对于最后一次递归调用 print(cout, 42)，两个 print 版本都是可行的。这个调用传递两个实参，第一个实参的类型为 ostream&。因此，可变参数版本的 print 可以实例化为只接受两个参数：一个是 ostream&参数，另一个是 const T&参数。

对于最后一个调用，两个函数提供同样好的匹配。但是，非可变参数模板比可变参数模板更特例化，因此编译器选择非可变参数版本（参见 16.3 节，第 615 页）。

当定义可变参数版本的 print 时，非可变参数版本的声明必须在作用域中。否则，可变参数版本会无限递归。

16.4.1 节练习

练习 16.53：编写你自己版本的 print 函数，并打印一个、两个及五个实参来测试它，要打印的每个实参都应有不同的类型。

练习 16.54：如果我们对一个没有<<运算符的类型调用 print，会发生什么？

练习 16.55：如果我们的可变参数版本 print 的定义之后声明非可变参数版本，解释可变参数的版本会如何执行。

16.4.2 包扩展

对于一个参数包，除了获取其大小外，我们能对它做的唯一的事情就是**扩展**（expand）它。当扩展一个包时，我们还要提供用于每个扩展元素的**模式**（pattern）。扩展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表。我们通过在模式右边放一个省略号（...）来触发扩展操作。 703

例如，我们的 print 函数包含两个扩展：

```
template <typename T, typename... Args>
ostream &
print(ostream &os, const T &t, const Args&... rest)    // 扩展 Args
{
    os << t << ", ";
    return print(os, rest...);                          // 扩展 rest
}
```

第一个扩展操作扩展模板参数包，为 print 生成函数参数列表。第二个扩展操作出现在对 print 的调用中。此模式为 print 调用生成实参列表。

对 Args 的扩展中，编译器将模式 const Arg&应用到模板参数包 Args 中的每个元素。因此，此模式的扩展结果是一个逗号分隔的零个或多个类型的列表，每个类型都形如 const type&。例如：

```
print(cout, i, s, 42); // 包中有两个参数
```

最后两个实参的类型和模式一起确定了尾置参数的类型。此调用被实例化为：

```
ostream&
print(ostream&, const int&, const string&, const int&);
```

第二个扩展发生在对 print 的（递归）调用中。在此情况下，模式是函数参数包的名字（即 rest）。此模式扩展出一个由包中元素组成的、逗号分隔的列表。因此，这个调

用等价于：

```
print(os, s, 42);
```

理解包扩展

print 中的函数参数包扩展仅仅将包扩展为其构成元素，C++语言还允许更复杂的扩展模式。例如，我们可以编写第二个可变参数函数，对其每个实参调用 debug_rep（参见 16.3 节，第 615 页），然后调用 print 打印结果 string：

```
// 在 print 调用中对每个实参调用 debug_rep
template <typename... Args>
ostream &errorMsg(ostream &os, const Args&... rest)
{
    // print(os, debug_rep(a1), debug_rep(a2), ..., debug_rep(an)
    return print(os, debug_rep(rest)...);
}
```

704 这个 print 调用使用了模式 debug_reg(rest)。此模式表示我们希望对函数参数包 rest 中的每个元素调用 debug_rep。扩展结果将是一个逗号分隔的 debug_rep 调用列表。即，下面调用：

```
errorMsg(cerr, fcnName, code.num(), otherData, "other", item);
```

就好像我们这样编写代码一样

```
print(cerr, debug_rep(fcnName), debug_rep(code.num()),
            debug_rep(otherData), debug_rep("otherData"),
            debug_rep(item));
```

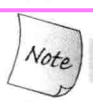
与之相对，下面的模式会编译失败

```
// 将包传递给 debug_rep; print(os, debug_rep(a1, a2, ..., an))
print(os, debug_rep(rest...)); // 错误：此调用无匹配函数
```

这段代码的问题是我们在 debug_rep 调用中扩展了 rest，它等价于

```
print(cerr, debug_rep(fcnName, code.num(),
                      otherData, "otherData", item));
```

在这个扩展中，我们试图用一个五个实参的列表来调用 debug_rep，但并不存在与此调用匹配的 debug_rep 版本。debug_rep 函数不是可变参数的，而且没有哪个 debug_rep 版本接受五个参数。



扩展中的模式会独立地应用于包中的每个元素。

16.4.2 节练习

练习 16.56：编写并测试可变参数版本的 errorMsg。

练习 16.57：比较你的可变参数版本的 errorMsg 和 6.2.6 节（第 198 页）中的 error_msg 函数。两种方法的优点和缺点各是什么？

16.4.3 转发参数包

C++11 在新标准下，我们可以组合使用可变参数模板与 forward 机制来编写函数，实现将

其实参不变地传递给其他函数。作为例子，我们将为 StrVec 类（参见 13.5 节，第 465 页）添加一个 emplace_back 成员。标准库容器的 emplace_back 成员是一个可变参数成员模板（参见 16.1.4 节，第 596 页），它用其实参在容器管理的内存空间中直接构造一个元素。

我们为 StrVec 设计的 emplace_back 版本也应该是可变参数的，因为 string 有多个构造函数，参数各不相同。由于我们希望能使用 string 的移动构造函数，因此还需要保持传递给 emplace_back 的实参的所有类型信息。 705

如我们所见，保持类型信息是一个两阶段的过程。首先，为了保持实参中的类型信息，必须将 emplace_back 的函数参数定义为模板类型参数的右值引用（参见 16.2.7 节，第 613 页）：

```
class StrVec {
public:
    template <class... Args> void emplace_back(Args&&...);
    // 其他成员的定义，同 13.5 节（第 465 页）
};
```

模板参数包扩展中的模式是&&，意味着每个函数参数将是一个指向其对应实参的右值引用。

其次，当 emplace_back 将这些实参传递给 construct 时，我们必须使用 forward 来保持实参的原始类型（参见 16.2.7 节，第 614 页）：

```
template <class... Args>
inline
void StrVec::emplace_back(Args&&... args)
{
    chk_n_alloc(); // 如果需要的话重新分配 StrVec 内存空间
    alloc.construct(first_free++, std::forward<Args>(args)...);
}
```

emplace_back 的函数体调用了 chk_n_alloc（参见 13.5 节，第 465 页）来确保有足够的空间容纳一个新元素，然后调用了 construct 在 first_free 指向的位置中创建了一个元素。construct 调用中的扩展为

```
std::forward<Args>(args)...
```

它既扩展了模板参数包 Args，也扩展了函数参数包 args。此模式生成如下形式的元素

```
std::forward<Ti>(ti)
```

其中 T_i 表示模板参数包中第 i 个元素的类型，t_i 表示函数参数包中第 i 个元素。例如，假定 svec 是一个 StrVec，如果我们调用

```
svec.emplace_back(10, 'c'); // 将 cccccccccc 添加为新的尾元素
```

construct 调用中的模式会扩展出

```
std::forward<int>(10), std::forward<char>(c)
```

通过在此调用中使用 forward，我们保证如果用一个右值调用 emplace_back，则 construct 也会得到一个右值。例如，在下面的调用中：

```
svec.emplace_back(s1 + s2); // 使用移动构造函数
```

传递给 emplace_back 的实参是一个右值，它将以如下形式传递给 construct

```
std::forward<string>(string("the end"))
```

forward<string>的结果类型是 string&&，因此 construct 将得到一个右值引用实参。construct 会继续将此实参传递给 string 的移动构造函数来创建新元素。

706

建议：转发和可变参数模板

可变参数函数通常将它们的参数转发给其他函数。这种函数通常具有与我们的 emplace_back 函数一样的形式：

```
// fun 有零个或多个参数，每个参数都是一个模板参数类型的右值引用
template<typename... Args>
void fun(Args&&... args) // 将 Args 扩展为一个右值引用的列表
{
    // work 的实参既扩展 Args 又扩展 args
    work(std::forward<Args>(args)...);
}
```

这里我们希望将 fun 的所有实参转发给另一个名为 work 的函数，假定由它完成函数的实际工作。类似 emplace_back 中对 construct 的调用，work 调用中的扩展既扩展了模板参数包也扩展了函数参数包。

由于 fun 的参数是右值引用，因此我们可以传递给它任意类型的实参；由于我们使用 std::forward 传递这些实参，因此它们的所有类型信息在调用 work 时都会得到保持。

16.4.3 节练习

练习 16.58：为你的 StrVec 类及你为 16.1.2 节（第 591 页）练习中编写的 Vec 类添加 emplace_back 函数。

练习 16.59：假定 s 是一个 string，解释调用 svec.emplace_back(s) 会发生什么。

练习 16.60：解释 make_shared（参见 12.1.1 节，第 401 页）是如何工作的。

练习 16.61：定义你自己版本的 make_shared。

16.5 模板特例化

编写单一模板，使之对任何可能的模板实参都是最适合的，都能实例化，这并不总是能办到。在某些情况下，通用模板的定义对特定类型是不适合的：通用定义可能编译失败或做得不正确。其他时候，我们也可以利用某些特定知识来编写更高效的代码，而不是从通用模板实例化。当我们不能（或不希望）使用模板版本时，可以定义类或函数模板的一个特例化版本。

我们的 compare 函数是一个很好的例子，它展示了函数模板的通用定义不适合一个特定类型（即字符指针）的情况。我们希望 compare 通过调用 strcmp 比较两个字符指针而非比较指针值。实际上，我们已经重载了 compare 函数来处理字符串字面常量（参见 16.1.1 节，第 579 页）：

707

```
// 第一个版本；可以比较任意两个类型
template <typename T> int compare(const T&, const T&);
// 第二个版本处理字符串字面常量
template<size_t N, size_t M>
int compare(const char (&)[N], const char (&)[M]);
```

但是，只有当我们传递给 compare 一个字符串字面常量或者一个数组时，编译器才会调用接受两个非类型模板参数的版本。如果我们传递给它字符指针，就会调用第一个版本：

```
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2);          // 调用第一个模板
compare("hi", "mom");     // 调用有两个非类型参数的版本
```

我们无法将一个指针转换为一个数组的引用，因此当参数是 p1 和 p2 时，第二个版本的 compare 是不可行的。

为了处理字符指针（而不是数组），可以为第一个版本的 compare 定义一个**模板特例化**（template specialization）版本。一个特例化版本就是模板的一个独立的定义，在其中一个或多个模板参数被指定为特定的类型。

