着会在 Screen 内查找匹配的声明，即使 height 的声明出现在 dummy_fcn 使用它之后，编译器也能正确地解析函数使用的是名为 height 的成员。

类作用域之后，在外围的作用域中查找

如果编译器在函数和类的作用域中都没有找到名字，它将接着在外围的作用域中查找。在我们的例子中，名字 height 定义在外层作用域中，且位于 Screen 的定义之前。然而，外层作用域中的对象被名为 height 的成员隐藏掉了。因此，如果我们需要的是外层作用域中的名字，可以显式地通过作用域运算符来进行请求：

```
// 不建议的写法：不要隐藏外层作用域中可能被用到的名字
void Screen::dummy_fcn(pos height) {
    cursor = width * ::height;            // 哪个 height? 是那个全局的
}
```

尽管外层的对象被隐藏掉了，但我们仍然可以用作用域运算符访问它。

287

在文件中名字的出现处对其进行解析

当成员定义在类的外部时，名字查找的第三步不仅要考虑类定义之前的全局作用域中的声明，还需要考虑在成员函数定义之前的全局作用域中的声明。例如：

```
int height;                          // 定义了一个名字，稍后将在 Screen 中使用
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0;                  // 隐藏了外层作用域中的 height
};
Screen::pos verify(Screen::pos);
void Screen::setHeight(pos var) {
    // var: 参数
    // height: 类的成员
    // verify: 全局函数
    height = verify(var);
}
```

请注意，全局函数 verify 的声明在 Screen 类的定义之前是不可见的。然而，名字查找的第三步包括了成员函数出现之前的全局作用域。在此例中，verify 的声明位于 setHeight 的定义之前，因此可以被正常使用。

7.4.1 节练习

练习 7.34：如果我们把第 256 页 Screen 类的 pos 的 typedef 放在类的最后一行会发生什么情况？

练习 7.35：解释下面代码的含义，说明其中的 Type 和 initVal 分别使用了哪个定义。如果代码存在错误，尝试修改它。

```
typedef string Type;
Type initVal();
class Exercise {
public:
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};
Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}
```

7.5 构造函数再探

288

对于任何 C++的类来说，构造函数都是其中重要的组成部分。我们已经在 7.1.4 节（第 235 页）中介绍了构造函数的基础知识，本节将继续介绍构造函数的一些其他功能，并对

之前已经介绍的内容进行一些更深入的讨论。

7.5.1 构造函数初始值列表

当我们定义变量时习惯于立即对其进行初始化，而非先定义、再赋值：

```
string foo = "Hello World!";          // 定义并初始化
string bar;                           // 默认初始化成空 string 对象
bar = "Hello World!";                 // 为 bar 赋一个新值
```

就对象的数据成员而言，初始化和赋值也有类似的区别。如果没有在构造函数的初始值列表中显式地初始化成员，则该成员将在构造函数体之前执行默认初始化。例如：

```
// Sales_data 构造函数的一种写法，虽然合法但比较草率：没有使用构造函数初始值
Sales_data::Sales_data(const string &s,
                       unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
```

这段代码和我们在 237 页的原始定义效果是相同的：当构造函数完成后，数据成员的值相同。区别是原来的版本初始化了它的数据成员，而这个版本是对数据成员执行了赋值操作。这一区别到底会有什么深层次的影响完全依赖于数据成员的类型。

构造函数的初始值有时必不可少

有时我们可以忽略数据成员初始化和赋值之间的差异，但并非总能这样。如果成员是 const 或者是引用的话，必须将其初始化。类似的，当成员属于某种类类型且该类没有定义默认构造函数时，也必须将这个成员初始化。例如：

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

289 和其他常量对象或者引用一样，成员 ci 和 ri 都必须被初始化。因此，如果我们没有为它们提供构造函数初始值的话将引发错误：

```
// 错误：ci 和 ri 必须被初始化
ConstRef::ConstRef(int ii)
{ // 赋值：
    i = ii;                   // 正确
    ci = ii;                  // 错误：不能给 const 赋值
    ri = i;                   // 错误：ri 没被初始化
}
```

随着构造函数体一开始执行，初始化就完成了。我们初始化 const 或者引用类型的数据成员的唯一机会就是通过构造函数初始值，因此该构造函数的正确形式应该是：

```
// 正确：显式地初始化引用和 const 成员
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) { }
```

如果成员是 const、引用，或者属于某种未提供默认构造函数的类类型，我们必须通过构造函数初始值列表为这些成员提供初值。

建议：使用构造函数初始值

在很多类中，初始化和赋值的区别事关底层效率问题：前者直接初始化数据成员，后者则先初始化再赋值。

除了效率问题外更重要的是，一些数据成员必须被初始化。建议读者养成使用构造函数初始值的习惯，这样能避免某些意想不到的编译错误，特别是遇到有的类含有需要构造函数初始值的成员时。

成员初始化的顺序

显然，在构造函数初始值中每个成员只能出现一次。否则，给同一个成员赋两个不同的初始值有什么意义呢？

不过让人稍感意外的是，构造函数初始值列表只说明用于初始化成员的值，而不限定初始化的具体执行顺序。

成员的初始化顺序与它们在类定义中的出现顺序一致：第一个成员先被初始化，然后第二个，以此类推。构造函数初始值列表中初始值的前后位置关系不会影响实际的初始化顺序。

一般来说，初始化的顺序没什么特别要求。不过如果一个成员是用另一个成员来初始化的，那么这两个成员的初始化顺序就很关键了。

举个例子，考虑下面这个类：

290

```
class X {
    int i;
    int j;
public:
    // 未定义的：i 在 j 之前被初始化
    X(int val): j(val), i(j) { }
};
```

在此例中，从构造函数初始值的形式上来看仿佛是先用 val 初始化了 j，然后再用 j 初始化 i。实际上，i 先被初始化，因此这个初始值的效果是试图使用未定义的值 j 初始化 i！

有的编译器具备一项比较友好的功能，即当构造函数初始值列表中的数据成员顺序与这些成员声明的顺序不符时会生成一条警告信息。

最好令构造函数初始值的顺序与成员声明的顺序保持一致。而且如果可能的话，尽量避免使用某些成员初始化其他成员。

如果可能的话，最好用构造函数的参数作为成员的初始值，而尽量避免使用同一个对

象的其他成员。这样的好处是我们可以不必考虑成员的初始化顺序。例如，X 的构造函数如果写成如下的形式效果会更好：

```
X(int val): i(val), j(val) { }
```

