

操作列于表 12.2 中。

表 12.1: `shared_ptr` 和 `unique_ptr` 都支持的操作

<code>shared_ptr<T> sp</code>	空智能指针，可以指向类型为 T 的对象
<code>unique_ptr<T> up</code>	
<code>p</code>	将 p 用作一个条件判断，若 p 指向一个对象，则为 true
<code>*p</code>	解引用 p，获得它指向的对象
<code>p->mem</code>	等价于 <code>(*p).mem</code>
<code>p.get()</code>	返回 p 中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
<code>swap(p, q)</code>	交换 p 和 q 中的指针
<code>p.swap(q)</code>	

表 12.2: `shared_ptr` 独有的操作

<code>make_shared<T>(args)</code>	返回一个 <code>shared_ptr</code> ，指向一个动态分配的类型为 T 的对象。使用 args 初始化此对象
<code>shared_ptr<T>p(q)</code>	p 是 <code>shared_ptr q</code> 的拷贝；此操作会递增 q 中的计数器。q 中的指针必须能转换为 <code>T*</code> （参见 4.11.2 节，第 143 页）
<code>p = q</code>	p 和 q 都是 <code>shared_ptr</code> ，所保存的指针必须能相互转换。此操作会递减 p 的引用计数，递增 q 的引用计数；若 p 的引用计数变为 0，则将其管理的原内存释放
<code>p.unique()</code>	若 <code>p.use_count()</code> 为 1，返回 <code>true</code> ；否则返回 <code>false</code>
<code>p.use_count()</code>	返回与 p 共享对象的智能指针数量；可能很慢，主要用于调试

make_shared 函数

最安全的分配和使用动态内存的方法是调用一个名为 `make_shared` 的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的 `shared_ptr`。与智能指针一样，`make_shared` 也定义在头文件 `memory` 中。

当要用 `make_shared` 时，必须指定想要创建的对象的类型。定义方式与模板类相同，在函数名之后跟一个尖括号，在其中给出类型：

```
// 指向一个值为 42 的 int 的 shared_ptr
shared_ptr<int> p3 = make_shared<int>(42);
// p4 指向一个值为"9999999999"的 string
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 指向一个值初始化的(参见 3.3.1 节，第 88 页)int，即，值为 0
shared_ptr<int> p5 = make_shared<int>();
```

类似顺序容器的 `emplace` 成员（参见 9.3.1 节，第 308 页），`make_shared` 用其参数来构造给定类型的对象。例如，调用 `make_shared<string>` 时传递的参数必须与 `string` 的某个构造函数相匹配，调用 `make_shared<int>` 时传递的参数必须能用来初始化一个 `int`，依此类推。如果我们不传递任何参数，对象就会进行值初始化（参见 3.3.1 节，第 88 页）。

当然，我们通常用 `auto`（参见 2.5.2 节，第 61 页）定义一个对象来保存 `make_shared` 的结果，这种方式较为简单：

```
// p6 指向一个动态分配的空 vector<string>
auto p6 = make_shared<vector<string>>();
```

shared_ptr 的拷贝和赋值

当进行拷贝或赋值操作时，每个 shared_ptr 都会记录有多少个其他 shared_ptr 指向相同的对象：

```
auto p = make_shared<int>(42); // p 指向的对象只有 p 一个引用者
auto q(p); // p 和 q 指向相同对象，此对象有两个引用者
```

452 我们可以认为每个 shared_ptr 都有一个关联的计数器，通常称其为引用计数 (reference count)。无论何时我们拷贝一个 shared_ptr，计数器都会递增。例如，当用一个 shared_ptr 初始化另一个 shared_ptr，或将它作为参数传递给一个函数（参见 6.2.1 节，第 188 页）以及作为函数的返回值（参见 6.3.2 节，第 201 页）时，它所关联的计数器就会递增。当我们给 shared_ptr 赋予一个新值或是 shared_ptr 被销毁（例如一个局部的 shared_ptr 离开其作用域（参见 6.1.1 节，第 184 页）时，计数器就会递减。

一旦一个 shared_ptr 的计数器变为 0，它就会自动释放自己所管理的对象：

```
auto r = make_shared<int>(42); // r 指向的 int 只有一个引用者
r = q; // 给 r 赋值，令它指向另一个地址
// 递增 q 指向的对象的引用计数
// 递减 r 原来指向的对象的引用计数
// r 原来指向的对象已没有引用者，会自动释放
```

此例中我们分配了一个 int，将其指针保存在 r 中。接下来，我们将一个新值赋予 r。在此情况下，r 是唯一指向此 int 的 shared_ptr，在把 q 赋给 r 的过程中，此 int 被自动释放。



到底是用一个计数器还是其他数据结构来记录有多少指针共享对象，完全由标准库的具体实现来决定。关键是智能指针类能记录有多少个 shared_ptr 指向相同的对象，并能在恰当的时候自动释放对象。

shared_ptr 自动销毁所管理的对象……

当指向一个对象的最后一个 shared_ptr 被销毁时，shared_ptr 类会自动销毁此对象。它是通过另一个特殊的成员函数——析构函数 (destructor) 完成销毁工作的。类似于构造函数，每个类都有一个析构函数。就像构造函数控制初始化一样，析构函数控制此类型的对象销毁时做什么操作。

453 析构函数一般用来释放对象所分配的资源。例如，string 的构造函数（以及其他 string 成员）会分配内存来保存构成 string 的字符。string 的析构函数就负责释放这些内存。类似的，vector 的若干操作都会分配内存来保存其元素。vector 的析构函数就负责销毁这些元素，并释放它们所占用的内存。

shared_ptr 的析构函数会递减它所指向的对象的引用计数。如果引用计数变为 0，shared_ptr 的析构函数就会销毁对象，并释放它占用的内存。

……shared_ptr 还会自动释放相关联的内存

当动态对象不再被使用时，shared_ptr 类会自动释放动态对象，这一特性使得动态内存的使用变得非常容易。例如，我们可能有一个函数，它返回一个 shared_ptr，指向

一个 `Foo` 类型的动态分配的对象，对象是通过一个类型为 `T` 的参数进行初始化的：

```
// factory 返回一个 shared_ptr，指向一个动态分配的对象
shared_ptr<Foo> factory(T arg)
{
    // 恰当地处理 arg
    // shared_ptr 负责释放内存
    return make_shared<Foo>(arg);
}
```

由于 `factory` 返回一个 `shared_ptr`，所以我们可以确保它分配的对象会在恰当的时刻被释放。例如，下面的函数将 `factory` 返回的 `shared_ptr` 保存在局部变量中：

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // 使用 p
} // p 离开了作用域，它指向的内存会被自动释放掉
```

由于 `p` 是 `use_factory` 的局部变量，在 `use_factory` 结束时它将被销毁（参见 6.1.1 ◀454 节，第 184 页）。当 `p` 被销毁时，将递减其引用计数并检查它是否为 0。在此例中，`p` 是唯一引用 `factory` 返回的内存的对象。由于 `p` 将要销毁，`p` 指向的这个对象也会被销毁，所占用的内存会被释放。

但如果其他 `shared_ptr` 也指向这块内存，它就不会被释放掉：

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // 使用 p
    return p; // 当我们返回 p 时，引用计数进行了递增操作
} // p 离开了作用域，但它指向的内存不会被释放掉
```

在此版本中，`use_factory` 中的 `return` 语句向此函数的调用者返回一个 `p` 的拷贝。拷贝一个 `shared_ptr` 会增加所管理对象的引用计数值。现在当 `p` 被销毁时，它所指向的内存还有其他使用者。对于一块内存，`shared_ptr` 类保证只要有任何 `shared_ptr` 对象引用它，它就不会被释放掉。

由于在最后一个 `shared_ptr` 销毁前内存都不会释放，保证 `shared_ptr` 在无用之后不再保留就非常重要了。如果你忘记了销毁程序不再需要的 `shared_ptr`，程序仍会正确执行，但会浪费内存。`shared_ptr` 在无用之后仍然保留的一种可能情况是，你将 `shared_ptr` 存放在一个容器中，随后重排了容器，从而不再需要某些元素。在这种情况下，你应该确保用 `erase` 删除那些不再需要的 `shared_ptr` 元素。



如果你将 `shared_ptr` 存放于一个容器中，而后不再需要全部元素，而只使用其中一部分，要记得用 `erase` 删除不再需要的那些元素。

使用了动态生存期的资源的类

程序使用动态内存出于以下三种原因之一：

1. 程序不知道自己需要使用多少对象
2. 程序不知道所需对象的准确类型

3. 程序需要在多个对象间共享数据

容器类是出于第一种原因而使用动态内存的典型例子，我们将在第 15 章看到出于第二种原因而使用动态内存的例子。在本节中，我们将定义一个类，它使用动态内存是为了让多个对象能共享相同的底层数据。

到目前为止，我们使用过的类中，分配的资源都与对应对象生存期一致。例如，每个 `vector` “拥有” 其自己的元素。当我们拷贝一个 `vector` 时，原 `vector` 和副本 `vector` 中的元素是相互分离的：

```
455> vector<string> v1; // 空 vector
{ // 新作用域
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // 从 v2 拷贝元素到 v1 中
} // v2 被销毁，其中的元素也被销毁
// v1 有三个元素，是原来 v2 中元素的拷贝
```

由一个 `vector` 分配的元素只有当这个 `vector` 存在时才存在。当一个 `vector` 被销毁时，这个 `vector` 中的元素也都被销毁。

但某些类分配的资源具有与原对象相独立的生存期。例如，假定我们希望定义一个名为 `Blob` 的类，保存一组元素。与容器不同，我们希望 `Blob` 对象的不同拷贝之间共享相同的元素。即，当我们拷贝一个 `Blob` 时，原 `Blob` 对象及其拷贝应该引用相同的底层元素。

一般而言，如果两个对象共享底层的数据，当某个对象被销毁时，我们不能单方面地销毁底层数据：

```
Blob<string> b1; // 空 Blob
{ // 新作用域
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2; // b1 和 b2 共享相同的元素
} // b2 被销毁了，但 b2 中的元素不能销毁
// b1 指向最初由 b2 创建的元素
```

在此例中，`b1` 和 `b2` 共享相同的元素。当 `b2` 离开作用域时，这些元素必须保留，因为 `b1` 仍然在使用它们。



使用动态内存的一个常见原因是允许多个对象共享相同的状态。

定义 `StrBlob` 类

最终，我们会将 `Blob` 类实现为一个模板，但我们直到 16.1.2 节（第 583 页）才会学习模板的相关知识。因此，现在我们先定义一个管理 `string` 的类，此版本命名为 `StrBlob`。

实现一个新的集合类型的最简单方法是使用某个标准库容器来管理元素。采用这种方法，我们可以借助标准库类型来管理元素所使用的内存空间。在本例中，我们将使用 `vector` 来保存元素。

但是，我们不能在一个 `Blob` 对象内直接保存 `vector`，因为一个对象的成员在对象销毁时也会被销毁。例如，假定 `b1` 和 `b2` 是两个 `Blob` 对象，共享相同的 `vector`。如果此 `vector` 保存在其中一个 `Blob` 中——例如 `b2` 中，那么当 `b2` 离开作用域时，此 `vector` 也将被销毁，也就是说其中的元素都将不复存在。为了保证 `vector` 中的元素继续存在，

我们将 `vector` 保存在动态内存中。

为了实现我们所希望的数据共享，我们为每个 `StrBlob` 设置一个 `shared_ptr` 来管理动态分配的 `vector`。此 `shared_ptr` 的成员将记录有多少个 `StrBlob` 共享相同的 `vector`，并在 `vector` 的最后一个使用者被销毁时释放 `vector`。456

我们还需要确定这个类应该提供什么操作。当前，我们将实现一个 `vector` 操作的小的子集。我们会修改访问元素的操作（如 `front` 和 `back`）：在我们的类中，如果用户试图访问不存在的元素，这些操作会抛出一个异常。

我们的类有一个默认构造函数和一个构造函数，接受单一的 `initializer_list<string>` 类型参数（参见 6.2.6 节，第 198 页）。此构造函数可以接受一个初始化器的花括号列表。

```
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const std::string &t) {data->push_back(t);}
    void pop_back();
    // 元素访问
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // 如果 data[i] 不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};
```

在此类中，我们实现了 `size`、`empty` 和 `push_back` 成员。这些成员通过指向底层 `vector` 的 `data` 成员来完成它们的工作。例如，对一个 `StrBlob` 对象调用 `size()` 会调用 `data->size()`，依此类推。

StrBlob 构造函数

两个构造函数都使用初始化列表（参见 7.1.4 节，第 237 页）来初始化其 `data` 成员，令它指向一个动态分配的 `vector`。默认构造函数分配一个空 `vector`：

```
StrBlob::StrBlob(): data(make_shared<vector<string>>()) {}
StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) {}
```

接受一个 `initializer_list` 的构造函数将其参数传递给对应的 `vector` 构造函数（参见 2.2.1 节，第 39 页）。此构造函数通过拷贝列表中的值来初始化 `vector` 的元素。

元素访问成员函数

`pop_back`、`front` 和 `back` 操作访问 `vector` 中的元素。这些操作在试图访问元素之前必须检查元素是否存在。由于这些成员函数需要做相同的检查操作，我们为 `StrBlob` 定义了一个名为 `check` 的 `private` 工具函数，它检查一个给定索引是否在合法范围内。457

除了索引，`check` 还接受一个 `string` 参数，它会将此参数传递给异常处理程序，这个 `string` 描述了错误内容：

```
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

`pop_back` 和元素访问成员函数首先调用 `check`。如果 `check` 成功，这些成员函数继续利用底层 `vector` 的操作来完成自己的工作：

```
string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}

string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

`front` 和 `back` 应该对 `const` 进行重载（参见 7.3.2 节，第 247 页），这些版本的定义留作练习。

StrBlob 的拷贝、赋值和销毁

类似 `Sales_data` 类，`StrBlob` 使用默认版本的拷贝、赋值和销毁成员函数来对此类型的对象进行这些操作（参见 7.1.5 节，第 239 页）。默认情况下，这些操作拷贝、赋值和销毁类的数据成员。我们的 `StrBlob` 类只有一个数据成员，它是 `shared_ptr` 类型。因此，当我们拷贝、赋值或销毁一个 `StrBlob` 对象时，它的 `shared_ptr` 成员会被拷贝、赋值或销毁。

如前所见，拷贝一个 `shared_ptr` 会递增其引用计数；将一个 `shared_ptr` 赋予另一个 `shared_ptr` 会递增赋值号右侧 `shared_ptr` 的引用计数，而递减左侧 `shared_ptr` 的引用计数。如果一个 `shared_ptr` 的引用计数变为 0，它所指向的对象会被自动销毁。因此，对于由 `StrBlob` 构造函数分配的 `vector`，当最后一个指向它的 `StrBlob` 对象被销毁时，它会随之被自动销毁。

458

12.1.1 节练习

练习 12.1：在此代码的结尾，`b1` 和 `b2` 各包含多少个元素？

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
```

```

    b1 = b2;
    b2.push_back("about");
}

```

练习 12.2: 编写你自己的 StrBlob 类，包含 const 版本的 front 和 back。

练习 12.3: StrBlob 需要 const 版本的 push_back 和 pop_back 吗？如果需要，添加进去。否则，解释为什么不需要。

练习 12.4: 在我们的 check 函数中，没有检查 i 是否大于 0。为什么可以忽略这个检查？

练习 12.5: 我们未编写接受一个 initializer_list explicit（参见 7.5.4 节，第 264 页）参数的构造函数。讨论这个设计策略的优点和缺点。

12.1.2 直接管理内存

C++语言定义了两个运算符来分配和释放动态内存。运算符 new 分配内存，delete 释放 new 分配的内存。

相对于智能指针，使用这两个运算符管理内存非常容易出错，随着我们逐步详细介绍这两个运算符，这一点会更为清楚。而且，自己直接管理内存的类与使用智能指针的类不同，它们不能依赖类对象拷贝、赋值和销毁操作的任何默认定义（参见 7.1.4 节，第 237 页）。因此，使用智能指针的程序更容易编写和调试。



在学习第 13 章之前，除非使用智能指针来管理内存，否则不要分配动态内存。

使用 new 动态分配和初始化对象

在自由空间分配的内存是无名的，因此 new 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int *pi = new int; // pi 指向一个动态分配的、未初始化的无名对象
```

此 new 表达式在自由空间构造一个 int 型对象，并返回指向该对象的指针。

默认情况下，动态分配的对象是默认初始化的（参见 2.2.1 节，第 40 页），这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化：

```
string *ps = new string; // 初始化为空 string
int *pi = new int; // pi 指向一个未初始化的 int
```

459

我们可以使用直接初始化方式（参见 3.2.1 节，第 76 页）来初始化一个动态分配的对象。我们可以使用传统的构造方式（使用圆括号），在新标准下，也可以使用列表初始化（使用花括号）：

```
int *pi = new int(1024); // pi 指向的对象的值为 1024
string *ps = new string(10, '9'); // *ps 为"9999999999"
// vector 有 10 个元素，值依次从 0 到 9

vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

C++
11

也可以对动态分配的对象进行值初始化（参见 3.3.1 节，第 88 页），只需在类型名之

后跟一对空括号即可：

```
string *ps1 = new string;      // 默认初始化为空 string
string *ps = new string();    // 值初始化为空 string
int *pi1 = new int;           // 默认初始化；*pi1 的值未定义
int *pi2 = new int();         // 值初始化为 0；*pi2 为 0
```

对于定义了自己的构造函数（参见 7.1.4 节，第 235 页）的类类型（例如 `string`）来说，要求值初始化是没有意义的；不管采用什么形式，对象都会通过默认构造函数来初始化。但对于内置类型，两种形式的差别就很大了：值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值则是未定义的。类似的，对于类中那些依赖于编译器合成的默认构造函数的内置类型成员，如果它们未在类内被初始化，那么它们的值也是未定义的（参见 7.1.4 节，第 236 页）。

Best Practices

出于与变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。

C++ 11 如果我们提供了一个括号包围的初始化器，就可以使用 `auto`（参见 2.5.2 节，第 61 页）从此初始化器来推断我们想要分配的对象的类型。但是，由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用 `auto`：

```
auto p1 = new auto(obj);      // p 指向一个与 obj 类型相同的对象
                               // 该对象用 obj 进行初始化
auto p2 = new auto{a,b,c};    // 错误：括号中只能有单个初始化器
```

`p1` 的类型是一个指针，指向从 `obj` 自动推断出的类型。若 `obj` 是一个 `int`，那么 `p1` 就是 `int*`；若 `obj` 是一个 `string`，那么 `p1` 是一个 `string*`；依此类推。新分配的对象用 `obj` 的值进行初始化。

动态分配的 `const` 对象

用 `new` 分配 `const` 对象是合法的：

```
// 分配并初始化一个 const int
const int *pci = new const int(1024);
// 分配并默认初始化一个 const 的空 string
const string *pcs = new const string;
```

460 类似其他任何 `const` 对象，一个动态分配的 `const` 对象必须进行初始化。对于一个定义了默认构造函数（参见 7.1.4 节，第 236 页）的类类型，其 `const` 动态对象可以隐式初始化，而其他类型的对象就必须显式初始化。由于分配的对象是 `const` 的，`new` 返回的指针是一个指向 `const` 的指针（参见 2.4.2 节，第 56 页）。

内存耗尽

虽然现代计算机通常都配备大容量内存，但是自由空间被耗尽的情况还是有可能发生。一旦一个程序用光了它所有可用的内存，`new` 表达式就会失败。默认情况下，如果 `new` 不能分配所要求的内存空间，它会抛出一个类型为 `bad_alloc`（参见 5.6 节，第 173 页）的异常。我们可以改变使用 `new` 的方式来阻止它抛出异常：

```
// 如果分配失败，new 返回一个空指针
int *p1 = new int; // 如果分配失败，new 抛出 std::bad_alloc
int *p2 = new (nothrow) int; // 如果分配失败，new 返回一个空指针
```

我们称这种形式的 new 为定位 new (placement new)，其原因我们将在 19.1.2 节（第 729 页）中解释。定位 new 表达式允许我们向 new 传递额外的参数。在此例中，我们传递给它一个由标准库定义的名为 noexcept 的对象。如果将 noexcept 传递给 new，我们的意图是告诉它不能抛出异常。如果这种形式的 new 不能分配所需内存，它会返回一个空指针。bad_alloc 和 noexcept 都定义在头文件 new 中。

释放动态内存

为了防止内存耗尽，在动态内存使用完毕后，必须将其归还给系统。我们通过 delete 表达式（delete expression）来将动态内存归还给系统。delete 表达式接受一个指针，指向我们想要释放的对象：

```
delete p; // p 必须指向一个动态分配的对象或是一个空指针
```

与 new 类型类似，delete 表达式也执行两个动作：销毁给定的指针指向的对象；释放对应的内存。

指针值和 delete

我们传递给 delete 的指针必须指向动态分配的内存，或者是一个空指针（参见 2.3.2 节，第 48 页）。释放一块并非 new 分配的内存，或者将相同的指针值释放多次，其行为是未定义的：

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;      // 错误: i 不是一个指针
delete pi1;    // 未定义: pi1 指向一个局部变量
delete pd;     // 正确
delete pd2;    // 未定义: pd2 指向的内存已经被释放了
delete pi2;    // 正确: 释放一个空指针总是没有错误的
```

对于 delete i 的请求，编译器会生成一个错误信息，因为它知道 i 不是一个指针。执行 delete pi1 和 pd2 所产生的错误则更具潜在危害：通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些 delete 表达式，大多数编译器会编译通过，尽管它们是错误的。

虽然一个 const 对象的值不能被改变，但它本身是可以被销毁的。如同任何其他动态对象一样，想要释放一个 const 动态对象，只要 delete 指向它的指针即可：

```
const int *pci = new const int(1024);
delete pci; // 正确: 释放一个 const 对象
```

动态对象的生存期直到被释放时为止

如 12.1.1 节（第 402 页）所述，由 shared_ptr 管理的内存在线程最后一个 shared_ptr 销毁时会被自动释放。但对于通过内置指针类型来管理的内存，就不是这样了。对于一个由内置指针管理的动态对象，直到被显式释放之前它都是存在的。

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担——调用者必须记得释放内存：

```
// factory 返回一个指针，指向一个动态分配的对象
Foo* factory(T arg)
```

```

{
    // 视情况处理 arg
    return new Foo(arg); // 调用者负责释放此内存
}

```

类似我们之前定义的 `factory` 函数（参见 12.1.1 节，第 403 页），这个版本的 `factory` 分配一个对象，但并不 `delete` 它。`factory` 的调用者负责在不需要此对象时释放它。不幸的是，调用者经常忘记释放对象：

```

void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p 但不 delete 它
} // p 离开了它的作用域，但它所指向的内存没有被释放！

```

此处，`use_factory` 函数调用 `factory`，后者分配一个类型为 `Foo` 的新对象。当 `use_factory` 返回时，局部变量 `p` 被销毁。此变量是一个内置指针，而不是一个智能指针。

与类类型不同，内置类型的对象被销毁时什么也不会发生。特别是，当一个指针离开其作用域时，它所指向的对象什么也不会发生。如果这个指针指向的是动态内存，那么内存将不会被自动释放。



WARNING 由内置指针(而不是智能指针)管理的动态内存存在被显式释放前一直都会存在。

462>

在本例中，`p` 是指向 `factory` 分配的内存的唯一指针。一旦 `use_factory` 返回，程序就没有办法释放这块内存了。根据整个程序的逻辑，修正这个错误的正确方法是在 `use_factory` 中记得释放内存：

```

void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    delete p; // 现在记得释放内存，我们已经不需要它了
}

```

还有一种可能，我们的系统中的其他代码要使用 `use_factory` 所分配的对象，我们就应该修改此函数，让它返回一个指针，指向它分配的内存：

```

Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    return p; // 调用者必须释放内存
}

```

小心：动态内存的管理非常容易出错

使用 `new` 和 `delete` 管理动态内存存在三个常见问题：

1. 忘记 `delete` 内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内存泄露错误是非常困难的，因为通常应用程序运行很长一段时间后，真正耗尽内存时，才能检测到这种错误。
2. 使用已经释放掉的对象。通过在释放内存后将指针置为空，有时可以检测出这

种错误。

3. 同一块内存释放两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行了 `delete` 操作，对象的内存就被归还给自由空间了。如果我们随后又 `delete` 第二个指针，自由空间就可能被破坏。

相对于查找和修正这些错误来说，制造出这些错误要简单得多。

Best Practices

坚持只使用智能指针，就可以避免所有这些问题。对于一块内存，只有在没有任何智能指针指向它的情况下，智能指针才会自动释放它。

delete 之后重置指针值……

当我们 `delete` 一个指针后，指针值就变为无效了。虽然指针已经无效，但在很多机器上指针仍然保存着（已经释放了的）动态内存的地址。在 `delete` 之后，指针就变成了人们所说的空悬指针（dangling pointer），即，指向一块曾经保存数据对象但现在已经无效的内存的指针。◀ 463

未初始化指针（参见 2.3.2 节，第 49 页）的所有缺点空悬指针也都有。有一种方法可以避免空悬指针的问题：在指针即将要离开其作用域之前释放掉它所关联的内存。这样，在指针关联的内存被释放掉之后，就没有机会继续使用指针了。如果我们需要保留指针，可以在 `delete` 之后将 `nullptr` 赋予指针，这样就清楚地指出指针不指向任何对象。

……这只是提供了有限的保护

动态内存的一个基本问题是可能有多个指针指向相同的内存。在 `delete` 内存之后重置指针的方法只对这个指针有效，对其他任何仍指向（已释放的）内存的指针是没有作用的。例如：

```
int *p(new int(42)); // p 指向动态内存
auto q = p;          // p 和 q 指向相同的内存
delete p;            // p 和 q 均变为无效
p = nullptr;         // 指出 p 不再绑定到任何对象
```

本例中 `p` 和 `q` 指向相同的动态分配的对象。我们 `delete` 此内存，然后将 `p` 置为 `nullptr`，指出它不再指向任何对象。但是，重置 `p` 对 `q` 没有任何作用，在我们释放 `p` 所指向的（同时也是 `q` 所指向的！）内存时，`q` 也变为无效了。在实际系统中，查找指向相同内存的所有指针是异常困难的。

12.1.2 节练习

练习 12.6：编写函数，返回一个动态分配的 `int` 的 `vector`。将此 `vector` 传递给另一个函数，这个函数读取标准输入，将读入的值保存在 `vector` 元素中。再将 `vector` 传递给另一个函数，打印读入的值。记得在恰当的时刻 `delete vector`。

练习 12.7：重做上一题，这次使用 `shared_ptr` 而不是内置指针。

练习 12.8：下面的函数是否有错误？如果有，解释错误原因。

```
bool b() {
    int* p = new int;
    // ...
```

```

    return p;
}

```

练习 12.9：解释下面代码执行的结果：

```

int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;

```

464 12.1.3 shared_ptr 和 new 结合使用

如前所述，如果我们不初始化一个智能指针，它就会被初始化为一个空指针。如表 12.3 所示，我们还可以用 new 返回的指针来初始化智能指针：

```

shared_ptr<double> p1; // shared_ptr 可以指向一个 double
shared_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int

```

接受指针参数的智能指针构造函数是 *explicit* 的（参见 7.5.4 节，第 265 页）。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式（参见 3.2.1 节，第 76 页）来初始化一个智能指针：

```

shared_ptr<int> p1 = new int(1024); // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024)); // 正确：使用了直接初始化形式

```

p1 的初始化隐式地要求编译器用一个 new 返回的 int* 来创建一个 shared_ptr。由于我们不能进行内置指针到智能指针间的隐式转换，因此这条初始化语句是错误的。出于相同的原因，一个返回 shared_ptr 的函数不能在其返回语句中隐式转换一个普通指针：

```

shared_ptr<int> clone(int p) {
    return new int(p); // 错误：隐式转换为 shared_ptr<int>
}

```

我们必须将 shared_ptr 显式绑定到一个想要返回的指针上：

```

shared_ptr<int> clone(int p) {
    // 正确：显式地用 int* 创建 shared_ptr<int>
    return shared_ptr<int>(new int(p));
}

```

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象。我们可以将智能指针绑定到一个指向其他类型的资源的指针上，但是为了这样做，必须提供自己的操作来替代 delete。我们将在 12.1.4 节（第 415 页）介绍如何定义自己的释放操作。

表 12.3：定义和改变 shared_ptr 的其他方法

<code>shared_ptr<T> p(q)</code>	<code>p</code> 管理内置指针 <code>q</code> 所指向的对象； <code>q</code> 必须指向 new 分配的内存，且能够转换为 <code>T*</code> 类型
<code>shared_ptr<T> p(u)</code>	<code>p</code> 从 <code>unique_ptr u</code> 那里接管了对象的所有权；将 <code>u</code> 置为空
<code>shared_ptr<T> p(q, d)</code>	<code>p</code> 接管了内置指针 <code>q</code> 所指向的对象的所有权。 <code>q</code> 必须能转换为 <code>T*</code> 类型（参见 4.11.2 节，第 143 页）。 <code>p</code> 将使用可调用对象 <code>d</code> （参见 10.3.2 节，第 346 页）来代替 <code>delete</code>

续表

<code>shared_ptr<T> p(p2, d)</code>	如表 12.2 所示, p 是 shared_ptr p2 的拷贝, 唯一的区别是 p 将用可调用对象 d 来代替 delete
<code>p.reset()</code>	若 p 是唯一指向其对象的 shared_ptr, reset 会释放此对象。若传递了可选的参数内置指针 q, 会令 p 指向 q, 否则会将 p 置为空。若还传递了参数 d, 将会调用 d 而不是 delete 来释放 q
<code>p.reset(q)</code>	
<code>p.reset(q, d)</code>	

不要混合使用普通指针和智能指针……



`shared_ptr` 可以协调对象的析构, 但这仅限于其自身的拷贝 (也是 `shared_ptr`) 之间。这也是为什么我们推荐使用 `make_shared` 而不是 `new` 的原因。这样, 我们就能在分配对象的同时就将 `shared_ptr` 与之绑定, 从而避免了无意中将同一块内存绑定到多个独立创建的 `shared_ptr` 上。

考虑下面对 `shared_ptr` 进行操作的函数:

```
// 在函数被调用时 ptr 被创建并初始化
void process(shared_ptr<int> ptr)
{
    // 使用 ptr
} // ptr 离开作用域, 被销毁
```

`process` 的参数是传值方式传递的, 因此实参会 ◀ 465 被拷贝到 `ptr` 中。拷贝一个 `shared_ptr` 会递增其引用计数, 因此, 在 `process` 运行过程中, 引用计数值至少为 2。当 `process` 结束时, `ptr` 的引用计数会递减, 但不会变为 0。因此, 当局部变量 `ptr` 被销毁时, `ptr` 指向的内存不会被释放。

使用此函数的正确方法是传递给它一个 `shared_ptr`:

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
process(p); // 拷贝 p 会递增它的引用计数; 在 process 中引用计数值为 2
int i = *p; // 正确: 引用计数值为 1
```

虽然不能传递给 `process` 一个内置指针, 但可以传递给它一个 (临时的) `shared_ptr`, 这个 `shared_ptr` 是用一个内置指针显式构造的。但是, 这样做很可能会导致错误:

```
int *x(new int(1024));           // 危险: x 是一个普通指针, 不是一个智能指针
process(x); // 错误: 不能将 int* 转换为一个 shared_ptr<int>
process(shared_ptr<int>(x)); // 合法的, 但内存会被释放!
int j = *x; // 未定义的: x 是一个空悬指针!
```

在上面的调用中, 我们将一个临时 `shared_ptr` 传递给 `process`。当这个调用所在的表达式结束时, 这个临时对象就被销毁了。销毁这个临时变量会递减引用计数, 此时引用计数就变为 0 了。因此, 当临时对象被销毁时, 它所指向的内存会被释放。

但 `x` 继续指向 (已经释放的) 内存, 从而变成一个空悬指针。如果试图使用 `x` 的值, 其行为是未定义的。

当将一个 `shared_ptr` 绑定到一个普通指针时, 我们就将内存的管理责任交给了这个 `shared_ptr`。一旦这样做了, 我们就不应该再使用内置指针来访问 `shared_ptr` 所指向的内存了。

466



WARNING

使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁。

467

……也不要使用 get 初始化另一个智能指针或为智能指针赋值

智能指针类型定义了一个名为 `get` 的函数（参见表 12.1），它返回一个内置指针，指向智能指针管理的对象。此函数是为了这样一种情况而设计的：我们需要向不能使用智能指针的代码传递一个内置指针。使用 `get` 返回的指针的代码不能 `delete` 此指针。

虽然编译器不会给出错误信息，但将另一个智能指针也绑定到 `get` 返回的指针上是错误的：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
int *q = p.get(); // 正确：但使用 q 时要注意，不要让它管理的指针被释放
{ // 新程序块
    // 未定义：两个独立的 shared_ptr 指向相同的内存
    shared_ptr<int>(q);
} // 程序块结束，q 被销毁，它指向的内存被释放
int foo = *p; // 未定义：p 指向的内存已经被释放了
```

