

习题

习题 6.21 修改 6.10 节最后一个习题的程序，使得它只寻找以大写字母开头的连续出现的单词。

6.12 goto 语句

goto 语句提供了函数内部的无条件跳转，实现从 goto 语句跳转到同一函数内某个带标号的语句。



从上世纪 60 年代后期开始，不主张使用 goto 语句。goto 语句使跟踪程序控制流程变得很困难，并且使程序难以理解，也难以修改。所有使用 goto 的程序都可以改写为不用 goto 语句，因此也就没有必要使用 goto 语句了。

goto 语句的语法规则如下：

```
goto label;
```

其中 *label* 是用于标识带标号的语句的标识符。在任何语句前提供一个标识符和冒号，即得带标号的语句 (labeled statement)：

```
end: return; // labeled statement, may be target of a goto
```

形成标号 (label) 的标识符只能用作 goto 的目标。因为这个原因，标号标识符可以与变量名以及程序里的其他标识符一样，不与别的标识符重名。goto 语句和获得所转移的控制权的带标号的语句必须位于同一个函数内。

goto 语句不能跨越变量的定义语句向前跳转：

```
// ...
goto end;
int ix = 10; // error: goto bypasses declaration statement
end:
// error: code here could use ix but the goto bypassed its declaration
ix = 42;
```

如果确实需要在 goto 和其跳转对应的标号之间定义变量，则定义必须放在一个块语句中：

```
// ...
goto end; // ok: jumps to a point where ix is not defined
{
    int ix = 10;
    // ... code using ix
}
end: // ix no longer visible here
```

向后跳过已经执行的变量定义语句则是合法的。为什么？向前跳过未执行的变量定义语句，意味着变量可能在没有定义的情况下使用。向后跳回到一个变量定义之前，则会使系统撤销这个变量，然后再重新创建它：

```
// backward jump over declaration statement ok
begin:
```

```

    int sz = get_size();
    if (sz <= 0) {
        goto begin;
    }

```

注意：执行 goto 语句时，首先撤销变量 sz，然后程序的控制流程跳转到带 begin: 标号的语句继续执行，再次重新创建和初始化 sz 变量。

习题

习题 6.22 对于本节的最后一个例子，跳回到 begin 标号的功能可以用循环更好地实现。请不使用 goto 语句重写这段代码。

6.13 try 块和异常处理

在设计各种软件系统的过程中，处理程序中的错误和其他反常行为是最困难的部分之一。像通信交换机和路由器这类长期运行的交互式系统必须将 90% 的程序代码用于实现错误检测和错误处理。随着基于 Web 的应用程序在运行时不确定性的增多，越来越多的程序员更加注重错误的处理。

215

异常就是运行时出现的不正常，例如运行时耗尽了内存或遇到意外的非法输入。异常存在于程序的正常功能之外，并要求程序立即处理。

在设计良好的系统中，异常是程序错误处理的一部分。当程序代码检查到无法处理的问题时，异常处理就特别有用。在这些情况下，检测出问题的那部分程序需要一种方法把控制权转到可以处理这个问题的那部分程序。错误检测程序还必须指出具体出现了什么问题，并且可能需要提供一些附加信息。

异常机制提供程序中错误检测与错误处理部分之间的通信。C++ 的异常处理中包括：

- **throw 表达式** (throw expression)，错误检测部分使用这种表达式来说明遇到了不可处理的错误。可以说，throw 引发 (raise) 了异常条件。
- **try 块** (try block)，错误处理部分使用它来处理异常。try 语句块以 try 关键字开始，并以一个或多个 **catch 子句** (catch clause) 结束。在 try 块中执行的代码所抛出 (throw) 的异常，通常会被其中一个 catch 子句处理。由于它们“处理”异常，catch 子句也称为 **处理代码** (handler)。
- 由标准库定义的一组 **异常类** (exception class)，用来在 throw 和相应的 catch 之间传递有关的错误信息。

在本节接下来的部分将要介绍这三种异常处理的构成。而 17.1 节将会进一步了解异常的相关内容。

6.13.1 throw 表达式

系统通过 throw 表达式抛出异常。throw 表达式由关键字 throw 以及尾随的表达式组成，通常以分号结束，这样它就成为了表达式语句。throw 表达式的类型决定了所抛出异常的类型。

回顾 1.5.2 节将两个 `Sales_item` 类型对象相加的程序，就是一个简单的例子。该程序检查读入的记录是否来自同一本书。如果不是，就输出一条信息然后退出程序。

```
Sales_item item1, item2;
std::cin >> item1 >> item2;
// first check that item1 and item2 represent the same book
if (item1.same_isbn(item2)) {
    std::cout << item1 + item2 << std::endl;
    return 0; // indicate success
} else {
    std::cerr << "Data must refer to same ISBN"
               << std::endl;
    return -1; // indicate failure
}
```

在使用 `Sales_item` 的更简单的程序中，把将对象相加的部分和负责跟用户交互的部分分开。在这个例子中，用 `throw` 抛出异常来改写检测代码：

```
// first check that data is for the same item
if (!item1.same_isbn(item2))
    throw runtime_error("Data must refer to same ISBN");
// ok, if we're still here the ISBNs are the same
std::cout << item1 + item2 << std::endl;
```

这段代码检查 ISBN 对象是否不相同。如果不同的话，停止程序的执行，并将控制转移给处理这种错误的处理代码。

`throw` 语句使用了一个表达式。在本例中，该表达式是 `runtime_error` 类型的对象。`runtime_error` 类型是标准库异常类中的一种，在 `stdexcept` 头文件中定义。在后续章节中很快会更详细地介绍这些类型。我们通过传递 `string` 对象来创建 `runtime_error` 对象，这样就可以提供更多关于所出现问题的相关信息。

6.13.2 try 块

`try` 块的通用语法形式是：

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

`try` 块以关键字 `try` 开始，后面是用花括号括起来的语句序列块。`try` 块后面是一个或多个 `catch` 子句。每个 `catch` 子句包括三部分：关键字 `catch`，圆括号内单个类型或者单个对象的声明——称为异常说明符(exception specifier)，以及通常用花括号括起来的语句块。如果选择了一个 `catch` 子句来处理异常，则执行相关的块语句。一旦 `catch` 子句执行结束，程序流程立即继续执行紧随着最后一个 `catch` 子句的语句。

`try` 语句内的 *program-statement* 形成程序的正常逻辑。这里面可以包含任意 C++ 语句，包括变量声明。与其他块语句一样，`try` 块引入局部作用域，在 `try` 块中声明的变量，包括 `catch` 子句中声明的变量，不能在 `try` 外面引用。

216

1. 编写处理代码

217 在前面的例子中，使用了 `throw` 来避免将两个表示不同书的 `Sales_item` 对象相加。想象一下将 `Sales_item` 对象相加的那部分程序与负责与用户交流的那部分是分开的，则与用户交互的部分也许会包含下面用于处理所捕获异常的代码：

```
while (cin >> item1 >> item2) {
    try {
        // execute code that will add the two Sales_items
        // if the addition fails, the code throws a runtime_error exception
    } catch (runtime_error err) {
        // remind the user that ISBN must match and prompt for another pair
        cout << err.what()
            << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
        if (cin && c == 'n')
            break;          // break out of the while loop
    }
}
```

关键字 `try` 后面是一个块语句。这个块语句调用处理 `Sales_item` 对象的程序部分。这部分也可能会抛出 `runtime_error` 类型的异常。

上述 `try` 块提供单个 `catch` 子句，用来处理 `runtime_error` 类型的异常。在执行 `try` 块代码的过程中，如果在 `try` 块中的代码抛出 `runtime_error` 类型的异常，则处理这类异常的动作在 `catch` 后面的块语句中定义。本例中，`catch` 输出信息并且询问用户是否继续进行异常处理。如果用户输入 'n'，则结束 `while`；否则继续循环，读入两个新的 `Sales_item` 对象。

通过输出 `err.what()` 的返回值提示用户。大家都知道 `err` 返回 `runtime_error` 类型的值，因此可以推断出 `what` 是 `runtime_error` 类的一个成员函数（1.5.2 节）。每一个标准库异常类都定义了名为 `what` 的成员函数。这个函数不需要参数，返回 C 风格字符串。在出现 `runtime_error` 的情况下，`what` 返回的 C 风格字符串，是用于初始化 `runtime_error` 的 `string` 对象的副本。如果在前面章节描述的代码抛出异常，那么执行这个 `catch` 将输出：

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

2. 函数在寻找处理代码的过程中退出

在复杂的系统中，程序的执行路径也许在遇到抛出异常的代码之前，就已经经过了多个 `try` 块。例如，一个 `try` 块可能调用了包含另一 `try` 块的函数，它的 `try` 块又调用了含有 `try` 块的另一个函数，如此类推。

218 寻找处理代码的过程与函数调用链刚好相反。抛出一个异常时，首先要搜索的是抛出异常的函数。如果没有找到匹配的 `catch`，则终止这个函数的执行，并在调用这个函数的函数中寻找匹配的 `catch`。如果仍然没有找到相应的处理代码，该函数同样要终止，搜索调用它的函数。如此类推，继续按执行路径回退，直到找到适当类型的 `catch` 为止。

如果不存在处理该异常的 `catch` 子句，程序的运行就要跳转到名为 `terminate` 的标准库函数，该函数在 `exception` 头文件中定义。这个标准库函数的行为依赖于系统，通常情况下，它的执

行将导致程序非正常退出。

在程序中出现的异常，如果没有经 try 块定义，则都以相同的方式来处理：毕竟，如果没有任何 try 块，也就没有捕获异常的处理代码（catch 子句）。此时，如果发生了异常，系统将自动调用 terminate 终止程序的执行。

习题

- 习题 6.23 bitset 类提供 to_ulong 操作，如果 bitset 提供的位数大于 unsigned long 的长度时，抛出一个 overflow_error 异常。编写产生这种异常的程序。
- 习题 6.24 修改上述的程序，使它能捕获这种异常并输出提示信息。

6.13.3 标准异常

C++标准库定义了一组类，用于报告在标准库中的函数遇到的问题。程序员可在自己编写的程序中使用这些标准异常类。标准库异常类定义在四个头文件中：

- (1) exception 头文件定义了最常见的异常类，它的类名是 exception。这个类只通知异常的产生，但不会提供更多的信息。
- (2) stdexcept 头文件定义了几种常见的异常类。这些类型在表 6-1 中列出。
- (3) new 头文件定义了 bad_alloc 异常类型，提供因无法分配内存而由 new（5.11 节）抛出的异常。
- (4) type_info 头文件定义了 bad_cast 异常类型，这种类型将在 18.2 节讨论。

表 6-1 在<stdexcept>头文件中定义的标准异常类	
exception	最常见的问题
runtime_error	运行时错误：仅在运行时才能检测到的问题
range_error	运行时错误：生成的结果超出了有意义的值域范围
overflow_error	运行时错误：计算上溢
underflow_error	运行时错误：计算下溢
logic_error	逻辑错误：可在运行前检测到的问题
domain_error	逻辑错误：参数的结果值不存在
invalid_argument	逻辑错误：不合适的参数
length_error	逻辑错误：试图生成一个超出该类型最大长度的对象
out_of_range	逻辑错误：使用一个超出有效范围的值

标准库异常类

标准库异常类只提供很少的操作，包括创建、复制异常类型对象以及异常类型对象的赋值。exception、bad_alloc 以及 bad_cast 类型只定义了默认构造函数（2.3.4 节），无法在创建这些类型的对象时为它们提供初值。其他的异常类型则只定义了一个使用 string 初始化式的构造函数。当需要定义这些异常类型的对象时，必须提供一个 string 参数。string 初始化式用于为所发生的错误提供更多的信息。

异常类型只定义了一个名为 `what` 的操作。这个函数不需要任何参数，并且返回 `const char*` 类型的值。它返回的指针指向一个 C 风格字符串（4.3 节）。使用 C 风格字符串的目的是为所抛出的异常提供更详细的文字描述。

`what` 函数所返回的指针指向 C 风格字符数组的内容，这个数组的内容依赖于异常对象的类型。对于接受 `string` 初始化式的异常类型，`what` 函数将返回该 `string` 作为 C 风格字符数组。对于其他异常类型，返回的值则根据编译器的变化而不同。

6.14 使用预处理器进行调试

2.9.2 节介绍了如何使用预处理变量来避免重复包含头文件。C++ 程序员有时也会使用类似的技术有条件地执行用于调试的代码。这种想法是：程序所包含的调试代码仅在开发过程中执行。当应用程序已经完成，并且准备提交时，就会将调试代码关闭。可使用 `NDEBUG` 预处理变量实现有条件的调试代码：

```
int main()
{
#ifdef NDEBUG
cerr << "starting main" << endl;
#endif
// ...
```

220

如果 `NDEBUG` 未定义，那么程序就会将信息写到 `cerr` 中。如果 `NDEBUG` 已经定义了，那么程序执行时将会跳过 `#ifdef` 和 `#endif` 之间的代码。

默认情况下，`NDEBUG` 未定义，这也就意味着必须执行 `#ifdef` 和 `#endif` 之间的代码。在开发程序的过程中，只要保持 `NDEBUG` 未定义就会执行其中的调试语句。开发完成后，要将程序交付给客户时，可通过定义 `NDEBUG` 预处理变量，（有效地）删除这些调试语句。大多数的编译器都提供定义 `NDEBUG` 的命令行选项：

```
$ CC -DNDEBUG main.C
```

这样的命令行等效于在 `main.c` 的开头提供 `#define NDEBUG` 预处理命令。

预处理器还定义了其余四种在调试时非常有用的常量：

```
_ _FILE_ _ 文件名
_ _LINE_ _ 当前行号
_ _TIME_ _ 文件被编译的时间
_ _DATE_ _ 文件被编译的日期
```

可使用这些常量在错误消息中提供更多的信息：

```
if (word.size() < threshold)
    cerr << "Error: " << _ _FILE_ _
        << " : line " << _ _LINE_ _ << endl
        << "      Compiled on " << _ _DATE_ _
        << " at " << _ _TIME_ _ << endl
        << "      Word read was " << word
        << ": Length too short" << endl;
```

如果给这个程序提供一个比 threshold 短的 string 对象, 则会产生下面的错误信息:

```
Error: wdebug.cc : line 21
    Compiled on Jan 12 2005 at 19:44:40
    Word read was "foo": Length too short
```

另一个常见的调试技术是使用 NDEBUG 预处理变量以及 assert (断言) 预处理宏 (preprocessor macro)。assert 宏是在 cassert 头文件中定义的, 所有使用 assert 的文件都必须包含这个头文件。

预处理宏有点像函数调用。assert 宏需要一个表达式作为它的条件:

```
assert(expr)
```

只要 NDEBUG 未定义, assert 宏就求解条件表达式 *expr*, 如果结果为 false, assert 输出信息并且终止程序的执行。如果该表达式有一个非零 (例如, true) 值, 则 assert 不做任何操作。

与异常不同 (异常用于处理程序执行时预期要发生的错误), 程序员使用 assert 来测试“不可能发生”的条件。例如, 对于处理输入文本的程序, 可以预测全部给出的单词都比指定的阈值长。那么程序可以包含这样一个语句:

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

在测试过程中, assert 等效于检验数据是否总是具有预期的大小。一旦开发和测试工作完成, 程序就已经建立好, 并且定义了 NDEBUG。在成品代码中, assert 语句不做任何工作, 因此也没有任何运行时代价。当然, 也不会引起任何运行时检查。assert 仅用于检查确实不可能的条件, 这只对程序的调试有帮助, 但不能用来代替运行时的逻辑检查, 也不能代替对程序可能产生的错误的检测。

习题

习题 6.25 修改 6.11 节习题所编写的程序, 使其可以有条件地输出运行时的信息。例如, 可以输出每一个读入的单词, 用来判断循环是否正确地找到第一个连续出现的以大写字母开头的单词。分别在打开和关闭调试器的情况下编译和运行这个程序。

习题 6.26 下面循环会导致什么现象的发生:

```
string s;
while (cin >> s) {
    assert(cin);
    // process s
}
```

解释这种用法是否是 assert 宏的一种恰当应用。

习题 6.27 解释下面的循环:

```
string s;
while (cin >> s && s != sought) { } // empty body
assert(cin);
// process s
```

221

222

小结

C++提供了种类相当有限的语句，其中大多数都会影响程序的控制流：

while、for 以及 do while 语句，实现反复循环；

if 和 switch，提供条件分支结构；

continue，终止当次循环；

break，退出一个循环或 switch 语句；

goto，将控制跳转到某个标号语句；

try、catch 语句，实现 try 块的定义，该块语句包含一个可能抛出异常的语句序列，catch 子句则用来处理在 try 块里抛出的异常；

throw 表达式，用于退出代码块的执行，将控制转移给相关的 catch 子句。

当然还有将在第 7 章介绍的 return 语句。

此外，C++还提供表达式语句和声明语句。表达式语句用于求解表达式。变量的声明和定义则已在第 2 章讲述过了。

术语

assert 一种预处理宏，使用单个表达式作为断言条件。如果预处理变量 NDEBUG 没有定义，则 assert 将求解它的条件表达式。若条件为 false，assert 输出信息并终止程序的执行。
block (块) 包含在一对花括号里的语句序列。在语法上，块就是单语句，可出现在任何单语句可以出现的地方。

break statement (break 语句) 一种语句，能够终止最近的循环或者 switch 语句的执行，将控制权转交给被终止的循环或者 switch 后的第一条语句。

case label (case 标号) switch 语句中跟在关键字 case 后的整型常量值。在同一个 switch 结构中不能有任何两个标号拥有相同的常量值。如果 switch 条件表达式的值与其中某个标号的值相等，则控制权转移到匹配标号后面的第一条语句，从这条语句开始依次继续各个语句，直到遇到 break 或者到达

switch 结尾为止。

catch clause (catch 子句) 一种语句，包括关键字 catch、圆括号内的异常说明符以及一个块语句。catch 子句中的代码实现某种异常的处理，该异常由圆括号内的异常说明符定义。

compound statement (复合语句) 块的同义词。

continue statement (continue 语句) 一种语句，能够结束最近的循环结构的当次循环迭代，将控制流转移到 while 或 do 的循环条件表达式，或者 for 语句头中第三个表达式。

dangling else (悬垂 else) 一个通俗术语，指出如何处理嵌套 if 语句中 if 多于 else 时发生的二义性问题。C++中，else 总是与最近的未匹配的 if 配对。注意使用花括号能有效地隐藏内层 if，使程序员可以控制给定的 else 与哪个 if 相匹配。

declaration statement (声明语句) 定义或者声明变量的语句。声明已在第2章中介绍。

default label (default 标号) switch 语句中的一种标号，当计算 switch 条件所得的值与所有 case 标号的值都不匹配时，则执行 default 标号关联的语句。

exception classes (异常类) 标准库定义的一组描述程序错误的类。表 6-1 列出了常见的异常。

exception handler (异常处理代码) 一段代码，用于处理程序某个部分所引起的异常。是 catch 子句的同义词。

exception specifier (异常说明符) 对象或类型的声明，用于指出当前的 catch 子句能处理的异常类型。

expression statement (表达式语句) 一种语句，由后接分号的表达式构成。表达式语句用于表达式的求解。

flow of control (控制流) 程序的执行路径。

goto statement (goto 语句) 一种语句，能够使程序控制流程无条件跳转到指定标号语句。goto 扰乱了程序内部的控制流，应尽可能避免使用。

if else statement (if else 语句) 一种语句，有条件地执行 if 或 else 后的代码，如何执行取决于条件表达式的真值。

if statement (if 语句) 基于指定条件值的条件分支语句。如果条件为真，则执行 if 语句体；否则，控制流转到 if 后面的语句。

labeled statement (带标号的语句) 以标号开头的语句。标号是后面带一个冒号的标识符。

null statement (空语句) 空白的语句。其语法形式为单个分号。

preprocessor macro (预处理宏) 与预处理器定义的设施相似的函数。assert 是一个宏。现代 C++ 程序很少使用预处理宏。

raise (引发) 常用作 throw 的同义词。C++ 程序员所说的“抛出 (throwing)”或者“引发 (raising)”异常表示一样的含义。

switch statement (switch 语句) 从计算关键字 switch 后面的表达式开始执行的条件分支语句。程序的控制流转跳到与表达式值匹配的 case 标号所标记的标号语句。如果没有匹配的标号，则执行 default 标号标记的分支，如果没有提供 default 分支则结束 switch 语句的执行。

terminate 异常未被捕获时调用的标准库函数。通常会终止程序的执行。

throw expression (throw 表达式) 中断当前执行路径的表达式。每个 throw 都会抛出一个对象，并将控制转移到最近的可处理该类型异常的 catch 子句。

try block (try 块) 跟在关键字 try 后面的块，以及一个或多个 catch 子句。如果 try 块中的代码产生了异常，而且该异常类型与其中某个 catch 子句匹配，则执行这个 catch 子句的语句处理这个异常。否则，异常将由外围 try 块处理，或者终止程序。

while loop (while 循环) 当指定条件为 true 时，执行目标代码的控制语句。根据条件的真值，目标代码可能执行零次或多次。

第 7 章

函 数

目录

7.1 函数的定义.....	196
7.2 参数传递.....	199
7.3 return 语句	211
7.4 函数声明.....	217
7.5 局部对象.....	220
7.6 内联函数.....	221
7.7 类的成员函数.....	222
7.8 重载函数.....	228
7.9 指向函数的指针.....	237
小结.....	239
术语.....	240

本章将介绍函数（function）的定义和声明。其中讨论了如何给函数传递参数以及如何从函数返回值。然后具体分析三类特殊的函数：内联（**inline**）函数、类成员函数和重载函数。最后以一个更高级的话题“函数指针”来结束全章。

函数可以看作程序员定义的操作。与内置操作符相同的是，每个函数都会实现一系列的计算，然后（大多数时候）生成一个计算结果。但与操作符不同的是，函数有自己的函数名，而且操作数没有数量限制。与操作符一样，函数可以重载，这意味着同样的函数名可以对应多个不同的函数。

7.1 函数的定义

函数由函数名以及一组操作数类型唯一地表示。函数的操作数，也即形参（parameter），在一对圆括号中声明，形参与形参之间以逗号分隔。函数执行的运算在一个称为函数体（function body）的块语句中定义。每一个函数都有一个相关联的返回类型（return type）。

考虑下面的例子，这个函数用来求出两个 int 型数的最大公约数：

```
// return the greatest common divisor
int gcd(int v1, int v2)
{
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

这里，定义了一个名为 gcd 的函数，该函数返回一个 int 型值，并带有两个 int 型形参。调用 gcd 函数时，必须提供两个 int 型值传递给函数，然后将得到一个 int 型的返回值。

1. 函数的调用

C++语言使用调用操作符（call operator）（即一对圆括号）实现函数的调用。正如其他操作符一样，调用操作符需要操作数并产生一个结果。调用操作符的操作数是函数名和一组（有可能是空的）由逗号分隔的实参（argument）。函数调用的结果类型就是函数返回值的类型，该运算的结果本身就是函数的返回值：

```
// get values from standard input
cout << "Enter two values: \n";
int i, j;
cin >> i >> j;
// call gcd on arguments i and j
// and print their greatest common divisor
cout << "gcd: " << gcd(i, j) << endl;
```

如果给定 15 和 123 作为程序的输入，程序将输出 3。

函数调用做了两件事情：用对应的实参初始化函数的形参，并将控制权转移给被调用函数。主调函数（calling function）的执行被挂起，被调函数（called function）开始执行。函数的运行以形参的（隐式）定义和初始化开始。也就是说，当我们调用 gcd 时，第一件事就是创建名为 v1 和 v2 的 int 型变量，并将这两个变量初始化为调用 gcd 时传递的实参值。在上例中，v1 的初值为 i，而 v2 则初始化为 j 的值。

2. 函数体是一个作用域

函数体是一个语句块，定义了函数的具体操作。通常，这个块语句包含在一对花括号中，形成了一个新的作用域。和其他的块语句一样，在函数体中可以定义变量。在函数体内定义的变量

只在该函数中才可以访问。这种变量称为局部变量 (local variable)，它们相对于定义它们的函数而言是“局部”的，其名字只能在该函数的作用域中可见。这种变量只在函数运行时存在。7.5 节将详细讨论局部变量。

当执行到 return 语句时，函数调用结束。被调用的函数完成时，将产生一个在 return 语句中指定的结果值。执行 return 语句后，被挂起的主调函数在调用处恢复执行，并将函数的返回值用作求解调用操作符的结果，继续处理在执行调用的语句中所剩余的工作。

3. 形参和实参

类似于局部变量，函数的形参为函数提供了已命名的局部存储空间。它们之间的差别在于形参是在函数的形参表中定义的，并由调用函数时传递给函数的实参初始化。

实参则是一个表达式。它可以是变量或字面值常量，甚至是包含一个或几个操作符的表达式。在调用函数时，所传递的实参个数必须与函数的形参个数完全相同。与初始化式的类型必须与被初始化对象的类型匹配一样，实参的类型也必须与其对应形参的类型完全匹配：实参必须具有与形参类型相同、或者能隐式转换 (5.12 节) 为形参类型的数据类型。本章 7.8.2 节将详细讨论实参与形参的匹配。

7.1.1 函数返回类型

函数的返回类型可以是内置类型 (如 int 或者 double)、类类型或复合类型 (如 int* 或 string*)，还可以是 void 类型，表示该函数不返回任何值。下面的例子列出了一些可能的函数返回类型：

```
bool is_present(int *, int);           // returns bool
int count(const string &, char);       // returns int
Date &calendar(const char*);          // returns reference to Date
void process();                       // process does not return a value
```

227

函数不能返回另一个函数或者内置数组类型，但可以返回指向函数的指针，或指向数组元素的指针的指针：

```
// ok: pointer to first element of the array
int *foo_bar() { /* ... */ }
```

这个函数返回一个 int 型指针，该指针可以指向数组中的一个元素。

7.9 节将介绍有关函数指针的内容。

函数必须指定返回类型

在定义或声明函数时，没有显式指定返回类型是不合法的：

```
// error: missing return type
test(double v1, double v2) { /* ... */ }
```

早期的 C++ 版本可以接受这样的程序，将 test 函数的返回类型隐式地定义为 int 型。但在标准 C++ 中，上述程序则是错误的。



在 C++ 标准化之前，如果缺少显式返回类型，函数的返回值将被假定为 int 型。早期未标准化的 C++ 编译器所编译的程序可能依然含有隐式返回 int 型的函数。

7.1.2 函数形参表

函数形参表可以为空，但不能省略。没有任何形参的函数可以用空形参表或含有单个关键字 `void` 的形参表来表示。例如，下面关于 `process` 的声明是等价的：

```
void process() { /* ... */ }           // implicit void parameter list
```

```
void process(void){ /* ... */ }        // equivalent declaration
```

形参表由一系列用逗号分隔的参数类型和（可选的）参数名组成。如果两个参数具有相同的类型，则其类型必须重复声明：

```
int manip(int v1, v2) { /* ... */ }     // error
int manip(int v1, int v2) { /* ... */ } // ok
```

参数表中不能出现同名的参数。类似地，局部于函数的变量也不能使用与函数的任意参数相同的名字。

228

参数名是可选的，但在函数定义中，通常所有参数都要命名。参数必须在命名后才能使用。

参数类型检查



注解 C++是一种静态强类型语言（2.3节），对于每一次的函数调用，编译时都会检查其实参。

调用函数时，对于每一个实参，其类型都必须与对应的形参类型相同，或具有可被转换（5.12节）为该形参类型的类型。函数的形参表为编译器提供了检查实参需要的类型信息。例如，7.1节定义的 `gcd` 函数有两个 `int` 型的形参：

```
gcd("hello", "world"); // error: wrong argument types
gcd(24312);             // error: too few arguments
gcd(42, 10, 0);         // error: too many arguments
```

以上所有的调用都会导致编译时的错误。在第一个调用中，实参的类型都是 `const char*`，这种类型无法转换为 `int` 型，因此该调用不合法。而第二和第三个调用传递的实参数量有误。在调用该函数时必须提供两个实参，实参数太多或太少都是不合法的。

如果两个实参都是 `double` 类型，又会怎么样呢？调用是否合法？