在这个版本中，i 和 j 初始化的顺序就没什么影响了。

默认实参和构造函数

Sales_data 默认构造函数的行为与只接受一个 string 实参的构造函数差不多。唯一的区别是接受 string 实参的构造函数使用这个实参初始化 bookNo，而默认构造函数（隐式地）使用 string 的默认构造函数初始化 bookNo。我们可以把它们重写成一个使用默认实参（参见 6.5.1 节，第 211 页）的构造函数：

```
class Sales_data {
public:
    // 定义默认构造函数，令其与只接受一个 string 实参的构造函数功能相同
    Sales_data(std::string s = ""): bookNo(s) { }
    // 其他构造函数与之前一致
    Sales_data(std::string s, unsigned cnt, double rev):
          bookNo(s), units_sold(cnt), revenue(rev*cnt) { }
    Sales_data(std::istream &is) { read(is, *this); }
    // 其他成员与之前的版本一致
};
```

在上面这段程序中，类的接口与第 237 页的代码是一样的。当没有给定实参，或者给定了一个 string 实参时，两个版本的类创建了相同的对象。因为我们不提供实参也能调用上述的构造函数，所以该构造函数实际上为我们的类提供了默认构造函数。

291

如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认构造函数。

值得注意的是，我们不应该为 Sales_data 接受三个实参的构造函数提供默认值。因为如果用户为售出书籍的数量提供了一个非零的值，则我们就会期望用户同时提供这些书籍的售出价格。

7.5.1 节练习

练习 7.36：下面的初始值是错误的，请找出问题所在并尝试修改它。

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

练习 7.37：使用本节提供的 Sales_data 类，确定初始化下面的变量时分别使用了哪个构造函数，然后罗列出每个对象所有数据成员的值。

```
Sales_data first_item(cin);
int main() {
    Sales_data next;
    Sales_data last("9-999-99999-9");
}
```

练习 7.38：有些情况下我们希望提供 cin 作为接受 istream&参数的构造函数的默认实参，请声明这样的构造函数。

练习 7.39：如果接受 string 的构造函数和接受 istream&的构造函数都使用默认实参，这种行为合法吗？如果不，为什么？

练习 7.40：从下面的抽象概念中选择一个（或者你自己指定一个），思考这样的类需要哪些数据成员，提供一组合理的构造函数并阐明这样做的原因。

(a) Book (b) Date (c) Employee
(d) Vehicle (e) Object (f) Tree

7.5.2 委托构造函数

C++11 新标准扩展了构造函数初始值的功能，使得我们可以定义所谓的**委托构造函数**（delegating constructor）。一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些（或者全部）职责委托给了其他构造函数。

C++11

和其他构造函数一样，一个委托构造函数也有一个成员初始值的列表和一个函数体。在委托构造函数内，成员初始值列表只有一个唯一的入口，就是类名本身。和其他成员初始值一样，类名后面紧跟圆括号括起来的参数列表，参数列表必须与类中另外一个构造函数匹配。

292

举个例子，我们使用委托构造函数重写 Sales_data 类，重写后的形式如下所示：

```
class Sales_data {
public:
    // 非委托构造函数使用对应的实参初始化成员
    Sales_data(std::string s, unsigned cnt, double price):
            bookNo(s), units_sold(cnt), revenue(cnt*price) { }
    // 其余构造函数全都委托给另一个构造函数
    Sales_data(): Sales_data("", 0, 0) {}
    Sales_data(std::string s): Sales_data(s, 0,0) {}
    Sales_data(std::istream &is): Sales_data()
                                        { read(is, *this); }
    // 其他成员与之前的版本一致
};
```

在这个 Sales_data 类中，除了一个构造函数外其他的都委托了它们的工作。第一个构造函数接受三个实参，使用这些实参初始化数据成员，然后结束工作。我们定义默认构造函数令其使用三参数的构造函数完成初始化过程，它也无须执行其他任务，这一点从空的构造函数体能看得出来。接受一个 string 的构造函数同样委托给了三参数的版本。

接受 istream&的构造函数也是委托构造函数，它委托给了默认构造函数，默认构造函数又接着委托给三参数构造函数。当这些受委托的构造函数执行完后，接着执行 istream&构造函数体的内容。它的构造函数体调用 read 函数读取给定的 istream。

当一个构造函数委托给另一个构造函数时，受委托的构造函数的初始值列表和函数体被依次执行。在 Sales_data 类中，受委托的构造函数体恰好是空的。假如函数体包含有代码的话，将先执行这些代码，然后控制权才会交还给委托者的函数体。

7.5.2 节练习

练习 7.41：使用委托构造函数重新编写你的 Sales_data 类，给每个构造函数体添加一条语句，令其一旦执行就打印一条信息。用各种可能的方式分别创建 Sales_data 对象，认真研究每次输出的信息直到你确实理解了委托构造函数的执行顺序。

练习 7.42：对于你在练习 7.40（参见 7.5.1 节，第 261 页）中编写的类，确定哪些构造函数可以使用委托。如果可以的话，编写委托构造函数。如果不可以，从抽象概念列表中重新选择一个你认为可以使用委托构造函数的，为挑选出的这个概念编写类定义。

7.5.3 默认构造函数的作用

293 当对象被默认初始化或值初始化时自动执行默认构造函数。默认初始化在以下情况下发生：

- 当我们在块作用域内不使用任何初始值定义一个非静态变量（参见 2.2.1 节，第 39 页）或者数组时（参见 3.5.1 节，第 101 页）。
- 当一个类本身含有类类型的成员且使用合成的默认构造函数时（参见 7.1.4 节，第 235 页）。
- 当类类型的成员没有在构造函数初始值列表中显式地初始化时（参见 7.1.4 节，第 237 页）。

值初始化在以下情况下发生：

- 在数组初始化的过程中如果我们提供的初始值数量少于数组的大小时（参见 3.5.1 节，第 101 页）。
- 当我们不使用初始值定义一个局部静态变量时（参见 6.1.1 节，第 185 页）。
- 当我们通过书写形如 T( ) 的表达式显式地请求值初始化时，其中 T 是类型名（vector 的一个构造函数只接受一个实参用于说明 vector 大小（参见 3.3.1 节，第 88 页），它就是使用一个这种形式的实参来对它的元素初始化器进行值初始化）。

