```
string window = screen();     // 调用 screen(ht(), 80,' ')
```

用作默认实参的名字在函数声明所在的作用域内解析，而这些名字的求值过程发生在函数调用时：

```
void f2()
{
    def = '*';                // 改变默认实参的值
    sz wd = 100;              // 隐藏了外层定义的 wd，但是没有改变默认值
    window = screen();        // 调用 screen(ht(), 80,'*')
}
```

我们在函数 f2 内部改变了 def 的值，所以对 screen 的调用将会传递这个更新过的值。另一方面，虽然我们的函数还声明了一个局部变量用于隐藏外层的 wd，但是该局部变量与传递给 screen 的默认实参没有任何关系。

6.5.1 节练习

238

练习 6.40：下面的哪个声明是错误的？为什么？

```
(a) int ff(int a, int b = 0, int c = 0);
(b) char *init(int ht = 24, int wd, char bckgrnd);
```

练习 6.41：下面的哪个调用是非法的？为什么？哪个调用虽然合法但显然与程序员的初衷不符？为什么？

```
char *init(int ht, int wd = 80, char bckgrnd = ' ');
(a) init();        (b) init(24,10);        (c) init(14, '*');
```

练习 6.42：给 make_plural 函数（参见 6.3.2 节，第 201 页）的第二个形参赋予默认实参's'，利用新版本的函数输出单词 success 和 failure 的单数和复数形式。

6.5.2 内联函数和 constexpr 函数

在 6.3.2 节（第 201 页）中我们编写了一个小函数，它的功能是比较两个 string 形参的长度并返回长度较小的 string 的引用。把这种规模较小的操作定义成函数有很多好处，主要包括：

- 阅读和理解 shorterString 函数的调用要比读懂等价的条件表达式容易得多。
- 使用函数可以确保行为的统一，每次相关操作都能保证按照同样的方式进行。
- 如果我们需要修改计算过程，显然修改函数要比先找到等价表达式所有出现的地方再逐一修改更容易。
- 函数可以被其他应用重复利用，省去了程序员重新编写的代价。

然而，使用 shorterString 函数也存在一个潜在的缺点：调用函数一般比求等价表达式的值要慢一些。在大多数机器上，一次函数调用其实包含着一系列工作：调用前要先保存寄存器，并在返回时恢复；可能需要拷贝实参；程序转向一个新的位置继续执行。

内联函数可避免函数调用的开销

将函数指定为**内联函数**（inline），通常就是将它在每个调用点上“内联地”展开。假设我们把 shorterString 函数定义成内联函数，则如下调用

239
```
cout << shorterString(s1, s2) << endl;
```

将在编译过程中展开成类似于下面的形式

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

从而消除了 shorterString 函数的运行时开销。

在 shorterString 函数的返回类型前面加上关键字 inline，这样就可以将它声明成内联函数了：

```
// 内联版本：寻找两个 string 对象中较短的那个
inline const string &
shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

内联说明只是向编译器发出的一个请求，编译器可以选择忽略这个请求。

一般来说，内联机制用于优化规模较小、流程直接、频繁调用的函数。很多编译器都不支持内联递归函数，而且一个 75 行的函数也不大可能在调用点内联地展开。

constexpr 函数

constexpr 函数（constexpr function）是指能用于常量表达式（参见 2.4.4 节，第 58 页）的函数。定义 constexpr 函数的方法与其他函数类似，不过要遵循几项约定：函数的返回类型及所有形参的类型都得是字面值类型（参见 2.4.4 节，第 59 页），而且函数体中必须有且只有一条 return 语句：

C++ 11

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz(); // 正确：foo 是一个常量表达式
```

我们把 new_sz 定义成无参数的 constexpr 函数。因为编译器能在程序编译时验证 new_sz 函数返回的是常量表达式，所以可以用 new_sz 函数初始化 constexpr 类型的变量 foo。

执行该初始化任务时，编译器把对 constexpr 函数的调用替换成其结果值。为了能在编译过程中随时展开，constexpr 函数被隐式地指定为内联函数。

constexpr 函数体内也可以包含其他语句，只要这些语句在运行时不执行任何操作就行。例如，constexpr 函数中可以有空语句、类型别名（参见 2.5.1 节，第 60 页）以及 using 声明。

我们允许 constexpr 函数的返回值并非一个常量：

```
// 如果 arg 是常量表达式，则 scale(arg) 也是常量表达式
constexpr size_t scale(size_t cnt) { return new_sz() * cnt; }
```

当 scale 的实参是常量表达式时，它的返回值也是常量表达式；反之则不然：

240
```
int arr[scale(2)];      // 正确：scale(2) 是常量表达式
int i = 2;              // i 不是常量表达式
int a2[scale(i)];       // 错误：scale(i) 不是常量表达式
```

如上例所示，当我们给 scale 函数传入一个形如字面值 2 的常量表达式时，它的返回类型也是常量表达式。此时，编译器用相应的结果值替换对 scale 函数的调用。

如果我们用一个非常量表达式调用 scale 函数，比如 int 类型的对象 i，则返回值是一个非常量表达式。当把 scale 函数用在需要常量表达式的上下文中时，由编译器负责检查函数的结果是否符合要求。如果结果恰好不是常量表达式，编译器将发出错误信息。

constexpr 函数不一定返回常量表达式。

把内联函数和 constexpr 函数放在头文件内

和其他函数不一样，内联函数和 constexpr 函数可以在程序中多次定义。毕竟，编译器要想展开函数仅有函数声明是不够的，还需要函数的定义。不过，对于某个给定的内联函数或者 constexpr 函数来说，它的多个定义必须完全一致。基于这个原因，内联函数和 constexpr 函数通常定义在头文件中。

6.5.2 节练习

练习 6.43：你会把下面的哪个声明和定义放在头文件中？哪个放在源文件中？为什么？

```
(a) inline bool eq(const BigInt&, const BigInt&) {...}
(b) void putValues(int *arr, int size);
```

练习 6.44：将 6.2.2 节（第 189 页）的 isShorter 函数改写成内联函数。

练习 6.45：回顾在前面的练习中你编写的那些函数，它们应该是内联函数吗？如果是，将它们改写成内联函数；如果不是，说明原因。

练习 6.46：能把 isShorter 函数定义成 constexpr 函数吗？如果能，将它改写成 constexpr 函数；如果不能，说明原因。

6.5.3 调试帮助

C++程序员有时会用到一种类似于头文件保护（参见 2.6.3 节，第 67 页）的技术，以便有选择地执行调试代码。基本思想是，程序可以包含一些用于调试的代码，但是这些代码只在开发程序时使用。当应用程序编写完成准备发布时，要先屏蔽掉调试代码。这种方法用到两项预处理功能：assert 和 NDEBUG。

assert 预处理宏

241

assert 是一种**预处理宏**（preprocessor marco）。所谓预处理宏其实是一个预处理变量，它的行为有点类似于内联函数。assert 宏使用一个表达式作为它的条件：

```
assert(expr);
```

首先对 *expr* 求值，如果表达式为假（即 0），assert 输出信息并终止程序的执行。如果表达式为真（即非 0），assert 什么也不做。

assert 宏定义在 cassert 头文件中。如我们所知，预处理名字由预处理器而非编译器管理（参见 2.3.2 节，第 49 页），因此我们可以直接使用预处理名字而无须提供 using 声明。也就是说，我们应该使用 assert 而不是 std::assert，也不需要为 assert 提供 using 声明。

和预处理变量一样，宏名字在程序内必须唯一。含有 cassert 头文件的程序不能再定义名为 assert 的变量、函数或者其他实体。在实际编程过程中，即使我们没有包含

cassert 头文件，也最好不要为了其他目的使用 assert。很多头文件都包含了 cassert，这就意味着即使你没有直接包含 cassert，它也很有可能通过其他途径包含在你的程序中。

assert 宏常用于检查“不能发生”的条件。例如，一个对输入文本进行操作的程序可能要求所有给定单词的长度都大于某个阈值。此时，程序可以包含一条如下所示的语句：

```
assert(word.size() > threshold);
```

NDEBUG 预处理变量

assert 的行为依赖于一个名为 NDEBUG 的预处理变量的状态。如果定义了 NDEBUG，则 assert 什么也不做。默认状态下没有定义 NDEBUG，此时 assert 将执行运行时检查。

我们可以使用一个#define 语句定义 NDEBUG，从而关闭调试状态。同时，很多编译器都提供了一个命令行选项使我们可以定义预处理变量：

```
$ CC -D NDEBUG main.C # use /D with the Microsoft compiler
```

这条命令的作用等价于在 main.c 文件的一开始写#define NDEBUG。

定义 NDEBUG 能避免检查各种条件所需的运行时开销，当然此时根本就不会执行运行时检查。因此，assert 应该仅用于验证那些确实不可能发生的事情。我们可以把 assert 当成调试程序的一种辅助手段，但是不能用它替代真正的运行时逻辑检查，也不能替代程序本身应该包含的错误检查。

除了用于 assert 外，也可以使用 NDEBUG 编写自己的条件调试代码。如果 NDEBUG 未定义，将执行#ifndef 和#endif 之间的代码；如果定义了 NDEBUG，这些代码将被忽略掉：

242

```
void print(const int ia[], size_t size)
{
#ifndef NDEBUG
     // __func__ 是编译器定义的一个局部静态变量，用于存放函数的名字
     cerr << __func__ << ": array size is " << size << endl;
#endif
// ...
```

在这段代码中，我们使用变量__func__输出当前调试的函数的名字。编译器为每个函数都定义了__func__，它是 const char 的一个静态数组，用于存放函数的名字。