```
gcd(3.14, 6.29);        // ok: arguments are converted to int
```

在 C++ 中，答案是肯定的：该调用合法！正如 5.12.1 节所示，`double` 型的值可以转换为 `int` 型的值。本例中的函数调用正涉及了这种转换——用 `double` 型的值来初始化 `int` 型对象。因此，把该调用标记为不合法未免过于严格。更确切地说，（通过截断）`double` 型实参被隐式地转换为 `int` 型。由于这样的转换可能会导致精度损失，大多数编译器都会给出警告。对于本例，该调用实际上变为：

```
gcd(3, 6);
```

返回值是 3。

调用函数时传递过多的实参、忽略某个实参或者传递错误类型的实参，几乎肯定会导致严重的运行时错误！对于大程序，在编译时检查出这些所谓的接口错误（`interface error`），将会大大

地缩短“编译-调试-测试”的周期。

习题

习题 7.1 形参和实参有什么区别？

习题 7.2 下列哪些函数是错误的？为什么？请给出修改意见。

```
(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;
```

习题 7.3 编写一个带有两个 int 型形参的函数，产生第一个参数的第二个参数次幂的值。编写程序传递两个 int 数值调用该函数，请检验其结果。

习题 7.4 编写一个函数，返回其形参的绝对值。

7.2 参数传递

每次调用函数时，都会重新创建该函数所有的形参，此时所传递的实参将会初始化对应的形参。



形参的初始化与变量的初始化一样：如果形参具有非引用类型，则复制实参的值，如果形参为引用类型（2.5 节），则它只是实参的别名。

229

7.2.1 非引用形参

普通的非引用类型的参数通过复制对应的实参实现初始化。当用实参副本初始化形参时，函数并没有访问调用所传递的实参本身，因此不会修改实参的值。下面再次观察 gcd 这个函数的定义：

```
// return the greatest common divisor
int gcd(int v1, int v2)
{
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

while 循环体虽然修改了 v1 与 v2 的值，但这些变化仅限于局部参数，而对调用 gcd 函数使用的实参没有任何影响。于是，如果有函数调用

```
gcd(i, j)
```

则 i 与 j 的值不受 gcd 内执行的赋值操作的影响。

230



非引用形参表示对应实参的局部副本。对这类形参的修改仅仅改变了局部副本的值。一旦函数执行结束，这些局部变量的值也就没有了。

1. 指针形参

函数的形参可以是指针（4.2 节），此时将复制实参指针。与其他非引用类型的形参一样，该类形参的任何改变也仅作用于局部副本。如果函数将新指针值赋给形参，主调函数使用的实参指针的值没有改变。

回顾 4.2.3 节的讨论，事实上被复制的指针只影响对指针的赋值。如果函数形参是非 const 类型的指针，则函数可通过指针实现赋值，修改指针所指向对象的值：

```
void reset(int *ip)
{
    *ip = 0;    // changes the value of the object to which ip points
    ip = 0;     // changes only the local value of ip; the argument is unchanged
}
```

调用 reset 后，实参依然保持原来的值，但它所指向的对象的值将变为 0：

```
int i = 42;
int *p = &i;
cout << "i: " << *p << '\n';    // prints i: 42
reset(p);                        // changes *p but not p
cout << "i: " << *p << endl;    // ok: prints i: 0
```

如需保护指针指向的值，则形参需定义为指向 const 对象的指针：

```
void use_ptr(const int *p)
{
    // use_ptr may read but not write to *p
}
```

指针形参是指向 const 类型还是非 const 类型，将影响函数调用所使用的实参。我们既可以用 int* 也可以用 const int* 类型的实参调用 use_ptr 函数；但仅能将 int* 类型的实参传递给 reset 函数。这个差别来源于指针的初始化规则（4.2.5 节）。可以将指向 const 对象的指针初始化为指向非 const 对象，但不可以让指向非 const 对象的指针指向 const 对象。

2. const 形参

在调用函数时，如果该函数使用非引用的非 const 形参，则既可给该函数传递 const 实参也可传递非 const 的实参。例如，可以传递两个 int 型 const 对象调用 gcd：

```
const int i = 3, j = 6;
int k = rgcd(3, 6);    // ok: k initialized to 3
```

这种行为源于 const 对象的标准初始化规则（2.4 节）。因为初始化复制了初始化式的值，所以可用 const 对象初始化非 const 对象，反之亦然。

231

如果将形参定义为非引用的 const 类型：

```
void fcn(const int i) { /* fcn can read but not write to i */ }
```

则在函数中，不可以改变实参的局部副本。由于实参仍然是以副本的形式传递，因此传递给 fcn 的既可以是 const 对象也可以是非 const 对象。

令人吃惊的是, 尽管函数的形参是 `const`, 但是编译器却将 `fcn` 的定义视为其形参被声明为普通的 `int` 型:

```
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* ... */ } // error: redefines fcn(int)
```

这种用法是为了支持对 C 语言的兼容, 因为在 C 语言中, 具有 `const` 形参或非 `const` 形参的函数并无区别。

3. 复制实参的局限性

复制实参并不是在所有的情况下都适合, 不适宜复制实参的情况包括:

- 当需要在函数中修改实参的值时。
- 当需要以大型对象作为实参传递时。对实际的应用而言, 复制对象所付出的时间和存储空间代价往往过大。
- 当没有办法实现对象的复制时。

对于上述几种情况, 有效的解决办法是将形参定义为引用或指针类型。

习题

习题 7.5 编写一个函数, 该函数具有两个形参, 分别为 `int` 型和指向 `int` 型的指针, 并返回这两个 `int` 值之中较大的数值。考虑应将其指针形参定义为什么类型?

习题 7.6 编写函数交换两个 `int` 型指针所指向的值, 调用并检验该函数, 输出交换后的值。

7.2.2 引用形参

考虑下面不适宜复制实参的例子, 该函数希望交换两个实参的值:

```
// incorrect version of swap: The arguments are not changed!
void swap(int v1, int v2)
{
    int tmp = v2;
    v2 = v1;      // assigns new value to local copy of the argument
    v1 = tmp;
}                // local objects v1 and v2 no longer exist
```

这个例子期望改变实参本身的值。但对于上述的函数定义, `swap` 无法影响实参本身。执行 `swap` 时, 只交换了其实参的局部副本 (local copy), 而传递给 `swap` 的实参并没有修改:

```
int main()
{
    int i = 10;
    int j = 20;
    cout << "Before swap():\ti: "
          << i << "\tj: " << j << endl;
    swap(i, j);
    cout << "After swap():\ti: "
          << i << "\tj: " << j << endl;
    return 0;
}
```

编译并执行程序，产生如下输出结果：

```
Before swap(): i: 10 j: 20
After swap(): i: 10 j: 20
```

为了使 swap 函数以期望的方式工作，交换实参的值，需要将形参定义为引用类型：

```
// ok: swap acts on references to its arguments
void swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

与所有引用一样，引用形参直接关联到其所绑定的对象，而并非这些对象的副本。定义引用时，必须用与该引用绑定的对象初始化该引用。引用形参完全以相同的方式工作。每次调用函数，引用形参被创建并与相应实参关联。此时，当调用 swap

```
swap(i, j);
```

形参 v1 只是对象 i 的另一个名字，而 v2 则是对象 j 的另一个名字。对 v1 的任何修改实际上也是对 i 的修改。同样地，v2 上的任何修改实际上也是对 j 的修改。重新编译使用 swap 的这个修订版本的 main 函数后，可以看到输出结果是正确的：

```
Before swap(): i: 10 j: 20
After swap(): i: 20 j: 10
```



从 C 语言背景转到 C++ 的程序员习惯通过传递指针来实现对实参的访问。在 C++ 中，使用引用形参则更安全和更自然。

233

1. 使用引用形参返回额外的信息

通过对 swap 这个例子的讨论，了解了如何利用引用形参让函数修改实参的值。引用形参的另一种用法是向主调函数返回额外的结果。

函数只能返回单个值，但有些时候，函数有不只一个的内容需要返回。例如，定义一个 find_val 函数，在一个整型 vector 对象的元素中搜索某个特定值。如果找到满足要求的元素，则返回指向该元素的迭代器；否则返回一个迭代器，指向该 vector 对象的 end 操作返回的元素。此外，如果该值出现了不止一次，我们还希望函数可以返回其出现的次数。在这种情况下，返回的迭代器应该指向具有要寻找的值的第一个元素。

如何定义既返回一个迭代器又返回出现次数的函数？我们可以定义一种包含一个迭代器和一个计数器的新类型。而更简便的解决方案是给 find_val 传递一个额外的引用实参，用于返回出现次数的统计结果：

```
// returns an iterator that refers to the first occurrence of value
// the reference parameter occurs contains a second return value
vector<int>::const_iterator find_val(
    vector<int>::const_iterator beg, // first element
    vector<int>::const_iterator end, // one past last element
    int value,                       // the value we want
    int& occurs)
```

```

        vector<int>::size_type &occurs)    // number of times it occurs
    {
        // res_iter will hold first occurrence, if any
        vector<int>::const_iterator res_iter = end;
        occurs = 0; // set occurrence count parameter
        for ( ; beg != end; ++beg)
            if (*beg == value) {
                // remember first occurrence of value
                if (res_iter == end)
                    res_iter = beg;
                ++occurs; // increment occurrence count
            }
        return res_iter; // count returned implicitly in occurs
    }
}

```

调用 `find_val` 时，需传递四个实参：一对标志 `vector` 对象中要搜索的元素范围（9.2.1 节）的迭代器，所查找的值，以及用于存储出现次数的 `size_type` 类型（3.2.3 节）对象。假设 `ivec` 是 `vector<int>` 类型的对象，`it` 是一个适当类型的迭代器，而 `ctr` 则是 `size_type` 类型的变量，则可如此调用该函数：

```
it = find_val(ivec.begin(), ivec.end(), 42, ctr);
```

调用后，`ctr` 的值将是 42 出现的次数，如果 42 在 `ivec` 中出现了，则 `it` 将指向其第一次出现的位置；否则，`it` 的值为 `ivec.end()`，而 `ctr` 则为 0。

234

2. 利用 **const** 引用避免复制

在向函数传递大型对象时，需要使用引用形参，这是引用形参适用的另一种情况。虽然复制实参对于内置数据类型的对象或者规模较小的类类型对象来说没有什么问题，但是对于大部分的类类型或者大型数组，它的效率（通常）太低了；此外，我们将在第 13 章学习到，某些类类型是无法复制的。使用引用形参，函数可以直接访问实参对象，而无须复制它。

编写一个比较两个 `string` 对象长度的函数作为例子。这个函数需要访问每个 `string` 对象的 `size`，但不必修改这些对象。由于 `string` 对象可能相当长，所以我们希望避免复制操作。使用 `const` 引用就可避免复制：

```

// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

```

其每一个形参都是 `const string` 类型的引用。因为形参是引用，所以不复制实参。又因为形参是 `const` 引用，所以 `isShorter` 函数不能使用该引用来修改实参。



如果使用引用形参的唯一目的是避免复制实参，则应将形参定义为 `const` 引用。

3. 更灵活的指向 **const** 的引用

如果函数具有普通的非 `const` 引用形参，则显然不能通过 `const` 对象进行调用。毕竟，此时函数可以修改传递进来的对象，这样就违背了实参的 `const` 特性。

但比较容易忽略的是，调用这样的函数时，传递一个右值（2.3.1 节）或具有需要转换的类型的对象同样是不允许的：

```
// function takes a non-const reference parameter
int incr(int &val)
{
    return ++val;
}
int main()
{
    short v1 = 0;
    const int v2 = 42;
    int v3 = incr(v1); // error: v1 is not an int
    v3 = incr(v2);     // error: v2 is const
    v3 = incr(0);      // error: literals are not lvalues
    v3 = incr(v1 + v2); // error: addition doesn't yield an lvalue
    int v4 = incr(v3); // ok: v3 is a non const object type int
}
```

235

问题的关键是非 const 引用形参（2.5 节）只能与完全同类型的非 const 对象关联。

应该将不修改相应实参的形参定义为 const 引用。如果将这样的形参定义为非 const 引用，则毫无必要地限制了该函数的使用。例如，可编写下面的程序在一个 string 对象中查找一个指定的字符：

```
// returns index of first occurrence of c in s or s.size() if c isn't in s
// Note: s doesn't change, so it should be a reference to const
string::size_type find_char(string &s, char c)
{
    string::size_type i = 0;
    while (i != s.size() && s[i] != c)
        ++i; // not found, look at next character
    return i;
}
```

这个函数将其 string 类型的实参当作普通（非 const）的引用，尽管函数并没有修改这个形参的值。这样的定义带来的问题是不能通过字符串字面值来调用这个函数：

```
if (find_char("Hello World", 'o')) // ...
```

虽然字符串字面值可以转换为 string 对象，但上述调用仍然会导致编译失败。

继续将这个问题延伸下去会发现，即使程序本身没有 const 对象，而且只使用 string 对象（而并非字符串字面值或产生 string 对象的表达式）调用 find_char 函数，编译阶段的问题依然会出现。例如，可能有另一个函数 is_sentence 调用 find_char 来判断一个 string 对象是否是句子：

```
bool is_sentence (const string &s)
{
    // if there's a period and it's the last character in s
    // then s is a sentence
    return (find_char(s, '.') == s.size() - 1);
}
```

如上代码，函数 is_sentence 中 find_char 的调用是一个编译错误。传递进 is_sentence 的形

参是指向 `const string` 对象的引用，不能将这种类型的参数传递给 `find_char`，因为后者期待得到一个指向非 `const string` 对象的引用。



最佳
实践

应该将不需要修改的引用形参定义为 `const` 引用。普通的非 `const` 引用形参在使用时不太灵活。这样的形参既不能用 `const` 对象初始化，也不能用字面值或产生右值的表达式实参初始化。

4. 传递指向指针的引用

假设我们想编写一个与前面交换两个整数的 `swap` 类似的函数，实现两个指针的交换。已知需用 `*` 定义指针，用 `&` 定义引用。现在，问题在于如何将这两个操作符结合起来以获得指向指针的引用。这里给出一个例子：

236

```
// swap values of two pointers to int
void ptrswap(int *&v1, int *&v2)
{
    int *tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

形参

```
int *&v1
```

的定义应从右至左理解：`v1` 是一个引用，与指向 `int` 型对象的指针相关联。也就是说，`v1` 只是传递进 `ptrswap` 函数的任意指针的别名。

重写 7.2.2 节的 `main` 函数，调用 `ptrswap` 交换分别指向值 10 和 20 的指针：

```
int main()
{
    int i = 10;
    int j = 20;
    int *pi = &i;    // pi points to i
    int *pj = &j;    // pj points to j
    cout << "Before ptrswap():\t*pi: "
         << *pi << "\t*pj: " << *pj << endl;
    ptrswap(pi, pj); // now pi points to j; pj points to i
    cout << "After ptrswap():\t*pi: "
         << *pi << "\t*pj: " << *pj << endl;
    return 0;
}
```

编译并执行后，该程序产生如下结果：

```
Before ptrswap(): *pi: 10 *pj: 20
After ptrswap():  *pi: 20 *pj: 10
```

即指针的值被交换了。在调用 `ptrswap` 时，`pi` 指向 `i`，而 `pj` 则指向 `j`。在 `ptrswap` 函数中，指针被交换，使得调用 `ptrswap` 结束后，`pi` 指向了原来 `pj` 所指向的对象。换句话说，现在 `pi` 指向 `j`，而 `pj` 则指向了 `i`。

习题

习题 7.7 解释下面两个形参声明的不同之处：

```
void f(T);
void f(T&);
```

习题 7.8 举一个例子说明什么时候应该将形参定义为引用类型。再举一个例子说明什么时候不应该将形参定义为引用。

习题 7.9 将 (7.2.2 节定义的) find_val 函数的形参表中 occurs 的声明修改为非引用参数类型，并重新执行这个程序，该函数的行为发生了什么改变？

习题 7.10 下面的程序虽然是合法的，但可用性还不够好，指出并改正该程序的局限：

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

习题 7.11 何时应将引用形参定义为 const 对象？如果在需要 const 引用时，将形参定义为普通引用，则会出现什么问题？

7.2.3 vector 和其他容器类型的形参



最佳
实践

通常，函数不应该有 vector 或其他标准库容器类型的形参。调用含有普通的非引用 vector 形参的函数将会复制 vector 的每一个元素。

237

从避免复制 vector 的角度出发，应考虑将形参声明为引用类型。然而，看过第 11 章后我们会知道，事实上，C++ 程序员倾向于通过传递指向容器中需要处理的元素的迭代器来传递容器：

```
// pass iterators to the first and one past the last element to print
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end)
{
    while (beg != end) {
        cout << *beg++;
        if (beg != end) cout << " "; // no space after last element
    }
    cout << endl;
}
```

这个函数将输出从 beg 指向的元素开始到 end 指向的元素（不含）为止的范围内所有的元素。除了最后一个元素外，每个元素后面都输出一个空格。

7.2.4 数组形参

数组有两个特殊的性质，影响我们定义和使用作用在数组上的函数：一是不能复制数组 (4.1.1 节)；二是使用数组名字时，数组名会自动转化为指向其第一个元素的指针 (4.2.4 节)。因为数组不能复制，所以无法编写使用数组类型形参的函数。因为数组会被自动转化为指针，所以处理数组的函数通常通过操纵指向数组中的元素的指针来处理数组。

238

1. 数组形参的定义

如果要编写一个函数，输出 int 型数组的内容，可用下面三种方式指定数组形参：

```
// three equivalent definitions of printValues
void printValues(int*) { /* ... */ }
void printValues(int[]) { /* ... */ }
void printValues(int[10]) { /* ... */ }
```

虽然不能直接传递数组，但是函数的形参可以写成数组的形式。虽然形参表示方式不同，但可将使用数组语法定义的形参看作指向数组元素类型的指针。上面的三种定义是等价的，形参类型都是 `int*`。



通常，将数组形参直接定义为指针要比使用数组语法定义更好。这样就明确地表示，函数操纵的是指向数组元素的指针，而不是数组本身。由于忽略了数组长度，形参定义中如果包含了数组长度则特别容易引起误解。

2. 形参的长度会引起误解

编译器忽略为任何数组形参指定的长度。根据数组的长度（权且这样说），可将函数 `printValues` 编写为：

```
// parameter treated as const int*, size of array is ignored
void printValues(const int ia[10])
{
    // this code assumes array has 10 elements;
    // disaster if argument has fewer than 10 elements!
    for (size_t i = 0; i != 10; ++i)
    {
        cout << ia[i] << endl;
    }
}
```

尽管上述代码假定所传递的数组至少含有 10 个元素，但 C++ 语言没有任何机制强制实现这个假设。下面的调用都是合法的：

```
int main()
{
    int i = 0, j[2] = {0, 1};
    printValues(&i); // ok: &i is int*; probable run-time error
    printValues(j); // ok: j is converted to pointer to 0th
                    // element; argument has type int*;
                    // probable run-time error

    return 0;
}
```

虽然编译没有问题，但是这两个调用都是错误的，可能导致运行失败。在这两个调用中，由于函数 `printValues` 假设传递进来的数组至少含有 10 个元素，因此造成数组内存的越界访问。程序的执行可能产生错误的输出，也可能崩溃，这取决于越界访问的内存中恰好存储的数值是什么。



当编译器检查数组形参关联的实参时，它只会检查实参是不是指针、指针的类型和数组元素的类型是否匹配，而不会检查数组的长度。

3. 数组实参

和其他类型一样，数组形参可定义为引用或非引用类型。大部分情况下，数组以普通的非引

用类型传递，此时数组会悄悄地转换为指针。一般来说，非引用类型的形参会初始化为其相应实参的副本。而在传递数组时，实参是指向数组第一个元素的指针，形参复制的是这个指针的值，而不是数组元素本身。函数操纵的是指针的副本，因此不会修改实参指针的值。然而，函数可通过该指针改变它所指向的数组元素的值。通过指针形参做的任何改变都在修改数组元素本身。



不需要修改数组形参的元素时，函数应该将形参定义为指向 `const` 对象的指针：

```
// f won't change the elements in the array
void f(const int*) { /* ... */ }
```

4. 通过引用传递数组

和其他类型一样，数组形参可声明为数组的引用。如果形参是数组的引用，编译器不会将数组实参转化为指针，而是传递数组的引用本身。在这种情况下，数组大小成为形参和实参类型的一部分。编译器检查数组实参的大小与形参的大小是否匹配：

```
// ok: parameter is a reference to an array; size of array is fixed
void printValues(int (&arr)[10]) { /* ... */ }
int main()
{
    int i = 0, j[2] = {0, 1};
    int k[10] = {0,1,2,3,4,5,6,7,8,9};
    printValues(&i);    // error: argument is not an array of 10 ints
    printValues(j);    // error: argument is not an array of 10 ints
    printValues(k);    // ok: argument is an array of 10 ints
    return 0;
}
```

240 这个版本的 `printValues` 函数只严格地接受含有 10 个 `int` 型数值的数组，这限制了哪些数组可以传递。然而，由于形参是引用，在函数体中依赖数组的大小是安全的：

```
// ok: parameter is a reference to an array; size of array is fixed
void printValues(int (&arr)[10])
{
    for (size_t i = 0; i != 10; ++i) {
        cout << arr[i] << endl;
    }
}
```



`&arr` 两边的圆括号是必需的，因为下标操作符具有更高的优先级：

```
f(int &arr[10])    // error: arr is an array of references
f(int (&arr)[10]) // ok: arr is a reference to an array of 10 ints
```

在 16.1.5 节将会介绍如何重新编写此函数，允许传递指向任意大小的数组的引用形参。

5. 多维数组的传递

回顾前面我们说过在 C++ 中没有多维数组（4.4 节）。所谓多维数组实际是指数组的数组。

和其他数组一样，多维数组以指向 0 号元素的指针方式传递。多维数组的元素本身就是数组。除了第一维以外的所有维的长度都是元素类型的一部分，必须明确指定：

```
// first parameter is an array whose elements are arrays of 10 ints
void printValues(int (matrix*)[10], int rowSize);
```

上面的语句将 `matrix` 声明为指向含有 10 个 `int` 型元素的数组的指针。



再次强调, `*matrix` 两边的圆括号是必需的:

```
int *matrix[10];           // array of 10 pointers
int (*matrix)[10];        // pointer to an array of 10 ints
```

我们也可以用数组语法定义多维数组。与一维数组一样, 编译器忽略第一维的长度, 所以最好不要把它包括在形参表内:

```
// first parameter is an array whose elements are arrays of 10 ints
void printValues(int matrix[][10], int rowSize);
```

这条语句把 `matrix` 声明为二维数组的形式。实际上, 形参是一个指针, 指向数组的数组中的元素。数组中的每个元素本身就是含有 10 个 `int` 型对象的数组。

7.2.5 传递给函数的数组的处理

就如刚才所见的, 非引用数组形参的类型检查只是确保实参是和数组元素具有同样类型的指针, 而不会检查实参实际上是否指向指定大小的数组。

241



任何处理数组的程序都要确保程序停留在数组的边界内。

有三种常见的编程技巧确保函数的操作不超出数组实参的边界。第一种方法是在数组本身放置一个标记来检测数组的结束。C 风格字符串就是采用这种方法的一个例子, 它是一种字符数组, 并且以空字符 `null` 作为结束的标记。处理 C 风格字符串的程序就是使用这个标记停止数组元素的处理。

1. 使用标准库规范

第二种方法是传递指向数组第一个和最后一个元素的下一个位置的指针。这种编程风格由标准库所使用的技术启发而得, 在第二部分将会进一步介绍这种编程风格。

使用这种方法重写函数 `printValues` 并调用该函数, 如下所示:

```
void printValues(const int *beg, const int *end)
{
    while (beg != end) {
        cout << *beg++ << endl;
    }
}

int main()
{
    int j[2] = {0, 1};
    // ok: j is converted to pointer to 0th element in j
    //      j + 2 refers one past the end of j
    printValues(j, j + 2);
    return 0;
}
```

`printValues` 中的循环很像用 `vector` 迭代器编写的程序。每次循环都使 `beg` 指针指向下一个元

素,从而实现数组的遍历。当 beg 指针等于结束标记时,循环结束。结束标记就是传递给函数的第二个形参。

调用这个版本的函数需要传递两个指针:一个指向要输出的第一个元素,另一个则指向最后一个元素的下一个位置。只要正确计算指针,使它们标记一段有效的元素范围,程序就会安全。

2. 显式传递表示数组大小的形参

第三种方法是将第二个形参定义为表示数组的大小,这种用法在 C 程序和标准化之前的 C++ 程序中十分普遍。

用这种方法再次重写函数 printValues,新版本及其调用如下所示:

```
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void printValues(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
int main()
{
    int j[] = { 0, 1 }; // int array of size 2
    printValues(j, sizeof(j)/sizeof(*j));
    return 0;
}
```

这个版本使用了形参 size 来确定要输出的元素的个数。调用 printValues 时,要额外传递一个形参。只要传递给函数的 size 值不超过数组的实际大小,程序就能安全运行。

习题

习题 7.12 什么时候应使用指针形参? 什么时候应使用引用形参? 解释两者的优点和缺点。

习题 7.13 编写程序计算数组元素之和。要求编写函数三次,每次以不同的方法处理数组边界。

习题 7.14 编写程序求 vector<double>对象中所有元素之和。

7.2.6 main: 处理命令行选项

主函数 main 是演示 C 程序如何将数组传递给函数的好例子。直到现在,我们所定义的主函数都只有空的形参表:

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

但是,我们通常需要给 main 传递实参。传统上,主函数的实参是可选的,用来确定程序要执行的操作。比如,假设我们的主函数 main 位于名为 prog 的可执行文件中,可如下将实参选项传递给程序:

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

这种用法的处理方法实际上是在主函数 main 中定义了两个形参:

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

第二个形参 `argv` 是一个 C 风格字符串数组。第一个形参 `argc` 则用于传递该数组中字符串的个数。由于第二个参数是一个数组，主函数 `main` 也可以这样定义：

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

表示 `argv` 是指向 `char*` 的指针。

当将实参传递给主函数 `main` 时，`argv` 中的第一个字符串（如果有的话）通常是程序的名字。接下来的元素将额外的可选字符串传递给主函数 `main`。以前面的命令行为例，`argc` 应设为 5，`argv` 会保存下面几个 C 风格字符串：

```
argv[0] = "prog";
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
```

习题

习题 7.15 编写一个主函数 `main`，使用两个值作为实参，并输出它们的和。

习题 7.16 编写程序使之可以接受本节介绍的命令行选项，并输出传递给 `main` 的实参的值。

7.2.7 含有可变形参的函数



C++ 中的省略形参是为了编译使用了 `varargs` 的 C 语言程序。关于如何使用 `varargs`，请查阅所用 C 语言编译器的文档。对于 C++ 程序，只能将简单数据类型传递给含有省略形参的函数。实际上，当需要传递给省略形参时，大多数类类型对象都不能正确地复制。

在无法列举出传递给函数的所有实参的类型和数目时，可以使用省略形参。省略符暂停了类型检查机制。它们的出现告知编译器，当调用函数时，可以有 0 或多个实参，而实参的类型未知。省略形参有下列两种形式：

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

第一种形式为特定数目的形参提供了声明。在这种情况下，当函数被调用时，对于与显示声明的形参相对应的实参进行类型检查，而对于与省略符对应的实参则暂停类型检查。在第一种形式中，形参声明后面的逗号是可选的。

大部分带有省略形参的函数都利用显式声明的参数中的一些信息，来获取函数调用中提供的其他可选实参的类型和数目。因此带有省略符的第一种形式的函数声明是最常用的。

7.3 return 语句

`return` 语句用于结束当前正在执行的函数，并将控制权返回给调用此函数的函数。`return` 语句有两种形式：

```
return;
return expression;
```

7.3.1 没有返回值的函数

不带返回值的 `return` 语句只能用于返回类型为 `void` 的函数。在返回类型为 `void` 的函数中，`return` 返回语句不是必需的，隐式的 `return` 发生在函数的最后一个语句完成时。

一般情况下，返回类型是 `void` 的函数使用 `return` 语句是为了引起函数的强制结束，这种 `return` 的用法类似于循环结构中的 `break` 语句（6.10 节）的作用。例如，可如下重写 `swap` 程序，使之在输入的两个数值相同时不执行任何工作：

```
// ok: swap acts on references to its arguments
void swap(int &v1, int &v2)
{
    // if values already the same, no need to swap, just return
    if (v1 == v2)
        return;
    // ok, have work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // no explicit return necessary
}
```