定义函数模板特例化

当我们特例化一个函数模板时，必须为原模板中的每个模板参数都提供实参。为了指出我们正在实例化一个模板，应使用关键字 template 后跟一个空尖括号对（<>）。空尖括号指出我们将为原模板的所有模板参数提供实参：

```
// compare 的特殊版本，处理字符数组的指针
template <>
int compare(const char* const &p1, const char* const &p2)
{
    return strcmp(p1, p2);
}
```

理解此特例化版本的困难之处是函数参数类型。当我们定义一个特例化版本时，函数参数类型必须与一个先前声明的模板中对应的类型匹配。本例中我们特例化：

```
template <typename T> int compare(const T&, const T&);
```

其中函数参数为一个 const 类型的引用。类似类型别名，模板参数类型、指针及 const 之间的相互作用会令人惊讶（参见 2.5.1 节，第 60 页）。

我们希望定义此函数的一个特例化版本，其中 T 为 const char*。我们的函数要求一个指向此类型 const 版本的引用。一个指针类型的 const 版本是一个常量指针而不是指向 const 类型的指针（参见 2.4.2 节，第 56 页）。我们需要在特例化版本中使用的类型是 const char * const &，即一个指向 const char 的 const 指针的引用。

函数重载与模板特例化

708

当定义函数模板的特例化版本时，我们本质上接管了编译器的工作。即，我们为原模板的一个特殊实例提供了定义。重要的是要弄清：一个特例化版本本质上是一个实例，而非函数名的一个重载版本。

特例化的本质是实例化一个模板，而非重载它。因此，特例化不影响函数匹配。

我们将一个特殊的函数定义为一个特例化版本还是一个独立的非模板函数，会影响到函数匹配。例如，我们已经定义了两个版本的 compare 函数模板，一个接受数组引用参数，另一个接受 const T&。我们还定义了一个特例化版本来处理字符指针，这对函数匹配没有影响。当我们对字符串字面常量调用 compare 时

```
compare("hi", "mom")
```

对此调用，两个函数模板都是可行的，且提供同样好的（即精确的）匹配。但是，接受字符数组参数的版本更特例化（参见 16.3 节，第 615 页），因此编译器会选择它。

如果我们将接受字符指针的 compare 版本定义为一个普通的非模板函数（而不是模板的一个特例化版本），此调用的解析就会不同。在此情况下，将会有三个可行的函数：两个模板和非模板的字符指针版本。所有三个函数都提供同样好的匹配。如前所述，当一个非模板函数提供与函数模板同样好的匹配时，编译器会选择非模板版本（参见 16.3 节，第 615 页）。

关键概念：普通作用域规则应用于特例化

为了特例化一个模板，原模板的声明必须在作用域中。而且，在任何使用模板实例的代码之前，特例化版本的声明也必须在作用域中。

对于普通类和函数，丢失声明的情况（通常）很容易发现——编译器将不能继续处理我们的代码。但是，如果丢失了一个特例化版本的声明，编译器通常可以用原模板生成代码。由于在丢失特例化版本时编译器通常会实例化原模板，很容易产生模板及其特例化版本声明顺序导致的错误，而这种错误又很难查找。

如果一个程序使用一个特例化版本，而同时原模板的一个实例具有相同的模板实参集合，就会产生错误。但是，这种错误编译器又无法发现。

Best Practices 模板及其特例化版本应该声明在同一个头文件中。所有同名模板的声明应该放在前面，然后是这些模板的特例化版本。

709 ## 类模板特例化

除了特例化函数模板，我们还可以特例化类模板。作为一个例子，我们将为标准库 hash 模板定义一个特例化版本，可以用它来将 Sales_data 对象保存在无序容器中。默认情况下，无序容器使用 hash<key_type>（参见 11.4 节，第 394 页）来组织其元素。为了让我们自己的数据类型也能使用这种默认组织方式，必须定义 hash 模板的一个特例化版本。一个特例化 hash 类必须定义：

- 一个重载的调用运算符（参见 14.8 节，第 506 页），它接受一个容器关键字类型的对象，返回一个 size_t。
- 两个类型成员，result_type 和 argument_type，分别调用运算符的返回类型和参数类型。
- 默认构造函数和拷贝赋值运算符（可以隐式定义，参见 13.1.2 节，第 443 页）。

在定义此特例化版本的 hash 时，唯一复杂的地方是：必须在原模板定义所在的命名空间中特例化它。我们将在 18.2 节（第 695 页）中介绍更多命名空间的相关内容。现在，我们只需知道——我们可以向命名空间添加成员。为了达到这一目的，首先必须打开命名空间：

```
// 打开 std 命名空间，以便特例化 std::hash
namespace std {
```

```
} // 关闭 std 命名空间；注意：右花括号之后没有分号
```

花括号对之间的任何定义都将成为命名空间 std 的一部分。

下面的代码定义了一个能处理 Sales_data 的特例化 hash 版本：

```
// 打开 std 命名空间，以便特例化 std::hash
namespace std {
template <> // 我们正在定义一个特例化版本，模板参数为 Sales_data
struct hash<Sales_data>
{
    // 用来散列一个无序容器的类型必须要定义下列类型
    typedef size_t result_type;
    typedef Sales_data argument_type; // 默认情况下，此类型需要==
    size_t operator()(const Sales_data& s) const;
    // 我们的类使用合成的拷贝控制成员和默认构造函数
};
size_t
hash<Sales_data>::operator()(const Sales_data& s) const
{
    return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue);
}
} // 关闭 std 命名空间；注意：右花括号之后没有分号
```

我们的 hash<Sales_data>定义以 template<>开始，指出我们正在定义一个全特例化 710
的模板。我们正在特例化的模板名为 hash，而特例化版本为 hash<Sales_data>。接下来的类成员是按照特例化 hash 的要求而定义的。

类似其他任何类，我们可以在类内或类外定义特例化版本的成员，本例中就是在类外定义的。重载的调用运算符必须为给定类型的值定义一个哈希函数。对于一个给定值，任何时候调用此函数都应该返回相同的结果。一个好的哈希函数对不相等的对象（几乎总是）应该产生不同的结果。

在本例中，我们将定义一个好的哈希函数的复杂任务交给了标准库。标准库为内置类型和很多标准库类型定义了 hash 类的特例化版本。我们使用一个（未命名的）hash<string>对象来生成 bookNo 的哈希值，用一个 hash<unsigned>对象来生成 units_sold 的哈希值，用一个 hash<double>对象来生成 revenue 的哈希值。我们将这些结果进行异或运算（参见 4.8 节，第 137 页），形成给定 Sales_data 对象的完整的哈希值。

值得注意的是，我们的 hash 函数计算所有三个数据成员的哈希值，从而与我们为 Sales_data 定义的 operator==（参见 14.3.1 节，第 497 页）是兼容的。默认情况下，为了处理特定关键字类型，无序容器会组合使用 key_type 对应的特例化 hash 版本和 key_type 上的相等运算符。

假定我们的特例化版本在作用域中，当将 Sales_data 作为容器的关键字类型时，编译器就会自动使用此特例化版本：

```
// 使用 hash<Sales_data>和 14.3.1 节（第 497 页）中 Sales_data 的 operator==
unordered_multiset<Sales_data> SDset;
```

由于 hash<Sales_data>使用 Sales_data 的私有成员，我们必须将它声明为

Sales_data 的友元：

```
template <class T> class std::hash; // 友元声明所需要的
class Sales_data {
friend class std::hash<Sales_data>;
    // 其他成员定义，如前
};
```

这段代码指出特殊实例 hash<Sales_data>是 Sales_data 的友元。由于此实例定义在 std 命名空间中，我们必须记得在 friend 声明中应使用 std::hash。

为了让 Sales_data 的用户能使用 hash 的特例化版本，我们应该在 Sales_data 的头文件中定义该特例化版本。

类模板部分特例化

与函数模板不同，类模板的特例化不必为所有模板参数提供实参。我们可以只指定一
711 部分而非所有模板参数，或是参数的一部分而非全部特性。一个类模板的**部分特例化**（partial specialization）本身是一个模板，使用它时用户还必须为那些在特例化版本中未指定的模板参数提供实参。

我们只能部分特例化类模板，而不能部分特例化函数模板。

在 16.2.3 节（第 605 页）中我们介绍了标准库 remove_reference 类型。该模板是通过一系列的特例化版本来完成其功能的：

```
// 原始的、最通用的版本
template <class T> struct remove_reference {
    typedef T type;
};
// 部分特例化版本，将用于左值引用和右值引用
template <class T> struct remove_reference<T&> // 左值引用
    { typedef T type; };
template <class T> struct remove_reference<T&&> // 右值引用
    { typedef T type; };
```

第一个模板定义了最通用的模板。它可以用任意类型实例化；它将模板实参作为 type 成员的类型。接下来的两个类是原始模板的部分特例化版本。

由于一个部分特例化版本本质是一个模板，与往常一样，我们首先定义模板参数。类似任何其他特例化版本，部分特例化版本的名字与原模板的名字相同。对每个未完全确定类型的模板参数，在特例化版本的模板参数列表中都有一项与之对应。在类名之后，我们为要特例化的模板参数指定实参，这些实参列于模板名之后的尖括号中。这些实参与原始模板中的参数按位置对应。

部分特例化版本的模板参数列表是原始模板的参数列表的一个子集或者是一个特例化版本。在本例中，特例化版本的模板参数的数目与原始模板相同，但是类型不同。两个特例化版本分别用于左值引用和右值引用类型：

```
int i;
// decltype(42)为 int，使用原始模板
remove_reference<decltype(42)>::type a;
```

```
// decltype(i)为int&，使用第一个（T&）部分特例化版本
remove_reference<decltype(i)>::type b;
// decltype(std::move(i))为int&&，使用第二个（即T&&）部分特例化版本
remove_reference<decltype(std::move(i))>::type c;
```

三个变量a、b和c均为int类型。

特例化成员而不是类

我们可以只特例化特定成员函数而不是特例化整个模板。例如，如果Foo是一个模板类，包含一个成员Bar，我们可以只特例化该成员：

712

```
template <typename T> struct Foo {
    Foo(const T &t = T()): mem(t) { }
    void Bar() { /* ... */ }
    T mem;
    // Foo的其他成员
};
template<>              // 我们正在特例化一个模板
void Foo<int>::Bar() // 我们正在特例化Foo<int>的成员Bar
{
    // 进行应用于int的特例化处理
}
```