除了 C++编译器定义的__func__之外，预处理器还定义了另外 4 个对于程序调试很有用的名字：

__FILE__ 存放文件名的字符串字面值。

__LINE__ 存放当前行号的整型字面值。

__TIME__ 存放文件编译时间的字符串字面值。

__DATE__ 存放文件编译日期的字符串字面值。

可以使用这些常量在错误消息中提供更多信息：

```
if (word.size() < threshold)
     cerr << "Error: " << __FILE__
          << " : in function " << __func__
```

```
        << " at line " << __LINE__ << endl
        << "        Compiled on " << __DATE__
        << " at " << __TIME__ << endl
        << "        Word read was \"" << word
        << "\": Length too short" << endl;
```

如果我们给程序提供了一个长度小于 threshold 的 string 对象，将得到下面的错误消息：

```
Error:wdebug.cc : in function main at line 27
      Compiled on Jul 11 2012 at 20:50:03
      Word read was "foo": Length too short
```

6.5.3 节练习

练习 6.47：改写 6.3.2 节（第 205 页）练习中使用递归输出 vector 内容的程序，使其有条件地输出与执行过程有关的信息。例如，每次调用时输出 vector 对象的大小。分别在打开和关闭调试器的情况下编译并执行这个程序。

练习 6.48：说明下面这个循环的含义，它对 assert 的使用合理吗？

```
string s;
while (cin >> s && s != sought) { } // 空函数体
assert(cin);
```

6.6 函数匹配

在大多数情况下，我们容易确定某次调用应该选用哪个重载函数。然而，当几个重载函数的形参数量相等以及某些形参的类型可以由其他类型转换得来时，这项工作就不那么容易了。以下面这组函数及其调用为例：

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);      // 调用 void f(double, double)
```

确定候选函数和可行函数

243

函数匹配的第一步是选定本次调用对应的重载函数集，集合中的函数称为**候选函数**（candidate function）。候选函数具备两个特征：一是与被调用的函数同名，二是其声明在调用点可见。在这个例子中，有 4 个名为 f 的候选函数。

第二步考察本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些新选出的函数称为**可行函数**（viable function）。可行函数也有两个特征：一是其形参数量与本次调用提供的实参数量相等，二是每个实参的类型与对应的形参类型相同，或者能转换成形参的类型。

我们能根据实参的数量从候选函数中排除掉两个。不使用形参的函数和使用两个 int 形参的函数显然都不适合本次调用，这是因为我们的调用只提供了一个实参，而它们分别有 0 个和两个形参。

使用一个 int 形参的函数和使用两个 double 形参的函数是可行的，它们都能用一

个实参调用。其中最后那个函数本应该接受两个 double 值，但是因为它含有一个默认实参，所以只用一个实参也能调用它。

如果函数含有默认实参（参见 6.5.1 节，第 211 页），则我们在调用该函数时传入的实参数量可能少于它实际使用的实参数量。

在使用实参数量初步判别了候选函数后，接下来考察实参的类型是否与形参匹配。和一般的函数调用类似，实参与形参匹配的含义可能是它们具有相同的类型，也可能是实参类型和形参类型满足转换规则。在上面的例子中，剩下的两个函数都是可行的：

- f(int) 是可行的，因为实参类型 double 能转换成形参类型 int。
- f(double, double) 是可行的，因为它的第二个形参提供了默认值，而第一个形参的类型正好是 double，与函数使用的实参类型完全一致。

244

如果没找到可行函数，编译器将报告无匹配函数的错误。

寻找最佳匹配（如果有的话）

函数匹配的第三步是从可行函数中选择与本次调用最匹配的函数。在这一过程中，逐一检查函数调用提供的实参，寻找形参类型与实参类型最匹配的那个可行函数。下一节将介绍“最匹配”的细节，它的基本思想是，实参类型与形参类型越接近，它们匹配得越好。

在我们的例子中，调用只提供了一个（显式的）实参，它的类型是 double。如果调用 f(int)，实参将不得不从 double 转换成 int。另一个可行函数 f(double, double) 则与实参精确匹配。精确匹配比需要类型转换的匹配更好，因此，编译器把 f(5.6) 解析成对含有两个 double 形参的函数的调用，并使用默认值填补我们未提供的第二个实参。

含有多个形参的函数匹配

当实参的数量有两个或更多时，函数匹配就比较复杂了。对于前面那些名为 f 的函数，我们来分析如下的调用会发生什么情况：

```
(42, 2.56);
```

选择可行函数的方法和只有一个实参时一样，编译器选择那些形参数量满足要求且实参类型和形参类型能够匹配的函数。此例中，可行函数包括 f(int, int) 和 f(double, double)。接下来，编译器依次检查每个实参以确定哪个函数是最佳匹配。如果有且只有一个函数满足下列条件，则匹配成功：

- 该函数每个实参的匹配都不劣于其他可行函数需要的匹配。
- 至少有一个实参的匹配优于其他可行函数提供的匹配。

如果在检查了所有实参之后没有任何一个函数脱颖而出，则该调用是错误的。编译器将报告二义性调用的信息。

在上面的调用中，只考虑第一个实参时我们发现函数 f(int, int) 能精确匹配；要想匹配第二个函数，int 类型的实参必须转换成 double 类型。显然需要内置类型转换的匹配劣于精确匹配，因此仅就第一个实参来说，f(int, int) 比 f(double, double) 更好。

接着考虑第二个实参 2.56，此时 f(double, double) 是精确匹配；要想调用 f(int, int) 必须将 2.56 从 double 类型转换成 int 类型。因此仅就第二个实参来说，f(double, double) 更好。 245

编译器最终将因为这个调用具有二义性而拒绝其请求：因为每个可行函数各自在一个实参上实现了更好的匹配，从整体上无法判断孰优孰劣。看起来我们似乎可以通过强制类型转换（参见 4.11.3 节，第 144 页）其中的一个实参来实现函数的匹配，但是在设计良好的系统中，不应该对实参进行强制类型转换。

调用重载函数时应尽量避免强制类型转换。如果在实际应用中确实需要强制类型转换，则说明我们设计的形参集合不合理。

6.6 节练习

练习 6.49：什么是候选函数？什么是可行函数？

练习 6.50：已知有第 217 页对函数 f 的声明，对于下面的每一个调用列出可行函数。其中哪个函数是最佳匹配？如果调用不合法，是因为没有可匹配的函数还是因为调用具有二义性？

(a) f(2.56, 42) (b) f(42) (c) f(42, 0) (d) f(2.56, 3.14)

练习 6.51：编写函数 f 的 4 个版本，令其各输出一条可以区分的消息。验证上一个练习的答案，如果你回答错了，反复研究本节的内容直到你弄清自己错在何处。

6.6.1 实参类型转换

为了确定最佳匹配，编译器将实参类型到形参类型的转换划分成几个等级，具体排序如下所示：

1. 精确匹配，包括以下情况：
   - 实参类型和形参类型相同。
   - 实参从数组类型或函数类型转换成对应的指针类型（参见 6.7 节，第 221 页，将介绍函数指针）。
   - 向实参添加顶层 const 或者从实参中删除顶层 const。
2. 通过 const 转换实现的匹配（参见 4.11.2 节，第 143 页）。
3. 通过类型提升实现的匹配（参见 4.11.1 节，第 142 页）。
4. 通过算术类型转换（参见 4.11.1 节，第 142 页）或指针转换（参见 4.11.2 节，第 143 页）实现的匹配。
5. 通过类类型转换实现的匹配（参见 14.9 节，第 514 页，将详细介绍这种转换）。

需要类型提升和算术类型转换的匹配

内置类型的提升和转换可能在函数匹配时产生意想不到的结果，但幸运的是，在设计良好的系统中函数很少会含有与下面例子类似的形参。 246

分析函数调用前，我们应该知道小整型一般都会提升到 int 类型或更大的整数类型。

假设有两个函数，一个接受 int、另一个接受 short，则只有当调用提供的是 short 类型的值时才会选择 short 版本的函数。有时候，即使实参是一个很小的整数值，也会直接将它提升成 int 类型；此时使用 short 版本反而会导致类型转换：

```
void ff(int);
void ff(short);
ff('a');                // char 提升成 int；调用 f(int)
```

所有算术类型转换的级别都一样。例如，从 int 向 unsigned int 的转换并不比从 int 向 double 的转换级别高。举个具体点的例子，考虑

```
void manip(long);
void manip(float);
manip(3.14);            // 错误：二义性调用
```

字面值 3.14 的类型是 double，它既能转换成 long 也能转换成 float。因为存在两种可能的算数类型转换，所以该调用具有二义性。

函数匹配和 const 实参

如果重载函数的区别在于它们的引用类型的形参是否引用了 const，或者指针类型的形参是否指向 const，则当调用发生时编译器通过实参是否是常量来决定选择哪个函数：

```
Record lookup(Account&);            // 函数的参数是 Account 的引用
Record lookup(const Account&);      // 函数的参数是一个常量引用
const Account a;
Account b;

lookup(a);                          // 调用 lookup(const Account&)
lookup(b);                          // 调用 lookup(Account&)
```

在第一个调用中，我们传入的是 const 对象 a。因为不能把普通引用绑定到 const 对象上，所以此例中唯一可行的函数是以常量引用作为形参的那个函数，并且调用该函数与实参 a 精确匹配。

在第二个调用中，我们传入的是非常量对象 b。对于这个调用来说，两个函数都是可行的，因为我们既可以使用 b 初始化常量引用也可以用它初始化非常量引用。然而，用非常量对象初始化常量引用需要类型转换，接受非常量形参的版本则与 b 精确匹配。因此，应该选用非常量版本的函数。

247 指针类型的形参也类似。如果两个函数的唯一区别是它的指针形参指向常量或非常量，则编译器能通过实参是否是常量决定选用哪个函数：如果实参是指向常量的指针，调用形参是 const* 的函数；如果实参是指向非常量的指针，调用形参是普通指针的函数。

6.6.1 节练习

练习 6.52： 已知有如下声明，

