

小结

< 168

C++语言提供了一套丰富的运算符，并定义了这些运算符作用于内置类型的运算对象时所执行的操作。此外，C++语言还支持运算符重载的机制，允许我们自己定义运算符作用于类类型时的含义。第14章将介绍如何定义作用于用户类型的运算符。

对于含有超过一个运算符的表达式，要想理解其含义关键要理解优先级、结合律和求值顺序。每个运算符都有其对应的优先级和结合律，优先级规定了复合表达式中运算符组合的方式，结合律则说明当运算符的优先级一样时应该如何组合。

大多数运算符并不明确规定运算对象的求值顺序：编译器有权自由选择先对左侧运算对象求值还是先对右侧运算对象求值。一般来说，运算对象的求值顺序对表达式的最终结果没有影响。但是，如果两个运算对象指向同一个对象而且其中一个改变了对象的值，就会导致程序出现不易发现的严重缺陷。

最后一点，运算对象经常从原始类型自动转换成某种关联的类型。例如，表达式中的小整型会自动提升成大整型。不论内置类型还是类类型都涉及类型转换的问题。如果需要，我们还可以显式地进行强制类型转换。

术语表

算术转换 (arithmetic conversion) 从一种算术类型转换成另一种算术类型。在二元运算符的上下文中，为了保留精度，算术转换通常把较小的类型转换成较大的类型（例如整型转换成浮点型）。

结合律 (associativity) 规定具有相同优先级的运算符如何组合在一起。结合律分为左结合律（运算符从左向右组合）和右结合律（运算符从右向左组合）。

二元运算符 (binary operator) 有两个运算对象参与运算的运算符。

强制类型转换 (cast) 一种显式的类型转换。

复合表达式 (compound expression) 含有一个以上的运算符的表达式。

const_cast 一种涉及 `const` 的强制类型转换。将底层 `const` 对象转换成对应的非常量类型，或者执行相反的转换。

转换 (conversion) 一种类型的值改变成另一种类型的值的过程。C++语言定义了内置类型的转换规则。类类型同样可以转换。

dynamic_cast 和继承及运行时类型识别一

起使用。参见 19.2 节（第 730 页）。

表达式 (expression) C++程序中最低级别的计算。表达式将运算符作用于一个或多个运算对象，每个表达式都有对应的求值结果。表达式本身也可以作为运算对象，这时就得到了对多个运算符求值的复合表达式。

隐式转换 (implicit conversion) 由编译器自动执行的类型转换。假如表达式需要某种特定的类型而运算对象是另外一种类型，此时只要规则允许，编译器就会自动地将运算对象转换成所需的类型。

整型提升 (integral promotion) 把一种较小的整数类型转换成与之最接近的较大整数类型的过程。不论是否真的需要，小整数类型（即 `short`、`char` 等）总是会得到提升。

左值 (lvalue) 是指那些求值结果为对象或函数的表达式。一个表示对象的非常量左值可以作为赋值运算符的左侧运算对象。

运算对象 (operand) 表达式在某些值上执行运算，这些值就是运算对象。一个运算

< 169

符有一个或多个相关的运算对象。

运算符 (operator) 决定表达式所做操作的符号。C++语言定义了一套运算符并说明了这些运算符作用于内置类型时的含义。C++还定义了运算符的优先级和结合律以及每种运算符处理的运算对象数量。可以重载运算符使其能处理类类型。

求值顺序 (order of evaluation) 是某个运算符的运算对象的求值顺序。大多数情况下，编译器可以任意选择运算对象求值的顺序。不过运算对象一定要在运算符之前得到求值结果。只有`&&`、`||`、条件和逗号四种运算符明确规定了求值顺序。

重载运算符 (overloaded operator) 针对某种运算符重新定义的适用于类类型的版本。第14章将介绍重载运算符的方法。

优先级 (precedence) 规定了复合表达式中不同运算符的执行顺序。与低优先级的运算符相比，高优先级的运算符组合得更紧密。

提升 (promoted) 参见整型提升。

reinterpret_cast 把运算对象的内容解释成另外一种类型。这种强制类型转换本质上依赖于机器而且非常危险。170

结果 (result) 计算表达式得到的值或对象。

右值 (rvalue) 是指一种表达式，其结果是值而非值所在的位置。

短路求值 (short-circuit evaluation) 是一个专有名词，描述逻辑与运算符和逻辑或运算符的执行过程。如果根据运算符的第一个运算对象就能确定整个表达式的结构，求值终止，此时第二个运算对象将不会被求值。

sizeof 是一个运算符，返回存储对象所需的字节数，该对象的类型可能是某个给定的类型名字，也可能由表达式的返回结果确定。

static_cast 显式地执行某种定义明确的类型转换，常用于替代由编译器隐式执行的类型转换。

一元运算符 (unary operators) 只有一个运算对象参与运算的运算符。

, 运算符 (, operator) 逗号运算符，是一种从左向右求值的二元运算符。逗号运算符的结果是右侧运算对象的值，当且仅当右侧运算对象是左值时逗号运算符的结果是左值。

? : 运算符 (?: operator) 条件运算符，以下述形式提供 if-then-else 逻辑的表达式

`cond ? expr1 : expr2;`

如果条件 `cond` 为真，对 `expr1` 求值；否则对 `expr2` 求值。`expr1` 和 `expr2` 的类型应该相同或者能转换成同一种类型。`expr1` 和 `expr2` 中只有一个会被求值。

&&运算符 (&& operator) 逻辑与运算符，如果两个运算对象都是真，结果才为真。只有当左侧运算对象为真时才会检查右侧运算对象。

&运算符 (& operator) 位与运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位都是 1，所得结果中该位为 1；否则所得结果中该位为 0。

^运算符 (^ operator) 位异或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位有且只有一个是 1，所得结果中该位为 1；否则所得结果中该位为 0。

||运算符 (|| operator) 逻辑或运算符，任何一个运算对象是真，结果就为真。只有当左侧运算对象为假时才会检查右侧运算对象。

| 运算符 (| operator) 位或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位至少有一个是 1，所得结果中该位为 1；否则所得结果中该位为 0。

++运算符 (++ operator) 递增运算符。包括两种形式：前置版本和后置版本。前置递增运算符得到一个左值，它给运算符加 1 并得到运算对象改变后的值。后置递增运算符得到一个右值，它给运算符加 1 并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义+运算符，也会

有++运算符。

-运算符 (— operator) 递减运算符。包括两种形式：前置版本和后置版本。前置递减运算符得到一个左值，它从运算符减 1 并得到运算对象改变后的值。后置递减运算符得到一个右值，它从运算符减 1 并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义-运算符，也会有--运算符。

<<运算符 (<< operator) 左移运算符，将左侧运算对象的值的（可能是提升后的）副本向左移位，移动的位数由右侧运算对象确定。右侧运算对象必须大于等于 0 而且小于结果的位数。左侧运算对象应该是无符号类型，如果它是带符号类型，则一旦移动改变了符号位的值就会产生未定义的结果。