本例中我们只特例化Foo<int>类的一个成员，其他成员将由Foo模板提供：

```
Foo<string> fs;          // 实例化Foo<string>::Foo()
fs.Bar();                // 实例化Foo<string>::Bar()
Foo<int> fi;             // 实例化Foo<int>::Foo()
fi.Bar();                // 使用我们特例化版本的Foo<int>::Bar()
```

当我们用int之外的任何类型使用Foo时，其成员像往常一样进行实例化。当我们用int使用Foo时，Bar之外的成员像往常一样进行实例化。如果我们使用Foo<int>的成员Bar，则会使用我们定义的特例化版本。

16.5 节练习

练习 16.62：定义你自己版本的hash<Sales_data>，并定义一个Sales_data对象的unordered_multiset。将多条交易记录保存到容器中，并打印其内容。

练习 16.63：定义一个函数模板，统计一个给定值在一个vector中出现的次数。测试你的函数，分别传递给它一个double的vector，一个int的vector以及一个string的vector。

练习 16.64：为上一题中的模板编写特例化版本来处理vector<const char*>。编写程序使用这个特例化版本。

练习 16.65：在16.3节（第617页）中我们定义了两个重载的debug_rep版本，一个接受const char*参数，另一个接受char*参数。将这两个函数重写为特例化版本。

练习 16.66：重载debug_rep函数与特例化它相比，有何优点和缺点？

练习 16.67：定义特例化版本会影响debug_rep的函数匹配吗？如果不影响，为什么？

713

小结

模板是C++语言与众不同的特性，也是标准库的基础。一个模板就是一个编译器用来生成特定类类型或函数的蓝图。生成特定类或函数的过程称为实例化。我们只编写一次模板，就可以将其用于多种类型和值，编译器会为每种类型和值进行模板实例化。

我们既可以定义函数模板，也可以定义类模板。标准库算法都是函数模板，标准库容器都是类模板。

显式模板实参允许我们固定一个或多个模板参数的类型或值。对于指定了显式模板实参的模板参数，可以应用正常的类型转换。

一个模板特例化就是一个用户提供的模板实例，它将一个或多个模板参数绑定到特定类型或值上。当我们不能（或不希望）将模板定义用于某些特定类型时，特例化非常有用。

最新C++标准的一个主要部分是可变参数模板。一个可变参数模板可以接受数目和类型可变的参数。可变参数模板允许我们编写像容器的 `emplace` 成员和标准库 `make_shared` 函数这样的函数，实现将实参传递给对象的构造函数。

术语表

类模板（class template） 模板定义，可从它实例化出特定的类。类模板的定义以关键字 `template` 开始，后跟尖括号对<和>，其内为一个用逗号分隔的一个或多个模板参数的列表，随后是类的定义。

默认模板实参（default template argument） 一个类型或一个值，当用户未提供对应模板实参时，模板会使用它。

显式实例化（explicit instantiation） 一个声明，为所有模板参数提供了显式实参。用来指导实例化过程。如果声明是 `extern` 的，模板将不会被实例化；否则，模板将利用指定的实参进行实例化。对每个 `extern` 模板声明，在程序中某处必须有一个非 `extern` 的显式实例化。

显式模板实参（explicit template argument） 在一个函数调用中或定义模板类类型时，由用户提供的模板实参。显式模板实参在紧跟在模板名的尖括号对中给出。

函数参数包（function parameter pack） 表示零个或多个函数参数的参数包。

函数模板（function template） 模板定义，可从它实例化出特定函数。函数模板的定义以关键字 `template` 开始，后跟尖括号对<和>，其内为一个用逗号分隔的一个或多个模板参数的列表，随后是函数的定义。

实例化（instantiate） 编译器处理过程，用实际的模板实参来生成模板的一个特殊实例，其中参数被替换为对应的实参。当函数模板被调用时，会自动根据传递给它的实参来实例化。而使用类模板时，则需要我们提供显式模板实参。

714 **实例（instantiation）** 编译器从模板生成的类或函数。

成员模板（member template） 本身是模板的成员函数。成员模板不能是虚函数。

非类型参数（nontype parameter） 表示值的模板参数。非类型模板参数的实参必须是常量表达式。

包扩展（pack expansion） 处理过程，将一个参数包替换为其中元素的列表。

参数包（parameter pack） 表示零个或多个参数的模板或函数参数。

部分特例化（partial specialization） 类模板的一个版本，其中指定了某些但不是所

有模板参数，或是一个或多个参数的属性未被完全指定。

模式（pattern） 定义了扩展后参数包中每个元素的形式。

模板实参（template argument） 用来实例化模板参数的类型或值。

模板实参推断（template argument deduction） 编译器确定实例化哪个函数模板的过程。编译器检查那些使用模板参数的实参的类型，将这些类型或值绑定到模板参数，来自动实例化一个函数版本。

模板参数（template parameter） 在模板参数列表中指定的名字，可在模板定义内部使用。模板参数可以是类型参数，也可以是非类型参数。为了使用一个类模板，我们必须为每个模板参数提供显式实参。编译器使用这些类型或值实例化出一个类版本，其中所有用到模板参数的地方都被替换为实际的实参。当使用一个函数模板时，编译器使用调用中的函数实参推断模板实参，并使用推断出的模板实参实例化出一个特定的函数。

模板参数列表（template parameter list） 用逗号分隔的参数列表，用于模板的定义或声明中。每个参数可以是一个类型参数，也可以是一个非类型参数。

模板参数包（template parameter pack） 表示零个或多个模板参数的参数包。

模板特例化（template specialization） 类模板、类模板的成员或函数模板的重定义，其中指定了某些（或全部）模板参数。模板特例化版本必须出现在原模板的声明之后，必须出现在任何利用特殊实参来使用模板的代码之前。一个函数模板中的每个模板参数都必须完全特例化。

类型参数（type parameter） 模板参数列表中的名字，用来表示类型。类型参数在关键字 `typename` 或 `class` 之后指定。

类型转换（type transformation） 由标准库定义的类模板，可将给定的模板类型参数转换为一个相关类型。

可变参数模板（variadic template） 接受可变数目模板实参的模板。模板参数包用省略号指定（如 `class...`、`typename...`或 *`type-name`*`...`）


here!

第Ⅳ部分
高级主题

内容

第Ⅳ部分将介绍 C++和标准库的一些附加特性，虽然这些特性在特定的情况下很有用，但并非每个 C++程序员都需要它们。这些特性分为两类：一类对于求解大规模的问题很有用；另一类适用于特殊问题而非通用问题。针对特殊问题的特性既有属于 C++语言的（将在第 19 章介绍），也有属于标准库的（将在第 17 章进行介绍）。

在第 17 章中我们介绍四个具有特殊目的的标准库设施：`bitset` 类和三个新标准库设施（`tuple`、正则表达式和随机数）。我们还将介绍 IO 库中某些不常用的部分。

第 18 章介绍异常处理、命名空间和多重继承。这些特性在设计大型程序时是最有用的。

即使是一个程序员就能编写的足够简单的程序，也能从异常处理机制受益，这也是为什么我们在第 5 章介绍了异常处理的基本知识的原因。但是，对于需要大型团队才能完成的程序设计问题，运行时错误处理才显得更为重要也更难于管理。在第 18 章中，我们会额外介绍一些有用的异常处理设施。我们还将详细讨论异常是如何处理的，并展示如何定义和使用自己的异常类。这一章还会介绍新标准中异常处理方面的改进——如何指出一个特定函数不会抛出异常。

大型应用程序通常会使用来自多个提供商的代码。如果提供商不得不将他们定义的名字放置在单一的命名空间中，那么将多个独立开发的库组合起来是很困难的（如果能组合的话）。独立开发的库几乎必然会使用与其他库相同的名字；对于某个库中定义的名字，如果另一个库中使用了相同的名字，就会引起冲突。为了避免名字冲突，我们可以在一个 `namespace` 中定义名字。

无论何时我们使用一个来自标准库的名字，实际上都是在使用名为 std 的命名空间中的名字。第 18 章将会展示如何定义我们自己的命名空间。

第 18 章最后介绍一个很重要但不太常用的语言特性：多重继承。多重继承对非常复杂的继承层次很有用。

第 19 章介绍几种用于特定类别问题的特殊工具和技术，包括如何重定义内存分配机制；C++对运行时类型识别（run-time type identification，RTTI）的支持——允许我们在运行时才确定一个表达式的实际类型；以及如何定义和使用指向类成员的指针。类成员指针不同于普通数据或函数指针。普通指针仅根据对象或函数的类型而变化，而类成员指针还必须反映成员所属的类。我们还将介绍三种附加的聚合类型：联合、嵌套类和局部类。这一章最后将简要介绍一组本质上不可移植的语言特性：volatile 修饰符、位域以及链接指令。

done!

dDrlo!

第 17 章
标准库特殊设施

内容

最新的 C++标准极大地扩充了标准库的规模和范围。实际上，从 1998 年的第一版标准到 2011 年的最新标准，标准库部分的篇幅增加了两倍以上。因此，介绍所有 C++标准库类的知识大大超出了本书范围。但是，有 4 个标准库设施，虽然它们比我们已经介绍的其他标准库设施更特殊，但也足够通用，应该放在一本入门书籍中进行介绍。这 4 个标准库设施是：`tuple`、`bitset`、随机数生成及正则表达式。此外，我们还将介绍 IO 库中一些具有特殊目的的部分。

718 标准库占据了新标准文本将近三分之二的篇幅。虽然我们不能详细介绍所有标准库设施，但仍有一些标准库设施在很多应用中都是有用的：tuple、bitset、正则表达式以及随机数。我们还将介绍一些附加的 IO 库功能：格式控制、未格式化 IO 和随机访问。

17.1 tuple 类型

C++ 11 **tuple** 是类似 pair（参见 11.2.3 节，第 379 页）的模板。每个 pair 的成员类型都不相同，但每个 pair 都恰好有两个成员。不同 tuple 类型的成员类型也不相同，但一个 tuple 可以有任意数量的成员。每个确定的 tuple 类型的成员数目是固定的，但一个 tuple 类型的成员数目可以与另一个 tuple 类型不同。

当我们希望将一些数据组合成单一对象，但又不想麻烦地定义一个新数据结构来表示这些数据时，tuple 是非常有用的。表 17.1 列出了 tuple 支持的操作。tuple 类型及其伴随类型和函数都定义在 tuple 头文件中。