```
void manip(int, int);
double dobj;
```

请指出下列调用中每个类型转换的等级（参见 6.6.1 节，第 219 页）。

(a) `manip('a', 'z');`　　(b) `manip(55.4, dobj);`

练习 6.53： 说明下列每组声明中的第二条语句会产生什么影响，并指出哪些不合法（如

果有的话)。

```
(a) int calc(int&, int&);
    int calc(const int&, const int&);
(b) int calc(char*, char*);
    int calc(const char*, const char*);
(c) int calc(char*, char*);
    int calc(char* const, char* const);
```

6.7 函数指针

函数指针指向的是函数而非对象。和其他指针一样，函数指针指向某种特定类型。函数的类型由它的返回类型和形参类型共同决定，与函数名无关。例如：

```
// 比较两个 string 对象的长度
bool lengthCompare(const string &, const string &);
```

该函数的类型是 bool(const string&, const string&)。要想声明一个可以指向该函数的指针，只需要用指针替换函数名即可：

```
// pf 指向一个函数，该函数的参数是两个 const string 的引用，返回值是 bool 类型
bool (*pf)(const string &, const string &); // 未初始化
```

从我们声明的名字开始观察，pf 前面有个*，因此 pf 是指针；右侧是形参列表，表示 pf 指向的是函数；再观察左侧，发现函数的返回类型是布尔值。因此，pf 就是一个指向函数的指针，其中该函数的参数是两个 const string 的引用，返回值是 bool 类型。

*pf 两端的括号必不可少。如果不写这对括号，则 pf 是一个返回值为 bool 指针的函数：

```
// 声明一个名为 pf 的函数，该函数返回 bool*
bool *pf(const string &, const string &);
```

248

使用函数指针

当我们把函数名作为一个值使用时，该函数自动地转换成指针。例如，按照如下形式我们可以将 lengthCompare 的地址赋给 pf：

```
pf = lengthCompare;          // pf 指向名为 lengthCompare 的函数
pf = &lengthCompare;         // 等价的赋值语句：取地址符是可选的
```

此外，我们还能直接使用指向函数的指针调用该函数，无须提前解引用指针：

```
bool b1 = pf("hello", "goodbye");          // 调用 lengthCompare 函数
bool b2 = (*pf)("hello", "goodbye");       // 一个等价的调用
bool b3 = lengthCompare("hello", "goodbye"); // 另一个等价的调用
```

在指向不同函数类型的指针间不存在转换规则。但是和往常一样，我们可以为函数指针赋一个 nullptr（参见 2.3.2 节，第 48 页）或者值为 0 的整型常量表达式，表示该指针没有指向任何一个函数：

```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(const char*, const char*);
pf = 0;                  // 正确：pf 不指向任何函数
pf = sumLength;          // 错误：返回类型不匹配
```

```
pf = cstringCompare;      // 错误：形参类型不匹配
pf = lengthCompare;       // 正确：函数和指针的类型精确匹配
```

重载函数的指针

当我们使用重载函数时，上下文必须清晰地界定到底应该选用哪个函数。如果定义了指向重载函数的指针

```
void ff(int*);
void ff(unsigned int);

void (*pf1)(unsigned int) = ff;  // pf1 指向 ff(unsigned)
```

编译器通过指针类型决定选用哪个函数，指针类型必须与重载函数中的某一个精确匹配

```
void (*pf2)(int) = ff;          // 错误：没有任何一个 ff 与该形参列表匹配
double (*pf3)(int*) = ff;       // 错误：ff 和 pf3 的返回类型不匹配
```

249 ### 函数指针形参

和数组类似（参见6.2.4节，第193页），虽然不能定义函数类型的形参，但是形参可以是指向函数的指针。此时，形参看起来是函数类型，实际上却是当成指针使用：

```
// 第三个形参是函数类型，它会自动地转换成指向函数的指针
void useBigger(const string &s1, const string &s2,
               bool pf(const string &, const string &));
// 等价的声明：显式地将形参定义成指向函数的指针
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string &, const string &));
```

我们可以直接把函数作为实参使用，此时它会自动转换成指针：

```
// 自动将函数 lengthCompare 转换成指向该函数的指针
useBigger(s1, s2, lengthCompare);
```

正如useBigger的声明语句所示，直接使用函数指针类型显得冗长而烦琐。类型别名（参见2.5.1节，第60页）和decltype（参见2.5.3节，第62页）能让我们简化使用了函数指针的代码：

```
// Func 和 Func2 是函数类型
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2;          // 等价的类型
// FuncP 和 FuncP2 是指向函数的指针
typedef bool(*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2;        // 等价的类型
```

我们使用typedef定义自己的类型。Func和Func2是函数类型，而FuncP和FuncP2是指针类型。需要注意的是，decltype返回函数类型，此时不会将函数类型自动转换成指针类型。因为decltype的结果是函数类型，所以只有在结果前面加上*才能得到指针。可以使用如下的形式重新声明useBigger：

```
// useBigger 的等价声明，其中使用了类型别名
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP2);
```

这两个声明语句声明的是同一个函数，在第一条语句中，编译器自动地将 Func 表示的函数类型转换成指针。

返回指向函数的指针

和数组类似（参见 6.3.3 节，第 205 页），虽然不能返回一个函数，但是能返回指向函数类型的指针。然而，我们必须把返回类型写成指针形式，编译器不会自动地将函数返回类型当成对应的指针类型处理。与往常一样，要想声明一个返回函数指针的函数，最简单的办法是使用类型别名：

```
using F = int(int*, int);          // F 是函数类型，不是指针
using PF = int(*)(int*, int);      // PF 是指针类型
```

其中我们使用类型别名（参见 2.5.1 节，第 60 页）将 F 定义成函数类型，将 PF 定义成指向函数类型的指针。必须时刻注意的是，和函数类型的形参不一样，返回类型不会自动地转换成指针。我们必须显式地将返回类型指定为指针：

250

```
PF f1(int);          // 正确：PF 是指向函数的指针，f1 返回指向函数的指针
F f1(int);           // 错误：F 是函数类型，f1 不能返回一个函数
F *f1(int);          // 正确：显式地指定返回类型是指向函数的指针
```

当然，我们也能用下面的形式直接声明 f1：

```
int (*f1(int))(int*, int);
```

按照由内向外的顺序阅读这条声明语句：我们看到 f1 有形参列表，所以 f1 是个函数；f1 前面有*，所以 f1 返回一个指针；进一步观察发现，指针的类型本身也包含形参列表，因此指针指向函数，该函数的返回类型是 int。

出于完整性的考虑，有必要提醒读者我们还可以使用尾置返回类型的方式（参见 6.3.3 节，第 206 页）声明一个返回函数指针的函数：

```
auto f1(int) -> int (*)(int*, int);
```

将 auto 和 decltype 用于函数指针类型

如果我们明确知道返回的函数是哪一个，就能使用 decltype 简化书写函数指针返回类型的过程。例如假定有两个函数，它们的返回类型都是 string::size_type，并且各有两个 const string&类型的形参，此时我们可以编写第三个函数，它接受一个 string 类型的参数，返回一个指针，该指针指向前两个函数中的一个：

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
// 根据其形参的取值，getFcn 函数返回指向 sumLength 或者 largerLength 的指针
decltype(sumLength) *getFcn(const string &);
```

声明 getFcn 唯一需要注意的地方是，牢记当我们将 decltype 作用于某个函数时，它返回函数类型而非指针类型。因此，我们显式地加上*以表明我们需要返回指针，而非函数本身。

6.7 节练习

练习 6.54：编写函数的声明，令其接受两个 int 形参并且返回类型也是 int；然后声明一个 vector 对象，令其元素是指向该函数的指针。

练习 6.55：编写 4 个函数，分别对两个 int 值执行加、减、乘、除运算；在上一题创建的 vector 对象中保存指向这些函数的指针。

练习 6.56：调用上述 vector 对象中的每个元素并输出其结果。

小结

251

函数是命名了的计算单元，它对程序（哪怕是不大的程序）的结构化至关重要。每个函数都包含返回类型、名字、（可能为空的）形参列表以及函数体。函数体是一个块，当函数被调用的时候执行该块的内容。此时，传递给函数的实参类型必须与对应的形参类型相容。

在C++语言中，函数可以被重载：同一个名字可用于定义多个函数，只要这些函数的形参数量或形参类型不同就行。根据调用时所使用的实参，编译器可以自动地选定被调用的函数。从一组重载函数中选取最佳函数的过程称为函数匹配。

术语表

二义性调用（ambiguous call） 是一种编译时发生的错误，造成二义性调用的原因是在函数匹配时两个或多个函数提供的匹配一样好，编译器找不到唯一的最佳匹配。

实参（argument） 函数调用时提供的值，用于初始化函数的形参。

Assert 是一个预处理宏，作用于一条表示条件的表达式。当未定义预处理变量NDEBUG时，`assert`对条件求值。如果条件为假，输出一条错误信息并终止当前程序的执行。

自动对象（automatic object） 仅存在于函数执行过程中的对象。当程序的控制流经过此类对象的定义语句时，创建该对象；当到达了定义所在的块的末尾时，销毁该对象。

最佳匹配（best match） 从一组重载函数中为调用选出的一个函数。如果存在最佳匹配，则选出的函数与其他所有可行函数相比，至少在一个实参上是更优的匹配，同时在其他实参的匹配上不会更差。