>>运算符 (>> operator) 右移运算符，除了移动方向相反，其他性质都和左移运算符类似。如果左侧运算对象是带符号类型，那么根据实现的不同新移入的内容也不同，新移入的位可能都是 0，也可能都是符号位的副本。

~运算符 (~ operator) 位求反运算符，生成一个新的整型值。该值的每一位恰好与（可能是提升后的）运算对象的对应位相反。

!运算符 (! operator) 逻辑非运算符，将它的运算对象的布尔值取反。如果运算对象是假，则结果为真，如果运算对象是真，则结果为假。

第 5 章

语句

171

内容

5.1 简单语句.....	154
5.2 语句作用域.....	155
5.3 条件语句.....	156
5.4 迭代语句.....	165
5.5 跳转语句.....	170
5.6 try 语句块和异常处理	172
小结	178
术语表.....	178

和大多数语言一样，C++提供了条件执行语句、重复执行相同代码的循环语句和用于中断当前控制流的跳转语句。本章将详细介绍 C++语言所支持的这些语句。

172 通常情况下，语句是顺序执行的。但除非是最简单的程序，否则仅有顺序执行远远不够。因此，C++语言提供了一组控制流（flow-of-control）语句以支持更复杂的执行路径。



5.1 简单语句

C++语言中的大多数语句都以分号结束，一个表达式，比如 `ival + 5`，末尾加上分号就变成了**表达式语句**（expression statement）。表达式语句的作用是执行表达式并丢弃掉求值结果：

```
ival + 5;           // 一条没什么实际用处的表达式语句
cout << ival;      // 一条有用的表达式语句
```

第一条语句没什么用处，因为虽然执行了加法，但是相加的结果没被使用。比较普遍的情况是，表达式语句中的表达式在求值时附带有其他效果，比如给变量赋了新值或者输出了结果。

空语句

最简单的语句是**空语句**（null statement），空语句中只含有一个单独的分号：

```
; // 空语句
```

如果在程序的某个地方，语法上需要一条语句但是逻辑上不需要，此时应该使用空语句。一种常见的情况是，当循环的全部工作在条件部分就可以完成时，我们通常会用到空语句。例如，我们想读取输入流的内容直到遇到一个特定的值为止，除此之外什么事情也不做：

```
// 重复读入数据直至到达文件末尾或某次输入的值等于 sought
while (cin >> s && s != sought)
    ; // 空语句
```

`while` 循环的条件部分首先从标准输入读取一个值并且隐式地检查 `cin`，判断读取是否成功。假定读取成功，条件的后半部分检查读进来的值是否等于 `sought` 的值。如果发现了想要的值，循环终止；否则，从 `cin` 中继续读取另一个值，再一次判断循环的条件。

Best Practices

使用空语句时应该加上注释，从而令读这段代码的人知道该语句是有意省略的。

别漏写分号，也别多写分号

因为空语句是一条语句，所以可用在任何允许使用语句的地方。由于这个原因，某些看起来非法的分号往往只不过是一条空语句而已，从语法上说得过去。下面的片段包含两条语句：表达式语句和空语句。

173 `ival = v1 + v2;; // 正确：第二个分号表示一条多余的空语句`

多余的空语句一般来说是无害的，但是如果在 `if` 或者 `while` 的条件后面跟了一个额外的分号就可能完全改变程序员的初衷。例如，下面的代码将无休止地循环下去：

```
// 出现了糟糕的情况：额外的分号，循环体是那条空语句
while (iter != svec.end()) ;           // while 循环体是那条空语句
    ++iter;                            // 递增运算不属于循环的一部分
```

虽然从形式上来看执行递增运算的语句前面有缩进，但它并不是循环的一部分。循环条件后面跟着的分号构成了一条空语句，它才是真正的循环体。



多余的空语句并非总是无害的。

复合语句（块）

复合语句（compound statement）是指用花括号括起来的（可能为空的）语句和声明的序列，复合语句也被称作块（block）。一个块就是一个作用域（参见 2.2.4 节，第 43 页），在块中引入的名字只能在块内部以及嵌套在块中的子块里访问。通常，名字在有限的区域内可见，该区域从名字定义处开始，到名字所在的（最内层）块的结尾为止。

如果在程序的某个地方，语法上需要一条语句，但是逻辑上需要多条语句，则应该使用复合语句。例如，`while` 或者 `for` 的循环体必须是一条语句，但是我们常常需要在循环体内做很多事情，此时就需要将多条语句用花括号括起来，从而把语句序列转变成块。

举个例子，回忆 1.4.1 节（第 10 页）的 `while` 循环：

```
while (val <= 10) {  
    sum += val;      // 把 sum + val 的值赋给 sum.  
    ++val;          // 给 val 加 1  
}
```

程序从逻辑上来说要执行两条语句，但是 `while` 循环只能容纳一条。此时，把要执行的语句用花括号括起来，就将其转换成了一条（复合）语句。



块不以分号作为结束。

所谓空块，是指内部没有任何语句的一对花括号。空块的作用等价于空语句：

```
while (cin >> s && s != sought)  
{ } // 空块
```

5.1 节练习

174

练习 5.1：什么是空语句？什么时候会用到空语句？

练习 5.2：什么是块？什么时候会用到块？

练习 5.3：使用逗号运算符（参见 4.10 节，第 140 页）重写 1.4.1 节（第 10 页）的 `while` 循环，使它不再需要块，观察改写之后的代码的可读性提高了还是降低了。

5.2 语句作用域

可以在 `if`、`switch`、`while` 和 `for` 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用范围了：

```
while (int i = get_num()) // 每次迭代时创建并初始化 i  
    cout << i << endl;  
i = 0; // 错误：在循环外部无法访问 i
```

如果其他代码也需要访问控制变量，则变量必须定义在语句的外部：

```
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0
```

因为控制结构定义的对象的值马上要由结构本身使用，所以这些变量必须初始化。

5.2 节练习

练习 5.4：说明下列例子的含义，如果存在问题，试着修改它。

```
(a) while (string::iterator iter != s.end()) { /* ... */ }
(b) while (bool status = find(word)) { /* ... */ }
    if (!status) { /* ... */ }
```

5.3 条件语句

C++语言提供了两种按条件执行的语句。一种是 **if** 语句，它根据条件决定控制流；另外一种是 **switch** 语句，它计算一个整型表达式的值，然后根据这个值从几条执行路径中选择一条。



5.3.1 if 语句

175> **if 语句 (if statement)** 的作用是：判断一个指定的条件是否为真，根据判断结果决定是否执行另外一条语句。**if** 语句包括两种形式：一种含有 **else** 分支，另外一种没有。简单 **if** 语句的语法形式是

```
if (condition)
    statement
```

if else 语句的形式是

```
if (condition)
    statement
else
    statement2
```

在这两个版本的 **if** 语句中，*condition* 都必须用圆括号包围起来。*condition* 可以是一个表达式，也可以是一个初始化了的变量声明（参见 5.2 节，第 155 页）。不管是表达式还是变量，其类型都必须能转换成（参见 4.11 节，第 141 页）布尔类型。通常情况下，*statement* 和 *statement2* 是块语句。