表 17.1：tuple 支持的操作

`tuple<T1, T2, ..., Tn> t;`	t 是一个 tuple，成员数为 n，第 i 个成员的类型为 Ti。所有成员都进行值初始化（参见 3.3.1 节，第 88 页）
`tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);`	t 是一个 tuple，成员类型为 T1...Tn，每个成员用对应的初始值 v_i 进行初始化。此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
`make_tuple(v1, v2, ..., vn)`	返回一个用给定初始值初始化的 tuple。tuple 的类型从初始值的类型推断
`t1 == t2` `t1 != t2`	当两个 tuple 具有相同数量的成员且成员对应相等时，两个 tuple 相等。这两个操作使用成员的==运算符来完成。一旦发现某对成员不等，接下来的成员就不用比较了
`t1` *relop* `t2`	tuple 的关系运算使用字典序（参见 9.2.7 节，第 304 页）。两个 tuple 必须具有相同数量的成员。使用<运算符比较 t1 的成员和 t2 中的对应成员
`get<i>(t)`	返回 t 的第 i 个数据成员的引用；如果 t 是一个左值，结果是一个左值引用；否则，结果是一个右值引用。tuple 的所有成员都是 public 的
`tuple_size<`*tupleType*`>::value`	一个类模板，可以通过一个 tuple 类型来初始化。它有一个名为 value 的 public constexpr static 数据成员，类型为 size_t，表示给定 tuple 类型中成员的数量
`tuple_element<i, tupleType>::type`	一个类模板，可以通过一个整型常量和一个 tuple 类型来初始化。它有一个名为 type 的 public 成员，表示给定 tuple 类型中指定成员的类型

Note 我们可以将 tuple 看作一个“快速而随意”的数据结构。

17.1.1 定义和初始化 tuple

当我们定义一个 tuple 时，需要指出每个成员的类型：

```
tuple<size_t, size_t, size_t> threeD; // 三个成员都设置为 0
tuple<string, vector<double>, int, list<int>>
    someVal("constants", {3.14, 2.718}, 42, {0,1,2,3,4,5})
```

当我们创建一个 tuple 对象时，可以使用 tuple 的默认构造函数，它会对每个成员进行值初始化（参见 3.3.1 节，第 88 页）；也可以像本例中初始化 someVal 一样，为每个成员提供一个初始值。tuple 的这个构造函数是 explicit 的（参见 7.5.4 节，第 265 页），因此我们必须使用直接初始化语法：

```
tuple<size_t, size_t, size_t> threeD = {1,2,3};   // 错误
tuple<size_t, size_t, size_t> threeD{1,2,3};      // 正确
```

类似 make_pair 函数（参见 11.2.3 节，第 381 页），标准库定义了 make_tuple 函数，我们还可以用它来生成 tuple 对象：

```
// 表示书店交易记录的 tuple，包含：ISBN、数量和每册书的价格
auto item = make_tuple("0-999-78345-X", 3, 20.00);
```

类似 make_pair，make_tuple 函数使用初始值的类型来推断 tuple 的类型。在本例中，item 是一个 tuple，类型为 tuple<const char*, int, double>。

访问 tuple 的成员

719

一个 pair 总是有两个成员，这样，标准库就可以为它们命名（如，first 和 second）。但这种命名方式对 tuple 是不可能的，因为一个 tuple 类型的成员数目是没有限制的。因此，tuple 的成员都是未命名的。要访问一个 tuple 的成员，就要使用一个名为 **get** 的标准库函数模板。为了使用 get，我们必须指定一个显式模板实参（参见 16.2.2 节，第 603 页），它指出我们想要访问第几个成员。我们传递给 get 一个 tuple 对象，它返回指定成员的引用：

```
auto book = get<0>(item);          // 返回 item 的第一个成员
auto cnt = get<1>(item);           // 返回 item 的第二个成员
auto price = get<2>(item)/cnt;     // 返回 item 的最后一个成员
get<2>(item) *= 0.8;               // 打折 20%
```

尖括号中的值必须是一个整型常量表达式（参见 2.4.4 节，第 58 页）。与往常一样，我们从 0 开始计数，意味着 get<0>是第一个成员。

如果不知道一个 tuple 准确的类型细节信息，可以用两个辅助类模板来查询 tuple 成员的数量和类型：

720

```
typedef decltype(item) trans; // trans 是 item 的类型
// 返回 trans 类型对象中成员的数量
size_t sz = tuple_size<trans>::value; // 返回 3
// cnt 的类型与 item 中第二个成员相同
tuple_element<1, trans>::type cnt = get<1>(item); // cnt 是一个 int
```

为了使用 tuple_size 或 tuple_element，我们需要知道一个 tuple 对象的类型。与往常一样，确定一个对象的类型的最简单方法就是使用 decltype（参见 2.5.3 节，第 62 页）。在本例中，我们使用 decltype 来为 item 类型定义一个类型别名，用它来实例化

两个模板。

tuple_size 有一个名为 value 的 public static 数据成员，它表示给定 tuple 中成员的数量。tuple_element 模板除了一个 tuple 类型外，还接受一个索引值。它有一个名为 type 的 public 类型成员，表示给定 tuple 类型中指定成员的类型。类似 get，tuple_element 所使用的索引也是从 0 开始计数的。

关系和相等运算符

tuple 的关系和相等运算符的行为类似容器的对应操作（参见 9.2.7 节，第 304 页）。这些运算符逐对比较左侧 tuple 和右侧 tuple 的成员。只有两个 tuple 具有相同数量的成员时，我们才可以比较它们。而且，为了使用 tuple 的相等或不等运算符，对每对成员使用==运算符必须都是合法的；为了使用关系运算符，对每对成员使用<必须都是合法的。例如：

```
tuple<string, string> duo("1", "2");
tuple<size_t, size_t> twoD(1, 2);
bool b = (duo == twoD);   // 错误：不能比较 size_t 和 string
tuple<size_t, size_t, size_t> threeD(1, 2, 3);
b = (twoD < threeD);      // 错误：成员数量不同
tuple<size_t, size_t> origin(0, 0);
b = (origin < twoD);      // 正确：b 为 true
```

由于 tuple 定义了<和==运算符，我们可以将 tuple 序列传递给算法，并且可以在无序容器中将 tuple 作为关键字类型。

17.1.1 节练习

练习 17.1：定义一个保存三个 int 值的 tuple，并将其成员分别初始化为 10、20 和 30。

练习 17.2：定义一个 tuple，保存一个 string、一个 vector<string>和一个 pair<string, int>。

练习 17.3：重写 12.3 节（第 430 页）中的 TextQuery 程序，使用 tuple 代替 QueryResult 类。你认为哪种设计更好？为什么？

721

17.1.2 使用 tuple 返回多个值

tuple 的一个常见用途是从一个函数返回多个值。例如，我们的书店可能是多家连锁书店中的一家。每家书店都有一个销售记录文件，保存每本书近期的销售数据。我们可能希望在所有书店中查询某本书的销售情况。

假定每家书店都有一个销售记录文件。每个文件都将每本书的所有销售记录存放在一起。进一步假定已有一个函数可以读取这些销售记录文件，为每个书店创建一个 vector<Sales_data>，并将这些 vector 保存在 vector 的 vector 中：

```
// files 中的每个元素保存一家书店的销售记录
vector<vector<Sales_data>> files;
```

我们将编写一个函数，对于一本给定的书，在 files 中搜索出售过这本书的书店。对每家有匹配销售记录的书店，我们将创建一个 tuple 来保存这家书店的索引和两个迭代器。

索引指出了书店在 files 中的位置，而两个迭代器则标记了给定书籍在此书店的 vector<Sales_data>中第一条销售记录和最后一条销售记录之后的位置。

返回 tuple 的函数

我们首先编写查找给定书籍的函数。此函数的参数是刚刚提到的 vector 的 vector 以及一个表示书籍 ISBN 的 string。我们的函数将返回一个 tuple 的 vector，凡是销售了给定书籍的书店，都在 vector 中有对应的一项：

```
// matches 有三个成员：一家书店的索引和两个指向书店 vector 中元素的迭代器
typedef tuple<vector<Sales_data>::size_type,
              vector<Sales_data>::const_iterator,
              vector<Sales_data>::const_iterator> matches;
// files 保存每家书店的销售记录
// findBook 返回一个 vector，每家销售了给定书籍的书店在其中都有一项
vector<matches>
findBook(const vector<vector<Sales_data>> &files,
         const string &book)
{
    vector<matches> ret; // 初始化为空 vector
    // 对每家书店，查找与给定书籍匹配的记录范围（如果存在的话）
    for (auto it = files.cbegin(); it != files.cend(); ++it) {
        // 查找具有相同 ISBN 的 Sales_data 范围
        auto found = equal_range(it->cbegin(), it->cend(),
                                 book, compareIsbn);
        if (found.first != found.second) // 此书店销售了给定书籍
            // 记住此书店的索引及匹配的范围
            ret.push_back(make_tuple(it - files.cbegin(),
                                     found.first, found.second));
    }
    return ret; // 如果未找到匹配记录的话，ret 为空
}
```

for 循环遍历 files 中的元素，每个元素都是一个 vector。在 for 循环内，我们调用了一个名为 equal_range 的标准库算法，它的功能与关联容器的同名成员类似（参见 11.3.5 节，第 390 页）。equal_range 的前两个实参是表示输入序列的迭代器（参见 10.1 节，第 336 页），第三个参数是一个值。默认情况下，equal_range 使用<运算符来比较元素。由于 Sales_data 没有<运算符，因此我们传递给它一个指向 compareIsbn 函数的指针（参见 11.2.2 节，第 379 页）。

722

equal_range 算法返回一个迭代器 pair，表示元素的范围。如果未找到 book，则两个迭代器相等，表示空范围。否则，返回的 pair 的 first 成员将表示第一条匹配的记录，second 则表示匹配的尾后位置。

使用函数返回的 tuple

一旦我们创建了 vector 保存包含匹配的销售记录的书店，就需要处理这些记录了。在此程序中，对每家包含匹配销售记录的书店，我们将打印其汇总销售信息：

```
void reportResults(istream &in, ostream &os,
                   const vector<vector<Sales_data>> &files)
{
    string s; // 要查找的书
```

```
    while (in >> s) {
        auto trans = findBook(files, s); // 销售了这本书的书店
        if (trans.empty()) {
            cout << s << " not found in any stores" << endl;
            continue; // 获得下一本要查找的书
        }
        for (const auto &store : trans) // 对每家销售了给定书籍的书店
            // get<n>返回 store 中 tuple 的指定的成员
            os << "store " << get<0>(store) << " sales: "
               << accumulate(get<1>(store), get<2>(store),
                             Sales_data(s))
               << endl;
    }
}
```

while 循环反复读取名为 in 的 istream 来获得下一本要处理的书。我们调用 findBook 来检查 s 是否存在，并将结果赋予 trans。我们使用 auto 来简化 trans 类型的代码编写，它是一个 tuple 的 vector。