这个函数首先检查两个值是否相等，如果相等则退出函数；如果不相等，则交换这两个值，隐式的 `return` 发生在最后一个赋值语句后。

返回类型为 `void` 的函数通常不能使用第二种形式的 `return` 语句，但是，它可以返回另一个返回类型同样是 `void` 的函数的调用结果：

245

```
void do_swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // ok: void function doesn't need an explicit return
}
void swap(int &v1, int &v2)
{
    if (v1 == v2)
        return false; // error: void function cannot return a value
    return do_swap(v1, v2); // ok: returns call to a void function
}
```

返回任何其他表达式的尝试都会导致编译时的错误。

7.3.2 具有返回值的函数

`return` 语句的第二种形式提供了函数的结果。任何返回类型不是 `void` 的函数都必须返回一个值，而且这个返回值的类型必须和函数的返回类型相同，或者能隐式转化为函数的返回类型。

尽管 C++ 不能确保结果的正确性，但能保证函数每一次 `return` 都返回适当类型的结果。例

如，下面的程序就不能通过编译：

```
// Determine whether two strings are equal.
// If they differ in size, determine whether the smaller
// one holds the same characters as the larger one
bool str_subrange(const string &str1, const string &str2)
{
    // same sizes: return normal equality test
    if (str1.size() == str2.size())
        return str1 == str2;    // ok, == returns bool
    // find size of smaller string
    string::size_type size = (str1.size() < str2.size())
        ? str1.size() : str2.size();
    string::size_type i = 0;
    // look at each element up to size of smaller string
    while (i != size) {
        if (str1[i] != str2[i])
            return;    // error: no return value
    }
    // error: control might flow off the end of the function without a return
    // the compiler is unlikely to detect this error
}
```

246

while 循环中的 return 语句是错误的，因为它没有返回任何值，编译器将检查出这个错误。

第二个错误源于函数没有在 while 循环后提供 return 语句。调用这个函数时，如果一个 string 是另一个 string 的子集，执行会退出 while 循环。这里应该有一个 return 语句来处理这种情况。编译器有可能检查出也有可能检查不出这种错误。执行程序时，不确定在运行阶段会出现什么问题。



在含有 return 语句的循环后没有提供 return 语句是很危险的，因为大部分的编译器不能检测出这个漏洞，运行时会出现什么问题是不确定的。

1. 主函数 main 的返回值

返回类型不是 void 的函数必须返回一个值，但此规则有一个例外情况：允许主函数 main 没有返回值就可结束。如果程序控制执行到主函数 main 的最后一个语句都还没有返回，那么编译器会隐式地插入返回 0 的语句。

关于主函数 main 返回的另一个特别之处在于如何处理它的返回值。在 1.1 节已知，可将主函数 main 返回的值视为状态指示器。返回 0 表示程序运行成功，其他大部分返回值则表示失败。非 0 返回值的意义因机器不同而不同，为了使返回值独立于机器，cstdlib 头文件定义了两个预处理变量（2.9.2 节），分别用于表示程序运行成功和失败：

```
#include <cstdlib>
int main()
{
    if (some_failure)
        return EXIT_FAILURE;
    else
        return EXIT_SUCCESS;
}
```

我们的代码不再需要使用那些依赖于机器的精确返回值。相应地，这些值都在 cstdlib 库中定义，我们的代码不需要做任何修改。

2. 返回非引用类型

函数的返回值用于初始化在调用函数处创建的临时对象 (temporary object)。在求解表达式时, 如果需要一个地方储存其运算结果, 编译器会创建一个没有命名的对象, 这就是临时对象。在英语中, C++程序员通常用 temporary 这个术语来代替 temporary object。

用函数返回值初始化临时对象与用实参初始化形参的方法是一样的。如果返回类型不是引用, 在调用函数的地方会将函数返回值复制给临时对象。当函数返回非引用类型时, 其返回值既可以是局部对象, 也可以是求解表达式的结果。

247

例如, 下面的程序提供了一个计数器、一个单词 word 和单词结束字符串 ending, 当计数器的值大于 1 时, 返回该单词的复数版本:

```
// return plural version of word if ctr isn't 1
string make_plural(size_t ctr, const string &word,
                  const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}
```

我们可以使用这样的函数来输出单词的单数或复数形式。

这个函数要么返回其形参 word 的副本, 要么返回一个未命名的临时 string 对象, 这个临时对象是由字符串 word 和 ending 的相加而产生的。这两种情况下, return 都在调用该函数的地方复制了返回的 string 对象。

3. 返回引用

当函数返回引用类型时, 没有复制返回值。相反, 返回的是对象本身。例如, 考虑下面的函数, 此函数返回两个 string 类型形参中较短的那个字符串的引用:

```
// find longer of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

形参和返回类型都是指向 const string 对象的引用, 调用函数和返回结果时, 都没有复制这些 string 对象。

4. 千万不要返回局部对象的引用



理解返回引用至关重要的是: 千万不能返回局部变量的引用。

当函数执行完毕时, 将释放分配给局部对象的存储空间。此时, 对局部对象的引用就会指向不确定的内存。考虑下面的程序:

```
// Disaster: Function returns a reference to a local object
const string &manip(const string& s)
{
    string ret = s;
    // transform ret in some way
    return ret; // Wrong: Returning reference to a local object!
}
```

248

这个函数会在运行时出错，因为它返回了局部对象的引用。当函数执行完毕，字符串 `ret` 占用的储存空间被释放，函数返回值指向了对于这个程序来说不再有效的内存空间。



提示 确保返回引用安全的一个好方法是：请自问，这个引用指向哪个在此之前存在的对象？

5. 引用返回左值

返回引用的函数返回一个左值。因此，这样的函数可用于任何要求使用左值的地方：

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix];
}
int main()
{
    string s("a value");
    cout << s << endl;    // prints a value
    get_val(s, 0) = 'A';  // changes s[0] to A

    cout << s << endl;    // prints A value
    return 0;
}
```

给函数返回值赋值可能让人惊讶，由于函数返回的是一个引用，因此这是正确的，该引用是被返回元素的同义词。

如果不希望引用返回值被修改，返回值应该声明为 `const`：

```
const char &get_val(...
```

6. 千万不要返回指向局部对象的指针

函数的返回类型可以是大多数类型。特别地，函数也可以返回指针类型。和返回局部对象的引用一样，返回指向局部对象的指针也是错误的。一旦函数结束，局部对象被释放，返回的指针就变成了指向不再存在的对象的悬垂指针（5.11 节）。

习题

习题 7.17 什么时候返回引用是正确的？而什么时候返回 `const` 引用是正确的？

习题 7.18 下面函数存在什么潜在的运行时问题？

```
string &processText() {
    string text;
    while (cin >> text) { /* ... */ }
    // ....
    return text;
}
```

习题 7.19 判断下面程序是否合法；如果合法，解释其功能；如果不合法，更正它并解释原因。

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = 0;
}
```

7.3.3 递归

直接或间接调用自己的函数称为递归函数 (recursion function)。一个简单的递归函数例子是阶乘的计算。数 n 的阶乘是从 1 到 n 的乘积。例如，5 的阶乘就是 120。

249 $1 * 2 * 3 * 4 * 5 = 120$

解决这个问题的自然方法就是递归：

```
// calculate val!, which is 1*2 *3 ... * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

递归函数必须定义一个终止条件；否则，函数就会“永远”递归下去，这意味着函数会一直调用自身直到程序栈耗尽。有时候，这种现象称为“无限递归错误” (infinite recursion error)。对于函数 `facorial`，`val` 为 1 是终止条件。

另一个例子是求最大公约数的递归函数：

```
// recursive version greatest common divisor program
int rgcd(int v1, int v2)
{
    if (v2 != 0) // we're done once v2 gets to zero
        return rgcd(v2, v1%v2); // recurse, reducing v2 on each call
    return v1;
}
```

这个例子中，终止条件是余数为 0。如果用实参 (15, 123) 来调用 `rgcd` 函数，结果为 3。表 7-1 跟踪了它的执行过程。

250

表 7-1 `rgcd(15,123)` 的跟踪过程

v1	v2	Return
15	123	<code>rgcd(123,15)</code>
123	15	<code>rgcd(15,3)</code>
15	3	<code>rgcd(3,0)</code>
3	0	3

最后一次调用：

```
rgcd(3,0)
```

满足了终止条件，它返回最大公约数 3。该值依次成为前面每个调用的返回值。这个过程称为此值向上回渗 (percolate)，直到执行返回到第一次调用 `rgcd` 的函数。



主函数 `main` 不能调用自身。

习题

习题 7.20 将函数 `factorial` 重写为迭代函数（即非递归函数）。

习题 7.21 如果函数 `factorial` 的终止条件为：

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

会出现什么问题？

7.4 函数声明

正如变量必须先声明后使用一样，函数也必须在被调用之前先声明。与变量的定义（2.3.5 节）类似，函数的声明也可以和函数的定义分离；一个函数只能定义一次，但是可声明多次。

函数声明由函数返回类型、函数名和形参列表组成。形参列表必须包括形参类型，但是不必对形参命名。这三个元素被称为**函数原型**（`function prototype`），函数原型描述了函数的接口。



函数原型为定义函数的程序员和使用函数的程序员之间提供了接口。在使用函数时，程序员只对函数原型编程即可。

251

函数声明中的形参名会被忽略，如果在声明中给出了形参的名字，它应该用作辅助文档：

```
void print(int *array, int size);
```

在头文件中提供函数声明

回顾前面章节，变量可在头文件中声明（2.9 节），而在源文件中定义。同理，函数也应当在头文件中声明，并在源文件中定义。

把函数声明直接放到每个使用该函数的源文件中，这可能是大家希望的方式，而且也是合法的。但问题在于这种用法比较呆板而且容易出错。解决的方法是把函数声明放在头文件中，这样可以确保对于指定函数其所有声明保持一致。如果函数接口发生变化，则只要修改其唯一的声明即可。



定义函数的源文件应包含声明该函数的头文件。

将提供函数声明的头文件包含在定义该函数的源文件中，可使编译器能检查该函数的定义和声明是否一致。特别地，如果函数定义和函数声明的形参列表一致，但返回类型不一致，编译器会发出警告或出错信息来指出这种差异。

习题

习题 7.22 编写下面函数的原型：

(a) 函数名为 `compare`，有两个形参，都是名为 `matrix` 的类的引用，返回 `bool` 类型的值。

(b) 函数名为 `change_val`，返回 `vector<int>` 类型的迭代器，有两个形参：一个是 `int` 类型形参，另一个是 `vector<int>` 类型的迭代器。

提示：写函数原型时，函数名应当暗示函数的功能。考虑这个提示会如何影响你用的类型？

习题 7.23 给出下面函数声明，判断哪些调用是合法的，哪些是不合法的。对于那些不合法的调用，解释原因。

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
```

- (a) `calc(23.4, 55.1);`
- (b) `count("abcda", 'a');`
- (c) `calc(66);`
- (d) `sum(vec.begin(), vec.end(), 3.8);`

252

默认实参

默认实参是一种虽然并不普遍、但在多数情况下仍然适用的实参值。调用函数时，可以省略有默认值的实参。编译器会为我们省略的实参提供默认值。

默认实参是通过给形参表中的形参提供明确的初始值来指定的。程序员可为一个或多个形参定义默认值。但是，如果有一个形参具有默认实参，那么，它后面所有的形参都必须有默认实参。

例如，下面的函数创建并初始化了一个 `string` 对象，用于模拟窗口屏幕。此函数为窗口屏幕的高、宽和背景字符提供了默认实参：

```
string screenInit(string::size_type height = 24,
                  string::size_type width = 80,
                  char background = ' ');
```

调用包含默认实参的函数时，可以为该形参提供实参，也可以不提供。如果提供了实参，则它将覆盖默认的实参值；否则，函数将使用默认实参值。下面的函数 `screenInit` 的调用都是正确的：

```
string screen;
screen = screenInit();           // equivalent to screenInit (24,80,' ')
screen = screenInit(66);        // equivalent to screenInit (66,80,' ')
screen = screenInit(66, 256);    // screenInit(66,256,' ')
screen = screenInit(66, 256, '#');
```

函数调用的实参按位置解析，默认实参只能用来替换函数调用缺少的尾部实参。例如，如果要给 `background` 提供实参，那么也必须给 `height` 和 `width` 提供实参：

```
screen = screenInit(, , '?'); // error, can omit only trailing arguments
screen = screenInit( '?');    // calls screenInit( '?',80,' ')
```

注意第二个调用，只传递了一个字符值，虽然这是合法的，但是却并不是程序员的原意。因为 `'?'` 是一个 `char`，`char` 可提升为最左边形参的类型，所以这个调用是合法的。最左边的形参具有 `string::size_type` 类型，这是 `unsigned` 整型。在这个调用中，`char` 实参隐式地提升为 `string::size_type` 类型，并作为实参传递给形参 `height`。



因为 `char` 是整型（2.1.1 节），因此把一个 `char` 值传递给 `int` 型形参是合法的，反之亦然。这个事实会导致很多误解。例如，如果函数同时含有 `char` 型和 `int` 型形参，则调用者很容易以错误的顺序传递实参。如果使用默认实参，则这个问题会变得更加复杂。

设计带有默认实参的函数，其中部分工作就是排列形参，使最少使用默认实参的形参排在最前，最可能使用默认实参的形参排在最后。

1. 默认实参的初始化式

默认实参可以是任何适当类型的表达式：

```
string::size_type screenHeight();
string::size_type screenWidth(string::size_type);
char screenDefault(char = ' ');
string screenInit(
    string::size_type height = screenHeight(),
    string::size_type width = screenWidth(screenHeight()),
    char background = screenDefault());
```

如果默认实参是一个表达式，而且默认值用作实参，则在调用函数时求解该表达式。例如，每次不带第三个实参调用函数 `screenInit` 时，编译器都会调用函数 `screenDefault` 为 `background` 获得一个值。

2. 指定默认实参的约束

既可以在函数声明也可以在函数定义中指定默认实参。但是，在一个文件中，只能为一个形参指定默认实参一次。下面的例子是错误的：

```
// ff.h
int ff(int = 0);

// ff.cc
#include "ff.h"
int ff(int i = 0) { /* ... */ } // error
```



通常，应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。

如果在函数定义的形参表中提供默认实参，那么只有在包含该函数定义的源文件中调用该函数时，默认实参才是有效的。

习题

习题 7.24 如果有的话，指出下面哪些函数声明是错误的？为什么？

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

习题 7.25 假设有如下函数声明和调用，指出哪些调用是不合法的？为什么？哪些是合法的但可能不符合程序员的原意？为什么？

```
// declarations
char *init(int ht, int wd = 80, char bckgrnd = ' ');
```

- (a) `init();`
- (b) `init(24, 10);`
- (c) `init(14, '*');`

习题 7.26 用字符 `'s'` 作为默认实参重写函数 `make_plural`。利用这个版本的函数输出单词 “success” 和 “failure” 的单数和复数形式。

7.5 局部对象

在 C++ 语言中，每个名字都有作用域，而每个对象都有生命期 (lifetime)。要弄清楚函数是怎么运行的，理解这两个概念十分重要。名字的作用域指的是知道该名字的程序文本区。对象的生命期则是在程序执行过程中对象存在的时间。

在函数中定义的形参和变量的名字只位于函数的作用域中：这些名字只在函数体中可见。通常，变量名从声明或定义的地方开始到包围它的作用域结束处都是可用的。

7.5.1 自动对象

默认情况下，局部变量的生命期局限于所在函数的每次执行期间。只有当定义它的函数被调用时才存在的对象称为自动对象 (automatic object)。自动对象在每次调用函数时创建和撤销。

局部变量所对应的自动对象在函数控制经过变量定义语句时创建。如果在定义时提供了初始化式，那么每次创建对象时，对象都会被赋予指定的初值。对于未初始化的内置类型局部变量，其初值不确定。当函数调用结束时，自动对象就会被撤销。

形参也是自动对象。形参所占用的存储空间在调用函数时创建，而在函数结束时撤销。

自动对象，包括形参，都在定义它们的块语句结束时撤销。形参在函数块中定义，因此当函数执行结束时撤销。当函数结束时，会释放它的局部存储空间。在函数结束后，自动对象和形参的值都不能再访问了。

7.5.2 静态局部对象

一个变量如果位于函数的作用域内，但生命期却跨越了这个函数的多次调用，这种变量往往很有用。则应该将这样的对象定义为 static (静态的)。

static 局部对象 (static local object) 确保不迟于在程序执行流程第一次经过该对象的定义语句时进行初始化。这种对象一旦被创建，在程序结束前都不会被撤销。当定义静态局部对象的函数结束时，静态局部对象不会撤销。在该函数被多次调用的过程中，静态局部对象会持续存在并保持它的值。考虑下面的小例子，这个函数计算了自己被调用的次数：

```
size_t count_calls()
{
    static size_t ctr = 0; // value will persist across calls
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

这个程序会依次输出 1 到 10 (包含 10) 的整数。

在第一次调用函数 count_calls 之前，ctr 就已创建并赋予初值 0。每次函数调用都使 ctr 加 1，并且返回其当前值。在执行函数 count_calls 时，变量 ctr 就已经存在并且保留上次调

用该函数时的值。因此，第二次调用时，ctr 的值为 1，第三次为 2，依此类推。

习题

习题 7.27 解释形参、局部变量和静态局部变量的差别。并给出一个有效使用了这三种变量的程序例子。

习题 7.28 编写函数，使其在第一次调用时返回 0，然后再次调用时按顺序产生正整数（即返回其当前的调用次数）。

7.6 内联函数

回顾在 7.3.2 节编写的那个返回两个 string 形参中较短的字符串的函数：

```
// find longer of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

为这样的小操作定义一个函数的好处是：

- 阅读和理解函数 shorterString 的调用，要比读一条用等价的条件表达式取代函数调用表达式并解释它的含义要容易得多。
- 如果需要做任何修改，修改函数要比找出并修改每一处等价表达式容易得多。
- 使用函数可以确保统一的行为，每个测试都保证以相同的方式实现。
- 函数可以重用，不必为其他应用重写代码。

但是，将 shorterString 写成函数有一个潜在的缺点：调用函数比求解等价表达式要慢得多。在大多数的机器上，调用函数都要做很多工作：调用前要先保存寄存器，并在返回时恢复；复制实参；程序还必须转向一个新位置执行。

1. 内联函数避免函数调用的开销

将函数指定为内联函数，（通常）就是将它程序中每个调用点上“内联地”展开。假设我们将 shorterString 定义为内联函数，则调用：

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

在编译时将展开为：

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

从而消除了把 shorterString 写成函数的额外执行开销。

在函数返回类型前加上关键字 inline 就可以将 shorterString 函数指定为内联函数：

```
// inline version: find longer of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
```

```
return s1.size() < s2.size() ? s1 : s2;
```

```
}
```



内联说明 (inline specification) 对于编译器来说只是一个建议, 编译器可以选择忽略这个建议。

一般来说, 内联机制适用于优化小的、只有几行的而且经常被调用的函数。大多数的编译器都不支持递归函数的内联。一个 1200 行的函数也不太可能在调用点内联展开。

2. 把内联函数放入头文件



内联函数应该在头文件中定义, 这一点不同于其他函数。

257

内联函数的定义对编译器而言必须是可见的, 以便编译器能够在调用点内联展开该函数的代码。此时, 仅有函数原型是不够的。

内联函数可能要在程序中定义不止一次, 只要内联函数的定义在某个源文件中只出现一次, 而且在所有源文件中, 其定义必须是完全相同的。把内联函数的定义放在头文件中, 可以确保在调用函数时所使用的定义是相同的, 并且保证在调用点该函数的定义对编译器可见。



在头文件中加入或修改内联函数时, 使用了该头文件的所有源文件都必须重新编译。

习题

习题 7.29 对于下面的声明和定义, 你会将哪个放在头文件, 哪个放在程序文本文件呢? 为什么?

(a) `inline bool eq(const BigInt&, const BigInt&) {...}`

(b) `void putValues(int *arr, int size);`

习题 7.30 把 7.2.2 节的函数 `isShorter` 改写为内联函数。

7.7 类的成员函数

2.8 节开始定义类 `Sales_item`, 用于解决第 1 章的书店问题。至此, 我们已经了解了如何定义普通函数, 现在来定义类的成员函数, 以继续完善这个类。

成员函数的定义与普通函数的定义类似。和任何函数一样, 成员函数也包含下面四个部分:

- 函数返回类型。
- 函数名。
- 用逗号隔开的形参表 (也可能是空的)。
- 包含在一对花括号里面的函数体。

正如我们知道的, 前面三部分组成函数原型。函数原型定义了所有和函数相关的类型信息: 函数返回类型是什么、函数的名字、应该给这个函数传递什么类型的实参。函数原型必须在类中定义。但是, 函数体则既可以在类中也可以在类外定义。

知道这些后,观察下面扩展的类定义,我们为这个类增加了两个新成员:成员函数 `avg_price` 和 `same_isbn`。其中 `avg_price` 函数的形参表是空的,返回 `double` 类型的值。而 `same_isbn` 函数则返回 `bool` 对象,有一个 `const Sales_item` 类型的引用形参。

258

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // private members as before
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

在解释跟在形参表后面的 `const` 之前,必须先说明成员函数是如何定义的。

7.7.1 定义成员函数的函数体

类的所有成员都必须在类定义的花括号里面声明,此后,就不能再为类增加任何成员。类的成员函数必须如声明的一般定义。类的成员函数既可以在类的定义内也可以在类的定义外定义。在类 `Sales_item` 中,这两种情况各有一例说明:函数 `same_isbn` 在类内定义,而函数 `avg_price` 则在类内声明,在类外定义。

编译器隐式地将在类内定义的成员函数当作内联函数(7.6节)。

再仔细观察函数 `same_isbn` 的定义:

```
bool same_isbn(const Sales_item &rhs) const
{ return isbn == rhs.isbn; }
```

与任何函数一样,该函数的函数体也是一个块。在这个函数中,块中只有一个语句,比较两个 `Sales_item` 对象的数据成员 `isbn` 的值,并返回比较结果。

首先要注意的是:成员 `isbn` 是 `private` 的。尽管如此,上述语句却没有任何错误。



类的成员函数可以访问该类的 `private` 成员。

更有意思的是,函数从哪个 `Sales_item` 类对象得到这个用于比较的值? 函数涉及到 `isbn` 和 `rhs.isbn`。很明显, `rhs.isbn` 使用的是传递给此函数的实参的 `isbn` 成员。没有前缀的 `isbn` 的用法更加有意思。正如我们所看见的,这个没有前缀的 `isbn` 指的是用于调用函数的对象的 `isbn` 成员。

259

1. 成员函数含有额外的、隐含的形参

调用成员函数时,实际上是使用对象来调用的。例如,调用 1.6 节书店程序中的函数 `same_isbn`,是通过名为 `total` 的对象来执行 `same_isbn` 函数的:

```
if (total.same_isbn(trans))
```

在这个调用中,传递了对象 `trans`。作为执行调用的一部分,使用对象 `trans` 初始化形参 `rhs`。

于是, `rhs.isbn` 是 `trans.isbn` 的引用。

而没有前缀的 `isbn` 使用了相同的实参绑定过程, 使之与名为 `total` 的对象绑定起来。每个成员函数都有一个额外的、隐含的形参将该成员函数与调用该函数的类对象捆绑在一起。当调用名为 `total` 的对象的 `same_isbn` 时, 这个对象也传递给了函数。而 `same_isbn` 函数使用 `isbn` 时, 就隐式地使用了调用该函数的对象的 `isbn` 成员。这个函数调用的效果是比较 `total.isbn` 和 `trans.isbn` 两个值。

2. this 指针的引入

每个成员函数 (除了在 12.6 节介绍的 `static` 成员函数外) 都有一个额外的、隐含的形参 **this**。在调用成员函数时, 形参 `this` 初始化为调用函数的对象的地址。为了理解成员函数的调用, 可考虑下面的语句:

```
total.same_isbn(trans);
```

就如编译器这样重写这个函数调用:

```
// pseudo-code illustration of how a call to a member function is translated
Sales_item::same_isbn(&total, trans);
```

在这个调用中, 函数 `same_isbn` 中的数据成员 `isbn` 属于对象 `total`。

3. const 成员函数的引入

现在, 可以理解跟在 `Sales_item` 成员函数声明的形参表后面的 `const` 所起的作用了: `const` 改变了隐含的 `this` 形参的类型。在调用 `total.same_isbn(trans)` 时, 隐含的 `this` 形参将是一个指向 `total` 对象的 `const Sales_Item*` 类型的指针。就像如下编写 `same_isbn` 的函数体一样:

```
// pseudo-code illustration of how the implicit this pointer is used
// This code is illegal: We may not explicitly define the this pointer ourselves
// Note that this is a pointer to const because same_isbn is a const member
bool Sales_item::same_isbn(const Sales_item *const this,
                           const Sales_item &rhs) const
{ return (this->isbn == rhs.isbn); }
```

260

用这种方式使用 `const` 的函数称为**常量成员函数** (`const member function`)。由于 `this` 是指向 `const` 对象的指针, `const` 成员函数不能修改调用该函数的对象。因此, 函数 `avg_price` 和函数 `same_isbn` 只能读取而不能修改调用它们的数据成员。



注解 `const` 对象、指向 `const` 对象的指针或引用只能用于调用其 `const` 成员函数, 如果尝试用它们来调用非 `const` 成员函数, 则是错误的。

4. this 指针的使用

在成员函数中, 不必显式地使用 `this` 指针来访问被调用函数所属对象的成员。对这个类的成员的任何没有前缀的引用, 都被假定为通过指针 `this` 实现的引用:

```
bool same_isbn(const Sales_item &rhs) const
{ return isbn == rhs.isbn; }
```

在这个函数中 `isbn` 的用法与 `this->units_sold` 或 `this->revenue` 的用法一样。

由于 `this` 指针是隐式定义的, 因此不需要在函数的形参表中包含 `this` 指针, 实际上, 这

样做也是非法的。但是，在函数体中可以显式地使用 `this` 指针。如下定义函数 `same_isbn` 尽管没有必要，但是却是合法的：

```
bool same_isbn(const Sales_item &rhs) const
{ return this->isbn == rhs.isbn; }
```

7.7.2 在类外定义成员函数

在类的定义外面定义成员函数必须指明它们是类的成员：

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

上述定义和其他函数一样：该函数返回类型为 `double`，在函数名后面的圆括号括起了一个空的形参表。新的内容则包括跟在形参表后面的 `const` 和函数名的形式。函数名：

```
Sales_item::avg_price
```

261

使用作用域操作符（1.2.2 节）指明函数 `avg_price` 是在类 `Sales_item` 的作用域范围内定义的。

形参表后面的 `const` 则反映了在类 `Sales_item` 中声明成员函数的形式。在任何函数定义中，返回类型和形参表必须和函数声明（如果有的话）一致。对于成员函数，函数声明必须与其定义一致。如果函数被声明为 `const` 成员函数，那么函数定义时形参表后面也必须有 `const`。

现在可以完全理解第一行代码了：这行代码说明现在正在定义类 `Sales_item` 的函数 `avg_price`，而且这是一个 `const` 成员函数，这个函数没有（显式的）形参，返回 `double` 类型的值。

函数体更加容易理解：检查 `units_sold` 是否为 0，如果不为 0，返回 `revenue` 除以 `units_sold` 的结果；如果 `units_sold` 是 0，不能安全地进行除法运算——除以 0 是未定义的行为。此时程序返回 0，表示没有任何销售时平均售价为 0。根据异常错误处理策略，也可以抛出异常来代替刚才的处理（6.13 节）。

7.7.3 编写 `Sales_item` 类的构造函数

还必须编写一个成员，那就是构造函数。正如在 2.8 节所学习的，在定义类时没有初始化它的数据成员，而是通过构造函数来初始化其数据成员。

1. 构造函数是特殊的成员函数

构造函数（`constructor`）是特殊的成员函数，与其他成员函数不同，构造函数和类同名，而且没有返回类型。而与其他成员函数相同的是，构造函数也有形参表（可能为空）和函数体。一个类可以有多个构造函数，每个构造函数必须有与其他构造函数不同数目或类型的形参。

构造函数的形参指定了创建类类型对象时使用的初始化式。通常，这些初始化式会用于初始化新创建对象的数据成员。构造函数通常应确保其每个数据成员都完成了初始化。

`Sales_item` 类只需要显式定义一个构造函数：没有形参的默认构造函数（`default constructor`）。默认构造函数说明当定义对象却没有为它提供（显式的）初始化式时应该怎么办：

