```
            if (!cin || c == 'n')
                break; // 跳出 while 循环
        }
    }
```

程序本来要执行的任务出现在 try 语句块中，这是因为这段代码可能会抛出一个 runtime_error 类型的异常。

try 语句块对应一个 catch 子句，该子句负责处理类型为 runtime_error 的异常。如果 try 语句块的代码抛出了 runtime_error 异常，接下来执行 catch 块内的语句。在我们书写的 catch 子句中，输出一段提示信息要求用户指定程序是否继续。如果用户输入'n'，执行 break 语句并退出 while 循环；否则，直接执行 while 循环的右侧花括号，意味着程序控制权跳回到 while 条件部分准备下一次迭代。

给用户的提示信息中输出了 err.what() 的返回值。我们知道 err 的类型是 runtime_error，因此能推断 what 是 runtime_error 类的一个成员函数（参见 1.5.2 节，第 20 页）。每个标准库异常类都定义了名为 what 的成员函数，这些函数没有参数，返回值是 C 风格字符串（即 const char*）。其中，runtime_error 的 what 成员返回的是初始化一个具体对象时所用的 string 对象的副本。如果上一节编写的代码抛出异常，则本节的 catch 子句输出

196

```
Data must refer to same ISBN
Try Again? Enter y or n
```

函数在寻找处理代码的过程中退出

在复杂系统中，程序在遇到抛出异常的代码前，其执行路径可能已经经过了多个 try 语句块。例如，一个 try 语句块可能调用了包含另一个 try 语句块的函数，新的 try 语句块可能调用了包含又一个 try 语句块的新函数，以此类推。

寻找处理代码的过程与函数调用链刚好相反。当异常被抛出时，首先搜索抛出该异常的函数。如果没找到匹配的 catch 子句，终止该函数，并在调用该函数的函数中继续寻找。如果还是没有找到匹配的 catch 子句，这个新的函数也被终止，继续搜索调用它的函数。以此类推，沿着程序的执行路径逐层回退，直到找到适当类型的 catch 子句为止。

如果最终还是没能找到任何匹配的 catch 子句，程序转到名为 **terminate** 的标准库函数。该函数的行为与系统有关，一般情况下，执行该函数将导致程序非正常退出。

对于那些没有任何 try 语句块定义的异常，也按照类似的方式处理：毕竟，没有 try 语句块也就意味着没有匹配的 catch 子句。如果一段程序没有 try 语句块且发生了异常，系统会调用 terminate 函数并终止当前程序的执行。

提示：编写异常安全的代码非常困难

要好好理解这句话：异常中断了程序的正常流程。异常发生时，调用者请求的一部分计算可能已经完成了，另一部分则尚未完成。通常情况下，略过部分程序意味着某些对象处理到一半就戛然而止，从而导致对象处于无效或未完成的状态，或者资源没有正常释放，等等。那些在异常发生期间正确执行了“清理”工作的程序被称作异常安全（exception safe）的代码。然而经验表明，编写异常安全的代码非常困难，这部分知识也（远远）超出了本书的范围。

对于一些程序来说，当异常发生时只是简单地终止程序。此时，我们不怎么需要担

心异常安全的问题。

但是对于那些确实要处理异常并继续执行的程序，就要加倍注意了。我们必须时刻清楚异常何时发生，异常发生后程序应如何确保对象有效、资源无泄漏、程序处于合理状态，等等。

我们会在本书中介绍一些比较常规的提升异常安全性的技术。但是读者需要注意，如果你的程序要求非常鲁棒的异常处理，那么仅有我们介绍的这些技术恐怕还是不够的。

197

5.6.3 标准异常

C++标准库定义了一组类，用于报告标准库函数遇到的问题。这些异常类也可以在用户编写的程序中使用，它们分别定义在4个头文件中：

- exception 头文件定义了最通用的异常类 exception。它只报告异常的发生，不提供任何额外信息。
- stdexcept 头文件定义了几种常用的异常类，详细信息在表5.1中列出。
- new 头文件定义了 bad_alloc 异常类型，这种类型将在12.1.2节（第407页）详细介绍。
- type_info 头文件定义了 bad_cast 异常类型，这种类型将在19.2节（第731页）详细介绍。

表 5.1：<stdexcept>定义的异常类

exception	最常见的问题
runtime_error	只有在运行时才能检测出的问题
range_error	运行时错误：生成的结果超出了有意义的值域范围
overflow_error	运行时错误：计算上溢
underflow_error	运行时错误：计算下溢
logic_error	程序逻辑错误
domain_error	逻辑错误：参数对应的结果值不存在
invalid_argument	逻辑错误：无效参数
length_error	逻辑错误：试图创建一个超出该类型最大长度的对象
out_of_range	逻辑错误：使用一个超出有效范围的值

标准库异常类只定义了几种运算，包括创建或拷贝异常类型的对象，以及为异常类型的对象赋值。

我们只能以默认初始化（参见2.2.1节，第40页）的方式初始化 exception、bad_alloc 和 bad_cast 对象，不允许为这些对象提供初始值。

其他异常类型的行为则恰好相反：应该使用 string 对象或者C风格字符串初始化这些类型的对象，但是不允许使用默认初始化的方式。当创建此类对象时，必须提供初始值，该初始值含有错误相关的信息。

异常类型只定义了一个名为 what 的成员函数，该函数没有任何参数，返回值是一个指向C风格字符串（参见3.5.4节，第109页）的 const char*。该字符串的目的是提供关于异常的一些文本信息。

what 函数返回的 C 风格字符串的内容与异常对象的类型有关。如果异常类型有一个字符串初始值，则 what 返回该字符串。对于其他无初始值的异常类型来说，what 返回的内容由编译器决定。 198

5.6.3 节练习

练习 5.23：编写一段程序，从标准输入读取两个整数，输出第一个数除以第二个数的结果。

练习 5.24：修改你的程序，使得当第二个数是 0 时抛出异常。先不要设定 catch 子句，运行程序并真的为除数输入 0，看看会发生什么？

练习 5.25：修改上一题的程序，使用 try 语句块去捕获异常。catch 子句应该为用户输出一条提示信息，询问其是否输入新数并重新执行 try 语句块的内容。

199

小结

C++语言仅提供了有限的语句类型，它们中的大多数会影响程序的控制流程：

- while、for 和 do while 语句，执行迭代操作。
- if 和 switch 语句，提供条件分支结构。
- continue 语句，终止循环的当前一次迭代。
- break 语句，退出循环或者 switch 语句。
- goto 语句，将控制权转移到一条带标签的语句。
- try 和 catch，将一段可能抛出异常的语句序列括在花括号里构成 try 语句块。catch 子句负责处理代码抛出的异常。
- throw 表达式语句，存在于代码块中，将控制权转移到相关的 catch 子句。
- return 语句，终止函数的执行。我们将在第 6 章介绍 return 语句。

除此之外还有表达式语句和声明语句。表达式语句用于求解表达式，关于变量的声明和定义在第 2 章已经介绍过了。

术语表

块（block）包围在花括号内的由 0 条或多条语句组成的序列。块也是一条语句，所以只要是能使用语句的地方，就可以使用块。

break 语句（break statement）终止离它最近的循环或 switch 语句。控制权转移到循环或 switch 之后的第一条语句。

case 标签（case label）在 switch 语句中紧跟在 case 关键字之后的常量表达式（参见2.4.4节，第58页）。在同一个 switch 语句中任意两个 case 标签的值不能相同。

catch 子句（catch clause）由三部分组成：catch 关键字、括号里的异常声明以及一个语句块。catch 子句的代码负责处理在异常声明中定义的异常。

复合语句（compound statement）和块是同义词。

continue 语句（continue statement）终止离它最近的循环的当前迭代。控制权转移到 while 或 do while 语句的条件部分、或者范围 for 循环的下一次迭代、或者传统 for 循环头部的表达式。

悬垂 else（dangling else）是一个俗语，指的是如何处理嵌套 if 语句中 if 分支多于 else 分支的情况。C++语言规定，else 应该与前一个未匹配的 if 匹配在一起。使用花括号可以把位于内层的 if 语句隐藏起来，这样程序员就能更好地控制 else 该与哪个 if 匹配。

default 标签（default label）是一种特殊的 case 标签，当 switch 表达式的值与所有 case 标签都无法匹配时，程序执行 default 标签下的内容。

200 **do while 语句**（do while statement）与 while 语句类似，区别是 do while 语句先执行循环体，再判断条件。循环体代码至少会执行一次。

异常类（exception class）是标准库定义的一组类，用于表示程序发生的错误。表 5.1（第 176 页）列出了不同用途的异常类。

异常声明（exception declaration）位于 catch 子句中的声明，指定了该 catch 子句能处理的异常类型。

异常处理代码（exception handler）程序某处引发异常后，用于处理该异常的另一

处代码。和 catch 子句是同义词。

异常安全（exception safe）是一个术语，表示的含义是当抛出异常后，程序能执行正确的行为。

表达式语句（expression statement）即一条表达式后面跟上一个分号，令表达式执行求值过程。

控制流（flow of control）程序的执行路径。

for 语句（for statement）提供迭代执行的迭代语句。常常用于遍历一个容器或者重复计算若干次。

goto 语句（goto statement）令控制权无条件转移到同一函数中一个指定的带标签语句。goto 语句容易造成程序的控制流混乱，应禁止使用。

if else 语句（if else statement）判断条件，根据其结果分别执行 if 分支或 else 分支的语句。

if 语句（if statement）判断条件，根据其结果有选择地执行语句。如果条件为真，执行 if 分支的代码；如果条件为假，控制权转移到 if 结构之后的第一条语句。

带标签语句（labeled statement）前面带有标签的语句。所谓标签是指一个标识符以及紧跟着的一个冒号。对于同一个标识符来说，用作标签的同时还能用于其他目的，互不干扰。

空语句（null statement）只含有一个分号的语句。

引发（raise）含义类似于 throw。在 C++ 语言中既可以说抛出异常，也可以说引发异常。

范围 for 语句（range for statement）在一个序列中进行迭代的语句。

switch 语句（switch statement）一种条件语句，首先求 switch 关键字后面表达式的值，如果某个 case 标签的值与表达式的值相等，程序直接跨过之前的代码从这个 case 标签开始执行。当所有 case 标签都无法匹配时，如果有 default 标签，从 default 标签继续执行；如果没有，结束 switch 语句。

terminate 是一个标准库函数，当异常没有被捕捉到时调用。terminate 终止当前程序的执行。

throw 表达式（throw expression）一种中断当前执行路径的表达式。throw 表达式抛出一个异常并把控制权转移到能处理该异常的最近的 catch 子句。

try 语句块（try block）跟在 try 关键字后面的块，以及一个或多个 catch 子句。如果 try 语句块的代码引发异常并且其中一个 catch 子句匹配该异常类型，则异常被该 catch 子句处理。否则，异常将由外围 try 语句块处理，或者程序终止。

while 语句（while statement）只要指定的条件为真，就一直迭代执行目标语句。随着条件真值的不同，循环可能执行多次，也可能一次也不执行。

第 6 章
函数

内容

6.1 函数基础 182
6.2 参数传递 187
6.3 返回类型和 return 语句 199
6.4 函数重载 206
6.5 特殊用途语言特性 211
6.6 函数匹配 217
6.7 函数指针 221
小结 225
术语表 225