如果 *condition* 为真，执行 *statement*。当 *statement* 执行完成后，程序继续执行 **if** 语句后面的其他语句。

如果 *condition* 为假，跳过 *statement*。对于简单 **if** 语句来说，程序继续执行 **if** 语句后面的其他语句；对于 **if else** 语句来说，执行 *statement2*。

使用 if else 语句

我们举个例子来说明 if 语句的功能，程序的目的是把数字形式表示的成绩转换成字母形式。假设数字成绩的范围是从 0 到 100（包括 100 在内），其中 100 分对应的字母形式是“A++”，低于 60 分的成绩对应的字母形式是“F”。其他成绩每 10 个划分成一组：60 到 69（包括 69 在内）对应字母“D”、70 到 79 对应字母“C”，以此类推。使用 vector 对象存放字母成绩所有可能的取值：

```
const vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

我们使用 if else 语句解决该问题，根据成绩是否合格执行不同的操作：

```
// 如果 grade 小于 60，对应的字母是 F；否则计算其下标
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

判断 grade 的值是否小于 60，根据结果选择执行 if 分支还是 else 分支。在 else 分支中，由成绩计算得到一个下标，具体过程是：首先从 grade 中减去 50，然后执行整数除法（参见 4.2 节，在 125 页），去掉余数后所得的商就是数组 scores 对应的下标。

嵌套 if 语句

176

接下来让我们的程序更有趣点儿，试着给那些合格的成绩后面添加一个加号或减号。如果成绩的末位是 8 或者 9，添加一个加号；如果末位是 0、1 或 2，添加一个减号：

```
if (grade % 10 > 7)
    lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
else if (grade % 10 < 3)
    lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

我们使用取模运算符（参见 4.2 节，第 125 页）计算余数，根据余数决定添加哪种符号。

接着把这段添加符号的代码整合到转换成绩形式的代码中去：

```
// 如果成绩不合格，不需要考虑添加加号减号的问题
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10]; // 获得字母形式的成绩
    if (grade != 100) // 只要不是 A++，就考虑添加加号或减号
        if (grade % 10 > 7)
            lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
        else if (grade % 10 < 3)
            lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
}
```

注意，我们使用花括号把第一个 else 后面的两条语句组合成了一个块。如果 grade 不小于 60 要做两件事：从数组 scores 中获取对应的字母成绩，然后根据条件设置加号或减号。

注意使用花括号

有一种常见的错误：本来程序中有几条语句应该作为一个块来执行，但是我们忘了用花括号把这些语句包围。在下面的例子中，添加加号减号的代码将被无条件地执行，这显然违背了我们的初衷：

```
if (grade < 60)
    lettergrade = scores[0];
else // 错误：缺少花括号
    lettergrade = scores[(grade - 50) / 10];
// 虽然下面的语句从形式上看有缩进，但是因为没有花括号，
// 所以无论什么情况都会执行接下来的代码
// 不及格的成绩也会添加上加号或减号，这显然是错误的
if (grade != 100)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else if (grade % 10 < 3)
        lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

要想发现这个错误可能非常困难，毕竟这段代码“看起来”是正确的。

177 为了避免此类问题，有些编码风格要求在 `if` 或 `else` 之后必须写上花括号（对 `while` 和 `for` 语句的循环体两端也有同样的要求）。这么做的好处是可以避免代码混乱不清，以后修改代码时如果想添加别的语句，也可以很容易地找到正确位置。



许多编辑器和开发环境都提供一种辅助工具，它可以自动地缩进代码以匹配其语法结构。善用此类工具益处多多。

悬垂 `else`

当一个 `if` 语句嵌套在另一个 `if` 语句内部时，很可能 `if` 分支会多于 `else` 分支。事实上，之前那个成绩转换的程序就有 4 个 `if` 分支，而只有 2 个 `else` 分支。这时候问题出现了：我们怎么知道某个给定的 `else` 是和哪个 `if` 匹配呢？

这个问题通常称作 **悬垂 `else`**（dangling `else`），在那些既有 `if` 语句又有 `if` `else` 语句的编程语言中是个普遍存在的问题。不同语言解决该问题的思路也不同，就 C++ 而言，它规定 `else` 与离它最近的尚未匹配的 `if` 匹配，从而消除了程序的二义性。

当代码中 `if` 分支多于 `else` 分支时，程序员有时会感觉比较麻烦。举个例子来说明，对于添加加号减号的那个最内层的 `if` `else` 语句，我们用另外一组条件改写它：

```
// 错误：实际的执行过程并非像缩进格式显示的那样；else 分支匹配的是内层 if 语句
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else
        lettergrade += '-'; // 末尾是 3、4、5、6 或者 7 的成绩添加一个减号！
```

从代码的缩进格式来看，程序的初衷应该是希望 `else` 和外层的 `if` 匹配，也就是说，我们希望当 `grade` 的末位小于 3 时执行 `else` 分支。然而，不管我们是什么意图，也不管程序如何缩进，这里的 `else` 分支其实是内层 `if` 语句的一部分。最终，上面的代码将在末位大于 3 小于等于 7 的成绩后面添加减号！它的执行过程实际上等价于如下形式：

```
// 缩进格式与执行过程相符，但不是程序员的意图
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else
        lettergrade += '-'; // 末尾是 3、4、5、6 或者 7 的成绩添加一个减号！
```

使用花括号控制执行路径

要想使 `else` 分支和外层的 `if` 语句匹配起来，可以在内层 `if` 语句的两端加上花括号，使其成为一个块：

```
// 末尾是 8 或者 9 的成绩添加一个加号，末尾是 0、1 或者 2 的成绩添加一个减号
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
} else // 花括号强迫 else 与外层 if 匹配
    lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

语句属于块，意味着语句一定在块的边界之内，因此内层 `if` 语句在关键字 `else` 前面的那个花括号处已经结束了。`else` 不会再作为内层 `if` 的一部分。此时，最近的尚未匹配的 `if` 是外层 `if`，也就是我们希望 `else` 匹配的那个。

<178

5.3.1 节练习

练习 5.5: 写一段自己的程序，使用 `if else` 语句实现把数字成绩转换成字母成绩的要求。

练习 5.6: 改写上一题的程序，使用条件运算符（参见 4.7 节，第 134 页）代替 `if else` 语句。

练习 5.7: 改正下列代码段中的错误。

- `if (ival1 != ival2)`
 `ival1 = ival2`
`else ival1 = ival2 = 0;`
- `if (ival < minval)`
 `minval = ival;`
 `occurs = 1;`
- `if (int ival = get_value())`
 `cout << "ival = " << ival << endl;`
`if (!ival)`
 `cout << "ival = 0\n";`
- `if (ival = 0)`
 `ival = get_value();`

练习 5.8: 什么是“悬垂 `else`”？C++语言是如何处理 `else` 子句的？

5.3.2 switch 语句

switch 语句 (`switch statement`) 提供了一条便利的途径使得我们能够在若干固定选项中做出选择。举个例子，假如我们想统计五个元音字母在文本中出现的次数，程序逻辑应该如下所示：

- 从输入的内容中读取所有字符。
- 令每一个字符都与元音字母的集合比较。
- 如果字符与某个元音字母匹配，将该字母的数量加 1。
- 显示结果。

例如，以（原书中）本章的文本作为输入内容，程序的输出结果将是：