传引用调用（call by reference） 参见引用传递。

传值调用（call by value） 参见值传递。

候选函数（candidate function） 解析某次函数调用时考虑的一组函数。候选函数的名字应该与函数调用使用的名字一致，并且在调用点候选函数的声明在作用域之内。

constexpr 可以返回常量表达式的函数，一个`constexpr`函数被隐式地声明成内联函数。

默认实参（default argument） 当调用缺少了某个实参时，为该实参指定的默认值。

可执行文件（executable file） 是操作系统能够执行的文件，包含着与程序有关的代码。

函数（function）可调用的计算单元。

函数体（function body） 是一个块，用于定义函数所执行的操作。

函数匹配（function matching） 编译器解析重载函数调用的过程，在此过程中，实参与每个重载函数的形参列表逐一比较。

函数原型（function prototype） 函数的声明，包含函数名字、返回类型和形参类型。要想调用某函数，在调用点之前必须声明该函数的原型。

隐藏名字（hidden name） 某个作用域内声明的名字会隐藏掉外层作用域中声明的同名实体。

initializer_list 是一个标准类，表示的是一组花括号包围的类型相同的对象，对象之间以逗号隔开。

内联函数（inline function） 请求编译器在可能的情况下在调用点展开函数。内联函数可以避免常见的函数调用开销。 252

链接（link）是一个编译过程，负责把若干

对象文件链接起来形成可执行程序。

局部静态对象（local static object） 它的值在函数调用结束后仍然存在。在第一次使用局部静态对象前创建并初始化它，当程序结束时局部静态对象才被销毁。

局部变量（local variable） 定义在块中的变量。

无匹配（no match）是一种编译时发生的错误，原因是在函数匹配过程中所有函数的形参都不能与调用提供的实参匹配。

对象代码（object code） 编译器将我们的源代码转换成对象代码格式。

对象文件（object file） 编译器根据给定的源文件生成的保存对象代码的文件。一个或多个对象文件经过链接生成可执行文件。

对象生命周期（object lifetime） 每个对象都有相应的生命周期。块内定义的非静态对象的生命周期从它的定义开始，到定义所在的块末尾为止。程序启动后创建全局对象，程序控制流经过局部静态对象的定义时创建该局部静态对象；当 `main` 函数结束时销毁全局对象和局部静态对象。

重载确定（overload resolution） 参见函数匹配。

重载函数（overloaded function） 函数名与其他函数相同的函数。多个重载函数必须在形参数量或形参类型上有所区别。

形参（parameter） 在函数的形参列表中声明的局部变量。用实参初始化形参。

引用传递（pass by reference） 描述如何将实参传递给引用类型的形参。引用形参和其他形式的引用工作机理类似，形参被绑定到相应的实参上。

值传递（pass by value） 描述如何将实参传递给非引用类型的形参。非引用类型的形参实际上是相应实参值的一个副本。

预处理宏（preprocessor macro） 类似于内联函数的一种预处理功能。除了 `assert` 之外，现代 C++程序很少再使用预处理宏了。

递归循环（recursion loop） 描述某个递归函数没有终止条件，因而不断调用自身直至耗尽程序栈空间的过程。

递归函数（recursive function） 直接或间接调用自身的函数。

返回类型（return type） 是函数声明的一部分，用于指定函数返回值的类型。

分离式编译（separate compilation） 把一个程序分割成多个独立源文件的能力。

尾置返回类型（trailing return type） 在参数列表后面指定的返回类型。

可行函数（viable function） 是候选函数的子集。可行函数能匹配本次调用，它的形参数量与调用提供的实参数量相等，并且每个实参类型都能转换成相应的形参类型。

()运算符（() operator） 调用运算符，用于执行某函数。括号前面是函数名或函数指针，括号内是以逗号隔开的实参列表（可能为空）。

第 7 章

类

内容

在 C++语言中，我们使用类定义自己的数据类型。通过定义新的类型来反映待解决问题中的各种概念，可以使我们更容易编写、调试和修改程序。

本章是第 2 章关于类的话题的延续，主要关注数据抽象的重要性。数据抽象能帮助我们将对象的具体实现与对象所能执行的操作分离开来。第 13 章将讨论如何控制对象拷贝、移动、赋值和销毁等行为，在第 14 章中我们将学习如何自定义运算符。

254 类的基本思想是**数据抽象**（data abstraction）和**封装**（encapsulation）。数据抽象是一种依赖于**接口**（interface）和**实现**（implementation）分离的编程（以及设计）技术。类的接口包括用户所能执行的操作；类的实现则包括类的数据成员、负责接口实现的函数体以及定义类所需的各种私有函数。

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而无法访问实现部分。

类要想实现数据抽象和封装，需要首先定义一个**抽象数据类型**（abstract data type）。在抽象数据类型中，由类的设计者负责考虑类的实现过程；使用该类的程序员则只需要抽象地思考类型做了什么，而无须了解类型的工作细节。

7.1 定义抽象数据类型

在第 1 章中使用的 `Sales_item` 类是一个抽象数据类型，我们通过它的接口（例如 1.5.1 节(第17页)描述的操作)来使用一个`Sales_item`对象。我们不能访问`Sales_item`对象的数据成员，事实上，我们甚至根本不知道这个类有哪些数据成员。

与之相反，`Sales_data` 类（参见 2.6.1 节，第 64 页）不是一个抽象数据类型。它允许类的用户直接访问它的数据成员，并且要求由用户来编写操作。要想把 `Sales_data` 变成抽象数据类型，我们需要定义一些操作以供类的用户使用。一旦 `Sales_data` 定义了它自己的操作，我们就可以封装（隐藏）它的数据成员了。

7.1.1 设计 Sales_data 类

我们的最终目的是令 `Sales_data` 支持与 `Sales_item` 类完全一样的操作集合。`Sales_item` 类有一个名为 `isbn` 的**成员函数**（member function）(参见 1.5.2 节，第 20 页)，并且支持+、=、+=、<<和>>运算符。

我们将在第 14 章学习如何自定义运算符。现在，我们先为这些运算定义普通（命名的）函数形式。由于 14.1 节（第 490 页）将要解释的原因，执行加法和 IO 的函数不作为 `Sales_data` 的成员，相反的，我们将其定义成普通函数；执行复合赋值运算的函数是成员函数。`Sales_data` 类无须专门定义赋值运算，其原因将在 7.1.5 节（第 239 页）介绍。

综上所述，`Sales_data` 的接口应该包含以下操作：

- 一个 `isbn` 成员函数，用于返回对象的 ISBN 编号
- 一个 `combine` 成员函数，用于将一个 `Sales_data` 对象加到另一个对象上
- 一个名为 `add` 的函数，执行两个 `Sales_data` 对象的加法
- 一个 `read` 函数，将数据从 `istream` 读入到 `Sales_data` 对象中
- 一个 `print` 函数，将 `Sales_data` 对象的值输出到 `ostream`

255 **关键概念：不同的编程角色**

程序员们常把运行其程序的人称作用户（user）。类似的，类的设计者也是为其用户设计并实现一个类的人；显然，类的用户是程序员，而非应用程序的最终使用者。

当我们提及“用户”一词时，不同的语境决定了不同的含义。如果我们说用户代码或者 `Sales_data` 类的用户，指的是使用类的程序员；如果我们说书店应用程序的用

户，则意指运行该应用程序的书店经理。

C++程序员们无须刻意区分应用程序的用户以及类的用户。

在一些简单的应用程序中，类的用户和类的设计者常常是同一个人。尽管如此，还是最好把角色区分开来。当我们设计类的接口时，应该考虑如何才能使得类易于使用；而当我们使用类时，不应该顾及类的实现机理。

要想开发一款成功的应用程序，其作者必须充分了解并实现用户的需求。同样，优秀的类设计者也应该密切关注那些有可能使用该类的程序员的需求。作为一个设计良好的类，既要有直观且易于使用的接口，也必须具备高效的实现过程。

使用改进的 Sales_data 类

在考虑如何实现我们的类之前，首先来看看应该如何使用上面这些接口函数。举个例子，我们使用这些函数编写 1.6 节（第 21 页）书店程序的另外一个版本，其中不再使用 Sales_item 对象，而是使用 Sales_data 对象：

```
Sales_data total;                    // 保存当前求和结果的变量
if (read(cin, total)) {              // 读入第一笔交易
    Sales_data trans;                // 保存下一条交易数据的变量
    while(read(cin, trans)) {        // 读入剩余的交易
        if (total.isbn() == trans.isbn())  // 检查 isbn
            total.combine(trans);      // 更新变量 total 当前的值
        else {
            print(cout, total) << endl;     // 输出结果
            total = trans;                  // 处理下一本书
        }
    }
    print(cout, total) << endl;             // 输出最后一条交易
} else {                                    // 没有输入任何信息
    cerr << "No data?!" << endl;            // 通知用户
}
```

一开始我们定义了一个 Sales_data 对象用于保存实时的汇总信息。在 if 条件内部，调用 read 函数将第一条交易读入到 total 中，这里的条件部分与之前我们使用>>运算符的效果是一样的。read 函数返回它的流参数，而条件部分负责检查这个返回值（参见 4.11.2 节，第 144 页），如果 read 函数失败，程序将直接跳转到 else 语句并输出一条错误信息。

256

如果检测到读入了数据，我们定义变量 trans 用于存放每一条交易。while 语句的条件部分同样是检查 read 函数的返回值，只要输入操作成功，条件就被满足，意味着我们可以处理一条新的交易。

在 while 循环内部，我们分别调用 total 和 trans 的 isbn 成员以比较它们的 ISBN 编号。如果 total 和 trans 指示的是同一本书，我们调用 combine 函数将 trans 的内容添加到 total 表示的实时汇总结果中去。如果 trans 指示的是一本新书，我们调用 print 函数将之前一本书的汇总信息输出出来。因为 print 返回的是它的流参数的引用，所以我们可以把 print 的返回值作为<<运算符的左侧运算对象。通过这种方式，我们输出 print 函数的处理结果，然后转到下一行。接下来，把 trans 赋给 total，从而为接着处理文件中下一本书的记录做好了准备。