本章首先介绍函数的定义和声明，包括参数如何传入函数以及函数如何返回结果。在C++语言中允许重载函数，也就是几个不同的函数可以使用同一个名字。所以接下来我们介绍重载函数的方法，以及编译器如何从函数的若干重载形式中选取一个与调用匹配的版本。最后，我们将介绍一些关于函数指针的知识。

202 函数是一个命名了的代码块，我们通过调用函数执行相应的代码。函数可以有 0 个或多个参数，而且（通常）会产生一个结果。可以重载函数，也就是说，同一个名字可以对应几个不同的函数。

6.1 函数基础

一个典型的函数（function）定义包括以下部分：返回类型（return type）、函数名字、由 0 个或多个形参（parameter）组成的列表以及函数体。其中，形参以逗号隔开，形参的列表位于一对圆括号之内。函数执行的操作在语句块（参见 5.1 节，第 155 页）中说明，该语句块称为函数体（function body）。

我们通过**调用运算符**（call operator）来执行函数。调用运算符的形式是一对圆括号，它作用于一个表达式，该表达式是函数或者指向函数的指针；圆括号之内是一个用逗号隔开的实参（argument）列表，我们用实参初始化函数的形参。调用表达式的类型就是函数的返回类型。

编写函数

举个例子，我们准备编写一个求数的阶乘的程序。n 的阶乘是从 1 到 n 所有数字的乘积，例如 5 的阶乘是 120。

```
1 * 2 * 3 * 4 * 5 = 120
```

程序如下所示：

```
// val 的阶乘是 val*(val - 1)*(val - 2) ...*((val - (val - 1))* 1)
int fact(int val)
{
    int ret = 1;          // 局部变量，用于保存计算结果
    while (val > 1)
        ret *= val--;     // 把 ret 和 val 的乘积赋给 ret，然后将 val 减 1
    return ret;           // 返回结果
}
```

函数的名字是 fact，它作用于一个整型参数，返回一个整型值。在 while 循环内部，在每次迭代时用后置递减运算符（参见 4.5 节，第 131 页）将 val 的值减 1。return 语句负责结束 fact 并返回 ret 的值。

调用函数

要调用 fact 函数，必须提供一个整数值，调用得到的结果也是一个整数：

```
int main()
{
    int j = fact(5);      // j 等于 120，即 fact(5)的结果
    cout << "5! is " << j << endl;
    return 0;
}
```

203 函数的调用完成两项工作：一是用实参初始化函数对应的形参，二是将控制权转移给被调用函数。此时，**主调函数**（calling function）的执行被暂时中断，被调函数（called function）开始执行。

执行函数的第一步是（隐式地）定义并初始化它的形参。因此，当调用 fact 函数时，首先创建一个名为 val 的 int 变量，然后将它初始化为调用时所用的实参 5。

当遇到一条 return 语句时函数结束执行过程。和函数调用一样，return 语句也完成两项工作：一是返回 return 语句中的值（如果有的话），二是将控制权从被调函数转移回主调函数。函数的返回值用于初始化调用表达式的结果，之后继续完成调用所在的表达式的剩余部分。因此，我们对 fact 函数的调用等价于如下形式：

```
int val = 5;          // 用字面值 5 初始化 val
int ret = 1;          // fact 函数体内的代码
while (val > 1)
    ret *= val--;
int j = ret;          // 用 ret 的副本初始化 j
```

形参和实参

实参是形参的初始值。第一个实参初始化第一个形参，第二个实参初始化第二个形参，以此类推。尽管实参与形参存在对应关系，但是并没有规定实参的求值顺序（参见 4.1.3 节，第 123 页）。编译器能以任意可行的顺序对实参求值。

实参的类型必须与对应的形参类型匹配，这一点与之前的规则是一致的，我们知道在初始化过程中初始值的类型也必须与初始化对象的类型匹配。函数有几个形参，我们就必须提供相同数量的实参。因为函数的调用规定实参数量应与形参数量一致，所以形参一定会被初始化。

在上面的例子中，fact 函数只有一个 int 类型的形参，所以每次我们调用它的时候，都必须提供一个能转换（参见 4.11 节，第 141 页）成 int 的实参：

```
fact("hello");        // 错误：实参类型不正确
fact();               // 错误：实参数量不足
fact(42, 10, 0);      // 错误：实参数量过多
fact(3.14);           // 正确：该实参能转换成 int 类型
```

因为不能将 const char*转换成 int，所以第一个调用失败。第二个和第三个调用也会失败，不过错误的原因与第一个不同，它们是因为传入的实参数量不对。要想调用 fact 函数只能使用一个实参，只要实参数量不是一个，调用都将失败。最后一个调用是合法的，因为 double 可以转换成 int。执行调用时，实参隐式地转换成 int 类型（截去小数部分），调用等价于

```
fact(3);
```

函数的形参列表

204

函数的形参列表可以为空，但是不能省略。要想定义一个不带形参的函数，最常用的办法是书写一个空的形参列表。不过为了与 C 语言兼容，也可以使用关键字 void 表示函数没有形参：

```
void f1(){ /* ...*/ }          // 隐式地定义空形参列表
void f2(void){ /* ...*/ }      // 显式地定义空形参列表
```

形参列表中的形参通常用逗号隔开，其中每个形参都是含有一个声明符的声明。即使两个形参的类型一样，也必须把两个类型都写出来：

```
int f3(int v1, v2) { /* ...*/ }        // 错误
int f4(int v1, int v2) { /* ...*/ }    // 正确
```

任意两个形参都不能同名，而且函数最外层作用域中的局部变量也不能使用与函数形参一样的名字。

形参名是可选的，但是由于我们无法使用未命名的形参，所以形参一般都应该有个名字。偶尔，函数确实有个别形参不会被用到，则此类形参通常不命名以表示在函数体内不会使用它。不管怎样，是否设置未命名的形参并不影响调用时提供的实参数量。即使某个形参不被函数使用，也必须为它提供一个实参。

函数返回类型

大多数类型都能用作函数的返回类型。一种特殊的返回类型是 void，它表示函数不返回任何值。函数的返回类型不能是数组（参见 3.5 节，第 101 页）类型或函数类型，但可以是指向数组或函数的指针。我们将在 6.3.3 节（第 205 页）介绍如何定义一种特殊的函数，它的返回值是数组的指针（或引用），在 6.7 节（第 221 页）将介绍如何返回指向函数的指针。

6.1 节练习

练习 6.1：实参和形参的区别是什么？

练习 6.2：请指出下列函数哪个有错误，为什么？应该如何修改这些错误呢？

```
(a) int f() {
        string s;
        //...
        return s;
    }
(b) f2(int i) { /* ...*/ }
(c) int calc(int v1, int v1) /* ...*/ }
(d) double square(double x) return x * x;
```

练习 6.3：编写你自己的 fact 函数，上机检查是否正确。

练习 6.4：编写一个与用户交互的函数，要求用户输入一个数字，计算生成该数字的阶乘。在 main 函数中调用该函数。

练习 6.5：编写一个函数输出其实参的绝对值。



6.1.1 局部对象

在 C++语言中，名字有作用域（参见 2.2.4 节，第 43 页），对象有**生命周期**（lifetime）。理解这两个概念非常重要。

- 名字的作用域是程序文本的一部分，名字在其中可见。
- 对象的生命周期是程序执行过程中该对象存在的一段时间。

如我们所知，函数体是一个语句块。块构成一个新的作用域，我们可以在其中定义变量。形参和函数体内部定义的变量统称为**局部变量**（local variable）。它们对函数而言是“局部”的，仅在函数的作用域内可见，同时局部变量还会**隐藏**（hide）在外层作用域中同名的其他所有声明中。

205

在所有函数体之外定义的对象存在于程序的整个执行过程中。此类对象在程序启动时被创建，直到程序结束才会销毁。局部变量的生命周期依赖于定义的方式。

自动对象

对于普通局部变量对应的对象来说，当函数的控制路径经过变量定义语句时创建该对象，当到达定义所在的块末尾时销毁它。我们把只存在于块执行期间的对象称为**自动对象**（automatic object）。当块的执行结束后，块中创建的自动对象的值就变成未定义的了。

形参是一种自动对象。函数开始时为形参申请存储空间，因为形参定义在函数体作用域之内，所以一旦函数终止，形参也就被销毁。

我们用传递给函数的实参初始化形参对应的自动对象。对于局部变量对应的自动对象来说，则分为两种情况：如果变量定义本身含有初始值，就用这个初始值进行初始化；否则，如果变量定义本身不含初始值，执行默认初始化（参见 2.2.1 节，第 40 页）。这意味着内置类型的未初始化局部变量将产生未定义的值。

局部静态对象

某些时候，有必要令局部变量的生命周期贯穿函数调用及之后的时间。可以将局部变量定义成 `static` 类型从而获得这样的对象。**局部静态对象**（local static object）在程序的执行路径第一次经过对象定义语句时初始化，并且直到程序终止才被销毁，在此期间即使对象所在的函数结束执行也不会对它有影响。 206

举个例子，下面的函数统计它自己被调用了多少次，这样的函数也许没什么实际意义，但是足够说明问题：

```
size_t count_calls()
{
    static size_t ctr = 0; // 调用结束后，这个值仍然有效
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

这段程序将输出从 1 到 10（包括 10 在内）的数字。

在控制流第一次经过 `ctr` 的定义之前，`ctr` 被创建并初始化为 0。每次调用将 `ctr` 加 1 并返回新值。每次执行 `count_calls` 函数时，变量 `ctr` 的值都已经存在并且等于函数上一次退出时 `ctr` 的值。因此，第二次调用时 `ctr` 的值是 1，第三次调用时 `ctr` 的值是 2，以此类推。

如果局部静态变量没有显式的初始值，它将执行值初始化（参见 3.3.1 节，第 88 页），内置类型的局部静态变量初始化为 0。

6.1.1 节练习

练习 6.6：说明形参、局部变量以及局部静态变量的区别。编写一个函数，同时用到这三种形式。

练习 6.7：编写一个函数，当它第一次被调用时返回 0，以后每次被调用返回值加 1。

6.1.2 函数声明

和其他名字一样，函数的名字也必须在使用之前声明。类似于变量（参见 2.2.2 节，第 41 页），函数只能定义一次，但可以声明多次。唯一的例外是如 15.3 节（第 535 页）将要介绍的，如果一个函数永远也不会被我们用到，那么它可以只有声明没有定义。

函数的声明和函数的定义非常类似，唯一的区别是函数声明无须函数体，用一个分号替代即可。

因为函数的声明不包含函数体，所以也就无须形参的名字。事实上，在函数的声明中经常省略形参的名字。尽管如此，写上形参的名字还是有用处的，它可以帮助使用者更好地理解函数的功能：

207