如果 trans 为空，表示没有关于 s 的销售记录。在此情况下，我们打印一条信息并返回，执行下一步 while 循环来获取下一本要查找的书。

for 循环将 store 绑定到 trans 中的每个元素。由于不希望改变 trans 中的元素，我们将 store 声明为 const 的引用。我们使用 get 来打印相关数据：get<0>表示对应书店的索引、get<1>表示第一条交易记录的迭代器、get<2>表示尾后位置的迭代器。

由于 Sales_data 定义了加法运算符（参见 14.3 节，第 497 页），因此我们可以用标准库的 accumulate 算法（参见 10.2.1 节，第 338 页）来累加销售记录。我们用 Sales_data 的接受一个 string 参数的构造函数（参见 7.1.4 节，第 236 页）来初始化一个 Sales_data 对象，将此对象传递给 accumulate 作为求和的起点。此构造函数用给定的 string 初始化 bookNo，并将 units_sold 和 revenue 成员置为 0。

723

17.1.2 节练习

练习 17.4：编写并测试你自己版本的 findBook 函数。

练习 17.5：重写 findBook，令其返回一个 pair，包含一个索引和一个迭代器 pair。

练习 17.6：重写 findBook，不使用 tuple 或 pair。

练习 17.7：解释你更倾向于哪个版本的 findBook，为什么。

练习 17.8：在本节最后一段代码中，如果我们将 Sales_data() 作为第三个参数传递给 accumulate，会发生什么？

17.2 bitset 类型

在 4.8 节（第 135 页）中我们介绍了将整型运算对象当作二进制位集合处理的一些内置运算符。标准库还定义了 **bitset** 类，使得位运算的使用更为容易，并且能够处理超过最长整型类型大小的位集合。bitset 类定义在头文件 bitset 中。

17.2.1 定义和初始化 bitset

表 17.2 列出了 bitset 的构造函数。bitset 类是一个类模板，它类似 array 类，具有固定的大小（参见 9.2.4 节，第 301 页）。当我们定义一个 bitset 时，需要声明它包含多少个二进制位：

```
bitset<32> bitvec(1U); // 32 位；低位为 1，其他位为 0
```

大小必须是一个常量表达式（参见 2.4.4 节，第 58 页）。这条语句定义 bitvec 为一个包含 32 位的 bitset。就像 vector 包含未命名的元素一样，bitset 中的二进制位也是未命名的，我们通过位置来访问它们。二进制位的位置是从 0 开始编号的。因此，bitvec 包含编号从 0 到 31 的 32 个二进制位。编号从 0 开始的二进制位被称为**低位**（low-order），编号到 31 结束的二进制位被称为**高位**（high-order）。

表 17.2：初始化 bitset 的方法

`bitset<n> b;`	b 有 n 位；每一位均为 0。此构造函数是一个 constexpr（参见 7.5.6 节，第 267 页）
`bitset<n> b(u);`	b 是 unsigned long long 值 u 的低 n 位的拷贝。如果 n 大于 unsigned long long 的大小，则 b 中超出 unsigned long long 的高位被置为 0。此构造函数是一个 constexpr（参见 7.5.6 节，第 267 页）
`bitset<n> b(s, pos, m, zero, one);`	b 是 string s 从位置 pos 开始 m 个字符的拷贝。s 只能包含字符 zero 或 one；如果 s 包含任何其他字符，构造函数会抛出 invalid_argument 异常。字符在 b 中分别保存为 zero 和 one。pos 默认为 0，m 默认为 string::npos，zero 默认为'0'，one 默认为'1'
`bitset<n> b(cp, pos, m, zero, one);`	与上一个构造函数相同，但从 cp 指向的字符数组中拷贝字符。如果未提供 m，则 cp 必须指向一个 C 风格字符串。如果提供了 m，则从 cp 开始必须至少有 m 个 zero 或 one 字符

接受一个 string 或一个字符指针的构造函数是 explicit 的（参见 7.5.4 节，第 265 页）。在新标准中增加了为 0 和 1 指定其他字符的功能。

用 unsigned 值初始化 bitset

当我们使用一个整型值来初始化 bitset 时，此值将被转换为 unsigned long long 类型并被当作位模式来处理。bitset 中的二进制位将是此模式的一个副本。如果 bitset 的大小大于一个 unsigned long long 中的二进制位数，则剩余的高位被置为 0。如果 bitset 的大小小于一个 unsigned long long 中的二进制位数，则只使用给定值中的低位，超出 bitset 大小的高位被丢弃：

724

```
// bitvec1 比初始值小；初始值中的高位被丢弃
bitset<13> bitvec1(0xbeef);   // 二进制位序列为 1111011101111
// bitvec2 比初始值大；它的高位被置为 0
bitset<20> bitvec2(0xbeef);   // 二进制位序列为 00001011111011101111
// 在 64 位机器中，long long 0ULL 是 64 个 0 比特，因此~0ULL 是 64 个 1
bitset<128> bitvec3(~0ULL);   // 0~63 位为 1；63~127 位为 0
```

从一个 string 初始化 bitset

我们可以从一个 string 或一个字符数组指针来初始化 bitset。两种情况下，字符都直接表示位模式。与往常一样，当我们使用字符串表示数时，字符串中下标最小的字符对应高位，反之亦然：

```
bitset<32> bitvec4("1100"); // 2、3 两位为 1，剩余两位为 0
```

如果 string 包含的字符数比 bitset 少，则 bitset 的高位被置为 0。

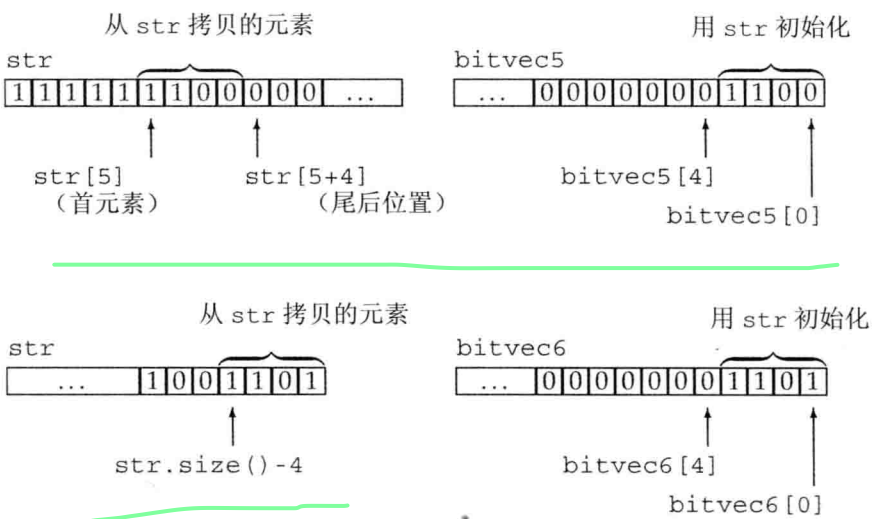
string 的下标编号习惯与 bitset 恰好相反：string 中下标最大的字符（最右字符）用来初始化 bitset 中的低位（下标为 0 的二进制位）。当你用一个 string 初始化一个 bitset 时，要记住这个差别。

725

我们不必使用整个 string 来作为 bitset 的初始值，可以只用一个子串作为初始值：

```
string str("1111111000000011001101");
bitset<32> bitvec5(str, 5, 4); // 从 str[5] 开始的四个二进制位，1100
bitset<32> bitvec6(str, str.size()-4); // 使用最后四个字符
```

此处，bitvec5 用 str 中从 str[5] 开始的长度为 4 的子串进行初始化。与往常一样，子串的最右字符表示最低位。因此，bitvec5 中第 3 位到第 0 位被设置为 1100，剩余位被设置为 0。传递给 bitvec6 的初始值是一个 string 和一个开始位置，因此 bitvec6 用 str 中倒数第四个字符开始的子串进行初始化。bitvec6 中剩余二进制位被初始化为 0。下图说明了这两个初始化过程



17.2.1 节练习

练习 17.9：解释下列每个 bitset 对象所包含的位模式：

(a) bitset<64> bitvec(32);
(b) bitset<32> bv(1010101);
(c) string bstr; cin >> bstr; bitset<8>bv(bstr);

17.2.2 bitset 操作

bitset 操作（参见表 17.3）定义了多种检测或设置一个或多个二进制位的方法。bitset 类还支持我们在 4.8 节(第 136 页)中介绍过的位运算符。这些运算符用于 bitset 对象的含义与内置运算符用于 unsigned 运算对象相同。

726

表 17.3：bitset 操作

b.any()	b 中是否存在置位的二进制位
b.all()	b 中所有位都置位了吗
b.none()	b 中不存在置位的二进制位吗
b.count()	b 中置位的位数
b.size()	一个 constexpr 函数（参见 2.4.4 节，第 58 页），返回 b 中的位数
b.test(pos)	若 pos 位置的位是置位的，则返回 true，否则返回 false
b.set(pos,v) b.set()	将位置 pos 处的位设置为 bool 值 v。v 默认为 true。如果未传递实参，则将 b 中所有位置位
b.reset(pos) b.reset()	将位置 pos 处的位复位或将 b 中所有位复位
b.flip(pos) b.flip()	改变位置 pos 处的位的状态或改变 b 中每一位的状态
b[pos]	访问 b 中位置 pos 处的位；如果 b 是 const 的，则当该位置位时 b[pos] 返回一个 bool 值 true，否则返回 false
b.to_ulong() b.to_ullong()	返回一个 unsigned long 或一个 unsigned long long 值，其位模式与 b 相同。如果 b 中位模式不能放入指定的结果类型，则抛出一个 overflow_error 异常
b.to_string(zero, one)	返回一个 string，表示 b 中的位模式。zero 和 one 的默认值分别为 0 和 1，用来表示 b 中的 0 和 1
os << b	将 b 中二进制位打印为字符 1 或 0，打印到流 os
is >> b	从 is 读取字符存入 b。当下一个字符不是 1 或 0 时，或是已经读入 b.size() 个位时，读取过程停止

count、size、all、any 和 none 等几个操作都不接受参数，返回整个 bitset 的状态。其他操作——set、reset 和 flip 则改变 bitset 的状态。改变 bitset 状态的成员函数都是重载的。对每个函数，不接受参数的版本对整个集合执行给定的操作；接受一个位置参数的版本则对指定位执行操作：

```
bitset<32> bitvec(1U);              // 32 位；低位为 1，剩余位为 0
bool is_set = bitvec.any();         // true，因为有 1 位置位
bool is_not_set = bitvec.none();    // false，因为有 1 位置位了
bool all_set = bitvec.all();        // false，因为只有 1 位置位
size_t onBits = bitvec.count();     // 返回 1
size_t sz = bitvec.size();          // 返回 32
bitvec.flip();    // 翻转 bitvec 中的所有位
bitvec.reset();   // 将所有位复位
bitvec.set();     // 将所有位置位
```