```
Number of vowel a: 3195
Number of vowel e: 6230
Number of vowel i: 3102
Number of vowel o: 3289
Number of vowel u: 1033
```

179 要想实现这项功能，直接使用 switch 语句即可：

```
// 为每个元音字母初始化其计数值
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // 如果 ch 是元音字母，将其对应的计数值加 1
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// 输出结果
cout << "Number of vowel a: \t" << aCnt << '\n'
    << "Number of vowel e: \t" << eCnt << '\n'
    << "Number of vowel i: \t" << iCnt << '\n'
    << "Number of vowel o: \t" << oCnt << '\n'
    << "Number of vowel u: \t" << uCnt << endl;
```

switch 语句首先对括号里的表达式求值，该表达式紧跟在关键字 switch 的后面，可以是一个初始化的变量声明（参见 5.2 节，第 155 页）。表达式的值转换成整数类型，然后与每个 case 标签的值比较。

如果表达式和某个 case 标签的值匹配成功，程序从该标签之后的第一条语句开始执行，直到到达了 switch 的结尾或者是遇到一条 break 语句为止。

我们将在 5.5.1 节（第 170 页）详细介绍 break 语句，简言之，break 语句的作用是

中断当前的控制流。此例中, `break` 语句将控制权转移到 `switch` 语句外面。因为 `switch` 是 `while` 循环体内唯一的语句, 所以从 `switch` 语句中断出来以后, 程序的控制权将移到 `while` 语句的右花括号处。此时 `while` 语句内部没有其他语句要执行, 所以 `while` 会返回去再一次判断条件是否满足。

如果 `switch` 语句的表达式和所有 `case` 都没有匹配上, 将直接跳转到 `switch` 结构之后的第一条语句。刚刚说过, 在上面的例子中, 退出 `switch` 后控制权回到 `while` 语句的条件部分。

`case` 关键字和它对应的值一起被称为 **case 标签** (`case label`)。`case` 标签必须是整型常量表达式 (参见 2.4.4 节, 第 58 页):

```
char ch = getVal();
int ival = 42;
switch(ch) {
    case 3.14: // 错误: case 标签不是一个整数
    case ival: // 错误: case 标签不是一个常量
    //...
```

<180

任何两个 `case` 标签的值不能相同, 否则就会引发错误。另外, `default` 也是一种特殊的 `case` 标签, 关于它的知识将在第 162 页介绍。

switch 内部的控制流

理解程序在 `case` 标签之间的执行流程非常重要。如果某个 `case` 标签匹配成功, 将从该标签开始往后顺序执行所有 `case` 分支, 除非程序显式地中断了这一过程, 否则直到 `switch` 的结尾处才会停下来。要想避免执行后续 `case` 分支的代码, 我们必须显式地告诉编译器终止执行过程。大多数情况下, 在下一个 `case` 标签之前应该有一条 `break` 语句。

然而, 也有一些时候默认的 `switch` 行为才是程序真正需要的。每个 `case` 标签只能对应一个值, 但是有时候我们希望两个或更多个值共享同一组操作。此时, 我们就故意省略掉 `break` 语句, 使得程序能够连续执行若干个 `case` 标签。

例如, 也许我们想统计的是所有元音字母出现的总次数:

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    // 出现了 a、e、i、o 或 u 中的任意一个都会将 vowelCnt 的值加 1
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

在上面的代码中, 几个 `case` 标签连写在一起, 中间没有 `break` 语句。因此只要 `ch` 是元音字母, 不管到底是五个中的哪一个都执行相同的代码。

C++程序的形式比较自由, 所以 `case` 标签之后不一定非得换行。把几个 `case` 标签

写在一行里，强调这些 case 代表的是某个范围内的值：

```
switch (ch)
{
    // 另一种合法的书写形式
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

181



一般不要省略 case 分支最后的 break 语句。如果没写 break 语句，最好加一段注释说清楚程序的逻辑。

漏写 break 容易引发缺陷

有一种常见的错觉是程序只执行匹配成功的那个 case 分支的语句。例如，下面程序的统计结果是错误的：

```
// 警告：不正确的程序逻辑！
switch (ch) {
    case 'a':
        ++aCnt; // 此处应该有一条 break 语句
    case 'e':
        ++eCnt; // 此处应该有一条 break 语句
    case 'i':
        ++iCnt; // 此处应该有一条 break 语句
    case 'o':
        ++oCnt; // 此处应该有一条 break 语句
    case 'u':
        ++uCnt;
}
```

要想理解这段程序的执行过程，不妨假设 ch 的值是 'e'。此时，程序直接执行 case 'e' 标签后面的代码，该代码把 eCnt 的值加 1。接下来，程序将跨越 case 标签的边界，接着递增 iCnt、oCnt 和 uCnt。



尽管 switch 语句不是非得在最后一个标签后面写上 break，但是为了安全起见，最好这么做。因为这样的话，即使以后再增加新的 case 分支，也不用再在前面补充 break 语句了。

default 标签

如果没有任何一个 case 标签能匹配上 switch 表达式的值，程序将执行紧跟在 **default 标签 (default label)** 后面的语句。例如，可以增加一个计数值来统计非元音字母的数量，只要在 default 分支内不断递增名为 otherCnt 的变量就可以了：

```
// 如果 ch 是一个元音字母，将相应的计数值加 1
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
```

```

        ++otherCnt;
        break;
    }
}

```

在这个版本的程序中，如果 ch 不是元音字母，就从 default 标签开始执行并把 otherCnt 加 1。 ◀182



即使不准备在 default 标签下做任何工作，定义一个 default 标签也是有用的。其目的在于告诉程序的读者，我们已经考虑到了默认的情况，只是目前什么也没做。

标签不应该孤零零地出现，它后面必须跟上一条语句或者另外一个 case 标签。如果 switch 结构以一个空的 default 标签作为结束，则该 default 标签后面必须跟上一条空语句或一个空块。

switch 内部的变量定义

如前所述，switch 的执行流程有可能会跨过某些 case 标签。如果程序跳转到了某个特定的 case，则 switch 结构中该 case 标签之前的部分会被忽略掉。这种忽略掉一部分代码的行为引出了一个有趣的问题：如果被略过的代码中含有变量的定义该怎么办？

答案是：如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法行为。

```

case true:
    // 因为程序的执行流程可能绕开下面的初始化语句，所以该 switch 语句不合法
    string file_name;      // 错误：控制流绕过一个隐式初始化的变量
    int ival = 0;           // 错误：控制流绕过一个显式初始化的变量
    int jval;               // 正确：因为 jval 没有初始化
    break;
case false:
    // 正确：jval 虽然在作用域内，但是它没有被初始化
    jval = next_num();     // 正确：给 jval 赋一个值
    if (file_name.empty()) // file_name 在作用域内，但是没有被初始化
        ...