```
// 我们选择 beg 和 end 作为形参的名字以表示这两个迭代器划定了输出值的范围
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end);
```

函数的三要素（返回类型、函数名、形参类型）描述了函数的接口，说明了调用该函数所需的全部信息。函数声明也称作**函数原型**（function prototype）。

在头文件中进行函数声明

回忆之前所学的知识，我们建议变量在头文件（参见 2.6.3 节，第 68 页）中声明，在源文件中定义。与之类似，函数也应该在头文件中声明而在源文件中定义。

看起来把函数的声明直接放在使用该函数的源文件中是合法的，也比较容易被人接受；但是这么做可能会很烦琐而且容易出错。相反，如果把函数声明放在头文件中，就能确保同一函数的所有声明保持一致。而且一旦我们想改变函数的接口，只需改变一条声明即可。

定义函数的源文件应该把含有函数声明的头文件包含进来，编译器负责验证函数的定义和声明是否匹配。



含有函数声明的头文件应该被包含到定义函数的源文件中。

6.1.2 节练习

练习 6.8：编写一个名为 `Chapter6.h` 的头文件，令其包含 6.1 节练习（第 184 页）中的函数声明。

6.1.3 分离式编译

随着程序越来越复杂，我们希望把程序的各个部分分别存储在不同文件中。例如，可以把 6.1 节练习（第 184 页）的函数存在一个文件里，把使用这些函数的代码存在其他源文件中。为了允许编写程序时按照逻辑关系将其划分开来，C++语言支持所谓的**分离式编译**（separate compilation）。分离式编译允许我们把程序分割到几个文件中去，每个文件独立编译。

编译和链接多个源文件

举个例子，假设 `fact` 函数的定义位于一个名为 `fact.cc` 的文件中，它的声明位于

名为 Chapter6.h 的头文件中。显然与其他所有用到 fact 函数的文件一样，fact.cc 应该包含 Chapter6.h 头文件。另外，我们在名为 factMain.cc 的文件中创建 main 函数，main 函数将调用 fact 函数。要生成**可执行文件**（executable file），必须告诉编译器我们用到的代码在哪里。对于上述几个文件来说，编译的过程如下所示：

208

```
$ CC factMain.cc fact.cc # generates factMain.exe or a.out
$ CC factMain.cc fact.cc -o main # generates main or main.exe
```

其中，CC 是编译器的名字，$是系统提示符，#后面是命令行下的注释语句。接下来运行可执行文件，就会执行我们定义的 main 函数。

如果我们修改了其中一个源文件，那么只需重新编译那个改动了的文件。大多数编译器提供了分离式编译每个文件的机制，这一过程通常会产生一个后缀名是.obj（Windows）或.o（UNIX）的文件，后缀名的含义是该文件包含**对象代码**（object code）。

接下来编译器负责把对象文件**链接**在一起形成可执行文件。在我们的系统中，编译的过程如下所示：

```
$ CC -c factMain.cc # generates factMain.o
$ CC -c fact.cc # generates fact.o
$ CC factMain.o fact.o # generates factMain.exe or a.out
$ CC factMain.o fact.o -o main # generates main or main.exe
```

你可以仔细阅读编译器的用户手册，弄清楚由多个文件组成的程序是如何编译并执行的。

6.1.3 节练习

练习 6.9：编写你自己的 fact.cc 和 factMain.cc，这两个文件都应该包含上一小节的练习中编写的 Chapter6.h 头文件。通过这些文件，理解你的编译器是如何支持分离式编译的。

6.2 参数传递



如前所述，每次调用函数时都会重新创建它的形参，并用传入的实参对形参进行初始化。



形参初始化的机理与变量初始化一样。

和其他变量一样，形参的类型决定了形参和实参交互的方式。如果形参是引用类型（参见 2.3.1 节，第 45 页），它将绑定到对应的实参上；否则，将实参的值拷贝后赋给形参。

当形参是引用类型时，我们说它对应的实参被**引用传递**（passed by reference）或者函数被**传引用调用**（called by reference）。和其他引用一样，引用形参也是它绑定的对象的别名；也就是说，引用形参是它对应的实参的别名。

当实参的值被拷贝给形参时，形参和实参是两个相互独立的对象。我们说这样的实参被**值传递**（passed by value）或者函数被**传值调用**（called by value）。

209

6.2.1 传值参数



当初始化一个非引用类型的变量时，初始值被拷贝给变量。此时，对变量的改动不会

影响初始值：

```
int n = 0;          // int 类型的初始变量
int i = n;          // i 是 n 的值的副本
i = 42;             // i 的值改变；n 的值不变
```

传值参数的机理完全一样，函数对形参做的所有操作都不会影响实参。例如，在 fact 函数（参见 6.1 节，第 182 页）内对变量 val 执行递减操作：

```
ret *= val--;       // 将 val 的值减 1
```

尽管 fact 函数改变了 val 的值，但是这个改动不会影响传入 fact 的实参。调用 fact(i) 不会改变 i 的值。

指针形参

指针的行为和其他非引用类型一样。当执行指针拷贝操作时，拷贝的是指针的值。拷贝之后，两个指针是不同的指针。因为指针使我们可以间接地访问它所指的对象，所以通过指针可以修改它所指对象的值：

```
int n = 0, i = 42;
int *p = &n, *q = &i;     // p 指向 n；q 指向 i
*p = 42;                  // n 的值改变；p 不变
p = q;                    // p 现在指向了 i；但是 i 和 n 的值都不变
```

指针形参的行为与之类似：

```
// 该函数接受一个指针，然后将指针所指的值置为 0
void reset(int *ip)
{
    *ip = 0;      // 改变指针 ip 所指对象的值
    ip = 0;       // 只改变了 ip 的局部拷贝，实参未被改变
}
```

调用 reset 函数之后，实参所指的对象被置为 0，但是实参本身并没有改变：

```
int i = 42;
reset(&i);                          // 改变 i 的值而非 i 的地址
cout << "i = " << i << endl;        // 输出 i = 0
```

210 Best Practices 熟悉 C 的程序员常常使用指针类型的形参访问函数外部的对象。在 C++语言中，建议使用引用类型的形参替代指针。

6.2.1 节练习

练习 6.10：编写一个函数，使用指针形参交换两个整数的值。在代码中调用该函数并输出交换后的结果，以此验证函数的正确性。

6.2.2 传引用参数

回忆过去所学的知识，我们知道对于引用的操作实际上是作用在引用所引的对象上（参见 2.3.1 节，第 45 页）：

```
int n = 0, i = 42;
int &r = n;       // r 绑定了 n（即 r 是 n 的另一个名字）
```

```
r = 42;                    // 现在 n 的值是 42
r = i;                     // 现在 n 的值和 i 相同
i = r;                     // i 的值和 n 相同
```

引用形参的行为与之类似。通过使用引用形参，允许函数改变一个或多个实参的值。

举个例子，我们可以改写上一小节的 reset 程序，使其接受的参数是引用类型而非指针：

```
//该函数接受一个 int 对象的引用，然后将对象的值置为 0
void reset(int &i)    // i 是传给 reset 函数的对象的另一个名字
{
    i = 0;            // 改变了 i 所引对象的值
}
```

和其他引用一样，引用形参绑定初始化它的对象。当调用这一版本的 reset 函数时，i 绑定我们传给函数的 int 对象，此时改变 i 也就是改变 i 所引对象的值。此例中，被改变的对象是传入 reset 的实参。

调用这一版本的 reset 函数时，我们直接传入对象而无须传递对象的地址：

```
int j = 42;
reset(j);                          // j 采用传引用方式，它的值被改变
cout << "j = " << j << endl;       // 输出 j = 0
```

在上述调用过程中，形参 i 仅仅是 j 的又一个名字。在 reset 内部对 i 的使用即是对 j 的使用。

使用引用避免拷贝

211

拷贝大的类类型对象或者容器对象比较低效，甚至有的类类型（包括 IO 类型在内）根本就不支持拷贝操作。当某种类型不支持拷贝操作时，函数只能通过引用形参访问该类型的对象。

举个例子，我们准备编写一个函数比较两个 string 对象的长度。因为 string 对象可能会非常长，所以应该尽量避免直接拷贝它们，这时使用引用形参是比较明智的选择。又因为比较长度无须改变 string 对象的内容，所以把形参定义成对常量的引用（参见 2.4.1 节，第 54 页）：

```
// 比较两个 string 对象的长度
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

如 6.2.3 节（第 191 页）将要介绍的，当函数无须修改引用形参的值时最好使用常量引用。

Best Practices 如果函数无须改变引用形参的值，最好将其声明为常量引用。

使用引用形参返回额外信息

一个函数只能返回一个值，然而有时函数需要同时返回多个值，引用形参为我们一次返回多个结果提供了有效的途径。举个例子，我们定义一个名为 find_char 的函数，它返回在 string 对象中某个指定字符第一次出现的位置。同时，我们也希望函数能返回该

字符出现的总次数。

该如何定义函数使得它能够既返回位置也返回出现次数呢？一种方法是定义一个新的数据类型，让它包含位置和数量两个成员。还有另一种更简单的方法，我们可以给函数传入一个额外的引用实参，令其保存字符出现的次数：

```
// 返回 s 中 c 第一次出现的位置索引
// 引用形参 occurs 负责统计 c 出现的总次数
string::size_type find_char(const string &s, char c,
                            string::size_type &occurs)
{
    auto ret = s.size();          // 第一次出现的位置（如果有的话）
    occurs = 0;                   // 设置表示出现次数的形参的值
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;          // 记录 c 第一次出现的位置
            ++occurs;             // 将出现的次数加 1
        }
    }
    return ret;                   // 出现次数通过 occurs 隐式地返回
}
```

212 当我们调用 find_char 函数时，必须传入三个实参：作为查找范围的一个 string 对象、要找的字符以及一个用于保存字符出现次数的 size_type（参见 3.2.2 节，第 79 页）对象。假设 s 是一个 string 对象，ctr 是一个 size_type 对象，则我们通过如下形式调用 find_char 函数：

```
auto index = find_char(s, 'o', ctr);
```

调用完成后，如果 string 对象中确实存在 o，那么 ctr 的值就是 o 出现的次数，index 指向 o 第一次出现的位置；否则如果 string 对象中没有 o，index 等于 s.size() 而 ctr 等于 0。