类必须包含一个默认构造函数以便在上述情况下使用，其中的大多数情况非常容易判断。

不那么明显的一种情况是类的某些数据成员缺少默认构造函数：

```
class NoDefault {
public:
    NoDefault(const std::string&);
    // 还有其他成员，但是没有其他构造函数了
};
struct A {          // 默认情况下 my_mem 是 public 的（参见 7.2 节，第 240 页）
    NoDefault my_mem;
};
A a;                // 错误：不能为 A 合成构造函数
struct B {
    B() {}          // 错误：b_member 没有初始值
    NoDefault b_member;
};
```

Best Practices 在实际中，如果定义了其他构造函数，那么最好也提供一个默认构造函数。

使用默认构造函数

294

下面的 obj 的声明可以正常编译通过：

```
Sales_data obj();                              // 正确：定义了一个函数而非对象
if (obj.isbn() == Primer_5th_ed.isbn())        // 错误：obj 是一个函数
```

但当我们试图使用 obj 时，编译器将报错，提示我们不能对函数使用成员访问运算符。问题在于，尽管我们想声明一个默认初始化的对象，obj 实际的含义却是一个不接受任何参数的函数并且其返回值是 Sales_data 类型的对象。

如果想定义一个使用默认构造函数进行初始化的对象，正确的方法是去掉对象名之后的空的括号对：

```
// 正确：obj 是个默认初始化的对象
Sales_data obj;
```

对于 C++的新手程序员来说有一种常犯的错误，它们试图以如下的形式声明一个用默认构造函数初始化的对象：

```
Sales_data obj();              // 错误：声明了一个函数而非对象
Sales_data obj2;               // 正确：obj2 是一个对象而非函数
```

7.5.3 节练习

练习 7.43：假定有一个名为 NoDefault 的类，它有一个接受 int 的构造函数，但是没有默认构造函数。定义类 C，C 有一个 NoDefault 类型的成员，定义 C 的默认构造函数。

练习 7.44：下面这条声明合法吗？如果不，为什么？

```
vector<NoDefault> vec(10);
```

练习 7.45：如果在上一个练习中定义的 vector 的元素类型是 C，则声明合法吗？为什么？

练习 7.46：下面哪些论断是不正确的？为什么？

（a）一个类必须至少提供一个构造函数。

（b）默认构造函数是参数列表为空的构造函数。

（c）如果对于类来说不存在有意义的默认值，则类不应该提供默认构造函数。

（d）如果类没有定义默认构造函数，则编译器将为其生成一个并把每个数据成员初始化成相应类型的默认值。

7.5.4 隐式的类类型转换

4.11 节（第 141 页）曾经介绍过 C++语言在内置类型之间定义了几种自动转换规则。同样的，我们也能为类定义隐式转换规则。如果构造函数只接受一个实参，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作**转换构造函数**（converting constructor）。我们将在 14.9 节（第 514 页）介绍如何定义将一种类类型转换为另一种类类型的转换规则。

295

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。

在 Sales_data 类中，接受 string 的构造函数和接受 istream 的构造函数分别定义了从这两种类型向 Sales_data 隐式转换的规则。也就是说，在需要使用 Sales_data 的地方，我们可以使用 string 或者 istream 作为替代：

```
string null_book = "9-999-99999-9";
// 构造一个临时的 Sales_data 对象
// 该对象的 units_sold 和 revenue 等于 0，bookNo 等于 null_book
item.combine(null_book);
```

在这里我们用一个 string 实参调用了 Sales_data 的 combine 成员。该调用是合法的，编译器用给定的 string 自动创建了一个 Sales_data 对象。新生成的这个（临时）Sales_data 对象被传递给 combine。因为 combine 的参数是一个常量引用，所以我们可以给该参数传递一个临时量。

只允许一步类类型转换

在 4.11.2 节（第 143 页）中我们指出，编译器只会自动地执行一步类型转换。例如，因为下面的代码隐式地使用了两种转换规则，所以它是错误的：

```
// 错误：需要用户定义的两种转换：
// (1) 把“9-999-99999-9”转换成 string
// (2) 再把这个（临时的）string 转换成 Sales_data
item.combine("9-999-99999-9");
```

如果我们想完成上述调用，可以显式地把字符串转换成 string 或者 Sales_data 对象：

```
// 正确：显式地转换成 string，隐式地转换成 Sales_data
item.combine(string("9-999-99999-9"));
// 正确：隐式地转换成 string，显式地转换成 Sales_data
item.combine(Sales_data("9-999-99999-9"));
```

类类型转换不是总有效

是否需要从 string 到 Sales_data 的转换依赖于我们对用户使用该转换的看法。在此例中，这种转换可能是对的。null_book 中的 string 可能表示了一个不存在的 ISBN 编号。

另一个是从 istream 到 Sales_data 的转换：

```
// 使用 istream 构造函数创建一个函数传递给 combine
item.combine(cin);
```

这段代码隐式地把 cin 转换成 Sales_data，这个转换执行了接受一个 istream 的 Sales_data 构造函数。该构造函数通过读取标准输入创建了一个(临时的)Sales_data 对象，随后将得到的对象传递给 combine。

296 Sales_data 对象是个临时量（参见 2.4.1 节，第 54 页），一旦 combine 完成我们就不能再访问它了。实际上，我们构建了一个对象，先将它的值加到 item 中，随后将其丢弃。

抑制构造函数定义的隐式转换

在要求隐式转换的程序上下文中，我们可以通过将构造函数声明为 **explicit** 加以阻止：

```
class Sales_data {
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
               bookNo(s), units_sold(n), revenue(p*n) { }
    explicit Sales_data(const std::string &s): bookNo(s) { }
    explicit Sales_data(std::istream&);
    // 其他成员与之前的版本一致
};
```

此时，没有任何构造函数能用于隐式地创建 Sales_data 对象，之前的两种用法都无法通过编译：

```
item.combine(null_book);      // 错误：string 构造函数是 explicit 的
item.combine(cin);            // 错误：istream 构造函数是 explicit 的
```