在本例中，`p` 和 `q` 指向相同的内存。由于它们是相互独立创建的，因此各自的引用计数都是 1。当 `q` 所在的程序块结束时，`q` 被销毁，这会导致 `q` 指向的内存被释放。从而 `p` 变成一个空悬指针，意味着当我们试图使用 `p` 时，将发生未定义的行为。而且，当 `p` 被销毁时，这块内存会被第二次 `delete`。



WARNING

`get` 用来将指针的访问权限传递给代码，你只有在确定代码不会 `delete` 指针的情况下，才能使用 `get`。特别是，永远不要用 `get` 初始化另一个智能指针或者为另一个智能指针赋值。

其他 `shared_ptr` 操作

`shared_ptr` 还定义了其他一些操作，参见表 12.2 和表 12.3 所示。我们可以用 `reset` 来将一个新的指针赋予一个 `shared_ptr`：

```
p = new int(1024);           // 错误：不能将一个指针赋予 shared_ptr
p.reset(new int(1024));     // 正确：p 指向一个新对象
```

与赋值类似，`reset` 会更新引用计数，如果需要的话，会释放 `p` 指向的对象。`reset` 成员经常与 `unique` 一起使用，来控制多个 `shared_ptr` 共享的对象。在改变底层对象之前，我们检查自己是否是当前对象仅有的用户。如果不是，在改变之前要制作一份新的拷贝：

```
if (!p.unique())
    p.reset(new string(*p)); // 我们不是唯一用户；分配新的拷贝
*p += newVal; // 现在我们知道我们是唯一的用户，可以改变对象的值
```

467

12.1.3 节练习

练习 12.10：下面的代码调用了第 413 页中定义的 `process` 函数，解释此调用是否正确。如果不正确，应如何修改？

```
shared_ptr<int> p(new int(42));
```

```
process(shared_ptr<int>(p));
```

练习 12.11: 如果我们像下面这样调用 process，会发生什么？

```
process(shared_ptr<int>(p.get()));
```

练习 12.12: p 和 q 的定义如下，对于接下来的对 process 的每个调用，如果合法，解释它做了什么，如果不合法，解释错误原因：

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

练习 12.13: 如果执行下面的代码，会发生什么？

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 智能指针和异常



5.6.2 节（第 175 页）中介绍了使用异常处理的程序能在异常发生后令程序流程继续，我们注意到，这种程序需要确保在异常发生后资源能被正确地释放。一个简单的确保资源被释放的方法是使用智能指针。

如果使用智能指针，即使程序块过早结束，智能指针类也能确保在内存不再需要时将其释放，：

```
void f()
{
    shared_ptr<int> sp(new int(42)); // 分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
} // 在函数结束时 shared_ptr 自动释放内存
```

函数的退出有两种可能，正常处理结束或者发生了异常，无论哪种情况，局部对象都会被销毁。在上面的程序中，sp 是一个 shared_ptr，因此 sp 销毁时会检查引用计数。在此例中，sp 是指向这块内存的唯一指针，因此内存会被释放掉。

与之相对的，当发生异常时，我们直接管理的内存是不会自动释放的。如果使用内置指针管理内存，且在 new 之后在对应的 delete 之前发生了异常，则内存不会被释放：

```
void f()
{
    int *ip = new int(42); // 动态分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
    delete ip;           // 在退出之前释放内存
}
```

468

如果在 new 和 delete 之间发生异常，且异常未在 f 中被捕获，则内存就永远不会被释放了。在函数 f 之外没有指针指向这块内存，因此就无法释放它了。



智能指针和哑类

包括所有标准库类在内的很多 C++ 类都定义了析构函数（参见 12.1.1 节，第 402 页），负责清理对象使用的资源。但是，不是所有的类都是这样良好定义的。特别是那些为 C 和 C++ 两种语言设计的类，通常都要求用户显式地释放所使用的任何资源。

那些分配了资源，而又没有定义析构函数来释放这些资源的类，可能会遇到与使用动态内存相同的错误——程序员非常容易忘记释放资源。类似的，如果在资源分配和释放之间发生了异常，程序也会发生资源泄漏。

与管理动态内存类似，我们通常可以使用类似的技术来管理不具有良好定义的析构函数的类。例如，假定我们正在使用一个 C 和 C++ 都使用的网络库，使用这个库的代码可能是这样的：

```
struct destination;           // 表示我们正在连接什么
struct connection;            // 使用连接所需的信息
connection connect(destination*); // 打开连接
void disconnect(connection);   // 关闭给定的连接
void f(destination &d /* 其他参数 */)
{
    // 获得一个连接；记住使用完后要关闭它
    connection c = connect(&d);
    // 使用连接
    // 如果我们在 f 退出前忘记调用 disconnect，就无法关闭 c 了
}
```

如果 `connection` 有一个析构函数，就可以在 `f` 结束时由析构函数自动关闭连接。但是，`connection` 没有析构函数。这个问题与我们上一个程序中使用 `shared_ptr` 避免内存泄漏几乎是等价的。使用 `shared_ptr` 来保证 `connection` 被正确关闭，已被证明是一种有效的方法。



使用我们自己的释放操作

469 默认情况下，`shared_ptr` 假定它们指向的是动态内存。因此，当一个 `shared_ptr` 被销毁时，它默认地对它管理的指针进行 `delete` 操作。为了用 `shared_ptr` 来管理一个 `connection`，我们必须首先定义一个函数来代替 `delete`。这个删除器（deleter）函数必须能够完成对 `shared_ptr` 中保存的指针进行释放的操作。在本例中，我们的删除器必须接受单个类型为 `connection*` 的参数：

```
void end_connection(connection *p) { disconnect(*p); }
```

当我们创建一个 `shared_ptr` 时，可以传递一个（可选的）指向删除器函数的参数（参见 6.7 节，第 221 页）：

```
void f(destination &d /* 其他参数 */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由异常而退出），connection 会被正确关闭
}
```

当 p 被销毁时，它不会对自己保存的指针执行 delete，而是调用 end_connection。接下来，end_connection 会调用 disconnect，从而确保连接被关闭。如果 f 正常退出，那么 p 的销毁会作为结束处理的一部分。如果发生了异常，p 同样会被销毁，从而连接被关闭。

注意：智能指针陷阱

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范：

- 不使用相同的内置指针值初始化（或 reset）多个智能指针。
- 不 delete get() 返回的指针。
- 不使用 get() 初始化或 reset 另一个智能指针。
- 如果你使用 get() 返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。
- 如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器（参见 12.1.4 节，第 415 页和 12.1.5 节，第 419 页）。

12.1.4 节练习

练习 12.14：编写你自己版本的用 shared_ptr 管理 connection 的函数。

练习 12.15：重写第一题的程序，用 lambda（参见 10.3.2 节，第 346 页）代替 end_connection 函数。

12.1.5 unique_ptr

< 470

一个 unique_ptr “拥有” 它所指向的对象。与 shared_ptr 不同，某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。表 12.4 列出了 unique_ptr 特有的操作。与 shared_ptr 相同的操作列在表 12.1（第 401 页）中。

C++
11

与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr。当我们定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。类似 shared_ptr，初始化 unique_ptr 必须采用直接初始化形式：

```
unique_ptr<double> p1; // 可以指向一个 double 的 unique_ptr
unique_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作：

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);    // 错误：unique_ptr 不支持拷贝
unique_ptr<string> p3;
p3 = p2;                    // 错误：unique_ptr 不支持赋值
```

表 12.4: unique_ptr 操作 (另参见表 12.1, 第 401 页)	
unique_ptr<T> u1	空 unique_ptr, 可以指向类型为 T 的对象。u1 会使用 delete 来释放它的指针;
unique_ptr<T, D> u2	u2 会使用一个类型为 D 的可调用对象来释放它的指针
unique_ptr<T, D> u(d)	空 unique_ptr, 指向类型为 T 的对象, 用类型为 D 的对象 d 替代 delete
u = nullptr	释放 u 指向的对象, 将 u 置为空
u.release()	u 放弃对指针的控制权, 返回指针, 并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q)	如果提供了内置指针 q, 令 u 指向这个对象; 否则将 u 置为空
u.reset(nullptr)	

虽然我们不能拷贝或赋值 unique_ptr, 但可以通过调用 release 或 reset 将指针的所有权从一个 (非 const) unique_ptr 转移给另一个 unique:

```
// 将所有权从 p1 (指向 string Stegosaurus) 转移给 p2
unique_ptr<string> p2(p1.release()); // release 将 p1 置为空
unique_ptr<string> p3(new string("Trex"));
// 将所有权从 p3 转移给 p2
p2.reset(p3.release()); // reset 释放了 p2 原来指向的内存
```

release 成员返回 unique_ptr 当前保存的指针并将其置为空。因此, p2 被初始化为 p1 原来保存的指针, 而 p1 被置为空。

471 > reset 成员接受一个可选的指针参数, 令 unique_ptr 重新指向给定的指针。如果 unique_ptr 不为空, 它原来指向的对象被释放。因此, 对 p2 调用 reset 释放了用 "Stegosaurus" 初始化的 string 所使用的内存, 将 p3 对指针的所有权转移给 p2, 并将 p3 置为空。

调用 release 会切断 unique_ptr 和它原来管理的对象间的联系。release 返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。在本例中, 管理内存的责任简单地从一个智能指针转移给另一个。但是, 如果我们不用另一个智能指针来保存 release 返回的指针, 我们的程序就要负责资源的释放:

```
p2.release(); // 错误: p2 不会释放内存, 而且我们丢失了指针
auto p = p2.release(); // 正确, 但我们必须记得 delete(p)
```

传递 unique_ptr 参数和返回 unique_ptr

不能拷贝 unique_ptr 的规则有一个例外: 我们可以拷贝或赋值一个将要被销毁的 unique_ptr。最常见的例子是从函数返回一个 unique_ptr:

```
unique_ptr<int> clone(int p) {
    // 正确: 从 int* 创建一个 unique_ptr<int>
    return unique_ptr<int>(new int(p));
}
```

还可以返回一个局部对象的拷贝:

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

对于两段代码，编译器都知道要返回的对象将要被销毁。在此情况下，编译器执行一种特殊的“拷贝”，我们将在 13.6.2 节（第 473 页）中介绍它。

向后兼容：auto_ptr

标准库的较早版本包含了一个名为 `auto_ptr` 的类，它具有 `unique_ptr` 的部分特性，但不是全部。特别是，我们不能在容器中保存 `auto_ptr`，也不能从函数中返回 `auto_ptr`。

虽然 `auto_ptr` 仍是标准库的一部分，但编写程序时应该使用 `unique_ptr`。

向 unique_ptr 传递删除器

类似 `shared_ptr`, `unique_ptr` 默认情况下用 `delete` 释放它指向的对象。与 `shared_ptr` 一样，我们可以重载一个 `unique_ptr` 中默认的删除器（参见 12.1.4 节，第 415 页）。但是，`unique_ptr` 管理删除器的方式与 `shared_ptr` 不同，其原因我们将在 16.1.6 节（第 599 页）中介绍。

< 472

重载一个 `unique_ptr` 中的删除器会影响到 `unique_ptr` 类型以及如何构造（或 `reset`）该类型的对象。与重载关联容器的比较操作（参见 11.2.2 节，第 378 页）类似，我们必须在尖括号中 `unique_ptr` 指向类型之后提供删除器类型。在创建或 `reset` 一个这种 `unique_ptr` 类型的对象时，必须提供一个指定类型的可调用对象（删除器）：

```
// p 指向一个类型为 objT 的对象，并使用一个类型为 delT 的对象释放 objT 对象
// 它会调用一个名为 fcn 的 delT 类型对象
unique_ptr<objT, delT> p (new objT, fcn);
```

作为一个更具体的例子，我们将重写连接程序，用 `unique_ptr` 来代替 `shared_ptr`，如下所示：

```
void f(destination &d /* 其他需要的参数 */)
{
    connection c = connect(&d); // 打开连接
    // 当 p 被销毁时，连接将会关闭
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

在本例中我们使用了 `decltype`（参见 2.5.3 节，第 62 页）来指明函数指针类型。由于 `decltype(end_connection)` 返回一个函数类型，所以我们必须添加一个*来指出我们正在使用该类型的一个指针（参见 6.7 节，第 223 页）。

12.1.5 节练习

练习 12.16：如果你试图拷贝或赋值 `unique_ptr`，编译器并不总是能给出易于理解的错误信息。编写包含这种错误的程序，观察编译器如何诊断这种错误。

练习 12.17：下面的 `unique_ptr` 声明中，哪些是合法的，哪些可能导致后续的程序错误？解释每个错误的问题在哪里。

```

int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix);           (b) IntP p1(pi);
(c) IntP p2(pi2);         (d) IntP p3(&ix);
(e) IntP p4(new int(2048)); (f) IntP p5(p2.get());

```

练习 12.18: `shared_ptr` 为什么没有 `release` 成员?



12.1.6 weak_ptr

473

C++
11

`weak_ptr` (见表 12.5) 是一种不控制所指向对象生存期的智能指针, 它指向由一个 `shared_ptr` 管理的对象。将一个 `weak_ptr` 绑定到一个 `shared_ptr` 不会改变 `shared_ptr` 的引用计数。一旦最后一个指向对象的 `shared_ptr` 被销毁, 对象就会被释放。即使有 `weak_ptr` 指向对象, 对象也还是会被释放, 因此, `weak_ptr` 的名字抓住了这种智能指针“弱”共享对象的特点。

表 12.5: `weak_ptr`

<code>weak_ptr<T> w</code>	空 <code>weak_ptr</code> 可以指向类型为 <code>T</code> 的对象
<code>weak_ptr<T> w(sp)</code>	与 <code>shared_ptr</code> <code>sp</code> 指向相同对象的 <code>weak_ptr</code> 。 <code>T</code> 必须能转换为 <code>sp</code> 指向的类型
<code>w = p</code>	<code>p</code> 可以是一个 <code>shared_ptr</code> 或一个 <code>weak_ptr</code> 。赋值后 <code>w</code> 与 <code>p</code> 共享对象
<code>w.reset()</code>	将 <code>w</code> 置为空
<code>w.use_count()</code>	与 <code>w</code> 共享对象的 <code>shared_ptr</code> 的数量
<code>w.expired()</code>	若 <code>w.use_count()</code> 为 0, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>w.lock()</code>	如果 <code>expired</code> 为 <code>true</code> , 返回一个空 <code>shared_ptr</code> ; 否则返回一个指向 <code>w</code> 的对象的 <code>shared_ptr</code>

当我们创建一个 `weak_ptr` 时, 要用一个 `shared_ptr` 来初始化它:

```

auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp 弱共享 p; p 的引用计数未改变

```

本例中 `wp` 和 `p` 指向相同的对象。由于是弱共享, 创建 `wp` 不会改变 `p` 的引用计数; `wp` 指向的对象可能被释放掉。

由于对象可能存在, 我们不能使用 `weak_ptr` 直接访问对象, 而必须调用 `lock`。此函数检查 `weak_ptr` 指向的对象是否仍存在。如果存在, `lock` 返回一个指向共享对象的 `shared_ptr`。与任何其他 `shared_ptr` 类似, 只要此 `shared_ptr` 存在, 它所指向的底层对象也就会一直存在。例如:

```

if (shared_ptr<int> np = wp.lock()) { // 如果 np 不为空则条件成立
    // 在 if 中, np 与 p 共享对象
}

```

在这段代码中, 只有当 `lock` 调用返回 `true` 时我们才会进入 `if` 语句体。在 `if` 中, 使用 `np` 访问共享对象是安全的。

核查指针类

作为 `weak_ptr` 用途的一个展示, 我们将为 `StrBlob` 类定义一个伴随指针类。我们

的指针类将命名为 StrBlobPtr，会保存一个 `weak_ptr`，指向 `StrBlob` 的 `data` 成员，这是初始化时提供给它的。通过使用 `weak_ptr`，不会影响一个给定的 `StrBlob` 所指向的 `vector` 的生存期。但是，可以阻止用户访问一个不再存在的 `vector` 的企图。 ◀ 474

`StrBlobPtr` 会有两个数据成员：`wptr`，或者为空，或者指向一个 `StrBlob` 中的 `vector`；`curr`，保存当前对象所表示的元素的下标。类似它的伴随类 `StrBlob`，我们的指针类也有一个 `check` 成员来检查解引用 `StrBlobPtr` 是否安全：

```
// 对于访问一个不存在元素的尝试，StrBlobPtr 抛出一个异常
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
        wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr(); // 前缀递增
private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr，意味着底层 vector 可能会被销毁
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr; // 在数组中的当前位置
};
```

默认构造函数生成一个空的 `StrBlobPtr`。其构造函数初始化列表（参见 7.1.4 节，第 237 页）将 `curr` 显式初始化为 0，并将 `wptr` 隐式初始化为一个空 `weak_ptr`。第二个构造函数接受一个 `StrBlob` 引用和一个可选的索引值。此构造函数初始化 `wptr`，令其指向给定 `StrBlob` 对象的 `shared_ptr` 中的 `vector`，并将 `curr` 初始化为 `sz` 的值。我们使用了默认参数（参见 6.5.1 节，第 211 页），表示默认情况下将 `curr` 初始化为第一个元素的下标。我们将会看到，`StrBlob` 的 `end` 成员将会用到参数 `sz`。

值得注意的是，我们不能将 `StrBlobPtr` 绑定到一个 `const StrBlob` 对象。这个限制是由于构造函数接受一个非 `const StrBlob` 对象的引用而导致的。

`StrBlobPtr` 的 `check` 成员与 `StrBlob` 中的同名成员不同，它还要检查指针指向的 `vector` 是否还存在：

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗？
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // 否则，返回指向 vector 的 shared_ptr
}
```

由于一个 `weak_ptr` 不参与其对应的 `shared_ptr` 的引用计数，`StrBlobPtr` 指向的 `vector` 可能已经被释放了。如果 `vector` 已销毁，`lock` 将返回一个空指针。在本例中，任何 `vector` 的引用都会失败，于是抛出一个异常。否则，`check` 会检查给定索引，如果索引合法，`check` 返回从 `lock` 获得的 `shared_ptr`。 ◀ 475

指针操作

我们将在第 14 章学习如何定义自己的运算符。现在，我们将定义名为 deref 和 incr 的函数，分别用来解引用和递增 StrBlobPtr。

deref 成员调用 check，检查使用 vector 是否安全以及 curr 是否在合法范围内：

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) 是对象所指向的 vector
}
```

如果 check 成功，p 就是一个 shared_ptr，指向 StrBlobPtr 所指向的 vector。表达式 (*p)[curr] 解引用 shared_ptr 来获得 vector，然后使用下标运算符提取并返回 curr 位置上的元素。

incr 成员也调用 check：

```
// 前缀递增：返回递增后的对象的引用
StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置，就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 推进当前位置
    return *this;
}
```

当然，为了访问 data 成员，我们的指针类必须声明为 StrBlob 的 friend（参见 7.3.4 节，第 250 页）。我们还要为 StrBlob 类定义 begin 和 end 操作，返回一个指向它自身的 StrBlobPtr：

```
// 对于 StrBlob 中的友元声明来说，此前置声明是必要的
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // 其他成员与 12.1.1 节（第 405 页）中声明相同
    // 返回指向首元素和尾后元素的 StrBlobPtr
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
    { auto ret = StrBlobPtr(*this, data->size());
      return ret; }
};
```

12.1.6 节练习

练习 12.19： 定义你自己版本的 StrBlobPtr，更新 StrBlob 类，加入恰当的 friend 声明及 begin 和 end 成员。

练习 12.20： 编写程序，逐行读入一个输入文件，将内容存入一个 StrBlob 中，用一个 StrBlobPtr 打印出 StrBlob 中的每个元素。

练习 12.21： 也可以这样编写 StrBlobPtr 的 deref 成员：

```
std::string& deref() const
```

```
{ return (*check(curr, "dereference past end))[curr]; }
```

你认为哪个版本更好？为什么？

练习 12.22: 为了能让 `StrBlobPtr` 使用 `const StrBlob`, 你觉得应该如何修改？定义一个名为 `ConstStrBlobPtr` 的类，使其能够指向 `const StrBlob`。



12.2 动态数组

`new` 和 `delete` 运算符一次分配/释放一个对象，但某些应用需要一次为很多对象分配内存的功能。例如，`vector` 和 `string` 都是在连续内存中保存它们的元素，因此，当容器需要重新分配内存时（参见 9.4 节，第 317 页），必须一次性为很多元素分配内存。

为了支持这种需求，C++语言和标准库提供了两种一次分配一个对象数组的方法。C++语言定义了另一种 `new` 表达式语法，可以分配并初始化一个对象数组。标准库中包含一个名为 `allocator` 的类，允许我们将分配和初始化分离。使用 `allocator` 通常会提供更好的性能和更灵活的内存管理能力，原因我们将在 12.2.2 节（第 427 页）中解释。

很多（可能是大多数）应用都没有直接访问动态数组的需求。当一个应用需要可变数量的对象时，我们在 `StrBlob` 中所采用的方法几乎总是更简单、更快速并且更安全的——即，使用 `vector`（或其他标准库容器）。如我们将在 13.6 节（第 470 页）中看到的，使用标准库容器的优势在新标准下更为显著。在支持新标准的标准库中，容器操作比之前的版本要快速得多。

Best Practices

大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单、更不容易出现内存管理错误并且可能有更好的性能。

如前所述，使用容器的类可以使用默认版本的拷贝、赋值和析构操作（参见 7.1.5 节，第 239 页）。分配动态数组的类则必须定义自己版本的操作，在拷贝、复制以及销毁对象时管理所关联的内存。



直到学习完第 13 章，不要在类内的代码中分配动态内存。



12.2.1 new 和数组

477

为了让 `new` 分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象的数目。在下例中，`new` 分配要求数量的对象并（假定分配成功后）返回指向第一个对象的指针：

```
// 调用 get_size 确定分配多少个 int
int *pia = new int[get_size()]; // pia 指向第一个 int
```

方括号中的大小必须是整型，但不必是常量。

也可以用一个表示数组类型的类型别名（参见 2.5.1 节，第 60 页）来分配一个数组，这样，`new` 表达式中就不需要方括号了：

```
typedef int arrT[42];      // arrT 表示 42 个 int 的数组类型
int *p = new arrT;         // 分配一个 42 个 int 的数组；p 指向第一个 int
```

在本例中，`new` 分配一个 `int` 数组，并返回指向第一个 `int` 的指针。即使这段代码中没

有方括号，编译器执行这个表达式时还是会用 `new[]`。即，编译器执行如下形式：

```
int *p = new int[42];
```

分配一个数组会得到一个元素类型的指针

虽然我们通常称 `new T[]` 分配的内存为“动态数组”，但这种叫法某种程度上有些误导。当用 `new` 分配一个数组时，我们并未得到一个数组类型的对象，而是得到一个数组元素类型的指针。即使我们使用类型别名定义了一个数组类型，`new` 也不会分配一个数组类型的对象。在上例中，我们正在分配一个数组的事实甚至都是不可见的——连 `[num]` 都没有。`new` 返回的是一个元素类型的指针。

由于分配的内存并不是一个数组类型，因此不能对动态数组调用 `begin` 或 `end`（参见 3.5.3 节，第 106 页）。这些函数使用数组维度（回忆一下，维度是数组类型的一部分）来返回指向首元素和尾后元素的指针。出于相同的原因，也不能用范围 `for` 语句来处理（所谓的）动态数组中的元素。



WARNING

要记住我们所说的动态数组并不是数组类型，这是很重要的。

初始化动态分配对象的数组

默认情况下，`new` 分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化（参见 3.3.1 节，第 88 页），方法是在大小之后跟一对空括号。

```
int *pia = new int[10];           // 10 个未初始化的 int
int *pia2 = new int[10]();        // 10 个值初始化为 0 的 int
string *psa = new string[10];     // 10 个空 string
string *psa2 = new string[10]();   // 10 个空 string
```

478 在新标准中，我们还可以提供一个元素初始化器的花括号列表：

```
// 10 个 int 分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// 10 个 string，前 4 个用给定的初始化器初始化，剩余的进行值初始化
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

与内置数组对象的列表初始化（参见 3.5.1 节，第 102 页）一样，初始化器会用来初始化动态数组中开始部分的元素。如果初始化器数目小于元素数目，剩余元素将进行值初始化。如果初始化器数目大于元素数目，则 `new` 表达式失败，不会分配任何内存。在本例中，`new` 会抛出一个类型为 `bad_array_new_length` 的异常。类似 `bad_alloc`，此类型定义在头文件 `new` 中。

C++ 11 虽然我们用空括号对数组中元素进行值初始化，但不能在括号中给出初始化器，这意味着不能用 `auto` 分配数组（参见 12.1.2 节，第 407 页）。

动态分配一个空数组是合法的

可以用任意表达式来确定要分配的对象的数目：

```
size_t n = get_size();    // get_size 返回需要的元素的数目
int* p = new int[n];      // 分配数组保存元素
for (int* q = p; q != p + n; ++q)
/* 处理数组 */;
```

这产生了一个有意思的问题：如果 `get_size` 返回 0，会发生什么？答案是代码仍能正常工作。虽然我们不能创建一个大小为 0 的静态数组对象，但当 `n` 等于 0 时，调用 `new[n]` 是合法的：

```
char arr[0];           // 错误：不能定义长度为 0 的数组
char *cp = new char[0]; // 正确：但 cp 不能解引用
```

当我们用 `new` 分配一个大小为 0 的数组时，`new` 返回一个合法的非空指针。此指针保证与 `new` 返回的其他任何指针都不相同。对于零长度的数组来说，此指针就像尾后指针一样（参见 3.5.3 节，第 106 页），我们可以像使用尾后迭代器一样使用这个指针。可以用此指针进行比较操作，就像上面循环代码中那样。可以向此指针加上（或从此指针减去）0，也可以从此指针减去自身从而得到 0。但此指针不能解引用——毕竟它不指向任何元素。

在我们假想的循环中，若 `get_size` 返回 0，则 `n` 也是 0，`new` 会分配 0 个对象。`for` 循环中的条件会失败（`p` 等于 `q+n`，因为 `n` 为 0）。因此，循环体不会被执行。

释放动态数组

为了释放动态数组，我们使用一种特殊形式的 `delete`——在指针前加上一个空方括号对：

```
delete p;           // p 必须指向一个动态分配的对象或为空
delete [] pa;      // pa 必须指向一个动态分配的数组或为空
```

< 479

第二条语句销毁 `pa` 指向的数组中的元素，并释放对应的内存。数组中的元素按逆序销毁，即，最后一个元素首先被销毁，然后是倒数第二个，依此类推。

当我们释放一个指向数组的指针时，空方括号对是必需的：它指示编译器此指针指向一个对象数组的第一个元素。如果我们在 `delete` 一个指向数组的指针时忽略了方括号（或者在 `delete` 一个指向单一对象的指针时使用了方括号），其行为是未定义的。

回忆一下，当我们使用一个类型别名来定义一个数组类型时，在 `new` 表达式中不使用 `[]`。即使这样，在释放一个数组指针时也必须使用方括号：

```
typedef int arrT[42];    // arrT 是 42 个 int 的数组的类型别名
int *p = new arrT;       // 分配一个 42 个 int 的数组；p 指向第一个元素
delete [] p;             // 方括号是必需的，因为我们当初分配的是一个数组
```

不管外表如何，`p` 指向一个对象数组的首元素，而不是一个类型为 `arrT` 的单一对象。因此，在释放 `p` 时我们必须使用 `[]`。



如果我们在 `delete` 一个数组指针时忘记了方括号，或者在 `delete` 一个单一对象的指针时使用了方括号，编译器很可能不会给出警告。我们的程序可能在执行过程中在没有任何警告的情况下行为异常。

智能指针和动态数组

标准库提供了一个可以管理 `new` 分配的数组的 `unique_ptr` 版本。为了用一个 `unique_ptr` 管理动态数组，我们必须在对象类型后面跟一对空方括号：

```
// up 指向一个包含 10 个未初始化 int 的数组
unique_ptr<int[]> up(new int[10]);
up.release(); // 自动用 delete[] 销毁其指针
```

类型说明符中的方括号 (`<int[]>`) 指出 `up` 指向一个 `int` 数组而不是一个 `int`。由于 `up` 指向一个数组，当 `up` 销毁它管理的指针时，会自动使用 `delete[]`。

指向数组的 `unique_ptr` 提供的操作与我们在 12.1.5 节（第 417 页）中使用的那些操作有一些不同，我们在表 12.6 中描述了这些操作。当一个 `unique_ptr` 指向一个数组时，我们不能使用点和箭头成员运算符。毕竟 `unique_ptr` 指向的是一个数组而不是单个对象，因此这些运算符是无意义的。另一方面，当一个 `unique_ptr` 指向一个数组时，我们可以使用下标运算符来访问数组中的元素：

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i; // 为每个元素赋予一个新值
```

480 >

表 12.6：指向数组的 `unique_ptr`

指向数组的 `unique_ptr` 不支持成员访问运算符（点和箭头运算符）。

其他 `unique_ptr` 操作不变。

`unique_ptr<T[]> u` `u` 可以指向一个动态分配的数组，数组元素类型为 `T`

`unique_ptr<T[]> u(p)` `u` 指向内置指针 `p` 所指向的动态分配的数组。`p` 必须能转换为类型 `T*`（参见 4.11.2 节，第 143 页）

`u[i]` 返回 `u` 拥有的数组中位置 `i` 处的对象

`u` 必须指向一个数组

与 `unique_ptr` 不同，`shared_ptr` 不直接支持管理动态数组。如果希望使用 `shared_ptr` 管理一个动态数组，必须提供自己定义的删除器：

```
// 为了使用 shared_ptr，必须提供一个删除器
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // 使用我们提供的 lambda 释放数组，它使用 delete[]
```

本例中我们传递给 `shared_ptr` 一个 `lambda`（参见 10.3.2 节，第 346 页）作为删除器，它使用 `delete[]` 释放数组。

如果未提供删除器，这段代码将是未定义的。默认情况下，`shared_ptr` 使用 `delete` 销毁它指向的对象。如果此对象是一个动态数组，对其使用 `delete` 所产生的问题与释放一个动态数组指针时忘记 `[]` 产生的问题一样（参见 12.2.1 节，第 425 页）。

`shared_ptr` 不直接支持动态数组管理这一特性会影响我们如何访问数组中的元素：

```
// shared_ptr 未定义下标运算符，并且不支持指针的算术运算
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // 使用 get 获取一个内置指针
```

`shared_ptr` 未定义下标运算符，而且智能指针类型不支持指针算术运算。因此，为了访问数组中的元素，必须用 `get` 获取一个内置指针，然后用它来访问数组元素。

12.2.1 节练习

练习 12.23： 编写一个程序，连接两个字符串字面常量，将结果保存在一个动态分配的 `char` 数组中。重写这个程序，连接两个标准库 `string` 对象。

练习 12.24： 编写一个程序，从标准输入读取一个字符串，存入一个动态分配的字符数组中。描述你的程序如何处理变长输入。测试你的程序，输入一个超出你分配的数组长度的字符串。

练习 12.25：给定下面的 new 表达式，你应该如何释放 pa？

```
int *pa = new int[10];
```



12.2.2 allocator 类

<481

new 有一些灵活性上的局限，其中一方面表现在它将内存分配和对象构造组合在了一起。类似的，delete 将对象析构和内存释放组合在了一起。我们分配单个对象时，通常希望将内存分配和对象初始化组合在一起。因为在这种情况下，我们几乎肯定知道对象应有什么值。

当分配一大块内存时，我们通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。

一般情况下，将内存分配和对象构造组合在一起可能会导致不必要的浪费。例如：