处理完所有输入数据后，使用 while 循环之后的 print 语句将最后一条交易的信息输出出来。

7.1.1 节练习

练习 7.1：使用 2.6.1 节练习定义的 Sales_data 类为 1.6 节（第 21 页）的交易处理程序编写一个新版本。

7.1.2 定义改进的 Sales_data 类

改进之后的类的数据成员将与 2.6.1 节（第 64 页）定义的版本保持一致，它们包括：bookNo，string 类型，表示 ISBN 编号；units_sold，unsigned 类型，表示某本书的销量；以及 revenue，double 类型，表示这本书的总销售收入。

如前所述，我们的类将包含两个成员函数：combine 和 isbn。此外，我们还将赋予 Sales_data 另一个成员函数用于返回售出书籍的平均价格，这个函数被命名为 avg_price。因为 avg_price 的目的并非通用，所以它应该属于类的实现的一部分，而非接口的一部分。

定义（参见 6.1 节，第 182 页）和声明（参见 6.1.2 节，第 186 页）成员函数的方式与普通函数差不多。成员函数的声明必须在类的内部，它的定义则既可以在类的内部也可以在类的外部。作为接口组成部分的非成员函数，例如 add、read 和 print 等，它们的定义和声明都在类的外部。

由此可知，改进的 Sales_data 类应该如下所示：

```
struct Sales_data {
    // 新成员：关于 Sales_data 对象的操作
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
257 // 数据成员和 2.6.1 节（第 64 页）相比没有改变
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// Sales_data 的非成员接口函数
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
```

定义在类内部的函数是隐式的 inline 函数（参见 6.5.2 节，第 214 页）。

定义成员函数

尽管所有成员都必须在类的内部声明，但是成员函数体可以定义在类内也可以定义在类外。对于 Sales_data 类来说，isbn 函数定义在了类内，而 combine 和 avg_price 定义在了类外。

我们首先介绍 isbn 函数，它的参数列表为空，返回值是一个 string 对象：

```
std::string isbn() const { return bookNo; }
```

和其他函数一样，成员函数体也是一个块。在此例中，块只有一条 return 语句，用于返回 Sales_data 对象的 bookNo 数据成员。关于 isbn 函数一件有意思的事情是：它是如何获得 bookNo 成员所依赖的对象的呢？

引入 this

让我们再一次观察对 isbn 成员函数的调用：

```
total.isbn()
```

在这里，我们使用了点运算符（参见 4.6 节，第 133 页）来访问 total 对象的 isbn 成员，然后调用它。

7.6 节（第 268 页）将介绍一种例外的形式，当我们调用成员函数时，实际上是在替某个对象调用它。如果 isbn 指向 Sales_data 的成员（例如 bookNo），则它隐式地指向调用该函数的对象的成员。在上面所示的调用中，当 isbn 返回 bookNo 时，实际上它隐式地返回 total.bookNo。

成员函数通过一个名为 **this** 的额外的隐式参数来访问调用它的那个对象。当我们调用一个成员函数时，用请求该函数的对象地址初始化 this。例如，如果调用

```
total.isbn()
```

则编译器负责把 total 的地址传递给 isbn 的隐式形参 this，可以等价地认为编译器将该调用重写成了如下的形式：

```
// 伪代码，用于说明调用成员函数的实际执行过程
Sales_data::isbn(&total)
```

258

其中，调用 Sales_data 的 isbn 成员时传入了 total 的地址。

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 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。

对于我们来说，this 形参是隐式定义的。实际上，任何自定义名为 this 的参数或变量的行为都是非法的。我们可以在成员函数体内部使用 this，因此尽管没有必要，但我们还是能把 isbn 定义成如下的形式：

```
std::string isbn() const { return this->bookNo; }
```

因为 this 的目的总是指向“这个”对象，所以 this 是一个常量指针（参见 2.4.2 节，第 56 页），我们不允许改变 this 中保存的地址。

引入 const 成员函数

isbn 函数的另一个关键之处是紧随参数列表之后的 const 关键字，这里，const 的作用是修改隐式 this 指针的类型。

默认情况下，this 的类型是指向类类型非常量版本的常量指针。例如在 Sales_data 成员函数中，this 的类型是 Sales_data *const。尽管 this 是隐式的，但它仍然需要遵循初始化规则，意味着（在默认情况下）我们不能把 this 绑定到一个常量对象上（参见 2.4.2 节，第 56 页）。这一情况也就使得我们不能在一个常量对象上调用普通的成员函数。

如果 isbn 是一个普通函数而且 this 是一个普通的指针参数，则我们应该把 this 声明成 const Sales_data *const。毕竟，在 isbn 的函数体内不会改变 this 所指的对象，所以把 this 设置为指向常量的指针有助于提高函数的灵活性。

然而，this 是隐式的并且不会出现在参数列表中，所以在哪儿将 this 声明成指向常量的指针就成为我们必须面对的问题。C++语言的做法是允许把 const 关键字放在成员函数的参数列表之后，此时，紧跟在参数列表后面的 const 表示 this 是一个指向常量的指针。像这样使用 const 的成员函数被称作**常量成员函数**（const member function）。

可以把 isbn 的函数体想象成如下的形式：

```
// 伪代码，说明隐式的 this 指针是如何使用的
// 下面的代码是非法的：因为我们不能显式地定义自己的 this 指针
// 谨记此处的 this 是一个指向常量的指针，因为 isbn 是一个常量成员
std::string Sales_data::isbn(const Sales_data *const this)
{ return this->isbn; }
```

因为 this 是指向常量的指针，所以常量成员函数不能改变调用它的对象的内容。在上例中，isbn 可以读取调用它的对象的数据成员，但是不能写入新值。

259 Note 常量对象，以及常量对象的引用或指针都只能调用常量成员函数。

类作用域和成员函数

回忆之前我们所学的知识，类本身就是一个作用域（参见 2.6.1 节，第 64 页）。类的成员函数的定义嵌套在类的作用域之内，因此，isbn 中用到的名字 bookNo 其实就是定义在 Sales_data 内的数据成员。

值得注意的是，即使 bookNo 定义在 isbn 之后，isbn 也还是能够使用 bookNo。就如我们将在 7.4.1 节（第 254 页）学习到的那样，编译器分两步处理类：首先编译成员的声明，然后才轮到成员函数体（如果有的话）。因此，成员函数体可以随意使用类中的其他成员而无须在意这些成员出现的次序。

在类的外部定义成员函数

像其他函数一样，当我们在类的外部定义成员函数时，成员函数的定义必须与它的声明匹配。也就是说，返回类型、参数列表和函数名都得与类内部的声明保持一致。如果成员被声明成常量成员函数，那么它的定义也必须在参数列表后明确指定 const 属性。同时，类外部定义的成员的名字必须包含它所属的类名：

```
double Sales_data::avg_price() const {
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

函数名 Sales_data::avg_price 使用作用域运算符（参见 1.2 节，第 7 页）来说明如下的事实：我们定义了一个名为 avg_price 的函数，并且该函数被声明在类 Sales_data 的作用域内。一旦编译器看到这个函数名，就能理解剩余的代码是位于类的作用域内的。因此，当 avg_price 使用 revenue 和 units_sold 时，实际上它隐式地使用了

Sales_data 的成员。

定义一个返回 this 对象的函数

函数 combine 的设计初衷类似于复合赋值运算符+=，调用该函数的对象代表的是赋值运算符左侧的运算对象，右侧运算对象则通过显式的实参被传入函数：

```
Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += rhs.units_sold; // 把 rhs 的成员加到 this 对象的成员上
    revenue += rhs.revenue;
    return *this;                 // 返回调用该函数的对象
}
```

当我们的交易处理程序调用如下的函数时，

260

```
total.combine(trans);             // 更新变量 total 当前的值
```

total 的地址被绑定到隐式的 this 参数上，而 rhs 绑定到了 trans 上。因此，当 combine 执行下面的语句时，

```
units_sold += rhs.units_sold;     // 把 rhs 的成员添加到 this 对象的成员中
```

效果等同于求 total.units_sold 和 trans.unit_sold 的和，然后把结果保存到 total.units_sold 中。

该函数一个值得关注的部分是它的返回类型和返回语句。一般来说，当我们定义的函数类似于某个内置运算符时，应该令该函数的行为尽量模仿这个运算符。内置的赋值运算符把它的左侧运算对象当成左值返回（参见 4.4 节，第 129 页），因此为了与它保持一致，combine 函数必须返回引用类型（参见 6.3.2 节，第 202 页）。因为此时的左侧运算对象是一个 Sales_data 的对象，所以返回类型应该是 Sales_data&。

如前所述，我们无须使用隐式的 this 指针访问函数调用者的某个具体成员，而是需要把调用函数的对象当成一个整体来访问：

```
return *this;                     // 返回调用该函数的对象
```

其中，return 语句解引用 this 指针以获得执行该函数的对象，换句话说，上面的这个调用返回 total 的引用。

7.1.2 节练习

练习 7.2：曾在 2.6.2 节的练习（第 67 页）中编写了一个 Sales_data 类，请向这个类添加 combine 和 isbn 成员。

练习 7.3：修改 7.1.1 节（第 229 页）的交易处理程序，令其使用这些成员。

练习 7.4：编写一个名为 Person 的类，使其表示人员的姓名和住址。使用 string 对象存放这些元素，接下来的练习将不断充实这个类的其他特征。

练习 7.5：在你的 Person 类中提供一些操作使其能够返回姓名和住址。这些函数是否应该是 const 的呢？解释原因。

7.1.3 定义类相关的非成员函数

类的作者常常需要定义一些辅助函数，比如 add、read 和 print 等。尽管这些函数定义的操作从概念上来说属于类的接口的组成部分，但它们实际上并不属于类本身。

我们定义非成员函数的方式与定义其他函数一样，通常把函数的声明和定义分离开来
261 （参见 6.1.2 节，第 168 页）。如果函数在概念上属于类但是不定义在类中，则它一般应与类声明（而非定义）在同一个头文件内。在这种方式下，用户使用接口的任何部分都只需要引入一个文件。

一般来说，如果非成员函数是类接口的组成部分，则这些函数的声明应该与类在同一个头文件内。

定义 read 和 print 函数

下面的 read 和 print 函数与 2.6.2 节（第 66 页）中的代码作用一样，而且代码本身也非常相似：