```
vector<int> vi;      // default constructor: empty vector
string s;           // default constructor: empty string
Sales_item item;    // default constructor: ???
```

262 我们知道 `string` 和 `vector` 类默认构造函数的行为：这些构造函数会将对象初始化为合理的默认状态。`string` 的默认构造函数会产生空字符串，相当于 `" "`。`vector` 的默认构造函数则生成一个没有元素的 `vector` 向量对象。

同样地，我们希望类 `Sales_item` 的默认构造函数为它生成一个空的 `Sales_item` 对象。这里的“空”意味着对象中的 `isbn` 是空字符串，`units_sold` 和 `revenue` 则初始化为 0。

2. 构造函数的定义

和其他成员函数一样，构造函数也必须在类中声明，但是可以在类中或类外定义。由于我们的构造函数很简单，因此在类中定义它：

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
    // private members as before
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

在解释任何构造函数的定义之前，注意到构造函数是放在类的 `public` 部分的。通常构造函数会作为类的接口的一部分，这个例子也是这样。毕竟，我们希望使用类 `Sales_item` 的代码可以定义和初始化类 `Sales_item` 的对象。如果将构造函数定义为 `private` 的，则不能定义类 `Sales_item` 的对象，这样的话，这个类就没有什么用了。

对于定义本身：

```
// default constructor needed to initialize members of built-in type
Sales_item(): units_sold(0), revenue(0.0) { }
```

上述语句说明现在正在定义类 `Sales_item` 的构造函数，这个构造函数的形参表和函数体都为空。令人感兴趣的是冒号和冒号与定义（空）函数体的花括号之间的代码。

3. 构造函数的初始化列表

在冒号和花括号之间的代码称为构造函数的初始化列表（`constructor initializer list`）。构造函数的初始化列表为类的一个或多个数据成员指定初值。它跟在构造函数的形参表之后，以冒号开头。构造函数的初始化式是一系列成员名，每个成员后面是括在圆括号中的初始值。多个成员的初始化用逗号分隔。

263

上述例题的初始化列表表明 `units_sold` 和 `revenue` 成员都应初始化为 0。每当创建 `Sales_item` 对象时，它的这两个成员都以初值 0 出现。而 `isbn` 成员可以不必准确指明其初值。除非在初始化列表中有其他表述，否则具有类类型的成员皆被其默认构造函数自动初始化。于是，

isbn 由 string 类的默认构造函数初始化为空串。当然，如果有必要的话，也可以在初始化列表中指明 isbn 的默认初值。

解释了初始化列表后，就可以深入地了解这个构造函数了：它的形参表和函数体都为空。形参表为空是因为正在定义的构造函数是默认调用的，无需提供任何初值。函数体为空是因为除了初始化 units_sold 和 revenue 成员外没有其他工作可做了。初始化列表显式地将 units_sold 和 revenue 初始化为 0，并隐式地将 isbn 初始化为空串。当创建新的 Sales_item 对象时，数据成员将以这些值出现。

4. 合成的默认构造函数



如果没有为一个类显式定义任何构造函数，编译器将自动为这个类生成默认构造函数。

由编译器创建的默认构造函数通常称为合成的默认构造函数 (synthesized default constructor)，它将依据如同变量初始化 (2.3.4 节) 的规则初始化类中所有成员。对于具有类类型的成员，如 isbn，则会调用该成员所属类自身的默认构造函数实现初始化。内置类型成员的初值依赖于对象如何定义。如果对象在全局作用域中定义（即不在任何函数中）或定义为静态局部对象，则这些成员将被初始化为 0。如果对象在局部作用域中定义，则这些成员没有初始化。除了给它们赋值之外，出于其他任何目的对未初始化成员的使用都没有定义。



合成的默认构造函数一般适用于仅包含类类型成员的类。而对于含有内置类型或复合类型成员的类，则通常应该定义他们自己的默认构造函数初始化这些成员。

由于合成的默认构造函数不会自动初始化内置类型的成员，所以必须明确定义 Sales_item 类的默认构造函数。

7.7.4 类代码文件的组织

正如在 2.9 节提及的，通常将类的声明放置在头文件中。大多数情况下，在类外定义的成员函数则置于源文件中。C++ 程序员习惯使用一些简单的规则给头文件及其关联的类定义代码命名。类定义应置于名为 *type.h* 或 *type.H* 的文件中，*type* 指在该文件中定义的类的名字。成员函数的定义则一般存储在与类同名的源文件中。依照这些规则，我们将类 Sales_item 放在名为 Sales_item.h 的文件中定义。任何需使用这个类的程序，都必须包含这个头文件。而 Sales_item 的成员函数的定义则应该放在名为 Sales_item.cc 的文件中。这个文件同样也必须包含 Sales_item.h 头文件。

264

习题

习题 7.31 编写你自己的 Sales_item 类，添加两个公用 (public) 成员用于读和写 Sales_item 对象。这两个成员函数的功能应类似于第 1 章介绍的输入输出操作符。交易也应类似于那一章所定义的。利用这个类读入并输出一组交易。

习题 7.32 编写一个头文件，包含你自己的 Sales_item 类。使用通用的 C++ 规则给这个头文件以

及任何相关的文件命名，这些文件用于存储在类外定义的非内联函数。

习题 7.33 在 `Sales_item` 类中加入一个成员，用于添加两个 `Sales_item` 对象。使用修改后的类重新解决第 1 章给出的平均价格问题。

7.8 重载函数

出现在相同作用域中的两个函数，如果具有相同的名字而形参表不同，则称为**重载函数** (overloaded function)。

使用某种程序设计语言编写过算术表达式的程序员都肯定使用过重载函数。表达式

`1 + 3`

调用了针对整型操作数的加法操作符，而表达式

`1.0 + 3.0`

调用了另外一个专门处理浮点操作数的不同的加法操作。根据操作数的类型来区分不同的操作，并应用适当的操作，是编译器的责任，而不是程序员的事情。

类似地，程序员可以定义一组函数，它们执行同样的一般性动作，但是应用在不同的形参类型上。调用这些函数时，无需担心调用的是哪个函数，就像我们不必操心执行的是整数算术操作还是浮点数算术操作就可以实现 `int` 型加法或 `double` 型加法一样。

通过省去为函数起名并记住函数名字的麻烦，函数重载 (function overloading) 简化了程序的实现，使程序更容易理解。函数名只是为了帮助编译器判断调用的是哪个函数而已。例如，一个数据库应用可能需要提供多个 `lookup` 函数，分别实现基于姓名、电话号码或账号之类的查询功能。函数重载使我们可以定义一系列的函数，它们的名字都是 `lookup`，不同之处在于用于查询的值不相同。如此可传递几种类型中的任一种值调用 `lookup` 函数：

```
Record lookup(const Account&); // find by Account
Record lookup(const Phone&);   // find by Phone
Record lookup(const Name&);    // find by Name
Record r1, r2;
r1 = lookup(acct);             // call version that takes an Account
r2 = lookup(phone);            // call version that takes a Phone
```

这里的三个函数共享同一个函数名，但却是三个不同的函数。编译器将根据所传递的实参类型来判断调用的是哪个函数。

要理解函数重载，必须理解如何定义一组重载函数和编译器如何决定对某一调用使用哪个函数。本节的其余部分将会回顾这些主题。



任何程序都仅有一个 `main` 函数的实例。`main` 函数不能重载。

函数重载和重复声明的区别

如果两个函数声明的返回类型和形参表完全匹配，则将第二个函数声明视为第一个的重复声明。如果两个函数的形参表完全相同，但返回类型不同，则第二个声明是错误的：

```
Record lookup(const Account&);
```

```
bool lookup(const Account&); // error: only return type is different
```

函数不能仅仅基于不同的返回类型而实现重载。

有些看起来不相同的形参表本质上是相同的：

```
// each pair declares the same function
Record lookup(const Account &acct);
Record lookup(const Account&); // parameter names are ignored
typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&); // Telno and Phone are the same type
Record lookup(const Phone&, const Name&);
// default argument doesn't change the number of parameters
Record lookup(const Phone&, const Name& = "");
// const is irrelevant for nonreference parameters
Record lookup(Phone);
Record lookup(const Phone); // redeclaration
```

在第一对函数声明中，第一个声明给它的形参命了名。形参名只是帮助文档，并没有修改形参表。

266

在第二对函数声明中，看似形参类型不同，但注意到 Telno 其实并不是新类型，只是 Phone 类型的同义词。typedef 给已存在的数据类型提供别名，但并没有创建新的数据类型。所以，如果两个形参的差别只是一个使用 typedef 定义的类型名，而另一个使用 typedef 对应的原类型名，则这两个形参并无不同。

在第三对中，形参列表只有默认实参不同。默认实参并没有改变形参的个数。无论实参是由用户还是由编译器提供的，这个函数都带有两个实参。

最后一对的区别仅在于是否将形参定义为 const。这种差异并不影响传递至函数的对象；第二个函数声明被视为第一个的重复声明。其原因在于实参传递的方式。复制形参时并不考虑形参是否为 const——函数操纵的只是副本。函数无法修改实参。结果，既可将 const 对象传递给 const 形参，也可传递给非 const 形参，这两种形参并无本质区别。

值得注意的是，形参与 const 形参的等价性仅适用于非引用形参。有 const 引用形参的函数与有非 const 引用形参的函数是不同的。类似地，如果函数带有指向 const 类型的指针形参，则与带有指向相同类型的非 const 对象的指针形参的函数不相同。

建议：何时不重载函数名

虽然，对于通常的操作，重载函数能避免不必要的函数命名（和名字记忆），但很容易就会过分使用重载。在一些情况下，使用不同的函数名能提供较多的信息，使程序易于理解。考虑下面 Screen 类的一组用于移动屏幕光标的成员函数：

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

乍看上去，似乎把这组函数重载为名为 move 的函数更好一些：

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

其实不然，重载过后的函数失去了原来函数名所包含的信息，如此一来，程序变得晦涩难懂了。

虽则这几个函数共享的一般性动作都是光标移动，但特殊的移动性质却互不相同。例如，moveHome 表示的是光标移动的一个特殊实例。对于程序的读者，下面两种调用中，哪种更易于理解？而对于使用 Screen 类的程序员，哪一个调用又更容易记忆呢？

```
// which is easier to understand?
myScreen.home(); // we think this one!
myScreen.move();
```

267

7.8.1 重载与作用域

2.3.6 节的程序演示了 C++ 作用域的嵌套。在函数中局部声明的名字将屏蔽在全局作用域 (2.3.6 节) 内声明的同名名字。这个关于变量性质的性质对于函数名同样成立：

```
/* Program for illustration purposes only:
 * It is bad style for a function to define a local variable
 * with the same name as a global name it wants to use
 */
string init(); // the name init has global scope
void fcn()
{
    int init = 0; // init is local and hides global init
    string s = init(); // error: global init is hidden
}
```

一般的作用域规则同样适用于重载函数名。如果局部地声明一个函数，则该函数将屏蔽而不是重载在外层作用域中声明的同名函数。由此推论，每一个版本的重载函数都应在同一个作用域中声明。



一般来说，局部地声明函数是一种不明智的选择。函数的声明应放在头文件中。但为了解释作用域与重载的相互作用，我们将违反上述规则而使用局部函数声明。

作为例子，考虑下面的程序：

```
void print(const string &);
void print(double); // overloads the print function
void fooBar(int ival)
{
    void print(int); // new scope: hides previous instances of print
    print("Value: "); // error: print(const string &) is hidden
    print(ival); // ok: print(int) is visible
    print(3.14); // ok: calls print(int); print(double) is hidden
}
```

函数 fooBar 中的 print(int) 声明将屏蔽 print 的其他声明，就像只有一个有效的 print 函数一样：该函数仅带有一个 int 型形参。在这个作用域或嵌套在这个作用域里的其他作用域中，名字 print 的任何使用都将解释为这个 print 函数实例。

调用 print 时，编译器首先检索这个名字的声明，找到只有一个 int 型形参的 print 函数的局部声明。一旦找到这个名字，编译器将不再继续检查这个名字是否在外层作用域中存在，即编译器将认同找到的这个声明即是程序需要调用的函数，余下的工作只是检查该名字的使用是否有效。

第一个函数调用传递了一个字符串字面值，但是函数的形参却是 `int` 型的。字符串字面值无法隐式地转换为 `int` 型，因而该调用是错误的。`print(const string&)` 函数与这个函数调用匹配，但已被屏蔽，因此不在解释该调用时考虑。

268

当传递一个 `double` 数据调用 `print` 函数时，编译器重复了同样的匹配过程：首先检索到 `print(int)` 的局部声明，然后将 `double` 型的实参隐式转换为 `int` 型。因此，该调用合法。



在 C++ 中，名字查找发生在类型检查之前。

另一种情况是，在与其他 `print` 函数相同的作用域中声明 `print(int)`，这样，它就成为 `print` 函数的另一个重载版本。此时，所有的调用将以不同的方式解释：

```
void print(const string &);
void print(double); // overloads print function
void print(int);    // another overloaded instance.
void fooBar2(int ival)
{
    print("Value: "); // ok: calls print(const string &)
    print(ival);      // ok: print(int)
    print(3.14);      // ok: calls print (double)
}
```

现在，编译器在检索名字 `print` 时，将找到这个名字的三个函数。每一个调用都将选择与其传递的实参相匹配的 `print` 版本。

习题

习题 7.34 定义一组名为 `error` 的重载函数，使之与下面的调用匹配：

```
int index, upperBound;
char selectVal;
// ...
error("Subscript out of bounds: ", index, upperBound);
error("Division by zero");
error("Invalid selection", selectVal);
```

习题 7.35 下面提供了三组函数声明，解释每组中第二个声明的效果，并指出哪些（如果有的话）是不合法的。

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

7.8.2 函数匹配与实参转换

函数重载确定（`overload resolution`，即函数匹配 `function matching`）是将函数调用与重载函数集合中的一个函数相关联的过程。通过自动提取函数调用中实际使用的实参与重载集合中各个

函数提供的形参做比较，编译器实现该调用与函数的匹配。匹配结果有三种可能：

- (1) 编译器找到与实参最佳匹配 (best match) 的函数，并生成调用该函数的代码。
- (2) 找不到形参与函数调用的实参匹配的函数，在这种情况下，编译器将给出编译错误信息。
- (3) 存在多个与实参匹配的函数，但没有一个是明显的最佳选择。这种情况也是错误的，该调用具有二义性 (ambiguous)。

大多数情况下，编译器都可以直接明确地判断一个实际的调用是否合法，如果合法，则应该调用哪一个函数。重载集合中的函数通常有不同个数的参数或无关联的参数类型。当多个函数的形参具有可通过隐式转换 (5.12 节) 关联起来的类型，则函数匹配将相当灵活。在这种情况下，需要程序员充分地掌握函数匹配的过程。

269

7.8.3 重载确定的三个步骤

考虑下面的这组函数和函数调用：

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

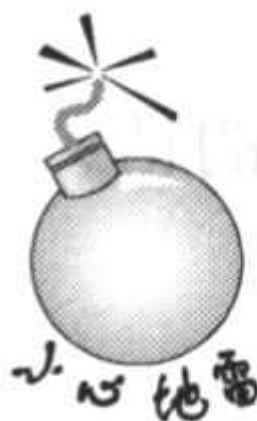
1. 候选函数

函数重载确定的第一步是确定该调用所考虑的重载函数集合，该集合中的函数称为**候选函数** (candidate function)。候选函数是与被调函数同名的函数，并且在调用点上，它的声明可见。在这个例子中，有四个名为 `f` 的候选函数。

2. 选择可行函数

第二步是从候选函数中选择一个或多个函数，它们能够用该调用中指定的实参来调用。因此，选出来的函数称为**可行函数** (viable function)。可行函数必须满足两个条件：第一，函数的形参个数与该调用的实参个数相同；第二，每一个实参的类型必须与对应形参的类型匹配，或者可被隐式转换为对应的形参类型。

270



如果函数具有默认实参 (7.4.1 节)，则调用该函数时，所用的实参可能比实际需要的少。默认实参也是实参，在函数匹配过程中，它的处理方式与其他实参一样。

对于函数调用 `f(5.6)`，可首先排除两个实参个数不匹配的候选函数。没有形参的 `f` 函数和有两个 `int` 型形参的 `f` 函数对于这个函数调用来说都不可行。例中的调用只有一个实参，而这些函数分别带有零个和两个形参。

另一方面，有两个 `double` 型参数的 `f` 函数可能是可行的。调用带有默认实参 (7.4.1 节) 的函数时可忽略这个实参。编译器自动将默认实参的值提供给被忽略的实参。因此，某个调用拥有的实参可能比显式给出的多。

根据实参个数选出潜在的可行函数后，必须检查实参的类型是否与对应的形参类型匹配。与任意函数调用一样，实参必须与它的形参匹配，它们的类型要么精确匹配，要么实参类型能够转换为形参类型。在这个例子中，余下的两个函数都是可行的。

- `f(int)` 是一个可行函数，因为通过隐式转换可将函数调用中的 `double` 型实参转换为该函数唯一的 `int` 型形参。
- `f(double, double)` 也是一个可行函数，因为该函数为其第二个形参提供了默认实参，而且第一个形参是 `double` 类型，与实参类型精确匹配。



如果没有找到可行函数，则该调用错误。

3. 寻找最佳匹配（如果有的话）

函数重载确定的第三步是确定与函数调用中使用的实际参数匹配最佳的可行函数。这个过程考虑函数调用中的每一个实参，选择对应形参与之最匹配的一个或多个可行函数。这里所谓“最佳”的细节将在下一节中解释，其原则是实参类型与形参类型越接近则匹配越佳。因此，实参类型与形参类型之间的精确类型匹配比需要转换的匹配好。

在上述例子中，只需考虑一个 `double` 类型的显式实参。如果调用 `f(int)`，实参需从 `double` 型转换为 `int` 型。而另一个可行函数 `f(double, double)` 则与该实参精确匹配。由于精确匹配优于需要类型转换的匹配，因此编译器将会把函数调用 `f(5.6)` 解释为对带有两个 `double` 形参的 `f` 函数的调用。

271

4. 含有多个形参的重载确定

如果函数调用使用了两个或两个以上的显式实参，则函数匹配会更加复杂。假设有同样的名为 `f` 的函数，分析下面的函数调用：

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

可行函数集合将以同样的方式选出。编译器将选出形参个数和类型都与实参匹配的函数。在本例中，可行函数是 `f(int, int)` 和 `f(double, double)`。接下来，编译器通过依次检查每一个实参来决定哪个或哪些函数匹配最佳。如果有且仅有一个函数满足下列条件，则匹配成功：

- (1) 其每个实参的匹配都不劣于其他可行函数需要的匹配。
- (2) 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后，仍找不到唯一最佳匹配函数，则该调用错误。编译器将提示该调用具有二义性。

在本例子的调用中，首先分析第一个实参，发现函数 `f(int, int)` 匹配精确。如果使之与第二个函数匹配，就必须将 `int` 型实参 42 转换为 `double` 型的值。通过内置转换的匹配“劣于”精确匹配。所以，如果只考虑这个形参，带有两个 `int` 型形参的函数比带有两个 `double` 型形参的函数匹配更佳。

但是，当分析第二个实参时，有两个 `double` 型形参的函数为实参 2.56 提供了精确匹配。而调用两个 `int` 型形参的 `f` 函数版本则需要把 2.56 从 `double` 型转换为 `int` 型。所以只考虑第二个形参的话，函数 `f(double, double)` 匹配更佳。

因此，这个调用有二义性：每个可行函数都对函数调用的一个实参实现更好的匹配。编译器将产生错误。解决这样的二义性，可通过显式的强制类型转换强制函数匹配：

```
f(static_cast<double>(42), 2.56); // calls f(double, double)
f(42, static_cast<int>(2.56));    // calls f(int, int)
```



在实际应用中，调用重载函数时应尽量避免对实参做强制类型转换：需要使用强制类型转换意味着所设计的形参集合不合理。

习题

习题 7.36 什么是候选函数？什么是可行函数？

习题 7.37 已知本节所列出的 `f` 函数的声明，判断下面哪些函数调用是合法的。如果有的话，列出每个函数调用的可行函数。如果调用非法，指出是没有函数匹配还是该调用存在二义性。如果调用合法，指出哪个函数是最佳匹配。

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

7.8.4 实参类型转换

为了确定最佳匹配，编译器将实参类型到相应形参类型的转换划分等级。转换等级以降序排列如下：

- (1) 精确匹配 (exact match)。实参与形参类型相同。
- (2) 通过类型提升 (promotion) 实现的匹配 (5.12.2 节)。
- (3) 通过标准转换 (standard conversion) 实现的匹配 (5.12.3 节)。
- (4) 通过类类型转换 (class-type conversion) 实现的匹配 (14.9 节将介绍这类转换)。



内置类型的提升和转换可能会使函数匹配产生意想不到的结果。但幸运的是，设计良好的系统很少会包含与下面例子类似的形参类型如此接近的函数。

通过这些例子，学习并加深了解特殊的函数匹配和内置类型之间的一般关系。

1. 需要类型提升或转换的匹配

类型提升或转换适用于实参类型可通过某种标准转换提升或转换为适当的形参类型的情况。

必须注意的一个重点是较小的整型提升为 `int` 型。假设有两个函数，一个的形参为 `int` 型，另一个的形参则是 `short` 型。对于任意整型的实参值，`int` 型版本都是优于 `short` 型版本的较佳匹配，即使从形式上看 `short` 型版本的匹配较佳：

```
void ff(int);
void ff(short);
ff('a');    // char promotes to int, so matches f(int)
```

字符字面值是 `char` 类型，`char` 类型可提升为 `int` 型。提升后的类型与函数 `ff(int)` 的形参类型匹配。`char` 类型同样也可转换为 `short` 型，但需要类型转换的匹配“劣于”需要类型提升的匹配。结果应将该调用解释为对 `ff(int)` 的调用。

通过类型提升实现的转换优于其他标准转换。例如，对于 `char` 型实参来说，有 `int` 型形参的函数是优于有 `double` 型形参的函数的较佳匹配。其他的标准转换也以相同的规则处理。例如，从 `char` 型到 `unsigned char` 型的转换的优先级不比从 `char` 型到 `double` 型的转换高。再举一

个具体的例子，考虑：

```
extern void manip(long);
extern void manip(float);
manip(3.14); // error: ambiguous call
```

字面值常量 3.14 的类型为 double。这种类型既可转为 long 型也可转为 float 型。由于两者都是可行的标准转换，因此该调用具有二义性。没有哪个标准转换比其他标准转换具有更高的优先级。

2. 参数匹配和枚举类型

回顾枚举类型 enum，我们知道这种类型的对象只能用同一枚举类型的另一个对象或一个枚举成员（enumerator）进行初始化（2.7 节）。整数对象即使具有与枚举元素相同的值也不能用于调用期望获得枚举类型实参的函数。

```
enum Tokens {INLINE = 128, VIRTUAL = 129};
void ff(Tokens);
void ff(int);
int main() {
    Tokens curTok = INLINE;
    ff(128); // exactly matches ff(int)
    ff(INLINE); // exactly matches ff(Tokens)
    ff(curTok); // exactly matches ff(Tokens)
    return 0;
}
```

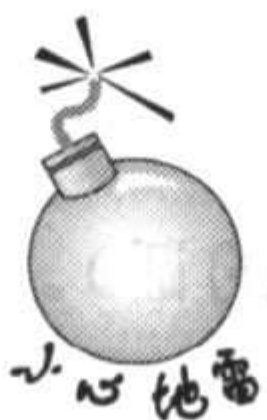
传递字面值常量 128 的函数调用与有一个 int 型参数的 ff 版本匹配。

虽然无法将整型值传递给枚举类型的形参，但可以将枚举值传递给整型形参。此时，枚举值被提升为 int 型或更大的整型。具体的提升类型取决于枚举成员的值。如果是重载函数，枚举值提升后的类型将决定调用哪个函数：

```
void newf(unsigned char);
void newf(int);
unsigned char uc = 129;
newf(VIRTUAL); // calls newf (int)
newf(uc); // calls newf (unsigned char)
```

枚举类型 Tokens 只有两个枚举成员，最大的值为 129。这个值可以用 unsigned char 类型表示，很多编译器会将这个枚举类型存储为 unsigned char 类型。然而，枚举成员 VIRTUAL 却并不是 unsigned char 类型。就算枚举成员的值能存储在 unsigned char 类型中，枚举成员和枚举类型的值也不会提升为 unsigned char 类型。

274



在使用有枚举类型形参的重载函数时，请记住：由于不同枚举类型的枚举常量值不相同，在函数重载确定过程中，不同的枚举类型会具有完全不同的行为。其枚举成员决定了它们提升的类型，而所提升的类型依赖于机器。

3. 重载和 const 形参



仅当形参是引用或指针时，形参是否为 const 才有影响。

可基于函数的引用形参是指向 const 对象还是指向非 const 对象，实现函数重载。将引用形参

定义为 const 来重载函数是合法的，因为编译器可以根据实参是否为 const 确定调用哪一个函数：

```
Record lookup(Account&);
Record lookup(const Account&); // new function
const Account a(0);
Account b;
lookup(a); // calls lookup(const Account&)
lookup(b); // calls lookup(Account&)
```

如果形参是普通的引用，则不能将 const 对象传递给这个形参。如果传递了 const 对象，则只有带 const 引用形参的版本才是该调用的可行函数。

如果传递的是非 const 对象，则上述任意一种函数皆可行。非 const 对象既可用于初始化 const 引用，也可用于初始化非 const 引用。但是，将 const 引用初始化为非 const 对象，需通过转换来实现，而非 const 形参的初始化则是精确匹配。

对指针形参的相关处理如出一辙。可将 const 对象的地址值只传递给带有指向 const 对象的指针形参的函数。也可将指向非 const 对象的指针传递给函数的 const 或非 const 类型的指针形参。如果两个函数仅在指针形参是否指向 const 对象上不同，则指向非 const 对象的指针形参对于指向非 const 对象的指针（实参）来说是最佳的匹配。重复强调，编译器可以判断：如果实参是 const 对象，则调用带有 const* 类型形参的函数；否则，如果实参不是 const 对象，将调用带有普通指针形参的函数。

注意不能基于指针本身是否为 const 来实现函数的重载：

```
f(int *);
f(int *const); // redeclaration
```

此时，const 用于修饰指针本身，而不是修饰指针所指向的类型。在上述两种情况中，都复制了指针，指针本身是否为 const 并没有带来区别。正如前面 7.8 节所提到的，当形参以副本传递时，不能基于形参是否为 const 来实现重载。

275

习题

习题 7.38 给出如下声明：

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

对于下面两组函数调用，请指出实参上每个转换的优先级等级（7.8.4 节）？

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

习题 7.39 解释以下每组声明中的第二个函数声明所造成的影响，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);
```

习题 7.40 下面的函数调用是否合法？如果不合法，请解释原因。

```
enum Stat { Fail, Pass };
void test(Stat);
test(0);
```

7.9 指向函数的指针

函数指针是指指向函数而非指向对象的指针。像其他指针一样，函数指针也指向某个特定的类型。函数类型由其返回类型以及形参表确定，而与函数名无关：

```
// pf points to function returning bool that takes two const string references
bool (*pf)(const string &, const string &);
```

这个语句将 pf 声明为指向函数的指针，它所指向的函数带有两个 const string&类型的形参和 bool 类型的返回值。



*pf 两侧的圆括号是必需的：

```
// declares a function named pf that returns a bool*
bool *pf(const string &, const string &);
```

1. 用 typedef 简化函数指针的定义

函数指针类型相当地冗长。使用 typedef 为指针类型定义同义词，可将函数指针的使用大大简化（2.6 节）：

```
typedef bool (*cmpFcn)(const string &, const string &);
```

该定义表示 cmpFcn 是一种指向函数的指针类型的名字。该指针类型为“指向返回 bool 类型并带有两个 const string 引用形参的函数的指针”。在要使用这种函数指针类型时，只需直接使用 cmpFcn 即可，不必每次都把整个类型声明全部写出来。

2. 指向函数的指针的初始化和赋值

在引用函数名但又没有调用该函数时，函数名将被自动解释为指向函数的指针。假设有函数：

```
// compares lengths of two strings
bool lengthCompare(const string &, const string &);
```

除了用作函数调用的左操作数以外，对 lengthCompare 的任何使用都被解释为如下类型的指针：

```
bool (*)(const string &, const string &);
```