关键字 explicit 只对一个实参的构造函数有效。需要多个实参的构造函数不能用于执行隐式转换，所以无须将这些构造函数指定为 explicit 的。只能在类内声明构造函数时使用 explicit 关键字，在类外部定义时不应重复：

```
// 错误：explicit 关键字只允许出现在类内的构造函数声明处
explicit Sales_data::Sales_data(istream& is)
{
    read(is, *this);
}
```

explicit 构造函数只能用于直接初始化

发生隐式转换的一种情况是当我们执行拷贝形式的初始化时（使用=）（参见 3.2.1 节，第 76 页）。此时，我们只能使用直接初始化而不能使用 explicit 构造函数：

```
Sales_data item1(null_book);      // 正确：直接初始化
// 错误：不能将 explicit 构造函数用于拷贝形式的初始化过程
Sales_data item2 = null_book;
```

当我们用 explicit 关键字声明构造函数时，它将只能以直接初始化的形式使用（参见 3.2.1 节，第 76 页）。而且，编译器将不会在自动转换过程中使用该构造函数。

为转换显式地使用构造函数

297

尽管编译器不会将 explicit 的构造函数用于隐式转换过程，但是我们可以使用这样的构造函数显式地强制进行转换：

```
// 正确：实参是一个显式构造的 Sales_data 对象
item.combine(Sales_data(null_book));
// 正确：static_cast 可以使用 explicit 的构造函数
item.combine(static_cast<Sales_data>(cin));
```

在第一个调用中，我们直接使用 Sales_data 的构造函数，该调用通过接受 string 的

构造函数创建了一个临时的 Sales_data 对象。在第二个调用中，我们使用 static_cast（参见 4.11.3 节，第 145 页）执行了显式的而非隐式的转换。其中，static_cast 使用 istream 构造函数创建了一个临时的 Sales_data 对象。

标准库中含有显式构造函数的类

我们用过的一些标准库中的类含有单参数的构造函数：

- 接受一个单参数的 const char*的 string 构造函数（参见 3.2.1 节，第 76 页）不是 explicit 的。
- 接受一个容量参数的 vector 构造函数（参见 3.3.1 节，第 87 页）是 explicit 的。

7.5.4 节练习

练习 7.47：说明接受一个 string 参数的 Sales_data 构造函数是否应该是 explicit 的，并解释这样做的优缺点。

练习 7.48：假定 Sales_data 的构造函数不是 explicit 的，则下述定义将执行什么样的操作？

```
string null_isbn("9-999-99999-9");
Sales_data item1(null_isbn);
Sales_data item2("9-999-99999-9");
```

如果 Sales_data 的构造函数是 explicit 的，又会发生什么呢？

练习 7.49：对于 combine 函数的三种不同声明，当我们调用 i.combine(s)时分别发生什么情况？其中 i 是一个 Sales_data，而 s 是一个 string 对象。

(a) Sales_data &combine(Sales_data);
(b) Sales_data &combine(Sales_data&);
(c) Sales_data &combine(const Sales_data&) const;

练习 7.50：确定在你的 Person 类中是否有一些构造函数应该是 explicit 的。

练习 7.51：vector 将其单参数的构造函数定义成 explicit 的，而 string 则不是，你觉得原因何在？

7.5.5 聚合类

298

聚合类（aggregate class）使得用户可以直接访问其成员，并且具有特殊的初始化语法形式。当一个类满足如下条件时，我们说它是聚合的：

- 所有成员都是 public 的。
- 没有定义任何构造函数。
- 没有类内初始值（参见 2.6.1 节，第 64 页）。
- 没有基类，也没有 virtual 函数，关于这部分知识我们将在第 15 章详细介绍。

例如，下面的类是一个聚合类：

```
struct Data {
    int ival;
    string s;
};
```

我们可以提供一个花括号括起来的成员初始值列表，并用它初始化聚合类的数据成员：

```
// val1.ival = 0; val1.s = string("Anna")
Data val1 = { 0, "Anna" };
```

初始值的顺序必须与声明的顺序一致，也就是说，第一个成员的初始值要放在第一个，然后是第二个，以此类推。下面的例子是错误的：

```
// 错误：不能使用"Anna"初始化 ival，也不能使用 1024 初始化 s
Data val2 = { "Anna" , 1024 };
```

与初始化数组元素的规则（参见 3.5.1 节，第 101 页）一样，如果初始值列表中的元素个数少于类的成员数量，则靠后的成员被值初始化（参见 3.5.1 节，第 101 页）。初始值列表的元素个数绝对不能超过类的成员数量。

值得注意的是，显式地初始化类的对象的成员存在三个明显的缺点：

- 要求类的所有成员都是 public 的。
- 将正确初始化每个对象的每个成员的重任交给了类的用户（而非类的作者）。因为用户很容易忘掉某个初始值，或者提供一个不恰当的初始值，所以这样的初始化过程冗长乏味且容易出错。
- 添加或删除一个成员之后，所有的初始化语句都需要更新。

7.5.5 节练习

299

练习 7.52：使用 2.6.1 节（第 64 页）的 Sales_data 类，解释下面的初始化过程。如果存在问题，尝试修改它。

```
Sales_data item = {"978-0590353403", 25, 15.99};
```

7.5.6 字面值常量类

在 6.5.2 节（第 214 页）中我们提到过 constexpr 函数的参数和返回值必须是字面值类型。除了算术类型、引用和指针外，某些类也是字面值类型。和其他类不同，字面值类型的类可能含有 constexpr 函数成员。这样的成员必须符合 constexpr 函数的所有要求，它们是隐式 const 的（参见 7.1.2 节，第 231 页）。

数据成员都是字面值类型的聚合类（参见 7.5.5 节，第 266 页）是字面值常量类。如果一个类不是聚合类，但它符合下述要求，则它也是一个字面值常量类：

- 数据成员都必须是字面值类型。
- 类必须至少含有一个 constexpr 构造函数。
- 如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一条常量表达式（参见 2.4.4 节，第 58 页）；或者如果成员属于某种类类型，则初始值必须使用成员自己的 constexpr 构造函数。
- 类必须使用析构函数的默认定义，该成员负责销毁类的对象（参见 7.1.5 节，第 239 页）。

constexpr 构造函数

尽管构造函数不能是 const 的（参见 7.1.4 节，第 235 页），但是字面值常量类的构造函数可以是 constexpr（参见 6.5.2 节，第 213 页）函数。事实上，一个字面值常量类必须至少提供一个 constexpr 构造函数。

constexpr 构造函数可以声明成= default（参见 7.1.4 节，第 237 页）的形式（或者是删除函数的形式，我们将在 13.1.6 节（第 449 页）介绍相关知识）。否则，constexpr 构造函数就必须既符合构造函数的要求（意味着不能包含返回语句），又符合 constexpr 函数的要求（意味着它能拥有的唯一可执行语句就是返回语句（参见 6.5.2 节，第 214 页））。综合这两点可知，constexpr 构造函数体一般来说应该是空的。我们通过前置关键字 constexpr 就可以声明一个 constexpr 构造函数了：