当 bitset 对象的一个或多个位置位（即，等于 1）时，操作 any 返回 true。相反，当所有位复位时，none 返回 true。新标准引入了 all 操作，当所有位置位时返回 true。

C++ 11

727 操作 count 和 size 返回 size_t 类型的值（参见 3.5.2 节，第 103 页），分别表示对象中置位的位数或总位数。函数 size 是一个 constexpr 函数，因此可以用在要求常量表达式的地方（参见 2.4.4 节，第 58 页）。

成员 flip、set、reset 及 test 允许我们读写指定位置的位：

```
bitvec.flip(0);       // 翻转第一位
bitvec.set(bitvec.size() - 1); // 置位最后一位
bitvec.set(0, 0);     // 复位第一位
bitvec.reset(i);      // 复位第 i 位
bitvec.test(0);       // 返回 false，因为第一位是复位的
```

下标运算符对 const 属性进行了重载。const 版本的下标运算符在指定位置位时返回 true，否则返回 false。非 const 版本返回 bitset 定义的一个特殊类型，它允许我们操纵指定位的值：

```
bitvec[0] = 0;            // 将第一位复位
bitvec[31] = bitvec[0];   // 将最后一位设置为与第一位一样
bitvec[0].flip();         // 翻转第一位
~bitvec[0];               // 等价操作，也是翻转第一位
bool b = bitvec[0];       // 将 bitvec[0] 的值转换为 bool 类型
```

提取 bitset 的值

to_ulong 和 to_ullong 操作都返回一个值，保存了与 bitset 对象相同的位模式。只有当 bitset 的大小小于等于对应的大小（to_ulong 为 unsigned long，to_ullong 为 unsigned long long）时，我们才能使用这两个操作：

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```

如果 bitset 中的值不能放入给定类型中，则这两个操作会抛出一个 overflow_error 异常（参见 5.6 节，第 173 页）。

bitset 的 IO 运算符

输入运算符从一个输入流读取字符，保存到一个临时的 string 对象中。直到读取的字符数达到对应 bitset 的大小时，或是遇到不是 1 或 0 的字符时，或是遇到文件尾或输入错误时，读取过程才停止。随即用临时 string 对象来初始化 bitset（参见 17.2.1 节，第 642 页）。如果读取的字符数小于 bitset 的大小，则与往常一样，高位将被置为 0。

输出运算符打印一个 bitset 对象中的位模式：

```
bitset<16> bits;
cin >> bits; // 从 cin 读取最多 16 个 0 或 1
cout << "bits: " << bits << endl; // 打印刚刚读取的内容
```

728 ### 使用 bitset

为了说明如何使用 bitset，我们重新实现 4.8 节（第 137 页）中的评分程序，用 bitset 代替 unsigned long 表示 30 个学生的测验结果——“通过/失败”：

```
bool status;
// 使用位运算符的版本
unsigned long quizA = 0;          // 此值被当作位集合使用
```

```
quizA |= 1UL << 27;                    // 指出第 27 个学生通过了测验
status = quizA & (1UL << 27);          // 检查第 27 个学生是否通过了测验
quizA &= ~(1UL << 27);                 // 第 27 个学生未通过测验
// 使用标准库类 bitset 完成等价的工作
bitset<30> quizB;                      // 每个学生分配一位，所有位都被初始化为 0
quizB.set(27);                         // 指出第 27 个学生通过了测验
status = quizB[27];                    // 检查第 27 个学生是否通过了测验
quizB.reset(27);                       // 第 27 个学生未通过测验
```

17.2.2 节练习

练习 17.10：使用序列 1、2、3、5、8、13、21 初始化一个 bitset，将这些位置置位。对另一个 bitset 进行默认初始化，并编写一小段程序将其恰当的位置位。

练习 17.11：定义一个数据结构，包含一个整型对象，记录一个包含 10 个问题的真/假测验的解答。如果测验包含 100 道题，你需要对数据结构做出什么改变（如果需要的话）？

练习 17.12：使用前一题中的数据结构，编写一个函数，它接受一个问题编号和一个表示真/假解答的值，函数根据这两个参数更新测验的解答。

练习 17.13：编写一个整型对象，包含真/假测验的正确答案。使用它来为前两题中的数据结构生成测验成绩。

17.3 正则表达式

正则表达式（regular expression）是一种描述字符序列的方法，是一种极其强大的计算工具。但是，用于定义正则表达式的描述语言已经大大超出了本书的范围。因此，我们重点介绍如何使用 C++正则表达式库（RE 库），它是新标准库的一部分。RE 库定义在头文件 regex 中，它包含多个组件，列于表 17.4 中。 C++11

表 17.4：正则表达式库组件

regex	表示有一个正则表达式的类
regex_match	将一个字符序列与一个正则表达式匹配
regex_search	寻找第一个与正则表达式匹配的子序列
regex_replace	使用给定格式替换一个正则表达式
sregex_iterator	迭代器适配器，调用 regex_search 来遍历一个 string 中所有匹配的子串
smatch	容器类，保存在 string 中搜索的结果
ssub_match	string 中匹配的子表达式的结果

如果你还不熟悉正则表达式的使用，你应该浏览这一节，以获得正则表达式可以做什么的一些概念。

regex 类表示一个正则表达式。除了初始化和赋值之外，regex 还支持其他一些操作。表 17.6（第 647 页）列出了 regex 支持的操作。 729

函数 **regex_match** 和 **regex_search** 确定一个给定字符序列与一个给定 regex

是否匹配。如果整个输入序列与表达式匹配，则 regex_match 函数返回 true；如果输入序列中一个子串与表达式匹配，则 regex_search 函数返回 true。还有一个 regex_replace 函数，我们将在 17.3.4 节（第 657 页）中介绍。

表 17.5 列出了 regex 的函数的参数。这些函数都返回 bool 值，且都被重载了：其中一个版本接受一个类型为 **smatch** 的附加参数。如果匹配成功，这些函数将成功匹配的相关信息保存在给定的 smatch 对象中。

表 17.5：regex_search 和 regex_match 的参数

注意：这些操作返回 bool 值，指出是否找到匹配。

(*seq*, m, r, mft) (*seq*, r, mft)	在字符序列 *seq* 中查找 regex 对象 r 中的正则表达式。*seq* 可以是一个 string、表示范围的一对迭代器以及一个指向空字符结尾的字符数组的指针
	m 是一个 *match* 对象，用来保存匹配结果的相关细节。m 和 *seq* 必须具有兼容的类型（参见 17.3.1 节，第 649 页）
	mft 是一个可选的 regex_constants::match_flag_type 值。表 17.13（第 659 页）描述了这些值，它们会影响匹配过程

17.3.1 使用正则表达式库

我们从一个非常简单的例子开始——查找违反众所周知的拼写规则“i 除非在 c 之后，否则必须在 e 之前”的单词：

```
// 查找不在字符 c 之后的字符串 ei
string pattern("[^c]ei");
// 我们需要包含 pattern 的整个单词
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern);      // 构造一个用于查找模式的 regex
smatch results;        // 定义一个对象保存搜索结果
// 定义一个 string 保存与模式匹配和不匹配的文本
string test_str = "receipt freind theif receive";
// 用 r 在 test_str 中查找与 pattern 匹配的子串
if (regex_search(test_str, results, r))    // 如果有匹配子串
    cout << results.str() << endl;         // 打印匹配的单词
```

我们首先定义了一个 string 来保存希望查找的正则表达式。正则表达式[^c]表明我们希望匹配任意不是'c'的字符，而[^c]ei 指出我们想要匹配这种字符后接 ei 的字符串。此模式描述的字符串恰好包含三个字符。我们想要包含此模式的单词的完整内容。为了与整个单词匹配，我们还需要一个正则表达式与这个三字母模式之前和之后的字母匹配。

730 这个正则表达式包含零个或多个字母后接我们的三字母的模式，然后再接零个或多个额外的字母。默认情况下，regex 使用的正则表达式语言是 ECMAScript。在 ECMAScript 中，模式[[::alpha:]]匹配任意字母，符号+和*分别表示我们希望“一个或多个”或“零个或多个”匹配。因此[[::alpha:]]*将匹配零个或多个字母。

将正则表达式存入 pattern 后，我们用它来初始化一个名为 r 的 regex 对象。接下来我们定义了一个 string，用来测试正则表达式。我们将 test_str 初始化为与模式匹配的单词（如"freind"和"thief"）和不匹配的单词（如"recepit"和"receive"）。我们还定义了一个名为 results 的 smatch 对象，它将被传递给 regex_search。如果找到匹配子串，results 将会保存匹配位置的细节信息。

接下来我们调用了 regex_search。如果它找到匹配子串，就返回 true。我们用 results 的 str 成员来打印 test_str 中与模式匹配的部分。函数 regex_search 在输入序列中只要找到一个匹配子串就会停止查找。因此，程序的输出将是

```
freind
```

17.3.2 节（第 650 页）将会介绍如何查找输入序列中所有的匹配子串。

指定 regex 对象的选项

当我们定义一个 regex 或是对一个 regex 调用 assign 为其赋予新值时，可以指定一些标志来影响 regex 如何操作。这些标志控制 regex 对象的处理过程。表 17.6 列出的最后 6 个标志指出编写正则表达式所用的语言。对这 6 个标志，我们必须设置其中之一，且只能设置一个。默认情况下，ECMAScript 标志被设置，从而 regex 会使用 ECMA-262 规范，这也是很多 Web 浏览器所使用的正则表达式语言。