```
// 输入的交易信息包括 ISBN、售出总数和售出价格
istream &read(istream &is, Sales_data &item)
{
    double price = 0;
    is >> item.bookNo >> item.units_sold >> price;
    item.revenue = price * item.units_sold;
    return is;
}
ostream &print(ostream &os, const Sales_data &item)
{
    os  << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

read 函数从给定流中将数据读到给定的对象里，print 函数则负责将给定对象的内容打印到给定的流中。

除此之外，关于上面的函数还有两点是非常重要的。第一点，read 和 print 分别接受一个各自 IO 类型的引用作为其参数，这是因为 IO 类属于不能被拷贝的类型，因此我们只能通过引用来传递它们（参见 6.2.2 节，第 188 页）。而且，因为读取和写入的操作会改变流的内容，所以两个函数接受的都是普通引用，而非对常量的引用。

第二点，print 函数不负责换行。一般来说，执行输出任务的函数应该尽量减少对格式的控制，这样可以确保由用户代码来决定是否换行。

定义 add 函数

add 函数接受两个 Sales_data 对象作为其参数，返回值是一个新的 Sales_data，用于表示前两个对象的和：

```
Sales_data add(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;          // 把 lhs 的数据成员拷贝给 sum
```

```
    sum.combine(rhs);              // 把 rhs 的数据成员加到 sum 当中
    return sum;
}
```

在函数体中，我们定义了一个新的 Sales_data 对象并将其命名为 sum。sum 将用于存放两笔交易的和，我们用 lhs 的副本来初始化 sum。默认情况下，拷贝类的对象其实拷贝的是对象的数据成员。在拷贝工作完成之后，sum 的 bookNo、units_sold 和 revenue 将和 lhs 一致。接下来我们调用 combine 函数，将 rhs 的 units_sold 和 revenue 添加给 sum。最后，函数返回 sum 的副本。

262

7.1.3 节练习

练习 7.6：对于函数 add、read 和 print，定义你自己的版本。

练习 7.7：使用这些新函数重写 7.1.2 节（第 233 页）练习中的交易处理程序。

练习 7.8：为什么 read 函数将其 Sales_data 参数定义成普通的引用，而 print 将其参数定义成常量引用？

练习 7.9：对于 7.1.2 节（第 233 页）练习中的代码，添加读取和打印 Person 对象的操作。

练习 7.10：在下面这条 if 语句中，条件部分的作用是什么？

```
if (read(read(cin, data1), data2))
```

7.1.4 构造函数

每个类都分别定义了它的对象被初始化的方式，类通过一个或几个特殊的成员函数来控制其对象的初始化过程，这些函数叫做**构造函数**（constructor）。构造函数的任务是初始化类对象的数据成员，无论何时只要类的对象被创建，就会执行构造函数。

在这一节中，我们将介绍定义构造函数的基础知识。构造函数是一个非常复杂的问题，我们还会在 7.5 节（第 257 页）、15.7 节（第 551 页）、18.1.3 节（第 689 页）和第 13 章介绍更多关于构造函数的知识。

构造函数的名字和类名相同。和其他函数不一样的是，构造函数没有返回类型；除此之外类似于其他的函数，构造函数也有一个（可能为空的）参数列表和一个（可能为空的）函数体。类可以包含多个构造函数，和其他重载函数差不多（参见 6.4 节，第 206 页），不同的构造函数之间必须在参数数量或参数类型上有所区别。

不同于其他成员函数，构造函数不能被声明成 const 的（参见 7.1.2 节，第 231 页）。当我们创建类的一个 const 对象时，直到构造函数完成初始化过程，对象才能真正取得其“常量”属性。因此，构造函数在 const 对象的构造过程中可以向其写值。

合成的默认构造函数

我们的 Sales_data 类并没有定义任何构造函数，可是之前使用了 Sales_data 对象的程序仍然可以正确地编译和运行。举个例子，第 229 页的程序定义了两个对象：

```
Sales_data total;          // 保存当前求和结果的变量
Sales_data trans;          // 保存下一条交易数据的变量
```

263 这时我们不禁要问：total 和 trans 是如何初始化的呢？

我们没有为这些对象提供初始值，因此我们知道它们执行了默认初始化（参见 2.2.1 节，第 40 页）。类通过一个特殊的构造函数来控制默认初始化过程，这个函数叫做**默认构造函数**（default constructor）。默认构造函数无须任何实参。

如我们所见，默认构造函数在很多方面都有其特殊性。其中之一是，如果我们的类没有显式地定义构造函数，那么编译器就会为我们隐式地定义一个默认构造函数。

编译器创建的构造函数又被称为**合成的默认构造函数**（synthesized default constructor）。对于大多数类来说，这个合成的默认构造函数将按照如下规则初始化类的数据成员：

- 如果存在类内的初始值（参见 2.6.1 节，第 64 页），用它来初始化成员。
- 否则，默认初始化（参见 2.2.1 节，第 40 页）该成员。

因为 Sales_data 为 units_sold 和 revenue 提供了初始值，所以合成的默认构造函数将使用这些值来初始化对应的成员；同时，它把 bookNo 默认初始化成一个空字符串。

某些类不能依赖于合成的默认构造函数

合成的默认构造函数只适合非常简单的类，比如现在定义的这个 Sales_data 版本。对于一个普通的类来说，必须定义它自己的默认构造函数，原因有三：第一个原因也是最容易理解的一个原因就是编译器只有在发现类不包含任何构造函数的情况下才会替我们生成一个默认的构造函数。一旦我们定义了一些其他的构造函数，那么除非我们再定义一个默认的构造函数，否则类将没有默认构造函数。这条规则的依据是，如果一个类在某种情况下需要控制对象初始化，那么该类很可能在所有情况下都需要控制。

只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数。

第二个原因是对于某些类来说，合成的默认构造函数可能执行错误的操作。回忆我们之前介绍过的，如果定义在块中的内置类型或复合类型（比如数组和指针）的对象被默认初始化（参见 2.2.1 节，第 40 页），则它们的值将是未定义的。该准则同样适用于默认初始化的内置类型成员。因此，含有内置类型或复合类型成员的类应该在类的内部初始化这些成员，或者定义一个自己的默认构造函数。否则，用户在创建类的对象时就可能得到未定义的值。

如果类包含有内置类型或者复合类型的成员，则只有当这些成员全都被赋予了类内的初始值时，这个类才适合于使用合成的默认构造函数。

264 第三个原因是有的时候编译器不能为某些类合成默认的构造函数。例如，如果类中包含一个其他类类型的成员且这个成员的类型没有默认构造函数，那么编译器将无法初始化该成员。对于这样的类来说，我们必须自定义默认构造函数，否则该类将没有可用的默认构造函数。在 13.1.6 节（第 449 页）中我们将看到还有其他一些情况也会导致编译器无法生成一个正确的默认构造函数。

定义 Sales_data 的构造函数

对于我们的 Sales_data 类来说，我们将使用下面的参数定义 4 个不同的构造函数：

- 一个 istream&，从中读取一条交易信息。

- 一个 const string&，表示 ISBN 编号；一个 unsigned，表示售出的图书数量；以及一个 double，表示图书的售出价格。
- 一个 const string&，表示 ISBN 编号；编译器将赋予其他成员默认值。
- 一个空参数列表（即默认构造函数），正如刚刚介绍的，既然我们已经定义了其他构造函数，那么也必须定义一个默认构造函数。

给类添加了这些成员之后，将得到

```
struct Sales_data {
    // 新增的构造函数
    Sales_data() = default;
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n, double p):
               bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(std::istream &);
    // 之前已有的其他成员
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

= default 的含义

我们从解释默认构造函数的含义开始：

```
Sales_data() = default;
```

首先请明确一点：因为该构造函数不接受任何实参，所以它是一个默认构造函数。我们定义这个构造函数的目的仅仅是因为我们既需要其他形式的构造函数，也需要默认的构造函数。我们希望这个函数的作用完全等同于之前使用的合成默认构造函数。

在 C++11 新标准中，如果我们需要默认的行为，那么可以通过在参数列表后面写上 **= default** 来要求编译器生成构造函数。其中，= default 既可以和声明一起出现在类的内部，也可以作为定义出现在类的外部。和其他函数一样，如果= default 在类的内部，则默认构造函数是内联的；如果它在类的外部，则该成员默认情况下不是内联的。

265

C++ 11

上面的默认构造函数之所以对 Sales_data 有效，是因为我们为内置类型的数据成员提供了初始值。如果你的编译器不支持类内初始值，那么你的默认构造函数就应该使用构造函数初始值列表（马上就会介绍）来初始化类的每个成员。

构造函数初始值列表

接下来我们介绍类中定义的另外两个构造函数：

```
Sales_data(const std::string &s): bookNo(s) { }
Sales_data(const std::string &s, unsigned n, double p):
           bookNo(s), units_sold(n), revenue(p*n) { }
```

这两个定义中出现了新的部分，即冒号以及冒号和花括号之间的代码，其中花括号定义了

（空的）函数体。我们把新出现的部分称为**构造函数初始值列表**（constructor initialize list），它负责为新创建的对象的一个或几个数据成员赋初值。构造函数初始值是成员名字的一个列表，每个名字后面紧跟括号括起来的（或者在花括号内的）成员初始值。不同成员的初始化通过逗号分隔开来。

含有三个参数的构造函数分别使用它的前两个参数初始化成员 bookNo 和 units_sold，revenue 的初始值则通过将售出图书总数和每本书单价相乘计算得到。

只有一个 string 类型参数的构造函数使用这个 string 对象初始化 bookNo，对于 units_sold 和 revenue 则没有显式地初始化。当某个数据成员被构造函数初始值列表忽略时，它将以与合成默认构造函数相同的方式隐式初始化。在此例中，这样的成员使用类内初始值初始化，因此只接受一个 string 参数的构造函数等价于