C++11

```
class Debug {
public:
    constexpr Debug(bool b = true): hw(b), io(b), other(b) { }
    constexpr Debug(bool h, bool i, bool o):
                              hw(h), io(i), other(o) { }
    constexpr bool any() { return hw || io || other; }
    void set_io(bool b) { io = b; }
    void set_hw(bool b) { hw = b; }
    void set_other(bool b) { hw = b; }
private:
    bool hw;                          // 硬件错误，而非 IO 错误
    bool io;                          // IO 错误
    bool other;                       // 其他错误
};
```

300

constexpr 构造函数必须初始化所有数据成员，初始值或者使用 constexpr 构造函数，或者是一条常量表达式。

constexpr 构造函数用于生成 constexpr 对象以及 constexpr 函数的参数或返回类型：

```
constexpr Debug io_sub(false, true, false);        // 调试 IO
if (io_sub.any())                                  // 等价于 if(true)
    cerr << "print appropriate error messages" << endl;
constexpr Debug prod(false);                       // 无调试
if (prod.any())                                    // 等价于 if(false)
    cerr << "print an error message" << endl;
```

7.5.6 节练习

练习 7.53：定义你自己的 Debug。

练习 7.54：Debug 中以 set_开头的成员应该被声明成 constexpr 吗？如果不，为什么？

练习 7.55：7.5.5 节（第 266 页）的 Data 类是字面值常量类吗？请解释原因。

7.6 类的静态成员

有的时候类需要它的一些成员与类本身直接相关，而不是与类的各个对象保持关联。例如，一个银行账户类可能需要一个数据成员来表示当前的基准利率。在此例中，我们希望利率与类关联，而非与类的每个对象关联。从实现效率的角度来看，没必要每个对象都存储利率信息。而且更加重要的是，一旦利率浮动，我们希望所有的对象都能使用新值。

声明静态成员

我们通过在成员的声明之前加上关键字 static 使得其与类关联在一起。和其他成员一样，静态成员可以是 public 的或 private 的。静态数据成员的类型可以是常量、引用、指针、类类型等。

举个例子，我们定义一个类，用它表示银行的账户记录：

301

```
class Account {
public:
    void calculate() { amount += amount * interestRate; }
    static double rate() { return interestRate; }
    static void rate(double);
private:
    std::string owner;
    double amount;
    static double interestRate;
    static double initRate();
};
```

类的静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据。因此，每个 Account 对象将包含两个数据成员：owner 和 amount。只存在一个 interestRate 对象而且它被所有 Account 对象共享。

类似的，静态成员函数也不与任何对象绑定在一起，它们不包含 this 指针。作为结果，静态成员函数不能声明成 const 的，而且我们也不能在 static 函数体内使用 this 指针。这一限制既适用于 this 的显式使用，也对调用非静态成员的隐式使用有效。

使用类的静态成员

我们使用作用域运算符直接访问静态成员：

```
double r;
r = Account::rate();          // 使用作用域运算符访问静态成员
```

虽然静态成员不属于类的某个对象，但是我们仍然可以使用类的对象、引用或者指针来访问静态成员：

```
Account ac1;
Account *ac2 = &ac1;
// 调用静态成员函数 rate 的等价形式
r = ac1.rate();           // 通过 Account 的对象或引用
r = ac2->rate();          // 通过指向 Account 对象的指针
```

成员函数不用通过作用域运算符就能直接使用静态成员：

```
class Account {
public:
    void calculate() { amount += amount * interestRate; }
private:
    static double interestRate;
    // 其他成员与之前的版本一致
};
```

302 定义静态成员

和其他的成员函数一样，我们既可以在类的内部也可以在类的外部定义静态成员函数。当在类的外部定义静态成员时，不能重复 static 关键字，该关键字只出现在类内部的声明语句：

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```

和类的所有成员一样，当我们指向类外部的静态成员时，必须指明成员所属的类名。static 关键字则只出现在类内部的声明语句中。

因为静态数据成员不属于类的任何一个对象，所以它们并不是在创建类的对象时被定义的。这意味着它们不是由类的构造函数初始化的。而且一般来说，我们不能在类的内部初始化静态成员。相反的，必须在类的外部定义和初始化每个静态成员。和其他对象一样，一个静态数据成员只能定义一次。

类似于全局变量（参见 6.1.1 节，第 184 页），静态数据成员定义在任何函数之外。因此一旦它被定义，就将一直存在于程序的整个生命周期中。

我们定义静态数据成员的方式和在类的外部定义成员函数差不多。我们需要指定对象的类型名，然后是类名、作用域运算符以及成员自己的名字：

```
// 定义并初始化一个静态成员
double Account::interestRate = initRate();
```

这条语句定义了名为 interestRate 的对象，该对象是类 Account 的静态成员，其类型是 double。从类名开始，这条定义语句的剩余部分就都位于类的作用域之内了。因此，我们可以直接使用 initRate 函数。注意，虽然 initRate 是私有的，我们也能用它初始化 interestRate。和其他成员的定义一样，interestRate 的定义也可以访问类的私有成员。

要想确保对象只定义一次，最好的办法是把静态数据成员的定义与其他非内联函数的定义放在同一个文件中。

静态成员的类内初始化

通常情况下，类的静态成员不应该在类的内部初始化。然而，我们可以为静态成员提供 const 整数类型的类内初始值，不过要求静态成员必须是字面值常量类型的
303 constexpr（参见 7.5.6 节，第 267 页）。初始值必须是常量表达式，因为这些成员本身就是常量表达式，所以它们能用在所有适合于常量表达式的地方。例如，我们可以用一个初始化了的静态数据成员指定数组成员的维度：

```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double);
private:
```

```
    static constexpr int period = 30;     // period是常量表达式
    double daily_tbl[period];
};
```