可使用函数名对函数指针做初始化或赋值：

```
cmpFcn pf1 = 0; // ok: unbound pointer to function
cmpFcn pf2 = lengthCompare; // ok: pointer type matches function's type
pf1 = lengthCompare; // ok: pointer type matches function's type
pf2 = pf1; // ok: pointer types match
```

此时，直接引用函数名等效于在函数名上应用取地址操作符：

```
cmpFcn pf1 = lengthCompare;
cmpFcn pf2 = &lengthCompare;
```



函数指针只能通过同类型的函数或函数指针或 0 值常量表达式进行初始化或赋值。

将函数指针初始化为 0，表示该指针不指向任何函数。

指向不同函数类型的指针之间不存在转换：

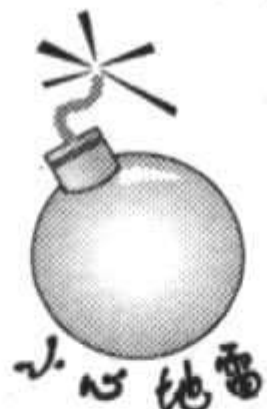
```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(char*, char*);
// pointer to function returning bool taking two const string&
cmpFcn pf;
pf = sumLength;           // error: return type differs
pf = cstringCompare;      // error: parameter types differ
pf = lengthCompare;       // ok: function and pointer types match exactly
```

277

3. 通过指针调用函数

指向函数的指针可用于调用它所指向的函数。可以不需要使用解引用操作符，直接通过指针调用函数：

```
cmpFcn pf = lengthCompare;
lengthCompare("hi", "bye"); // direct call
pf("hi", "bye");             // equivalent call: pf1 implicitly dereferenced
(*pf)("hi", "bye");          // equivalent call: pf1 explicitly dereferenced
```



如果指向函数的指针没有初始化，或者具有 0 值，则该指针不能在函数调用中使用。只有当指针已经初始化，或被赋值为指向某个函数，方能安全地用来调用函数。

4. 函数指针形参

函数的形参可以是指向函数的指针。这种形参可以用以下两种形式编写：

```
/* useBigger function's third parameter is a pointer to function
 * that function returns a bool and takes two const string references
 * two ways to specify that parameter:
 */
// third parameter is a function type and is automatically treated as a pointer to function
void useBigger(const string &, const string &,
               bool(const string &, const string &));
// equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &, const string &,
               bool (*)(const string &, const string &));
```

5. 返回指向函数的指针

函数可以返回指向函数的指针，但是，正确写出这种返回类型相当不容易：

```
// ff is a function taking an int and returning a function pointer
// the function pointed to returns an int and takes an int* and an int
int (*ff(int))(int*, int);
```



阅读函数指针声明的最佳方法是从声明的名字开始由里而外理解。

要理解该声明的含义，首先观察：

```
ff(int)
```

将 ff 声明为一个函数，它带有一个 int 型的形参。该函数返回

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

它是一个指向函数的指针，所指向的函数返回 int 型并带有两个分别是 int* 型和 int 型的形参。

使用 typedef 可使该定义更简明易懂：

```
// PF is a pointer to a function returning an int, taking an int* and an int
typedef int (*PF)(int*, int);
PF ff(int); // ff returns a pointer to function
```



允许将形参定义为函数类型，但函数的返回类型则必须是指向函数的指针，而不能是函数。

具有函数类型的形参所对应的实参将被自动转换为指向相应函数类型的指针。但是，当返回的是函数时，同样的转换操作则无法实现：

```
// func is a function type, not a pointer to function!
typedef int func(int*, int);
void f1(func); // ok: f1 has a parameter of function type
func f2(int); // error: f2 has a return type of function type
func *f3(int); // ok: f3 returns a pointer to function type
```

6. 指向重载函数的指针：

C++ 语言允许使用函数指针指向重载的函数：

```
extern void ff(vector<double>);
extern void ff(unsigned int);

// which function does pf1 refer to?
void (*pf1)(unsigned int) = &ff; // ff(unsigned)
```

指针的类型必须与重载函数的一个版本精确匹配。如果没有精确匹配的函数，则对该指针的初始化或赋值都将导致编译错误：

```
// error: no match: invalid parameter list
void (*pf2)(int) = &ff;

// error: no match: invalid return type
double (*pf3)(vector<double>);
pf3 = &ff;
```

小结

函数是有名字的计算单元，对程序（就算是小程序）的结构化至关重要。函数的定义由返回类型、函数名、形参表（可能为空）以及函数体组成。函数体是调用函数时执行的语句块。在调用函数时，传递给函数的实参必须与相应的形参类型兼容。

给函数传递实参遵循变量初始化的规则。非引用类型的形参以相应实参的副本初始化。对（非引用）形参的任何修改仅作用于局部副本，并不影响实参本身。

复制庞大而复杂的值有昂贵的开销。为了避免传递副本的开销，可将形参指定为引用类型。对引用形参的任何修改会直接影响实参本身。应将不需要修改相应实参的引用形参定义为 const

引用。

在 C++ 中，函数可以重载。只要函数中形参的个数或类型不同，则同一个函数名可用于定义不同的函数。编译器将根据函数调用时的实参确定调用哪一个函数。在重载函数集合中选择适合的函数的过程称为函数匹配。

C++ 提供了两种特殊的函数：内联函数和成员函数。将函数指定为内联是建议编译器在调用点直接把函数代码展开。内联函数避免了调用函数的代价。成员函数则是身为类成员的函数。本章介绍了简单的成员函数，在第 12 章将会更详细地介绍成员函数。

术语

ambiguous call (有二义性的调用) 一种编译错误，当调用重载函数，找不到唯一的最佳匹配时产生。

argument (实参) 调用函数时提供的值。这些值用于初始化相应的形参，其方式类似于初始化同类型变量的方法。

automatic object (自动对象) 局部于函数的对象。自动对象会在每一次函数调用时重新创建和初始化，并在定义它的函数块结束时撤销。一旦函数执行完毕，这些对象就不再存在了。

best match (最佳匹配) 在重载函数集合里找到的与给定调用的实参达到最佳匹配的唯一函数。

call operator (调用操作符) 使函数执行的操作符。该操作符是一对圆括号，并且有两个操作数：被调用函数的名字，以及由逗号分隔的（也可能为空）形参表。

candidate function (候选函数) 在解析函数调用时考虑的函数集合。候选函数包括了所有在该调用发生的作用域中声明的、具有该调用所使用的名字的函数。

const member function (常量成员函数) 类的成员的函数，并可以由该类类型的常量对象调用。常量成员函数不能修改所操纵的对象的数据成员。

constructor (构造函数) 与所属类同名的类成员函数。构造函数说明如何初始化本类的对象。构造函数没有返回类型，而且可以重载。

constructor initializer list (构造函数初始化列表) 在构造函数中用于为数据成员指定初值的表。初始化列表出现在构造函数的定义中，位于构造函数体与形参表之间。该表由冒号和冒号后面的一组用逗号分隔的成员名组成，每一个成员名后面跟着用圆括号括起来的该成员的初值。

default constructor (默认构造函数) 在没有显式提供初始化式时调用的构造函数。如果类中没有定义任何构造函数，编译器会自动为这个类合成默认构造函数。

function (函数) 可调用的计算单元。

function body (函数体) 定义函数动作的语句块。

function matching (函数匹配) 确定重载函数调用的编译器过程。调用时使用的实参将与每个重载函数的形参表作比较。

function prototype (函数原型) 函数声明的同义词。包括了函数的名字、返回类型和形参类型。调用函数时，必须在调用点之前声明函数原型。

inline function (内联函数) 如果可能的话, 将在调用点展开的函数。内联函数直接以函数代码替代了函数调用语句, 从而避免了一般函数调用的开销。

local static object (局部静态对象) 在函数第一次调用前就已经创建和初始化的局部对象, 其值在函数的调用之间保持有效。

local variable (局部变量) 在函数内定义的变量, 仅能在函数体内访问。

object lifetime (对象生命期) 每个对象皆有与之关联的生命期。在块中定义的对象从定义时开始存在, 直到它的定义所在的语句块结束为止。静态局部对象和在函数外定义的全局变量则在程序开始执行时创建, 当 main 函数结束时撤销。动态创建的对象由 new 表达式创建, 从此开始存在, 直到由相应的 delete 表达式释放所占据的内存空间为止。

overload resolution (重载确定) 函数匹配的同义词。

overloaded function (重载函数) 和至少一个其他函数同名的函数。重载函数必须在形参的个数或类型上有所不同。

parameter (形参) 函数的局部变量, 其初值由函数调用提供。

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

return type (返回类型) 函数返回值的类型。

synthesized default constructor (合成默认构造函数) 如果类没有定义任何构造函数, 则编译器会为这个类创建(合成)一个默认构造函数。该函数以默认的方式初始化类中的所有数据成员。

temporary object (临时对象) 在求解表达式的过程中由编译器自动创建的没有名字的对象。“临时对象”这个术语通常简称为“临时”。临时对象一直存在直到最大表达式结束为止, 最大表达式指的是包含创建该临时对象的表达式的最大范围内的表达式。

this pointer (this 指针) 成员函数的隐式形参。this 指针指向调用该函数的对象, 是指向类类型的指针。在 const 成员函数中, 该指针也指向 const 对象。

viable function (可行函数) 重载函数中可与指定的函数调用匹配的子集。可行函数的形参个数必须与该函数调用的实参个数相同, 而且每个实参类型都可潜在地转换为相应形参的类型。

第 8 章

标准 IO 库

目录

8.1 面向对象的标准库.....	244
8.2 条件状态.....	247
8.3 输出缓冲区的管理.....	249
8.4 文件的输入和输出.....	251
8.5 字符串流.....	257
小结.....	259
术语.....	259

C++的输入/输出（input/output）由标准库提供。标准库定义了一族类型，支持对文件和控制窗口等设备的读写（IO）。还定义了一些其他类型，使 string 对象能够像文件一样操作，从而使我们无须 IO 就能实现数据与字符之间的转换。这些 IO 类型都定义了如何读写内置数据类型的值。此外，一般来说，类的设计者还可以很方便地使用 IO 标准库设施读写自定义类的对象。类类型通常使用 IO 标准库为内置类型定义的操作符和规则来进行读写。

本章将介绍 IO 标准库的基础知识，而更多的内容会在后续章节中介绍：第 14 章考虑如何编写自己的输入输出操作符；附录 A 则介绍格式控制以及文件的随机访问。

前面的程序已经使用了多种 IO 标准库提供的工具：

- `istream`（输入流）类型，提供输入操作。
- `ostream`（输出流）类型，提供输出操作。
- `cin`（发音为 see-in）：读入标准输入的 `istream` 对象。
- `cout`（发音为 see-out）：写到标准输出的 `ostream` 对象。
- `cerr`（发音为 see-err）：输出标准错误的 `ostream` 对象。`cerr` 常用于程序错误信息。
- `>>` 操作符，用于从 `istream` 对象中读入输入。
- `<<` 操作符，用于把输出写到 `ostream` 对象中。
- `getline` 函数，需要分别取 `istream` 类型和 `string` 类型的两个引用形参，其功能是从 `istream` 对象读取一个单词，然后写入 `string` 对象中。

本章简要地介绍一些附加的 IO 操作，并讨论文件对象和 `string` 对象的读写。附录 A 会介绍如何控制 IO 操作的格式、文件的随机访问以及无格式的 IO。本书是初级读本，因此不会详细讨论完整的 `iostream` 标准库——特别是，我们不但没有涉及系统特定的实现细则，也不讨论标准库管理输入输出缓冲区的机制，以及如何编写自定义的缓冲区类。这些话题已超出了本书的范围。相对而言，本书把重点放在 IO 标准库对普通程序最有用的部分。

8.1 面向对象的标准库

迄今为止，我们已经使用 IO 类型和对象读写数据流，它们常用于与用户控制窗口的交互。当然，实际的程序不能仅限于对控制窗口的 IO，通常还需要读或写已命名的文件。此外，程序还应该能方便地使用 IO 操作格式化内存中的数据，从而避免读写磁盘或其他设备的复杂性和运行代价。应用程序还需要支持宽字符（wide-character）语言的读写。

从概念上看，无论是设备的类型还是字符的大小，都不影响需要执行的 IO 操作。例如，不管我们是从控制窗口、磁盘文件或内存中的字符串读入数据，都可使用 `>>` 操作符。相似地，无论我们读的是 `char` 类型的字符还是 `wchar_t` 类型（2.1.1 节）的字符，也都可以使用该操作符。

乍看起来，要同时支持或使用不同类型的设备以及不同大小的字符流，其复杂程度似乎相当可怕。为了管理这样的复杂性，标准库使用了**继承**（inheritance）来定义一组**面向对象**（object-oriented）类。在本书的第 IV 部分将会更详细地讨论继承和面向对象程序设计，不过，一般而言，通过继承关联起来的类型都共享共同的接口。当一个类继承另一个类时，这两个类通常可以使用相同的操作。更确切地说，如果两种类型存在继承关系，则可以说一个类“继承”了其父类的行为——接口。C++中所提及的父类称为**基类**（base class），而继承而来的类则称为**派生类**（derived class）。

IO 类型在三个独立的头文件中定义：`iostream` 定义读写控制窗口的类型，`fstream` 定义读写已命名文件的类型，而 `sstream` 所定义的类型则用于读写存储在内存中的 `string` 对象。在 `fstream` 和 `sstream` 里定义的每种类型都是从 `iostream` 头文件中定义的相关类型派生而来。表 8-1 列出了 C++ 的 IO 类，而图 8-1 则阐明这些类型之间的继承关系。继承关系通常可以用类似于家族树的图解说明。最顶端的圆圈代表基类（或称“父类”），基类和派生类（或称“子类”）之

间用线段连接。因此，如图 8-1 所示，`istream` 是 `ifstream` 和 `istringstream` 的基类，同时也是 `iostream` 的基类，而 `iostream` 则是 `stringstream` 和 `fstream` 的基类。

表 8-1 IO 标准库类型和头文件	
头文件	类 型
<code>iostream</code>	<code>istream</code> 从流中读取 <code>ostream</code> 写到流中去 <code>iostream</code> 对流进行读写；从 <code>istream</code> 和 <code>ostream</code> 派生而来
<code>fstream</code>	<code>ifstream</code> 从文件中读取；由 <code>istream</code> 派生而来 <code>ofstream</code> 写到文件中去；由 <code>ostream</code> 派生而来 <code>fstream</code> 读写文件；由 <code>iostream</code> 派生而来
<code>sstream</code>	<code>istringstream</code> 从 <code>string</code> 对象中读取；由 <code>istream</code> 派生而来 <code>ostringstream</code> 写到 <code>string</code> 对象中去；由 <code>ostream</code> 派生而来 <code>stringstream</code> 对 <code>string</code> 对象进行读写；由 <code>iostream</code> 派生而来

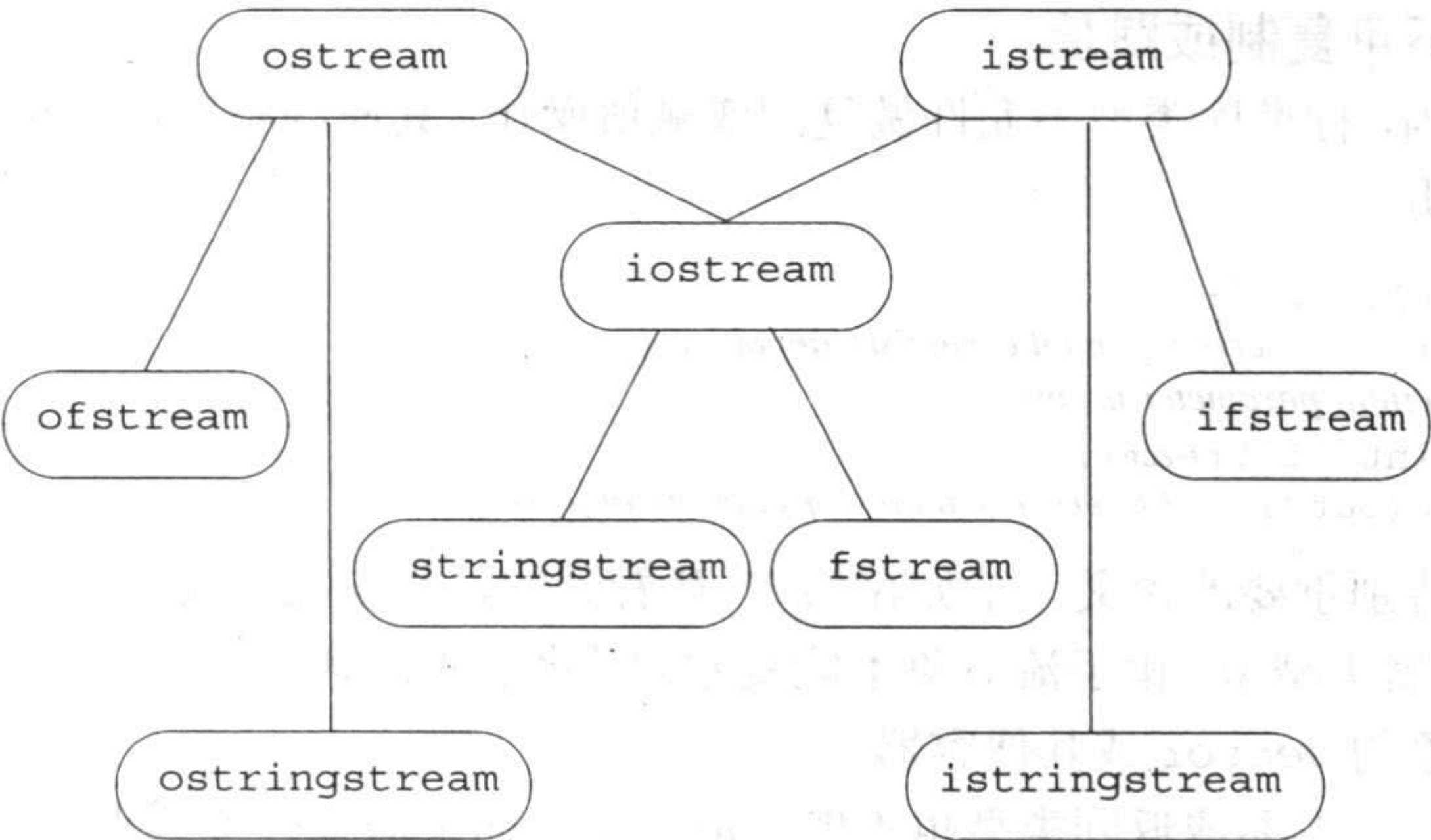


图 8-1 简单的 `iostream` 继承层次

由于 `ifstream` 和 `istringstream` 类型继承了 `istream` 类，因此已知这两种类型的大量用法。我们曾经编写过的读 `istream` 对象的程序也可用于读文件（使用 `ifstream` 类型）或者 `string` 对象（使用 `istringstream` 类型）。类似地，提供输出功能的程序同样可用 `ofstream` 或 `ostringstream` 取代 `ostream` 类型实现。除了 `istream` 和 `ostream` 类型之外，`iostream` 头文件还定义了 `iostream` 类型。尽管我们的程序还没用过这种类型，但事实上可以多了解一些关于 `iostream` 的用法。`iostream` 类型由 `istream` 和 `ostream` 两者派生而来。这意味着 `iostream` 对象共享了它的两个父类的接口。也就是说，可使用 `iostream` 类型在同一个流上实现输入和输出操作。标准库还定义了另外两个继承 `iostream` 的类型。这些类型可用于读写文件或 `string` 对象。

对 IO 类型使用继承还有另外一个重要的含义：正如在第 15 章可以看到的，如果函数有基类类型的引用形参时，可以给函数传递其派生类型的对象。这就意味着：对 `istream&` 进行操作的函数，也可使用 `ifstream` 或者 `istringstream` 对象来调用。类似地，形参为 `ostream&` 类型的

函数也可用 `ofstream` 或者 `ostringstream` 对象调用。因为 IO 类型通过继承关联，所以可以只编写一个函数，而将它应用到三种类型的流上：控制台、磁盘文件或者字符串流。

1. 国际字符的支持

迄今为止，所描述的流类（stream class）读写的是由 `char` 类型组成的流。此外，标准库还定义了一组相关的类型，支持 `wchar_t` 类型。每个类都加上“w”前缀，以此与 `char` 类型的版本区分开来。于是，`wostream`、`wistream` 和 `wiostream` 类型从控制窗口读写 `wchar_t` 数据。相应的文件输入输出类是 `wifstream`、`wofstream` 和 `wfstream`。而 `wchar_t` 版本的 string 输入/输出流则是 `wstringstream`、`wostream` 和 `wstringstream`。标准库还定义了从标准输入输出读写宽字符的对象。这些对象加上“w”前缀，以此与 `char` 类型版本区分：`wchar_t` 类型的标准输入对象是 `wcin`；标准输出是 `wcout`；而标准错误则是 `wcerr`。

每一个 IO 头文件都定义了 `char` 和 `wchar_t` 类型的类和标准输入/输出对象。基于流的 `wchar_t` 类型的类和对象在 `iostream` 中定义，宽字符文件流类型在 `fstream` 中定义，而宽字符 `stringstream` 则在 `sstream` 头文件中定义。

286

2. IO 对象不可复制或赋值

出于某些原因，标准库类型不允许做复制或赋值操作。其原因将在后面第 III 和第 IV 部分学习类和继承时阐明。

```
ofstream out1, out2;
out1 = out2;    // error: cannot assign stream objects
// print function: parameter is copied
ofstream print(out1);
out2 = print(out2); // error: cannot copy stream objects
```

这个要求有两层特别重要的含义。正如在第 9 章看到的，只有支持复制的元素类型可以存储在 `vector` 或其他容器类型里。由于流对象不能复制，因此不能存储在 `vector`（或其他）容器中（即不存在存储流对象的 `vector` 或其他容器）。

第二个含义是：形参或返回类型也不能为流类型。如果需要传递或返回 IO 对象，则必须传递或返回指向该对象的指针或引用：

```
ofstream &print(out1);    // ok: takes a reference, no copy
while (print(out2)) { /* ... */ } // ok: pass reference to out2
```

一般情况下，如果要传递 IO 对象以便对它进行读写，可用非 `const` 引用的方式传递这个流对象。对 IO 对象的读写会改变它的状态，因此引用必须是非 `const` 的。

习题

习题 8.1 假设 `os` 是一个 `ofstream` 对象，下面程序做了什么？

```
os << "Goodbye!" << endl;
```

如果 `os` 是 `ostringstream` 对象呢？或者，`os` 是 `ifstream` 对象呢？

习题 8.2 下面的声明是错误的，指出其错误并改正之：

```
ostream print(ostream os);
```

8.2 条件状态

在展开讨论 `fstream` 和 `sstream` 头文件中定义的类型之前,需要了解更多 IO 标准库如何管理其缓冲区及其流状态的相关内容。谨记本节和下一节所介绍的内容同样适用于普通流、文件流以及 `string` 流。

实现 IO 的继承正是错误发生的根源。一些错误是可恢复的;一些错误则发生在系统底层,位于程序可修正的范围之外。IO 标准库管理一系列条件状态 (condition state) 成员,用来标记给定的 IO 对象是否处于可用状态,或者碰到了哪种特定的错误。表 8-2 列出了标准库定义的一组函数和标记,提供访问和操纵流状态的手段。

287

表 8-2 IO 标准库的条件状态	
<code>strm::iostate</code>	机器相关的整型名,由各个 <code>iostream</code> 类定义,用于定义条件状态
<code>strm::badbit</code>	<code>strm::iostate</code> 类型的值,用于指出被破坏的流
<code>strm::failbit</code>	<code>strm::iostate</code> 类型的值,用于指出失败的 IO 操作
<code>strm::eofbit</code>	<code>strm::iostate</code> 类型的值,用于指出流已经到达文件结束符
<code>s.eof()</code>	如果设置了流 <code>s</code> 的 <code>eofbit</code> 值,则该函数返回 <code>true</code>
<code>s.fail()</code>	如果设置了流 <code>s</code> 的 <code>failbit</code> 值,则该函数返回 <code>true</code>
<code>s.bad()</code>	如果设置了流 <code>s</code> 的 <code>badbit</code> 值,则该函数返回 <code>true</code>
<code>s.good()</code>	如果流 <code>s</code> 处于有效状态,则该函数返回 <code>true</code>
<code>s.clear()</code>	将流 <code>s</code> 中的所有状态值都重设为有效状态
<code>s.clear(flag)</code>	将流 <code>s</code> 中的某个指定条件状态设置为有效。 <code>flag</code> 的类型是 <code>strm::iostate</code>
<code>s.setstate(flag)</code>	给流 <code>s</code> 添加指定条件。 <code>flag</code> 的类型是 <code>strm::iostate</code>
<code>s.rdstate()</code>	返回流 <code>s</code> 的当前条件,返回值类型为 <code>strm::iostate</code>

考虑下面 IO 错误的例子:

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

如果在标准输入设备输入 `Borges`,则 `cin` 在尝试将输入的字符串读为 `int` 型数据失败后,会生成一个错误状态。类似地,如果输入文件结束符 (end-of-file),`cin` 也会进入错误状态。而如果输入 `1024`,则成功读取,`cin` 将处于正确的无错误状态。

流必须处于无错误状态,才能用于输入或输出。检测流是否可用的最简单的方法是检查其真值:

```
if (cin)
    // ok to use cin, it is in a valid state

while (cin >> word)
    // ok: read operation successful ...
```

`if` 语句直接检查流的状态,而 `while` 语句则检测条件表达式返回的流,从而间接地检查了流的状态。如果成功输入,则条件检测为 `true`。

1. 条件状态

许多程序只需知道流是否有效。而某些程序则需要更详细地访问或控制流的状态,此时,除

288

了知道流处于错误状态外，还必须了解它遇到了哪种类型的错误。例如，程序员也许希望弄清是到达了文件的结尾，还是遇到了 IO 设备上的错误。

所有流对象都包含一个条件状态成员，该成员由 `setstate` 和 `clear` 操作管理。这个状态成员为 `iostate` 类型，这是由各个 `istream` 类分别定义的机器相关的整型。该状态成员以二进制位 (bit) 的形式使用，类似于 5.3.1 节的例子中用于记录测验成绩的 `int_quiz1` 变量。

每个 IO 类还定义了三个 `iostate` 类型的常量值，分别表示特定的位模式。这些常量值用于指出特定类型的 IO 条件，可与位操作符 (5.3 节) 一起使用，以便在一次操作中检查或设置多个标志。

`badbit` 标志着系统级的故障，如无法恢复的读写错误。如果出现了这类错误，则该流通常就不能再继续使用。如果出现的是可恢复的错误，如在希望获得数值型数据时输入了字符，此时则设置 `failbit` 标志，这种导致设置 `failbit` 的问题通常是可以修正的。`eofbit` 是在遇到文件结束符时设置的，此时同时还设置了 `failbit`。

流的状态由 `bad`、`fail`、`eof` 和 `good` 操作揭示。如果 `bad`、`fail` 或者 `eof` 中的任意一个为 `true`，则检查流本身将显示该流处于错误状态。类似地，如果这三个条件没有一个为 `true`，则 `good` 操作将返回 `true`。

`clear` 和 `setstate` 操作用于改变条件成员的状态。`clear` 操作将条件重设为有效状态。在流的使用出现了问题并做出补救后，如果我们希望把流重设为有效状态，则可以调用 `clear` 操作。使用 `setstate` 操作可打开某个指定的条件，用于表示某个问题的发生。除了添加的标记状态，`setstate` 将保留其他已存在的状态变量不变。

2. 流状态的查询和控制

可以如下管理输入操作：

```
int ival;
// read cin and test only for EOF; loop is executed even if there are other IO failures
while (cin >> ival, !cin.eof()) {
    if (cin.bad()) // input stream is corrupted; bail out
        throw runtime_error("IO stream corrupted");
    if (cin.fail()) { // bad input
        cerr<< "bad data, try again"; // warn the user
        cin.clear(istream::failbit); // reset the stream
        continue; // get next input
    }
    // ok to process ival
}
```

这个循环不断读入 `cin`，直到到达文件结束符或者发生不可恢复的读取错误为止。循环条件使用了逗号操作符 (5.9 节)。回顾逗号操作符的求解过程：首先计算它的每一个操作数，然后返回最右边操作数作为整个操作的结果。因此，循环条件只读入 `cin` 而忽略了其结果。该条件的结果是 `!cin.eof()` 的值。如果 `cin` 到达文件结束符，条件则为假，退出循环。如果 `cin` 没有到达文件结束符，则不管在读取时是否发生了其他可能遇到的错误，都进入循环。