```
string *const p = new string[n]; // 构造 n 个空 string
string s;
string *q = p; // q 指向第一个 string
while (cin >> s && q != p + n)
    *q++ = s; // 赋予*q 一个新值
const size_t size = q - p; // 记住我们读取了多少个 string
// 使用数组
delete[] p; // p 指向一个数组；记得用 delete[] 来释放
```

new 表达式分配并初始化了 n 个 string。但是，我们可能不需要 n 个 string，少量 string 可能就足够了。这样，我们就可能创建了一些永远也用不到的对象。而且，对于那些确实要使用的对象，我们也在初始化之后立即赋予了它们新值。每个使用到的元素都被赋值了两次：第一次是在默认初始化时，随后是在赋值时。

更重要的是，那些没有默认构造函数的类就不能动态分配数组了。

allocator 类

标准库 **allocator** 类定义在头文件 `memory` 中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。表 12.7 概述了 **allocator** 支持的操作。在本节中，我们将介绍这些 **allocator** 操作。在 13.5 节（第 464 页），我们将看到如何使用这个类的典型例子。

类似 `vector`，`allocator` 是一个模板（参见 3.3 节，第 86 页）。为了定义一个 `allocator` 对象，我们必须指明这个 `allocator` 可以分配的对象类型。当一个 `allocator` 对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc; // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n); // 分配 n 个未初始化的 string
```

这个 `allocate` 调用为 n 个 string 分配了内存。

482

表 12.7：标准库 allocator 类及其算法

allocator<T> a	定义了一个名为 a 的 allocator 对象，它可以为类型为 T 的对象分配内存
a.allocate(n)	分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象
a.deallocate(p, n)	释放从 T* 指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。在调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy
a.construct(p, args)	p 必须是一个类型为 T* 的指针，指向一块原始内存；arg 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象
a.destroy(p)	p 为 T* 类型的指针，此算法对 p 指向的对象执行析构函数（参见 12.1.1 节，第 402 页）

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。我们按需要在此内存中构造对象。在新标准库中，construct 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。类似 make_shared 的参数（参见 12.1.1 节，第 401 页），这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器：

```
auto q = p; // q 指向最后构造的元素之后的位置
alloc.construct(q++); // *q 为空字符串
alloc.construct(q++, 10, 'c'); // *q 为 cccccccccc
alloc.construct(q++, "hi"); // *q 为 hi!
```

在早期版本的标准库中，construct 只接受两个参数：指向创建对象位置的指针和一个元素类型的值。因此，我们只能将一个元素拷贝到未构造空间中，而不能用元素类型的任何其他构造函数来构造一个元素。

还未构造对象的情况下就使用原始内存是错误的：

```
cout << *p << endl; // 正确：使用 string 的输出运算符
cout << *q << endl; // 灾难：q 指向未构造的内存！
```



为了使用 allocate 返回的内存，我们必须用 construct 构造对象。使用未构造的内存，其行为是未定义的。

当我们用完对象后，必须对每个构造的元素调用 destroy 来销毁它们。函数 destroy 接受一个指针，对指向的对象执行析构函数（参见 12.1.1 节，第 402 页）：

```
483 while (q != p)
        alloc.destroy(--q); // 释放我们真正构造的 string
```

在循环开始处，q 指向最后构造的元素之后的位置。我们在调用 destroy 之前对 q 进行了递减操作。因此，第一次调用 destroy 时，q 指向最后一个构造的元素。最后一步循环中我们 destroy 了第一个构造的元素，随后 q 将与 p 相等，循环结束。



我们只能对真正构造了的元素进行 destroy 操作。

一旦元素被销毁后，就可以重新使用这部分内存来保存其他 string，也可以将其归

还给系统。释放内存通过调用 `deallocate` 来完成：

```
alloc.deallocate(p, n);
```

我们传递给 `deallocate` 的指针不能为空，它必须指向由 `allocate` 分配的内存。而且，传递给 `deallocate` 的大小参数必须与调用 `allocate` 分配内存时提供的大小参数具有一样的值。

拷贝和填充未初始化内存的算法

标准库还为 `allocator` 类定义了两个伴随算法，可以在未初始化内存中创建对象。表 12.8 描述了这些函数，它们都定义在头文件 `memory` 中。

表 12.8: `allocator` 算法

这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。

<code>uninitialized_copy(b, e, b2)</code>	从迭代器 <code>b</code> 和 <code>e</code> 指出的输入范围内拷贝元素到迭代器 <code>b2</code> 指定的未构造的原始内存中。 <code>b2</code> 指向的内存必须足够大，能容纳输入序列中元素的拷贝
<code>uninitialized_copy_n(b, n, b2)</code>	从迭代器 <code>b</code> 指向的元素开始，拷贝 <code>n</code> 个元素到 <code>b2</code> 开始的内存中
<code>uninitialized_fill(b, e, t)</code>	在迭代器 <code>b</code> 和 <code>e</code> 指定的原始内存范围内创建对象，对象的值均为 <code>t</code> 的拷贝
<code>uninitialized_fill_n(b, n, t)</code>	从迭代器 <code>b</code> 指向的内存地址开始创建 <code>n</code> 个对象。 <code>b</code> 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

作为一个例子，假定有一个 `int` 的 `vector`，希望将其内容拷贝到动态内存中。我们将分配一块比 `vector` 中元素所占用空间大一倍的动态内存，然后将原 `vector` 中的元素拷贝到前一半空间，对后一半空间用一个给定值进行填充：

```
// 分配比 vi 中元素所占用空间大一倍的动态内存
auto p = alloc.allocate(vi.size() * 2);
// 通过拷贝 vi 中的元素来构造从 p 开始的元素
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// 将剩余元素初始化为 42
uninitialized_fill_n(q, vi.size(), 42);
```

484

类似拷贝算法（参见 10.2.2 节，第 341 页），`uninitialized_copy` 接受三个迭代器参数。前两个表示输入序列，第三个表示这些元素将要拷贝到的目的空间。传递给 `uninitialized_copy` 的目的位置迭代器必须指向未构造的内存。与 `copy` 不同，`uninitialized_copy` 在给定目的位置构造元素。

类似 `copy`，`uninitialized_copy` 返回（递增后的）目的位置迭代器。因此，一次 `uninitialized_copy` 调用会返回一个指针，指向最后一个构造的元素之后的位置。在本例中，我们将此指针保存在 `q` 中，然后将 `q` 传递给 `uninitialized_fill_n`。此函数类似 `fill_n`（参见 10.2.2 节，第 340 页），接受一个指向目的位置的指针、一个计数和一个值。它会在目的位置指针指向的内存中创建给定数目个对象，用给定值对它们进行初始化。

12.2.2 节练习

练习 12.26: 用 allocator 重写第 427 页中的程序。



12.3 使用标准库: 文本查询程序

我们将实现一个简单的文本查询程序, 作为标准库相关内容学习的总结。我们的程序允许用户在一个给定文件中查询单词。查询结果是单词在文件中出现的次数及其所在行的列表。如果一个单词在一行中出现多次, 此行只列出一次。行会按照升序输出——即, 第 7 行会在第 9 行之前显示, 依此类推。

例如, 我们可能读入一个包含本章内容 (指英文版中的文本) 的文件, 在其中寻找单词 element。输出结果的前几行应该是这样的:

```
element occurs 112 times
  (line 36) A set element contains only a key;
  (line 158) operator creates a new element
  (line 160) Regardless of whether the element
  (line 168) When we fetch an element from a map, we
  (line 214) If the element is not found, find returns
```

接下来还有大约 100 行, 都是单词 element 出现的位置。



12.3.1 文本查询程序设计

485

开始一个程序的设计的一种好方法是列出程序的操作。了解需要哪些操作会帮助我们分析出需要什么样的数据结构。从需求入手, 我们的文本查询程序需要完成如下任务:

- 当程序读取输入文件时, 它必须记住单词出现的每一行。因此, 程序需要逐行读取输入文件, 并将每一行分解为独立的单词
- 当程序生成输出时,
 - 它必须能提取每个单词所关联的行号
 - 行号必须按升序出现且无重复
 - 它必须能打印给定行号中的文本。

利用多种标准库设施, 我们可以很漂亮地实现这些要求:

- 我们将使用一个 `vector<string>` 来保存整个输入文件的一份拷贝。输入文件中的每行保存为 `vector` 中的一个元素。当需要打印一行时, 可以用行号作为下标来提取行文本。
- 我们使用一个 `istringstream` (参见 8.3 节, 第 287 页) 来将每行分解为单词。
- 我们使用一个 `set` 来保存每个单词在输入文本中出现的行号。这保证了每行只出现一次且行号按升序保存。
- 我们使用一个 `map` 来将每个单词与它出现的行号 `set` 关联起来。这样我们就可以方便地提取任意单词的 `set`。

我们的解决方案还使用了 `shared_ptr`, 原因稍后进行解释。

数据结构

虽然我们可以用 `vector`、`set` 和 `map` 来直接编写文本查询程序, 但如果定义一个更

为抽象的解决方案，会更为有效。我们将从定义一个保存输入文件的类开始，这会令文件查询更为容易。我们将这个类命名为 `TextQuery`，它包含一个 `vector` 和一个 `map`。`vector` 用来保存输入文件的文本，`map` 用来关联每个单词和它出现的行号的 `set`。这个类将会有个用来读取给定输入文件的构造函数和一个执行查询的操作。

查询操作要完成的任务非常简单：查找 `map` 成员，检查给定单词是否出现。设计这个函数的难点是确定应该返回什么内容。一旦找到了一个单词，我们需要知道它出现了多少次、它出现的行号以及每行的文本。

返回所有这些内容的最简单的方法是定义另一个类，可以命名为 `QueryResult`，来保存查询结果。这个类会有一个 `print` 函数，完成结果打印工作。

在类之间共享数据

486

我们的 `QueryResult` 类要表达查询的结果。这些结果包括与给定单词关联的行号的 `set` 和这些行对应的文本。这些数据都保存在 `TextQuery` 类型的对象中。

由于 `QueryResult` 所需要的数据都保存在一个 `TextQuery` 对象中，我们就必须确定如何访问它们。我们可以拷贝行号的 `set`，但这样做可能很耗时。而且，我们当然不希望拷贝 `vector`，因为这可能会引起整个文件的拷贝，而目标只不过是为了打印文件的一小部分而已（通常会是这样）。

通过返回指向 `TextQuery` 对象内部的迭代器（或指针），我们可以避免拷贝操作。但是，这种方法开启了一个陷阱：如果 `TextQuery` 对象在对应的 `QueryResult` 对象之前被销毁，会发生什么？在此情况下，`QueryResult` 就将引用一个不再存在的对象中的数据。

对于 `QueryResult` 对象和对应的 `TextQuery` 对象的生存期应该同步这一观察结果，其实已经暗示了问题的解决方案。考虑到这两个类概念上“共享”了数据，可以使用 `shared_ptr`（参见 12.1.1 节，第 400 页）来反映数据结构中的这种共享关系。

使用 `TextQuery` 类

当我们设计一个类时，在真正实现成员之前先编写程序使用这个类，是一种非常有用的方法。通过这种方法，可以看到类是否具有我们所需要的操作。例如，下面的程序使用了 `TextQuery` 和 `QueryResult` 类。这个函数接受一个指向要处理的文件的 `ifstream`，并与用户交互，打印给定单词的查询结果

```
void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要处理的文件
    TextQuery tq(infile); // 保存文件并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // 若遇到文件尾或用户输入了'q'时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        print(cout, tq.query(s)) << endl;
    }
}
```

我们首先用给定的 `ifstream` 初始化一个名为 `tq` 的 `TextQuery` 对象。`TextQuery` 的构造函数读取输入文件，保存在 `vector` 中，并建立单词到所在行号的 `map`。

487 while (无限) 循环提示用户输入一个要查询的单词，并打印出查询结果，如此往复。循环条件检测字面常量 `true` (参见 2.1.3 节，第 37 页)，因此永远成功。循环的退出是通过 `if` 语句中的 `break` (参见 5.5.1 节，第 170 页) 实现的。此 `if` 语句检查输入是否成功。如果成功，它再检查用户是否输入了 `q`。输入失败或用户输入了 `q` 都会使循环终止。一旦用户输入了要查询的单词，我们要求 `tq` 查找这个单词，然后调用 `print` 打印搜索结果。

12.3.1 节练习

练习 12.27: `TextQuery` 和 `QueryResult` 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

练习 12.28: 编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 `vector`、`map` 和 `set` 容器来保存来自文件的数据并生成查询结果。

练习 12.29: 我们曾经用 `do while` 循环来编写管理用户交互的循环 (参见 5.4.4 节，第 169 页)。用 `do while` 重写本节程序，解释你倾向于哪个版本，为什么。



12.3.2 文本查询程序类的定义

我们以 `TextQuery` 类的定义开始。用户创建此类的对象时会提供一个 `istream`，用来读取输入文件。这个类还提供一个 `query` 操作，接受一个 `string`，返回一个 `QueryResult` 表示 `string` 出现的那些行。

设计类的数据成员时，需要考虑与 `QueryResult` 对象共享数据的需求。`QueryResult` 类需要共享保存输入文件的 `vector` 和保存单词关联的行号的 `set`。因此，这个类应该有两个数据成员：一个指向动态分配的 `vector` (保存输入文件) 的 `shared_ptr` 和一个 `string` 到 `shared_ptr<set>` 的 `map`。`map` 将文件中每个单词关联到一个动态分配的 `set` 上，而此 `set` 保存了该单词所出现的行号。

为了使代码更易读，我们还会定义一个类型成员 (参见 7.3.1 节，第 243 页) 来引用行号，即 `string` 的 `vector` 中的下标：

```
class QueryResult; // 为了定义函数 query 的返回类型，这个定义是必需的
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 每个单词到它所在的行号的集合的映射
    std::map<std::string,
            std::shared_ptr<std::set<line_no>>> wm;
};
```

488 这个类定义最困难的部分是解开类名。与往常一样，对于可能置于头文件中的代码，在使用标准库名字时要加上 `std::` (参见 3.1 节，第 74 页)。在本例中，我们反复使用了 `std::`，

使得代码开始可能有些难读。例如，

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

如果写成下面的形式可能就更好理解一些

```
map<string, shared_ptr<set<line_no>>> wm;
```

TextQuery 构造函数

TextQuery 的构造函数接受一个 ifstream，逐行读取输入文件：

```
// 读取输入文件并建立单词到行号的映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {           // 对文件中每一行
        file->push_back(text);          // 保存此行文本
        int n = file->size() - 1;        // 当前行号
        istringstream line(text);         // 将行文本分解为单词
        string word;
        while (line >> word) {          // 对行中每个单词
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            auto &lines = wm[word];      // lines 是一个 shared_ptr
            if (!lines) // 在我们第一次遇到这个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的 set
            lines->insert(n);           // 将此行号插入 set 中
        }
    }
}
```

构造函数的初始化器分配一个新的 vector 来保存输入文件中的文本。我们用 getline 逐行读取输入文件，并存入 vector 中。由于 file 是一个 shared_ptr，我们用-> 运算符解引用 file 来提取 file 指向的 vector 对象的 push_back 成员。

接下来我们用一个 istringstream (参见 8.3 节，第 287 页) 来处理刚刚读入的一行中的每个单词。内层 while 循环用 istringstream 的输入运算符来从当前行读取每个单词，存入 word 中。在 while 循环内，我们用 map 下标运算符提取与 word 相关联的 shared_ptr<set>，并将 lines 绑定到此指针。注意，lines 是一个引用，因此改变 lines 也会改变 wm 中的元素。

若 word 不在 map 中，下标运算符会将 word 添加到 wm 中 (参见 11.3.4 节，第 387 页)，与 word 关联的值进行值初始化。这意味着，如果下标运算符将 word 添加到 wm 中，lines 将是一个空指针。如果 lines 为空，我们分配一个新的 set，并调用 reset 更新 lines 引用的 shared_ptr，使其指向这个新分配的 set。

不管是否创建了一个新的 set，我们都调用 insert 将当前行号添加到 set 中。由于 lines 是一个引用，对 insert 的调用会将新元素添加到 wm 中的 set 中。如果一个 [489] 给定单词在同一行中出现多次，对 insert 的调用什么都不会做。

QueryResult 类

QueryResult 类有三个数据成员：一个 string，保存查询单词；一个 shared_ptr，指向保存输入文件的 vector；一个 shared_ptr，指向保存单词出现行号的 set。它唯

一的一个成员函数是一个构造函数，初始化这三个数据成员：

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought; // 查询单词
    std::shared_ptr<std::set<line_no>> lines;           // 出现的行号
    std::shared_ptr<std::vector<std::string>> file;       // 输入文件
};
```

构造函数的唯一工作是将参数保存在对应的数据成员中，这是在其初始化器列表中完成的（参见 7.1.4 节，第 237 页）。

query 函数

query 函数接受一个 string 参数，即查询单词，query 用它来在 map 中定位对应的行号 set。如果找到了这个 string，query 函数构造一个 QueryResult，保存给定 string、TextQuery 的 file 成员以及从 wm 中提取的 set。

唯一的问题是：如果给定 string 未找到，我们应该返回什么？在这种情况下，没有可返回的 set。为了解决此问题，我们定义了一个局部 static 对象，它是一个指向空的行号 set 的 shared_ptr。当未找到给定单词时，我们返回此对象的一个拷贝：

```
QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，我们将返回一个指向此 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
    else
        return QueryResult(sought, loc->second, file);
}
```

490 打印结果

print 函数在给定的流上打印出给定的 QueryResult 对象：

```
ostream &print(ostream &os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数和所有出现的位置
    os << qr.sought << " occurs " << qr.lines->size() << " "
        << make_plural(qr.lines->size(), "time", "s") << endl;
    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个单词
        // 避免行号从 0 开始给用户带来的困惑
        os << "\t(line " << num + 1 << ") "
```

```
    << * (qr.file->begin() + num) << endl;
return os;
}
```

我们调用 qr.lines 指向的 set 的 size 成员来报告单词出现了多少次。由于 set 是一个 shared_ptr，必须解引用 lines。调用 make_plural（参见 6.3.2 节，第 201 页）来根据大小是否等于 1 打印 time 或 times。

在 for 循环中，我们遍历 lines 所指向的 set。for 循环体打印行号，并按人们习惯的方式调整计数值。set 中的数值就是 vector 中元素的下标，从 0 开始编号。但大多数用户认为第一行的行号应该是 1，因此我们对每个行号都加上 1，转换为人们更习惯的形式。

我们用行号从 file 指向的 vector 中提取一行文本。回忆一下，当给一个迭代器加上一个数时，会得到 vector 中相应偏移之后位置的元素（参见 3.4.2 节，第 99 页）。因此，file->begin() + num 即为 file 指向的 vector 中第 num 个位置的元素。

注意此函数能正确处理未找到单词的情况。在此情况下，set 为空。第一条输出语句会注意到单词出现了 0 次。由于 *res.lines 为空，for 循环一次也不会执行。

12.3.2 节练习

练习 12.30: 定义你自己版本的 TextQuery 和 QueryResult 类，并执行 12.3.1 节（第 431 页）中的 runQueries 函数。

练习 12.31: 如果用 vector 代替 set 保存行号，会有什么差别？哪种方法更好？为什么？

练习 12.32: 重写 TextQuery 和 QueryResult 类，用 StrBlob 代替 vector<string> 保存输入文件。

练习 12.33: 在第 15 章中我们将扩展查询系统，在 QueryResult 类中将会需要一些额外的成员。添加名为 begin 和 end 的成员，返回一个迭代器，指向一个给定查询返回的行号的 set 中的位置。再添加一个名为 get_file 的成员，返回一个 shared_ptr，指向 QueryResult 对象中的文件。

491

小结

在 C++ 中，内存是通过 `new` 表达式分配，通过 `delete` 表达式释放的。标准库还定义了一个 `allocator` 类来分配动态内存块。

分配动态内存的程序应负责释放它所分配的内存。内存的正确释放是非常容易出错的地方：要么内存永远不会被释放，要么在仍有指针引用它时就被释放了。新的标准库定义了智能指针类型——`shared_ptr`、`unique_ptr` 和 `weak_ptr`，可令动态内存管理更为安全。对于一块内存，当没有任何用户使用它时，智能指针会自动释放它。现代 C++ 程序应尽可能使用智能指针。

术语表

allocator 标准库类，用来分配未构造的内存。

空悬指针（dangling pointer） 一个指针，指向曾经保存一个对象但现在已释放的内存。众所周知，空悬指针引起的程序错误非常难以调试。

delete 释放 `new` 分配的内存。`delete p` 释放对象，`delete []p` 释放 `p` 指向的数组。`p` 可以为空，或者指向 `new` 分配的内存。

释放器（deleter） 传递给智能指针的函数，用来代替 `delete` 释放指针绑定的对象。

析构函数（destructor） 特殊的成员函数，负责在对象离开作用域或被释放时完成清理工作。

动态分配的（dynamically allocated） 在自由空间中分配的对象。在自由空间中分配的对象直到被显式释放或程序结束才会销毁。

自由空间（free store） 程序可用的内存池，保存动态分配的对象。

堆（heap） 自由空间的同义词。

new 从自由空间分配内存。`new T` 分配并构造一个类型为 `T` 的对象，并返回一个指向该对象的指针。如果 `T` 是一个数组类型，`new` 返回一个指向数组首元素的指针。类似的，`new [n] T` 分配 `n` 个类型为 `T` 的对象，并返回指向数组首元素的指针。

默认情况下，分配的对象进行默认初始化。我们也可以提供可选的初始化器。

定位 new（placement new） 一种 `new` 表达式形式，接受一些额外的参数，在 `new` 关键字后面的括号中给出。例如，`new (nothrow) int` 告诉 `new` 不要抛出异常。

引用计数（reference count） 一个计数器，记录有多少用户共享一个对象。智能指针用它来判断什么时候释放所指向的对象是安全的。

shared_ptr 提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的 `shared_ptr` 被销毁时会被释放。

智能指针（smart pointer） 标准库类型，行为类似指针，但可以检查什么时候使用指针是安全的。智能指针类型负责在恰当的时候释放内存。

unique_ptr 提供独享所有权的智能指针：当 `unique_ptr` 被销毁时，它指向的对象被释放。`unique_ptr` 不能直接拷贝或赋值。

weak_ptr 一种智能指针，指向由 `shared_ptr` 管理的对象。在确定是否应释放对象时，`shared_ptr` 并不把 `weak_ptr` 统计在内。

第III部分

类设计者的工具

内容

第 13 章 拷贝控制.....	439
第 14 章 操作重载与类型转换.....	489
第 15 章 面向对象程序设计.....	525
第 16 章 模板与泛型编程.....	577

类是 C++ 的核心概念。我们已经从第 7 章开始详细介绍了如何定义类。第 7 章涵盖了使用类的所有基本知识：类作用域、数据隐藏以及构造函数，还介绍了类的一些重要特性：成员函数、隐式 this 指针、友元以及 const、static 和 mutable 成员。在第 III 部分中，我们将延伸类的有关话题的讨论，将介绍拷贝控制、重载运算符、继承和模板。

如前所述，在 C++ 中，我们通过定义构造函数来控制在类类型的对象初始化时做什么。类还可以控制在对象拷贝、赋值、移动和销毁时做什么。在这方面，C++ 与其他语言是不同的，其他很多语言都没有给予类设计者控制这些操作的能力。第 13 章将介绍这些内容。本章还会介绍新标准引入的两个重要概念：右值引用和移动操作。

第 14 章介绍运算符重载，这种机制允许内置运算符作用于类类型的运算对象。这样，我们创建的类型直观上就可以像内置类型一样使用，运算符重载是 C++ 借以实现这一目的方法之一。

类可以重载的运算符中有一种特殊的运算符——函数调用运算符。对于重载了这种运算符的类，我们可以“调用”其对象，就好像它们是函数一样。新标准库中提供了一些设施，使得不同类型的可调用对象可以以一种一致的方式来使用，我们也将介绍这部分内容。

第 14 章最后将介绍另一种特殊类型的类成员函数——转换运算符。这些运算符定义了类类型对象的隐式转换机制。编译器应用这种转换机制的场合与原因都与内置类型转换是一样的。

第 III 部分的最后两章将介绍 C++ 如何支持面向对象编程和泛型编程。

第 15 章介绍继承和动态绑定。继承和动态绑定与数据抽象一起构成了面向对象编程的基础。继承令关联类型的定义更为简单，而动态绑定可以帮助我们编写类型无关的代码，可以忽略具有继承关系的类型之间的差异。

第 16 章介绍函数模板和类模板。模板可以让我们写出类型无关的通用类和函数。新标准引入了一些模板相关的新特性：可变参数模板、模板类型别名以及控制实例化的新方法。

编写我们自己的面向对象的或是泛型的类型需要对 C++ 有深刻的理解。幸运的是，我们无须掌握如何构建面向对象和泛型类型的细节也可以使用它们。例如，标准库中广泛使用了我们将在第 15 章和第 16 章中学习的技术，虽然我们已经使用过标准库类型和算法，但实际上我们并不了解它们是如何实现的。

因此，读者应该明白第 III 部分涉及的是相当深入的内容。编写模板或面向对象的类要求对 C++ 的基本知识和基本类的定义有着深刻的理解。

第 13 章

拷贝控制

内容

13.1 拷贝、赋值与销毁	440
13.2 拷贝控制和资源管理	452
13.3 交换操作	457
13.4 拷贝控制示例	460
13.5 动态内存管理类	464
13.6 对象移动	470
小结	486
术语表	486

如我们在第 7 章所见，每个类都定义了一个新类型和在此类型对象上可执行的操作。在本章中，我们还将学到，类可以定义构造函数，用来控制在创建此类型对象时做什么。

在本章中，我们还将学习类如何控制该类型对象拷贝、赋值、移动或销毁时做什么。类通过一些特殊的成员函数控制这些操作，包括：拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符以及析构函数。

496

当定义一个类时，我们显式地或隐式地指定在此类型的对象拷贝、移动、赋值和销毁时做什么。一个类通过定义五种特殊的成员函数来控制这些操作，包括：**拷贝构造函数**（copy constructor）、**拷贝赋值运算符**（copy-assignment operator）、**移动构造函数**（move constructor）、**移动赋值运算符**（move-assignment operator）和**析构函数**（destructor）。拷贝和移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么。拷贝和移动赋值运算符定义了将一个对象赋予同类型的另一个对象时做什么。析构函数定义了当此类型对象销毁时做什么。我们称这些操作为**拷贝控制操作**（copy control）。

如果一个类没有定义所有这些拷贝控制成员，编译器会自动为它定义缺失的操作。因此，很多类会忽略这些拷贝控制操作（参见 7.1.5 节，第 239 页）。但是，对一些类来说，依赖这些操作的默认定义会导致灾难。通常，实现拷贝控制操作最困难的地方是首先认识到什么时候需要定义这些操作。



WARNING

在定义任何 C++ 类时，拷贝控制操作都是必要部分。对初学 C++ 的程序员来说，必须定义对象拷贝、移动、赋值或销毁时做什么，这常常令他们感到困惑。这种困扰很复杂，因为如果我们不显式定义这些操作，编译器也会为我们定义，但编译器定义的版本的行为可能并非我们所想。

13.1 拷贝、赋值与销毁

我们将以最基本的操作——拷贝构造函数、拷贝赋值运算符和析构函数作为开始。我们在 13.6 节（第 470 页）中将介绍移动操作（新标准所引入的操作）。



13.1.1 拷贝构造函数

如果一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。

```
class Foo {
public:
    Foo();           // 默认构造函数
    Foo(const Foo&); // 拷贝构造函数
    // ...
};
```

拷贝构造函数的第一个参数必须是一个引用类型，原因我们稍后解释。虽然我们可以定义一个接受非 const 引用的拷贝构造函数，但此参数几乎总是一个 const 的引用。拷贝构造函数在几种情况下都会被隐式地使用。因此，拷贝构造函数通常不应该是 explicit 的（参见 7.5.4 节，第 265 页）。

497

合成拷贝构造函数

如果我们没有为一个类定义拷贝构造函数，编译器会为我们定义一个。与合成默认构造函数（参见 7.1.4 节，第 235 页）不同，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。

如我们将在 13.1.6 节（第 450 页）中所见，对某些类来说，**合成拷贝构造函数**（synthesized copy constructor）用来阻止我们拷贝该类类型的对象。而一般情况，合成的拷贝构造函数会将其参数的成员逐个拷贝到正在创建的对象中（参见 7.1.5 节，第 239 页）。编译器从给

定对象中依次将每个非 static 成员拷贝到正在创建的对象中。

每个成员的类型决定了它如何拷贝：对类类型的成员，会使用其拷贝构造函数来拷贝；内置类型的成员则直接拷贝。虽然我们不能直接拷贝一个数组（参见 3.5.1 节，第 102 页），但合成拷贝构造函数会逐元素地拷贝一个数组类型的成员。如果数组元素是类类型，则使用元素的拷贝构造函数来进行拷贝。

作为一个例子，我们的 Sales_data 类的合成拷贝构造函数等价于：

```
class Sales_data {
public:
    // 其他成员和构造函数的定义，如前
    // 与合成的拷贝构造函数等价的拷贝构造函数的声明
    Sales_data(const Sales_data&);

private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};

// 与 Sales_data 的合成的拷贝构造函数等价
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),           // 使用 string 的拷贝构造函数
    units_sold(orig.units_sold),   // 拷贝 orig.units_sold
    revenue(orig.revenue)         // 拷贝 orig.revenue
{                                // 空函数体}
```

拷贝初始化

现在，我们可以完全理解直接初始化和拷贝初始化之间的差异了（参见 3.2.1 节，第 76 页）：

```
string dots(10, '.');           // 直接初始化
string s(dots);                // 直接初始化
string s2 = dots;               // 拷贝初始化
string null_book = "9-999-99999-9"; // 拷贝初始化
string nines = string(100, '9'); // 拷贝初始化
```

当使用直接初始化时，我们实际上是要求编译器使用普通的函数匹配（参见 6.4 节，第 209 页）来选择与我们提供的参数最匹配的构造函数。当我们使用 **拷贝初始化**（copy initialization）时，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换（参见 7.5.4 节，第 263 页）。

拷贝初始化通常使用拷贝构造函数来完成。但是，如我们将在 13.6.2 节（第 473 页）◀ 498 所见，如果一个类有一个移动构造函数，则拷贝初始化有时会使用移动构造函数而非拷贝构造函数来完成。但现在，我们只需了解拷贝初始化何时发生，以及拷贝初始化是依靠拷贝构造函数或移动构造函数来完成的就可以了。

拷贝初始化不仅在我们用 = 定义变量时会发生，在下列情况下也会发生

- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素或一个聚合类中的成员（参见 7.5.5 节，第 266 页）

某些类类型还会对它们所分配的对象使用拷贝初始化。例如，当我们初始化标准库容器或是调用其 `insert` 或 `push` 成员（参见 9.3.1 节，第 306 页）时，容器会对其元素进行拷贝初始化。与之相对，用 `emplace` 成员创建的元素都进行直接初始化（参见 9.3.1 节，第 308 页）。

参数和返回值

在函数调用过程中，具有非引用类型的参数要进行拷贝初始化（参见 6.2.1 节，第 188 页）。类似的，当一个函数具有非引用的返回类型时，返回值会被用来初始化调用方的结果（参见 6.3.2 节，第 201 页）。

拷贝构造函数被用来初始化非引用类类型参数，这一特性解释了为什么拷贝构造函数自己的参数必须是引用类型。如果其参数不是引用类型，则调用永远也不会成功——为了调用拷贝构造函数，我们必须拷贝它的实参，但为了拷贝实参，我们又需要调用拷贝构造函数，如此无限循环。

拷贝初始化的限制

如前所述，如果我们使用的初始化值要求通过一个 `explicit` 的构造函数来进行类型转换（参见 7.5.4 节，第 265 页），那么使用拷贝初始化还是直接初始化就不是无关紧要的了：