6.2.2 节练习

练习 6.11：编写并验证你自己的 reset 函数，使其作用于引用类型的参数。

练习 6.12：改写 6.2.1 节中练习 6.10（第 188 页）的程序，使用引用而非指针交换两个整数的值。你觉得哪种方法更易于使用呢？为什么？

练习 6.13：假设 T 是某种类型的名字，说明以下两个函数声明的区别：一个是 void f(T)，另一个是 void f(&T)。

练习 6.14：举一个形参应该是引用类型的例子，再举一个形参不能是引用类型的例子。

练习 6.15：说明 find_char 函数中的三个形参为什么是现在的类型，特别说明为什么 s 是常量引用而 occurs 是普通引用？为什么 s 和 occurs 是引用类型而 c 不是？如果令 s 是普通引用会发生什么情况？如果令 occurs 是常量引用会发生什么情况？



6.2.3 const 形参和实参

当形参是 const 时，必须要注意 2.4.3 节（第 57 页）关于顶层 const 的讨论。如前

所述，顶层 const 作用于对象本身：

```
const int ci = 42;    // 不能改变 ci，const 是顶层的
int i = ci;           // 正确：当拷贝 ci 时，忽略了它的顶层 const
int * const p = &i;   // const 是顶层的，不能给 p 赋值
*p = 0;               // 正确：通过 p 改变对象的内容是允许的，现在 i 变成了 0
```

和其他初始化过程一样，当用实参初始化形参时会忽略掉顶层 const。换句话说，形参的顶层 const 被忽略掉了。当形参有顶层 const 时，传给它常量对象或者非常量对象都是可以的：

```
void fcn(const int i) { /* fcn 能够读取 i，但是不能向 i 写值*/ }
```

调用 fcn 函数时，既可以传入 const int 也可以传入 int。忽略掉形参的顶层 const 可能产生意想不到的结果：

```
void fcn(const int i) { /* fcn 能够读取 i，但是不能向 i 写值 */ }
void fcn(int i) { /* ...*/ } // 错误：重复定义了 fcn(int)
```

213

在 C++语言中，允许我们定义若干具有相同名字的函数，不过前提是不同函数的形参列表应该有明显的区别。因为顶层 const 被忽略掉了，所以在上面的代码中传入两个 fcn 函数的参数可以完全一样。因此第二个 fcn 是错误的，尽管形式上有差异，但实际上它的形参和第一个 fcn 的形参没什么不同。

指针或引用形参与 const

形参的初始化方式和变量的初始化方式是一样的，所以回顾通用的初始化规则有助于理解本节知识。我们可以使用非常量初始化一个底层 const 对象，但是反过来不行；同时一个普通的引用必须用同类型的对象初始化。

```
int i = 42;
const int *cp = &i; // 正确：但是 cp 不能改变 i（参见 2.4.2 节，第 56 页）
const int &r = i;   // 正确：但是 r 不能改变 i（参见 2.4.1 节，第 55 页）
const int &r2 = 42; // 正确：（参见 2.4.1 节，第 55 页）
int *p = cp;        // 错误：p 的类型和 cp 的类型不匹配（参见 2.4.2 节，第 56 页）
int &r3 = r;        // 错误：r3 的类型和 r 的类型不匹配（参见 2.4.1 节，第 55 页）
int &r4 = 42;       // 错误：不能用字面值初始化一个非常量引用（参见 2.3.1 节，第 45 页）
```

将同样的初始化规则应用到参数传递上可得如下形式：

```
int i = 0;
const int ci = i;
string::size_type ctr = 0;
reset(&i);          // 调用形参类型是 int*的 reset 函数
reset(&ci);         // 错误：不能用指向 const int 对象的指针初始化 int*
reset(i);           // 调用形参类型是 int&的 reset 函数
reset(ci);          // 错误：不能把普通引用绑定到 const 对象 ci 上
reset(42);          // 错误：不能把普通应用绑定到字面值上
reset(ctr);         // 错误：类型不匹配，ctr 是无符号类型
// 正确：find_char 的第一个形参是对常量的引用
find_char("Hello World!", 'o', ctr);
```

要想调用引用版本的 reset（参见 6.2.2 节，第 189 页），只能使用 int 类型的对象，而不能使用字面值、求值结果为 int 的表达式、需要转换的对象或者 const int 类型的对象。类似的，要想调用指针版本的 reset（参见 6.2.1 节，第 188 页）只能使用 int*。

另一方面，我们能传递一个字符串字面值作为 find_char（参见 6.2.2 节，第 189 页）的第一个实参，这是因为该函数的引用形参是常量引用，而 C++允许我们用字面值初始化常量引用。

尽量使用常量引用

把函数不会改变的形参定义成（普通的）引用是一种比较常见的错误，这么做带给函数的调用者一种误导，即函数可以修改它的实参的值。此外，使用引用而非常量引用也会极大地限制函数所能接受的实参类型。就像刚刚看到的，我们不能把 const 对象、字面值或者需要类型转换的对象传递给普通的引用形参。

214

这种错误绝不像看起来那么简单，它可能造成出人意料的后果。以 6.2.2 节（第 189 页）的 find_char 函数为例，那个函数（正确地）将它的 string 类型的形参定义成常量引用。假如我们把它定义成普通的 string&：

```
//不良设计：第一个形参的类型应该是 const string&
string::size_type find_char(string &s, char c,
                            string::size_type &occurs);
```

则只能将 find_char 函数作用于 string 对象。类似下面这样的调用

```
find_char("Hello World", 'o', ctr);
```

将在编译时发生错误。

还有一个更难察觉的问题，假如其他函数（正确地）将它们的形参定义成常量引用，那么第二个版本的 find_char 无法在此类函数中正常使用。举个例子，我们希望在一个判断 string 对象是否是句子的函数中使用 find_char：

```
bool is_sentence(const string &s)
{
    // 如果在 s 的末尾有且只有一个句号，则 s 是一个句子
    string::size_type ctr = 0;
    return find_char(s, '.', ctr) == s.size() - 1 && ctr == 1;
}
```

如果 find_char 的第一个形参类型是 string&，那么上面这条调用 find_char 的语句将在编译时发生错误。原因在于 s 是常量引用，但 find_char 被（不正确地）定义成只能接受普通引用。

解决该问题的一种思路是修改 is_sentence 的形参类型，但是这么做只不过转移了错误而已，结果是 is_sentence 函数的调用者只能接受非常量 string 对象了。

正确的修改思路是改正 find_char 函数的形参。如果实在不能修改 find_char，就在 is_sentence 内部定义一个 string 类型的变量，令其为 s 的副本，然后把这个 string 对象传递给 find_char。

6.2.3 节练习

练习 6.16：下面的这个函数虽然合法，但是不算特别有用。指出它的局限性并设法改善。

```
bool is_empty(string& s) { return s.empty(); }
```

练习 6.17：编写一个函数，判断 string 对象中是否含有大写字母。编写另一个函数，把 string 对象全都改成小写形式。在这两个函数中你使用的形参类型相同吗？为什么？

练习 6.18：为下面的函数编写函数声明，从给定的名字中推测函数具备的功能。

(a) 名为 compare 的函数，返回布尔值，两个参数都是 matrix 类的引用。
(b) 名为 change_val 的函数，返回 vector<int>的迭代器，有两个参数：一个是 int，另一个是 vector<int>的迭代器。

练习 6.19：假定有如下声明，判断哪个调用合法、哪个调用不合法。对于不合法的函数调用，说明原因。

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
(a) calc(23.4, 55.1);        (b) count("abcda", 'a');
(c) calc(66);                (d) sum(vec.begin(), vec.end(), 3.8);
```

练习 6.20：引用形参什么时候应该是常量引用？如果形参应该是常量引用，而我们将其设为了普通引用，会发生什么情况？

6.2.4 数组形参

数组的两个特殊性质对我们定义和使用作用在数组上的函数有影响，这两个性质分别是：不允许拷贝数组（参见 3.5.1 节，第 102 页）以及使用数组时（通常）会将其转换成指针（参见 3.5.3 节，第 105 页）。因为不能拷贝数组，所以我们无法以值传递的方式使用数组参数。因为数组会被转换成指针，所以当我们为函数传递一个数组时，实际上传递的是指向数组首元素的指针。

尽管不能以值传递的方式传递数组，但是我们可以把形参写成类似数组的形式：

```
// 尽管形式不同，但这三个 print 函数是等价的                                    <215
// 每个函数都有一个 const int*类型的形参
void print(const int*);
void print(const int[]);      // 可以看出来，函数的意图是作用于一个数组
void print(const int[10]);    // 这里的维度表示我们期望数组含有多少元素，实际
                              // 不一定
```

尽管表现形式不同，但上面的三个函数是等价的：每个函数的唯一形参都是 const int* 类型的。当编译器处理对 print 函数的调用时，只检查传入的参数是否是 const int* 类型：

```
int i = 0, j[2] = {0, 1};
print(&i);                    // 正确：&i 的类型是 int*
print(j);                     // 正确：j 转换成 int*并指向 j[0]
```

如果我们传给 print 函数的是一个数组，则实参自动地转换成指向数组首元素的指针，数组的大小对函数的调用没有影响。



和其他使用数组的代码一样，以数组作为形参的函数也必须确保使用数组时不会越界。

因为数组是以指针的形式传递给函数的，所以一开始函数并不知道数组的确切尺寸，调用者应该为此提供一些额外的信息。管理指针形参有三种常用的技术。 <216

使用标记指定数组长度

管理数组实参的第一种方法是要求数组本身包含一个结束标记，使用这种方法的典型示例是C风格字符串（参见3.5.4节，第109页）。C风格字符串存储在字符数组中，并且在最后一个字符后面跟着一个空字符。函数在处理C风格字符串时遇到空字符停止：

```
void print(const char *cp)
{
    if (cp)                    // 若cp不是一个空指针
        while (*cp)            // 只要指针所指的字符不是空字符
            cout << *cp++;     // 输出当前字符并将指针向前移动一个位置
}
```

这种方法适用于那些有明显结束标记且该标记不会与普通数据混淆的情况，但是对于像int这样所有取值都是合法值的数据就不太有效了。

使用标准库规范

管理数组实参的第二种技术是传递指向数组首元素和尾后元素的指针，这种方法受到了标准库技术的启发，关于其细节将在第II部分详细介绍。使用该方法，我们可以按照如下形式输出元素内容：

```
void print(const int *beg, const int *end)
{
    // 输出beg到end之间（不含end）的所有元素
    while (beg != end)
        cout << *beg++ << endl; // 输出当前元素并将指针向前移动一个位置
}
```

while循环使用解引用运算符和后置递减运算符（参见4.5节，第131页）输出当前元素并在数组内将beg向前移动一个元素，当beg和end相等时结束循环。

为了调用这个函数，我们需要传入两个指针：一个指向要输出的首元素，另一个指向尾元素的下一位置：