表 17.6：regex（和 wregex）选项

`regex r(re)` `regex r(re, f)`	*re* 表示一个正则表达式，它可以是一个 string、一个表示字符范围的迭代器对、一个指向空字符结尾的字符数组的指针、一个字符指针和一个计数器或是一个花括号包围的字符列表。f 是指出对象如何处理的标志。f 通过下面列出的值来设置。如果未指定 f，其默认值为 ECMAScript
`r1 = re`	将 r1 中的正则表达式替换为 *re*。*re* 表示一个正则表达式，它可以是另一个 regex 对象、一个 string、一个指向空字符结尾的字符数组的指针或是一个花括号包围的字符列表
`r1.assign(re, f)`	与使用赋值运算符（=）效果相同；可选的标志 f 也与 regex 的构造函数中对应的参数含义相同
`r.mark_count()`	r 中子表达式的数目（我们将在 17.3.3 节（第 654 页）中介绍）
`r.flags()`	返回 r 的标志集

注：构造函数和赋值操作可能抛出类型为 regex_error 的异常。

定义 regex 时指定的标志

定义在 regex 和 regex_constants::syntax_option_type 中

icase	在匹配过程中忽略大小写
nosubs	不保存匹配的子表达式
optimize	执行速度优先于构造速度
ECMAScript	使用 ECMA-262 指定的语法
basic	使用 POSIX 基本的正则表达式语法
extended	使用 POSIX 扩展的正则表达式语法
awk	使用 POSIX 版本的 *awk* 语言的语法
grep	使用 POSIX 版本的 grep 的语法
egrep	使用 POSIX 版本的 egrep 的语法

其他 3 个标志允许我们指定正则表达式处理过程中与语言无关的方面。例如，我们可以指出希望正则表达式以大小写无关的方式进行匹配。

作为一个例子，我们可以用 icase 标志查找具有特定扩展名的文件名。大多数操作系统都是按大小写无关的方式来识别扩展名的——可以将一个 C++程序保存在 .cc 结尾的文件中，也可以保存在 .Cc、.cC 或是 .CC 结尾的文件中，效果是一样的。如下所示，我

们可以编写一个正则表达式来识别上述任何一种扩展名以及其他普通文件扩展名：

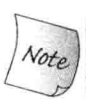
```
// 一个或多个字母或数字字符后接一个'.'再接"cpp"或"cxx"或"cc"
regex r("[[:alnum:]]+\\.(cpp|cxx|cc)$", regex::icase);
smatch results;
string filename;
while (cin >> filename)
    if (regex_search(filename, results, r))
        cout << results.str() << endl; // 打印匹配结果
```

此表达式将匹配这样的字符串：一个或多个字母或数字后接一个句点再接三个文件扩展名之一。这样，此正则表达式将会匹配指定的文件扩展名而不理会大小写。

就像 C++语言中有特殊字符一样（参见 2.1.3 节，第 36 页），正则表达式语言通常也有特殊字符。例如，字符点（.）通常匹配任意字符。与 C++一样，我们可以在字符之前放置一个反斜线来去掉其特殊含义。由于反斜线也是 C++中的一个特殊字符，我们在字符串字面常量中必须连续使用两个反斜线来告诉 C++我们想要一个普通反斜线字符。因此，为了表示与句点字符匹配的正则表达式，必须写成\\.（第一个反斜线去掉 C++语言中反斜线的特殊含义，即，正则表达式字符串为\.，第二个反斜线则表示在正则表达式中去掉.的特殊含义）。

732 ### 指定或使用正则表达式时的错误

我们可以将正则表达式本身看作用一种简单程序设计语言编写的“程序”。这种语言不是由 C++编译器解释的。正则表达式是在运行时，当一个 regex 对象被初始化或被赋予一个新模式时，才被“编译”的。与任何其他程序设计语言一样，我们用这种语言编写的正则表达式也可能有错误。



需要意识到的非常重要的一点是，一个正则表达式的语法是否正确是在运行时解析的。

如果我们编写的正则表达式存在错误，则在运行时标准库会抛出一个类型为 **regex_error** 的异常（参见 5.6 节，第 173 页）。类似标准异常类型，regex_error 有一个 what 操作来描述发生了什么错误（参见 5.6.2 节，第 175 页）。regex_error 还有一个名为 code 的成员，用来返回某个错误类型对应的数值编码。code 返回的值是由具体实现定义的。RE 库能抛出的标准错误如表 17.7 所示。

例如，我们可能在模式中意外遇到一个方括号：

```
try {
    // 错误：alnum 漏掉了右括号，构造函数会抛出异常
    regex r("[[:alnum:]+\\.(cpp|cxx|cc)$", regex::icase);
} catch (regex_error e)
  { cout << e.what() << "\ncode: " << e.code() << endl; }
```

当这段程序在我们的系统上运行时，程序会生成：

```
regex_error(error_brack):
The expression contained mismatched [ and ].
code: 4
```

表 17.7：正则表达式错误类型

定义在 regex 和 regex_constants::error_type 中	
error_collate	无效的元素校对请求
error_ctype	无效的字符类
error_escape	无效的转义字符或无效的尾置转义
error_backref	无效的向后引用
error_brack	不匹配的方括号（[或]）
error_paren	不匹配的小括号（(或)）
error_brace	不匹配的花括号（{或}）
error_badbrace	{}中无效的范围
error_range	无效的字符范围（如[z-a]）
error_space	内存不足，无法处理此正则表达式
error_badrepeat	重复字符（*、?、+或{）之前没有有效的正则表达式
error_complexity	要求的匹配过于复杂
error_stack	栈空间不足，无法处理匹配

我们的编译器定义了 code 成员，返回表 17.7 列出的错误类型的编号，与往常一样，编号从 0 开始。 733

建议：避免创建不必要的正则表达式

如我们所见，一个正则表达式所表示的“程序”是在运行时而非编译时编译的。正则表达式的编译是一个非常慢的操作，特别是在你使用了扩展的正则表达式语法或是复杂的正则表达式时。因此，构造一个 regex 对象以及向一个已存在的 regex 赋予一个新的正则表达式可能是非常耗时的。为了最小化这种开销，你应该努力避免创建很多不必要的 regex。特别是，如果你在一个循环中使用正则表达式，应该在循环外创建它，而不是在每步迭代时都编译它。

正则表达式类和输入序列类型

我们可以搜索多种类型的输入序列。输入可以是普通 char 数据或 wchar_t 数据，字符可以保存在标准库 string 中或是 char 数组中（或是宽字符版本，wstring 或 wchar_t 数组中）。RE 为这些不同的输入序列类型都定义了对应的类型。

例如，regex 类保存类型 char 的正则表达式。标准库还定义了一个 wregex 类保存类型 wachar_t，其操作与 regex 完全相同。两者唯一的差别是 wregex 的初始值必须使用 wchar_t 而不是 char。

匹配和迭代器类型（我们将在下面小节中介绍）更为特殊。这些类型的差异不仅在于字符类型，还在于序列是在标准库 string 中还是在数组中：smatch 表示 string 类型的输入序列；cmatch 表示字符数组序列；wsmatch 表示宽字符串（wstring）输入；而 wcmatch 表示宽字符数组。

重点在于我们使用的 RE 库类型必须与输入序列类型匹配。表 17.8 指出了 RE 库类型与输入序列类型的对应关系。例如：

```
regex r("[[:alnum:]]+\\.(cpp|cxx|cc)$", regex::icase);
smatch results; // 将匹配 string 输入序列，而不是 char*
if (regex_search("myfile.cc", results, r)) // 错误：输入为 char*
    cout << results.str() << endl;
```

734 这段代码会编译失败，因为 match 参数的类型与输入序列的类型不匹配。如果我们希望搜索一个字符数组，就必须使用 cmatch 对象：

```
cmatch results; // 将匹配字符数组输入序列
if (regex_search("myfile.cc", results, r))
    cout << results.str() << endl; // 打印当前匹配
```

本书程序一般会使用 string 输入序列和对应的 string 版本的 RE 库组件。

表 17.8：正则表达式库类

如果输入序列类型	则使用正则表达式类
string	regex、smatch、ssub_match 和 sregex_iterator
const char*	regex、cmatch、csub_match 和 cregex_iterator
wstring	wregex、wsmatch、wssub_match 和 wsregex_iterator
const wchar_t*	wregex、wcmatch、wcsub_match 和 wcregex_iterator

17.3.1 节练习

练习 17.14：编写几个正则表达式，分别触发不同错误。运行你的程序，观察编译器对每个错误的输出。

练习 17.15：编写程序，使用模式查找违反“i 在 e 之前，除非在 c 之后”规则的单词。你的程序应该提示用户输入一个单词，然后指出此单词是否符合要求。用一些违反和未违反规则的单词测试你的程序。

练习 17.16：如果前一题程序中的 regex 对象用"[^c]ei"进行初始化，将会发生什么？用此模式测试你的程序，检查你的答案是否正确。

17.3.2 匹配与 Regex 迭代器类型

第 646 页中的程序查找违反“i 在 e 之前，除非在 c 之后”规则的单词，它只打印输入序列中第一个匹配的单词。我们可以使用 **sregex_iterator** 来获得所有匹配。regex 迭代器是一种迭代器适配器（参见 9.6 节，第 329 页），被绑定到一个输入序列和一个 regex 对象上。如表 17.8 所述，每种不同输入序列类型都有对应的特殊 regex 迭代器类型。迭代器操作如表 17.9 所述。

表 17.9：sregex_iterator 操作

这些操作也适用于 cregex_iterator、wsregex_iterator 和 wcregex_iterator。

sregex_iterator	一个 sregex_iterator，遍历迭代器 b 和 e 表示的 string。
it(b, e, r);	它调用 sregex_search(b, e, r) 将 it 定位到输入中第一个匹配的位置

续表

sregex_iterator end;	sregex_iterator 的尾后迭代器
*it it->	根据最后一个调用 regex_search 的结果，返回一个 smatch 对象的引用或一个指向 smatch 对象的指针
++it it++	从输入序列当前匹配位置开始调用 regex_search。前置版本返回递增后迭代器；后置版本返回旧值
it1 == it2 it1 != it2	如果两个 sregex_iterator 都是尾后迭代器，则它们相等两个非尾后迭代器是从相同的输入序列和 regex 对象构造，则它们相等

当我们将一个 sregex_iterator 绑定到一个 string 和一个 regex 对象时，迭代器自动定位到给定 string 中第一个匹配位置。即，sregex_iterator 构造函数对给定 string 和 regex 调用 regex_search。当我们解引用迭代器时，会得到一个对应最近一次搜索结果的 smatch 对象。当我们递增迭代器时，它调用 regex_search 在输入 string 中查找下一个匹配。