```
vector<int> v1(10); // 正确：直接初始化
vector<int> v2 = 10; // 错误：接受大小参数的构造函数是 explicit 的
void f(vector<int>); // f 的参数进行拷贝初始化
f(10); // 错误：不能用一个 explicit 的构造函数拷贝一个实参
f(vector<int>(10)); // 正确：从一个 int 直接构造一个临时 vector
```

直接初始化 `v1` 是合法的，但看起来与之等价的拷贝初始化 `v2` 则是错误的，因为 `vector` 的接受单一大小参数的构造函数是 `explicit` 的。出于同样的原因，当传递一个实参或从函数返回一个值时，我们不能隐式使用一个 `explicit` 构造函数。如果我们希望使用一个 `explicit` 构造函数，就必须显式地使用，像此代码中最后一行那样。

499 > 编译器可以绕过拷贝构造函数

在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象。即，编译器被允许将下面的代码

```
string null_book = "9-999-99999-9"; // 拷贝初始化
```

改写为

```
string null_book("9-999-99999-9"); // 编译器略过了拷贝构造函数
```

但是，即使编译器略过了拷贝/移动构造函数，但在这个程序点上，拷贝/移动构造函数必须是存在且可访问的（例如，不能是 `private` 的）。

13.1.1 节练习

练习 13.1：拷贝构造函数是什么？什么时候使用它？

练习 13.2：解释为什么下面的声明是非法的：

```
Sales_data::Sales_data(Sales_data rhs);
```

练习 13.3: 当我们拷贝一个 StrBlob 时，会发生什么？拷贝一个 StrBlobPtr 呢？

练习 13.4: 假定 Point 是一个类类型，它有一个 public 的拷贝构造函数，指出下面程序片段中哪些地方使用了拷贝构造函数：

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

练习 13.5: 给定下面的类框架，编写一个拷贝构造函数，拷贝所有成员。你的构造函数应该动态分配一个新的 string（参见 12.1.2 节，第 407 页），并将对象拷贝到 ps 指向的位置，而不是 ps 本身的位置。

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

13.1.2 拷贝赋值运算符

与类控制其对象如何初始化一样，类也可以控制其对象如何赋值：

```
Sales_data trans, accum;
trans = accum; // 使用 Sales_data 的拷贝赋值运算符
```

与拷贝构造函数一样，如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个。

重载赋值运算符

在介绍合成赋值运算符之前，我们需要了解一点儿有关重载运算符（overloaded operator）的知识，详细内容将在第 14 章中进行介绍。

重载运算符本质上是函数，其名字由 operator 关键字后接表示要定义的运算符的符号组成。因此，赋值运算符就是一个名为 operator= 的函数。类似于任何其他函数，运算符函数也有一个返回类型和一个参数列表。

重载运算符的参数表示运算符的运算对象。某些运算符，包括赋值运算符，必须定义为成员函数。如果一个运算符是一个成员函数，其左侧运算对象就绑定到隐式的 this 参数（参见 7.1.2 节，第 231 页）。对于一个二元运算符，例如赋值运算符，其右侧运算对象作为显式参数传递。

拷贝赋值运算符接受一个与其所在类相同类型的参数：

```
class Foo {
public:
```

```
Foo& operator=(const Foo&); // 赋值运算符
// ...
};
```

为了与内置类型的赋值（参见 4.4 节，第 129 页）保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用。另外值得注意的是，标准库通常要求保存在容器中的类型要具有赋值运算符，且其返回值是左侧运算对象的引用。



赋值运算符通常应该返回一个指向其左侧运算对象的引用。

合成拷贝赋值运算符

与处理拷贝构造函数一样，如果一个类未定义自己的拷贝赋值运算符，编译器会为它生成一个合成拷贝赋值运算符（synthesized copy-assignment operator）。类似拷贝构造函数，对于某些类，合成拷贝赋值运算符用来禁止该类型对象的赋值（参见 13.1.6 节，第 450 页）。如果拷贝赋值运算符并非出于此目的，它会将右侧运算对象的每个非 static 成员赋予左侧运算对象的对应成员，这一工作是通过成员类型的拷贝赋值运算符来完成的。对于数组类型的成员，逐个赋值数组元素。合成拷贝赋值运算符返回一个指向其左侧运算对象的引用。

501

作为一个例子，下面的代码等价于 Sales_data 的合成拷贝赋值运算符：

```
// 等价于合成拷贝赋值运算符
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;           // 调用 string::operator=
    units_sold = rhs.units_sold; // 使用内置的 int 赋值
    revenue = rhs.revenue;        // 使用内置的 double 赋值
    return *this;                 // 返回一个此对象的引用
}
```

13.1.2 节练习

练习 13.6：拷贝赋值运算符是什么？什么时候使用它？合成拷贝赋值运算符完成什么工作？什么时候会生成合成拷贝赋值运算符？

练习 13.7：当我们将一个 StrBlob 赋值给另一个 StrBlob 时，会发生什么？赋值 StrBlobPtr 呢？

练习 13.8：为 13.1.1 节（第 443 页）练习 13.5 中的 HasPtr 类编写赋值运算符。类似拷贝构造函数，你的赋值运算符应该将对象拷贝到 ps 指向的位置。



13.1.3 析构函数

析构函数执行与构造函数相反的操作：构造函数初始化对象的非 static 数据成员，还可能做一些其他工作；析构函数释放对象使用的资源，并销毁对象的非 static 数据成员。

析构函数是类的一个成员函数，名字由波浪号接类名构成。它没有返回值，也不接受参数：

```
class Foo {
public:
    ~Foo(); // 析构函数
```

```
//...  
};
```

由于析构函数不接受参数，因此它不能被重载。对一个给定类，只会有唯一一个析构函数。

析构函数完成什么工作

如同构造函数有一个初始化部分和一个函数体（参见 7.5.1 节，第 257 页），析构函数也有一个函数体和一个析构部分。在一个构造函数中，成员的初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化。在一个析构函数中，首先执行函数体，502然后销毁成员。成员按初始化顺序的逆序销毁。

在对象最后一次使用之后，析构函数的函数体可执行类设计者希望执行的任何收尾工作。通常，析构函数释放对象在生存期分配的所有资源。

在一个析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。成员销毁时发生什么完全依赖于成员的类型。销毁类类型的成员需要执行成员自己的析构函数。内置类型没有析构函数，因此销毁内置类型成员什么也不需要做。



隐式销毁一个内置指针类型的成员不会 `delete` 它所指向的对象。

与普通指针不同，智能指针（参见 12.1.1 节，第 402 页）是类类型，所以具有析构函数。因此，与普通指针不同，智能指针成员在析构阶段会被自动销毁。

什么时候会调用析构函数

无论何时一个对象被销毁，就会自动调用其析构函数：

- 变量在离开其作用域时被销毁。
- 当一个对象被销毁时，其成员被销毁。
- 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁。
- 对于动态分配的对象，当对指向它的指针应用 `delete` 运算符时被销毁（参见 12.1.2 节，第 409 页）。
- 对于临时对象，当创建它的完整表达式结束时被销毁。

由于析构函数自动运行，我们的程序可以按需要分配资源，而（通常）无须担心何时释放这些资源。

例如，下面代码片段定义了四个 `Sales_data` 对象：

```
{ // 新作用域  
    // p 和 p2 指向动态分配的对象  
    Sales_data *p = new Sales_data;           // p 是一个内置指针  
    auto p2 = make_shared<Sales_data>();      // p2 是一个 shared_ptr  
    Sales_data item(*p);                      // 拷贝构造函数将*p 拷贝到 item 中  
    vector<Sales_data> vec;                   // 局部对象  
    vec.push_back(*p2);                       // 拷贝 p2 指向的对象  
    delete p;                                // 对 p 指向的对象执行析构函数  
} // 退出局部作用域；对 item、p2 和 vec 调用析构函数  
// 销毁 p2 会递减其引用计数；如果引用计数变为 0，对象被释放  
// 销毁 vec 会销毁它的元素
```

503 每个 Sales_data 对象都包含一个 string 成员，它分配动态内存来保存 bookNo 成员中的字符。但是，我们的代码唯一需要直接管理的内存就是我们直接分配的 Sales_data 对象。我们的代码只需直接释放绑定到 p 的动态分配对象。

其他 Sales_data 对象会在离开作用域时被自动销毁。当程序块结束时，vec、p2 和 item 都离开了作用域，意味着在这些对象上分别会执行 vector、shared_ptr 和 Sales_data 的析构函数。vector 的析构函数会销毁我们添加到 vec 的元素。shared_ptr 的析构函数会递减 p2 指向的对象的引用计数。在本例中，引用计数会变为 0，因此 shared_ptr 的析构函数会 delete p2 分配的 Sales_data 对象。

在所有情况下，Sales_data 的析构函数都会隐式地销毁 bookNo 成员。销毁 bookNo 会调用 string 的析构函数，它会释放用来保存 ISBN 的内存。



当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

合成析构函数

当一个类未定义自己的析构函数时，编译器会为它定义一个合成析构函数（synthesized destructor）。类似拷贝构造函数和拷贝赋值运算符，对于某些类，合成析构函数被用来阻止该类型的对象被销毁（参见 13.1.6 节，第 450 页）。如果不是这种情况，合成析构函数的函数体就为空。

例如，下面的代码片段等价于 Sales_data 的合成析构函数：

```
class Sales_data {
public:
    // 成员会被自动销毁，除此之外不需要做其他事情
    ~Sales_data() { }
    // 其他成员的定义，如前
};
```

在（空）析构函数体执行完毕后，成员会被自动销毁。特别的，string 的析构函数会被调用，它将释放 bookNo 成员所用的内存。

认识到析构函数体自身并不直接销毁成员是非常重要的。成员是在析构函数体之后隐含的析构阶段中被销毁的。在整个对象销毁过程中，析构函数体是作为成员销毁步骤之外的另一部分而进行的。

13.1.3 节练习

练习 13.9：析构函数是什么？合成析构函数完成什么工作？什么时候会生成合成析构函数？

练习 13.10：当一个 StrBlob 对象销毁时会发生什么？一个 StrBlobPtr 对象销毁时呢？

练习 13.11：为前面练习中的 HasPtr 类添加一个析构函数。

练习 13.12：在下面的代码片段中会发生几次析构函数调用？

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
```

```

        return item1.isbn() != item2.isbn();
    }
}

```

练习 13.13：理解拷贝控制成员和构造函数的一个好方法是定义一个简单的类，为该类定义这些成员，每个成员都打印出自己的名字：

```

struct X {
    X() {std::cout << "X()" << std::endl;}
    X(const X&) {std::cout << "X(const X&)" << std::endl;}
};

```

给 `X` 添加拷贝赋值运算符和析构函数，并编写一个程序以不同方式使用 `X` 的对象：将它们作为非引用和引用参数传递；动态分配它们；将它们存放于容器中；诸如此类。观察程序的输出，直到你确认理解了什么时候会使用拷贝控制成员，以及为什么会使用它们。当你观察程序输出时，记住编译器可以略过对拷贝构造函数的调用。

13.1.4 三/五法则



如前所述，有三个基本操作可以控制类的拷贝操作：拷贝构造函数、拷贝赋值运算符和析构函数。而且，在新标准下，一个类还可以定义一个移动构造函数和一个移动赋值运算符，我们将在 13.6 节（第 470 页）中介绍这些内容。

C++语言并不要求我们定义所有这些操作：可以只定义其中一个或两个，而不必定义所有。但是，这些操作通常应该被看作一个整体。通常，只需要其中一个操作，而不需要定义所有操作的情况是很少见的。

< 504

需要析构函数的类也需要拷贝和赋值操作

当我们决定一个类是否要定义它自己版本的拷贝控制成员时，一个基本原则是首先确定这个类是否需要一个析构函数。通常，对析构函数的需求要比对拷贝构造函数或赋值运算符的需求更为明显。如果这个类需要一个析构函数，我们几乎可以肯定它也需要一个拷贝构造函数和一个拷贝赋值运算符。

我们在练习中用过的 `HasPtr` 类是一个好例子（参见 13.1.1 节，第 443 页）。这个类在构造函数中分配动态内存。合成析构函数不会 `delete` 一个指针数据成员。因此，此类需要定义一个析构函数来释放构造函数分配的内存。

应该怎么做可能还有点儿不清晰，但基本原则告诉我们，`HasPtr` 也需要一个拷贝构造函数和一个拷贝赋值运算符。

< 505

如果我们为 `HasPtr` 定义一个析构函数，但使用合成版本的拷贝构造函数和拷贝赋值运算符，考虑会发生什么：

```

class HasPtr {
public:
    HasPtr(const std::string &s = std::string()): ps(new std::string(s)), i(0) { }
    ~HasPtr() { delete ps; }
    // 错误：HasPtr 需要一个拷贝构造函数和一个拷贝赋值运算符
    // 其他成员的定义，如前
};

```

在这个版本的类定义中，构造函数中分配的内存将在 `HasPtr` 对象销毁时被释放。但不幸的是，我们引入了一个严重的错误！这个版本的类使用了合成的拷贝构造函数和拷贝

赋值运算符。这些函数简单拷贝指针成员，这意味着多个 HasPtr 对象可能指向相同的内存：

```
HasPtr f(HasPtr hp)           // HasPtr 是传值参数，所以将被拷贝
{
    HasPtr ret = hp;          // 拷贝给定的 HasPtr
    // 处理 ret
    return ret;               // ret 和 hp 被销毁
}
```

当 `f` 返回时，`hp` 和 `ret` 都被销毁，在两个对象上都会调用 `HasPtr` 的析构函数。此析构函数会 `delete` `ret` 和 `hp` 中的指针成员。但这两个对象包含相同的指针值。此代码会导致此指针被 `delete` 两次，这显然是一个错误（参见 12.1.2 节，第 411 页）。将要发生什么是未定义的。

此外，`f` 的调用者还会使用传递给 `f` 的对象：

```
HasPtr p("some values");
f(p);                      // 当 f 结束时，p.ps 指向的内存被释放
HasPtr q(p);                // 现在 p 和 q 都指向无效内存！
```

`p`（以及 `q`）指向的内存不再有效，在 `hp`（或 `ret`）销毁时它就被归还给系统了。



如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数。

需要拷贝操作的类也需要赋值操作，反之亦然

虽然很多类需要定义所有（或是不需要定义任何）拷贝控制成员，但某些类所要完成的工作，只需要拷贝或赋值操作，不需要析构函数。

作为一个例子，考虑一个类为每个对象分配一个独有的、唯一的序号。这个类需要一个拷贝构造函数为每个新创建的对象生成一个新的、独一无二的序号。除此之外，这个拷贝构造函数从给定对象拷贝所有其他数据成员。这个类还需要自定义拷贝赋值运算符来避免将序号赋予目的对象。但是，这个类不需要自定义析构函数。

这个例子引出了第二个基本原则：如果一个类需要一个拷贝构造函数，几乎可以肯定它也需要一个拷贝赋值运算符。反之亦然——如果一个类需要一个拷贝赋值运算符，几乎可以肯定它也需要一个拷贝构造函数。然而，无论是需要拷贝构造函数还是需要拷贝赋值运算符都不必然意味着也需要析构函数。

13.1.4 节练习

练习 13.14：假定 `numbered` 是一个类，它有一个默认构造函数，能为每个对象生成一个唯一的序号，保存在名为 `mysn` 的数据成员中。假定 `numbered` 使用合成的拷贝控制成员，并给定如下函数：

```
void f (numbered s) { cout << s.mysn << endl; }
```

则下面代码输出什么内容？

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

练习 13.15：假定 `numbered` 定义了一个拷贝构造函数，能生成一个新的序号。这会改变上一题中调用的输出结果吗？如果会改变，为什么？新的输出结果是什么？

练习 13.16: 如果 `f` 中的参数是 `const numbered&`, 将会怎样? 这会改变输出结果吗? 如果会改变, 为什么? 新的输出结果是什么?

练习 13.17: 分别编写前三题中所描述的 `numbered` 和 `f`, 验证你是否正确预测了输出结果。

13.1.5 使用`=default`

我们可以通过将拷贝控制成员定义为`=default` 来显式地要求编译器生成合成的版本 (参见 7.1.4 节, 第 237 页):

```
class Sales_data {
public:
    // 拷贝控制成员; 使用 default
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // 其他成员的定义, 如前
};

Sales_data& Sales_data::operator=(const Sales_data&) = default;
```

当我们在类内用`=default` 修饰成员的声明时, 合成的函数将隐式地声明为内联的 (就像任何其他类内声明的成员函数一样)。如果我们不希望合成的成员是内联函数, 应该只对成员的类外定义使用`=default`, 就像对拷贝赋值运算符所做的那样。



我们只能对具有合成版本的成员函数使用`=default` (即, 默认构造函数或拷贝控制成员)。

13.1.6 阻止拷贝



大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符, 无论是隐式地还是显式地。

虽然大多数类应该定义 (而且通常也的确定义了) 拷贝构造函数和拷贝赋值运算符, 但对某些类来说, 这些操作没有合理的意义。在此情况下, 定义类时必须采用某种机制阻止拷贝或赋值。例如, `iostream` 类阻止了拷贝, 以避免多个对象写入或读取相同的 IO 缓冲。为了阻止拷贝, 看起来可能应该不定义拷贝控制成员。但是, 这种策略是无效的: 如果我们的类未定义这些操作, 编译器为它生成合成的版本。

定义删除的函数

在新标准下, 我们可以通过将拷贝构造函数和拷贝赋值运算符定义为删除的函数 (deleted function) 来阻止拷贝。删除的函数是这样一种函数: 我们虽然声明了它们, 但不能以任何方式使用它们。在函数的参数列表后面加上`=delete` 来指出我们希望将它定义为删除的:

```
struct NoCopy {
    NoCopy() = default;           // 使用合成的默认构造函数
    NoCopy(const NoCopy&) = delete;        // 阻止拷贝
    NoCopy &operator=(const NoCopy&) = delete;    // 阻止赋值
```

```

~NoCopy() = default;      // 使用合成的析构函数
// 其他成员
};

=delete 通知编译器（以及我们代码的读者），我们不希望定义这些成员。

```

与`=default`不同，`=delete`必须出现在函数第一次声明的时候，这个差异与这些声明的含义在逻辑上是吻合的。一个默认的成员只影响为这个成员而生成的代码，因此`=default`直到编译器生成代码时才需要。而另一方面，编译器需要知道一个函数是删除的，以便禁止试图使用它的操作。

与`=default`的另一个不同之处是，我们可以对任何函数指定`=delete`（我们只能对编译器可以合成的默认构造函数或拷贝控制成员使用`=default`）。虽然删除函数的主要用途是禁止拷贝控制成员，但当我们希望引导函数匹配过程时，删除函数有时也是有用的。

析构函数不能是删除的成员

值得注意的是，我们不能删除析构函数。如果析构函数被删除，就无法销毁此类型的对象了。对于一个删除了析构函数的类型，编译器将不允许定义该类型的变量或创建该类的临时对象。而且，如果一个类有某个成员的类型删除了析构函数，我们也不能定义该类的变量或临时对象。因为如果一个成员的析构函数是删除的，则该成员无法被销毁。而如果一个成员无法被销毁，则对象整体也就无法被销毁了。

对于删除了析构函数的类型，虽然我们不能定义这种类型的变量或成员，但可以动态分配这种类型的对象。但是，不能释放这些对象：

```

struct NoDtor {
    NoDtor() = default; // 使用合成默认构造函数
    ~NoDtor() = delete; // 我们不能销毁 NoDtor 类型的对象
};

NoDtor nd; // 错误：NoDtor 的析构函数是删除的
NoDtor *p = new NoDtor(); // 正确：但我们不能 delete p
delete p; // 错误：NoDtor 的析构函数是删除的

```



对于析构函数已删除的类型，不能定义该类型的变量或释放指向该类型动态分配对象的指针。

合成的拷贝控制成员可能是删除的

如前所述，如果我们未定义拷贝控制成员，编译器会为我们定义合成的版本。类似的，如果一个类未定义构造函数，编译器会为其合成一个默认构造函数（参见 7.1.4 节，第 235 页）。对某些类来说，编译器将这些合成的成员定义为删除的函数：

- 如果类的某个成员的析构函数是删除的或不可访问的（例如，是 `private` 的），则类的合成析构函数被定义为删除的。
- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的。如果类的某个成员的析构函数是删除的或不可访问的，则类合成的拷贝构造函数也被定义为删除的。
- 如果类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个 `const` 的或引用成员，则类的合成拷贝赋值运算符被定义为删除的。
- 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化器（参见 2.6.1 节，第 65 页），或是类有一个 `const` 成员，它没有

类内初始化器且其类型未显式定义默认构造函数，则该类的默认构造函数被定义为删除的。

本质上，这些规则的含义是：如果一个类有数据成员不能默认构造、拷贝、复制或销毁，509则对应的成员函数将被定义为删除的。

一个成员有删除的或不可访问的析构函数会导致合成的默认和拷贝构造函数被定义为删除的，这看起来可能有些奇怪。其原因是，如果没有这条规则，我们可能会创建出无法销毁的对象。

对于具有引用成员或无法默认构造的 `const` 成员的类，编译器不会为其合成默认构造函数，这应该不奇怪。同样不出人意料的规则是：如果一个类有 `const` 成员，则它不能使用合成的拷贝赋值运算符。毕竟，此运算符试图赋值所有成员，而将一个新值赋予一个 `const` 对象是不可能的。

虽然我们可以将一个新值赋予一个引用成员，但这样做改变的是引用指向的对象的值，而不是引用本身。如果为这样的类合成拷贝赋值运算符，则赋值后，左侧运算对象仍然指向与赋值前一样的对象，而不会与右侧运算对象指向相同的对象。由于这种行为看起来并不是我们所期望的，因此对于有引用成员的类，合成拷贝赋值运算符被定义为删除的。

我们将在 13.6.2 节（第 476 页）、15.7.2 节（第 553 页）及 19.6 节（第 751 页）中介绍导致类的拷贝控制成员被定义为删除函数的其他原因。



本质上，当不可能拷贝、赋值或销毁类的成员时，类的合成拷贝控制成员就被定义为删除的。

private 拷贝控制

在新标准发布之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为 `private` 的来阻止拷贝：

```
class PrivateCopy {
    // 无访问说明符；接下来的成员默认为 private 的；参见 7.2 节（第 240 页）
    // 拷贝控制成员是 private 的，因此普通用户代码无法访问
    PrivateCopy(const PrivateCopy&);
    PrivateCopy &operator=(const PrivateCopy&);
    // 其他成员

public:
    PrivateCopy() = default; // 使用合成的默认构造函数
    ~PrivateCopy(); // 用户可以定义此类型的对象，但无法拷贝它们
};
```

由于析构函数是 `public` 的，用户可以定义 `PrivateCopy` 类型的对象。但是，由于拷贝构造函数和拷贝赋值运算符是 `private` 的，用户代码将不能拷贝这个类型的对象。但是，友元和成员函数仍旧可以拷贝对象。为了阻止友元和成员函数进行拷贝，我们将这些拷贝控制成员声明为 `private` 的，但并不定义它们。

声明但不定义一个成员函数是合法的（参见 6.1.2 节，第 186 页），对此只有一个例外，我们将在 15.2.1 节（第 528 页）中介绍。试图访问一个未定义的成员将导致一个链接时错误。通过声明（但不定义）`private` 的拷贝构造函数，我们可以预先阻止任何拷贝该类型对象的企图：试图拷贝对象的用户代码将在编译阶段被标记为错误；成员函数或友元函数中的拷贝操作将会导致链接时错误。510



希望阻止拷贝的类应该使用`=delete` 来定义它们自己的拷贝构造函数和拷贝赋值运算符，而不应该将它们声明为 `private` 的。

13.1.6 节练习

练习 13.18: 定义一个 `Employee` 类，它包含雇员的姓名和唯一的雇员证号。为这个类定义默认构造函数，以及接受一个表示雇员姓名的 `string` 的构造函数。每个构造函数应该通过递增一个 `static` 数据成员来生成一个唯一的证号。

练习 13.19: 你的 `Employee` 类需要定义它自己的拷贝控制成员吗？如果需要，为什么？如果不呢，为什么？实现你认为 `Employee` 需要的拷贝控制成员。

练习 13.20: 解释当我们拷贝、赋值或销毁 `TextQuery` 和 `QueryResult` 类（参见 12.3 节，第 430 页）对象时会发生什么。

练习 13.21: 你认为 `TextQuery` 和 `QueryResult` 类需要定义它们自己版本的拷贝控制成员吗？如果需要，为什么？如果不呢，为什么？实现你认为这两个类需要的拷贝控制操作。



13.2 拷贝控制和资源管理

通常，管理类外资源的类必须定义拷贝控制成员。如我们在 13.1.4 节（第 447 页）中所见，这种类需要通过析构函数来释放对象所分配的资源。一旦一个类需要析构函数，那么它几乎肯定也需要一个拷贝构造函数和一个拷贝赋值运算符。

为了定义这些成员，我们首先必须确定此类型对象的拷贝语义。一般来说，有两种选择：可以定义拷贝操作，使类的行为看起来像一个值或者像一个指针。

类的行为像一个值，意味着它应该也有自己的状态。当我们拷贝一个像值的对象时，副本和原对象是完全独立的。改变副本不会对原对象有任何影响，反之亦然。

行为像指针的类则共享状态。当我们拷贝一个这种类的对象时，副本和原对象使用相同的底层数据。改变副本也会改变原对象，反之亦然。

在我们使用过的标准库类中，标准库容器和 `string` 类的行为像一个值。而不出意外的，`shared_ptr` 类提供类似指针的行为，就像我们的 `StrBlob` 类（参见 12.1.1 节，第 511 > 405 页）一样，`IO` 类型和 `unique_ptr` 不允许拷贝或赋值，因此它们的行为既不像值也不像指针。

为了说明这两种方式，我们会为练习中的 `HasPtr` 类定义拷贝控制成员。首先，我们将令类的行为像一个值；然后重新实现类，使它的行为像一个指针。

我们的 `HasPtr` 类有两个成员，一个 `int` 和一个 `string` 指针。通常，类直接拷贝内置类型（不包括指针）成员；这些成员本身就是值，因此通常应该让它们的行为像值一样。我们如何拷贝指针成员决定了像 `HasPtr` 这样的类是具有类值行为还是类指针行为。

13.2 节练习

练习 13.22: 假定我们希望 `HasPtr` 的行为像一个值。即，对于对象所指向的 `string`

成员，每个对象都有一份自己的拷贝。我们将在下一节介绍拷贝控制成员的定义。但是，你已经学习了定义这些成员所需的所有知识。在继续学习下一节之前，为 HasPtr 编写拷贝构造函数和拷贝赋值运算符。

13.2.1 行为像值的类



为了提供类值的行为，对于类管理的资源，每个对象都应该拥有一份自己的拷贝。这意味着对于 ps 指向的 string，每个 HasPtr 对象都必须有自己的拷贝。为了实现类值行为，HasPtr 需要

- 定义一个拷贝构造函数，完成 string 的拷贝，而不是拷贝指针
- 定义一个析构函数来释放 string
- 定义一个拷贝赋值运算符来释放对象当前的 string，并从右侧运算对象拷贝 string

类值版本的 HasPtr 如下所示

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    // 对 ps 指向的 string，每个 HasPtr 对象都有自己的拷贝
    HasPtr(const HasPtr &p):
        ps(new std::string(*p.ps)), i(p.i) { }
    HasPtr& operator=(const HasPtr &);

    ~HasPtr() { delete ps; }

private:
    std::string *ps;
    int      i;
};
```

我们的类足够简单，在类内就已定义了除赋值运算符之外的所有成员函数。第一个构造函数接受一个（可选的）string 参数。这个构造函数动态分配它自己的 string 副本，并将指向 string 的指针保存在 ps 中。拷贝构造函数也分配它自己的 string 副本。析构函数对指针成员 ps 执行 delete，释放构造函数中分配的内存。

<512

类值拷贝赋值运算符

赋值运算符通常组合了析构函数和构造函数的操作。类似析构函数，赋值操作会销毁左侧运算对象的资源。类似拷贝构造函数，赋值操作会从右侧运算对象拷贝数据。但是，非常重要的一点是，这些操作是以正确的顺序执行的，即使将一个对象赋予它自身，也保证正确。而且，如果可能，我们编写的赋值运算符还应该是异常安全的——当异常发生时能将左侧运算对象置于一个有意义的状态（参见 5.6.2 节，第 175 页）。

在本例中，通过先拷贝右侧运算对象，我们可以处理自赋值情况，并能保证在异常发生时代码也是安全的。在完成拷贝后，我们释放左侧运算对象的资源，并更新指针指向新分配的 string：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps); // 拷贝底层 string
    delete ps; // 释放旧内存
```

```

    ps = newp;           // 从右侧运算对象拷贝数据到本对象
    i = rhs.i;
    return *this;        // 返回本对象
}

```

在这个赋值运算符中，非常清楚，我们首先进行了构造函数的工作：newp 的初始化器等价于 HasPtr 的拷贝构造函数中 ps 的初始化器。接下来与析构函数一样，我们 delete 当前 ps 指向的 string。然后就只剩下拷贝指向新分配的 string 的指针，以及从 rhs 拷贝 int 值到本对象了。

关键概念：赋值运算符

当你编写赋值运算符时，有两点需要记住：

- 如果将一个对象赋予它自身，赋值运算符必须能正确工作。
- 大多数赋值运算符组合了析构函数和拷贝构造函数的工作。

当你编写一个赋值运算符时，一个好的模式是先将右侧运算对象拷贝到一个局部临时对象中。当拷贝完成后，销毁左侧运算对象的现有成员就是安全的了。一旦左侧运算对象的资源被销毁，就只剩下将数据从临时对象拷贝到左侧运算对象的成员中了。

513 >

为了说明防范自赋值操作的重要性，考虑如果赋值运算符如下编写将会发生什么

```

// 这样编写赋值运算符是错误的!
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps; // 释放对象指向的 string
    // 如果 rhs 和*this 是同一个对象，我们就将从已释放的内存中拷贝数据!
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}

```

如果 rhs 和本对象是同一个对象，delete ps 会释放*this 和 rhs 指向的 string。接下来，当我们在 new 表达式中试图拷贝*(rhs.ps) 时，就会访问一个指向无效内存的指针，其行为和结果是未定义的。



对于一个赋值运算符来说，正确工作是非常重要的，即使是将一个对象赋予它自身，也要能正确工作。一个好的方法是在销毁左侧运算对象资源之前拷贝右侧运算对象。

13.2.1 节练习

练习 13.23：比较上一节练习中你编写的拷贝控制成员和这一节中的代码。确定你理解了你的代码和我们的代码之间的差异（如果有的话）。

练习 13.24：如果本节中的 HasPtr 版本未定义析构函数，将会发生什么？如果未定义拷贝构造函数，将会发生什么？

练习 13.25：假定希望定义 StrBlob 的类值版本，而且希望继续使用 shared_ptr，

这样我们的 `StrBlobPtr` 类就仍能使用指向 `vector` 的 `weak_ptr` 了。你修改后的类将需要一个拷贝构造函数和一个拷贝赋值运算符，但不需要析构函数。解释拷贝构造函数和拷贝赋值运算符必须要做什么。解释为什么不需要析构函数。

练习 13.26：对上一题中描述的 `StrBlob` 类，编写你自己的版本。

13.2.2 定义行为像指针的类



对于行为类似指针的类，我们需要为其定义拷贝构造函数和拷贝赋值运算符，来拷贝指针成员本身而不是它指向的 `string`。我们的类仍然需要自己的析构函数来释放接受 `string` 参数的构造函数分配的内存（参见 13.1.4 节，第 447 页）。但是，在本例中，析构函数不能单方面地释放关联的 `string`。只有当最后一个指向 `string` 的 `HasPtr` 销毁时，它才可以释放 `string`。

令一个类展现类似指针的行为的最好方法是使用 `shared_ptr` 来管理类中的资源。拷贝（或赋值）一个 `shared_ptr` 会拷贝（赋值）`shared_ptr` 所指向的指针。<514>
`shared_ptr` 类自己记录有多少用户共享它所指向的对象。当没有用户使用对象时，`shared_ptr` 类负责释放资源。

但是，有时我们希望直接管理资源。在这种情况下，使用引用计数（reference count）（参见 12.1.1 节，第 402 页）就很有用了。为了说明引用计数如何工作，我们将重新定义 `HasPtr`，令其行为像指针一样，但我们不使用 `shared_ptr`，而是设计自己的引用计数。