```
int j[2] = {0, 1};
// j转换成指向它首元素的指针
// 第二个实参是指向j的尾后元素的指针
print(begin(j), end(j));       // begin和end函数，参见第3.5.3节（106页）
```

只要调用者能正确地计算指针所指的位置，那么上述代码就是安全的。在这里，我们使用标准库begin和end函数（参见3.5.3节，第106页）提供所需的指针。

217 显式传递一个表示数组大小的形参

第三种管理数组实参的方法是专门定义一个表示数组大小的形参，在C程序和过去的C++程序中常常使用这种方法。使用该方法，可以将print函数重写成如下形式：

```
// const int ia[]等价于const int* ia
// size表示数组的大小，将它显式地传给函数用于控制对ia元素的访问
void print(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
```

这个版本的程序通过形参 size 的值确定要输出多少个元素，调用 print 函数时必须传入这个表示数组大小的值：

```
int j[] = { 0, 1 };  // 大小为 2 的整型数组
print(j, end(j) - begin(j));
```

只要传递给函数的 size 值不超过数组实际的大小，函数就是安全的。

数组形参和 const

我们的三个 print 函数都把数组形参定义成了指向 const 的指针，6.2.3 节（第 191 页）关于引用的讨论同样适用于指针。当函数不需要对数组元素执行写操作的时候，数组形参应该是指向 const 的指针（参见 2.4.2 节，第 56 页）。只有当函数确实要改变元素值的时候，才把形参定义成指向非常量的指针。

数组引用形参

C++语言允许将变量定义成数组的引用（参见 3.5.1 节，第 101 页），基于同样的道理，形参也可以是数组的引用。此时，引用形参绑定到对应的实参上，也就是绑定到数组上：

```
// 正确：形参是数组的引用，维度是类型的一部分
void print(int (&arr)[10])
{
    for (auto elem : arr)
        cout << elem << endl;
}
```



&arr 两端的括号必不可少（参见 3.5.1 节，第 101 页）：

```
f(int &arr[10])      // 错误：将 arr 声明成了引用的数组
f(int (&arr)[10])    // 正确：arr 是具有 10 个整数的整型数组的引用
```

因为数组的大小是构成数组类型的一部分，所以只要不超过维度，在函数体内就可以放心地使用数组。但是，这一用法也无形中限制了 print 函数的可用性，我们只能将函数作用于大小为 10 的数组：

218

```
int i = 0, j[2] = {0, 1};
int k[10] = {0,1,2,3,4,5,6,7,8,9};
print(&i);                  // 错误：实参不是含有 10 个整数的数组
print(j);                   // 错误：实参不是含有 10 个整数的数组
print(k);                   // 正确：实参是含有 10 个整数的数组
```

16.1.1 节（第 578 页）将要介绍我们应该如何编写这个函数，使其可以给引用类型的形参传递任意大小的数组。

传递多维数组

我们曾经介绍过，在 C++语言中实际上没有真正的多维数组（参见 3.6 节，第 112 页），所谓多维数组其实是数组的数组。

和所有数组一样，当将多维数组传递给函数时，真正传递的是指向数组首元素的指针（参见 3.6 节，第 115 页）。因为我们处理的是数组的数组，所以首元素本身就是一个数组，指针就是一个指向数组的指针。数组第二维（以及后面所有维度）的大小都是数组类型的一部分，不能省略：

```
// matrix指向数组的首元素，该数组的元素是由10个整数构成的数组
void print(int (*matrix)[10], int rowSize) { /* ...*/ }
```

上述语句将matrix声明成指向含有10个整数的数组的指针。



再一次强调，*matrix两端的括号必不可少：

```
int *matrix[10];          // 10个指针构成的数组
int (*matrix)[10];        // 指向含有10个整数的数组的指针
```

我们也可以使用数组的语法定义函数，此时编译器会一如既往地忽略掉第一个维度，所以最好不要把它包括在形参列表内：

```
// 等价定义
void print(int matrix[][10], int rowSize) { /* ... */ }
```

matrix的声明看起来是一个二维数组，实际上形参是指向含有10个整数的数组的指针。

6.2.4 节练习

练习6.21：编写一个函数，令其接受两个参数：一个是int型的数，另一个是int指针。函数比较int的值和指针所指的值，返回较大的那个。在该函数中指针的类型应该是什么？

练习6.22：编写一个函数，令其交换两个int指针。

练习6.23：参考本节介绍的几个print函数，根据理解编写你自己的版本。依次调用每个函数使其输入下面定义的i和j：

```
int i = 0, j[2] = {0, 1};
```

练习6.24：描述下面这个函数的行为。如果代码中存在问题，请指出并改正。

```
void print(const int ia[10])
{
    for (size_t i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

6.2.5 main：处理命令行选项

main函数是演示C++程序如何向函数传递数组的好例子。到目前为止，我们定义的main函数都只有空形参列表：

```
int main() { ... }
```

然而，有时我们确实需要给main传递实参，一种常见的情况是用户通过设置一组选项来确定函数所要执行的操作。例如，假定main函数位于可执行文件prog之内，我们可以向程序传递下面的选项：

219

```
prog -d -o ofile data0
```

这些命令行选项通过两个（可选的）形参传递给main函数：

```
int main(int argc, char *argv[]) { ... }
```

第二个形参 argv 是一个数组，它的元素是指向 C 风格字符串的指针；第一个形参 argc 表示数组中字符串的数量。因为第二个形参是数组，所以 main 函数也可以定义成：

```
int main(int argc, char **argv) { ... }
```

其中 argv 指向 char*。

当实参传给 main 函数之后，argv 的第一个元素指向程序的名字或者一个空字符串，接下来的元素依次传递命令行提供的实参。最后一个指针之后的元素值保证为 0。

以上面提供的命令行为例，argc 应该等于 5，argv 应该包含如下的 C 风格字符串：

```
argv[0] = "prog";    // 或者 argv[0]也可以指向一个空字符串
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
```



当使用 argv 中的实参时，一定要记得可选的实参从 argv[1]开始；argv[0]保存程序的名字，而非用户输入。

6.2.5 节练习

220

练习 6.25：编写一个 main 函数，令其接受两个实参。把实参的内容连接成一个 string 对象并输出出来。

练习 6.26：编写一个程序，使其接受本节所示的选项；输出传递给 main 函数的实参的内容。

6.2.6 含有可变形参的函数

有时我们无法提前预知应该向函数传递几个实参。例如，我们想要编写代码输出程序产生的错误信息，此时最好用同一个函数实现该项功能，以便对所有错误的处理能够整齐划一。然而，错误信息的种类不同，所以调用错误输出函数时传递的实参也各不相同。

为了编写能处理不同数量实参的函数，C++11 新标准提供了两种主要的方法：如果所有的实参类型相同，可以传递一个名为 initializer_list 的标准库类型；如果实参的类型不同，我们可以编写一种特殊的函数，也就是所谓的可变参数模板，关于它的细节将在 16.4 节（第 618 页）介绍。

C++还有一种特殊的形参类型（即省略符），可以用它传递可变数量的实参。本节将简要介绍省略符形参，不过需要注意的是，这种功能一般只用于与 C 函数交互的接口程序。

initializer_list 形参

如果函数的实参数量未知但是全部实参的类型都相同，我们可以使用 **initializer_list** 类型的形参。initializer_list 是一种标准库类型，用于表示某种特定类型的值的数组（参见 3.5 节，第 101 页）。initializer_list 类型定义在同名的头文件中，它提供的操作如表 6.1 所示。



表 6.1：initializer_list 提供的操作	
`initializer_list<T> lst;`	默认初始化；T 类型元素的空列表
`initializer_list<T> lst{a,b,c...};`	lst 的元素数量和初始值一样多；lst 的元素是对应初始值的副本；列表中的元素是 const
`lst2(lst)` `lst2 = lst`	拷贝或赋值一个 initializer_list 对象不会拷贝列表中的元素；拷贝后，原始列表和副本共享元素
`lst.size()`	列表中的元素数量
`lst.begin()`	返回指向 lst 中首元素的指针
`lst.end()`	返回指向 lst 中尾元素下一位置的指针

221 和 vector 一样，initializer_list 也是一种模板类型（参见 3.3 节，第 86 页）。定义 initializer_list 对象时，必须说明列表中所含元素的类型：

```
initializer_list<string> ls; // initializer_list 的元素类型是 string
initializer_list<int> li;    // initializer_list 的元素类型是 int
```

和 vector 不一样的是，initializer_list 对象中的元素永远是常量值，我们无法改变 initializer_list 对象中元素的值。

我们使用如下的形式编写输出错误信息的函数，使其可以作用于可变数量的实参：

```
void error_msg(initializer_list<string> il)
{
    for (auto beg = il.begin(); beg != il.end(); ++beg)
        cout << *beg << " " ;
    cout << endl;
}
```

作用于 initializer_list 对象的 begin 和 end 操作类似于 vector 对应的成员（参见 3.4.1 节，第 195 页）。begin()成员提供一个指向列表首元素的指针，end()成员提供一个指向列表尾后元素的指针。我们的函数首先初始化 beg 令其表示首元素，然后依次遍历列表中的每个元素。在循环体中，解引用 beg 以访问当前元素并输出它的值。

如果想向 initializer_list 形参中传递一个值的序列，则必须把序列放在一对花括号内：

```
// expected 和 actual 是 string 对象
if (expected != actual)
    error_msg({"functionX", expected, actual});
else
    error_msg({"functionX", "okay"});
```

在上面的代码中我们调用了同一个函数 error_msg，但是两次调用传递的参数数量不同：第一次调用传入了三个值，第二次调用只传入了两个。

含有 initializer_list 形参的函数也可以同时拥有其他形参。例如，调试系统可能有个名为 ErrCode 的类用来表示不同类型的错误，因此我们可以改写之前的程序，使其包含一个 initializer_list 形参和一个 ErrCode 形参：

```
void error_msg(ErrCode e, initializer_list<string> il)
{
```

```
    cout << e.msg() << ": ";
    for (const auto &elem : il)
        cout << elem << " " ;
    cout << endl;
}
```

因为 initializer_list 包含 begin 和 end 成员，所以我们可以使用范围 for 循环（参见 5.4.3 节，第 167 页）处理其中的元素。和之前的版本类似，这段程序遍历传给 il 形参的列表值，每次迭代时访问一个元素。

为了调用这个版本的 error_msg 函数，需要额外传递一个 ErrCode 实参：

222

```
if (expected != actual)
    error_msg(ErrCode(42), {"functionX", expected, actual});
else
    error_msg(ErrCode(0), {"functionX", "okay"});
```