使用 sregex_iterator

作为一个例子，我们将扩展之前的程序，在一个文本文件中查找所有违反“i 在 e 之前，除非在 c 之后”规则的单词。我们假定名为 file 的 string 保存了我们要搜索的输入文件的全部内容。这个版本的程序将使用与前一个版本一样的 pattern，但会使用一个 sregex_iterator 来进行搜索：

735

```
// 查找前一个字符不是 c 的字符串 ei
string pattern("[^c]ei");
// 我们想要包含 pattern 的单词的全部内容
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase); // 在进行匹配时将忽略大小写
// 它将反复调用 regex_search 来寻找文件中的所有匹配
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
          it != end_it; ++it)
        cout << it->str() << endl; // 匹配的单词
```

for 循环遍历 file 中每个与 r 匹配的子串。for 语句中的初始值定义了 it 和 end_it。当我们定义 it 时，sregex_iterator 的构造函数调用 regex_search 将 it 定位到 file 中第一个与 r 匹配的位置。而 end_it 是一个空 sregex_iterator，起到尾后迭代器的作用。for 语句中的递增运算通过 regex_search 来“推进”迭代器。当我们解引用迭代器时，会得到一个表示当前匹配结果的 smatch 对象。我们调用它的 str 成员来打印匹配的单词。

我们可以将此循环想象为不断从一个匹配位置跳到下一个匹配位置，如图 17.1 所示。

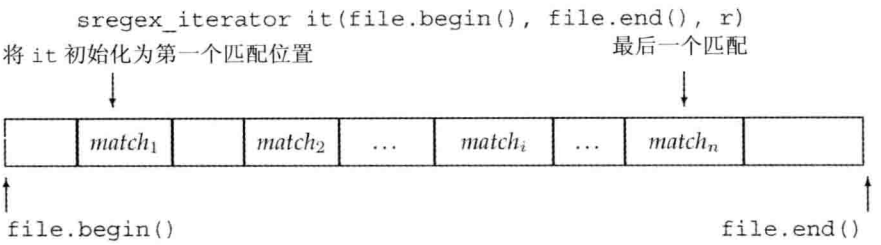


图 17.1：使用 sregex_iterator

使用匹配数据

如果我们对最初版本程序中的 test_str 运行此循环，则输出将是

```
freind
theif
```

但是，仅获得与我们的正则表达式匹配的单词还不是那么有用。如果我们在一个更大的输入序列——例如，在本章英文版的文本上运行此程序——可能希望看到匹配单词出现的上下文，如

```
hey read or write according to the type
      >>> being <<<
handled. The input operators ignore whi
```

除了允许打印输入字符串中匹配的部分之外，匹配结果类还提供了有关匹配结果的更多细节信息。表 17.10 和表 17.11 列出了这些类型支持的操作。

736 我们将在下一节中介绍更多有关 smatch 和 **ssub_match** 类型的内容。目前，我们只需知道它们允许我们获得匹配的上下文即可。匹配类型有两个名为 prefix 和 suffix 的成员，分别返回表示输入序列中当前匹配之前和之后部分的 ssub_match 对象。一个 ssub_match 对象有两个名为 str 和 length 的成员，分别返回匹配的 string 和该 string 的大小。我们可以用这些操作重写语法程序的循环。

```
// 循环头与之前一样
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
          it != end_it; ++it) {
     auto pos = it->prefix().length();          // 前缀的大小
     pos = pos > 40 ? pos - 40 : 0;             // 我们想要最多 40 个字符
     cout << it->prefix().str().substr(pos)     // 前缀的最后一部分
          << "\n\t\t>>> " << it->str() << " <<<\n" // 匹配的单词
          << it->suffix().str().substr(0, 40)       // 后缀的第一部分
          << endl;
}
```

循环本身的工作方式与前一个程序相同。改变的是循环内部，如图 17.2 所示。

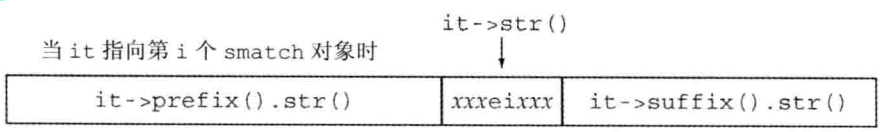


图 17.2：smatch 对象表示一个特定匹配

我们调用 prefix，返回一个 ssub_match 对象，表示 file 中当前匹配之前的部分。

我们对此 ssub_match 对象调用 length，获得前缀部分的字符数目。接下来调整 pos，使之指向前缀部分末尾向前 40 个字符的位置。如果前缀部分的长度小于 40 个字符，我们将 pos 置为 0，表示要打印整个前缀部分。我们用 substr（参见 9.5.1 节，第 321 页）来打印指定位置到前缀部分末尾的内容。

打印了当前匹配之前的字符之后，我们接下来用特殊格式打印匹配的单词本身，使得它在输出中能突出显示出来。打印匹配单词之后，我们打印 file 中匹配部分之后的前（最多）40 个字符。 737

表 17.10：smatch 操作

这些操作也适用于 cmatch、wsmatch、wcmatch 和对应的 csub_match、wssub_match 和 wcsub_match。

m.ready()	如果已经通过调用 regex_serach 或 regex_match 设置了 m，则返回 true；否则返回 false。如果 ready 返回 false，则对 m 进行操作是未定义的
m.size()	如果匹配失败，则返回 0；否则返回最近一次匹配的正则表达式中子表达式的数目
m.empty()	若 m.size() 为 0，则返回 true
m.prefix()	一个 ssub_match 对象，表示当前匹配之前的序列
m.suffix()	一个 ssub_match 对象，表示当前匹配之后的部分
m.format(…)	见表 17.12（第 657 页）

在接受一个索引的操作中，n 的默认值为 0 且必须小于 m.size()。
第一个子匹配（索引为 0）表示整个匹配。

m.length(n)	第 n 个匹配的子表达式的大小
m.position(n)	第 n 个子表达式距序列开始的距离
m.str(n)	第 n 个子表达式匹配的 string
m[n]	对应第 n 个子表达式的 ssub_match 对象
m.begin(),m.end() m.cbegin(),m.cend()	表示 m 中 sub_match 元素范围的迭代器。与往常一样，cbegin 和 cend 返回 const_iterator

17.3.2 节练习

练习 17.17：更新你的程序，令它查找输入序列中所有违反“ei”语法规则的单词。

练习 17.18：修改你的程序，忽略包含“ei”但并非拼写错误的单词，如“albeit”和“neighbor”。

17.3.3 使用子表达式 738

正则表达式中的模式通常包含一个或多个**子表达式**（subexpression）。一个子表达式是模式的一部分，本身也具有意义。正则表达式语法通常用括号表示子表达式。

例如，我们用来匹配 C++文件的模式（参见 17.3.1 节，第 646 页）就是用括号来分组可能的文件扩展名。每当我们用括号分组多个可行选项时，同时也就声明了这些选项形成子表达式。我们可以重写扩展名表达式，以使得模式中点之前表示文件名的部分也形成子表达式，如下所示：

```
// r 有两个子表达式：第一个是点之前表示文件名的部分，第二个表示文件扩展名
regex r("([[:alnum:]]+)\\.(cpp|cxx|cc)$", regex::icase);
```

现在我们的模式包含两个括号括起来的子表达式：

- `([[:alnum:]]+)`，匹配一个或多个字符的序列
- `(cpp|cxx|cc)`，匹配文件扩展名

我们还可以重写 17.3.1 节（第 646 页）中的程序，通过修改输出语句使之只打印文件名。

```
if (regex_search(filename, results, r))
    cout << results.str(1) << endl; // 打印第一个子表达式
```

与最初的程序一样，我们还是调用 `regex_search` 在名为 `filename` 的 `string` 中查找模式 `r`，并且传递 `smatch` 对象 `results` 来保存匹配结果。如果调用成功，我们打印结果。但是，在此版本中，我们打印的是 `str(1)`，即，与第一个子表达式匹配的部分。

匹配对象除了提供匹配整体的相关信息外，还提供访问模式中每个子表达式的能力。子匹配是按位置来访问的。第一个子匹配位置为 0，表示整个模式对应的匹配，随后是每个子表达式对应的匹配。因此，本例模式中第一个子表达式，即表示文件名的子表达式，其位置为 1，而文件扩展名对应的子表达式位置为 2。

例如，如果文件名为 `foo.cpp`，则 `results.str(0)` 将保存 `foo.cpp`；`results.str(1)` 将保存 `foo`；而 `results.str(2)` 将保存 `cpp`。在此程序中，我们想要点之前的那部分名字，即第一个子表达式，因此我们打印 `results.str(1)`。

子表达式用于数据验证

子表达式的一个常见用途是验证必须匹配特定格式的数据。例如，美国的电话号码有十位数字，包含一个区号和一个七位的本地号码。区号通常放在括号里，但这并不是必需的。剩余七位数字可以用一个短横线、一个点或是一个空格分隔，但也可以完全不用分隔符。我们可能希望接受任何这种格式的数据而拒绝任何其他格式的数。我们将分两步来实现这一目标：首先，我们将用一个正则表达式找到可能是电话号码的序列，然后再调用一个函数来完成数据验证。

739

在编写电话号码模式之前，我们需要介绍一下 ECMAScript 正则表达式语言的一些特性：

- `\{d}` 表示单个数字而 `\{d}{n}` 则表示一个 *n* 个数字的序列。（如，`\{d}{3}` 匹配三个数字的序列。）
- 在方括号中的字符集合表示匹配这些字符中任意一个。（如，`[-. ]` 匹配一个短横线或一个点或一个空格。注意，点在括号中没有特殊含义。）
- 后接 `'?'` 的组件是可选的。（如，`\{d}{3}[-. ]?\{d}{4}` 匹配这样的序列：开始是三个数字，后接一个可选的短横线或点或空格，然后是四个数字。此模式可以匹配 `555-0132` 或 `555.0132` 或 `555 0132` 或 `5550132`。）
- 类似 C++，ECMAScript 使用反斜线表示一个字符本身而不是其特殊含义。由于我们的模式包含括号，而括号是 ECMAScript 中的特殊字符，因此我们必须用 `\(` 和 `\)` 来表示括号是我们的模式的一部分而不是特殊字符。

由于反斜线是 C++中的特殊字符，在模式中每次出现 `\` 的地方，我们都必须用一个额外的反斜线来告知 C++我们需要一个反斜线字符而不是一个特殊符号。因此，我们用 `\\{d}{3}` 来表示正则表达式 `\{d}{3}`。