```

假设上述代码合法，则一旦控制流直接跳到 false 分支，也就同时略过了变量 file_name 和 ival 的初始化过程。此时这两个变量位于作用域之内，跟在 false 之后的代码试图在尚未初始化的情况下使用它们，这显然是行不通的。因此 C++ 语言规定，不允许跨过变量的初始化语句直接跳转到该变量作用域内的另一个位置。

如果需要为某个 case 分支定义并初始化一个变量，我们应该把变量定义在块内，从而确保后面的所有 case 标签都在变量的作用域之外。

```

case true:
{
    // 正确：声明语句位于语句块内部
    string file_name = get_file_name();
    ...
}
break;
case false:

```

```
if (file_name.empty()) // 错误: file_name 不在作用域之内
```

183 >

5.3.2 节练习

练习 5.9: 编写一段程序，使用一系列 if 语句统计从 cin 读入的文本中有多少元音字母。

练习 5.10: 我们之前实现的统计元音字母的程序存在一个问题：如果元音字母以大写形式出现，不会被统计在内。编写一段程序，既统计元音字母的小写形式，也统计大写形式，也就是说，新程序遇到‘a’和‘A’都应该递增 aCnt 的值，以此类推。

练习 5.11: 修改统计元音字母的程序，使其也能统计空格、制表符和换行符的数量。

练习 5.12: 修改统计元音字母的程序，使其能统计以下含有两个字符的字符序列的数量：ff、fl 和 fi。

练习 5.13: 下面显示的每个程序都含有一个常见的编程错误，指出错误在哪里，然后修改它们。

```
(a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
    char ch = next_text();
    switch (ch) {
        case 'a': aCnt++;
        case 'e': eCnt++;
        default: iouCnt++;
    }

(b) unsigned index = some_value();
    switch (index) {
        case 1:
            int ix = get_value();
            ivec[ ix ] = index;
            break;
        default:
            ix = ivec.size()-1;
            ivec[ ix ] = index;
    }

(c) unsigned evenCnt = 0, oddCnt = 0;
    int digit = get_num() % 10;
    switch (digit) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
    }

(d) unsigned ival=512, jval=1024, kval=4096;
    unsigned bufsize;
    unsigned swt = get_bufCnt();
    switch(swt) {
```

```

    case ival:
        bufsize = ival * sizeof(int);
        break;
    case jval:
        bufsize = jval * sizeof(int);
        break;
    case kval:
        bufsize = kval * sizeof(int);
        break;
}

```

5.4 迭代语句

迭代语句通常称为循环，它重复执行操作直到满足某个条件才停下来。`while` 和 `for` 语句在执行循环体之前检查条件，`do while` 语句先执行循环体，然后再检查条件。

5.4.1 while 语句



只要条件为真，`while` 语句（while statement）就重复地执行循环体，它的语法形式是

```

while (condition)
    statement

```

在 `while` 结构中，只要 `condition` 的求值结果为真就一直执行 `statement`（常常是一个块）。`condition` 不能为空，如果 `condition` 第一次求值就得 `false`，`statement` 一次都不执行。

`while` 的条件部分可以是一个表达式或者是一个带初始化的变量声明（参见 5.2 节，第 155 页）。通常来说，应该由条件本身或者是循环体设法改变表达式的值，否则循环可能无法终止。



定义在 `while` 条件部分或者 `while` 循环体内的变量每次迭代都经历从创建到销毁的过程。

使用 while 循环

当不确定到底要迭代多少次时，使用 `while` 循环比较合适，比如读取输入的内容就是如此。还有一种情况也应该使用 `while` 循环，这就是我们想在循环结束后访问循环控制变量。例如：

```

vector<int> v;
int i;
// 重复读入数据，直至到达文件末尾或者遇到其他输入问题
while (cin >> i)
    v.push_back(i);
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0

```

第一个循环从标准输入中读取数据，我们一开始不清楚循环要执行多少次，当 `cin` 读取到无效数据、遇到其他一些输入错误或是到达文件末尾时循环条件失效。第二个循环重复执行直到遇到一个负值为止，循环终止后，`beg` 或者等于 `v.end()`，或者指向 `v` 中一个小于 0 的元素。可以在 `while` 循环外继续使用 `beg` 的状态以进行其他处理。

5.4.1 节练习

练习 5.14：编写一段程序，从标准输入中读取若干 `string` 对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词。如果这样的单词存在，输出重复出现的最大次数；如果不存在，输出一条信息说明任何单词都没有连续出现过。例如，如果输入是

```
how now now now brown cow cow
```

那么输出应该表明单词 `now` 连续出现了 3 次。



5.4.2 传统的 for 语句

for 语句的语法形式是

```
for (init-statement; condition; expression)
    statement
```

关键字 `for` 及括号里的部分称作 `for` 语句头。

`init-statement` 必须是以下三种形式中的一种：声明语句、表达式语句或者空语句，因为这些语句都以分号作为结束，所以 `for` 语句的语法形式也可以看做

```
for (initializer; condition; expression)
    statement
```

一般情况下，`init-statement` 负责初始化一个值，这个值将随着循环的进行而改变。

`condition` 作为循环控制的条件，只要 `condition` 为真，就执行一次 `statement`。如果 `condition` 第一次的求值结果就是 `false`，则 `statement` 一次也不会执行。`expression` 负责修改 `init-statement` 初始化的变量，这个变量正好就是 `condition` 检查的对象，修改发生在每次循环迭代之后。`statement` 可以是一条单独的语句也可以是一条复合语句。

传统 for 循环的执行流程

我们以 3.2.3 节（第 85 页）的 `for` 循环为例：

```
// 重复处理 s 中的字符直至我们处理完全部字符或者遇到了一个表示空白的字符
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
    s[index] = toupper(s[index]); // 将当前字符改成大写形式
```

求值的顺序如下所示：

1. 循环开始时，首先执行一次 `init-statement`。此例中，定义 `index` 并初始化为 0。
2. 接下来判断 `condition`。如果 `index` 不等于 `s.size()` 而且在 `s[index]` 位置的字符不是空白，则执行 `for` 循环体的内容。否则，循环终止。如果第一次迭代时条件就为假，`for` 循环体一次也不会执行。
3. 如果条件为真，执行循环体。此例中，`for` 循环体将 `s[index]` 位置的字符改写

成大写形式。

4. 最后执行 *expression*。此例中，将 *index* 的值加 1。

这 4 步说明了 *for* 循环第一次迭代的过程。其中第 1 步只在循环开始时执行一次，第 2、3、4 步重复执行直到条件为假时终止，也就是在 *s* 中遇到一个空白字符或者 *index* 大于 *s.size()* 时终止。