省略符形参

省略符形参是为了便于 C++程序访问某些特殊的 C 代码而设置的，这些代码使用了名为 varargs 的 C 标准库功能。通常，省略符形参不应用于其他目的。你的 C 编译器文档会描述如何使用 varargs。



省略符形参应该仅仅用于 C 和 C++通用的类型。特别应该注意的是，大多数类类型的对象在传递给省略符形参时都无法正确拷贝。

省略符形参只能出现在形参列表的最后一个位置，它的形式无外乎以下两种：

```
void foo(parm_list, ...);
void foo(...);
```

第一种形式指定了 foo 函数的部分形参的类型，对应于这些形参的实参将会执行正常的类型检查。省略符形参所对应的实参无须类型检查。在第一种形式中，形参声明后面的逗号是可选的。

6.2.6 节练习

练习 6.27：编写一个函数，它的参数是 initializer_list<int>类型的对象，函数的功能是计算列表中所有元素的和。

练习 6.28：在 error_msg 函数的第二个版本中包含 ErrCode 类型的参数，其中循环内的 elem 是什么类型？

练习 6.29：在范围 for 循环中使用 initializer_list 对象时，应该将循环控制变量声明成引用类型吗？为什么？

6.3 返回类型和 return 语句

return 语句终止当前正在执行的函数并将控制权返回到调用该函数的地方。return 语句有两种形式：

```
return;
return expression;
```

6.3.1 无返回值函数

223 没有返回值的 return 语句只能用在返回类型是 void 的函数中。返回 void 的函数不要求非得有 return 语句，因为在这类函数的最后一句后面会隐式地执行 return。

通常情况下，void 函数如果想在它的中间位置提前退出，可以使用 return 语句。return 的这种用法有点类似于我们用 break 语句（参见 5.5.1 节，第 170 页）退出循环。例如，可以编写一个 swap 函数，使其在参与交换的值相等时什么也不做直接退出：

```
void swap(int &v1, int &v2)
{
    // 如果两个值是相等的，则不需要交换，直接退出
    if (v1 == v2)
        return;
    // 如果程序执行到了这里，说明还需要继续完成某些功能
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // 此处无须显式的 return 语句
}
```

这个函数首先检查值是否相等，如果相等直接退出函数；如果不相等才交换它们的值。在最后一条赋值语句后面隐式地执行 return。

一个返回类型是void的函数也能使用return语句的第二种形式，不过此时return语句的 *expression* 必须是另一个返回 void 的函数。强行令 void 函数返回其他类型的表达式将产生编译错误。

6.3.2 有返回值函数

return 语句的第二种形式提供了函数的结果。只要函数的返回类型不是 void，则该函数内的每条 return 语句必须返回一个值。return 语句返回值的类型必须与函数的返回类型相同，或者能隐式地转换成（参见 4.11 节，第 141 页）函数的返回类型。

尽管 C++无法确保结果的正确性，但是可以保证每个 return 语句的结果类型正确。也许无法顾及所有情况，但是编译器仍然尽量确保具有返回值的函数只能通过一条有效的 return 语句退出。例如：

```
// 因为含有不正确的返回值，所以这段代码无法通过编译
bool str_subrange(const string &str1, const string &str2)
{
    // 大小相同：此时用普通的相等性判断结果作为返回值
    if (str1.size() == str2.size())
        return str1 == str2;          // 正确：==运算符返回布尔值
    // 得到较短 string 对象的大小，条件运算符参见第 4.7 节（134 页）
    auto size = (str1.size() < str2.size())
                ? str1.size() : str2.size();
224 // 检查两个 string 对象的对应字符是否相等，以较短的字符串长度为限
    for (decltype(size) i = 0; i != size; ++i) {
        if (str1[i] != str2[i])
            return; // 错误 #1：没有返回值，编译器将报告这一错误
    }
```

```
    // 错误 #2：控制流可能尚未返回任何值就结束了函数的执行
    // 编译器可能检查不出这一错误
}
```

for 循环内的 return 语句是错误的，因为它没有返回值，编译器能检测到这个错误。

第二个错误是函数在 for 循环之后没有提供 return 语句。在上面的程序中，如果一个 string 对象是另一个的子集，则函数在执行完 for 循环后还将继续其执行过程，显然应该有一条 return 语句专门处理这种情况。编译器也许能检测到这个错误，也许不能；如果编译器没有发现这个错误，则运行时的行为将是未定义的。



在含有 return 语句的循环后面应该也有一条 return 语句，如果没有的话该程序就是错误的。很多编译器都无法发现此类错误。

值是如何被返回的

返回一个值的方式和初始化一个变量或形参的方式完全一样：返回的值用于初始化调用点的一个临时量，该临时量就是函数调用的结果。

必须注意当函数返回局部变量时的初始化规则。例如我们书写一个函数，给定计数值、单词和结束符之后，判断计数值是否大于 1：如果是，返回单词的复数形式；如果不是，返回单词原形：

```
// 如果 ctr 的值大于 1，返回 word 的复数形式
string make_plural(size_t ctr, const string &word,
                                const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
```

该函数的返回类型是 string，意味着返回值将被拷贝到调用点。因此，该函数将返回 word 的副本或者一个未命名的临时 string 对象，该对象的内容是 word 和 ending 的和。

同其他引用类型一样，如果函数返回引用，则该引用仅是它所引对象的一个别名。举个例子来说明，假定某函数挑出两个 string 形参中较短的那个并返回其引用：

```
// 挑出两个 string 对象中较短的那个，返回其引用
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

其中形参和返回类型都是 const string 的引用，不管是调用函数还是返回结果都不会真正拷贝 string 对象。 225

不要返回局部对象的引用或指针

函数完成后，它所占用的存储空间也随之被释放掉（参见 6.1.1 节，第 184 页）。因此，函数终止意味着局部变量的引用将指向不再有效的内存区域：

```
// 严重错误：这个函数试图返回局部对象的引用
const string &manip()
{
    string ret;
```

```
    // 以某种方式改变一下 ret
    if (!ret.empty())
        return ret;        // 错误：返回局部对象的引用！
    else
        return "Empty";  // 错误："Empty"是一个局部临时量
}
```

上面的两条 return 语句都将返回未定义的值，也就是说，试图使用 manip 函数的返回值将引发未定义的行为。对于第一条 return 语句来说，显然它返回的是局部对象的引用。在第二条 return 语句中，字符串字面值转换成一个局部临时 string 对象，对于 manip 来说，该对象和 ret 一样都是局部的。当函数结束时临时对象占用的空间也就随之释放掉了，所以两条 return 语句都指向了不再可用的内存空间。



要想确保返回值安全，我们不妨提问：引用所引的是在函数之前已经存在的哪个对象？

如前所述，返回局部对象的引用是错误的；同样，返回局部对象的指针也是错误的。一旦函数完成，局部对象被释放，指针将指向一个不存在的对象。

返回类类型的函数和调用运算符

和其他运算符一样，调用运算符也有优先级和结合律（参见 4.1.2 节，第 121 页）。调用运算符的优先级与点运算符和箭头运算符（参见 4.6 节，第 133 页）相同，并且也符合左结合律。因此，如果函数返回指针、引用或类的对象，我们就能使用函数调用的结果访问结果对象的成员。

例如，我们可以通过如下形式得到较短 string 对象的长度：

```
// 调用 string 对象的 size 成员，该 string 对象是由 shorterString 函数返回的
auto sz = shorterString(s1, s2).size();
```

因为上面提到的运算符都满足左结合律，所以 shorterString 的结果是点运算符的左侧运算对象，点运算符可以得到该 string 对象的 size 成员，size 又是第二个调用运算符的左侧运算对象。

226 引用返回左值

函数的返回类型决定函数调用是否是左值（参见 4.1.1 节，第 121 页）。调用一个返回引用的函数得到左值，其他返回类型得到右值。可以像使用其他左值那样来使用返回引用的函数的调用，特别是，我们能为返回类型是非常量引用的函数的结果赋值：

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix];                 // get_val 假定索引值是有效的
}
int main()
{
    string s("a value");
    cout << s << endl;              // 输出 a value
    get_val(s, 0) = 'A';            // 将 s[0]的值改为 A
    cout << s << endl;              // 输出 A value
```

```
    return 0;
}
```

把函数调用放在赋值语句的左侧可能看起来有点奇怪，但其实这没什么特别的。返回值是引用，因此调用是个左值，和其他左值一样它也能出现在赋值运算符的左侧。

如果返回类型是常量引用，我们不能给调用的结果赋值，这一点和我们熟悉的情况是一样的：

```
shorterString("hi", "bye") = "X"; // 错误：返回值是个常量
```

列表初始化返回值

C++11 新标准规定，函数可以返回花括号包围的值的列表。类似于其他返回结果，此处的列表也用来对表示函数返回的临时量进行初始化。如果列表为空，临时量执行值初始化（参见 3.3.1 节，第 88 页）；否则，返回的值由函数的返回类型决定。

C++ 11

举个例子，回忆 6.2.6 节（第 198 页）的 error_msg 函数，该函数的输入是一组可变数量的 string 实参，输出由这些 string 对象组成的错误信息。在下面的函数中，我们返回一个 vector 对象，用它存放表示错误信息的 string 对象：

```
vector<string> process()
{
    // ...
    // expected 和 actual 是 string 对象
    if (expected.empty())
        return {};                          // 返回一个空 vector 对象
    else if (expected == actual)
        return {"functionX", "okay"};       // 返回列表初始化的 vector 对象
    else
        return {"functionX", expected, actual};
}
```

第一条 return 语句返回一个空列表，此时，process 函数返回的 vector 对象是空的。如果 expected 不为空，根据 expected 和 actual 是否相等，函数返回的 vector 对象分别用两个或三个元素初始化。

227

如果函数返回的是内置类型，则花括号包围的列表最多包含一个值，而且该值所占空间不应该大于目标类型的空间（参见 2.2.1 节，第 39 页）。如果函数返回的是类类型，由类本身定义初始值如何使用（参见 3.3.1 节，第 89 页）。

主函数 main 的返回值

之前介绍过，如果函数的返回类型不是 void，那么它必须返回一个值。但是这条规则有个例外：我们允许 main 函数没有 return 语句直接结束。如果控制到达了 main 函数的结尾处而且没有 return 语句，编译器将隐式地插入一条返回 0 的 return 语句。

如 1.1 节（第 2 页）介绍的，main 函数的返回值可以看做是状态指示器。返回 0 表示执行成功，返回其他值表示执行失败，其中非 0 值的具体含义依机器而定。为了使返回值与机器无关，cstdlib 头文件定义了两个预处理变量（参见 2.3.2 节，第 49 页），我们可以使用这两个变量分别表示成功与失败：

```
int main()
{
    if (some_failure)
```

```
        return EXIT_FAILURE;      // 定义在 cstdlib 头文件中
    else
        return EXIT_SUCCESS;      // 定义在 cstdlib 头文件中
}
```