引用计数

引用计数的工作方式如下：

- 除了初始化对象外，每个构造函数（拷贝构造函数除外）还要创建一个引用计数，用来记录有多少对象与正在创建的对象共享状态。当我们创建一个对象时，只有一个对象共享状态，因此将计数器初始化为 1。
- 拷贝构造函数不分配新的计数器，而是拷贝给定对象的数据成员，包括计数器。拷贝构造函数递增共享的计数器，指出给定对象的状态又被一个新用户所共享。
- 析构函数递减计数器，指出共享状态的用户少了一个。如果计数器变为 0，则析构函数释放状态。
- 拷贝赋值运算符递增右侧运算对象的计数器，递减左侧运算对象的计数器。如果左侧运算对象的计数器变为 0，意味着它的共享状态没有用户了，拷贝赋值运算符就必须销毁状态。

唯一的难题是确定在哪里存放引用计数。计数器不能直接作为 `HasPtr` 对象的成员。下面的例子说明了原因：

```
HasPtr p1("Hiya!");
HasPtr p2(p1); // p1 和 p2 指向相同的 string
HasPtr p3(p1); // p1、p2 和 p3 都指向相同的 string
```

如果引用计数保存在每个对象中，当创建 `p3` 时我们应该如何正确更新它呢？可以递增 `p1` 中的计数器并将其拷贝到 `p3` 中，但如何更新 `p2` 中的计数器呢？

解决此问题的一种方法是将计数器保存在动态内存中。当创建一个对象时，我们也分配一个新的计数器。当拷贝或赋值对象时，我们拷贝指向计数器的指针。使用这种方法，副本和原对象都会指向相同的计数器。

定义一个使用引用计数的类

通过使用引用计数，我们就可以编写类指针的 HasPtr 版本了：

```
515> class HasPtr {
public:
    // 构造函数分配新的 string 和新的计数器，将计数器置为 1
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0), use(new std::size_t(1)) {}
    // 拷贝构造函数拷贝所有三个数据成员，并递增计数器
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& operator=(const HasPtr &);

    ~HasPtr();

private:
    std::string *ps;
    int i;
    std::size_t *use; // 用来记录有多少个对象共享*ps 的成员
};
```

在此，我们添加了一个名为 `use` 的数据成员，它记录有多少对象共享相同的 `string`。接受 `string` 参数的构造函数分配新的计数器，并将其初始化为 1，指出当前有一个用户使用本对象的 `string` 成员。

类指针的拷贝成员“篡改”引用计数

当拷贝或赋值一个 `HasPtr` 对象时，我们希望副本和原对象都指向相同的 `string`。即，当拷贝一个 `HasPtr` 时，我们将拷贝 `ps` 本身，而不是 `ps` 指向的 `string`。当我们进行拷贝时，还会递增该 `string` 关联的计数器。

(我们在类内定义的) 拷贝构造函数拷贝给定 `HasPtr` 的所有三个数据成员。这个构造函数还递增 `use` 成员，指出 `ps` 和 `p.ps` 指向的 `string` 又有了一个新的用户。

析构函数不能无条件地 `delete ps`——可能还有其他对象指向这块内存。析构函数应该递减引用计数，指出共享 `string` 的对象少了一个。如果计数器变为 0，则析构函数释放 `ps` 和 `use` 指向的内存：

```
HasPtr::~HasPtr()
{
    if (--*use == 0) { // 如果引用计数变为 0
        delete ps; // 释放 string 内存
        delete use; // 释放计数器内存
    }
}
```

拷贝赋值运算符与往常一样执行类似拷贝构造函数和析构函数的工作。即，它必须递增右侧运算对象的引用计数（即，拷贝构造函数的工作），并递减左侧运算对象的引用计数，在必要时释放使用的内存（即，析构函数的工作）。

而且与往常一样，赋值运算符必须处理自赋值。我们通过先递增 `rhs` 中的计数然后递减左侧运算对象中的计数来实现这一点。通过这种方法，当两个对象相同时，在我们检查 `ps`（及 `use`）是否应该释放之前，计数器就已经被递增过了：

```
516> HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    if (this != &rhs) {
```

```

++*rhs.use; // 递增右侧运算对象的引用计数
if (--*use == 0) { // 然后递减本对象的引用计数
    delete ps; // 如果没有其他用户
    delete use; // 释放本对象分配的成员
}
ps = rhs.ps; // 将数据从 rhs 拷贝到本对象
i = rhs.i;
use = rhs.use;
return *this; // 返回本对象
}

```

13.2.2 节练习

练习 13.27: 定义你自己的使用引用计数版本的 HasPtr。

练习 13.28: 给定下面的类，为其实现一个默认构造函数和必要的拷贝控制成员。

<p>(a) class TreeNode { private: std::string value; int count; TreeNode *left; TreeNode *right; };</p>	<p>(b) class BinStrTree { private: TreeNode *root; };</p>
--	--

13.3 交换操作

除了定义拷贝控制成员，管理资源的类通常还定义一个名为 swap 的函数（参见 9.2.5 节，第 303 页）。对于那些与重排元素顺序的算法（参见 10.2.3 节，第 342 页）一起使用的类，定义 swap 是非常重要的。这类算法在需要交换两个元素时会调用 swap。

如果一个类定义了自己的 swap，那么算法将使用类自定义版本。否则，算法将使用标准库定义的 swap。虽然与往常一样我们不知道 swap 是如何实现的，但理论上很容易理解，为了交换两个对象我们需要进行一次拷贝和两次赋值。例如，交换两个类值 HasPtr 对象（参见 13.2.1 节，第 453 页）的代码可能像下面这样：

```

HasPtr temp = v1; // 创建 v1 的值的一个临时副本
v1 = v2; // 将 v2 的值赋予 v1
v2 = temp; // 将保存的 v1 的值赋予 v2

```

这段代码将原来 v1 中的 string 拷贝了两次——第一次是 HasPtr 的拷贝构造函数将 v1 拷贝给 temp，第二次是赋值运算符将 temp 赋予 v2。将 v2 赋予 v1 的语句还拷贝了原来 v2 中的 string。如我们所见，拷贝一个类值的 HasPtr 会分配一个新 string 并将其拷贝到 HasPtr 指向的位置。517

理论上，这些内存分配都是不必要的。我们更希望 swap 交换指针，而不是分配 string 的新副本。即，我们希望这样交换两个 HasPtr：

```

string *temp = v1.ps; // 为 v1.ps 中的指针创建一个副本
v1.ps = v2.ps; // 将 v2.ps 中的指针赋予 v1.ps
v2.ps = temp; // 将保存的 v1.ps 中原来的指针赋予 v2.ps

```

编写我们自己的 swap 函数

可以在我们的类上定义一个自己版本的 swap 来重载 swap 的默认行为。swap 的典型实现如下：

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // 其他成员定义，与 13.2.1 节（第 453 页）中一样
};

inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps);      // 交换指针，而不是 string 数据
    swap(lhs.i, rhs.i);        // 交换 int 成员
}
```

我们首先将 swap 定义为 friend，以便能访问 HasPtr 的（private 的）数据成员。由于 swap 的存在就是为了优化代码，我们将其声明为 inline 函数（参见 6.5.2 节，第 213 页）。swap 的函数体对给定对象的每个数据成员调用 swap。我们首先 swap 绑定到 rhs 和 lhs 的对象的指针成员，然后是 int 成员。



与拷贝控制成员不同，swap 并不是必要的。但是，对于分配了资源的类，定义 swap 可能是一种很重要的优化手段。



swap 函数应该调用 swap，而不是 std::swap

此代码中有一个很重要的微妙之处：虽然这一点在这个特殊的例子中并不重要，但在一般情况下它非常重要——swap 函数中调用的 swap 不是 std::swap。在本例中，数据成员是内置类型的，而内置类型是没有特定版本的 swap 的，所以在本例中，对 swap 的调用会调用标准库 std::swap。

518 但是，如果一个类的成员有自己类型特定的 swap 函数，调用 std::swap 就是错误的了。例如，假定我们有另一个命名为 Foo 的类，它有一个类型为 HasPtr 的成员 h。如果我们未定义 Foo 版本的 swap，那么就会使用标准库版本的 swap。如我们所见，标准库 swap 对 HasPtr 管理的 string 进行了不必要的拷贝。

我们可以为 Foo 编写一个 swap 函数，来避免这些拷贝。但是，如果这样编写 Foo 版本的 swap：

```
void swap(Foo &lhs, Foo &rhs)
{
    // 错误：这个函数使用了标准库版本的 swap，而不是 HasPtr 版本
    std::swap(lhs.h, rhs.h);
    // 交换类型 Foo 的其他成员
}
```

此编码会编译通过，且正常运行。但是，使用此版本与简单使用默认版本的 swap 并没有任何性能差异。问题在于我们显式地调用了标准库版本的 swap。但是，我们不希望使用 std 中的版本，我们希望调用为 HasPtr 对象定义的版本。

正确的 swap 函数如下所示：

```
void swap(Foo &lhs, Foo &rhs)
```

```
{  
    using std::swap;  
    swap(lhs.h, rhs.h); // 使用 HasPtr 版本的 swap  
    // 交换类型 Foo 的其他成员  
}
```

每个 `swap` 调用应该都是未加限定的。即，每个调用都应该是 `swap`，而不是 `std::swap`。如果存在类型特定的 `swap` 版本，其匹配程度会优于 `std` 中定义的版本，原因我们将在 16.3 节（第 616 页）中进行解释。因此，如果存在类型特定的 `swap` 版本，`swap` 调用会与之匹配。如果不存在类型特定的版本，则会使用 `std` 中的版本（假定作用域中有 `using` 声明）。

非常仔细的读者可能会奇怪为什么 `swap` 函数中的 `using` 声明没有隐藏 `HasPtr` 版本 `swap` 的声明（参见 6.4.1 节，第 210 页）。我们将在 18.2.3 节（第 706 页）中解释为什么这段代码能正常工作。

在赋值运算符中使用 `swap`

定义 `swap` 的类通常用 `swap` 来定义它们的赋值运算符。这些运算符使用了一种名为拷贝并交换（copy and swap）的技术。这种技术将左侧运算对象与右侧运算对象的一个副本进行交换：

```
// 注意 rhs 是按值传递的，意味着 HasPtr 的拷贝构造函数  
// 将右侧运算对象中的 string 拷贝到 rhs  
HasPtr& HasPtr::operator=(HasPtr rhs)  
{  
    // 交换左侧运算对象和局部变量 rhs 的内容  
    swap(*this, rhs); // rhs 现在指向本对象曾经使用的内存  
    return *this; // rhs 被销毁，从而 delete 了 rhs 中的指针  
}
```

在这个版本的赋值运算符中，参数并不是一个引用，我们将右侧运算对象以传值方式传递给了赋值运算符。因此，`rhs` 是右侧运算对象的一个副本。参数传递时拷贝 `HasPtr` 的操作会分配该对象的 `string` 的一个新副本。519

在赋值运算符的函数体中，我们调用 `swap` 来交换 `rhs` 和 `*this` 中的数据成员。这个调用将左侧运算对象中原来保存的指针存入 `rhs` 中，并将 `rhs` 中原来的指针存入 `*this` 中。因此，在 `swap` 调用之后，`*this` 中的指针成员将指向新分配的 `string`——右侧运算对象中 `string` 的一个副本。

当赋值运算符结束时，`rhs` 被销毁，`HasPtr` 的析构函数将执行。此析构函数 `delete rhs` 现在指向的内存，即，释放掉左侧运算对象中原来的内存。

这个技术的有趣之处是它自动处理了自赋值情况且天然就是异常安全的。它通过在改变左侧运算对象之前拷贝右侧运算对象保证了自赋值的正确，这与我们在原来的赋值运算符中使用的方法是一致的（参见 13.2.1 节，第 453 页）。它保证异常安全的方法也与原来的赋值运算符实现一样。代码中唯一可能抛出异常的是拷贝构造函数中的 `new` 表达式。如果真发生了异常，它也会在我们改变左侧运算对象之前发生。



使用拷贝和交换的赋值运算符自动就是异常安全的，且能正确处理自赋值。

13.3 节练习

练习 13.29: 解释 swap(HasPtr&, HasPtr&) 中对 swap 的调用不会导致递归循环。

练习 13.30: 为你的类值版本的 HasPtr 编写 swap 函数, 并测试它。为你的 swap 函数添加一个打印语句, 指出函数什么时候执行。

练习 13.31: 为你的 HasPtr 类定义一个<运算符, 并定义一个 HasPtr 的 vector。为这个 vector 添加一些元素, 并对它执行 sort。注意何时会调用 swap。

练习 13.32: 类指针的 HasPtr 版本会从 swap 函数受益吗? 如果会, 得到了什么益处? 如果不是, 为什么?

13.4 拷贝控制示例

虽然通常来说分配资源的类更需要拷贝控制, 但资源管理并不是一个类需要定义自己的拷贝控制成员的唯一原因。一些类也需要拷贝控制成员的帮助来进行簿记工作或其他操作。

作为类需要拷贝控制来进行簿记操作的例子, 我们将概述两个类的设计, 这两个类可能用于邮件处理应用中。两个类命名为 Message 和 Folder, 分别表示电子邮件 (或者其他类型的) 消息和消息目录。每个 Message 对象可以出现在多个 Folder 中。但是, 任意给定的 Message 的内容只有一个副本。这样, 如果一条 Message 的内容被改变, 则我们从它所在的任何 Folder 来浏览此 Message 时, 都会看到改变后的内容。

为了记录 Message 位于哪些 Folder 中, 每个 Message 都会保存一个它所在 Folder 的指针的 set, 同样的, 每个 Folder 都保存一个它包含的 Message 的指针的 set。图 13.1 说明了这种设计思路。

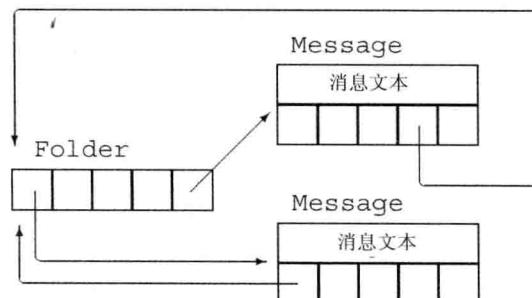


图 13.1: Message 和 Folder 类设计

我们的 Message 类会提供 save 和 remove 操作, 来向一个给定 Folder 添加一条 Message 或是从中删除一条 Message。为了创建一个新的 Message, 我们会指明消息内容, 但不会指出 Folder。为了将一条 Message 放到一个特定 Folder 中, 我们必须调用 save。

当我们拷贝一个 Message 时, 副本和原对象将是不同的 Message 对象, 但两个 Message 都出现在相同的 Folder 中。因此, 拷贝 Message 的操作包括消息内容和 Folder 指针 set 的拷贝。而且, 我们必须在每个包含此消息的 Folder 中都添加一个指向新创建的 Message 的指针。

当我们销毁一个 Message 时, 它将不复存在。因此, 我们必须从包含此消息的所有

Folder 中删除指向此 Message 的指针。

当我们把一个 Message 对象赋予另一个 Message 对象时，左侧 Message 的内容会被右侧 Message 的内容所替代。我们还必须更新 Folder 集合，从原来包含左侧 Message 的 Folder 中将它删除，并将它添加到包含右侧 Message 的 Folder 中。

观察这些操作，我们可以看到，析构函数和拷贝赋值运算符都必须从包含一条 Message 的所有 Folder 中删除它。类似的，拷贝构造函数和拷贝赋值运算符都要将一个 Message 添加到给定的一组 Folder 中。我们将定义两个 private 的工具函数来完成这些工作。



拷贝赋值运算符通常执行拷贝构造函数和析构函数中也要做的工作。这种情况下，公共的工作应该放在 private 的工具函数中完成。

Folder 类也需要类似的拷贝控制成员，来添加或删除它保存的 Message。

521

我们将 Folder 类的设计和实现留作练习。但是，我们将假定 Folder 类包含名为 addMsg 和 remMsg 的成员，分别完成在给定 Folder 对象的消息集合中添加和删除 Message 的工作。

Message 类

根据上述设计，我们可以编写 Message 类，如下所示：

```
class Message {
    friend class Folder;
public:
    // folders 被隐式初始化为空集合
    explicit Message(const std::string &str = "") :
        contents(str) { }
    // 拷贝控制成员，用来管理指向本 Message 的指针
    Message(const Message&);           // 拷贝构造函数
    Message& operator=(const Message&); // 拷贝赋值运算符
    ~Message();                         // 析构函数
    // 从给定 Folder 集合中添加/删除本 Message
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents;           // 实际消息文本
    std::set<Folder*> folders;    // 包含本 Message 的 Folder
    // 拷贝构造函数、拷贝赋值运算符和析构函数所使用的工具函数
    // 将本 Message 添加到指向参数的 Folder 中
    void add_to_Folders(const Message&);
    // 从 folders 中的每个 Folder 中删除本 Message
    void remove_from_Folders();
};
```

这个类定义了两个数据成员：contents，保存消息文本；folders，保存指向本 Message 所在 Folder 的指针。接受一个 string 参数的构造函数将给定 string 拷贝给 contents，并将 folders（隐式）初始化为空集。由于此构造函数有一个默认参数，因此它也被当作 Message 的默认构造函数（参见 7.5.1 节，第 260 页）。

save 和 remove 成员

除拷贝控制成员外，Message 类只有两个公共成员：save，将本 Message 存放在给定 Folder 中；remove，删除本 Message：

```
void Message::save(Folder &f)
{
    folders.insert(&f); // 将给定 Folder 添加到我们的 Folder 列表中
    f.addMsg(this); // 将本 Message 添加到 f 的 Message 集合中
}
522 void Message::remove(Folder &f)
{
    folders.erase(&f); // 将给定 Folder 从我们的 Folder 列表中删除
    f.remMsg(this); // 将本 Message 从 f 的 Message 集合中删除
}
```

为了保存（或删除）一个 Message，需要更新本 Message 的 folders 成员。当 save 一个 Message 时，我们应保存一个指向给定 Folder 的指针；当 remove 一个 Message 时，我们要删除此指针。

这些操作还必须更新给定的 Folder。更新一个 Folder 的任务是由 Folder 类的 addMsg 和 remMsg 成员来完成的，分别添加和删除给定 Message 的指针。

Message 类的拷贝控制成员

当我们拷贝一个 Message 时，得到的副本应该与原 Message 出现在相同的 Folder 中。因此，我们必须遍历 Folder 指针的 set，对每个指向原 Message 的 Folder 添加一个指向新 Message 的指针。拷贝构造函数和拷贝赋值运算符都需要做这个工作，因此我们定义一个函数来完成这个公共操作：

```
// 将本 Message 添加到指向 m 的 Folder 中
void Message::add_to_Folders(const Message &m)
{
    for (auto f : m.folders) // 对每个包含 m 的 Folder
        f->addMsg(this); // 向该 Folder 添加一个指向本 Message 的指针
}
```

此例中我们对 m.folders 中每个 Folder 调用 addMsg。函数 addMsg 会将本 Message 的指针添加到每个 Folder 中。

Message 的拷贝构造函数拷贝给定对象的数据成员：

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    add_to_Folders(m); // 将本消息添加到指向 m 的 Folder 中
}
```

并调用 add_to_Folders 将新创建的 Message 的指针添加到每个包含原 Message 的 Folder 中。

Message 的析构函数

当一个 Message 被销毁时，我们必须从指向此 Message 的 Folder 中删除它。拷贝赋值运算符也要执行此操作，因此我们会定义一个公共函数来完成此工作：

```
// 从对应的 Folder 中删除本 Message
void Message::remove_from_Folders()
{
    for (auto f : folders) // 对 folders 中每个指针
        f->remMsg(this); // 从该 Folder 中删除本 Message
}
```

函数 `remove_from_Folders` 的实现类似 `add_to_Folders`, 不同之处是它调用 [523](#) `remMsg` 来删除当前 `Message` 而不是调用 `addMsg` 来添加 `Message`。

有了 `remove_from_Folders` 函数, 编写析构函数就很简单了:

```
Message::~Message()
{
    remove_from_Folders();
}
```

调用 `remove_from_Folders` 确保没有任何 `Folder` 保存正在销毁的 `Message` 的指针。编译器自动调用 `string` 的析构函数来释放 `contents`, 并自动调用 `set` 的析构函数来清理集合成员使用的内存。

Message 的拷贝赋值运算符

与大多数赋值运算符相同, 我们的 `Message` 类的拷贝赋值运算符必须执行拷贝构造函数和析构函数的工作。与往常一样, 最重要的是我们要组织好代码结构, 使得即使左侧和右侧运算对象是同一个 `Message`, 拷贝赋值运算符也能正确执行。

在本例中, 我们先从左侧运算对象的 `folders` 中删除此 `Message` 的指针, 然后再将指针添加到右侧运算对象的 `folders` 中, 从而实现了自赋值的正确处理:

```
Message& Message::operator=(const Message &rhs)
{
    // 通过先删除指针再插入它们来处理自赋值情况
    remove_from_Folders(); // 更新已有 Folder
    contents = rhs.contents; // 从 rhs 拷贝消息内容
    folders = rhs.folders; // 从 rhs 拷贝 Folder 指针
    add_to_Folders(rhs); // 将本 Message 添加到那些 Folder 中
    return *this;
}
```

如果左侧和右侧运算对象是相同的 `Message`, 则它们具有相同的地址。如果我们在 `add_to_Folders` 之后调用 `remove_from_Folders`, 就会将此 `Message` 从它所在的所有 `Folder` 中删除。

Message 的 swap 函数

标准库中定义了 `string` 和 `set` 的 `swap` 版本 (参见 9.2.5 节, 第 303 页)。因此, 如果为我们的 `Message` 类定义它自己的 `swap` 版本, 它将从中受益。通过定义一个 `Message` 特定版本的 `swap`, 我们可以避免对 `contents` 和 `folders` 成员进行不必要的拷贝。

但是, 我们的 `swap` 函数必须管理指向被交换 `Message` 的 `Folder` 指针。在调用 `swap(m1, m2)` 之后, 原来指向 `m1` 的 `Folder` 现在必须指向 `m2`, 反之亦然。

我们通过两遍扫描 `folders` 中每个成员来正确处理 `Folder` 指针。第一遍扫描将 `Message` 从它们所在的 `Folder` 中删除。接下来我们调用 `swap` 来交换数据成员。最后

对 `folders` 进行第二遍扫描来添加交换过的 `Message`:

```
524> void swap(Message &lhs, Message &rhs)
{
    using std::swap; // 在本例中严格来说并不需要，但这是一个好习惯
    // 将每个消息的指针从它（原来）所在 Folder 中删除
    for (auto f: lhs.folders)
        f->remMsg(&lhs);
    for (auto f: rhs.folders)
        f->remMsg(&rhs);
    // 交换 contents 和 Folder 指针 set
    swap(lhs.folders, rhs.folders);           // 使用 swap(set&, set&)
    swap(lhs.contents, rhs.contents);         // swap(string&, string&)
    // 将每个 Message 的指针添加到它的（新）Folder 中
    for (auto f: lhs.folders)
        f->addMsg(&lhs);
    for (auto f: rhs.folders)
        f->addMsg(&rhs);
}
```

13.4 节练习

练习 13.33: 为什么 `Message` 的成员 `save` 和 `remove` 的参数是一个 `Folder&`? 为什么我们不将参数定义为 `Folder` 或是 `const Folder&`?

练习 13.34: 编写本节所描述的 `Message`。

练习 13.35: 如果 `Message` 使用合成的拷贝控制成员，将会发生什么?

练习 13.36: 设计并实现对应的 `Folder` 类。此类应该保存一个指向 `Folder` 中包含的 `Message` 的 `set`。

练习 13.37: 为 `Message` 类添加成员，实现向 `folders` 添加或删除一个给定的 `Folder*`。这两个成员类似 `Folder` 类的 `addMsg` 和 `remMsg` 操作。

练习 13.38: 我们并未使用拷贝和交换方式来设计 `Message` 的赋值运算符。你认为其原因是什么?



13.5 动态内存管理类

某些类需要在运行时分配可变大小的内存空间。这种类通常可以（并且如果它们确实可以说的话，一般应该）使用标准库容器来保存它们的数据。例如，我们的 `StrBlob` 类使用一个 `vector` 来管理其元素的底层内存。

但是，这一策略并不是对每个类都适用；某些类需要自己进行内存分配。这些类一般来说必须定义自己的拷贝控制成员来管理所分配的内存。



例如，我们将实现标准库 `vector` 类的一个简化版本。我们所做的一个简化是不使用模板，我们的类只用于 `string`。因此，它被命名为 `StrVec`。

StrVec 类的设计

回忆一下，`vector` 类将其元素保存在连续内存中。为了获得可接受的性能，`vector`

预先分配足够的内存来保存可能需要的更多元素（参见 9.4 节，第 317 页）。`vector` 的每个添加元素的成员函数会检查是否有空间容纳更多的元素。如果有，成员函数会在下一个可用位置构造一个对象。如果没有可用空间，`vector` 就会重新分配空间：它获得新的空间，将已有元素移动到新空间中，释放旧空间，并添加新元素。

我们在 `StrVec` 类中使用类似的策略。我们将使用一个 `allocator` 来获得原始内存（参见 12.2.2 节，第 427 页）。由于 `allocator` 分配的内存是未构造的，我们将在需要添加新元素时用 `allocator` 的 `construct` 成员在原始内存中创建对象。类似的，当我们需要删除一个元素时，我们将使用 `destroy` 成员来销毁元素。

每个 `StrVec` 有三个指针成员指向其元素所使用的内存：

- `elements`, 指向分配的内存中的首元素
- `first_free`, 指向最后一个实际元素之后的位置
- `cap`, 指向分配的内存末尾之后的位置

图 13.2 说明了这些指针的含义。

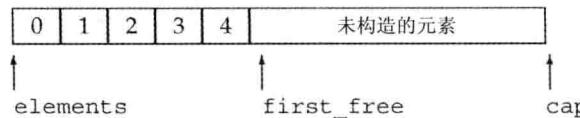


图 13.2: StrVec 内存分配策略

除了这些指针之外，`StrVec` 还有一个名为 `alloc` 的静态成员，其类型为 `allocator<string>`。`alloc` 成员会分配 `StrVec` 使用的内存。我们的类还有 4 个工具函数：

- `alloc_n_copy` 会分配内存，并拷贝一个给定范围中的元素。
- `free` 会销毁构造的元素并释放内存。
- `chk_n_alloc` 保证 `StrVec` 至少有容纳一个新元素的空间。如果没有空间添加新元素，`chk_n_alloc` 会调用 `reallocate` 来分配更多内存。
- `reallocate` 在内存用完时为 `StrVec` 分配新内存。

虽然我们关注的是类的实现，但我们将定义 `vector` 接口中的一些成员。

StrVec 类定义

526

有了上述实现概要，我们现在可以定义 `StrVec` 类，如下所示：

```
// 类 vector 类内存分配策略的简化实现
class StrVec {
public:
    StrVec(): // allocator 成员进行默认初始化
        elements(nullptr), first_free(nullptr), cap(nullptr) { }
    StrVec(const StrVec&); // 拷贝构造函数
    StrVec &operator=(const StrVec&); // 拷贝赋值运算符
    ~StrVec(); // 析构函数
    void push_back(const std::string&); // 拷贝元素
    size_t size() const { return first_free - elements; }
    size_t capacity() const { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
```

```

    // ...
private:
    Static std::allocator<std::string> alloc; // 分配元素
    // 被添加元素的函数所使用
    void chk_n_alloc()
    {
        if (size() == capacity()) reallocate();
    }
    // 工具函数，被拷贝构造函数、赋值运算符和析构函数所使用
    std::pair<std::string*, std::string*> alloc_n_copy
        (const std::string*, const std::string*);
    void free(); // 销毁元素并释放内存
    void reallocate(); // 获得更多内存并拷贝已有元素
    std::string *elements; // 指向数组首元素的指针
    std::string *first_free; // 指向数组第一个空闲元素的指针
    std::string *cap; // 指向数组尾后位置的指针
};

};

类体定义了多个成员：
```

- 默认构造函数(隐式地)默认初始化 alloc 并(显式地)将指针初始化为 nullptr，表明没有元素。
- size 成员返回当前真正在使用的元素的数目，等于 first_free-elements。
- capacity 成员返回 StrVec 可以保存的元素的数量，等价于 cap-elements。
- 当没有空间容纳新元素，即 cap==first_free 时，chk_n_alloc 会为 StrVec 重新分配内存。
- begin 和 end 成员分别返回指向首元素(即 elements)和最后一个构造的元素之后位置(即 first_free)的指针。

使用 construct

函数 push_back 调用 chk_n_alloc 确保有空间容纳新元素。如果需要，
527 chk_n_alloc 会调用 reallocate。当 chk_n_alloc 返回时，push_back 知道必有空间容纳新元素。它要求其 allocator 成员来 construct 新的尾元素：

```

void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的副本
    alloc.construct(first_free++, s);
}

```

当我们用 allocator 分配内存时，必须记住内存是未构造的(参见 12.2.2 节，第 428 页)。为了使用此原始内存，我们必须调用 construct，在此内存中构造一个对象。传递给 construct 的第一个参数必须是一个指针，指向调用 allocate 所分配的未构造的内存空间。剩余参数确定用哪个构造函数来构造对象。在本例中，只有一个额外参数，类型为 string，因此会使用 string 的拷贝构造函数。

值得注意的是，对 construct 的调用也会递增 first_free，表示已经构造了一个新元素。它使用前置递增(参见 4.5 节，第 131 页)，因此这个调用会在 first_free 当前值指定的地址构造一个对象，并递增 first_free 指向下一个未构造的元素。

alloc_n_copy 成员

我们在拷贝或赋值 StrVec 时，可能会调用 alloc_n_copy 成员。类似 vector，我们的 StrVec 类有类值的行为（参见 13.2.1 节，第 453 页）。当我们拷贝或赋值 StrVec 时，必须分配独立的内存，并从原 StrVec 对象拷贝元素至新对象。

alloc_n_copy 成员会分配足够的内存来保存给定范围的元素，并将这些元素拷贝到新分配的内存中。此函数返回一个指针的 pair（参见 11.2.3 节，第 379 页），两个指针分别指向新空间的开始位置和拷贝的尾后的位置：

```
pair<string*, string*>
StrVec::alloc_n_copy(const string *b, const string *e)
{
    // 分配空间保存给定范围中的元素
    auto data = alloc.allocate(e - b);
    // 初始化并返回一个 pair，该 pair 由 data 和 uninitialized_copy 的返回值构成
    return {data, uninitialized_copy(b, e, data)};
}
```

alloc_n_copy 用尾后指针减去首元素指针，来计算需要多少空间。在分配内存之后，它必须在此空间中构造给定元素的副本。

它是在返回语句中完成拷贝工作的，返回语句中对返回值进行了列表初始化（参见 6.3.2 节，第 203 页）。返回的 pair 的 first 成员指向分配的内存的开始位置；second 成员则是 uninitialized_copy（参见 12.2.2 节，第 429 页）的返回值，此值是一个指针，指向最后一个构造元素之后的位置。528

free 成员

free 成员有两个责任：首先 destroy 元素，然后释放 StrVec 自己分配的内存空间。for 循环调用 allocator 的 destroy 成员，从构造的尾元素开始，到首元素为止，逆序销毁所有元素：

```
void StrVec::free()
{
    // 不能传递给 deallocate 一个空指针，如果 elements 为 0，函数什么也不做
    if (elements) {
        // 逆序销毁旧元素
        for (auto p = first_free; p != elements; /* 空 */)
            alloc.destroy(--p);
        alloc.deallocate(elements, cap - elements);
    }
}
```

destroy 函数会运行 string 的析构函数。string 的析构函数会释放 string 自己分配的内存空间。

一旦元素被销毁，我们就调用 deallocate 来释放本 StrVec 对象分配的内存空间。我们传递给 deallocate 的指针必须是之前某次 allocate 调用所返回的指针。因此，在调用 deallocate 之前我们首先检查 elements 是否为空。

拷贝控制成员

实现了 alloc_n_copy 和 free 成员后，为我们的类实现拷贝控制成员就很简单了。

拷贝构造函数调用 alloc_n_copy:

```
StrVec::StrVec(const StrVec &s)
{
    // 调用 alloc_n_copy 分配空间以容纳与 s 中一样多的元素
    auto newdata = alloc_n_copy(s.begin(), s.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}
```