牢记 *for* 语句头中定义的对象只在 *for* 循环体内可见。因此在上面的例子中，*for* 循环结束后 *index* 就不可用了。

for 语句头中的多重定义

和其他的声明一样，*init-statement* 也可以定义多个对象。但是 *init-statement* 只能有一条声明语句，因此，所有变量的基础类型必须相同（参见 2.3 节，第 45 页）。举个例子，我们用下面的循环把 *vector* 的元素拷贝一份添加到原来的元素后面：

```
// 记录下 v 的大小，当到达原来的最后一个元素后结束循环
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

在这个循环中，我们在 *init-statement* 里同时定义了索引 *i* 和循环控制变量 *sz*。

省略 for 语句头的某些部分

187

for 语句头能省略掉 *init-statement*、*condition* 和 *expression* 中的任何一个（或者全部）。

如果无须初始化，则我们可以使用一条空语句作为 *init-statement*。例如，对于在 *vector* 对象中寻找第一个负数的程序，完全能用 *for* 循环改写：

```
auto beg = v.begin();
for (/* 空语句 */; beg != v.end() && *beg >= 0; ++beg)
    ; // 什么也不做
```

注意，分号必须保留以表明我们省略掉了 *init-statement*。说得更准确一点，分号表示的是一个空的 *init-statement*。在这个循环中，因为所有要做的工作都在 *for* 语句头的条件和表达式部分完成了，所以 *for* 循环体也是空的。其中，条件部分决定何时停止查找，表达式部分递增迭代器。

省略 *condition* 的效果等价于在条件部分写了一个 *true*。因为条件的值永远是 *true*，所以在循环体内必须有语句负责退出循环，否则循环就会无休止地执行下去：

```
for (int i = 0; /* 条件为空 */; ++i) {
    // 对 i 进行处理，循环内部的代码必须负责终止迭代过程!
}
```

我们也能省略掉 *for* 语句头中的 *expression*，但是在这样的循环中就要求条件部分或者循环体必须改变迭代变量的值。举个例子，之前有一个将整数读入 *vector* 的 *while* 循环，我们使用 *for* 语句改写它：

```
vector<int> v;
for (int i; cin >> i; /* 表达式为空 */)
    v.push_back(i);
```

因为条件部分能改变 *i* 的值，所以这个循环无须表达式部分。其中，条件部分不断检查输

入流的内容，只要读取完所有的输入或者遇到一个输入错误就终止循环。

5.4.2 节练习

练习 5.15：说明下列循环的含义并改正其中的错误。

```
(a) for (int ix = 0; ix != sz; ++ix) { /* ... */ }
    if (ix != sz)
        // ...
(b) int ix;
    for (ix != sz; ++ix) { /* ... */ }
(c) for (int ix = 0; ix != sz; ++ix, ++sz) { /* ... */ }
```

练习 5.16：while 循环特别适用于那种条件保持不变、反复执行操作的情况，例如，当未达到文件末尾时不断读取下一个值。for 循环则更像是在按步骤迭代，它的索引值在某个范围内依次变化。根据每种循环的习惯用法各自编写一段程序，然后分别用另一种循环改写。如果只能使用一种循环，你倾向于使用哪种呢？为什么？

练习 5.17：假设有两个包含整数的 vector 对象，编写一段程序，检验其中一个 vector 对象是否是另一个的前缀。为了实现这一目标，对于两个不等长的 vector 对象，只需挑出长度较短的那个，把它的所有元素和另一个 vector 对象比较即可。例如，如果两个 vector 对象的元素分别是 0、1、1、2 和 0、1、1、2、3、5、8，则程序的返回结果应该为真。



5.4.3 范围 for 语句

C++11 新标准引入了一种更简单的 for 语句，这种语句可以遍历容器或其他序列的所有元素。范围 for 语句（range for statement）的语法形式是：

```
for (declaration : expression)
    statement
```

expression 表示的必须是一个序列，比如用花括号括起来的初始值列表（参见 3.3.1 节，第 88 页）、数组（参见 3.5 节，第 101 页）或者 vector 或 string 等类型的对象，这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员（参见 3.4 节，第 95 页）。

declaration 定义一个变量，序列中的每个元素都得能转换成该变量的类型（参见 4.11 节，第 141 页）。确保类型相容最简单的办法是使用 auto 类型说明符（参见 2.5.2 节，第 61 页），这个关键字可以令编译器帮助我们指定合适的类型。如果需要对序列中的元素执行写操作，循环变量必须声明成引用类型。

每次迭代都会重新定义循环控制变量，并将其初始化成序列中的下一个值，之后才会执行 *statement*。像往常一样，*statement* 可以是一条单独的语句也可以是一个块。所有元素都处理完毕后循环终止。

之前我们已经接触过几个这样的循环。接下来的例子将把 vector 对象中的每个元素都翻倍，它涵盖了范围 for 语句的几乎所有语法特征：

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// 范围变量必须是引用类型，这样才能对元素执行写操作
for (auto &r : v)      // 对于 v 中的每一个元素
```

```
r *= 2;           // 将 v 中每个元素的值翻倍
```

for 语句头声明了循环控制变量 r，并把它和 v 关联在一起，我们使用关键字 auto 令编译器为 r 指定正确的类型。由于准备修改 v 的元素的值，因此将 r 声明成引用类型。此时，在循环体内给 r 赋值，即改变了 r 所绑定的元素的值。

范围 for 语句的定义来源于与之等价的传统 for 语句：

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // r 必须是引用类型，这样才能对元素执行写操作
    r *= 2;          // 将 v 中每个元素的值翻倍
}
```

学习了范围 for 语句的原理之后，我们也就不难理解为什么在 3.3.2 节（第 90 页）强调不能通过范围 for 语句增加 vector 对象（或者其他容器）的元素了。在范围 for 语句中，预存了 end() 的值。一旦在序列中添加（删除）元素，end 函数的值就可能变得无效了（参见 3.4.1 节，第 98 页）。关于这一点，将在 9.3.6 节（第 315 页）做更详细的介绍。189

5.4.4 do while 语句

do while 语句（do while statement）和 while 语句非常相似，唯一的区别是，do while 语句先执行循环体后检查条件。不管条件的值如何，我们都至少执行一次循环。do while 语句的语法形式如下所示：

```
do
    statement
    while (condition);
```



do while 语句应该在括号包围起来的条件后面用一个分号表示语句结束。

在 do 语句中，求 condition 的值之前首先执行一次 statement，condition 不能为空。如果 condition 的值为假，循环终止；否则，重复循环过程。condition 使用的变量必须定义在循环体之外。

我们可以使用 do while 循环（不断地）执行加法运算：

```
// 不断提示用户输入一对数，然后求其和
string rsp; // 作为循环的条件，不能定义在 do 的内部
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
        << " = " << val1 + val2 << "\n\n"
        << "More? Enter yes or no: ";
    cin >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

循环首先提示用户输入两个数字，然后输出它们的和并询问用户是否继续。条件部分检查用户做出的回答，如果用户没有回答，或者用户的回答以字母 n 开始，循环都将终止。否则循环继续执行。

因为对于 do while 来说先执行语句或者块，后判断条件，所以不允许在条件部分

定义变量:

```
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // 错误: 将变量声明放在了 do 的条件部分
```

如果允许在条件部分定义变量, 则变量的使用出现在定义之前, 这显然是不合常理的!