因为它们是预处理变量，所以既不能在前面加上 std::，也不能在 using 声明中出现。

递归

如果一个函数调用了它自身，不管这种调用是直接的还是间接的，都称该函数为*递归函数*（recursive function）。举个例子，我们可以使用递归函数重新实现求阶乘的功能：

```
// 计算 val 的阶乘，即 1 * 2 * 3 ...* val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

在上面的代码中，我们递归地调用 factorial 函数以求得从 val 中减去 1 后新数字的阶乘。当 val 递减到 1 时，递归终止，返回 1。

在递归函数中，一定有某条路径是不包含递归调用的；否则，函数将“永远”递归下去，换句话说，函数将不断地调用它自身直到程序栈空间耗尽为止。我们有时候会说这种
228 函数含有**递归循环**（recursion loop）。在 factorial 函数中，递归终止的条件是 val 等于 1。

下面的表格显示了当给 factorial 函数传入参数 5 时，函数的执行轨迹。

factorial(5)的执行轨迹

调用	返回	值
factorial(5)	factorial(4) * 5	120
factorial(4)	factorial(3) * 4	24
factorial(3)	factorial(2) * 3	6
factorial(2)	factorial(1) * 2	2
factorial(1)	1	1



main 函数不能调用它自己。

6.3.2 节练习

练习 6.30：编译第 200 页的 str_subrange 函数，看看你的编译器是如何处理函数中的错误的。

练习 6.31：什么情况下返回的引用无效？什么情况下返回常量的引用无效？

练习 6.32：下面的函数合法吗？如果合法，说明其功能；如果不合法，修改其中的错误并解释原因。

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
```

```
        int ia[10];
        for (int i = 0; i != 10; ++i)
            get(ia, i) = i;
    }
```

练习 6.33：编写一个递归函数，输出 vector 对象的内容。

练习 6.34：如果 factorial 函数的停止条件如下所示，将发生什么情况？

```
if (val != 0)
```

练习 6.35：在调用 factorial 函数时，为什么我们传入的值是 val-1 而非 val--？

6.3.3 返回数组指针

因为数组不能被拷贝，所以函数不能返回数组。不过，函数可以返回数组的指针或引用（参见 3.5.1 节，第 102 页）。虽然从语法上来说，要想定义一个返回数组的指针或引用的函数比较烦琐，但是有一些方法可以简化这一任务，其中最直接的方法是使用类型别名（参见 2.5.1 节，第 60 页）：

229

```
typedef int arrT[10];     // arrT 是一个类型别名，它表示的类型是含有 10 个
                          // 整数的数组
using arrT = int[10];     // arrT 的等价声明，参见 2.5.1 节（第 60 页）
arrT* func(int i);        // func 返回一个指向含有 10 个整数的数组的指针
```

其中 arrT 是含有 10 个整数的数组的别名。因为我们无法返回数组，所以将返回类型定义成数组的指针。因此，func 函数接受一个 int 实参，返回一个指向包含 10 个整数的数组的指针。

声明一个返回数组指针的函数

要想在声明 func 时不使用类型别名，我们必须牢记被定义的名字后面数组的维度：

```
int arr[10];              // arr 是一个含有 10 个整数的数组
int *p1[10];              // p1 是一个含有 10 个指针的数组
int (*p2)[10] = &arr;     // p2 是一个指针，它指向含有 10 个整数的数组
```

和这些声明一样，如果我们想定义一个返回数组指针的函数，则数组的维度必须跟在函数名字之后。然而，函数的形参列表也跟在函数名字后面且形参列表应该先于数组的维度。因此，返回数组指针的函数形式如下所示：

Type (**function* (*parameter_list*)) [*dimension*]

类似于其他数组的声明，*Type* 表示元素的类型，*dimension* 表示数组的大小。(**function*(*parameter_list*))两端的括号必须存在，就像我们定义 p2 时两端必须有括号一样。如果没有这对括号，函数的返回类型将是指针的数组。

举个具体点的例子，下面这个 func 函数的声明没有使用类型别名：

```
int (*func(int i))[10];
```

可以按照以下的顺序来逐层理解该声明的含义：

- func(int i)表示调用 func 函数时需要一个 int 类型的实参。
- (*func(int i))意味着我们可以对函数调用的结果执行解引用操作。
- (*func(int i))[10]表示解引用 func 的调用将得到一个大小是 10 的数组。

- int (*func(int i))[10]表示数组中的元素是int类型。

使用尾置返回类型

C++11 在C++11新标准中还有一种可以简化上述func声明的方法，就是使用**尾置返回类型**（trailing return type）。任何函数的定义都能使用尾置返回，但是这种形式对于返回类型比较复杂的函数最有效，比如返回类型是数组的指针或者数组的引用。尾置返回类型跟在形参列表后面并以一个->符号开头。为了表示函数真正的返回类型跟在形参列表之后，我们在本应该出现返回类型的地方放置一个auto：

230

```
// func接受一个int类型的实参，返回一个指针，该指针指向含有10个整数的数组
auto func(int i) -> int(*)[10];
```

因为我们把函数的返回类型放在了形参列表之后，所以可以清楚地看到func函数返回的是一个指针，并且该指针指向了含有10个整数的数组。

使用decltype

还有一种情况，如果我们知道函数返回的指针将指向哪个数组，就可以使用decltype关键字声明返回类型。例如，下面的函数返回一个指针，该指针根据参数i的不同指向两个已知数组中的某一个：

```
int odd[] = {1,3,5,7,9};
int even[] = {0,2,4,6,8};
// 返回一个指针，该指针指向含有5个整数的数组
decltype(odd) *arrPtr(int i)
{
    return (i % 2) ? &odd : &even; // 返回一个指向数组的指针
}
```

C++11 arrPtr使用关键字decltype表示它的返回类型是个指针，并且该指针所指的对象与odd的类型一致。因为odd是数组，所以arrPtr返回一个指向含有5个整数的数组的指针。有一个地方需要注意：decltype并不负责把数组类型转换成对应的指针，所以decltype的结果是个数组，要想表示arrPtr返回指针还必须在函数声明时加一个*符号。

6.3.3 节练习

练习6.36：编写一个函数的声明，使其返回数组的引用并且该数组包含10个string对象。不要使用尾置返回类型、decltype或者类型别名。

练习6.37：为上一题的函数再写三个声明，一个使用类型别名，另一个使用尾置返回类型，最后一个使用decltype关键字。你觉得哪种形式最好？为什么？

练习6.38：修改arrPtr函数，使其返回数组的引用。



6.4 函数重载

如果同一作用域内的几个函数名字相同但形参列表不同，我们称之为**重载**（overloaded）**函数**。例如，在6.2.4节（第193页）中我们定义了几个名为print的函数：

```
void print(const char *cp);
```

```
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

这些函数接受的形参类型不一样，但是执行的操作非常类似。当调用这些函数时，编译器会根据传递的实参类型推断想要的是哪个函数：

231

```
int j[2] = {0,1};
print("Hello World");          // 调用 print(const char*)
print(j, end(j) - begin(j)); // 调用 print(const int*, size_t)
print(begin(j), end(j));      // 调用 print(const int*, const int*)
```

函数的名字仅仅是让编译器知道它调用的是哪个函数，而函数重载可以在一定程度上减轻程序员起名字、记名字的负担。



main 函数不能重载。

定义重载函数

有一种典型的数据库应用，需要创建几个不同的函数分别根据名字、电话、账户号码等信息查找记录。函数重载使得我们可以定义一组函数，它们的名字都是 lookup，但是查找的依据不同。我们能通过以下形式中的任意一种调用 lookup 函数：

```
Record lookup(const Account&);        // 根据 Account 查找记录
Record lookup(const Phone&);          // 根据 Phone 查找记录
Record lookup(const Name&);           // 根据 Name 查找记录
Account acct;
Phone phone;
Record r1 = lookup(acct);             // 调用接受 Account 的版本
Record r2 = lookup(phone);            // 调用接受 Phone 的版本
```

其中，虽然我们定义的三个函数各不相同，但它们都有同一个名字。编译器根据实参的类型确定应该调用哪一个函数。

对于重载的函数来说，它们应该在形参数量或形参类型上有所不同。在上面的代码中，虽然每个函数都只接受一个参数，但是参数的类型不同。

不允许两个函数除了返回类型外其他所有的要素都相同。假设有两个函数，它们的形参列表一样但是返回类型不同，则第二个函数的声明是错误的：

```
Record lookup(const Account&);
bool lookup(const Account&); // 错误：与上一个函数相比只有返回类型不同
```

判断两个形参的类型是否相异

有时候两个形参列表看起来不一样，但实际上是相同的：

```
// 每对声明的是同一个函数
Record lookup(const Account &acct);
Record lookup(const Account&);    // 省略了形参的名字

typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&);      // Telno 和 Phone 的类型相同
```

在第一对声明中，第一个函数给它的形参起了名字，第二个函数没有。形参的名字仅仅起

232

到帮助记忆的作用，有没有它并不影响形参列表的内容。

第二对声明看起来类型不同，但事实上 Telno 不是一种新类型，它只是 Phone 的别名而已。类型别名（参见 2.5.1 节，第 60 页）为已存在的类型提供另外一个名字，它并不是创建新类型。因此，第二对中两个形参的区别仅在于一个使用类型原来的名字，另一个使用它的别名，从本质上来说它们没什么不同。

重载和 const 形参

如 6.2.3 节（第 190 页）介绍的，顶层 const（参见 2.4.3 节，第 57 页）不影响传入函数的对象。一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开来：

```
Record lookup(Phone);
Record lookup(const Phone);        // 重复声明了 Record lookup(Phone)

Record lookup(Phone*);
Record lookup(Phone* const);       // 重复声明了 Record lookup(Phone*)
```

在这两组函数声明中，每一组的第二个声明和第一个声明是等价的。

另一方面，如果形参是某种类型的指针或引用，则通过区分其指向的是常量对象还是非常量对象可以实现函数重载，此时的 const 是底层的：

```
// 对于接受引用或指针的函数来说，对象是常量还是非常量对应的形参不同
// 定义了 4 个独立的重载函数
Record lookup(Account&);           // 函数作用于 Account 的引用
Record lookup(const Account&);     // 新函数，作用于常量引用

Record lookup(Account*);           // 新函数，作用于指向 Account 的指针
Record lookup(const Account*);     // 新函数，作用于指向常量的指针
```

在上面的例子中，编译器可以通过实参是否是常量来推断应该调用哪个函数。因为 const 不能转换成其他类型（参见 4.11.2 节，第 144 页），所以我们只能把 const 对象（或指向 const 的指针）传递给 const 形参。相反的，因为非常量可以转换成 const，所以上面的 4 个函数都能作用于非常量对象或者指向非常量对象的指针。不过，如 6.6.1 节（第 220 页）将要介绍的，当我们传递一个非常量对象或者指向非常量对象的指针时，编译器会优先选用非常量版本的函数。

233

建议：何时不应该重载函数

尽管函数重载能在一定程度上减轻我们为函数起名字、记名字的负担，但是最好只重载那些确实非常相似的操作。有些情况下，给函数起不同的名字能使得程序更易理解。举个例子，下面是几个负责移动屏幕光标的函数：