在循环中，首先检查流是否已破坏。如果是的话，抛出异常 (6.13 节) 并退出循环。如果输入无效，则输出警告并清除 `failbit` 状态。在本例中，执行 `continue` 语句 (6.11 节) 回到 `while`

的开头，读入另一个值给 `ival`。如果没有出现任何错误，那么循环体中余下的部分则可以很安全地使用 `ival`。

3. 条件状态的访问

`rdstate` 成员函数返回一个 `iostate` 类型的值，该值对应于流当前的整个条件状态：

```
// remember current state of cin
istream::iostate old_state = cin.rdstate();
cin.clear();
process_input(); // use cin
cin.clear(old_state); // now reset cin to old state
```

4. 多种状态的处理

常常会出现需要设置或清除多个状态二进制位的情况。此时，可以通过多次调用 `setstate` 或者 `clear` 函数实现。另外一种方法则是使用按位或（OR）操作符（5.3 节）在一次调用中生成“传递两个或更多状态位”的值。按位或操作使用其操作数的二进制位模式产生一个整型数值。对于结果中的每一个二进制位，如果其值为 1，则该操作的两个操作数中至少有一个的对应二进制位是 1。例如：

```
// sets both the badbit and the failbit
is.setstate(istream::badbit | istream::failbit);
```

将对象 `is` 的 `failbit` 和 `badbit` 位同时打开。实参：

```
is.badbit | is.failbit
```

生成了一个值，其对应于 `badbit` 和 `failbit` 的位都打开了，也就是将这两个位都设置为 1，该值的其他位则都为 0。在调用 `setstate` 时，使用这个值来开启流条件状态成员中对应的 `badbit` 和 `failbit` 位。

习题

习题 8.3 编写一个函数，其唯一的形参和返回值都是 `istream&` 类型。该个函数应一直读取流直到到达文件结束符为止，还应将读到的内容输出到标准输出中。最后，重设流使其有效，并返回该流。

习题 8.4 通过以 `cin` 为实参实现调用来测试上题编写的函数。

习题 8.5 导致下面的 `while` 终止的原因是什么？

```
while (cin >> i) /* . . . */
```

8.3 输出缓冲区的管理

每个 IO 对象管理一个缓冲区，用于存储程序读写的数据。如有下面语句：

```
os << "please enter a value: ";
```

系统将字符串字面值存储在与流 `os` 关联的缓冲区中。下面几种情况将导致缓冲区的内容被刷新，即写入到真实的输出设备或者文件：

- (1) 程序正常结束。作为 `main` 返回工作的一部分，将清空所有输出缓冲区。
- (2) 在一些不确定的时候，缓冲区可能已经满了，在这种情况下，缓冲区将会在写下一个值

之前刷新。

(3) 用操纵符 (manipulator, 1.2.2 节) 显式地刷新缓冲区, 例如行结束符 `endl`。

(4) 在每次输出操作执行完后, 用 `unitbuf` 操纵符设置流的内部状态, 从而清空缓冲区。

(5) 可将输出流与输入流关联 (tie) 起来。在这种情况下, 在读输入流时将刷新其关联的输出缓冲区。

1. 输出缓冲区的刷新

我们的程序已经使用过 `endl` 操纵符, 用于输出一个换行符并刷新缓冲区。除此之外, C++ 语言还提供了另外两个类似的操纵符。第一个是经常使用的 `flush`, 用于刷新流, 但不在输出中添加任何字符。第二个则比较少用的 `ends`, 这个操纵符在缓冲区中插入空字符 `null`, 然后刷新它:

```
cout << "hi!" << flush;    // flushes the buffer; adds no data
cout << "hi!" << ends;      // inserts a null, then flushes the buffer
cout << "hi!" << endl;      // inserts a newline, then flushes the buffer
```

2. `unitbuf` 操纵符

如果需要刷新所有输出, 最好使用 `unitbuf` 操纵符。这个操纵符在每次执行完写操作后都刷新流:

```
cout << unitbuf << "first" << " second" << nunitbuf;
```

等价于:

```
cout << "first" << flush << " second" << flush;
```

`nunitbuf` 操纵符将流恢复为使用正常的、由系统管理的缓冲区刷新方式。

291

警告: 如果程序崩溃了, 则不会刷新缓冲区

如果程序不正常结束, 输出缓冲区将不会刷新。在尝试调试已崩溃的程序时, 通常会根据最后的输出找出程序发生错误的区域。如果崩溃出现在某个特定的输出语句后面, 则可知是在程序的这个位置之后出错。

调试程序时, 必须保证期待写入的每个输出都确实被刷新了。因为系统不会在程序崩溃时自动刷新缓冲区, 这就可能出现这样的情况: 程序做了写输出的工作, 但写的内容并没有显示在标准输出上, 仍然存储在输出缓冲区中等待输出。

如果需要使用最后的输出给程序错误定位, 则必须确定所有要输出的都已经输出。为了确保用户看到程序实际上处理的所有输出, 最好的方法是保证所有的输出操作都显式地调用了 `flush` 或 `endl`。

如果仅因为缓冲区没有刷新, 程序员将浪费大量的时间跟踪调试并没有执行的代码。基于这个原因, 输出时应多使用 `endl` 而非 `'\n'`。使用 `endl` 则不必担心程序崩溃时输出是否悬而未决 (即还留在缓冲区, 未输出到设备中)。

3. 将输入和输出绑在一起

当输入流与输出流绑在一起时, 任何读输入流的尝试都将首先刷新其输出流关联的缓冲区。标准库将 `cout` 与 `cin` 绑在一起, 因此语句:

```
cin >> ival;
```

导致 cout 关联的缓冲区被刷新。



交互式系统通常应确保它们的输入和输出流是绑在一起的。这样做意味着可以保证任何输出，包括给用户的提示，都在试图读之前输出。

tie 函数可用 istream 或 ostream 对象调用，使用一个指向 ostream 对象的指针形参。调用 tie 函数时，将实参流绑在调用该函数的对象上。如果一个流调用 tie 函数将其本身绑在传递给 tie 的 ostream 实参对象上，则该流上的任何 IO 操作都会刷新实参所关联的缓冲区。

```
cin.tie(&cout);    // illustration only: the library ties cin and cout for us
ostream *old_tie = cin.tie();
cin.tie(0); // break tie to cout, cout no longer flushed when cin is read
cin.tie(&cerr);    // ties cin and cerr, not necessarily a good idea!
// ...
cin.tie(0);        // break tie between cin and cerr
cin.tie(old_tie);  // reestablish normal tie between cin and cout
```

一个 ostream 对象每次只能与一个 istream 对象绑在一起。如果在调用 tie 函数时传递实参 0，则打破该流上已存在的捆绑。

292

8.4 文件的输入和输出

fstream 头文件定义了三种支持文件 IO 的类型：

- (1) ifstream，由 istream 派生而来，提供读文件的功能。
- (2) ofstream，由 ostream 派生而来，提供写文件的功能。
- (3) fstream，由 istream 派生而来，提供读写同一个文件的功能。

这些类型都由相应的 istream 类型派生而来，这个事实意味着我们已经知道使用 fstream 类型需要了解的大部分内容了。特别是，可使用 IO 操作符（<<和>>）在文件上实现格式化的 IO，而在前面章节介绍的条件状态也同样适用于 fstream 对象。

fstream 类型除了继承下来的行为外，还定义了两个自己的新操作——open 和 close，以及形参为要打开的文件名的构造函数。fstream、ifstream 或 ofstream 对象可调用这些操作，而其他的 IO 类型则不能调用。

8.4.1 文件流对象的使用

迄今为止，我们的程序已经使用过标准库定义的对象：cin、cout 和 cerr。需要读写文件时，则必须定义自己的对象，并将它们绑定在需要的文件上。假设 ifile 和 ofile 是存储希望读写的文件名的 string 对象，可如下编写代码：

```
// construct an ifstream and bind it to the file named ifile
ifstream infile(ifile.c_str());
// ofstream output file object to write file named ofile
ofstream outfile(ofile.c_str());
```

上述代码定义并打开了一对 `fstream` 对象。`infile` 是读的流，而 `outfile` 则是写的流。为 `ifstream` 或者 `ofstream` 对象提供文件名作为初始化式，就相当于打开了特定的文件。

```
ifstream infile;    // unbound input file stream
ofstream outfile;   // unbound output file stream
```

上述语句将 `infile` 定义为读文件的流对象，将 `outfile` 定义为写文件的对象。这两个对象都没有捆绑具体的文件。在使用 `fstream` 对象之前，还必须使这些对象捆绑要读写的文件：

```
infile.open("in");    // open file named "in" in the current directory
outfile.open("out");  // open file named "out" in the current directory
```

调用 `open` 成员函数将已存在的 `fstream` 对象与特定文件绑定。为了实现读写，需要将指定的文件打开并定位，`open` 函数完成系统指定的所有需要的操作。

293

警告：C++中的文件名

由于历史原因，IO 标准库使用 C 风格字符串（4.3 节）而不是 C++ `string` 类型的字符串作为文件名。在创建 `fstream` 对象时，如果调用 `open` 或使用文件名作初始化式，需要传递的实参应为 C 风格字符串，而不是标准库 `string` 对象。程序常常从标准输入获得文件名。通常，比较好的方法是将文件名读入 `string` 对象，而不是 C 风格字符数组。假设要使用的文件名保存在 `string` 对象中，则可调用 `c_str` 成员（4.3.2 节）获取 C 风格字符串。

1. 检查文件打开是否成功

打开文件后，通常要检验打开是否成功，这是一个好习惯：

```
// check that the open succeeded
if (!infile) {
    cerr << "error: unable to open input file: "
        << infile << endl;
    return -1;
}
```

这个条件与之前测试 `cin` 是否到达文件尾或遇到某些其他错误的条件类似。检查流等效于检查对象是否“适合”输入或输出。如果打开（`open`）失败，则说明 `fstream` 对象还没有为 IO 做好准备。当测试对象

```
if (outfile) // ok to use outfile?
```

返回 `true` 意味着文件已经可以使用。由于希望知道文件是否未准备好，则对返回值取反来检查流：

```
if (!outfile) // not ok to use outfile?
```

2. 将文件流与新文件重新捆绑

`fstream` 对象一旦打开，就保持与指定的文件相关联。如果要把 `fstream` 对象与另一个不同的文件关联，则必须先关闭（`close`）现在的文件，然后打开（`open`）另一个文件：

```
ifstream infile("in");    // opens file named "in" for reading
infile.close();            // closes "in"
infile.open("next");       // opens file named "next" for reading
```

要点是在尝试打开新文件之前，必须先关闭当前的文件流。open 函数会检查流是否已经打开。如果已经打开，则设置内部状态，以指出发生了错误。接下来使用文件流的任何尝试都会失败。

3. 清除文件流的状态

考虑这样的程序，它有一个 vector 对象，包含一些要打开并读取的文件名，程序要对每个文件中存储的单词做一些处理。假设该 vector 对象命名为 files，程序也许会有如下循环：

294

```
// for each file in the vector
while (it != files.end()) {
    ifstream input(it->c_str()); // open the file;
    // if the file is ok, read and "process" the input
    if (!input)
        break; // error: bail out!
    while(input >> s) // do the work on this file
        process(s);
    ++it; // increment iterator to get next file
}
```

每一次循环都构造了名为 input 的 ifstream 对象，打开并读取指定的文件。构造函数的初始化式使用了箭头操作符 (5.6 节) 对 it 进行解引用，从而获取 it 当前表示的 string 对象的 c_str 成员。文件由构造函数打开，并假设打开成功，读取文件直到到达文件结束符或者出现其他的错误条件为止。在这个点上，input 处于错误状态。任何读 input 的尝试都会失败。因为 input 是 while 循环的局部变量，在每次迭代中创建。这就意味着它在每次循环中都以干净的状态即 input.good() 为 true，开始使用。

如果希望避免在每次 while 循环过程中创建新的流对象，可将 input 的定义移到 while 之前。这点小小的改动意味着必须更仔细地管理流的状态。如果遇到文件结束符或其他错误，将设置流的内部状态，以便之后不允许再对该流做读写操作。关闭流并不能改变流对象的内部状态。如果最后的读写操作失败了，对象的状态将保持为错误模式，直到执行 clear 操作重新恢复流的状态为止。调用 clear 后，就像重新创建了该对象一样。

如果打算重用已存在的流对象，那么 while 循环必须在每次循环时记得关闭 (close) 和清空 (clear) 文件流：

```
ifstream input;
vector<string>::const_iterator it = files.begin();
// for each file in the vector
while (it != files.end()) {
    input.open(it->c_str()); // open the file
    // if the file is ok, read and "process" the input
    if (!input)
        break; // error: bail out!
    while(input >> s) // do the work on this file
        process(s);
    input.close(); // close file when we're done with it
    input.clear(); // reset state to ok
    ++it; // increment iterator to get next file
}
```

如果忽略 clear 的调用，则循环只能读入第一个文件。要了解其原因，就需要考虑在循环中发生了什么：首先打开指定的文件。假设打开成功，则读取文件直到文件结束或者出现其他错误条件

- 295 为止。在这个点上，input 处于错误状态。如果在关闭 (close) 该流前没有调用 clear 清除流的状态，接着在 input 上做的任何输入运算都会失败。一旦关闭该文件，再打开下一个文件时，在内层 while 循环上读 input 仍然会失败——毕竟最后一次对流的读操作到达了文件结束符，事实上该文件结束符对应的是另一个与本文件无关的其他文件。



如果程序员需要重用文件流读写多个文件，必须在读另一个文件之前调用 clear 清除该流的状态。

习题

- 习题 8.6 由于 ifstream 继承了 istream，因此可将 ifstream 对象传递给形参为 istream 引用的函数。使用 8.2 节第一个习题编写的函数读取已命名的文件。
- 习题 8.7 本节编写的两个程序，在打开 vector 容器中存放的任何文件失败时，使用 break 跳出 while 循环。重写这两个循环，如果文件无法打开，则输出警告信息，然后从 vector 中获取下一个文件名继续处理。
- 习题 8.8 上一个习题的程序可以不用 continue 语句实现。分别使用或不使用 continue 语句编写该程序。
- 习题 8.9 编写函数打开文件用于输入，将文件内容读入 string 类型的 vector 容器，每一行存储为该容器对象的一个元素。
- 习题 8.10 重写上面的程序，把文件中的每个单词存储为容器的一个元素。

8.4.2 文件模式

在打开文件时，无论是调用 open 还是以文件名作为流初始化的一部分，都需指定文件模式 (file mode)。每个 fstream 类都定义了一组表示不同模式的值，用于指定流打开的不同模式。与条件状态标志一样，文件模式也是整型常量，在打开指定文件时，可用位操作符 (5.3 节) 设置一个或多个模式。文件流构造函数和 open 函数都提供了默认实参 (7.4.1 节) 设置文件模式。默认值因流类型的不同而不同。此外，还可以显式地以模式打开文件。表 8-3 列出了文件模式及其含义。

表 8-3 文件模式

表 8-3 文件模式	
in	打开文件做读操作
out	打开文件做写操作
app	在每次写之前找到文件尾
ate	打开文件后立即将文件定位在文件尾
trunc	打开文件时清空已存在的文件流
binary	以二进制模式进行 IO 操作

out、trunc 和 app 模式只能用于指定与 ofstream 或 fstream 对象关联的文件；in 模式只能用于指定与 ifstream 或 fstream 对象关联的文件。所有的文件都可以用 ate 或 binary 模式

打开。ate 模式只在打开时有效：文件打开后将定位在文件尾。以 binary 模式打开的流则将文件以字节序列的形式处理，而不解释流中的字符。

296

默认时，与 ifstream 流对象关联的文件将以 in 模式打开，该模式允许文件做读的操作；与 ofstream 关联的文件则以 out 模式打开，使文件可写。以 out 模式打开的文件会被清空：丢弃该文件存储的所有数据。



从效果来看，为 ofstream 对象指定 out 模式等效于同时指定了 out 和 trunc 模式。

对于用 ofstream 打开的文件，要保存文件中已存在的数据，唯一方法是显式地指定 app 模式打开：

```
// output mode by default; truncates file named "file1"
ofstream outfile("file1");
// equivalent effect: "file1" is explicitly truncated
ofstream outfile2("file1", ofstream::out | ofstream::trunc);
// append mode; adds new data at end of existing file named "file2"
ofstream appfile("file2", ofstream::app);
```

outfile2 的定义使用了按位或操作符（5.3 节）将相应的文件同时以 out 和 trunc 模式打开。

1. 对同一个文件作输入和输出运算

fstream 对象既可以读也可以写它所关联的文件。fstream 如何使用它的文件取决于打开文件时指定的模式。

默认情况下，fstream 对象以 in 和 out 模式同时打开。当文件同时以 in 和 out 打开时不清空。如果打开 fstream 所关联的文件时，只使用 out 模式，而不指定 in 模式，则文件会清空已存在的数据。如果打开文件时指定了 trunc 模式，则无论是否同时指定了 in 模式，文件同样会被清空。下面的定义将 copyOut 文件同时以输入和输出的模式打开：

```
// open for input and output
fstream inOut("copyOut", fstream::in | fstream::out);
```

对于同时以输入和输出的模式打开的文件，附录 A.3.8 将讨论其使用方法。

297

2. 模式是文件的属性而不是流的属性

每次打开文件时都会设置模式：

```
ofstream outfile;
// output mode set to out, "scratchpad" truncated
outfile.open("scratchpad", ofstream::out);
outfile.close(); // close outfile so we can rebind it
// appends to file named "precious"
outfile.open("precious", ofstream::app);
outfile.close();
// output mode set by default, "out" truncated
outfile.open("out");
```

第一次调用 open 函数时，指定的模式是 ofstream::out。当前目录中名为“scratchpad”的文件以输出模式打开并清空。而名为“precious”的文件，则要求以添加模式打开：保存文件里的原

有数据，所有的新内容在文件尾部写入。在打开“out”文件时，没有明确指明输出模式，该文件则以 out 模式打开，这意味着当前存储在“out”文件中的任何数据都将被丢弃。



只要调用 open 函数，就要设置文件模式，其模式的设置可以是显式的也可以是隐式的。如果没有指定文件模式，将使用默认值。

3. 打开模式的有效组合

并不是所有的打开模式都可以同时指定。有些模式组合是没有意义的，例如同时以 in 和 trunc 模式打开文件，准备读取所生成的流，但却因为 trunc 操作而导致无数据可读。表 8-4 列出了有效的模式组合及其含义。

表 8-4 文件模式的组合	
out	打开文件做写操作，删除文件中已有的数据
out app	打开文件做写操作，在文件尾写入
out trunc	与 out 模式相同
in	打开文件做读操作
in out	打开文件做读、写操作，并定位于文件开头处
in out trunc	打开文件做读、写操作，删除文件中已有的数据

上述所有的打开模式组合还可以添加 ate 模式。对这些模式添加 ate 只会改变文件打开时的初始定位，在第一次读或写之前，将文件定位于文件末尾处。

298

8.4.3 一个打开并检查输入文件的程序

本书有好几个程序都要打开给定文件用于输入。由于需要在多个程序里做这件工作，我们编写一个名为 open_file 的函数实现这个功能。这个函数有两个引用形参，分别是 ifstream 和 string 类型，其中 string 类型的引用形参存储与指定 ifstream 对象关联的文件名：

```
// opens in binding it to the given file
ifstream& open_file(ifstream &in, const string &file)
{
    in.close();    // close in case it was already open
    in.clear();    // clear any existing errors
    // if the open fails, the stream will be in an invalid state
    in.open(file.c_str()); // open the file we were given
    return in; // condition state is good if open succeeded
}
```

由于不清楚流 in 的当前状态，因此首先调用 close 和 clear 将这个流设置为有效状态。然后尝试打开给定的文件。如果打开失败，流的条件状态将标志这个流是不可用的。最后返回流对象 in，此时，in 要么已经与指定文件绑定起来了，要么处于错误条件状态。

习题

习题 8.11 对于 open_file 函数，请解释为什么在调用 open 前先调用 clear 函数。如果忽略这

个函数调用，会出现什么问题？如果在 open 后面调用 clear 函数，又会怎样？

习题 8.12 对于 open_file 函数，请解释如果程序执行 close 函数失败，会产生什么结果？

习题 8.13 编写类似 open_file 的程序打开文件用于输出。

习题 8.14 使用 open_file 函数以及 8.2 节第一个习题编写的程序，打开给定的文件并读取其内容。

8.5 字符串流

iostream 标准库支持内存中的输入/输出，只要将流与存储在程序内存中的 string 对象捆绑起来即可。此时，可使用 iostream 输入和输出操作符读写这个 string 对象。标准库定义了三种类型的字符串流：

- istringstream，由 istream 派生而来，提供读 string 的功能。
- ostreamstream，由 ostream 派生而来，提供写 string 的功能。
- stringstream，由 iostream 派生而来，提供读写 string 的功能。

299

要使用上述类，必须包含 sstream 头文件。

与 fstream 类型一样，上述类型由 iostream 类型派生而来，这意味着 iostream 上所有的操作都适用于 stringstream 中的类型。stringstream 类型除了继承的操作外，还各自定义了一个有 string 形参的构造函数，这个构造函数将 string 类型的实参复制给 stringstream 对象。对 stringstream 的读写操作实际上读写的就是该对象中的 string 对象。这些类还定义了名为 str 的成员，用来读取或设置 stringstream 对象所操纵的 string 值。

注意到尽管 fstream 和 stringstream 共享相同的基类，但它们没有其他相互关系。特别是，stringstream 对象不能使用 open 和 close 函数，而 fstream 对象则不允许使用 str。

表 8-5 stringstream 特定的操作

stringstream strm;	创建自由的 stringstream 对象
stringstream strm(s);	创建存储 s 的副本的 stringstream 对象，其中 s 是 string 类型的对象
strm.str()	返回 strm 中存储的 string 类型对象
strm.str(s)	将 string 类型的 s 复制给 strm，返回 void

1. stringstream 对象的使用

前面已经见过以每次一个单词或每次一行的方式处理输入的程序。第一种程序用 string 输入操作符，而第二种则使用 getline 函数。然而，有些程序需要同时使用这两种方式：有些处理基于每行实现，而其他处理则要操纵每行中每个单词。可用 stringstream 对象实现：

```
string line, word;           // will hold a line and word from input, respectively
while (getline(cin, line)) { // read a line from the input into line
    // do per-line processing
    istringstream stream(line); // bind to stream to the line we read
    while (stream >> word){     // read a word from line
        // do per-word processing
    }
}
```

这里，使用 `getline` 函数从输入读取整行内容。然后为了获得每行中的单词，将一个 `istringstream` 对象与所读取的行绑定起来，这样只需使用普通的 `string` 输入操作符即可读出每行中的单词。

2. `stringstream` 提供的转换和/或格式化

`stringstream` 对象的一个常见用法是，需要在多种数据类型之间实现自动格式化时使用该类类型。例如，有一个数值型数据集，要获取它们的 `string` 表示形式，或反之。`sstream` 输入和输出操作可自动地把算术类型转化为相应的 `string` 表示形式，反过来也可以。

300

```
int val1 = 512, val2 = 1024;
ostringstream format_message;
// ok: converts values to a string representation
format_message << "val1: " << val1 << "\n"
               << "val2: " << val2 << "\n";
```

这里创建了一个名为 `format_message` 的 `ostringstream` 类型空对象，并将指定的内容插入该对象。重点在于 `int` 型值自动转换为等价的可打印的字符串。`format_message` 的内容是以下字符：

```
val1: 512\nval2: 1024
```

相反，用 `istringstream` 读 `string` 对象，即可重新将数值型数据找回来。读取 `istringstream` 对象自动地将数值型数据的字符表示方式转换为相应的算术值。

```
// str member obtains the string associated with a stringstream
istringstream input_istring(format_message.str());
string dump; // place to dump the labels from the formatted message
// extracts the stored ascii values, converting back to arithmetic types
input_istring >> dump >> val1 >> dump >> val2;
cout << val1 << " " << val2 << endl; // prints 512 1024
```

这里使用 `str` 成员获取与之前创建的 `ostringstream` 对象关联的 `string` 副本。再将 `input_istring` 与 `string` 绑定起来。在读 `input_istring` 时，相应的值恢复为它们原来的数值型表示形式。



为了读取 `input_string`，必须把该 `string` 对象分解为若干个部分。我们要的是数值型数据；为了得到它们，必须读取（和忽略）处于所需数据周围的标号。

因为输入操作符读取的是有类型的值，因此读入的对象类型必须和由 `stringstream` 读入的值的类型一致。在本例中，`input_istring` 分成四个部分：`string` 类型的值 `val1`，接着是 512，然后是 `string` 类型的值 `val2`，最后是 1024。一般情况下，使用输入操作符读 `string` 时，空白符将会忽略。于是，在读与 `format_message` 关联的 `string` 时，忽略其中的换行符。

习题

习题 8.15 使用 8.2 节第一个习题编写的函数输出 `istringstream` 对象的内容。

习题 8.16 编写程序将文件中的每一行存储在 `vector<string>` 容器对象中，然后使用 `istringstream` 从 `vector` 里以每次读一个单词的形式读取所存储的行。

301

小结

C++使用标准库类处理输入和输出:

- `iostream` 类处理面向流的输入和输出。
- `fstream` 类处理已命名文件的 IO。
- `stringstream` 类处理内存中字符串的 IO。

所有的这些类都是通过继承相互关联的。输入类继承了 `istream`，而输出类则继承了 `ostream`。因此，可在 `istream` 对象上执行的操作同样适用于 `ifstream` 或 `istringstream` 对象。而继承 `ostream` 的输出类也是类似的。

所有 IO 对象都有一组条件状态，用来指示是否可以通过该对象进行 IO 操作。如果出现了错误（例如遇到文件结束符）对象的状态将标志无法再进行输入，直到修正了错误为止。标准库提供了一组函数设置和检查这些状态。

术语

base class (基类) 是其他类的父类。基类定义了派生类所继承的接口。

condition state (条件状态) 流类用于指示给定的流是否可用的标志以及相关函数。表 8-2 列出了流的状态以及获取和设置这些状态的函数。

derived class (派生类) 与父类共享接口的类。

file mode (文件模式) 由 `fstream` 类定义的标志，在打开文件和控制文件何时使用时指定。表 8-3 列出了所有的文件模式。

`fstream` 用来读或写已命名文件的流对象。除了普通的 `iostream` 操作外，`fstream` 类还定义了 `open` 和 `close` 成员。`open` 成员函数有一个表示打开文件名的 C 风格字符串参数和一个可选的打开模式参数。默认时，`ifstream` 对象以 `in` 模式打开，`ofstream` 对象以 `out` 模式打开，而 `fstream` 对象则同时以 `in` 和 `out` 模式打开。`close` 成员关闭流关联的文件，必

须在打开另一个文件前调用。

inheritance (继承) 有继承关系的类型共享相同的接口。派生类继承其基类的属性。第 15 章将介绍继承。

object-oriented library (面向对象标准库) 有继承关系的类的集合。一般来说，面向对象标准库的基类定义了接口，由继承这个基类的各个派生类共享。在 IO 标准库中，`istream` 和 `ostream` 类是 `fstream` 和 `sstream` 头文件中定义的类型基类。派生类的对象可当做基类对象使用。例如，可在 `ifstream` 对象上使用 `istream` 定义的操作。

`stringstream` 读写字符串的流对象。除了普通的 `iostream` 操作外，它还定义了名为 `str` 的重载成员。无实参地调用 `str` 将返回 `stringstream` 所关联的 `string` 值。用 `string` 对象做实参调用它则将该 `stringstream` 对象与实参副本相关联。

容器和算法

目录

第 9 章 顺序容器	263
第 10 章 关联容器	305
第 11 章 泛型算法	335

C++提供了使用抽象进行高效率编程的方式。标准库就是一个很好的例子：标准库定义了许多容器类以及一系列泛型算法，使程序员可以更简洁、抽象和有效地编写程序。这样可以让标准库操心那些繁琐的细节，特别是内存管理，我们的程序只需关注要解决的实际问题就行了。

第 3 章介绍了 `vector` 容器类型。我们将会在第 9 章进一步深入探讨 `vector` 和其他顺序容器类型，而且还会学习 `string` 类型提供的更多操作，这些容器类型都是由标准库定义的。我们可将 `string` 视为仅包含字符的特殊容器，`string` 类型提供大量（但并不是全部）的容器操作。