5.4.4 节练习

练习 5.18: 说明下列循环的含义并改正其中的错误。

```
(a) do
    int v1, v2;
    cout << "Please enter two numbers to sum:" ;
    if (cin >> v1 >> v2)
        cout << "Sum is: " << v1 + v2 << endl;
    while (cin);
(b) do {
    // ...
} while (int ival = get_response());
(c) do {
    int ival = get_response();
} while (ival);
```

练习 5.19: 编写一段程序, 使用 do while 循环重复地执行下述任务: 首先提示用户输入两个 string 对象, 然后挑出较短的那个并输出它。

5.5 跳转语句

跳转语句中断当前的执行过程。C++语言提供了 4 种跳转语句: break、continue、goto 和 return。本章介绍前三种跳转语句, return 语句将在 6.3 节 (第 199 页) 进行介绍。

5.5.1 break 语句

break 语句 (break statement) 负责终止离它最近的 while、do while、for 或 switch 语句, 并从这些语句之后的第一条语句开始继续执行。

break 语句只能出现在迭代语句或者 switch 语句内部 (包括嵌套在此类循环里的语句或块的内部)。break 语句的作用范围仅限于最近的循环或者 switch:

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // 处理到第一个空白为止
        for (auto it = buf.begin() + 1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #!, 离开 for 循环
        }
    }
}
```

```

    }
    // break #1 将控制权转移到这里
    // 剩余的'-'处理:
    break; // #2, 离开 switch 语句
    case '+':
        //...
    } // 结束 switch
    // 结束 switch: break #2 将控制权转移到这里
} // 结束 while

```

< 191

标记为#1 的 break 语句负责终止连字符 case 标签后面的 for 循环。它不但不会终止 switch 语句，甚至连当前的 case 分支也终止不了。接下来，程序继续执行 for 循环之后的第一条语句，这条语句可能接着处理连字符的情况，也可能是另一条用于终止当前分支的 break 语句。

标记为#2 的 break 语句负责终止 switch 语句，但是不能终止 while 循环。执行完这个 break 后，程序继续执行 while 的条件部分。

5.5.1 节练习

练习 5.20: 编写一段程序，从标准输入中读取 string 对象的序列直到连续出现两个相同的单词或者所有单词都读完为止。使用 while 循环一次读取一个单词，当一个单词连续出现两次时使用 break 语句终止循环。输出连续重复出现的单词，或者输出一个消息说明没有任何单词是连续重复出现的。

5.5.2 continue 语句

continue 语句 (continue statement) 终止最近的循环中的当前迭代并立即开始下一次迭代。continue 语句只能出现在 for、while 和 do while 循环的内部，或者嵌套在此类循环里的语句或块的内部。和 break 语句类似的是，出现在嵌套循环中的 continue 语句也仅作用于离它最近的循环。和 break 语句不同的是，只有当 switch 语句嵌套在迭代语句内部时，才能在 switch 里使用 continue。

continue 语句中断当前的迭代，但是仍然继续执行循环。对于 while 或者 do while 语句来说，继续判断条件的值；对于传统的 for 循环来说，继续执行 for 语句头的 expression；而对于范围 for 语句来说，则是用序列中的下一个元素初始化循环控制变量。

例如，下面的程序每次从标准输入中读取一个单词。循环只对那些以下画线开头的单词感兴趣，其他情况下，我们直接终止当前的迭代并获取下一个单词：

```

string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // 接着读取下一个输入
    // 程序执行过程到了这里？说明当前的输入是以下画线开始的；接着处理 buf.....
}

```

5.5.2 节练习

练习 5.21: 修改 5.5.1 节（第 171 页）练习题的程序，使其找到的重复单词必须以大写字母开头。

< 192

5.5.3 goto 语句

goto 语句 (goto statement) 的作用是从 **goto** 语句无条件跳转到同一函数内的另一条语句。



不要在程序中使用 **goto** 语句，因为它使得程序既难理解又难修改。

goto 语句的语法形式是

```
goto label;
```

其中，*label* 是用于标识一条语句的标示符。带标签语句 (labeled statement) 是一种特殊的语句，在它之前有一个标示符以及一个冒号：

```
end: return; // 带标签语句，可以作为 goto 的目标
```

标签标示符独立于变量或其他标示符的名字，因此，标签标示符可以和程序中其他实体的标示符使用同一个名字而不会相互干扰。**goto** 语句和控制权转向的那条带标签的语句必须位于同一个函数之内。

和 **switch** 语句类似，**goto** 语句也不能将程序的控制权从变量的作用域之外转移到作用域之内：

```
// ...
goto end;
int ix = 10; // 错误：goto 语句绕过了一个带初始化的变量定义
end:
// 错误：此处的代码需要使用 ix，但是 goto 语句绕过了它的声明
ix = 42;
```

向后跳过一个已经执行的定义是合法的。跳回到变量定义之前意味着系统将销毁该变量，然后重新创建它：

```
// 向后跳过一个带初始化的变量定义是合法的
begin:
int sz = get_size();
if (sz <= 0) {
    goto begin;
}
```

在上面的代码中，**goto** 语句执行后将销毁 *sz*。因为跳回到 *begin* 的动作跨过了 *sz* 的定义语句，所以 *sz* 将重新定义并初始化。

193

5.5.3 节练习

练习 5.22：本节的最后一个例子跳回到 *begin*，其实使用循环能更好地完成该任务。重写这段代码，注意不再使用 **goto** 语句。

5.6 try 语句块和异常处理

异常是指存在于运行时的反常行为，这些行为超出了函数正常功能的范围。典型的异常包括失去数据库连接以及遇到意外输入等。处理反常行为可能是设计所有系统最难的一部分。

当程序的某部分检测到一个它无法处理的问题时，需要用到异常处理。此时，检测出问题的部分应该发出某种信号以表明程序遇到了故障，无法继续下去了，而且信号的发出方无须知道故障将在何处得到解决。一旦发出异常信号，检测出问题的部分也就完成了任务。

如果程序中含有可能引发异常的代码，那么通常也会有专门的代码处理问题。例如，如果程序的问题是输入无效，则异常处理部分可能会要求用户重新输入正确的数据；如果丢失了数据库连接，会发出报警信息。

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持。在 C++ 语言中，异常处理包括：

- **throw 表达式 (throw expression)**，异常检测部分使用 throw 表达式来表示它遇到了无法处理的问题。我们说 throw 引发 (raise) 了异常。
- **try 语句块 (try block)**，异常处理部分使用 try 语句块处理异常。try 语句块以关键字 try 开始，并以一个或多个 catch 子句 (catch clause) 结束。try 语句块中代码抛出的异常通常会被某个 catch 子句处理。因为 catch 子句“处理”异常，所以它们也被称作异常处理代码 (exception handler)。
- 一套异常类 (exception class)，用于在 throw 表达式和相关的 catch 子句之间传递异常的具体信息。

在本节的剩余部分，我们将分别介绍异常处理的这三个组成部分。在 18.1 节（第 684 页）还将介绍更多关于异常的知识。

5.6.1 throw 表达式

程序的异常检测部分使用 throw 表达式引发一个异常。throw 表达式包含关键字 throw 和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。throw 表达式后面通常紧跟一个分号，从而构成一条表达式语句。

举个简单的例子，回忆 1.5.2 节（第 20 页）把两个 Sales_item 对象相加的程序。◀ 194这个程序检查它读入的记录是否是关于同一种书籍的，如果不是，输出一条信息然后退出。