```
// 与上面定义的那个构造函数效果相同
Sales_data(const std::string &s):
           bookNo(s), units_sold(0), revenue(0){ }
```

通常情况下，构造函数使用类内初始值不失为一种好的选择，因为只要这样的初始值存在我们就能确保为成员赋予了一个正确的值。不过，如果你的编译器不支持类内初始值，则所有构造函数都应该显式地初始化每个内置类型的成员。

Best Practices

构造函数不应该轻易覆盖掉类内的初始值，除非新赋的值与原值不同。如果你不能使用类内初始值，则所有构造函数都应该显式地初始化每个内置类型的成员。

266 有一点需要注意，在上面的两个构造函数中函数体都是空的。这是因为这些构造函数的唯一目的就是为数据成员赋初值，一旦没有其他任务需要执行，函数体也就为空了。

在类的外部定义构造函数

与其他几个构造函数不同，以 istream 为参数的构造函数需要执行一些实际的操作。在它的函数体内，调用了 read 函数以给数据成员赋以初值：

```
Sales_data::Sales_data(std::istream &is)
{
    read(is, *this); // read 函数的作用是从 is 中读取一条交易信息然后
                     // 存入 this 对象中
}
```

构造函数没有返回类型，所以上述定义从我们指定的函数名字开始。和其他成员函数一样，当我们在类的外部定义构造函数时，必须指明该构造函数是哪个类的成员。因此，Sales_data::Sales_data 的含义是我们定义 Sales_data 类的成员，它的名字是 Sales_data。又因为该成员的名字和类名相同，所以它是一个构造函数。

这个构造函数没有构造函数初始值列表，或者讲得更准确一点，它的构造函数初始值列表是空的。尽管构造函数初始值列表是空的，但是由于执行了构造函数体，所以对象的成员仍然能被初始化。

没有出现在构造函数初始值列表中的成员将通过相应的类内初始值（如果存在的话）初始化，或者执行默认初始化。对于 Sales_data 来说，这意味着一旦函数开始执行，则 bookNo 将被初始化成空 string 对象，而 units_sold 和 revenue 将是 0。

为了更好地理解调用函数 read 的意义，要特别注意 read 的第二个参数是一个 Sales_data 对象的引用。在 7.1.2 节（第 232 页）中曾经提到过，使用 this 来把对象当成一个整体访问，而非直接访问对象的某个成员。因此在此例中，我们使用*this 将“this”对象作为实参传递给 read 函数。

7.1.4 节练习

练习 7.11：在你的 Sales_data 类中添加构造函数，然后编写一段程序令其用到每个构造函数。

练习 7.12：把只接受一个 istream 作为参数的构造函数定义移到类的内部。

练习 7.13：使用 istream 构造函数重写第 229 页的程序。

练习 7.14：编写一个构造函数，令其用我们提供的类内初始值显式地初始化成员。

练习 7.15：为你的 Person 类添加正确的构造函数。

7.1.5 拷贝、赋值和析构

除了定义类的对象如何初始化之外，类还需要控制拷贝、赋值和销毁对象时发生的行为。对象在几种情况下会被拷贝，如我们初始化变量以及以值的方式传递或返回一个对象等（参见 6.2.1 节，第 187 页和 6.3.2 节，第 200 页）。当我们使用了赋值运算符（参见 4.4 节，第 129 页）时会发生对象的赋值操作。当对象不再存在时执行销毁的操作，比如一个局部对象会在创建它的块结束时被销毁（参见 6.1.1 节，第 184 页），当 vector 对象（或者数组）销毁时存储在其中的对象也会被销毁。

267

如果我们不主动定义这些操作，则编译器将替我们合成它们。一般来说，编译器生成的版本将对对象的每个成员执行拷贝、赋值和销毁操作。例如在 7.1.1 节（第 229 页）的书店程序中，当编译器执行如下赋值语句时，

```
total = trans; // 处理下一本书的信息
```

它的行为与下面的代码相同

```
// Sales_data 的默认赋值操作等价于：
total.bookNo = trans.bookNo;
total.units_sold = trans.units_sold;
total.revenue = trans.revenue;
```

我们将在第 13 章中介绍如何自定义上述操作。

某些类不能依赖于合成的版本

尽管编译器能替我们合成拷贝、赋值和销毁的操作，但是必须要清楚的一点是，对于某些类来说合成的版本无法正常工作。特别是，当类需要分配类对象之外的资源时，合成的版本常常会失效。举个例子，第 12 章将介绍 C++程序是如何分配和管理动态内存的。而在 13.1.4 节（第 447 页）我们将会看到，管理动态内存的类通常不能依赖于上述操作的合成版本。

不过值得注意的是，很多需要动态内存的类能（而且应该）使用 vector 对象或者 string 对象管理必要的存储空间。使用 vector 或者 string 的类能避免分配和释放内存带来的复杂性。

进一步讲，如果类包含 vector 或者 string 成员，则其拷贝、赋值和销毁的合成版本能够正常工作。当我们对含有 vector 成员的对象执行拷贝或者赋值操作时，vector 类会设法拷贝或者赋值成员中的元素。当这样的对象被销毁时，将销毁 vector 对象，也就是依次销毁 vector 中的每一个元素。这一点与 string 是非常类似的。

在学习第 13 章关于如何自定义操作的知识之前，类中所有分配的资源都应该直接以类的数据成员的形式存储。

7.2 访问控制与封装

268

到目前为止，我们已经为类定义了接口，但并没有任何机制强制用户使用这些接口。我们的类还没有封装，也就是说，用户可以直达 Sales_data 对象的内部并且控制它的具体实现细节。在 C++语言中，我们使用**访问说明符**（access specifiers）加强类的封装性：

- 定义在 **public** 说明符之后的成员在整个程序内可被访问，public 成员定义类的接口。
- 定义在 **private** 说明符之后的成员可以被类的成员函数访问，但是不能被使用该类的代码访问，private 部分封装了（即隐藏了）类的实现细节。

再一次定义 Sales_data 类，其新形式如下所示：

```
class Sales_data {
public:              // 添加了访问说明符
    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&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:             // 添加了访问说明符
    double avg_price() const
        { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

作为接口的一部分，构造函数和部分成员函数（即 isbn 和 combine）紧跟在 public 说明符之后；而数据成员和作为实现部分的函数则跟在 private 说明符后面。

一个类可以包含 0 个或多个访问说明符，而且对于某个访问说明符能出现多少次也没有严格限定。每个访问说明符指定了接下来的成员的访问级别，其有效范围直到出现下一个访问说明符或者到达类的结尾处为止。

使用 class 或 struct 关键字

在上面的定义中我们还做了一个微妙的变化，我们使用了 **class** 关键字而非 **struct** 开始类的定义。这种变化仅仅是形式上有所不同，实际上我们可以使用这两个关键字中的任何一个定义类。唯一的一点区别是，struct 和 class 的默认访问权限不太一样。

类可以在它的第一个访问说明符之前定义成员，对这种成员的访问权限依赖于类定义

的方式。如果我们使用 struct 关键字，则定义在第一个访问说明符之前的成员是 public 的；相反，如果我们使用 class 关键字，则这些成员是 private 的。

出于统一编程风格的考虑，当我们希望定义的类的所有成员是 public 的时，使用 struct；反之，如果希望成员是 private 的，使用 class。 269

使用 class 和 struct 定义类唯一的区别就是默认的访问权限。

7.2 节练习

练习 7.16：在类的定义中对于访问说明符出现的位置和次数有限定吗？如果有，是什么？什么样的成员应该定义在 public 说明符之后？什么样的成员应该定义在 private 说明符之后？

练习 7.17：使用 class 和 struct 时有区别吗？如果有，是什么？

练习 7.18：封装是何含义？它有什么用处？

练习 7.19：在你的 Person 类中，你将把哪些成员声明成 public 的？哪些声明成 private 的？解释你这样做的原因。

7.2.1 友元

既然 Sales_data 的数据成员是 private 的，我们的 read、print 和 add 函数也就无法正常编译了，这是因为尽管这几个函数是类的接口的一部分，但它们不是类的成员。

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**（friend）。如果类想把一个函数作为它的友元，只需要增加一条以 friend 关键字开始的函数声明语句即可：

```
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&, const Sales_data&);
// 其他成员及访问说明符与之前一致
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&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// Sales_data 接口的非成员组成部分的声明
Sales_data add(const Sales_data&, const Sales_data&);
```

270

```
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
```

友元声明只能出现在类定义的内部，但是在类内出现的具体位置不限。友元不是类的成员也不受它所在区域访问控制级别的约束。我们将在 7.3.4 节（第 250 页）介绍更多关于友元的知识。

一般来说，最好在类定义开始或结束前的位置集中声明友元。

关键概念：封装的益处

封装有两个重要的优点：

- 确保用户代码不会无意间破坏封装对象的状态。
- 被封装的类的具体实现细节可以随时改变，而无须调整用户级别的代码。

一旦把数据成员定义成 private 的，类的作者就可以比较自由地修改数据了。当实现部分改变时，我们只需要检查类的代码本身以确认这次改变有什么影响；换句话说，只要类的接口不变，用户代码就无须改变。如果数据是 public 的，则所有使用了原来数据成员的代码都可能失效，这时我们必须定位并重写所有依赖于老版本实现的代码，之后才能重新使用该程序。