并将返回结果赋予数据成员。alloc_n_copy 的返回值是一个指针的 pair。其 first 成员指向第一个构造的元素，second 成员指向最后一个构造的元素之后的位置。由于 alloc_n_copy 分配的空间恰好容纳给定的元素，cap 也指向最后一个构造的元素之后的位置。

析构函数调用 free:

```
StrVec::~StrVec() { free(); }
```

拷贝赋值运算符在释放已有元素之前调用 alloc_n_copy，这样就可以正确处理自赋值了：

529

```
StrVec &StrVec::operator=(const StrVec &rhs)
{
    // 调用 alloc_n_copy 分配内存，大小与 rhs 中元素占用空间一样多
    auto data = alloc_n_copy(rhs.begin(), rhs.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

类似拷贝构造函数，拷贝赋值运算符使用 alloc_n_copy 的返回值来初始化它的指针。

在重新分配内存的过程中移动而不是拷贝元素

在编写 reallocate 成员函数之前，我们稍微思考一下此函数应该做什么。它应该

- 为一个新的、更大的 string 数组分配内存
- 在内存空间的前一部分构造对象，保存现有元素
- 销毁原内存空间中的元素，并释放这块内存

观察这个操作步骤，我们可以看出，为一个 StrVec 重新分配内存空间会引起从旧内存空间到新内存空间逐个拷贝 string。虽然我们不知道 string 的实现细节，但我们知道 string 具有类值行为。当拷贝一个 string 时，新 string 和原 string 是相互独立的。改变原 string 不会影响到副本，反之亦然。

由于 string 的行为类似值，我们可以得出结论，每个 string 对构成它的所有字符都会保存自己的一份副本。拷贝一个 string 必须为这些字符分配内存空间，而销毁一个 string 必须释放所占用的内存。

拷贝一个 string 就必须真的拷贝数据，因为通常情况下，在我们拷贝了一个 string 之后，它就会有两个用户。但是，如果是 reallocate 拷贝 StrVec 中的 string，则在拷贝之后，每个 string 只有唯一的用户。一旦将元素从旧空间拷贝到了新空间，我们就会立即销毁原 string。

因此，拷贝这些 `string` 中的数据是多余的。在重新分配内存空间时，如果我们能避免分配和释放 `string` 的额外开销，`StrVec` 的性能会好得多。

移动构造函数和 `std::move`

通过使用新标准库引入的两种机制，我们就可以避免 `string` 的拷贝。首先，有一些标准库类，包括 `string`，都定义了所谓的“移动构造函数”。关于 `string` 的移动构造函数如何工作的细节，以及有关实现的任何其他细节，目前都尚未公开。但是，我们知道，移动构造函数通常是将资源从给定对象“移动”而不是拷贝到正在创建的对象。而且我们知道标准库保证“移后源”（moved-from）`string` 仍然保持一个有效的、可析构的状态。对于 `string`，我们可以想象每个 `string` 都有一个指向 `char` 数组的指针。可以假定 `string` 的移动构造函数进行了指针的拷贝，而不是为字符分配内存空间然后拷贝字符。

C++ 11

< 530

我们使用的第二个机制是一个名为 `move` 的标准库函数，它定义在 `utility` 头文件中。目前，关于 `move` 我们需要了解两个关键点。首先，当 `reallocate` 在新内存中构造 `string` 时，它必须调用 `move` 来表示希望使用 `string` 的移动构造函数，原因我们将在 13.6.1 节（第 470 页）中解释。如果它漏掉了 `move` 调用，将会使用 `string` 的拷贝构造函数。其次，我们通常不为 `move` 提供一个 `using` 声明（参见 3.1 节，第 74 页），原因我们将在 18.2.3 节（第 706 页）中解释。当我们使用 `move` 时，直接调用 `std::move` 而不是 `move`。

`reallocate` 成员

了解了这些知识，现在就可以编写 `reallocate` 成员了。首先调用 `allocate` 分配新内存空间。我们每次重新分配内存时都会将 `StrVec` 的容量加倍。如果 `StrVec` 为空，我们将分配容纳一个元素的空间：

```
void StrVec::reallocate()
{
    // 我们将分配当前大小两倍的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    // 分配新内存
    auto newdata = alloc.allocate(newcapacity);
    // 将数据从旧内存移动到新内存
    auto dest = newdata;        // 指向新数组中下一个空闲位置
    auto elem = elements;      // 指向旧数组中下一个元素
    for (size_t i = 0; i != size(); ++i)
        alloc.construct(dest++, std::move(*elem++));
    free(); // 一旦我们移动完元素就释放旧内存空间
    // 更新我们的数据结构，执行新元素
    elements = newdata;
    first_free = dest;
    cap = elements + newcapacity;
}
```

`for` 循环遍历每个已有元素，并在新内存空间中 `construct` 一个对应元素。我们使用 `dest` 指向构造新 `string` 的内存，使用 `elem` 指向原数组中的元素。我们每次用后置递增运算将 `dest`（和 `elem`）推进到各自数组中的下一个元素。

`construct` 的第二个参数（即，确定使用哪个构造函数的参数（参见 12.2.2 节，第 428 页））是 `move` 返回的值。调用 `move` 返回的结果会令 `construct` 使用 `string` 的移

动构造函数。由于我们使用了移动构造函数，这些 `string` 管理的内存将不会被拷贝。相反，我们构造的每个 `string` 都会从 `elem` 指向的 `string` 那里接管内存的所有权。

[531] 在元素移动完毕后，我们调用 `free` 销毁旧元素并释放 `StrVec` 原来使用的内存。`string` 成员不再管理它们曾经指向的内存；其数据的管理职责已经转移给新 `StrVec` 内存中的元素了。我们不知道旧 `StrVec` 内存中的 `string` 包含什么值，但我们保证对它们执行 `string` 的析构函数是安全的。

剩下的就是更新指针，指向新分配并已初始化过的数组了。`first_free` 和 `cap` 指针分别被设置为指向最后一个构造的元素之后的位置及指向新分配空间的尾后位置。

13.5 节练习

练习 13.39：编写你自己版本的 `StrVec`，包括自己版本的 `reserve`、`capacity`（参见 9.4 节，第 318 页）和 `resize`（参见 9.3.5 节，第 314 页）。

练习 13.40：为你的 `StrVec` 类添加一个构造函数，它接受一个 `initializer_list<string>` 参数。

练习 13.41：在 `push_back` 中，我们为什么在 `construct` 调用中使用前置递增运算？如果使用后置递增运算的话，会发生什么？

练习 13.42：在你的 `TextQuery` 和 `QueryResult` 类（参见 12.3 节，第 431 页）中用你的 `StrVec` 类代替 `vector<string>`，以此来测试你的 `StrVec` 类。

练习 13.43：重写 `free` 成员，用 `for_each` 和 `lambda`（参见 10.3.2 节，第 346 页）来代替 `for` 循环 `destroy` 元素。你更倾向于哪种实现，为什么？

练习 13.44：编写标准库 `string` 类的简化版本，命名为 `String`。你的类应该至少有一个默认构造函数和一个接受 C 风格字符串指针参数的构造函数。使用 `allocator` 为你的 `String` 类分配所需内存。



13.6 对象移动

新标准的一个最主要的特性是可以移动而非拷贝对象的能力。如我们在 13.1.1 节（第 440 页）中所见，很多情况下都会发生对象拷贝。在其中某些情况下，对象拷贝后就立即被销毁了。在这些情况下，移动而非拷贝对象会大幅度提升性能。

如我们已经看到的，我们的 `StrVec` 类是这种不必要的拷贝的一个很好的例子。在重新分配内存的过程中，从旧内存将元素拷贝到新内存是不必要的，更好的方式是移动元素。使用移动而不是拷贝的另一个原因源于 `IO` 类或 `unique_ptr` 这样的类。这些类都包含不能被共享的资源（如指针或 `IO` 缓冲）。因此，这些类型的对象不能拷贝但可以移动。

[532]

在旧 C++ 标准中，没有直接的方法移动对象。因此，即使不必拷贝对象的情况下，我们也不得不拷贝。如果对象较大，或者是对象本身要求分配内存空间（如 `string`），进行不必要的拷贝代价非常高。类似的，在旧版本的标准库中，容器中所保存的类必须是可拷贝的。但在新标准中，我们可以用容器保存不可拷贝的类型，只要它们能被移动即可。



标准库容器、`string` 和 `shared_ptr` 类既支持移动也支持拷贝。IO 类和 `unique_ptr` 类可以移动但不能拷贝。

13.6.1 右值引用



C++ 11

为了支持移动操作，新标准引入了一种新的引用类型——右值引用（rvalue reference）。所谓右值引用就是必须绑定到右值的引用。我们通过`&&`而不是`&`来获得右值引用。如我们将要看到的，右值引用有一个重要的性质——只能绑定到一个将要销毁的对象。因此，我们可以自由地将一个右值引用的资源“移动”到另一个对象中。

回忆一下，左值和右值是表达式的属性（参见 4.1.1 节，第 121 页）。一些表达式生成或要求左值，而另外一些则生成或要求右值。一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值。

类似任何引用，一个右值引用也不过是某个对象的另一个名字而已。如我们所知，对于常规引用（为了与右值引用区分开来，我们可以称之为左值引用（lvalue reference）），我们不能将其绑定到要求转换的表达式、字面常量或是返回右值的表达式（参见 2.3.1 节，第 46 页）。右值引用有着完全相反的绑定特性：我们可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上：

```
int i = 42;
int &r = i;           // 正确: r 引用 i
int &&rr = i;         // 错误: 不能将一个右值引用绑定到一个左值上
int &r2 = i * 42;    // 错误: i*42 是一个右值
const int &r3 = i * 42; // 正确: 我们可以将一个 const 的引用绑定到一个右值上
int &&rr2 = i * 42;  // 正确: 将 rr2 绑定到乘法结果上
```

返回左值引用的函数，连同赋值、下标、解引用和前置递增/递减运算符，都是返回左值的表达式的例子。我们可以将一个左值引用绑定到这类表达式的结果上。

返回非引用类型的函数，连同算术、关系、位以及后置递增/递减运算符，都生成右值。我们不能将一个左值引用绑定到这类表达式上，但我们可以将一个 `const` 的左值引用或者一个右值引用绑定到这类表达式上。

左值持久：右值短暂

533

考察左值和右值表达式的列表，两者相互区别之处就很明显了：左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

由于右值引用只能绑定到临时对象，我们得知

- 所引用的对象将要被销毁
- 该对象没有其他用户

这两个特性意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。



右值引用指向将要被销毁的对象。因此，我们可以从绑定到右值引用的对象“窃取”状态。

变量是左值

变量可以看作只有一个运算对象而没有运算符的表达式，虽然我们很少这样看待变

量。类似其他任何表达式，变量表达式也有左值/右值属性。变量表达式都是左值。带来的结果就是，我们不能将一个右值引用绑定到一个右值引用类型的变量上，这有些令人惊讶：

```
int &&rr1 = 42; // 正确：字面常量是右值
int &&rr2 = rr1; // 错误：表达式 rr1 是左值！
```

其实有了右值表示临时对象这一观察结果，变量是左值这一特性并不令人惊讶。毕竟，变量是持久的，直至离开作用域时才被销毁。



变量是左值，因此我们不能将一个右值引用直接绑定到一个变量上，即使这个变量是右值引用类型也不行。

标准库 move 函数

C++ 11

虽然不能将一个右值引用直接绑定到一个左值上，但我们可以显式地将一个左值转换为对应的右值引用类型。我们还可以通过调用一个名为 **move** 的新标准库函数来获得绑定到左值上的右值引用，此函数定义在头文件 utility 中。**move** 函数使用了我们将在 16.2.6 节（第 610 页）中描述的机制来返回给定对象的右值引用。

```
int &&rr3 = std::move(rr1); // ok
```

move 调用告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它。我们必须认识到，调用 **move** 就意味着承诺：除了对 **rr1** 赋值或销毁它外，我们将不再使用它。在调用 **move** 之后，我们不能对移后源对象的值做任何假设。

534



我们可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值。

如前所述，与大多数标准库名字的使用不同，对 **move**（参见 13.5 节，第 469 页）我们不提供 **using** 声明（参见 3.1 节，第 74 页）。我们直接调用 **std::move** 而不是 **move**，其原因将在 18.2.3 节（第 707 页）中解释。



使用 **move** 的代码应该使用 **std::move** 而不是 **move**。这样做可以避免潜在的名字冲突。

13.6.1 节练习

练习 13.45：解释右值引用和左值引用的区别。

练习 13.46：什么类型的引用可以绑定到下面的初始化器上？

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

练习 13.47：对你在练习 13.44（13.5 节，第 470 页）中定义的 **String** 类，为它的拷贝构造函数和拷贝赋值运算符添加一条语句，在每次函数执行时打印一条信息。

练习 13.48: 定义一个 `vector<String>` 并在其上多次调用 `push_back`。运行你的程序，并观察 `String` 被拷贝了多少次。

13.6.2 移动构造函数和移动赋值运算符



类似 `string` 类（及其他标准库类），如果我们自己的类也同时支持移动和拷贝，那么也能从中受益。为了让我们自己的类型支持移动操作，需要为其定义移动构造函数和移动赋值运算符。这两个成员类似对应的拷贝操作，但它们从给定对象“窃取”资源而不是拷贝资源。

类似拷贝构造函数，移动构造函数的第一个参数是该类类型的一个引用。不同于拷贝构造函数的是，这个引用参数在移动构造函数中是一个右值引用。与拷贝构造函数一样，任何额外的参数都必须有默认实参。

除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态——销毁它是无害的。特别是，一旦资源完成移动，源对象必须不再指向被移动的资源——这些资源的所有权已经归属新创建的对象。

作为一个例子，我们为 `StrVec` 类定义移动构造函数，实现从一个 `StrVec` 到另一个 `StrVec` 的元素移动而非拷贝：

```
StrVec::StrVec(StrVec &&s) noexcept // 移动操作不应抛出任何异常
    // 成员初始化器接管 s 中的资源
    : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 令 s 进入这样的状态——对其运行析构函数是安全的
    s.elements = s.first_free = s.cap = nullptr;
}
```

我们将简短解释 `noexcept`（它通知标准库我们的构造函数不抛出任何异常），但让我们先分析一下此构造函数完成什么工作。

与拷贝构造函数不同，移动构造函数不分配任何新内存；它接管给定的 `StrVec` 中的内存。在接管内存之后，它将给定对象中的指针都置为 `nullptr`。这样就完成了从给定对象的移动操作，此对象将继续存在。最终，移后源对象会被销毁，意味着将在其上运行析构函数。`StrVec` 的析构函数在 `first_free` 上调用 `deallocate`。如果我们忘记了改变 `s.first_free`，则销毁移后源对象就会释放掉我们刚刚移动的内存。

移动操作、标准库容器和异常



由于移动操作“窃取”资源，它通常不分配任何资源。因此，移动操作通常不会抛出任何异常。当编写一个不抛出异常的移动操作时，我们应该将此事通知标准库。我们将看到，除非标准库知道我们的移动构造函数不会抛出异常，否则它会认为移动我们的类对象时可能会抛出异常，并且为了处理这种可能性而做一些额外的工作。

一种通知标准库的方法是在我们的构造函数中指明 `noexcept`。`noexcept` 是新标准引入的，我们将在 18.1.4 节（第 690 页）中讨论更多细节。目前重要的是要知道，`noexcept` 是我们承诺一个函数不抛出异常的一种方法。我们在一个函数的参数列表后指定 `noexcept`。在一个构造函数中，`noexcept` 出现在参数列表和初始化列表开始的冒号之间：

```
class StrVec {
```

C++
11

535

```

public:
    StrVec(StrVec&&) noexcept; // 移动构造函数
    // 其他成员的定义，如前
};

StrVec::StrVec(StrVec &&s) noexcept : /* 成员初始化器 */
{ /* 构造函数体 */ }

```

我们必须在类头文件的声明中和定义中（如果定义在类外的话）都指定 `noexcept`。



不抛出异常的移动构造函数和移动赋值运算符必须标记为 `noexcept`。

536

搞清楚为什么需要 `noexcept` 能帮助我们深入理解标准库是如何与我们自定义的类型交互的。我们需要指出一个移动操作不抛出异常，这是因为两个相互关联的事实：首先，虽然移动操作通常不抛出异常，但抛出异常也是允许的；其次，标准库容器能对异常发生时其自身的行为提供保障。例如，`vector` 保证，如果我们调用 `push_back` 时发生异常，`vector` 自身不会发生改变。

现在让我们思考 `push_back` 内部发生了什么。类似对应的 `StrVec` 操作（参见 13.5 节，第 466 页），对一个 `vector` 调用 `push_back` 可能要求为 `vector` 重新分配内存空间。当重新分配 `vector` 的内存时，`vector` 将元素从旧空间移动到新内存中，就像我们在 `reallocate` 中所做的那样（参见 13.5 节，第 469 页）。

如我们刚刚看到的那样，移动一个对象通常会改变它的值。如果重新分配过程使用了移动构造函数，且在移动了部分而不是全部元素后抛出了一个异常，就会产生问题。旧空间中的移动源元素已经被改变了，而新空间中未构造的元素可能尚不存在。在此情况下，`vector` 将不能满足自身保持不变的要求。

另一方面，如果 `vector` 使用了拷贝构造函数且发生了异常，它可以很容易地满足要求。在此情况下，当在新内存中构造元素时，旧元素保持不变。如果此时发生了异常，`vector` 可以释放新分配的（但还未成功构造的）内存并返回。`vector` 原有的元素仍然存在。

为了避免这种潜在问题，除非 `vector` 知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，它就必须使用拷贝构造函数而不是移动构造函数。如果希望在 `vector` 重新分配内存这类情况下对我们自定义类型的对象进行移动而不是拷贝，就必须显式地告诉标准库我们的移动构造函数可以安全使用。我们通过将移动构造函数（及移动赋值运算符）标记为 `noexcept` 来做到这一点。

移动赋值运算符

移动赋值运算符执行与析构函数和移动构造函数相同的工作。与移动构造函数一样，如果我们的移动赋值运算符不抛出任何异常，我们就应该将它标记为 `noexcept`。类似拷贝赋值运算符，移动赋值运算符必须正确处理自赋值：

```

StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // 直接检测自赋值
    if (this != &rhs) {
        free(); // 释放已有元素
        elements = rhs.elements; // 从 rhs 接管资源
        first_free = rhs.first_free;
    }
}

```

```

    cap = rhs.cap;
    // 将 rhs 置于可析构状态
    rhs.elements = rhs.first_free = rhs.cap = nullptr;
}
return *this;
}

```

在此例中，我们直接检查 `this` 指针与 `rhs` 的地址是否相同。如果相同，右侧和左侧运算对象指向相同的对象，我们不需要做任何事情。否则，我们释放左侧运算对象所使用的内存，并接管给定对象的内存。与移动构造函数一样，我们将 `rhs` 中的指针置为 `nullptr`。537

我们费心地去检查自赋值情况看起来有些奇怪。毕竟，移动赋值运算符需要右侧运算对象的一个右值。我们进行检查的原因是此右值可能是 `move` 调用的返回结果。与其他任何赋值运算符一样，关键点是我们不能在使用右侧运算对象的资源之前就释放左侧运算对象的资源（可能是相同的资源）。

移后源对象必须可析构



从一个对象移动数据并不会销毁此对象，但有时在移动操作完成后，源对象会被销毁。因此，当我们编写一个移动操作时，必须确保移后源对象进入一个可析构的状态。我们的 `StrVec` 的移动操作满足这一要求，这是通过将移后源对象的指针成员置为 `nullptr` 来实现的。

除了将移后源对象置为析构安全的状态之外，移动操作还必须保证对象仍然是有效的。一般来说，对象有效就是指可以安全地为其赋予新值或者可以安全地使用而不依赖其当前值。另一方面，移动操作对移后源对象中留下的值没有任何要求。因此，我们的程序不应该依赖于移后源对象中的数据。

例如，当我们从一个标准库 `string` 或容器对象移动数据时，我们知道移后源对象仍然保持有效。因此，我们可以对它执行诸如 `empty` 或 `size` 这些操作。但是，我们不知道将会得到什么结果。我们可能期望一个移后源对象是空的，但这并没有保证。

我们的 `StrVec` 类的移动操作将移后源对象置于与默认初始化的对象相同的状态。因此，我们可以继续对移后源对象执行所有的 `StrVec` 操作，与任何其他默认初始化的对象一样。而其他内部结构更为复杂的类，可能表现出完全不同的行为。



WARNING 在移动操作之后，移后源对象必须保持有效的、可析构的状态，但是用户不能对其值进行任何假设。

合成的移动操作

与处理拷贝构造函数和拷贝赋值运算符一样，编译器也会合成移动构造函数和移动赋值运算符。但是，合成移动操作的条件与合成拷贝操作的条件大不相同。

回忆一下，如果我们不声明自己的拷贝构造函数或拷贝赋值运算符，编译器总会为我们合成这些操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。拷贝操作要么被定义为逐成员拷贝，要么被定义为对象赋值，要么被定义为删除的函数。

与拷贝操作不同，编译器根本不会为某些类合成移动操作。特别是，如果一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。因此，某些类就没有移动构造函数或移动赋值运算符。如我们将在第 477 页所见，如果一个类没有移动操作，通过正常的函数匹配，类会使用对应的拷

538

贝操作来代替移动操作。

只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非 static 数据成员都可以移动时，编译器才会为它合成移动构造函数或移动赋值运算符。编译器可以移动内置类型的成员。如果一个成员是类类型，且该类有对应的移动操作，编译器也能移动这个成员：

```
// 编译器会为 X 和 hasX 合成移动操作
struct X {
    int i;           // 内置类型可以移动
    std::string s;  // string 定义了自己的移动操作
};

struct hasX {
    X mem; // X 有合成的移动操作
};

X x, x2 = std::move(x);           // 使用合成的移动构造函数
hasX hx, hx2 = std::move(hx);    // 使用合成的移动构造函数
```



只有当一个类没有定义任何自己版本的拷贝控制成员，且它的所有数据成员都能移动构造或移动赋值时，编译器才会为它合成移动构造函数或移动赋值运算符。

与拷贝操作不同，移动操作永远不会隐式定义为删除的函数。但是，如果我们显式地要求编译器生成`=default` 的（参见 7.1.4 节，第 237 页）移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数。除了一个重要例外，什么时候将合成的移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类似的原则（参见 13.1.6 节，第 449 页）：

- 与拷贝构造函数不同，移动构造函数被定义为删除的函数的条件是：有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者是有类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数。移动赋值运算符的情况类似。
- 如果有类成员的移动构造函数或移动赋值运算符被定义为删除的或是不可访问的，则类的移动构造函数或移动赋值运算符被定义为删除的。
- 类似拷贝构造函数，如果类的析构函数被定义为删除的或不可访问的，则类的移动构造函数被定义为删除的。
- 类似拷贝赋值运算符，如果有类成员是 `const` 的或是引用，则类的移动赋值运算符被定义为删除的。

539 例如，假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数：

```
// 假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数
struct hasY {
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; // hasY 将有一个删除的移动构造函数
};
hasY hy, hy2 = std::move(hy); // 错误：移动构造函数是删除的
```

编译器可以拷贝类型为 Y 的对象，但不能移动它们。类 `hasY` 显式地要求一个移动构造函数，但编译器无法为其生成。因此，`hasY` 会有一个删除的移动构造函数。如果 `hasY` 忽略了移动构造函数的声明，则编译器根本不能为它合成一个。如果移动操作可能被定义为

删除的函数，编译器就不会合成它们。

移动操作和合成的拷贝控制成员间还有最后一个相互作用关系：一个类是否定义了自己的移动操作对拷贝操作如何合成有影响。如果类定义了一个移动构造函数和/或一个移动赋值运算符，则该类的合成拷贝构造函数和拷贝赋值运算符会被定义为删除的。



定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作。
否则，这些成员默认地被定义为删除的。

移动右值，拷贝左值……

如果一个类既有移动构造函数，也有拷贝构造函数，编译器使用普通的函数匹配规则来确定使用哪个构造函数（参见 6.4 节，第 208 页）。赋值操作的情况类似。例如，在我们的 StrVec 类中，拷贝构造函数接受一个 `const StrVec` 的引用。因此，它可以用子任何可以转换为 `StrVec` 的类型。而移动构造函数接受一个 `StrVec&&`，因此只能用于实参是（非 `static`）右值的情形：

```
StrVec v1, v2;
v1 = v2;                                // v2 是左值；使用拷贝赋值
StrVec getVec(istream &);      // getVec 返回一个右值
v2 = getVec(cin);           // getVec(cin) 是一个右值；使用移动赋值
```

在第一个赋值中，我们将 `v2` 传递给赋值运算符。`v2` 的类型是 `StrVec`，表达式 `v2` 是一个左值。因此移动版本的赋值运算符是不可行的（参见 6.6 节，第 217 页），因为我们不能隐式地将一个右值引用绑定到一个左值。因此，这个赋值语句使用拷贝赋值运算符。

在第二个赋值中，我们赋予 `v2` 的是 `getVec` 调用的结果。此表达式是一个右值。在此情况下，两个赋值运算符都是可行的——将 `getVec` 的结果绑定到两个运算符的参数都是允许的。调用拷贝赋值运算符需要进行一次到 `const` 的转换，而 `StrVec&&` 则是精确匹配。因此，第二个赋值会使用移动赋值运算符。

……但如果没有移动构造函数，右值也被拷贝

540

如果一个类有一个拷贝构造函数但未定义移动构造函数，会发生什么呢？在此情况下，编译器不会合成移动构造函数，这意味着此类将有拷贝构造函数但不会有移动构造函数。如果一个类没有移动构造函数，函数匹配规则保证该类型的对象会被拷贝，即使我们试图通过调用 `move` 来移动它们时也是如此：

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&); // 拷贝构造函数
    // 其他成员定义，但 Foo 未定义移动构造函数
};

Foo x;
Foo y(x);           // 拷贝构造函数；x 是一个左值
Foo z(std::move(x)); // 拷贝构造函数，因为未定义移动构造函数
```

在对 `z` 进行初始化时，我们调用了 `move(x)`，它返回一个绑定到 `x` 的 `Foo&&`。`Foo` 的拷贝构造函数是可行的，因为我们可以将一个 `Foo&&` 转换为一个 `const Foo&`。因此，`z` 的初始化将使用 `Foo` 的拷贝构造函数。

值得注意的是，用拷贝构造函数代替移动构造函数几乎肯定是安全的（赋值运算符的

情况类似)。一般情况下，拷贝构造函数满足对应的移动构造函数的要求：它会拷贝给定对象，并将原对象置于有效状态。实际上，拷贝构造函数甚至都不会改变原对象的值。



如果一个类有一个可用的拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来“移动”的。拷贝赋值运算符和移动赋值运算符的情况类似。

拷贝并交换赋值运算符和移动操作

我们的 `HasPtr` 版本定义了一个拷贝并交换赋值运算符（参见 13.3 节，第 459 页），它是函数匹配和移动操作间相互关系的一个很好的示例。如果我们为此类添加一个移动构造函数，它实际上也会获得一个移动赋值运算符：

```
class HasPtr {
public:
    // 添加的移动构造函数
    HasPtr(HasPtr &p) noexcept : ps(p.ps), i(p.i) { p.ps = 0; }
    // 赋值运算符既是移动赋值运算符，也是拷贝赋值运算符
    HasPtr& operator=(HasPtr rhs)
        { swap(*this, rhs); return *this; }
    // 其他成员的定义，同 13.2.1 节（第 453 页）
};
```

541 在这个版本中，我们为类添加了一个移动构造函数，它接管了给定实参的值。构造函数体将给定的 `HasPtr` 的指针置为 0，从而确保销毁移后源对象是安全的。此函数不会抛出异常，因此我们将其标记为 `noexcept`（参见 13.6.2 节，第 473 页）。

现在让我们观察赋值运算符。此运算符有一个非引用参数，这意味着此参数要进行拷贝初始化（参见 13.1.1 节，第 441 页）。依赖于实参的类型，拷贝初始化要么使用拷贝构造函数，要么使用移动构造函数——左值被拷贝，右值被移动。因此，单一的赋值运算符就实现了拷贝赋值运算符和移动赋值运算符两种功能。

例如，假定 `hp` 和 `hp2` 都是 `HasPtr` 对象：

```
hp = hp2; // hp2 是一个左值；hp2 通过拷贝构造函数来拷贝
hp = std::move(hp2); // 移动构造函数移动 hp2
```

在第一个赋值中，右侧运算对象是一个左值，因此移动构造函数是不可行的。`rhs` 将使用拷贝构造函数来初始化。拷贝构造函数将分配一个新 `string`，并拷贝 `hp2` 指向的 `string`。

在第二个赋值中，我们调用 `std::move` 将一个右值引用绑定到 `hp2` 上。在此情况下，拷贝构造函数和移动构造函数都是可行的。但是，由于实参是一个右值引用，移动构造函数是精确匹配的。移动构造函数从 `hp2` 拷贝指针，而不会分配任何内存。

不管使用的是拷贝构造函数还是移动构造函数，赋值运算符的函数体都 `swap` 两个运算对象的状态。交换 `HasPtr` 会交换两个对象的指针（及 `int`）成员。在 `swap` 之后，`rhs` 中的指针将指向原来左侧运算对象所拥有的 `string`。当 `rhs` 离开其作用域时，这个 `string` 将被销毁。

建议：更新三/五法则

所有五个拷贝控制成员应该看作一个整体：一般来说，如果一个类定义了任何一个

拷贝操作，它就应该定义所有五个操作。如前所述，某些类必须定义拷贝构造函数、拷贝赋值运算符和析构函数才能正确工作（参见 13.1.4 节，第 447 页）。这些类通常拥有一个资源，而拷贝成员必须拷贝此资源。一般来说，拷贝一个资源会导致一些额外开销。在这种拷贝并非必要的情况下，定义了移动构造函数和移动赋值运算符的类就可以避免此问题。

Message 类的移动操作

定义了自己的拷贝构造函数和拷贝赋值运算符的类通常也会从移动操作受益。例如，我们的 Message 和 Folder 类（参见 13.4 节，第 460 页）就应该定义移动操作。通过定义移动操作，Message 类可以使用 string 和 set 的移动操作来避免拷贝 contents 和 folders 成员的额外开销。

但是，除了移动 folders 成员，我们还必须更新每个指向原 Message 的 Folder。我们必须删除指向旧 Message 的指针，并添加一个指向新 Message 的指针。

移动构造函数和移动赋值运算符都需要更新 Folder 指针，因此我们首先定义一个操作来完成这一共同的工作：

```
// 从本 Message 移动 Folder 指针
void Message::move_Folders(Message *m)
{
    folders = std::move(m->folders); // 使用 set 的移动赋值运算符
    for (auto f : folders) { // 对每个 Folder
        f->remMsg(m); // 从 Folder 中删除旧 Message
        f->addMsg(this); // 将本 Message 添加到 Folder 中
    }
    m->folders.clear(); // 确保销毁 m 是无害的
}
```

此函数首先移动 folders 集合。通过调用 move，我们使用了 set 的移动赋值运算符而不是它的拷贝赋值运算符。如果我们忽略了 move 调用，代码仍能正常工作，但带来了不必要的拷贝。函数然后遍历所有 Folder，从其中删除指向原 Message 的指针并添加指向新 Message 的指针。

值得注意的是，向 set 插入一个元素可能会抛出一个异常——向容器添加元素的操作要求分配内存，意味着可能会抛出一个 bad_alloc 异常（参见 12.1.2 节，第 409 页）。因此，与我们的 HasPtr 和 StrVec 类的移动操作不同，Message 的移动构造函数和移动赋值运算符可能会抛出异常。因此我们未将它们标记为 noexcept（参见 13.6.2 节，第 473 页）。

函数最后对 m.folders 调用 clear。在执行了 move 之后，我们知道 m.folders 是有效的，但不知道它包含什么内容。由于 Message 的析构函数遍历 folders，我们希望能确定 set 是空的。

Message 的移动构造函数调用 move 来移动 contents，并默认初始化自己的 folders 成员：

```
Message::Message(Message &m) : contents(std::move(m.contents))
{
    move_Folders(&m); // 移动 folders 并更新 Folder 指针
}
```