```
Sales_item item1, item2;
cin >> item1 >> item2;
// 首先检查 item1 和 item2 是否表示同一种书籍
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // 表示成功
} else {
    cerr << "Data must refer to same ISBN" << endl;
    return -1; // 表示失败
}
```

在真实的程序中，应该把对象相加的代码和用户交互的代码分离开来。此例中，我们改写程序使得检查完成后不再直接输出一条信息，而是抛出一个异常：

```
// 首先检查两条数据是否是关于同一种书籍的
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// 如果程序执行到了这里，表示两个 ISBN 是相同的
cout << item1 + item2 << endl;
```

在这段代码中，如果 ISBN 不一样就抛出一个异常，该异常是类型 `runtime_error` 的对象。抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码。

类型 `runtime_error` 是标准库异常类型的一种，定义在 `stdexcept` 头文件中。关于标准库异常类型更多的知识将在 5.6.3 节（第 176 页）介绍。我们必须初始化 `runtime_error` 的对象，方式是给它提供一个 `string` 对象或者一个 C 风格的字符串（参见 3.5.4 节，第 109 页），这个字符串中有一些关于异常的辅助信息。

5.6.2 try 语句块

`try` 语句块的通用语法形式是

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

`try` 语句块一开始是关键字 `try`，随后紧跟着一个块，这个块就像大多数时候那样是花括号括起来的语句序列。

195 跟在 `try` 块之后的是一个或多个 `catch` 子句。`catch` 子句包括三部分：关键字 `catch`、括号内一个（可能未命名的）对象的声明（称作 **异常声明**，`exception declaration`）以及一个块。当选中了某个 `catch` 子句处理异常之后，执行与之对应的块。`catch` 一旦完成，程序跳转到 `try` 语句块最后一个 `catch` 子句之后的那条语句继续执行。

`try` 语句块中的 `program-statements` 组成程序的正常逻辑，像其他任何块一样，`program-statements` 可以有包括声明在内的任意 C++ 语句。一如往常，`try` 语句块内声明的变量在块外部无法访问，特别是在 `catch` 子句内也无法访问。

编写处理代码

在之前的例子里，我们使用了一个 `throw` 表达式以避免把两个代表不同书籍的 `Sales_item` 相加。我们假设执行 `Sales_item` 对象加法的代码是与用户交互的代码分离开来的。其中与用户交互的代码负责处理发生的异常，它的形式可能如下所示：

```
while (cin >> item1 >> item2) {
    try {
        // 执行添加两个 Sales_item 对象的代码
        // 如果添加失败，代码抛出一个 runtime_error 异常
    } catch (runtime_error err) {
        // 提醒用户两个 ISBN 必须一致，询问是否重新输入
        cout << err.what()
            << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
```

```
    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。196 如果上一节编写的代码抛出异常，则本节的 `catch` 子句输出

```
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

<code>exception</code>	最常见的问题
<code>runtime_error</code>	只有在运行时才能检测出的问题
<code>range_error</code>	运行时错误：生成的结果超出了有意义的值域范围
<code>overflow_error</code>	运行时错误：计算上溢
<code>underflow_error</code>	运行时错误：计算下溢
<code>logic_error</code>	程序逻辑错误
<code>domain_error</code>	逻辑错误：参数对应的结果值不存在
<code>invalid_argument</code>	逻辑错误：无效参数
<code>length_error</code>	逻辑错误：试图创建一个超出该类型最大长度的对象
<code>out_of_range</code>	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，则异常将被该 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--;
    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数求值。

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。

Best Practices

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:

```
$ 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。

Note

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。



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 */ } ◀ 213
void fcn(int i) { /* ... */ } // 错误：重复定义了 fcn(int)
```

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

214 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。

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 ("abcd", '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
// 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
```

<215

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：

```
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。

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：

```
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：

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

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。

C++
11

表 6.1: initializer_list 提供的操作

<code>initializer_list<T> lst;</code>	默认初始化; T 类型元素的空列表
<code>initializer_list<T> lst{a,b,c...};</code>	<code>lst</code> 的元素数量和初始值一样多; <code>lst</code> 的元素是对应初始值的副本; 列表中的元素是 <code>const</code>
<code>lst2(lst)</code>	拷贝或赋值一个 <code>initializer_list</code> 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
<code>lst.size()</code>	列表中的元素数量
<code>lst.begin()</code>	返回指向 <code>lst</code> 中首元素的指针
<code>lst.end()</code>	返回指向 <code>lst</code> 中尾元素下一位位置的指针

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`。



WARNING 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();
    // 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。



WARNING 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](#)。

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。◀ 227
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。

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）：

```

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 231 会根据传递的实参类型推断想要的是哪个函数：

```
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，但实际上是一样的：

```
// 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。

- int calc(int, int);
int calc(const int, const int);
- int get();
double get();
- 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。

Note

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

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 backrnd = ' '');
```

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:

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

```
assert(expr);
```

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

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

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

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

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

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

NDEBUG 预处理变量

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

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

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

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

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

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

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

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

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

- FILE— 存放文件名的字符串字面值。
- LINE— 存放当前行号的整型字面值。
- TIME— 存放文件编译时间的字符串字面值。
- DATE— 存放文件编译日期的字符串字面值。

可以使用这些常量在错误消息中提供更多信息：

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

```

<< " at line " << __LINE__ << endl
<< "         Compiled on " << __DATE__ -
<< " at " << __TIME__ << endl
<< "         Word read was \""
<< "\": 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, <245 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。

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数集不合理。

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，最简单的方法是使用类型别名：

```
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，牢记当我们使用 `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。

链接 (link) 是一个编译过程，负责把若干

< 252

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

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
        }
    }
    print(cout, total); // 输出最后一条交易
} else {
    cerr << "No data?!" << endl; // 没有输入任何信息
}
```

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。

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;
    // 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&);
```

257

Note

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



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，

```
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。

261 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。



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

```