如果某个静态成员的应用场景仅限于编译器可以替换它的值的情况，则一个初始化的const或constexpr static不需要分别定义。相反，如果我们将它用于值不能替换的场景中，则该成员必须有一条定义语句。

例如，如果period的唯一用途就是定义daily_tbl的维度，则不需要在Account外面专门定义period。此时，如果我们忽略了这条定义，那么对程序非常微小的改动也可能造成编译错误，因为程序找不到该成员的定义语句。举个例子，当需要把Account::period传递给一个接受const int&的函数时，必须定义period。

如果在类的内部提供了一个初始值，则成员的定义不能再指定一个初始值了：

```
// 一个不带初始值的静态成员的定义
constexpr int Account::period;          // 初始值在类的定义内提供
```

即使一个常量静态数据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。

静态成员能用于某些场景，而普通成员不能

如我们所见，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。举个例子，静态数据成员可以是不完全类型（参见7.3.3节，第249页）。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，只能声明成它所属类的指针或引用：

```
class Bar {
public:
    // ...
private:
    static Bar mem1;          // 正确：静态成员可以是不完全类型
    Bar *mem2;                // 正确：指针成员可以是不完全类型
    Bar mem3;                 // 错误：数据成员必须是完全类型
};
```

静态成员和普通成员的另外一个区别是我们可以使用静态成员作为默认实参（参见6.5.1节，第211页）：

304

```
class Screen {
public:
    // bkground表示一个在类中稍后定义的静态成员
    Screen& clear(char = bkground);
private:
    static const char bkground;
};
```

非静态数据成员不能作为默认实参，因为它的值本身属于对象的一部分，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，最终将引发错误。