把数据成员的访问权限设成 private 还有另外一个好处，这么做能防止由于用户的原因造成数据被破坏。如果我们发现有程序缺陷破坏了对象的状态，则可以在有限的范围内定位缺陷：因为只有实现部分的代码可能产生这样的错误。因此，将查错限制在有限范围内将能极大地降低维护代码及修正程序错误的难度。

尽管当类的定义发生改变时无须更改用户代码，但是使用了该类的源文件必须重新编译。

友元的声明

友元的声明仅仅指定了访问的权限，而非一个通常意义上的函数声明。如果我们希望类的用户能够调用某个友元函数，那么我们就必须在友元声明之外再专门对函数进行一次声明。

为了使友元对类的用户可见，我们通常把友元的声明与类本身放置在同一个头文件中（类的外部）。因此，我们的 Sales_data 头文件应该为 read、print 和 add 提供独立的声明（除了类内部的友元声明之外）。

许多编译器并未强制限定友元函数必须在使用之前在类的外部声明。

271 一些编译器允许在尚无友元函数的初始声明的情况下就调用它。不过即使你的编译器支持这种行为，最好还是提供一个独立的函数声明。这样即使你更换了一个有这种强制要求的编译器，也不必改变代码。

7.2.1 节练习

练习 7.20：友元在什么时候有用？请分别列举出使用友元的利弊。

练习 7.21：修改你的 Sales_data 类使其隐藏实现的细节。你之前编写的关于 Sales_data 操作的程序应该继续使用，借助类的新定义重新编译该程序，确保其工作正常。

练习 7.22：修改你的 Person 类使其隐藏实现的细节。

7.3 类的其他特性

虽然 Sales_data 类非常简单，但是通过它我们已经了解 C++语言中关于类的许多语法要点。在本节中，我们将继续介绍 Sales_data 没有体现出来的一些类的特性。这些特性包括：类型成员、类的成员的类内初始值、可变数据成员、内联成员函数、从成员函数返回*this、关于如何定义并使用类类型及友元类的更多知识。

7.3.1 类成员再探

为了展示这些新的特性，我们需要定义一对相互关联的类，它们分别是 Screen 和 Window_mgr。

定义一个类型成员

Screen 表示显示器中的一个窗口。每个 Screen 包含一个用于保存 Screen 内容的 string 成员和三个 string::size_type 类型的成员，它们分别表示光标的位置以及屏幕的高和宽。

除了定义数据和函数成员之外，类还可以自定义某种类型在类中的别名。由类定义的类型名字和其他成员一样存在访问限制，可以是 public 或者 private 中的一种：

```
class Screen {
public:
    typedef std::string::size_type pos;
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

我们在 Screen 的 public 部分定义了 pos，这样用户就可以使用这个名字。Screen 的用户不应该知道 Screen 使用了一个 string 对象来存放它的数据，因此通过把 pos 定义成 public 成员可以隐藏 Screen 实现的细节。 272

关于 pos 的声明有两点需要注意。首先，我们使用了 typedef（参见 2.5.1 节，第 60 页），也可以等价地使用类型别名（参见 2.5.1 节，第 60 页）：

```
class Screen {
public:
    // 使用类型别名等价地声明一个类型名字
    using pos = std::string::size_type;
    // 其他成员与之前的版本一致
};
```

其次，用来定义类型的成员必须先定义后使用，这一点与普通成员有所区别，具体原因将

在 7.4.1 节（第 254 页）解释。因此，类型成员通常出现在类开始的地方。

Screen 类的成员函数

要使我们的类更加实用，还需要添加一个构造函数令用户能够定义屏幕的尺寸和内容，以及其他两个成员，分别负责移动光标和读取给定位置的字符：

```
class Screen {
public:
    typedef std::string::size_type pos;
    Screen() = default; // 因为 Screen 有另一个构造函数，
                        // 所以本函数是必需的
    // cursor 被其类内初始值初始化为 0
    Screen(pos ht, pos wd, char c): height(ht), width(wd),
    contents(ht * wd, c) { }
    char get() const                              // 读取光标处的字符
         { return contents[cursor]; }             // 隐式内联
    inline char get(pos ht, pos wd) const;        // 显式内联
    Screen &move(pos r, pos c);                   // 能在之后被设为内联
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

因为我们已经提供了一个构造函数，所以编译器将不会自动生成默认的构造函数。如果我们的类需要默认构造函数，必须显式地把它声明出来。在此例中，我们使用=default 告诉编译器为我们合成默认的构造函数（参见 7.1.4 节，第 237 页）。

需要指出的是，第二个构造函数（接受三个参数）为 cursor 成员隐式地使用了类内初始值（参见 7.1.4 节，第 238 页）。如果类中不存在 cursor 的类内初始值，我们就需要像其他成员一样显式地初始化 cursor 了。

273 令成员作为内联函数

在类中，常有一些规模较小的函数适合于被声明成内联函数。如我们之前所见的，定义在类内部的成员函数是自动 inline 的（参见 6.5.2 节，第 213 页）。因此，Screen 的构造函数和返回光标所指字符的 get 函数默认是 inline 函数。

我们可以在类的内部把 inline 作为声明的一部分显式地声明成员函数，同样的，也能在类的外部用 inline 关键字修饰函数的定义：

```
inline                                  // 可以在函数的定义处指定 inline
Screen &Screen::move(pos r, pos c)
{
    pos row = r * width;                // 计算行的位置
    cursor = row + c;                   // 在行内将光标移动到指定的列
    return *this;                       // 以左值的形式返回对象
}
char Screen::get(pos r, pos c) const    // 在类的内部声明成 inline
{
    pos row = r * width;                // 计算行的位置
    return contents[row + c];           // 返回给定列的字符
}
```

虽然我们无须在声明和定义的地方同时说明 inline，但这么做其实是合法的。不过，最好只在类外部定义的地方说明 inline，这样可以使类更容易理解。

和我们在头文件中定义 inline 函数的原因一样（参见 6.5.2 节，第 214 页），inline 成员函数也应该与相应的类定义在同一个头文件中。

重载成员函数

和非成员函数一样，成员函数也可以被重载（参见 6.4 节，第 206 页），只要函数之间在参数的数量和/或类型上有所区别就行。成员函数的函数匹配过程（参见 6.4 节，第 208 页）同样与非成员函数非常类似。

举个例子，我们的 Screen 类定义了两个版本的 get 函数。一个版本返回光标当前位置的字符；另一个版本返回由行号和列号确定的位置的字符。编译器根据实参的数量来决定运行哪个版本的函数：

```
Screen myscreen;
char ch = myscreen.get();      // 调用 Screen::get()
ch = myscreen.get(0,0);        // 调用 Screen::get(pos, pos)
```

可变数据成员

有时（但并不频繁）会发生这样一种情况，我们希望能修改类的某个数据成员，即使是在一个 const 成员函数内。可以通过在变量的声明中加入 mutable 关键字做到这一点。

一个**可变数据成员**（mutable data member）永远不会是 const，即使它是 const 对象的成员。因此，一个 const 成员函数可以改变一个可变成员的值。举个例子，我们将给 Screen 添加一个名为 access_ctr 的可变成员，通过它我们可以追踪每个 Screen 的成员函数被调用了多少次：

274

```
class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr;    // 即使在一个 const 对象内也能被修改
    // 其他成员与之前的版本一致
};
void Screen::some_member() const
{
    ++access_ctr;          // 保存一个计数值，用于记录成员函数被调用的次数
    // 该成员需要完成的其他工作
}
```

尽管 some_member 是一个 const 成员函数，它仍然能够改变 access_ctr 的值。该成员是个可变成员，因此任何成员函数，包括 const 函数在内都能改变它的值。

类数据成员的初始值

在定义好 Screen 类之后，我们将继续定义一个窗口管理类并用它表示显示器上的一组 Screen。这个类将包含一个 Screen 类型的 vector，每个元素表示一个特定的 Screen。默认情况下，我们希望 Window_mgr 类开始时总是拥有一个默认初始化的

C++11 Screen。在C++11新标准中，最好的方式就是把这个默认值声明成一个类内初始值（参见2.6.1节，第64页）：

```
class Window_mgr {
private:
    // 这个Window_mgr追踪的Screen
    // 默认情况下，一个Window_mgr包含一个标准尺寸的空白Screen
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};
```

当我们初始化类类型的成员时，需要为构造函数传递一个符合成员类型的实参。在此例中，我们使用一个单独的元素值对vector成员执行了列表初始化（参见3.3.1节，第87页），这个Screen的值被传递给vector<Screen>的构造函数，从而创建了一个单元素的vector对象。具体地说，Screen的构造函数接受两个尺寸参数和一个字符值，创建了一个给定大小的空白屏幕对象。

如我们之前所知的，类内初始值必须使用=的初始化形式（初始化Screen的数据成员时所用的）或者花括号括起来的直接初始化形式（初始化screens所用的）。

Note 当我们提供一个类内初始值时，必须以符号=或者花括号表示。

275

7.3.1 节练习

练习7.23：编写你自己的Screen类。

练习7.24：给你的Screen类添加三个构造函数：一个默认构造函数；另一个构造函数接受宽和高的值，然后将contents初始化成给定数量的空白；第三个构造函数接受宽和高的值以及一个字符，该字符作为初始化之后屏幕的内容。

练习7.25：Screen能安全地依赖于拷贝和赋值操作的默认版本吗？如果能，为什么？如果不能，为什么？

练习7.26：将Sales_data::avg_price定义成内联函数。

7.3.2 返回*this的成员函数

接下来我们继续添加一些函数，它们负责设置光标所在位置的字符或者其他任一给定位置的字符：

```
class Screen {
public:
    Screen &set(char);
    Screen &set(pos, pos, char);
    // 其他成员和之前的版本一致
};
inline Screen &Screen::set(char c)
{
    contents[cursor] = c;              // 设置当前光标所在位置的新值
    return *this;                      // 将this对象作为左值返回
```