在构造函数体中，我们调用了 `move_Folders` 来删除指向 `m` 的指针并插入指向本 `Message` 的指针。

移动赋值运算符直接检查自赋值情况：

```
Message& Message::operator=(Message &&rhs)
{
    if (this != &rhs) {           // 直接检查自赋值情况
        remove_from_Folders();
        contents = std::move(rhs.contents); // 移动赋值运算符
        move_Folders(&rhs);   // 重置 Folders 指向本 Message
    }
    return *this;
}
```

543 与任何赋值运算符一样，移动赋值运算符必须销毁左侧运算对象的旧状态。在本例中，销毁左侧运算对象要求我们从现有 `folders` 中删除指向本 `Message` 的指针，我们调用 `remove_from_Folders` 来完成这一工作。完成删除工作后，我们调用 `move` 从 `rhs` 将 `contents` 移动到 `this` 对象。剩下的就是调用 `move_Messages` 来更新 `Folder` 指针了。

移动迭代器

`StrVec` 的 `reallocate` 成员（参见 13.5 节，第 469 页）使用了一个 `for` 循环来调用 `construct` 从旧内存将元素拷贝到新内存中。作为一种替换方法，如果我们能调用 `uninitialized_copy` 来构造新分配的内存，将比循环更为简单。但是，`uninitialized_copy` 恰如其名：它对元素进行拷贝操作。标准库中并没有类似的函数将对象“移动”到未构造的内存中。

C++ 11 新标准库中定义了一种**移动迭代器**（move iterator）适配器（参见 10.4 节，第 358 页）。一个移动迭代器通过改变给定迭代器的解引用运算符的行为来适配此迭代器。一般来说，一个迭代器的解引用运算符返回一个指向元素的左值。与其他迭代器不同，移动迭代器的解引用运算符生成一个右值引用。

我们通过调用标准库的 `make_move_iterator` 函数将一个普通迭代器转换为一个移动迭代器。此函数接受一个迭代器参数，返回一个移动迭代器。

原迭代器的所有其他操作在移动迭代器中都照常工作。由于移动迭代器支持正常的迭代器操作，我们可以将一对移动迭代器传递给算法。特别是，可以将移动迭代器传递给 `uninitialized_copy`：

```
void StrVec::reallocate()
{
    // 分配大小两倍于当前规模的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    auto first = alloc.allocate(newcapacity);
    // 移动元素
    auto last = uninitialized_copy(make_move_iterator(begin()),
                                   make_move_iterator(end()),
                                   first);
    free();           // 释放旧空间
    elements = first; // 更新指针
    first_free = last;
```

```
    cap = elements + newcapacity;
}
```

uninitialized_copy 对输入序列中的每个元素调用 construct 来将元素“拷贝”到目的位置。此算法使用迭代器的解引用运算符从输入序列中提取元素。由于我们传递给它的是移动迭代器，因此解引用运算符生成的是一个右值引用，这意味着 construct 将使用移动构造函数来构造元素。

值得注意的是，标准库不保证哪些算法适用移动迭代器，哪些不适用。由于移动一个对象可能销毁掉原对象，因此你只有在确信算法在为一个元素赋值或将其传递给一个用户定义的函数后不再访问它时，才能将移动迭代器传递给算法。544

建议：不要随意使用移动操作

由于一个移后源对象具有不确定的状态，对其调用 std::move 是危险的。当我们调用 move 时，必须绝对确认移后源对象没有其他用户。

通过在类代码中小心地使用 move，可以大幅度提升性能。而如果随意在普通用户代码（与类实现代码相对）中使用移动操作，很可能导致莫名其妙的、难以查找的错误，而难以提升应用程序性能。

Best Practices 在移动构造函数和移动赋值运算符这些类实现代码之外的地方，只有当你确信需要进行移动操作且移动操作是安全的，才可以使用 std::move。

13.6.2 节练习

练习 13.49：为你的 StrVec、String 和 Message 类添加一个移动构造函数和一个移动赋值运算符。

练习 13.50：在你的 String 类的移动操作中添加打印语句，并重新运行 13.6.1 节（第 473 页）的练习 13.48 中的程序，它使用了一个 vector<String>，观察什么时候会避免拷贝。

练习 13.51：虽然 unique_ptr 不能拷贝，但我们在 12.1.5 节（第 418 页）中编写了一个 clone 函数，它以值方式返回一个 unique_ptr。解释为什么函数是合法的，以及为什么它能正确工作。

练习 13.52：详细解释第 478 页中的 HasPtr 对象的赋值发生了什么？特别是，一步一步描述 hp、hp2 以及 HasPtr 的赋值运算符中的参数 rhs 的值发生了什么变化。

练习 13.53：从底层效率的角度看，HasPtr 的赋值运算符并不理想，解释为什么。为 HasPtr 实现一个拷贝赋值运算符和一个移动赋值运算符，并比较你的新的移动赋值运算符中执行的操作和拷贝并交换版本中执行的操作。

练习 13.54：如果我们为 HasPtr 定义了移动赋值运算符，但未改变拷贝并交换运算符，会发生什么？编写代码验证你的答案。

13.6.3 右值引用和成员函数

除了构造函数和赋值运算符之外，如果一个成员函数同时提供拷贝和移动版本，它也能从中受益。这种允许移动的成员函数通常使用与拷贝/移动构造函数和赋值运算符相同的参数模式——一个版本接受一个指向 const 的左值引用，第二个版本接受一个指向非

`const` 的右值引用。

例如，定义了 `push_back` 的标准库容器提供两个版本：一个版本有一个右值引用参数，而另一个版本有一个 `const` 左值引用。假定 `X` 是元素类型，那么这些容器就会定义以下两个 `push_back` 版本：

```
void push_back(const X&);      // 拷贝：绑定到任意类型的 X
void push_back(X&&);         // 移动：只能绑定到类型 X 的可修改的右值
```

我们可以将能转换为类型 `X` 的任何对象传递给第一个版本的 `push_back`。此版本从其参数拷贝数据。对于第二个版本，我们只可以传递给它非 `const` 的右值。此版本对于非 `const` 的右值是精确匹配（也是更好的匹配）的，因此当我们传递一个可修改的右值（参见 13.6.2 节，第 477 页）时，编译器会选择运行这个版本。此版本会从其参数窃取数据。

一般来说，我们不需要为函数操作定义接受一个 `const X&&` 或是一个（普通的）`X&` 参数的版本。当我们希望从实参“窃取”数据时，通常传递一个右值引用。为了达到这一目的，实参不能是 `const` 的。类似的，从一个对象进行拷贝的操作不应该改变该对象。因此，通常不需要定义一个接受一个（普通的）`X&` 参数的版本。



区分移动和拷贝的重载函数通常有一个版本接受一个 `const T&`，而另一个版本接受一个 `T&&`。

作为一个更具体的例子，我们将为 `StrVec` 类定义另一个版本的 `push_back`：

```
class StrVec {
public:
    void push_back(const std::string&); // 拷贝元素
    void push_back(std::string&&);     // 移动元素
    // 其他成员的定义，如前
};

// 与 13.5 节（第 466 页）中的原版本相同
void StrVec::push_back(const string &s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的一个副本
    alloc.construct(first_free++, s);
}

void StrVec::push_back(string &&s)
{
    chk_n_alloc(); // 如果需要的话为 StrVec 重新分配内存
    alloc.construct(first_free++, std::move(s));
}
```

这两个成员几乎是相同的。差别在于右值引用版本调用 `move` 来将其参数传递给 `construct`。如前所述，`construct` 函数使用其第二个和随后的实参的类型来确定使用哪个构造函数。由于 `move` 返回一个右值引用，传递给 `construct` 的实参类型是 `string&&`。因此，会使用 `string` 的移动构造函数来构造新元素。

当我们调用 `push_back` 时，实参类型决定了新元素是拷贝还是移动到容器中：

```
StrVec vec; // 空 StrVec
string s = "some string or another";
vec.push_back(s); // 调用 push_back(const string&)
```

```
vec.push_back("done"); // 调用 push_back(string&&)
```

这些调用的差别在于实参是一个左值还是一个右值（从"done"创建的临时 string），具体调用哪个版本据此来决定。

右值和左值引用成员函数

通常，我们在一个对象上调用成员函数，而不管该对象是一个左值还是一个右值。例如：

```
string s1 = "a value", s2 = "another";
auto n = (s1 + s2).find('a');
```

此例中，我们在一个 string 右值上调用 find 成员（参见 9.5.3 节，第 325 页），该 string 右值是通过连接两个 string 而得到的。有时，右值的使用方式可能令人惊讶：

```
s1 + s2 = "wow!";
```

此处我们对两个 string 的连接结果——一个右值，进行了赋值。

在旧标准中，我们没有办法阻止这种使用方式。为了维持向后兼容性，新标准库类仍然允许向右值赋值。但是，我们可能希望在自己的类中阻止这种用法。在此情况下，我们希望强制左侧运算对象（即，this 指向的对象）是一个左值。

我们指出 this 的左值/右值属性的方式与定义 const 成员函数相同（参见 7.1.2 节，C++ 11 第 231 页），即，在参数列表后放置一个引用限定符（reference qualifier）：

```
class Foo {
public:
    Foo &operator=(const Foo&) &; // 只能向可修改的左值赋值
    // Foo 的其他参数
};

Foo &Foo::operator=(const Foo &rhs) &
{
    // 执行将 rhs 赋予本对象所需的工作
    return *this;
}
```

引用限定符可以是&或&&，分别指出 this 可以指向一个左值或右值。类似 const 限定符，引用限定符只能用于（非 static）成员函数，且必须同时出现在函数的声明和定义中。

对于&限定的函数，我们只能将它用于左值；对于&&限定的函数，只能用于右值：

```
Foo &retFoo(); // 返回一个引用；retFoo 调用是一个左值
Foo retVal(); // 返回一个值；retVal 调用是一个右值
Foo i, j; // i 和 j 是左值
i = j; // 正确：i 是左值
retFoo() = j; // 正确：retFoo() 返回一个左值
retVal() = j; // 错误：retVal() 返回一个右值
i = retVal(); // 正确：我们可以将一个右值作为赋值操作的右侧运算对象
```

一个函数可以同时用 const 和引用限定。在此情况下，引用限定符必须跟随在 const 限定符之后：

```
class Foo {
public:
    Foo someMem() & const; // 错误：const 限定符必须在前
    Foo anotherMem() const &; // 正确：const 限定符在前
};
```

重载和引用函数

就像一个成员函数可以根据是否有 `const` 来区分其重载版本一样（参见 7.3.2 节，第 247 页），引用限定符也可以区分重载版本。而且，我们可以综合引用限定符和 `const` 来区分一个成员函数的重载版本。例如，我们将为 `Foo` 定义一个名为 `data` 的 `vector` 成员和一个名为 `sorted` 的成员函数，`sorted` 返回一个 `Foo` 对象的副本，其中 `vector` 已被排序：

```
class Foo {
public:
    Foo sorted() &&;           // 可用于可改变的右值
    Foo sorted() const &;      // 可用于任何类型的 Foo
    // Foo 的其他成员的定义
private:
    vector<int> data;
};

// 本对象为右值，因此可以原址排序
Foo Foo::sorted() &&
{
    sort(data.begin(), data.end());
    return *this;
}
// 本对象是 const 或是一个左值，哪种情况我们都不能对其进行原址排序
Foo Foo::sorted() const & {
    Foo ret(*this);           // 拷贝一个副本
    sort(ret.data.begin(), ret.data.end()); // 排序副本
    return ret;               // 返回副本
}
```

当我们对一个右值执行 `sorted` 时，它可以安全地直接对 `data` 成员进行排序。对象是一个右值，意味着没有其他用户，因此我们可以改变对象。当对一个 `const` 右值或一个左值执行 `sorted` 时，我们不能改变对象，因此就需要在排序前拷贝 `data`。

编译器会根据调用 `sorted` 的对象的左值/右值属性来确定使用哪个 `sorted` 版本：

548 `retVal().sorted();` // `retVal()` 是一个右值，调用 `Foo::sorted() &&`
`retFoo().sorted();` // `retFoo()` 是一个左值，调用 `Foo::sorted() const &`

当我们定义 `const` 成员函数时，可以定义两个版本，唯一的差别是一个版本有 `const` 限定而另一个没有。引用限定的函数则不一样。如果我们定义两个或两个以上具有相同名字和相同参数列表的成员函数，就必须对所有函数都加上引用限定符，或者所有都不加：

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // 错误：必须加上引用限定符
    // Comp 是函数类型的类型别名（参见 6.7 节，第 222 页）
    // 此函数类型可以用来比较 int 值
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);           // 正确：不同的参数列表
    Foo sorted(Comp*) const;     // 正确：两个版本都没有引用限定符
};
```

本例中声明了一个没有参数的 `const` 版本的 `sorted`, 此声明是错误的。因为 `Foo` 类中还有一个无参的 `sorted` 版本, 它有一个引用限定符, 因此 `const` 版本也必须有引用限定符。另一方面, 接受一个比较操作指针的 `sorted` 版本是没问题的, 因为两个函数都没有引用限定符。



如果一个成员函数有引用限定符, 则具有相同参数列表的所有版本都必须有引用限定符。

13.6.3 节练习

练习 13.55: 为你的 `StrBlob` 添加一个右值引用版本的 `push_back`。

练习 13.56: 如果 `sorted` 定义如下, 会发生什么:

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

练习 13.57: 如果 `sorted` 定义如下, 会发生什么:

```
Foo Foo::sorted() const & { return Foo(*this).sorted(); }
```

练习 13.58: 编写新版本的 `Foo` 类, 其 `sorted` 函数中有打印语句, 测试这个类, 来验证你对前两题的答案是否正确。

549

小结

每个类都会控制该类型对象拷贝、移动、赋值以及销毁时发生什么。特殊的成员函数——拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符和析构函数定义了这些操作。移动构造函数和移动赋值运算符接受一个（通常是非 `const` 的）右值引用；而拷贝版本则接受一个（通常是 `const` 的）普通左值引用。

如果一个类未声明这些操作，编译器会自动为其生成。如果这些操作未定义成删除的，它们会逐成员初始化、移动、赋值或销毁对象：合成的操作依次处理每个非 `static` 数据成员，根据成员类型确定如何移动、拷贝、赋值或销毁它。

分配了内存或其他资源的类几乎总是需要定义拷贝控制成员来管理分配的资源。如果一个类需要析构函数，则它几乎肯定也需要定义移动和拷贝构造函数及移动和拷贝赋值运算符。

术语表

拷贝并交换（copy and swap） 涉及赋值运算符的技术，首先拷贝右侧运算对象，然后调用 `swap` 来交换副本和左侧运算对象。

拷贝赋值运算符（copy-assignment operator） 接受一个本类型对象的赋值运算符版本。通常，拷贝赋值运算符的参数是一个 `const` 的引用，并返回指向本对象的引用。如果类未显式定义拷贝赋值运算符，编译器会为它合成一个。

拷贝构造函数（copy constructor） 一种构造函数，将新对象初始化为同类型另一个对象的副本。当向函数传递对象，或以传值方式从函数返回对象时，会隐式使用拷贝构造函数。如果我们未提供拷贝构造函数，编译器会为我们合成一个。

拷贝控制（copy control） 特殊的成员函数，控制拷贝、移动、赋值及销毁本类类型对象时发生什么。如果类未定义这些操作，编译器会为它合成恰当的定义。

拷贝初始化（copy initialization） 一种初始化形式，当我们使用`=`为一个新创建的对象提供初始化器时，会使用拷贝初始化。如果我们向函数传递对象或以传值方式从函数返回对象，以及初始化一个数组或一个聚合类时，也会使用拷贝初始化。

删除的函数（deleted function） 不能使用的函数。我们在一个函数的声明上指定`=delete` 来删除它。删除的函数的一个常见用途是告诉编译器不要为类合成拷贝和/或移动操作。

析构函数（destructor） 特殊的成员函数，当对象离开作用域或被释放时进行清理工作。编译器会自动销毁每个数据成员。类类型的成员通过调用其析构函数来销毁；而内置类型或复合类型的成员的销毁则不需要做任何工作。特别是，析构函数不会释放指针成员指向的对象。

左值引用（lvalue reference） 可以绑定到左值的引用。

逐成员拷贝 / 赋值（memberwise copy/assign） 合成的拷贝与移动构造函数及拷贝与移动赋值运算符的工作方式。合成的拷贝或移动构造函数依次处理每个非 `static` 数据成员，通过从给定对象拷贝或移动对应成员来初始化本对象成员；拷贝或移动赋值运算符从右侧运算对象中将每个成员拷贝赋值或移动赋值到左侧运算对象中。内置类型或复合类型的成员直接进行初始化或赋值。类类型的成员通过成员对应的拷贝/移动构造函数或拷贝/移动赋值运算符进行初始化或赋值。

move 用来将一个右值引用绑定到一个左值的标准库函数。调用 `move` 隐含地承诺我们将不会再使用移后源对象，除了销毁它或赋予它一个新值之外。

移动赋值运算符（move-assignment operator） 接受一个本类型右值引用参数的赋值运算符版本。通常，移动赋值运算符将数据从右侧运算对象移动到左侧运算对象。赋值之后，对右侧运算对象执行析构函数必须是安全的。

移动构造函数（move constructor） 一种构造函数，接受一个本类型的右值引用。通常，移动构造函数将数据从其参数移动到新创建的对象中。移动之后，对给定的实参执行析构函数必须是安全的。

移动迭代器（move iterator） 迭代器适配器，它生成的迭代器在解引用时会得到一个右值引用。

重载运算符（overloaded operator） 一种函数，重定义了运算符应用于类类型的对象时的含义。本章介绍了如何定义赋值运算符；第 14 章中将介绍重载运算符的更多细节内容。

引用计数（reference count） 一种程序设计技术，通常用于拷贝控制成员的设计。引用计数记录了有多少对象共享状态。构造函数（不是拷贝/移动构造函数）将引用计数置为 1。每当创建一个新副本时，计数

值递增。当一个对象被销毁时，计数值递减。赋值运算符和析构函数检查递减的引用计数是否为 0，如果是，它们会销毁对象。

引用限定符（reference qualifier） 用来指出一个非 `static` 成员函数可以用于左值或右值的符号。限定符`&`和`&&`应该放在参数列表之后或 `const` 限定符之后（如果有的话）。被`&`限定的函数只能用于左值；被`&&`限定的函数只能用于右值。

右值引用（rvalue reference） 指向一个将要销毁的对象的引用。

合成赋值运算符（synthesized assignment operator） 编译器为未显式定义赋值运算符的类创建的（合成的）拷贝或移动赋值运算符版本。除非定义为删除的，合成赋值运算符会逐成员地将右侧运算对象赋予（移动到）左侧运算对象。

合成拷贝/移动构造函数（synthesized copy/move constructor） 编译器为未显式定义对应的构造函数的类生成的拷贝或移动构造函数版本。除非定义为删除的，合成拷贝或移动构造函数分别通过从给定对象拷贝或移动成员来逐成员地初始化新对象。

合成析构函数（synthesized destructor） 编译器为未显式定义析构函数的类创建的（合成的）版本。合成析构函数的函数体为空。

第 14 章

重载运算与类型转换

内容

14.1 基本概念	490
14.2 输入和输出运算符	494
14.3 算术和关系运算符	497
14.4 赋值运算符	499
14.5 下标运算符	501
14.6 递增和递减运算符	502
14.7 成员访问运算符	504
14.8 函数调用运算符	506
14.9 重载、类型转换与运算符	514
小结	523
术语表	523

在第 4 章中我们看到，C++语言定义了大量运算符以及内置类型的自动转换规则。这些特性使得程序员能编写出形式丰富、含有多种混合类型的表达式。

当运算符被用于类类型的对象时，C++语言允许我们为其指定新的含义；同时，我们也能自定义类类型之间的转换规则。和内置类型的转换一样，类类型转换隐式地将一种类型的对象转换成另一种我们所需类型的对象。

552 当运算符作用于类类型的运算对象时，可以通过运算符重载重新定义该运算符的含义。明智地使用运算符重载能令我们的程序更易于编写和阅读。举个例子，因为在 Sales_item 类（参见 1.5.1 节，第 17 页）中定义了输入、输出和加法运算符，所以可以通过下述形式输出两个 Sales_item 的和：

```
cout << item1 + item2; // 输出两个 Sales_item 的和
```

相反的，由于我们的 Sales_data 类（参见 7.1 节，第 228 页）还没有重载这些运算符，因此它的加法代码显得比较冗长而不清晰：

```
print(cout, add(data1, data2)); // 输出两个 Sales_data 的和
```



14.1 基本概念

重载的运算符是具有特殊名字的函数：它们的名字由关键字 operator 和其后要定义的运算符号共同组成。和其他函数一样，重载的运算符也包含返回类型、参数列表以及函数体。

重载运算符函数的参数数量与该运算符作用的运算对象数量一样多。一元运算符有一个参数，二元运算符有两个。对于二元运算符来说，左侧运算对象传递给第一个参数，而右侧运算对象传递给第二个参数。除了重载的函数调用运算符 operator() 之外，其他重载运算符不能含有默认实参（参见 6.5.1 节，第 211 页）。

如果一个运算符函数是成员函数，则它的第一个（左侧）运算对象绑定到隐式的 this 指针上（参见 7.1.2 节，第 231 页），因此，成员运算符函数的（显式）参数数量比运算符的运算对象总数少一个。



当一个重载的运算符是成员函数时，this 绑定到左侧运算对象。成员运算符函数的（显式）参数数量比运算对象的数量少一个。

对于一个运算符函数来说，它或者是类的成员，或者至少含有一个类类型的参数：

```
// 错误：不能为 int 重定义内置的运算符
int operator+(int, int);
```

这一约定意味着当运算符作用于内置类型的运算对象时，我们无法改变该运算符的含义。

我们可以重载大多数（但不是全部）运算符。表 14.1 指明了哪些运算符可以被重载，哪些不行。我们将在 19.1.1 节（第 726 页）介绍重载 new 和 delete 的方法。

我们只能重载已有的运算符，而无权发明新的运算符号。例如，我们不能提供 operator** 来执行幂操作。

有四个符号（+、-、*、&）既是一元运算符也是二元运算符，所有这些运算符都能被重载，从参数的数量我们可以推断到底定义的是哪种运算符。

553 对于一个重载的运算符来说，其优先级和结合律（参见 4.1.2 节，第 121 页）与对应的内置运算符保持一致。不考虑运算对象类型的话，

```
x == y + z;
```

永远等价于 $x == (y + z)$ 。

表 14.1: 运算符

可以被重载的运算符						
+	-	*	/	%	^	
&		~	!	,	=	
<	>	<=	>=	++	--	
<<	>>	==	!=	&&		
+=	-=	/=	%=	^=	&=	
=	*=	<<=	>>=	[]	()	
->	->*	new	new[]	delete	delete[]	
不能被重载的运算符						
:	.*	.	.	?:		

直接调用一个重载的运算符函数

通常情况下，我们将运算符作用于类型正确的实参，从而以这种间接方式“调用”重载的运算符函数。然而，我们也能像调用普通函数一样直接调用运算符函数，先指定函数名字，然后传入数量正确、类型适当的实参：

```
// 一个非成员运算符函数的等价调用
data1 + data2;                                // 普通的表达式
operator+(data1, data2);                      // 等价的函数调用
```

这两次调用是等价的，它们都调用了非成员函数 `operator+`，传入 `data1` 作为第一个实参、传入 `data2` 作为第二个实参。

我们像调用其他成员函数一样显式地调用成员运算符函数。具体做法是，首先指定运行函数的对象（或指针）的名字，然后使用点运算符（或箭头运算符）访问希望调用的函数：

```
data1 += data2;                                // 基于“调用”的表达式
data1.operator+=(data2);                      // 对成员运算符函数的等价调用
```

这两条语句都调用了成员函数 `operator+=`，将 `this` 绑定到 `data1` 的地址、将 `data2` 作为实参传入了函数。

某些运算符不应该被重载

回忆之前介绍过的，某些运算符指定了运算对象求值的顺序。因为使用重载的运算符本质上是一次函数调用，所以这些关于运算对象求值顺序的规则无法应用到重载的运算符上。特别是，逻辑与运算符、逻辑或运算符（参见 4.3 节，第 126 页）和逗号运算符（参见 4.10 节，第 140 页）的运算对象求值顺序规则无法保留下来。除此之外，`&&` 和 `||` 运算符的重载版本也无法保留内置运算符的短路求值属性，两个运算对象总是会被求值。554

因为上述运算符的重载版本无法保留求值顺序和/或短路求值属性，因此不建议重载它们。当代码使用了这些运算符的重载版本时，用户可能会突然发现他们一直习惯的求值规则不再适用了。

还有一个原因使得我们一般不重载逗号运算符和取地址运算符：C++语言已经定义了这两种运算符用于类类型对象时的特殊含义，这一点与大多数运算符都不相同。因为这两种运算符已经有了内置的含义，所以一般来说它们不应该被重载，否则它们的行为将异于常态，从而导致类的用户无法适应。

Best Practices

通常情况下，不应该重载逗号、取地址、逻辑与和逻辑或运算符。

使用与内置类型一致的含义

当你开始设计一个类时，首先应该考虑的是这个类将提供哪些操作。在确定类需要哪些操作之后，才能思考到底应该把每个类操作设成普通函数还是重载的运算符。如果某些操作在逻辑上与运算符相关，则它们适合于定义成重载的运算符：

- 如果类执行 IO 操作，则定义移位运算符使其与内置类型的 IO 保持一致。
- 如果类的某个操作是检查相等性，则定义 `operator==`；如果类有了 `operator==`，意味着它通常也应该有 `operator!=`。
- 如果类包含一个内在的单序比较操作，则定义 `operator<`；如果类有了 `operator<`，则它也应该含有其他关系操作。
- 重载运算符的返回类型通常情况下应该与其内置版本的返回类型兼容：逻辑运算符和关系运算符应该返回 `bool`，算术运算符应该返回一个类类型的值，赋值运算符和复合赋值运算符则应该返回左侧运算对象的一个引用。

提示：尽量明智地使用运算符重载

每个运算符在用于内置类型时都有比较明确的含义。以二元`+`运算符为例，它明显执行的是加法操作。因此，把二元`+`运算符映射到类类型的一个类似操作上可以极大地简化记忆。例如对于标准库类型 `string` 来说，我们就会使用`+`把一个 `string` 对象连接到另一个后面，很多编程语言都有类似的用法。

当在内置的运算符和我们自己的操作之间存在逻辑映射关系时，运算符重载的效果最好。此时，使用重载的运算符显然比另起一个名字更自然也更直观。不过，过分滥用运算符重载也会使我们的类变得难以理解。

在实际编程过程中，一般没有特别明显的滥用运算符重载的情况。例如，一般来说没有哪个程序员会定义 `operator+` 并让它执行减法操作。然而经常发生的一种情况是，程序员可能会强行扭曲了运算符的“常规”含义使得其适应某种给定的类型，这显然是我们不希望发生的。因此我们的建议是：只有当操作的含义对于用户来说清晰明了时才使用运算符。如果用户对运算符可能有几种不同的理解，则使用这样的运算符将产生二义性。

赋值和复合赋值运算符

赋值运算符的行为与复合版本的类似：赋值之后，左侧运算对象和右侧运算对象的值相等，并且运算符应该返回它左侧运算对象的一个引用。重载的赋值运算应该继承而非违背其内置版本的含义。

如果类含有算术运算符（参见 4.2 节，第 124 页）或者位运算符（参见 4.8 节，第 136 页），则最好也提供对应的复合赋值运算符。无须赘言，`+=` 运算符的行为显然应该与其内置版本一致，即先执行`+`，再执行`=`。

选择作为成员或者非成员

当我们定义重载的运算符时，必须首先决定是将其声明为类的成员函数还是声明为一个普通的非成员函数。在某些时候我们别无选择，因为有的运算符必须作为成员；另一些

情况下，运算符作为普通函数比作为成员更好。

下面的准则有助于我们在将运算符定义为成员函数还是普通的非成员函数做出抉择：

- 赋值（=）、下标（[]）、调用（()）和成员访问箭头（->）运算符必须是成员。
- 复合赋值运算符一般来说应该是成员，但并非必须，这一点与赋值运算符略有不同。
- 改变对象状态的运算符或者与给定类型密切相关的运算符，如递增、递减和解引用运算符，通常应该是成员。
- 具有对称性的运算符可能转换任意一端的运算对象，例如算术、相等性、关系和位运算符等，因此它们通常应该是普通的非成员函数。

程序员希望能在含有混合类型的表达式中使用对称性运算符。例如，我们能求一个 int 和一个 double 的和，因为它们中的任意一个都可以是左侧运算对象或右侧运算对象，所以加法是对称的。如果我们想提供含有类对象的混合类型表达式，则运算符必须定义成非成员函数。556

当我们把运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象。例如：

```
string s = "world";
string t = s + "!"; // 正确：我们能把一个 const char* 加到一个 string 对象中
string u = "hi" + s; // 如果+是 string 的成员，则产生错误
```

如果 operator+ 是 string 类的成员，则上面的第一个加法等价于 s.operator+("!"）。同样的，"hi"+s 等价于 "hi".operator+(s)。显然 "hi" 的类型是 const char*，这是一种内置类型，根本就没有成员函数。

因为 string 将 + 定义成了普通的非成员函数，所以 "hi"+s 等价于 operator+("hi", s)。和任何其他函数调用一样，每个实参都能被转换成形参类型。唯一的要求是至少有一个运算对象是类类型，并且两个运算对象都能准确无误地转换成 string。

14.1 节练习

练习 14.1：在什么情况下重载的运算符与内置运算符有所区别？在什么情况下重载的运算符又与内置运算符一样？

练习 14.2：为 Sales_data 编写重载的输入、输出、加法和复合赋值运算符。

练习 14.3： string 和 vector 都定义了重载的 == 以比较各自的对象，假设 svec1 和 svec2 是存放 string 的 vector，确定在下面的表达式中分别使用了哪个版本的 == ？

- | | |
|-------------------------|--------------------------|
| (a) "cobble" == "stone" | (b) svec1[0] == svec2[0] |
| (c) svec1 == svec2 | (d) "svec1[0] == "stone" |

练习 14.4：如何确定下列运算符是否应该是类的成员？

- (a) % (b) %= (c) ++ (d) -> (e) << (f) && (g) == (h) ()

练习 14.5：在 7.5.1 节的练习 7.40（第 261 页）中，编写了下列类中某一个的框架，请问在这个类中应该定义重载的运算符吗？如果是，请写出来。

- | | | |
|-------------|------------|--------------|
| (a) Book | (b) Date | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree |

14.2 输入和输出运算符

如我们所知，IO 标准库分别使用`>>`和`<<`执行输入和输出操作。对于这两个运算符来说，IO 库定义了用其读写内置类型的版本，而类则需要自定义适合其对象的新版本以支持 IO 操作。

14.2.1 重载输出运算符`<<`

通常情况下，输出运算符的第一个形参是一个非常量 `ostream` 对象的引用。之所以 `ostream` 是非常量是因为向流写入内容会改变其状态；而该形参是引用是因为我们无法直接复制一个 `ostream` 对象。

第二个形参一般来说是一个常量的引用，该常量是我们想要打印的类类型。第二个形参是引用的原因是我们希望避免复制实参；而之所以该形参可以是常量是因为（通常情况下）打印对象不会改变对象的内容。

为了与其他输出运算符保持一致，`operator<<`一般要返回它的 `ostream` 形参。

Sales_data 的输出运算符

举个例子，我们按照如下形式编写 `Sales_data` 的输出运算符：

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

除了名字之外，这个函数与之前的 `print` 函数（参见 7.1.3 节，第 234 页）完全一样。打印一个 `Sales_data` 对象意味着要分别打印它的三个数据成员以及通过计算得到的平均销售价格，每个元素以空格隔开。完成输出后，运算符返回刚刚使用的 `ostream` 的引用。

输出运算符尽量减少格式化操作

用于内置类型的输出运算符不太考虑格式化操作，尤其不会打印换行符，用户希望类的输出运算符也像如此行事。如果运算符打印了换行符，则用户就无法在对象的同一行内接着打印一些描述性的文本了。相反，令输出运算符尽量减少格式化操作可以使用户有权控制输出的细节。

Best Practices

通常，输出运算符应该主要负责打印对象的内容而非控制格式，输出运算符不应该打印换行符。

输入输出运算符必须是非成员函数

与 `iostream` 标准库兼容的输入输出运算符必须是普通的非成员函数，而不能是类的成员函数。否则，它们的左侧运算对象将是我们的类的一个对象：