标准库还定义了几种关联容器。关联容器中的元素不是顺序排列，而是按键（`key`）排序的。关联容器共享了许多顺序容器提供的操作，此外，还定义了自己特殊的操作。我们将在第 10 章学习相关的内容。

第 11 章介绍了泛型算法，这些算法通常作用于容器或序列中某一范围的元素。算法库提供了各种各样经典算法的有效实现，像查找、排序及其他常见的算法任务。例如，

复制算法将一个序列中所有的元素复制到另一个序列中；查找算法则用于寻找一个指定元素，等等。泛型算法中，所谓“泛型 (generic)”指的是两个方面：这些算法可作用于各种不同的容器类型，而这些容器又可以容纳多种不同类型的元素。

为容器类型提供通用接口是设计库的目的。如果两种容器提供相似的操作，则为它们定义的这个操作应该完全相同。例如，所有容器都有返回容器内元素个数的操作，于是所有容器都将该操作命名为 `size`，并将 `size` 返回值的类型都指定为 `size_type` 类型。类似地，算法具有一致的接口。例如，大部分算法都作用在由一对迭代器指定的元素范围上。

容器提供的操作和算法是一致定义的，这使得学习标准库更容易：只需理解一个操作如何工作，就能将该操作应用于其他的容器。更重要的是，接口的一致性使程序变得更灵活。通常不需要重新编写代码，就可以将一段使用某种容器类型的程序修改为使用不同容器实现。正如我们所看到的，容器提供了不同的性能折衷方案，可以改变容器类型对优化系统性能来说颇有价值。

第 9 章

顺 序 容 器

目录

9.1 顺序容器的定义.....	264
9.2 迭代器和迭代器范围.....	268
9.3 顺序容器的操作.....	272
9.4 vector 容器的自增长.....	284
9.5 容器的选用.....	287
9.6 再谈 string 类型.....	289
9.7 容器适配器.....	300
小结.....	303
术语.....	303

第 3 章介绍了最常用的顺序容器：vector 类型。本章将对第 3 章的内容进行扩充和完善，继续讨论标准库提供的顺序容器类型。顺序容器内的元素按其位置存储和访问。除顺序容器外，标准库还定义了几种关联容器，其元素按键（key）排序。我们将在下一章讨论它们。

容器类共享公共的接口，这使标准库更容易学习，只要学会其中一种类型就能运用另一种类型。每种容器类型提供一组不同的时间和功能折衷方案。通常不需要修改代码，只需改变类型声明，用一种容器类型替代另一种容器类型，就可以优化程序的性能。

容器容纳特定类型对象的集合。我们已经使用过一种容器类型：标准库 `vector` 类型，这是一种顺序容器（sequential container）。它将单一类型元素聚集起来成为容器，然后根据位置来存储和访问这些元素，这就是顺序容器。顺序容器的元素排列次序与元素值无关，而是由元素添加到容器里的次序决定。

标准库定义了三种顺序容器类型：`vector`、`list` 和 `deque`（是双端队列“double-ended queue”的简写，发音为“deck”）。它们的差别在于访问元素的方式，以及添加或删除元素相关操作的运行代价。标准库还提供了三种容器适配器（adaptor）。实际上，适配器是根据原始的容器类型所提供的操作，通过定义新的操作接口，来适应基础的容器类型。顺序容器适配器包括 `stack`、`queue` 和 `priority_queue` 类型，见表 9-1。

表 9-1 顺序容器类型

表 9-1 顺序容器类型	
顺序容器	
<code>vector</code>	支持快速随机访问
<code>list</code>	支持快速插入/删除
<code>deque</code>	双端队列
顺序容器适配器	
<code>stack</code>	后进先出（LIFO）栈
<code>queue</code>	先进先出（FIFO）队列
<code>priority_queue</code>	有优先级管理的队列

容器只定义了少量操作。大多数额外操作则由算法库提供，我们将在第 11 章学习算法库。标准库为由容器类型定义的操作强加了公共的接口。这些容器类型的差别在于它们提供哪些操作，但是如果两个容器提供了相同的操作，则它们的接口（函数名字和参数个数）应该相同。容器类型的操作集合形成了以下层次结构：

- 一些操作适用于所有容器类型。
- 另外一些操作则只适用于顺序或关联容器类型。
- 还有一些操作只适用于顺序或关联容器类型的一个子集。

306

在本章的后续部分，我们将详细描述顺序容器类型和它们所提供的操作。

9.1 顺序容器的定义

在 3.3 节中，我们已经了解了一些使用顺序容器类型的知识。为了定义一个容器类型的对象，必须先包含相关的头文件，即下列头文件之一：

```
#include <vector>
#include <list>
#include <deque>
```

所有的容器都是类模板（3.3 节）。要定义某种特殊的容器，必须在容器名后加一对尖括号，尖括号里面提供容器中存放的元素类型：

```
vector<string>    svec;           // empty vector that can hold strings
```

```
list<int>      ilist;           // empty list that can hold ints
deque<Sales_item> items;       // empty deque that holds Sales_items
```

所有的容器类型都定义了默认构造函数，用于创建指定类型的空容器对象。默认构造函数不带参数。



为了使程序更清晰、简短，容器类型最常用的构造函数是默认构造函数。在大多数的程序中，使用默认构造函数能达到最佳运行时性能，并且使容器更容易使用。

9.1.1 容器元素的初始化

除了默认构造函数，容器类型还提供其他的构造函数，使程序员可以指定元素初值，见表 9-2。

307

表 9-2 容器构造函数	
<code>C<T> c;</code>	创建一个名为 <code>c</code> 的空容器。 <code>C</code> 是容器类型名，如 <code>vector</code> ， <code>T</code> 是元素类型，如 <code>int</code> 或 <code>string</code> 。 适用于所有容器
<code>C c(c2);</code>	创建容器 <code>c2</code> 的副本 <code>c</code> ； <code>c</code> 和 <code>c2</code> 必须具有相同的容器类型，并存放相同类型的元素。适用于所有容器
<code>C c(b, e);</code>	创建 <code>c</code> ，其元素是迭代器 <code>b</code> 和 <code>e</code> 标示的范围内元素的副本。适用于所有容器
<code>C c(n, t);</code>	用 <code>n</code> 个值为 <code>t</code> 的元素创建容器 <code>c</code> ，其中值 <code>t</code> 必须是容器类型 <code>C</code> 的元素类型的值，或者是可转换为该类型的值 只适用于顺序容器
<code>C c(n);</code>	创建有 <code>n</code> 个值初始化（3.3.1 节）（ <code>value-initialized</code> ）元素的容器 <code>c</code> 只适用于顺序容器

1. 将一个容器初始化为另一个容器的副本

当不使用默认构造函数，而是用其他构造函数初始化顺序容器时，必须指出该容器有多少个元素，并提供这些元素的初值。同时指定元素个数和初值的一个方法是将新创建的容器初始化为一个同类型的已存在容器的副本：

```
vector<int> ivec;
vector<int> ivec2(ivec);    // ok: ivec is vector<int>
list<int>    ilist(ivec);    // error: ivec is not list<int>
vector<double> dvec(ivec);  // error: ivec holds int not double
```



将一个容器复制给另一个容器时，类型必须匹配：容器类型和元素类型都必须相同。

2. 初始化为一段元素的副本

尽管不能直接将一种容器内的元素复制给另一种容器，但系统允许通过传递一对迭代器（3.4 节）间接实现该功能。使用迭代器时，不要求容器类型相同。容器内的元素类型也可以不相同，只要它们相互兼容，能够将要复制的元素转换为所构建的新容器的元素类型，即可实现复制。

迭代器标记了要复制的元素范围，这些元素用于初始化新容器的元素。迭代器标记出要复制

的第一个元素和最后一个元素。采用这种初始化形式可复制不能直接复制的容器。更重要的是，可以实现复制其他容器的一个子序列：

```
// initialize slist with copy of each element of svec
list<string> slist(svec.begin(), svec.end());
// find midpoint in the vector
vector<string>::iterator mid = svec.begin() + svec.size()/2;
// initialize front with first half of svec: The elements up to but not including *mid
deque<string> front(svec.begin(), mid);
// initialize back with second half of svec: The elements *mid through end of svec
deque<string> back(mid, svec.end());
```

回顾一下指针，我们知道指针就是迭代器，因此允许通过使用内置数组中的一对指针初始化容器也就不奇怪了：

```
char *words[] = {"stately", "plump", "buck", "mulligan"};
// calculate how many elements in words
size_t words_size = sizeof(words)/sizeof(char *);
// use entire array to initialize words2
list<string> words2(words, words + words_size);
```

308

这里，使用 `sizeof` (5.8 节) 计算数组的长度。将数组长度加到指向第一个元素的指针上就可以得到指向超出数组末端的下一位置的指针。通过指向第一个元素的指针 `words` 和指向数组中最后一个元素的下一位置的指针，实现了 `words2` 的初始化。其中第二个指针提供停止复制的条件，其所指向的位置上存放的元素并没有复制。

3. 分配和初始化指定数目的元素

创建顺序容器时，可显式指定容器大小和一个（可选的）元素初始化式。容器大小可以是常量或非常量表达式，元素初始化式则必须是可用于初始化其元素类型的对象的值：

```
const list<int>::size_type list_size = 64;
list<string> slist(list_size, "eh?"); // 64 strings, each is eh?
```

这段代码表示 `slist` 含有 64 个元素，每个元素都被初始化为 "eh?" 字符串。

创建容器时，除了指定元素个数，还可选择是否提供元素初始化式。我们也可以只指定容器大小：

```
list<int> ilist(list_size); // 64 elements, each initialized to 0
// svec has as many elements as the return value from get_word_count
extern unsigned get_word_count(const string &file_name);
vector<string> svec(get_word_count("Chimera"));
```

不提供元素初始化式时，标准库将为该容器实现值初始化 (3.3.1 节)。采用这种类型的初始化，元素类型必须是内置或复合类型，或者是提供了默认构造函数的类类型。如果元素类型没有默认构造函数，则必须显式指定其元素初始化式。



接受容器大小做形参的构造函数只适用于顺序容器，而关联容器不支持这种初始化。

习题

习题 9.1 解释下列初始化, 指出哪些是错误的, 为什么?

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
    "Fort Sumter", "Manassas", "Perryville",
    "Vicksburg", "Meridian", "Chancellorsville" };
(a) vector<string> svec(sa, sa+6);
(b) list<int> ilist(ia+4, ia+6);
(c) vector<int> ivec(ia, ia+8);
(d) list<string> slist (sa+6, sa);
```

习题 9.2 创建和初始化一个 vector 对象有 4 种方式, 为每种方式提供一个例子, 并解释每个例子生成的 vector 对象包含什么值。

习题 9.3 解释复制容器对象的构造函数和使用两个迭代器的构造函数之间的差别。

9.1.2 容器内元素的类型约束

C++语言中, 大多数类型都可用作容器的元素类型。容器元素类型必须满足以下两个约束:

- 元素类型必须支持赋值运算。
- 元素类型的对象必须可以复制。

此外, 关联容器的键类型还需满足其他的约束, 我们将在第 10 章介绍相关内容。

大多数类型满足上述最低限度的元素类型要求。除了引用类型外, 所有内置或复合类型都可用做元素类型。引用不支持一般意义的赋值运算, 因此没有元素是引用类型的容器。

除输入输出 (IO) 标准库类型 (以及 17.19 节介绍的 `auto_ptr` 类型) 之外, 所有其他标准库类型都是有效的容器元素类型。特别地, 容器本身也满足上述要求, 因此, 可以定义元素本身就是容器类型的容器。Sales_item 类型也满足上述要求。

IO 库类型不支持复制或赋值。因此, 不能创建存放 IO 类型对象的容器。

1. 容器操作的特殊要求

支持复制和赋值功能是容器元素类型的最低要求。此外, 一些容器操作对元素类型还有特殊要求。如果元素类型不支持这些特殊要求, 则相关的容器操作就不能执行: 我们可以定义该类型的容器, 但不能使用某些特定的操作。

其中一种需外加类型要求的容器操作是指定容器大小并提供单个初始化式的构造函数。如果容器存储类类型的对象, 那么只有当其元素类型提供默认构造函数时, 容器才能使用这种构造函数。尽管有一些类没有提供默认构造函数, 但大多数类类型都会有。例如, 假设类 Foo 没有默认构造函数, 但提供了需要一个 int 型形参的构造函数。现在, 考虑下面的声明:

```
vector<Foo> empty;           // ok: no need for element default constructor
vector<Foo> bad(10);         // error: no default constructor for Foo
vector<Foo> ok(10, 1);       // ok: each element initialized to 1
```

我们定义了一个存放 Foo 类型对象的空容器, 但是, 只有在同时指定每个元素的初始化式时, 才能使用给定容器大小的构造函数来创建同类型的容器对象。

309

310

在描述容器操作时，我们应该留意（如果有的话）每个操作对元素类型的约束。

2. 容器的容器

因为容器受容器元素类型的约束，所以可定义元素是容器类型的容器。例如，可以定义 vector 类型的容器 lines，其元素为 string 类型的 vector 对象：

```
// note spacing: use ">>" not ">>" when specifying a container element type
vector< vector<string> > lines; // vector of vectors
```

注意，在指定容器元素为容器类型时，必须如下使用空格：

```
vector< vector<string> > lines; // ok: space required between close >
vector< vector<string>> lines; // error: >> treated as shift operator
```



必须用空格隔开两个相邻的 > 符号，以示这是两个分开的符号，否则，系统会认为 >> 是单个符号，为右移操作符，并结果导致编译时错误。

习题

- 习题 9.4 定义一个 list 对象来存储 deque 对象里的元素，该 deque 对象存放 int 型元素。
- 习题 9.5 为什么我们不可以使用容器来存储 iostream 对象？
- 习题 9.6 假设有一个名为 Foo 的类，这个类没有定义默认构造函数，但提供了需要一个 int 型参数的构造函数，定义一个存放 Foo 的 list 对象，该对象有 10 个元素。

9.2 迭代器和迭代器范围

在整个标准库中，经常使用形参为一对迭代器的构造函数。在深入探讨容器操作之前，先来了解一下迭代器和迭代器范围。

3.4 节首次介绍了 vector 类型的迭代器。每种容器类型都提供若干共同工作的迭代器类型。与容器类型一样，所有迭代器具有相同的接口：如果某种迭代器支持某种操作，那么支持这种操作的其他迭代器也会以相同的方式支持这种操作。例如，所有容器迭代器都支持以解引用运算从容器中读入一个元素。类似地，容器都提供自增和自减操作符来支持从一个元素到下一个元素的访问。

311

表 9-3 列出了迭代器为所有标准库容器类型所提供的运算。

表 9-3 常用迭代器运算	
*iter	返回迭代器 iter 所指向的元素的引用
iter->mem	对 iter 进行解引用，获取指定元素中名为 mem 的成员。等效于 (*iter).mem
++iter	给 iter 加 1，使其指向容器里的下一个元素
iter++	
--iter	给 iter 减 1，使其指向容器里的前一个元素
iter--	
iter1 == iter2	比较两个迭代器是否相等（或不等）。当两个迭代器指向同一个容器中的同一个元素，
iter1 != iter2	或者当它们都指向同一个容器的超出末端的下一位置时，两个迭代器相等

vector 和 deque 容器的迭代器提供额外的运算

C++定义的容器类型中，只有 vector 和 deque 容器提供下面两种重要的运算集合：迭代器算术运算（3.4.1 节），以及使用除了==和!=之外的关系操作符来比较两个迭代器（==和!=这两种关系运算适用于所有容器）。表 9-4 总结了这些相关的操作符。

表 9-4 vector 和 deque 类型迭代器支持的操作	
iter + n	在迭代器上加（减）整数值 n，将产生指向容器中前面（后面）第 n 个元素的迭代器。
iter - n	新计算出来的迭代器必须指向容器中的元素或超出容器末端的下一位置
iter1 += iter2	这是迭代器加减法的复合赋值运算：将 iter1 加上或减去 iter2 的运算结果赋给 iter1
iter1 -= iter2	
iter1 - iter2	两个迭代器的减法，其运算结果加上右边的迭代器即得左边的迭代器。这两个迭代器必须指向同一个容器中的元素或超出容器末端的下一位置
	只适用于 vector 和 deque 容器
>, >=, <, <=	迭代器的关系操作符。当一个迭代器指向的元素在容器中位于另一个迭代器指向的元素之前，则前一个迭代器小于后一个迭代器。关系操作符的两个迭代器必须指向同一个容器中的元素或超出容器末端的下一位置
	只适用于 vector 和 deque 容器

关系操作符只适用于 vector 和 deque 容器，这是因为只有这两种容器为其元素提供快速、随机的访问。它们确保可根据元素位置直接有效地访问指定的容器元素。这两种容器都支持通过元素位置实现的随机访问，因此它们的迭代器可以有效地实现算术和关系运算。

例如，下面的语句用于计算 vector 对象的中点位置：

```
vector<int>::iterator iter = vec.begin() + vec.size()/2;
```

另一方面，代码：

```
// copy elements from vec into ilist
list<int> ilist(vec.begin(), vec.end());
ilist.begin() + ilist.size()/2; // error: no addition on list iterators
```

是错误的。list 容器的迭代器既不支持算术运算（加法或减法），也不支持关系运算（<=, <, >=, >），它只提供前置和后置的自增、自减运算以及相等（不等）运算。

在第 11 章中，我们将会了解到迭代器提供的运算是使用标准库算法的基础。

习题

习题 9.7 下面的程序错在哪里？如何改正。

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* . . . */
```

习题 9.8 假设 vec_iter 与 vector 对象的一个元素捆绑在一起，该 vector 对象存放 string 类型的元素，请问下面的语句实现什么功能？

```
if (vec_iter->empty()) /* . . . */
```

习题 9.9 编写一个循环将 list 容器的元素逆序输出。

习题 9.10 下列迭代器的用法哪些（如果有的话）是错误的？

```
const vector< int > ivec(10);
vector< string >    svec(10);
list< int >         ilist(10);
```

- (a) vector<int>::iterator it = ivec.begin();
- (b) list<int>::iterator it = ilist.begin()+2;
- (c) vector<string>::iterator it = &svec[0];
- (d) for (vector<string>::iterator
it = svec.begin(); it != 0; ++it)
// ...

313

9.2.1 迭代器范围



迭代器范围这个概念是标准库的基础。

C++语言使用一对迭代器标记**迭代器范围**（iterator range），这两个迭代器分别指向同一个容器中的两个元素或超出末端的下一位置，通常将它们命名为 first 和 last，或 beg 和 end，用于标记容器中一段元素范围。

尽管 last 和 end 这两个名字很常见，但是它们却容易引起误解。其实第二个迭代器从来都不是指向元素范围的最后一个元素，而是指向最后一个元素的下一位置。该范围内的元素包括迭代器 first 指向的元素，以及从 first 开始一直到迭代器 last 指向的位置之前的所有元素。如果两个迭代器相等，则迭代器范围为空。

此类元素范围称为**左闭合区间**（left-inclusive interval），其标准表示方式为：

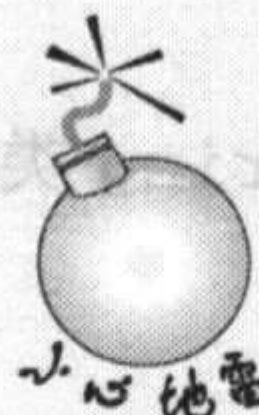
```
// to be read as: includes first and each element up to but not including last
[ first, last )
```

表示范围从 first 开始，到 last 结束，但不包括 last。迭代器 last 可以等于 first，或者指向 first 标记的元素后面的某个元素，但绝对不能指向 first 标记的元素前面的元素。

对形成迭代器范围的迭代器的要求

迭代器 first 和 last 如果满足以下条件，则可形成一个迭代器范围：

- 它们指向同一个容器中的元素或超出末端的下一位置。
- 如果这两个迭代器不相等，则对 first 反复做自增运算必须能够到达 last。换句话说，在容器中，last 绝对不能位于 first 之前。



编译器自己不能保证上述要求。编译器无法知道迭代器所关联的是哪个容器，也不知道容器内有多少个元素。若不能满足上述要求，将导致运行时未定义的行为。

使用左闭合区间的编程意义

因为左闭合区间有两个方便使用的性质，所以标准库使用此类区间。假设 `first` 和 `last` 标记了一个有效的迭代器范围，于是：

- (1) 当 `first` 与 `last` 相等时，迭代器范围为空；
 - (2) 当 `first` 与 `last` 不相等时，迭代器范围内至少有一个元素，而且 `first` 指向该区间中的第一个元素。此外，通过若干次自增运算可以使 `first` 的值不断增大，直到 `first==last` 为止。
- 这两个性质意味着程序员可以安全地编写如下的循环，通过测试迭代器处理一段元素：

```
while (first != last) {
    // safe to use *first because we know there is at least one element
    ++first;
}
```

假设 `first` 和 `last` 标记了一段有效的迭代器范围，于是我们知道要么 `first==last`，这是退出循环的情况；要么该区间非空，`first` 指向其第一个元素。因为 `while` 循环条件处理了空区间情况，所以对此无须再特别处理。当迭代器范围非空时，循环至少执行一次。由于循环体每次循环就给 `first` 加 1，因此循环必定会终止。而且在循环内可确保 `*first` 是安全的：它必然指向 `first` 和 `last` 之间非空区间内的某个特定元素。

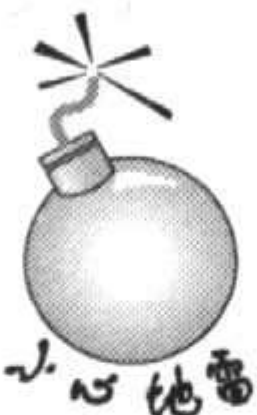
习题

- 习题 9.11 要标记出有效的迭代器范围，迭代器需满足什么约束？
- 习题 9.12 编写一个函数，其形参是一对迭代器和一个 `int` 型数值，实现在迭代器标记的范围内寻找该 `int` 型数值的功能，并返回一个 `bool` 结果，以指明是否找到指定数据。
- 习题 9.13 重写程序，查找元素的值，并返回指向找到的元素的迭代器。确保程序在要寻找的元素不存在时也能正确工作。
- 习题 9.14 使用迭代器编写程序，从标准输入设备读入若干 `string` 对象，并将它们存储在一个 `vector` 对象中，然后输出该 `vector` 对象中的所有元素。
- 习题 9.15 用 `list` 容器类型重写习题 9.14 得到的程序，列出改变了容器类型后要做的修改。

9.2.2 使迭代器失效的容器操作

在后面的几节里，我们将看到一些容器操作会修改容器的内在状态或移动容器内的元素。这样的操作使所有指向被移动的元素迭代器失效，也可能同时使其他迭代器失效。使用无效迭代器是没有定义的，可能会导致与悬垂指针相同的问题。

例如，每种容器都定义了一个或多个 `erase` 函数。这些函数提供了删除容器元素的功能。任何指向已删除元素的迭代器都具有无效值，毕竟，该迭代器指向了容器中不再存在的元素。



使用迭代器编写程序时，必须留意哪些操作会使迭代器失效。使用无效迭代器将会导致严重的运行时错误。

无法检查迭代器是否有效，也无法通过测试来发现迭代器是否已经失效。任何无效迭代器的使用都可能导致运行时错误，但程序不一定会崩溃，否则检查这种错误也许会容易些。

提示

使用迭代器时，通常可以编写程序使得要求迭代器有效的代码范围相对较短。然后，在该范围内，严格检查每一条语句，判断是否有元素添加或删除，从而相应地调整迭代器的值。

9.3 顺序容器的操作

每种顺序容器都提供了一组有用的类型定义以及以下操作：

- 在容器中添加元素。
- 在容器中删除元素。
- 设置容器大小。
- （如果有的话）获取容器内的第一个和最后一个元素。

9.3.1 容器定义的类型别名

在前面的章节里，我们已经使用过三种由容器定义的类型：`size_type`、`iterator` 和 `const_iterator`。所有容器都提供这三种类型以及表 9-5 所列出的其他类型。

表 9-5 容器定义的类型别名	
<code>size_type</code>	无符号整型，足以存储此容器类型的最大可能容器长度
<code>iterator</code>	此容器类型的迭代器类型
<code>const_iterator</code>	元素的只读迭代器类型
<code>reverse_iterator</code>	按逆序寻址元素的迭代器
<code>const_reverse_iterator</code>	元素的只读（不能写）逆序迭代器
<code>difference_type</code>	足够存储两个迭代器差值的有符号整型，可为负数
<code>value_type</code>	元素类型
<code>reference</code>	元素的左值类型，是 <code>value_type&</code> 的同义词
<code>const_reference</code>	元素的常量左值类型，等效于 <code>const value_type&</code>

316

我们将在 11.3.3 节中详细介绍逆序迭代器。简单地说，逆序迭代器从后向前遍历容器，并反转了某些相关的迭代器操作：例如，在逆序迭代器上做++运算将指向容器中的前一个元素。

表 9-5 的最后三种类型使程序员无须直接知道容器元素的真正类型，就能使用它。需要使用元素类型时，只要用 `value_type` 即可。如果要引用该类型，则通过 `reference` 和 `const_reference` 类型实现。在程序员编写自己的泛型程序（第 16 章）时，这些元素相关类型的定义非常有用。

使用容器定义类型的表达式看上去非常复杂：

```
// iter is the iterator type defined by list<string>
```

```
list<string>::iterator iter;
// cnt is the difference_type type defined by vector<int>
vector<int>::difference_type cnt;
```

iter 的声明使用了作用域操作符，以表明此时所使用的符号::右边的类型名字是在符号::左边指定容器的作用域内定义的。其效果是将 iter 声明为 iterator 类型，而 iterator 是存放 string 类型元素的 list 类的成员。

习题

习题 9.16 int 型的 vector 容器应该使用什么类型的索引？

习题 9.17 读取存放 string 对象的 list 容器时，应该使用什么类型？

9.3.2 begin 和 end 成员

begin 和 end 操作产生指向容器内第一个元素和最后一个元素的下一位置的迭代器，如表 9-6 所示。这两个迭代器通常用于标记包含容器中所有元素的迭代器范围。

表 9-6 容器的 begin 和 end 操作

c.begin()	返回一个迭代器，它指向容器 c 的第一个元素
c.end()	返回一个迭代器，它指向容器 c 的最后一个元素的下一位置
c.rbegin()	返回一个逆序迭代器，它指向容器 c 的最后一个元素
c.rend()	返回一个逆序迭代器，它指向容器 c 的第一个元素前面的位置

上述每个操作都有两个不同版本：一个是 const 成员（7.7.1 节），另一个是非 const 成员。这些操作返回什么类型取决于容器是否为 const。如果容器不是 const，则这些操作返回 iterator 或 reverse_iterator 类型。如果容器是 const，则其返回类型要加上 const_前缀，也就是 const_iterator 和 const_reverse_iterator 类型。我们将在 11.3.3 节中详细介绍逆序迭代器。

317

9.3.3 在顺序容器中添加元素

3.3.2 节介绍了添加元素的一种方法：push_back。所有顺序容器都支持 push_back 操作（表 9-7），提供在容器尾部插入一个元素的功能。下面的循环每次读入一个 string 类型的值，并存放在 text_word 对象中：

```
// read from standard input putting each word onto the end of container
string text_word;
while (cin >> text_word)
    container.push_back(text_word);
```

调用 push_back 函数会在容器 container 尾部创建一个新元素，并使容器的长度加 1。新元素的值为 text_word 对象的副本，而 container 的类型则可能是 list、vector 或 deque。

表 9-7 在顺序容器中添加元素的操作

c.push_back(t)	在容器 c 的尾部添加值为 t 的元素。返回 void 类型
c.push_front(t)	在容器 c 的前端添加值为 t 的元素。返回 void 类型 只适用于 list 和 deque 容器类型
c.insert(p,t)	在迭代器 p 所指向的元素前面插入值为 t 的新元素。返回指向新添加元素的迭代器
c.insert(p,n,t)	在迭代器 p 所指向的元素前面插入 n 个值为 t 的新元素。返回 void 类型
c.insert(p,b,e)	在迭代器 p 所指向的元素前面插入由迭代器 b 和 e 标记的范围内的元素。返回 void 类型

除了 push_back 运算，list 和 deque 容器类型还提供了类似的操作：push_front。这个操作实现在容器首部插入新元素的功能。例如：

```
list<int> ilist;
// add elements at the end of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_back(ix);
```