```
Screen& moveHome();
Screen& moveAbs(int, int);
Screen& moveRel(int, int, string direction);
```

乍看上去，似乎可以把这组函数统一命名为 move，从而实现函数的重载：

```
Screen& move();
Screen& move(int, int);
Screen& move(int, int, string direction);
```

其实不然，重载之后这些函数失去了名字中本来拥有的信息。尽管这些函数确实都是在

移动光标，但是具体移动的方式却各不相同。以 moveHome 为例，它表示的是移动光标的一种特殊实例。一般来说，是否重载函数要看哪个更容易理解：

```
// 哪种形式更容易理解呢?
myScreen.moveHome(); // 我们认为应该是这一个!
myScreen.move();
```

const_cast 和重载

在 4.11.3 节（第 145 页）中我们说过，const_cast 在重载函数的情景中最有用。举个例子，回忆 6.3.2 节（第 201 页）的 shorterString 函数：

```
// 比较两个 string 对象的长度，返回较短的那个引用
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

这个函数的参数和返回类型都是 const string 的引用。我们可以对两个非常量的 string 实参调用这个函数，但返回的结果仍然是 const string 的引用。因此我们需要一种新的 shorterString 函数，当它的实参不是常量时，得到的结果是一个普通的引用，使用 const_cast 可以做到这一点：

```
string &shorterString(string &s1, string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1),
                            const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

在这个版本的函数中，首先将它的实参强制转换成对 const 的引用，然后调用了 shorterString 函数的 const 版本。const 版本返回对 const string 的引用，这个引用事实上绑定在了某个初始的非常量实参上。因此，我们可以再将其转换回一个普通的 string&，这显然是安全的。

调用重载的函数

定义了一组重载函数后，我们需要以合理的实参调用它们。**函数匹配**（function matching）是指一个过程，在这个过程中我们把函数调用与一组重载函数中的某一个关联起来，函数匹配也叫做**重载确定**（overload resolution）。编译器首先将调用的实参与重载集合中每一个函数的形参进行比较，然后根据比较的结果决定到底调用哪个函数。 234

在很多（可能是大多数）情况下，程序员很容易判断某次调用是否合法，以及当调用合法时应该调用哪个函数。通常，重载集中的函数区别明显，它们要不然是参数的数量不同，要不就是参数类型毫无关系。此时，确定调用哪个函数比较容易。但是在另外一些情况下要想选择函数就比较困难了，比如当两个重载函数参数数量相同且参数类型可以相互转换时（第 4.11 节，141 页）。我们将在 6.6 节（第 217 页）介绍当函数调用存在类型转换时编译器处理的方法。

现在我们需要掌握的是，当调用重载函数时有三种可能的结果：

- 编译器找到一个与实参**最佳匹配**（best match）的函数，并生成调用该函数的代码。
- 找不到任何一个函数与调用的实参匹配，此时编译器发出**无匹配**（no match）的错

误信息。

- 有多于一个函数可以匹配，但是每一个都不是明显的最佳选择。此时也将发生错误，称为**二义性调用**（ambiguous call）。

6.4 节练习

练习 6.39：说明在下面的每组声明中第二条声明语句是何含义。如果有非法的声明，请指出来。

```
(a)  int calc(int, int);
     int calc(const int, const int);
(b)  int get();
     double get();
(c)  int *reset(int *);
     double *reset(double *);
```

6.4.1 重载与作用域



一般来说，将函数声明置于局部作用域内不是一个明智的选择。但是为了说明作用域和重载的相互关系，我们将暂时违反这一原则而使用局部函数声明。

对于刚接触C++的程序员来说，不太容易理清作用域和重载的关系。其实，重载对作用域的一般性质并没有什么改变：如果我们在内层作用域中声明名字，它将隐藏外层作用域中声明的同名实体。在不同的作用域中无法重载函数名：

235

```
string read();
void print(const string &);
void print(double);  // 重载 print 函数
void fooBar(int ival)
{
    bool read = false;    // 新作用域：隐藏了外层的 read
    string s = read();    // 错误：read 是一个布尔值，而非函数
    // 不好的习惯：通常来说，在局部作用域中声明函数不是一个好的选择
    void print(int);      // 新作用域：隐藏了之前的 print
    print("Value: ");     // 错误：print(const string &)被隐藏掉了
    print(ival);          // 正确：当前 print(int)可见
    print(3.14);      // 正确：调用 print(int); print(double)被隐藏掉了
}
```

大多数读者都能理解调用 read 函数会引发错误。因为当编译器处理调用 read 的请求时，找到的是定义在局部作用域中的 read。这个名字是个布尔变量，而我们显然无法调用一个布尔值，因此该语句非法。

调用 print 函数的过程非常相似。在 fooBar 内声明的 print(int)隐藏了之前两个 print 函数，因此只有一个 print 函数是可用的：该函数以 int 值作为参数。

当我们调用 print 函数时，编译器首先寻找对该函数名的声明，找到的是接受 int 值的那个局部声明。一旦在当前作用域中找到了所需的名字，编译器就会忽略掉外层作用域中的同名实体。剩下的工作就是检查函数调用是否有效了。



在C++语言中，名字查找发生在类型检查之前。

第一个调用传入一个字符串字面值，但是当前作用域内 print 函数唯一的声明要求参数是 int 类型。字符串字面值无法转换成 int 类型，所以这个调用是错误的。在外层作用域中的print(const string&)函数虽然与本次调用匹配，但是它已经被隐藏掉了，根本不会被考虑。

当我们为 print 函数传入一个 double 类型的值时，重复上述过程。编译器在当前作用域内发现了 print(int)函数，double 类型的实参转换成 int 类型，因此调用是合法的。

假设我们把 print(int)和其他 print 函数声明放在同一个作用域中，则它将成为另一种重载形式。此时，因为编译器能看到所有三个函数，上述调用的处理结果将完全不同：

```
void print(const string &);
void print(double);              // print 函数的重载形式
void print(int);                 // print 函数的另一种重载形式
void fooBar2(int ival)
{
    print("Value: ");            // 调用 print(const string &)
    print(ival);                 // 调用 print(int)
    print(3.14);                 // 调用 print(double)
}
```

6.5 特殊用途语言特性

236

本节我们介绍三种函数相关的语言特性，这些特性对大多数程序都有用，它们分别是：默认实参、内联函数和 constexpr 函数，以及在程序调试过程中常用的一些功能。

6.5.1 默认实参

某些函数有这样一种形参，在函数的很多次调用中它们都被赋予一个相同的值，此时，我们把这个反复出现的值称为函数的**默认实参**（default argument）。调用含有默认实参的函数时，可以包含该实参，也可以省略该实参。

例如，我们使用 string 对象表示窗口的内容。一般情况下，我们希望该窗口的高、宽和背景字符都使用默认值。但是同时我们也应该允许用户为这几个参数自由指定与默认值不同的数值。为了使得窗口函数既能接纳默认值，也能接受用户指定的值，我们把它定义成如下的形式：

```
typedef string::size_type sz; // 关于 typedef 参见 2.5.1 节（第 60 页）
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
```

其中我们为每一个形参都提供了默认实参，默认实参作为形参的初始值出现在形参列表中。我们可以为一个或多个形参定义默认值，不过需要注意的是，一旦某个形参被赋予了默认值，它后面的所有形参都必须有默认值。

使用默认实参调用函数

如果我们想使用默认实参，只要在调用函数的时候省略该实参就可以了。例如，

screen 函数为它的所有形参都提供了默认实参，所以我们可以使用 0、1、2 或 3 个实参调用该函数：

```
string window;
window = screen();                    // 等价于 screen(24,80,' ')
window = screen(66);                  // 等价于 screen(66,80,' ')
window = screen(66, 256);             // screen(66,256,' ')
window = screen(66, 256, '#');        // screen(66,256,'#')
```

函数调用时实参按其位置解析，默认实参负责填补函数调用缺少的尾部实参（靠右侧位置）。例如，要想覆盖 backgrnd 的默认值，必须为 ht 和 wid 提供实参：

```
window = screen(, , '?');             // 错误：只能省略尾部的实参
window = screen('?');                 // 调用 screen('?',80,' ')
```

需要注意，第二个调用传递一个字符值，是合法的调用。然而尽管如此，它的实际效果却与书写的意图不符。该调用之所以合法是因为'?'是个 char，而函数最左侧形参的类型 string::size_type 是一种无符号整数类型，所以 char 类型可以转换成（参见 4.11 节，第 141 页）函数最左侧形参的类型。当该调用发生时，char 类型的实参隐式地转换成 string::size_type，然后作为 height 的值传递给函数。在我们的机器上，'?'对应的十六进制数是 0x3F，也就是十进制数的 63，所以该调用把值 63 传给了形参 height。

237

当设计含有默认实参的函数时，其中一项任务是合理设置形参的顺序，尽量让不怎么使用默认值的形参出现在前面，而让那些经常使用默认值的形参出现在后面。

默认实参声明

对于函数的声明来说，通常的习惯是将其放在头文件中，并且一个函数只声明一次，但是多次声明同一个函数也是合法的。不过有一点需要注意，在给定的作用域中一个形参只能被赋予一次默认实参。换句话说，函数的后续声明只能为之前那些没有默认值的形参添加默认实参，而且该形参右侧的所有形参必须都有默认值。假如给定

```
// 表示高度和宽度的形参没有默认值
string screen(sz, sz, char = ' ');
```

我们不能修改一个已经存在的默认值：

```
string screen(sz, sz, char = '*');            // 错误：重复声明
```

但是可以按照如下形式添加默认实参：

```
string screen(sz = 24, sz = 80, char);        // 正确：添加默认实参
```



通常，应该在函数声明中指定默认实参，并将该声明放在合适的头文件中。

默认实参初始值

局部变量不能作为默认实参。除此之外，只要表达式的类型能转换成形参所需的类型，该表达式就能作为默认实参：

```
// wd、def 和 ht 的声明必须出现在函数之外
sz wd = 80;
char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
```

```
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__ << __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

7.1 定义抽象数据类型 228
7.2 访问控制与封装 240
7.3 类的其他特性 243
7.4 类的作用域 253
7.5 构造函数再探 257
7.6 类的静态成员 268
小结 273
术语表 273

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（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。

261



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）。



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
```

```
}
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。

在文件中名字的出现处对其进行解析

287

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：

```
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) { }
};
```

290

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;
};
```



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。