7.6 节练习

练习 7.56：什么是类的静态成员？它有何优点？静态成员与普通成员有何区别？

练习 7.57：编写你自己的 Account 类。

练习 7.58：下面的静态数据成员的声明和定义有错误吗？请解释原因。

```
// example.h
class Example {
public:
    static double rate = 6.5;
    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};
// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

小结

305

类是C++语言中最基本的特性。类允许我们为自己的应用定义新类型，从而使得程序更加简洁且易于修改。

类有两项基本能力：一是数据抽象，即定义数据成员和函数成员的能力；二是封装，即保护类的成员不被随意访问的能力。通过将类的实现细节设为 `private`，我们就能完成类的封装。类可以将其他类或者函数设为友元，这样它们就能访问类的非公有成员了。

类可以定义一种特殊的成员函数：构造函数，其作用是控制初始化对象的方式。构造函数可以重载，构造函数应该使用构造函数初始值列表来初始化所有数据成员。

类还能定义可变或者静态成员。一个可变成员永远都不会是 `const`，即使在 `const` 成员函数内也能修改它的值；一个静态成员可以是函数也可以是数据，静态成员存在于所有对象之外。

术语表

抽象数据类型（abstract data type） 封装（隐藏）了实现细节的数据结构。

访问说明符（access specifier） 包括关键字 `public` 和 `private`。用于定义成员对类的用户可见还是只对类的友元和成员可见。在类中说明符可以出现多次，每个说明符的有效范围从它自身开始，到下一个说明符为止。

聚合类（aggregate class） 只含有公有成员的类，并且没有类内初始值或者构造函数。聚合类的成员可以用花括号括起来的初始值列表进行初始化。

类（class） C++提供的自定义数据类型的机制。类可以包含数据、函数和类型成员。一个类定义一种新的类型和一个新的作用域。

类的声明（class declaration） 首先是关键字 `class`（或者 `struct`），随后是类名以及分号。如果类已经声明而尚未定义，则它是一个不完全类型。

class 关键字（class keyword） 用于定义类的关键字，默认情况下成员是 `private` 的。

类的作用域（class scope） 每个类定义一个作用域。类作用域比其他作用域更加复杂，类中定义的成员函数甚至有可能使用定义语句之后的名字。

常量成员函数（const member function） 一个成员函数，在其中不能修改对象的普通（即既不是 `static` 也不是 `mutable`）数据成员。`const` 成员的 `this` 指针是指向常量的指针，通过区分函数是否是 `const` 可以进行重载。

构造函数（constructor） 用于初始化对象的一种特殊的成员函数。构造函数应该给每个数据成员都赋一个合适的初始值。

构造函数初始值列表（constructor initializer list） 说明一个类的数据成员的初始值，在构造函数体执行之前首先用初始值列表中的值初始化数据成员。未经初始值列表初始化的成员将被默认初始化。

转换构造函数（converting constructor） 可以用一个实参调用的非显式构造函数。这样的函数隐式地将参数类型转换成类类型。

数据抽象（data abstraction） 着重关注类型接口的一种编程技术。数据抽象令程序员可以忽略类型的实现细节，只关注类型执行的操作即可。数据抽象是面向对象编程和泛型编程的基础。

306

默认构造函数（default constructor） 当没有提供任何实参时使用的构造函数。

委托构造函数（delegating constructor） 委托构造函数的初始值列表只有一个入口，指定类的另一个构造函数执行初始化操作。

封装（encapsulation） 分离类的实现与接口，从而隐藏了类的实现细节。在C++语言中，通过把实现部分设为 `private` 完成封装的任务。

显式构造函数（explicit constructor） 可以用一个单独的实参调用但是不能用于隐式转换的构造函数。通过在构造函数的声明之前加上 `explicit` 关键字就可以将其声明成显式构造函数。

前向声明（forward declaration） 对尚未定义的名字的声明，通常用于表示位于类定义之前的类声明。参见“不完全类型”。

友元（friend）类向外部提供其非公有成员访问权限的一种机制。友元的访问权限与成员函数一样。友元可以是类，也可以是函数。

实现（implementation）类的成员（通常是私有的），定义了不希望为使用类类型的代码所用的数据及任何操作。

不完全类型（incomplete type） 已经声明但是尚未定义的类型。不完全类型不能用于定义变量或者类的成员，但是用不完全类型定义指针或者引用是合法的。

接口（interface） 类型提供的（公有）操作。通常情况下，接口不包含数据成员。

成员函数（member function） 类的函数成员。普通的成员函数通过隐式的 `this` 指针与类的对象绑定在一起；静态成员函数不与对象绑定在一起也没有 `this` 指针。成员函数可以重载，此时隐式的 `this` 指针参与函数匹配的过程。

可变数据成员（mutable data member） 这种成员永远不是 `const`，即使它属于 `const` 对象。在 `const` 函数内可以修改可变数据成员。

名字查找（name lookup） 根据名字的使用寻找匹配的声明的过程。

私有成员（private member） 定义在 `private` 访问说明符之后的成员，只能被类的友元或者类的其他成员访问。数据成员以及仅供类本身使用而不作为接口的功能函数一般设为 `private`。

公有成员（public member） 定义在 `public` 访问说明符之后的成员，可以被类的所有用户访问。通常情况下，只有实现类的接口的函数才被设为 `public`。

struct 关键字（struct keyword） 用于定义类的关键字，默认情况下成员是 `public` 的。

合成默认构造函数（synthesized default constructor） 对于没有显式地定义任何构造函数的类，编译器为其创建（合成）的默认构造函数。该构造函数检查类的数据成员，如果提供了类内初始值，就用它执行初始化操作；否则就对数据成员执行默认初始化。

this 指针（this pointer） 是一个隐式的值，作为额外的实参传递给类的每个非静态成员函数。`this` 指针指向代表函数调用者的对象。

= default 一种语法形式，位于类内部默认构造函数声明语句的参数列表之后，要求编译器生成构造函数，而不管类是否已经有了其他构造函数。

第Ⅱ部分
C++标准库

内容

随着 C++版本的一次次修订，标准库也在不断成长。确实，新的 C++标准中有三分之二的文本都用来描述标准库。虽然我们不能深入讨论所有标准库设施，但有些核心库设施是每个 C++程序员都应该熟练掌握的，第二部分将介绍这些内容。

我们首先在第 8 章中介绍基本的 IO 库设施。除了使用标准库读写与控制台窗口相关联的流之外，我们还将学习其他一些库类型，可以帮助我们读写命名文件以及完成到 `string` 对象的内存 IO 操作。

标准库的核心是很多容器类和一族泛型算法，这些设施能帮助我们编写简洁高效的程序。标准库会去关注那些簿记操作的细节，特别是内存管理，这样我们的程序就可以将全部注意力投入到需要求解的问题上。

我们在第 3 章中已经介绍了容器类型 `vector`，在第 9 章中将介绍更多 `vector` 相关的内容，这一章也会涉及其他顺序容器。我们还会介绍更多 `string` 类型所支持的操作，可以将 `string` 看作一种只包含字符元素的特殊容器。`string` 支持很多容器操作，但并不是全部。

第 10 章介绍泛型算法。这类算法通常在顺序容器一定范围内的元素上或其他类型的序列上进行操作。算法库为各种经典算法提供了高效的实现，如排序和搜索算法，还提供了其他一些常用操作。例如，标准库提供了 `copy` 算法，完成一个序列到另一个序列的元素拷贝；还提供了 `find` 算法，实现给定元素的查找，等等。泛型算法的通用性体现在两个层面：可应用于不同类型的序列；对序列中元素的类型限制小，大多数类型都是允许的。

标准库还提供了一些关联容器，第 11 章介绍这部分内容。关联容器中的元素是通过关键字来访问的。关联容器支持很多顺序容器的操作，也定义了一些自己特有的操作。

第 12 章是第二部分的最后一章，这一章介绍动态内存管理相关的一些语言特性和库设施。这一章介绍智能指针的一个标准版本，它是新标准库中最重要的类之一。通过使用智能指针，我们可以大幅度提高使用动态内存的代码的鲁棒性。这一章最后将给出一个较大的例子，使用了第II部分介绍的所有标准库设施。

第 8 章

IO 库

内容

C++语言不直接处理输入输出，而是通过一族定义在标准库中的类型来处理 IO。这些类型支持从设备读取数据、向设备写入数据的 IO 操作，设备可以是文件、控制台窗口等。还有一些类型允许内存 IO，即，从 `string` 读取数据，向 `string` 写入数据。

IO 库定义了读写内置类型值的操作。此外，一些类，如 `string`，通常也会定义类似的 IO 操作，来读写自己的对象。

本章介绍 IO 库的基本内容。后续章节会介绍更多 IO 库的功能：第 14 章将会介绍如何编写自己的输入输出运算符，第 17 章将会介绍如何控制输出格式以及如何对文件进行随机访问。

310 我们的程序已经使用了很多 IO 库设施了。我们在 1.2 节（第 5 页）已经介绍了大部分 IO 库设施：

- istream（输入流）类型，提供输入操作。
- ostream（输出流）类型，提供输出操作。
- cin，一个 istream 对象，从标准输入读取数据。
- cout，一个 ostream 对象，向标准输出写入数据。
- cerr，一个 ostream 对象，通常用于输出程序错误消息，写入到标准错误。
- >>运算符，用来从一个 istream 对象读取输入数据。
- <<运算符，用来向一个 ostream 对象写入输出数据。
- getline 函数（参见 3.3.2 节，第 78 页），从一个给定的 istream 读取一行数据，存入一个给定的 string 对象中。

8.1 IO 类

到目前为止，我们已经使用过的 IO 类型和对象都是操纵 char 数据的。默认情况下，这些对象都是关联到用户的控制台窗口的。当然，我们不能限制实际应用程序仅从控制台窗口进行 IO 操作，应用程序常常需要读写命名文件。而且，使用 IO 操作处理 string 中的字符会很方便。此外，应用程序还可能读写需要宽字符支持的语言。

为了支持这些不同种类的 IO 处理操作，在 istream 和 ostream 之外，标准库还定义了其他一些 IO 类型，我们之前都已经使用过了。表 8.1 列出了这些类型，分别定义在三个独立的头文件中：iostream 定义了用于读写流的基本类型，fstream 定义了读写命名文件的类型，sstream 定义了读写内存 string 对象的类型。

表 8.1：IO 库类型和头文件

头文件	类型
iostream	istream，wistream 从流读取数据 ostream，wostream 向流写入数据 iostream，wiostream 读写流
fstream	ifstream，wifstream 从文件读取数据 ofstream，wofstream 向文件写入数据 fstream，wfstream 读写文件
sstream	istringstream，wistringstream 从 string 读取数据 ostringstream，wostringstream 向 string 写入数据 stringstream，wstringstream 读写 string

311 为了支持使用宽字符的语言，标准库定义了一组类型和对象来操纵 wchar_t 类型的数据（参见 2.1.1 节，第 30 页）。宽字符版本的类型和函数的名字以一个 w 开始。例如，wcin、wcout 和 wcerr 是分别对应 cin、cout 和 cerr 的宽字符版对象。宽字符版本的类型和对象与其对应的普通 char 版本的类型定义在同一个头文件中。例如，头文件 fstream 定义了 ifstream 和 wifstream 类型。

IO 类型间的关系

概念上，设备类型和字符大小都不会影响我们要执行的 IO 操作。例如，我们可以用>>读取数据，而不用管是从一个控制台窗口，一个磁盘文件，还是一个 string 读取。类似的，我们也不用管读取的字符能存入一个 char 对象内，还是需要一个 wchar_t 对象来存储。

标准库使我们能忽略这些不同类型的流之间的差异，这是通过**继承机制**（inheritance）实现的。利用模板（参见 3.3 节，第 87 页），我们可以使用具有继承关系的类，而不必了解继承机制如何工作的细节。我们将在第 15 章和 18.3 节（第 710 页）介绍 C++是如何支持继承机制的。

简单地说，继承机制使我们可以声明一个特定的类继承自另一个类。我们通常可以将一个派生类（继承类）对象当作其基类（所继承的类）对象来使用。

类型 ifstream 和 istringstream 都继承自 istream。因此，我们可以像使用 istream 对象一样来使用 ifstream 和 istringstream 对象。也就是说，我们是如何使用 cin 的，就可以同样地使用这些类型的对象。例如，可以对一个 ifstream 或 istringstream 对象调用 getline，也可以使用 >> 从一个 ifstream 或 istringstream 对象中读取数据。类似的，类型 ofstream 和 ostringstream 都继承自 ostream。因此，我们是如何使用 cout 的，就可以同样地使用这些类型的对象。

本节剩下部分所介绍的标准库流特性都可以无差别地应用于普通流、文件流和 string 流，以及 char 或宽字符流版本。

8.1.1 IO 对象无拷贝或赋值

如我们在 7.1.3 节（第 234 页）所见，我们不能拷贝或对 IO 对象赋值：