使用 push_back 操作在容器 ilist 尾部依次添加元素 0、1、2、3。

然后，我们选择用 push_front 操作再次在 ilist 中添加元素：

```
// add elements to the start of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

此时，元素 0、1、2、3 则被依次添加在 ilist 的开始位置。由于每个元素都在 ilist 的新起点插入，因此它们在容器中以逆序排列，循环结束后，ilist 内的元素序列为：3、2、1、0、0、1、2、3。

关键概念：容器元素都是副本

在容器中添加元素时，系统是将元素值复制到容器里。类似地，使用一段元素初始化新容器时，新容器存放的是原始元素的副本。被复制的原始值与新容器中的元素各不相关，此后，容器内元素值发生变化时，被复制的原值不会受到影响，反之亦然。

318

1. 在容器中的指定位置添加元素

使用 push_back 和 push_front 操作可以非常方便地在顺序容器的尾部或首部添加单个元素。而 insert 操作则提供了一组更通用的插入方法，实现在容器的任意指定位置插入新元素。insert 操作有三个版本（表 9-7）。第一个版本需要一个迭代器和一个元素值参数，迭代器指向插入新元素的位置。下面的程序就是使用了这个版本的 insert 函数在容器首部插入新元素：

```
vector<string> svec;
list<string> slist;
string spouse("Beth");
// equivalent to calling slist.push_front (spouse);
slist.insert(slist.begin(), spouse);
// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector is an expensive operation
svec.insert(svec.begin(), spouse);
```

新元素是插入在迭代器指向的位置之前。迭代器可以指向容器的任意位置，包括超出末端的下一位置。由于迭代器可能指向超出容器末端的下一位置，这是一个不存在的元素，因此 `insert` 函数是在其指向位置之前而非其后插入元素。代码

```
slist.insert(iter, spouse); // insert spouse just before iter
```

就在 `iter` 指向的元素前面插入 `spouse` 的副本。

这个版本的 `insert` 函数返回指向新插入元素的迭代器。可使用该返回值在容器中的指定位置重复插入元素：

```
list<string> lst;
list<string>::iterator iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // same as calling push_front
```



要彻底地理解上述循环是如何执行的，这一点非常重要——特别是要明白我们为什么说上述循环等效于调用 `push_front` 函数。

319

循环前，将 `iter` 初始化为 `lst.begin()`。此时，由于该 `list` 对象是空的，因此 `lst.begin()` 与 `lst.end()` 相等，于是 `iter` 指向该（空）容器的超出末端的下一位置。第一次调用 `insert` 函数时，将刚读入的元素插入到 `iter` 所指向位置的前面，容器 `lst` 得到第一个也是唯一的元素。然后 `insert` 函数返回指向这个新元素的迭代器，并赋给 `iter`，接着重复 `while` 循环，读入下一个单词。只要有单词要插入，每次 `while` 循环都将新元素插入到 `iter` 前面，然后重置 `iter` 指向新插入元素。新插入的元素总是容器中的第一个元素，因此，每次迭代都将元素插入在该 `list` 对象的第一个元素前面。

2. 插入一段元素

`insert` 函数的第二个版本提供在指定位置插入指定数量的相同元素的功能：

```
svec.insert(svec.end(), 10, "Anna");
```

上述代码在容器 `svec` 的尾部插入 10 个元素，每个新元素都初始化为 "Anna"。

`insert` 函数的最后一个版本实现在容器中插入由一对迭代器标记的一段范围内的元素。例如，给出以下 `string` 类型的数组：

```
string sarray[4] = {"quasi", "simba", "frollo", "scar"};
```

可将该数组中所有的或其中一部分元素插入到 `string` 类型的 `list` 容器中：

```
// insert all the elements in sarray at end of slist
slist.insert(slist.end(), sarray, sarray+4);
list<string>::iterator slist_iter = slist.begin();
// insert last two elements of sarray before slist_iter
slist.insert(slist_iter, sarray+2, sarray+4);
```

3. 添加元素可能会使迭代器失效

正如我们在 9.4 节中了解的一样，在 `vector` 容器中添加元素可能会导致整个容器的重新加载，这样的话，该容器涉及的所有迭代器都会失效。即使需要重新加载整个容器，指向新插入元素后面的那个元素的迭代器也会失效。



320

任何 `insert` 或 `push` 操作都可能导致迭代器失效。当编写循环将元素插入到 `vector` 或 `deque` 容器中时，程序必须确保迭代器在每次循环后都得到更新。

4. 避免存储 `end` 操作返回的迭代器

在 `vector` 或 `deque` 容器中添加元素时，可能会导致某些或全部迭代器失效。假设所有迭代器失效是最安全的做法。这个建议特别适用于由 `end` 操作返回的迭代器。在容器的任何位置插入任何元素都会使该迭代器失效。

例如，考虑一个读取容器中每个元素的循环，对读出元素做完处理后，在原始元素后面插入一个新元素。我们希望该循环可以处理每个原始元素，然后使用 `insert` 函数插入新元素，并返回指向刚插入元素的迭代器。在每次插入操作完成后，给返回的迭代器自增 1，以使循环定位在下一个要处理的原始元素。如果我们尝试通过存储 `end` 操作返回的迭代器来“优化”该循环，将导致灾难性错误：

```
vector<int>::iterator first = v.begin(),
                      last = v.end(); // cache end iterator
// disaster: behavior of this loop is undefined
while (first != last) {
    // do some processing
    // insert new value and reassign first, which otherwise would be invalid
    first = v.insert(first, 42);
    ++first; // advance first just past the element we added
}
```

上述代码的行为未定义。在很多实现中，该段代码将导致死循环。问题在于这个程序将 `end` 操作返回的迭代器值存储在名为 `last` 的局部变量中。循环体中实现了元素的添加运算，添加元素会使得存储在 `last` 中的迭代器失效。该迭代器既没有指向容器 `v` 的元素，也不再指向 `v` 的超出末端的下一位置。



提示 不要存储 `end` 操作返回的迭代器。添加或删除 `deque` 或 `vector` 容器内的元素都会导致存储的迭代器失效。

为了避免存储 `end` 迭代器，可以在每次做完插入运算后重新计算 `end` 迭代器值：

```
// safer: recalculate end on each trip whenever the loop adds/erases elements
while (first != v.end()) {
    // do some processing
    first = v.insert(first, 42); // insert new value
    ++first; // advance first just past the element we added
}
```

习题

习题 9.18 编写程序将 `int` 型的 `list` 容器的所有元素复制到两个 `deque` 容器中。`list` 容器的元素如果为偶数，则复制到一个 `deque` 容器中；如果为奇数，则复制到另一个 `deque` 容器里。

习题 9.19 假设 `iv` 是一个 `int` 型的 `vector` 容器，下列程序存在什么错误？如何改正之。

```
vector<int>::iterator mid = iv.begin() + iv.size()/2;
while ( vector<int>::iterator iter != mid )
    if (iter == some_val)
        iv.insert(iter, 2 * some_val);
```

9.3.4 关系操作符

所有的容器类型都支持用关系操作符（5.2 节）来实现两个容器的比较。比较的容器必须具有相同的容器类型，而且其元素类型也必须相同。例如，vector<int>容器只能与 vector<int>容器比较，而不能与 list<int>或 vector<double>类型的容器比较。

321

容器的比较是基于容器内元素的比较。容器的比较使用了元素类型定义的同一个关系操作符：两个容器做 != 比较使用了其元素类型定义的 != 操作符。如果容器的元素类型不支持某种操作符，则该类容器就不能做这种比较运算。

下面的操作类似于 string 类型的关系运算（3.2.3 节）：

- 如果两个容器具有相同的长度而且所有元素都相等，那么这两个容器就相等；否则，它们就不相等。
- 如果两个容器的长度不相同，但较短的容器中所有元素都等于较长容器中对应的元素，则称较短的容器小于另一个容器。
- 如果两个容器都不是对方的初始子序列，则它们的比较结果取决于所比较的第一个不相等的元素。

理解上述操作的最简单方法是研究例程：

```
/*
    ivec1: 1 3 5 7 9 12
    ivec2: 0 2 4 6 8 10 12
    ivec3: 1 3 9
    ivec4: 1 3 5 7
    ivec5: 1 3 5 7 9 12
*/
// ivec1 and ivec2 differ at element[0]: ivec1 greater than ivec2
ivec1 < ivec2 // false
ivec2 < ivec1 // true
// ivec1 and ivec3 differ at element[2]: ivec1 less than ivec3
ivec1 < ivec3 // true
// all elements equal, but ivec4 has fewer elements, so ivec1 is greater than ivec4
ivec1 < ivec4 // false
ivec1 == ivec5 // true; each element equal and same number of elements
ivec1 == ivec4 // false; ivec4 has fewer elements than ivec1
ivec1 != ivec4 // true; ivec4 has fewer elements than ivec1
```

322

使用元素提供的关系操作符实现容器的关系运算



C++语言只允许两个容器做其元素类型定义的关系运算。

所有容器都通过比较其元素对来实现关系运算：

```
ivec1 < ivec2
```

假设 `ivec1` 和 `ivec2` 都是 `vector<int>` 类型的容器，则上述比较使用了内置 `int` 型定义的小于操作符。如果这两个 `vector` 容器存储的是 `string` 对象，则使用 `string` 类型的小于操作符。

如果上述 `vector` 容器存储 1.5 节定义的 `Sales_item` 类型的对象，则该比较运算不合法。因为 `Sales_item` 类型没有定义关系运算，所以不能比较存放 `Sales_item` 对象的容器：

```
vector<Sales_item> storeA;  
vector<Sales_item> storeB;  
if (storeA < storeB) // error: Sales_item has no less-than operator
```

习题

习题 9.20 编写程序判断一个 `vector<int>` 容器所包含的元素是否与一个 `list<int>` 容器的完全相同。

习题 9.21 假设 `c1` 和 `c2` 都是容器，下列用法给 `c1` 和 `c2` 的元素类型带来什么约束？

```
if (c1<c2)
```

（如果有的话）对 `c1` 和 `c2` 的约束又是什么？

9.3.5 容器大小的操作

所有容器类型都提供四种与容器大小相关的操作(表 9-8)。3.2.3 节已经使用了 `size` 和 `empty` 函数：`size` 操作返回容器内元素的个数；`empty` 操作则返回一个布尔值，当容器的大小为 0 时，返回值为 `true`，否则为 `false`。

表 9-8 顺序容器的大小操作

<code>c.size()</code>	返回容器 <code>c</code> 中的元素个数。返回类型为 <code>c::size_type</code>
<code>c.max_size()</code>	返回容器 <code>c</code> 可容纳的最多元素个数 返回类型为 <code>c::size_type</code>
<code>c.empty()</code>	返回标记容器大小是否为 0 的布尔值
<code>c.resize(n)</code>	调整容器 <code>c</code> 的长度大小，使其能容纳 <code>n</code> 个元素 如果 <code>n<c.size()</code> ，则删除多出来的元素；否则，添加采用值初始化的新元素
<code>c.resize(n,t)</code>	调整容器 <code>c</code> 的长度大小，使其能容纳 <code>n</code> 个元素。所有新添加的元素值都为 <code>t</code>

容器类型提供 `resize` 操作来改变容器所包含的元素个数。如果当前的容器长度大于新的长度值，则该容器后部的元素会被删除；如果当前的容器长度小于新的长度值，则系统会在该容器后部添加新元素：

```
list<int> ilist(10, 42); // 10 ints: each has value 42  
ilist.resize(15);       // adds 5 elements of value 0 to back of ilist  
ilist.resize(25, -1);   // adds 10 elements of value -1 to back of ilist  
ilist.resize(5);        // erases 20 elements from the back of ilist
```

323

`resize` 操作可带有一个可选的元素值形参。如果在调用该函数时提供了这个参数，则所有新添加的元素都初始化为这个值。如果没有这个参数，则新添加的元素采用值初始化（3.3.1 节）。



resize 操作可能会使迭代器失效。在 vector 或 deque 容器上做 resize 操作有可能会使其所有的迭代器都失效。

对于所有的容器类型，如果 resize 操作压缩了容器，则指向已删除的元素的迭代器失效。

习题

习题 9.22 已知容器 vec 存放了 25 个元素，那么 vec.resize(100) 操作实现了什么功能？若再做操作 vec.resize(10)，实现的又是什么功能？

习题 9.23 使用只带有一个长度参数的 resize 操作对元素类型有什么要求（如果有的话）？

9.3.6 访问元素

如果容器非空，那么容器类型的 front 和 back 成员（表 9-9）将返回容器内第一个或最后一个元素的引用：

```
// check that there are elements before dereferencing an iterator
// or calling front or back
if (!ilist.empty()) {
    // val and val2 refer to the same element
    list<int>::reference val = *ilist.begin();
    list<int>::reference val2 = ilist.front();

    // last and last2 refer to the same element
    list<int>::reference last = *--ilist.end();
    list<int>::reference last2 = ilist.back();
}
```

324

表 9-9 访问顺序容器内元素的操作

c.back()	返回容器 c 的最后一个元素的引用。如果 c 为空，则该操作未定义
c.front()	返回容器 c 的第一个元素的引用。如果 c 为空，则该操作未定义
c[n]	返回下标为 n 的元素的引用 如果 n<0 或 n>=c.size()，则该操作未定义 只适用于 vector 和 deque 容器
c.at(n)	返回下标为 n 的元素的引用。如果下标越界，则该操作未定义 只适用于 vector 和 deque 容器

这段程序使用了两种不同的方法获取时 ilist 中的第一个和最后一个元素的引用。直接的方法是调用 front 或 back 函数。间接的方法是，通过对 begin 操作返回的迭代器进行解引用，或对 end 操作返回的迭代器的前一个元素位置进行解引用，来获取对同一元素的引用。在这段程序中，有两个地方值得注意：end 迭代器指向容器的超出末端的下一位置，因此必须先对其减 1 才能获取最后一个元素；另一点是，在调用 front 或 back 函数之前，或者在对 begin 或 end 返回的迭代器进行解引用运算之前，必须保证 ilist 容器非空。如果该 list 容器为空，则 if 语句内所有的操作都没有定义。

3.3.2 节介绍了下标运算，我们注意到程序员必须保证在指定下标位置上的元素确实存在。下标操作符本身不会做相关的检查。使用 `front` 或 `back` 运算时，必须注意同样的问题。如果容器为空，那么这些操作将产生未定义的结果。如果容器内只有一个元素，则 `front` 和 `back` 操作都返回对该元素的引用。



使用越界的下标，或调用空容器的 `front` 或 `back` 函数，都会导致程序出现严重的错误。

使用下标运算的另一个可选方案是 `at` 成员函数（表 9-9）。这个函数的行为和下标运算相似，但是如果给出的下标无效，`at` 函数将会抛出 `out_of_range` 异常（6.13 节）：

```
vector<string> svec;           // empty vector
cout << svec[0];              // run-time error: There are no elements in svec!
cout << svec.at(0);           // throws out_of_range exception
```

习题

习题 9.24 编写程序获取 `vector` 容器的第一个元素。分别使用下标操作符、`front` 函数以及 `begin` 函数实现该功能，并提供空的 `vector` 容器测试你的程序。

325

9.3.7 删除元素

回顾前面的章节，我们知道容器类型提供了通用的 `insert` 操作在容器的任何位置插入元素，并支持特定的 `push_front` 和 `push_back` 操作在容器首部或尾部插入新元素。类似地，容器类型提供了通用的 `erase` 操作和特定的 `pop_front` 和 `pop_back` 操作来删除容器内的元素（表 9-10）。

表 9-10 删除顺序容器内元素的操作

<code>c.erase(p)</code>	删除迭代器 <code>p</code> 所指向的元素 返回一个迭代器，它指向被删除元素后面的元素。如果 <code>p</code> 指向容器内的最后一个元素，则返回的迭代器指向容器的超出末端的下一位置。如果 <code>p</code> 本身就是指向超出末端的下一位置的迭代器，则该函数未定义
<code>c.erase(b, e)</code>	删除迭代器 <code>b</code> 和 <code>e</code> 所标记的范围内所有的元素 返回一个迭代器，它指向被删除元素段后面的元素。如果 <code>e</code> 本身就是指向超出末端的下一位置的迭代器，则返回的迭代器也指向容器的超出末端的下一位置
<code>c.clear()</code>	删除容器 <code>c</code> 内的所有元素。返回 <code>void</code>
<code>c.pop_back()</code>	删除容器 <code>c</code> 的最后一个元素。返回 <code>void</code> 。如果 <code>c</code> 为空容器，则该函数未定义
<code>c.pop_front()</code>	删除容器 <code>c</code> 的第一个元素。返回 <code>void</code> 。如果 <code>c</code> 为空容器，则该函数未定义 只适用于 list 或 deque 容器

1. 删除第一个或最后一个元素

`pop_front` 和 `pop_back` 函数用于删除容器内的第一个和最后一个元素。但 `vector` 容器类型不支持 `pop_front` 操作。这些操作删除指定的元素并返回 `void`。

pop_front 操作通常与 front 操作配套使用，实现以栈的方式处理容器：

```
while (!ilist.empty()) {
    process(ilist.front()); // do something with the current top of ilist
    ilist.pop_front();      // done; remove first element
}
```

这个循环非常简单：使用 front 操作获取要处理的元素，然后调用 pop_front 函数从容器 list 中删除该元素。



pop_front 和 pop_back 函数的返回值并不是删除的元素值，而是 void。要获取删除的元素值，则必须在删除元素之前调用 front 或 back 函数。

2. 删除容器内的一个元素

删除一个或一段元素更通用的方法是 erase 操作。该操作有两个版本：删除由一个迭代器指向的单个元素，或删除由一对迭代器标记的一段元素。erase 的这两种形式都返回一个迭代器，它指向被删除元素或元素段后面的元素。也就是说，如果元素 j 恰好紧跟在元素 i 后面，则将元素 i 从容器中删除后，删除操作返回指向 j 的迭代器。

326



如同其他操作一样，erase 操作也不会检查它的参数。程序员必须确保用作参数的迭代器或迭代器范围是有效的。

通常，程序员必须在容器中找到要删除的元素后，才使用 erase 操作。寻找一个指定元素的最简单方法是使用标准库的 find 算法。我们将在 11.1 节中进一步讨论 find 算法。为了使用 find 函数或其他泛型算法，在编程时，必须将 algorithm 头文件包含进来。find 函数需要一对标记查找范围的迭代器以及一个在该范围内查找的值作参数。查找完成后，该函数返回一个迭代器，它指向具有指定值的第一个元素，或超出末端的下一位置。

```
string searchValue("Quasimodo");
list<string>::iterator iter =
    find(slist.begin(), slist.end(), searchValue);
if (iter != slist.end())
    slist.erase(iter);
```

注意，在删除元素之前，必须确保迭代器是不是 end 迭代器。使用 erase 操作删除单个元素必须确保该元素确实存在——如果删除指向超出末端的下一位置的迭代器，那么 erase 操作的行为未定义。

3. 删除容器内所有元素

要删除容器内所有的元素，可以调用 clear 函数，或将 begin 和 end 迭代器传递给 erase 函数。

```
slist.clear(); // delete all the elements within the container
slist.erase(slist.begin(), slist.end()); // equivalent
```

erase 函数的迭代器对版本提供了删除一部分元素的功能：

```
// delete range of elements between two values
list<string>::iterator elem1, elem2;
```

```
// elem1 refers to val1
elem1 = find(slist.begin(), slist.end(), val1);
// elem2 refers to the first occurrence of val2 after val1
elem2 = find(elem1, slist.end(), val2);
// erase range from val1 up to but not including val2
slist.erase(elem1, elem2);
```

这段代码首先调用了 `find` 函数两次，以获得指向特定元素的两个迭代器。迭代器 `elem1` 指向第一个具有 `val1` 值的元素，如果容器 `list` 中不存在值为 `val1` 的元素，则该迭代器指向超出末端的下一位置。如果在 `val1` 元素后面存在值为 `val2` 的元素，那么迭代器 `elem2` 就指向这段范围内第一个具有 `val2` 值的元素，否则，`elem2` 就是一个超出末端的迭代器。最后，调用 `erase` 函数删除从迭代器 `elem1` 开始一直到 `elem2` 之间的所有元素，但不包括 `elem2` 指向的元素。

327



`erase`、`pop_front` 和 `pop_back` 函数使指向被删除元素的所有迭代器失效。对于 `vector` 容器，指向删除点后面的元素的迭代器通常也会失效。而对于 `deque` 容器，如果删除时不包含第一个元素或最后一个元素，那么该 `deque` 容器相关的所有迭代器都会失效。

习题

习题 9.25 需要删除一段元素时，如果 `val1` 与 `val2` 相等，那么程序会发生什么事情？如果 `val1` 和 `val2` 中的一个不存在，或两个都不存在，程序又会怎么样？

习题 9.26 假设有如下 `ia` 的定义，将 `ia` 复制到一个 `vector` 容器和一个 `list` 容器中。使用单个迭代器参数版本的 `erase` 函数将 `list` 容器中的奇数值元素删除掉，然后将 `vector` 容器中的偶数值元素删除掉。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

习题 9.27 编写程序处理一个 `string` 类型的 `list` 容器。在该容器中寻找一个特殊值，如果找到，则将它删除掉。用 `deque` 容器重写上述程序。

9.3.8 赋值与 `swap`

与赋值相关的操作符都作用于整个容器。除 `swap` 操作外，其他操作都可以用 `erase` 和 `insert` 操作实现（表 9-11）。赋值操作符首先删除其左操作数容器中的所有元素，然后将右操作数容器的所有元素插入到左边容器中：

```
c1 = c2; // replace contents of c1 with a copy of elements in c2
// equivalent operation using erase and insert
c1.erase(c1.begin(), c1.end()); // delete all elements in c1
c1.insert(c1.begin(), c2.begin(), c2.end()); // insert c2
```

赋值后，左右两边的容器相等：尽管赋值前两个容器的长度可能不相等，但赋值后两个容器都具有右操作数的长度。



赋值和 `assign` 操作使左操作数容器的所有迭代器失效。`swap` 操作则不会使迭代器失效。完成 `swap` 操作后，尽管被交换的元素已经存放在另一容器中，但迭代器仍然指向相同的元素。

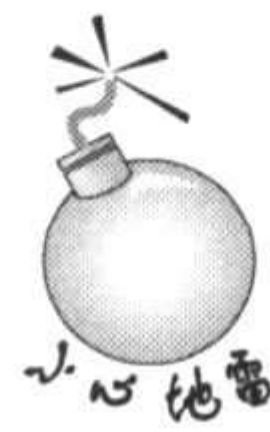
表 9-11 顺序容器的赋值操作

<code>c1 = c2</code>	删除容器 <code>c1</code> 的所有元素，然后将 <code>c2</code> 的元素复制给 <code>c1</code> 。 <code>c1</code> 和 <code>c2</code> 的类型（包括容器类型和元素类型）必须相同
<code>c1.swap(c2)</code>	交换内容：调用完该函数后， <code>c1</code> 中存放的是 <code>c2</code> 原来的元素， <code>c2</code> 中存放的则是 <code>c1</code> 原来的元素。 <code>c1</code> 和 <code>c2</code> 的类型必须相同。该函数的执行速度通常要比将 <code>c2</code> 的元素复制到 <code>c1</code> 的操作快
<code>c.assign(b,e)</code>	重新设置 <code>c</code> 的元素：将迭代器 <code>b</code> 和 <code>e</code> 标记的范围内所有的元素复制到 <code>c</code> 中。 <code>b</code> 和 <code>e</code> 必须不是指向 <code>c</code> 中元素的迭代器
<code>c.assign(n,t)</code>	将容器 <code>c</code> 重新设置为存储 <code>n</code> 个值为 <code>t</code> 的元素

1. 使用 `assign`

`assign` 操作首先删除容器中所有的元素，然后将其参数所指定的新元素插入到该容器中。与复制容器元素的构造函数一样，如果两个容器类型相同，其元素类型也相同，就可以使用赋值操作符（`=`）将一个容器赋值给另一个容器。如果在不同（或相同）类型的容器内，元素类型不相同但是相互兼容，则其赋值运算必须使用 `assign` 函数。例如，可通过 `assign` 操作实现将 `vector` 容器中一段 `char*` 类型的元素赋给 `string` 类型的 `list` 容器。

328



由于 `assign` 操作首先删除容器中原来存储的所有元素，因此，传递给 `assign` 函数的迭代器不能指向调用该函数的容器内的元素。

`assign` 函数的参数决定了要插入多少个元素以及新元素的值是什么。语句

```
// equivalent to slist1 = slist2
slist1.assign(slist2.begin(), slist2.end());
```

使用了带一对迭代器参数的 `assign` 函数版本。在删除 `slist1` 的元素后，该函数将 `slist2` 容器内一段指定的元素复制到 `slist1` 中。于是，这段代码等效于将 `slist2` 赋给 `slist1`。



带有一对迭代器参数的 `assign` 操作允许我们将一个容器的元素赋给另一个不同类型的容器。

`assign` 运算的第二个版本需要一个整型数值和一个元素值做参数，它将容器重置为存储指定数量的元素，并且每个元素的值都为指定值：

```
// equivalent to: slist1.clear();
// followed by slist1.insert(slist1.begin(), 10, "Hiya!");
slist1.assign(10, "Hiya!"); // 10 elements; each one is Hiya!
```

执行了上述语句后，容器 `slist1` 有 10 个元素，每个元素的值都是 `Hiya!`。

2. 使用 `swap` 操作以节省删除元素的成本

329

`swap` 操作实现交换两个容器内所有元素的功能。要交换的容器的类型必须匹配：操作数必须是相同类型的容器，而且所存储的元素类型也必须相同。调用了 `swap` 函数后，右操作数原来存储的元素被存放在左操作数中，反之亦然。

```
vector<string> svec1(10);    // vector with 10 elements
vector<string> svec2(24);    // vector with 24 elements
svec1.swap(svec2);
```

执行 `swap` 后，容器 `svec1` 中存储 24 个 `string` 类型的元素，而 `svec2` 则存储 10 个元素。



关于 `swap` 的一个重要问题在于：该操作不会删除或插入任何元素，而且保证在常量时间内实现交换。由于容器内没有移动任何元素，因此迭代器不会失效。

没有移动元素这个事实意味着迭代器不会失效。它们指向同一元素，就像没做 `swap` 运算之前一样。虽然，在 `swap` 运算后，这些元素已经被存储在不同的容器之中了。例如，在做 `swap` 运算之前，有一个迭代器 `iter` 指向 `svec1[3]` 字符串；实现 `swap` 运算后，该迭代器则指向 `svec2[3]` 字符串（这是同一个字符串，只是存储在不同的容器之中而已）。

习题

习题 9.28 编写程序将一个 `list` 容器的所有元素赋值给一个 `vector` 容器，其中 `list` 容器中存储的是指向 C 风格字符串的 `char*` 指针，而 `vector` 容器的元素则是 `string` 类型。

9.4 `vector` 容器的自增长

在容器对象中 `insert` 或压入一个元素时，该对象的大小增加 1。类似地，如果 `resize` 容器以扩充其容量，则必须在容器中添加额外的元素。标准库处理存储这些新元素的内存分配问题。

一般来说，我们不应该关心标准库类型是如何实现的：我们只需要关心如何使用这些标准库类型就可以了。然而，对于 `vector` 容器，有一些实现也与其接口相关。为了支持快速的随机访问，`vector` 容器的元素以连续的方式存放——每一个元素都紧挨着前一个元素存储。

已知元素是连续存储的，当我们在容器内添加一个元素时，想想会发生什么事情：如果容器中已经没有空间容纳新的元素，此时，由于元素必须连续存储以便索引访问，所以不能在内存中随便找个地方存储这个新元素。于是，`vector` 必须重新分配存储空间，用来存放原来的元素以及新添加的元素：存放在旧存储空间中的元素被复制到新存储空间里，接着插入新元素，最后撤销旧的存储空间。如果 `vector` 容器在每次添加新元素时，都要这么分配和撤销内存空间，其性能将会非常慢，简直无法接受。

对于不连续存储元素的容器，不存在这样的内存分配问题。例如，在 `list` 容器中添加一个元素，标准库只需创建一个新元素，然后将该新元素连接在已存在的链表中，不需要重新分配存储空间，也不必复制任何已存在的元素。

330

由此可以推论：一般而言，使用 `list` 容器优于 `vector` 容器。但是，通常出现的反而是以