```
ofstream out1, out2;
out1 = out2;                        // 错误：不能对流对象赋值
ofstream print(ofstream);           // 错误：不能初始化 ofstream 参数
out2 = print(out2);                 // 错误：不能拷贝流对象
```

由于不能拷贝 IO 对象，因此我们也不能将形参或返回类型设置为流类型（参见 6.2.1 节，第 188 页）。进行 IO 操作的函数通常以引用方式传递和返回流。读写一个 IO 对象会改变其状态，因此传递和返回的引用不能是 const 的。

8.1.2 条件状态

312

IO 操作一个与生俱来的问题是可能发生错误。一些错误是可恢复的，而其他错误则发生在系统深处，已经超出了应用程序可以修正的范围。表 8.2 列出了 IO 类所定义的一些函数和标志，可以帮助我们访问和操纵流的**条件状态**（condition state）。

表 8.2：IO 库条件状态

strm::iostate	*strm* 是一种 IO 类型，在表 8.1（第 278 页）中已列出。iostate 是一种机器相关的类型，提供了表达条件状态的完整功能
strm::badbit	*strm*::badbit 用来指出流已崩溃
strm::failbit	*strm*::failbit 用来指出一个 IO 操作失败了

续表

strm::eofbit	*strm*::eofbit 用来指出流到达了文件结束
strm::goodbit	*strm*::goodbit 用来指出流未处于错误状态。此值保证为零
s.eof()	若流 s 的 eofbit 置位，则返回 true
s.fail()	若流 s 的 failbit 或 badbit 置位，则返回 true
s.bad()	若流 s 的 badbit 置位，则返回 true
s.good()	若流 s 处于有效状态，则返回 true
s.clear()	将流 s 中所有条件状态位复位，将流的状态设置为有效。返回 void
s.clear(flags)	根据给定的 flags 标志位，将流 s 中对应条件状态位复位。flags 的类型为 *strm*::iostate。返回 void
s.setstate(flags)	根据给定的 flags 标志位，将流 s 中对应条件状态位置位。flags 的类型为 *strm*::iostate。返回 void
s.rdstate()	返回流 s 的当前条件状态，返回值类型为 strm::iostate

下面是一个 IO 错误的例子：

```
int ival;
cin >> ival;
```

如果我们在标准输入上键入 Boo，读操作就会失败。代码中的输入运算符期待读取一个 int，但却得到了一个字符 B。这样，cin 会进入错误状态。类似的，如果我们输入一个文件结束标识，cin 也会进入错误状态。

一个流一旦发生错误，其上后续的 IO 操作都会失败。只有当一个流处于无错状态时，我们才可以从它读取数据，向它写入数据。由于流可能处于错误状态，因此代码通常应该在使用一个流之前检查它是否处于良好状态。确定一个流对象的状态的最简单的方法是将它当作一个条件来使用：

```
while (cin >> word)
    // ok: 读操作成功……
```

while 循环检查>>表达式返回的流的状态。如果输入操作成功，流保持有效状态，则条件为真。

查询流的状态

将流作为条件使用，只能告诉我们流是否有效，而无法告诉我们具体发生了什么。有时我们也需要知道流为什么失败。例如，在键入文件结束标识后我们的应对措施，可能与遇到一个 IO 设备错误的处理方式是不同的。

IO 库定义了一个与机器无关的 iostate 类型，它提供了表达流状态的完整功能。这个类型应作为一个位集合来使用，使用方式与我们在 4.8 节中（第 137 页）使用 quiz1 的方式一样。IO 库定义了 4 个 iostate 类型的 constexpr 值（参见 2.4.4 节，第 58 页），表示特定的位模式。这些值用来表示特定类型的 IO 条件，可以与位运算符（参见 4.8 节，第 137 页）一起使用来一次性检测或设置多个标志位。

badbit 表示系统级错误，如不可恢复的读写错误。通常情况下，一旦 badbit 被置位，流就无法再使用了。在发生可恢复错误后，failbit 被置位，如期望读取数值却读出一个字符等错误。这种问题通常是可以修正的，流还可以继续使用。如果到达文件结束位置，eofbit 和 failbit 都会被置位。goodbit 的值为 0，表示流未发生错误。如果 badbit、failbit 和 eofbit 任一个被置位，则检测流状态的条件会失败。