```
Sales_data data;
data << cout;           // 如果 operator<< 是 Sales_data 的成员
```

假设输入输出运算符是某个类的成员，则它们也必须是 `istream` 或 `ostream` 的成员。然而，这两个类属于标准库，并且我们无法给标准库中的类添加任何成员。

因此，如果我们希望为类自定义 IO 运算符，则必须将其定义成非成员函数。当然，IO 运算符通常需要读写类的非公有数据成员，所以 IO 运算符一般被声明为友元（参见 7.2.1 节，第 241 页）。 558

14.2.1 节练习

练习 14.6：为你的 Sales_data 类定义输出运算符。

练习 14.7：你在 13.5 节的练习（第 470 页）中曾经编写了一个 String 类，为它定义一个输出运算符。

练习 14.8：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输出运算符。

14.2.2 重载输入运算符>>



通常情况下，输入运算符的第一个形参是运算符将要读取的流的引用，第二个形参是将要读入到的（非常量）对象的引用。该运算符通常会返回某个给定流的引用。第二个形参之所以必须是个非常量是因为输入运算符本身的目的就是将数据读入到这个对象中。

Sales_data 的输入运算符

举个例子，我们将按照如下形式编写 Sales_data 的输入运算符：

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price; // 不需要初始化，因为我们将先读入数据到 price，之后才使用它
    is >> item.bookNo >> item.units_sold >> price;
    if (is) // 检查输入是否成功
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // 输入失败：对象被赋予默认的状态
    return is;
}
```

除了 if 语句之外，这个定义与之前的 read 函数（参见 7.1.3 节，第 234 页）完全一样。if 语句检查读取操作是否成功，如果发生了 IO 错误，则运算符将给定的对象重置为空 Sales_data，这样可以确保对象处于正确的状态。



输入运算符必须处理输入可能失败的情况，而输出运算符不需要。

输入时的错误

559

在执行输入运算符时可能发生下列错误：

- 当流含有错误类型的数据时读取操作可能失败。例如在读取完 bookNo 后，输入运算符假定接下来读入的是两个数字数据，一旦输入的不是数字数据，则读取操作及后续对流的其他使用都将失败。
- 当读取操作到达文件末尾或者遇到输入流的其他错误时也会失败。

在程序中我们没有逐个检查每个读取操作，而是等读取了所有数据后赶在使用这些数据前一次性检查：

```

if (is)                                // 检查输入是否成功
    item.revenue = item.units_sold * price;
else
    item = Sales_data();      // 输入失败：对象被赋予默认的状态

```

如果读取操作失败，则 `price` 的值将是未定义的。因此，在使用 `price` 前我们需要首先检查输入流的合法性，然后才能执行计算并将结果存入 `revenue`。如果发生了错误，我们无须在意到底是哪部分输入失败，只要将一个新的默认初始化的 `Sales_data` 对象赋予 `item` 从而将其重置为空 `Sales_data` 就可以了。执行这样的赋值后，`item` 的 `bookNo` 成员将是一个空 `string`，`revenue` 和 `units_sold` 成员将等于 0。

如果在发生错误前对象已经有一部分被改变，则适时地将对象置为合法状态显得异常重要。例如在这个输入运算符中，我们可能在成功读取新的 `bookNo` 后遇到错误，这意味着对象的 `units_sold` 和 `revenue` 成员并没有改变，因此有可能会将这两个数据与一条完全不匹配的 `bookNo` 组合在一起。

通过将对象置为合法的状态，我们能（略微）保护使用者免于受到输入错误的影响。此时的对象处于可用状态，即它的成员都是被正确定义的。而且该对象也不会产生误导性的结果，因为它的数据在本质上确实是一体的。



当读取操作发生错误时，输入运算符应该负责从错误中恢复。

标示错误

一些输入运算符需要做更多数据验证的工作。例如，我们的输入运算符可能需要检查 `bookNo` 是否符合规范的格式。在这样的例子中，即使从技术上来看 IO 是成功的，输入运算符也应该设置流的条件状态以标示出失败信息（参见 8.1.2 节，第 279 页）。通常情况下，输入运算符只设置 `failbit`。除此之外，设置 `eofbit` 表示文件耗尽，而设置 `badbit` 表示流被破坏。最好的方式是由 IO 标准库自己来标示这些错误。

560 >

14.2.2 节练习

练习 14.9: 为你的 `Sales_data` 类定义输入运算符。

练习 14.10: 对于 `Sales_data` 的输入运算符来说如果给定了下面的输入将发生什么情况？

- (a) 0-201-99999-9 10 24.95 (b) 10 24.95 0-210-99999-9

练习 14.11: 下面的 `Sales_data` 输入运算符存在错误吗？如果有，请指出来。对于这个输入运算符如果仍然给定上个练习的输入将发生什么情况？

```

istream& operator>>(istream& in, Sales_data& s)
{
    double price;
    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}

```

练习 14.12: 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输入运算符并确保该运算符可以处理输入错误。

14.3 算术和关系运算符

通常情况下，我们把算术和关系运算符定义成非成员函数以允许对左侧或右侧的运算对象进行转换（参见 14.1 节，第 492 页）。因为这些运算符一般不需要改变运算对象的状态，所以形参都是常量的引用。

算术运算符通常会计算它的两个运算对象并得到一个新值，这个值有别于任意一个运算对象，常常位于一个局部变量之内，操作完成后返回该局部变量的副本作为其结果。如果类定义了算术运算符，则它一般也会定义一个对应的复合赋值运算符。此时，最有效的方式是使用复合赋值来定义算术运算符：

```
// 假设两个对象指向同一本书
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;           // 把 lhs 的数据成员拷贝给 sum
    sum += rhs;                   // 将 rhs 加到 sum 中
    return sum;
}
```

这个定义与原来的 add 函数（参见 7.1.3 节，第 234 页）是完全等价的。我们把 lhs 拷贝给局部变量 sum，然后使用 Sales_data 的复合赋值运算符（将在第 500 页定义）将 rhs 的值加到 sum 中，最后函数返回 sum 的副本。



如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符。

561

14.3 节练习

练习 14.13：你认为 Sales_data 类还应该支持哪些其他算术运算符（参见表 4.1，第 124 页）？如果有的话，请给出它们的定义。

练习 14.14：你觉得为什么调用 operator+= 来定义 operator+ 比其他方法更有效？

练习 14.15：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有其他算术运算符吗？如果是，请实现它们；如果不是，解释原因。

14.3.1 相等运算符



通常情况下，C++ 中的类通过定义相等运算符来检验两个对象是否相等。也就是说，它们会比较对象的每一个数据成员，只有当所有对应的成员都相等时才认为两个对象相等。依据这一思想，我们的 Sales_data 类的相等运算符不但应该比较 bookNo，还应该比较具体的销售数据：

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
```

```

{
    return !(lhs == rhs);
}

```

就上面这些函数的定义本身而言，它们似乎比较简单，也没什么价值，对于我们来说重要的是从这些函数中体现出来的设计准则：

- 如果一个类含有判断两个对象是否相等的操作，则它显然应该把函数定义成 `operator==` 而非一个普通的命名函数：因为用户肯定希望能使用 `==` 比较对象，所以提供了 `==` 就意味着用户无须再费时费力地学习并记忆一个全新的函数名字。此外，类定义了 `==` 运算符之后也更容易使用标准库容器和算法。
- 如果类定义了 `operator==`，则该运算符应该能判断一组给定的对象中是否含有重复数据。
- 通常情况下，相等运算符应该具有传递性，换句话说，如果 `a==b` 和 `b==c` 都为真，则 `a==c` 也应该为真。
- 如果类定义了 `operator==`，则这个类也应该定义 `operator!=`。对于用户来说，当他们能使用 `==` 时肯定也希望使用 `!=`，反之亦然。
- 相等运算符和不相等运算符中的一个应该把工作委托给另外一个，这意味着其中一个运算符应该负责实际比较对象的工作，而另一个运算符则只是调用那个真正工作的运算符。



如果某个类在逻辑上有相等性的含义，则该类应该定义 `operator==`，这样做可以使得用户更容易使用标准库算法来处理这个类。

14.3.1 节练习

练习 14.16：为你的 `StrBlob` 类（参见 12.1.1 节，第 405 页）、`StrBlobPtr` 类（参见 12.1.6 节，第 421 页）、`StrVec` 类（参见 13.5 节，第 465 页）和 `String` 类（参见 13.5 节，第 470 页）分别定义相等运算符和不相等运算符。

练习 14.17：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有相等运算符吗？如果是，请实现它；如果不是，解释原因。



14.3.2 关系运算符

定义了相等运算符的类也常常（但不总是）包含关系运算符。特别是，因为关联容器和一些算法要用到小于运算符，所以定义 `operator<` 会比较有用。

通常情况下关系运算符应该

1. 定义顺序关系，令其与关联容器中对关键字的要求一致（参见 11.2.2 节，第 378 页）；并且
2. 如果类同时也含有 `==` 运算符的话，则定义一种关系令其与 `==` 保持一致。特别是，如果两个对象是 `!=` 的，那么一个对象应该 `<` 另外一个。



尽管我们可能会认为 `Sales_data` 类应该支持关系运算符，但事实证明并非如此，其中的缘由比较微妙，值得读者深思。

一开始我们可能会认为应该像 `compareIsbn`（参见 11.2.2 节，第 379 页）那样定义 `<`，该函数通过比较 `ISBN` 来实现对两个对象的比较。然而，尽管 `compareIsbn` 提供的

顺序关系符合要求 1，但是函数得到的结果显然与我们定义的`==`不一致，因此它不满足要求 2。

对于 `Sales_data` 的`==`运算符来说，如果两笔交易的 `revenue` 和 `units_sold` 成员不同，那么即使它们的 `ISBN` 相同也无济于事，它们仍然是不相等的。如果我们定义的`<`运算符仅仅比较 `ISBN` 成员，那么将发生这样的情况：两个 `ISBN` 相同但 `revenue` 和 `units_sold` 不同的对象经比较是不相等的，但是其中的任何一个都不比另一个小。然而实际情况是，如果我们有两个对象并且哪个都不比另一个小，则从道理上来讲这两个对象应该是相等的。563

基于上述分析我们也许会认为，只要让 `operator<` 依次比较每个数据元素就能解决问题了，比方说让 `operator<` 先比较 `isbn`，相等的话继续比较 `units_sold`，还相等再继续比较 `revenue`。

然而，这样的排序没有任何必要。根据将来使用 `Sales_data` 类的实际需要，我们可能会希望先比较 `units_sold`，也可能希望先比较 `revenue`。有的时候，我们希望 `units_sold` 少的对象“小于”`units_sold` 多的对象；另一些时候，则可能希望 `revenue` 少的对象“小于”`revenue` 多的对象。

因此对于 `Sales_data` 类来说，不存在一种逻辑可靠的`<` 定义，这个类不定义`<` 运算符也许更好。



如果存在唯一一种逻辑可靠的`<` 定义，则应该考虑为这个类定义`<` 运算符。如果类同时还包含`==`，则当且仅当`<` 的定义和`==` 产生的结果一致时才定义`<` 运算符。

14.3.2 节练习

练习 14.18：为你的 `StrBlob` 类、`StrBlobPtr` 类、`StrVec` 类和 `String` 类定义关系运算符。

练习 14.19：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有关系运算符吗？如果是，请实现它；如果不是，解释原因。

14.4 赋值运算符

之前已经介绍过拷贝赋值和移动赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页），它们可以把类的一个对象赋值给该类的另一个对象。此外，类还可以定义其他赋值运算符以使用别的类型作为右侧运算对象。

举个例子，在拷贝赋值和移动赋值运算符之外，标准库 `vector` 类还定义了第三种赋值运算符，该运算符接受花括号内的元素列表作为参数（参见 9.2.5 节，第 302 页）。我们能以如下的形式使用该运算符：

```
vector<string> v;
v = {"a", "an", "the"};
```

同样，也可以把这个运算符添加到 `StrVec` 类中（参见 13.5 节，第 465 页）：

```
class StrVec {
public:
    StrVec &operator=(std::initializer_list<std::string>);
    // 其他成员与 13.5 节（第 465 页）一致
};
```

564> 为了与内置类型的赋值运算符保持一致（也与我们已经定义的拷贝赋值和移动赋值运算一致），这个新的赋值运算符将返回其左侧运算对象的引用：

```
StrVec &StrVec::operator=(initializer_list<string> il)
{
    // alloc_n_copy 分配内存空间并从给定范围内拷贝元素
    auto data = alloc_n_copy(il.begin(), il.end());
    free();           // 销毁对象中的元素并释放内存空间
    elements = data.first; // 更新数据成员使其指向新空间
    first_free = cap = data.second;
    return *this;
}
```

和拷贝赋值及移动赋值运算符一样，其他重载的赋值运算符也必须先释放当前内存空间，再创建一片新空间。不同之处是，这个运算符无须检查对象向自身的赋值，这是因为它的形参 `initializer_list<string>`（参见 6.2.6 节，第 198 页）确保 `il` 与 `this` 所指的不是同一个对象。



我们可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。

复合赋值运算符

复合赋值运算符非得是类的成员，不过我们还是倾向于把包括复合赋值在内的所有赋值运算都定义在类的内部。为了与内置类型的复合赋值保持一致，类中的复合赋值运算符也要返回其左侧运算对象的引用。例如，下面是 `Sales_data` 类中复合赋值运算符的定义：

```
// 作为成员的二元运算符：左侧运算对象绑定到隐式的 this 指针
// 假定两个对象表示的是同一本书
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```



赋值运算符必须定义成类的成员，复合赋值运算符通常情况下也应该这样做。这两类运算符都应该返回左侧运算对象的引用。

14.4 节练习

练习 14.20: 为你的 `Sales_data` 类定义加法和复合赋值运算符。

练习 14.21: 编写 `Sales_data` 类的`+和+=`运算符，使得`+`执行实际的加法操作而`+=`调用`+`。相比于 14.3 节（第 497 页）和 14.4 节（第 500 页）对这两个运算符的定义，本题的定义有何缺点？试讨论之。

练习 14.22: 定义赋值运算符的一个新版本，使得我们能把一个表示 ISBN 的 `string` 赋给一个 `Sales_data` 对象。

练习 14.23: 为你的 `StrVec` 类定义一个 `initializer_list` 赋值运算符。

练习 14.24: 你在 7.5.1 节的练习 7.40 (第 261 页) 中曾经选择并编写了一个类, 你认为它应该含有拷贝赋值和移动赋值运算符吗? 如果是, 请实现它们。

练习 14.25: 上题的这个类还需要定义其他赋值运算符吗? 如果是, 请实现它们; 同时说明运算对象应该是什么类型并解释原因。

14.5 下标运算符

表示容器的类通常可以通过元素在容器中的位置访问元素, 这些类一般会定义下标运算符 `operator[]`。



下标运算符必须是成员函数。

565

为了与下标的原始定义兼容, 下标运算符通常以所访问元素的引用作为返回值, 这样做的好处是下标可以出现在赋值运算符的任意一端。进一步, 我们最好同时定义下标运算符的常量版本和非常量版本, 当作用于一个常量对象时, 下标运算符返回常量引用以确保我们不会给返回的对象赋值。



如果一个类包含下标运算符, 则它通常会定义两个版本: 一个返回普通引用, 另一个是类的常量成员并且返回常量引用。

举个例子, 我们按照如下形式定义 `StrVec` (参见 13.5 节, 第 465 页) 的下标运算符:

```
class StrVec {
public:
    std::string& operator[](std::size_t n)
    { return elements[n]; }
    const std::string& operator[](std::size_t n) const
    { return elements[n]; }
    // 其他成员与 13.5 (第 465 页) 一致
private:
    std::string *elements;           // 指向数组首元素的指针
};
```

上面这两个下标运算符的用法类似于 `vector` 或者数组中的下标。因为下标运算符返回的是元素的引用, 所以当 `StrVec` 是非常量时, 我们可以给元素赋值; 而当我们对常量对象取下标时, 不能为其赋值:

```
// 假设 svec 是一个 StrVec 对象
const StrVec cvec = svec;           // 把 svec 的元素拷贝到 cvec 中
// 如果 svec 中含有元素, 对第一个元素运行 string 的 empty 函数
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero";               // 正确: 下标运算符返回 string 的引用
    cvec[0] = "Zip";                // 错误: 对 cvec 取下标返回的是常量引用
}
```

566

14.5 节练习

练习 14.26: 为你的 StrBlob 类、StrBlobPtr 类、StrVec 类和 String 类定义下标运算符。

14.6 递增和递减运算符

在迭代器类中通常会实现递增运算符（`++`）和递减运算符（`--`），这两种运算符使得类可以在元素的序列中前后移动。C++语言并不要求递增和递减运算符必须是类的成员，但是因为它们改变的正好是所操作对象的状态，所以建议将其设定为成员函数。

对于内置类型来说，递增和递减运算符既有前置版本也有后置版本。同样，我们也应该为类定义两个版本的递增和递减运算符。接下来我们首先介绍前置版本，然后实现后置版本。



定义递增和递减运算符的类应该同时定义前置版本和后置版本。这些运算符通常应该被定义成类的成员。

定义前置递增/递减运算符

为了说明递增和递减运算符，我们不妨在 StrBlobPtr 类（参见 12.1.6 节，第 421 页）中定义它们：

```
class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr& operator++(); // 前置运算符
    StrBlobPtr& operator--();
    // 其他成员和之前的版本一致
};
```



为了与内置版本保持一致，前置运算符应该返回递增或递减后对象的引用。

567

递增和递减运算符的工作机理非常相似：它们首先调用 `check` 函数检验 `StrBlobPtr` 是否有效，如果是，接着检查给定的索引值是否有效。如果 `check` 函数没有抛出异常，则运算符返回对象的引用。

在递增运算符的例子中，我们把 `curr` 的当前值传递给 `check` 函数。如果这个值小于 `vector` 的大小，则 `check` 正常返回；否则，如果 `curr` 已经到达了 `vector` 的末尾，`check` 将抛出异常：

```
// 前置版本：返回递增/递减对象的引用
StrBlobPtr& StrBlobPtr::operator++()
{
    // 如果 curr 已经指向了容器的尾后位置，则无法递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 将 curr 在当前状态下向前移动一个元素
    return *this;
}
```

```

StrBlobPtr& StrBlobPtr::operator--()
{
    // 如果 curr 是 0，则继续递减它将产生一个无效下标
    --curr;                                // 将 curr 在当前状态下向后移动一个元素
    check(curr, "decrement past begin of StrBlobPtr");
    return *this;
}

```

递减运算符先递减 curr，然后调用 check 函数。此时，如果 curr（一个无符号数）已经是 0 了，那么我们传递给 check 的值将是一个表示无效下标的非常大的正数值（参见 2.1.2 节，第 33 页）。

区分前置和后置运算符

要想同时定义前置和后置运算符，必须首先解决一个问题，即普通的重载形式无法区分这两种情况。前置和后置版本使用的是同一个符号，意味着其重载版本所用的名字将是相同的，并且运算对象的数量和类型也相同。

为了解决这个问题，后置版本接受一个额外的（不被使用）int 类型的形参。当我们使用后置运算符时，编译器为这个形参提供一个值为 0 的实参。尽管从语法上来说后置函数可以使用这个额外的形参，但是在实际过程中通常不会这么做。这个形参的唯一作用就是区分前置版本和后置版本的函数，而不是真的要在实现后置版本时参与运算。

接下来我们为 StrBlobPtr 添加后置运算符：

```

class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr operator++(int);           // 后置运算符
    StrBlobPtr operator--(int);
    // 其他成员和之前的版本一致
};

```



为了与内置版本保持一致，后置运算符应该返回对象的原值（递增或递减之前 的值），返回的形式是一个值而非引用。

568

对于后置版本来说，在递增对象之前需要首先记录对象的状态：

```

// 后置版本：递增/递减对象的值但是返回原值
StrBlobPtr StrBlobPtr::operator++(int)
{
    // 此处无须检查有效性，调用前置递增运算时才需要检查
    StrBlobPtr ret = *this; // 记录当前的值
    ++*this;              // 向前移动一个元素，前置++需要检查递增的有效性
    return ret;            // 返回之前记录的状态
}
StrBlobPtr StrBlobPtr::operator--(int)
{
    // 此处无须检查有效性，调用前置递减运算时才需要检查
    StrBlobPtr ret = *this; // 记录当前的值
    --*this;              // 向后移动一个元素，前置--需要检查递减的有效性
    return ret;            // 返回之前记录的状态
}

```

由上可知，我们的后置运算符调用各自的前置版本来完成实际的工作。例如后置递增运算符执行

```
++*this
```

该表达式调用前置递增运算符，前置递增运算符首先检查递增操作是否安全，根据检查的结果抛出一个异常或者执行递增 curr 的操作。假定通过了检查，则后置函数返回事先存好的 ret 的副本。因此最终的效果是，对象本身向前移动了一个元素，而返回的结果仍然反映对象在未递增之前原始的值。



因为我们不会用到 int 形参，所以无须为其命名。

显式地调用后置运算符

如在第 491 页介绍的，可以显式地调用一个重载的运算符，其效果与在表达式中以运算符号的形式使用它完全一样。如果我们想通过函数调用的方式调用后置版本，则必须为它的整型参数传递一个值：

```
StrBlobPtr p(a1);           // p 指向 a1 中的 vector
p.operator++(0);            // 调用后置版本的 operator++
p.operator++();             // 调用前置版本的 operator++
```

尽管传入的值通常会被运算符函数忽略，但却必不可少，因为编译器只有通过它才能知道应该使用后置版本。

569

14.6 节练习

练习 14.27：为你的 StrBlobPtr 类添加递增和递减运算符。

练习 14.28：为你的 StrBlobPtr 类添加加法和减法运算符，使其可以实现指针的算术运算（参见 3.5.3 节，第 106 页）。

练习 14.29：为什么不定义 const 版本的递增和递减运算符？

14.7 成员访问运算符

在迭代器类及智能指针类（参见 12.1 节，第 400 页）中常常用到解引用运算符 (*) 和箭头运算符 (->)。我们以如下形式向 StrBlobPtr 类添加这两种运算符：

```
class StrBlobPtr {
public:
    std::string& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr];           // (*p) 是对象所指的 vector
    }
    std::string* operator->() const
    { // 将实际工作委托给解引用运算符
      return & this->operator*();
    }
    // 其他成员与之前的版本一致
}
```

解引用运算符首先检查 curr 是否仍在作用范围内，如果是，则返回 curr 所指元素的一个引用。箭头运算符不执行任何自己的操作，而是调用解引用运算符并返回解引用结果元素的地址。



箭头运算符必须是类的成员。解引用运算符通常也是类的成员，尽管并非必须如此。

值得注意的是，我们将这两个运算符定义成了 const 成员，这是因为与递增和递减运算符不一样，获取一个元素并不会改变 StrBlobPtr 对象的状态。同时，它们的返回值分别是非常量 string 的引用或指针，因为一个 StrBlobPtr 只能绑定到非常量的 StrBlob 对象（参见 12.1.6 节，第 421 页）。

这两个运算符的用法与指针或者 vector 迭代器的对应操作完全一致：

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1);                                // p 指向 a1 中的 vector
*p = "okay";                                     // 给 a1 的首元素赋值
cout << p->size() << endl;                      // 打印 4，这是 a1 首元素的大小
cout << (*p).size() << endl;                     // 等价于 p->size()
```

对箭头运算符返回值的限定

570

和大多数其他运算符一样（尽管这么做不太好），我们能令 operator* 完成任何我们指定的操作。换句话说，我们可以让 operator* 返回一个固定值 42，或者打印对象的内容，或者其他。箭头运算符则不是这样，它永远不能丢掉成员访问这个最基本的含义。当我们重载箭头时，可以改变的是箭头从哪个对象当中获取成员，而箭头获取成员这一事实则永远不变。

对于形如 point->mem 的表达式来说，point 必须是指向类对象的指针或者是一个重载了 operator-> 的类的对象。根据 point 类型的不同，point->mem 分别等价于

```
(*point).mem;                                // point 是一个内置的指针类型
point.operator()->mem;                         // point 是类的一个对象
```

除此之外，代码都将发生错误。point->mem 的执行过程如下所示：

- 如果 point 是指针，则我们应用内置的箭头运算符，表达式等价于 (*point).mem。首先解引用该指针，然后从所得的对象中获取指定的成员。如果 point 所指的类型没有名为 mem 的成员，程序会发生错误。
- 如果 point 是定义了 operator-> 的类的一个对象，则我们使用 point.operator->() 的结果来获取 mem。其中，如果该结果是一个指针，则执行第 1 步；如果该结果本身含有重载的 operator->()，则重复调用当前步骤。最终，当这一过程结束时程序或者返回了所需的内容，或者返回一些表示程序错误的信息。



重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对象。

14.7 节练习

练习 14.30: 为你的 StrBlobPtr 类和在 12.1.6 节练习 12.22（第 423 页）中定义的 ConstStrBlobPtr 类分别添加解引用运算符和箭头运算符。注意：因为 ConstStrBlobPtr 的数据成员指向 const vector，所以 ConstStrBlobPtr 中的运算符必须返回常量引用。

练习 14.31: 我们的 StrBlobPtr 类没有定义拷贝构造函数、赋值运算符及析构函数，为什么？

练习 14.32: 定义一个类令其含有指向 StrBlobPtr 对象的指针，为这个类定义重载的箭头运算符。



14.8 函数调用运算符

571

如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。因为这样的类同时也能存储状态，所以与普通函数相比它们更加灵活。

举个简单的例子，下面这个名为 absInt 的 struct 含有一个调用运算符，该运算符负责返回其参数的绝对值：

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};
```

这个类只定义了一种操作：函数调用运算符，它负责接受一个 int 类型的实参，然后返回该实参的绝对值。

我们使用调用运算符的方式是令一个 absInt 对象作用于一个实参列表，这一过程看起来非常像调用函数的过程：

```
int i = -42;
absInt absObj;           // 含有函数调用运算符的对象
int ui = absObj(i);      // 将 i 传递给 absObj.operator()
```

即使 absObj 只是一个对象而非函数，我们也能“调用”该对象。调用对象实际上是在运行重载的调用运算符。在此例中，该运算符接受一个 int 值并返回其绝对值。



函数调用运算符必须是成员函数。一个类可以定义多个不同版本的调用运算符，相互之间应该在参数数量或类型上有所区别。

如果类定义了调用运算符，则该类的对象称作 **函数对象**（function object）。因为可以调用这种对象，所以我们说这些对象的“行为像函数一样”。

含有状态的函数对象类

和其他类一样，函数对象类除了 operator() 之外也可以包含其他成员。函数对象类通常含有一些数据成员，这些成员被用于定制调用运算符中的操作。

举个例子，我们将定义一个打印 string 实参内容的类。默认情况下，我们的类会将

内容写入到 cout 中，每个 string 之间以空格隔开。同时也允许类的用户提供其他可写入的流及其他分隔符。我们将该类定义如下：

```
class PrintString {  
public:  
    PrintString(ostream &o = cout, char c = ' '):  
        os(o), sep(c) {}  
    void operator()(const string &s) const { os << s << sep; }  
private:  
    ostream &os;           // 用于写入的目的流  
    char sep;              // 用于将不同输出隔开的字符  
};
```

我们的类有一个构造函数，它接受一个输出流的引用以及一个用于分隔的字符，这两个形参的默认实参（参见 6.5.1 节，第 211 页）分别是 cout 和空格。572之后的函数调用运算符使用这些成员协助其打印给定的 string。

当定义 PrintString 的对象时，对于分隔符及输出流既可以使用默认值也可以提供我们自己的值：

```
PrintString printer;          // 使用默认值，打印到 cout  
printer(s);                  // 在 cout 中打印 s，后面跟一个空格  
PrintString errors(cerr, '\n');  
errors(s);                   // 在 cerr 中打印 s，后面跟一个换行符
```

函数对象常常作为泛型算法的实参。例如，可以使用标准库 for_each 算法（参见 10.3.2 节，第 348 页）和我们自己的 PrintString 类来打印容器的内容：

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

for_each 的第三个实参是类型 PrintString 的一个临时对象，其中我们用 cerr 和换行符初始化了该对象。当程序调用 for_each 时，将会把 vs 中的每个元素依次打印到 cerr 中，元素之间以换行符分隔。

14.8 节练习

练习 14.33: 一个重载的函数调用运算符应该接受几个运算对象？

练习 14.34: 定义一个函数对象类，令其执行 if-then-else 的操作：该类的调用运算符接受三个形参，它首先检查第一个形参，如果成功返回第二个形参的值；如果不成功返回第三个形参的值。

练习 14.35: 编写一个类似于 PrintString 的类，令其从 istream 中读取一行输入，然后返回一个表示我们所读内容的 string。如果读取失败，返回空 string。

练习 14.36: 使用前一个练习定义的类读取标准输入，将每一行保存为 vector 的一个元素。

练习 14.37: 编写一个类令其检查两个值是否相等。使用该对象及标准库算法编写程序，令其替换某个序列中具有给定值的所有实例。

14.8.1 lambda 是函数对象

在前一节中，我们使用一个 PrintString 对象作为调用 for_each 的实参，这一

用法类似于我们在 10.3.2 节（第 346 页）中编写的使用 lambda 表达式的程序。当我们编写了一个 lambda 后，编译器将该表达式翻译成一个未命名类的未命名对象（参见 10.3.3 节，第 349 页）。在 lambda 表达式产生的类中含有一个重载的函数调用运算符，例如，对于我们传递给 stable_sort 作为其最后一个实参的 lambda 表达式来说：

```
// 根据单词的长度对其进行排序，对于长度相同的单词按照字母表顺序排序
stable_sort(words.begin(), words.end(),
[](const string &a, const string &b)
{ return a.size() < b.size();});
```

其行为类似于下面这个类的一个未命名对象

```
class ShorterString {
public:
    bool operator()(const string &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};
```

产生的类只有一个函数调用运算符成员，它负责接受两个 string 并比较它们的长度，它的形参列表和函数体与 lambda 表达式完全一样。如我们在 10.3.3 节（第 352 页）所见，默认情况下 lambda 不能改变它捕获的变量。因此在默认情况下，由 lambda 产生的类当中的函数调用运算符是一个 const 成员函数。如果 lambda 被声明为可变的，则调用运算符就不是 const 的了。

用这个类替代 lambda 表达式后，我们可以重写并重新调用 stable_sort：

```
stable_sort(words.begin(), words.end(), ShorterString());
```

第三个实参是新构建的 ShorterString 对象，当 stable_sort 内部的代码每次比较两个 string 时就会“调用”这一对象，此时该对象将调用运算符的函数体，判断第一个 string 的大小小于第二个时返回 true。

表示 lambda 及相应捕获行为的类

如我们所知，当一个 lambda 表达式通过引用捕获变量时，将由程序负责确保 lambda 执行时引用的对象确实存在（参见 10.3.3 节，第 350 页）。因此，编译器可以直接使用该引用而无须在 lambda 产生的类中将其存储为数据成员。

相反，通过值捕获的变量被拷贝到 lambda 中（参见 10.3.3 节，第 350 页）。因此，这种 lambda 产生的类必须为每个值捕获的变量建立对应的数据成员，同时创建构造函数，令其使用捕获的变量的值来初始化数据成员。举个例子，在 10.3.2 节（第 347 页）中有一个 lambda，它的作用是找到第一个长度不小于给定值的 string 对象：

```
// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(),
[sz](const string &a)
{ return a.size() >= sz;});
```

该 lambda 表达式产生的类将形如：

```
574> class SizeComp {
    SizeComp(size_t n): sz(n) {} // 该形参对应捕获的变量
    // 该调用运算符的返回类型、形参和函数体都与 lambda 一致
    bool operator()(const string &s) const
    { return s.size() >= sz; }
```

```

private:
    size_t sz; // 该数据成员对应通过值捕获的变量
};

```

和我们的 `ShorterString` 类不同，上面这个类含有一个数据成员以及一个用于初始化该成员的构造函数。这个合成的类不含有默认构造函数，因此要想使用这个类必须提供一个实参：

```

// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));

```

`lambda` 表达式产生的类不含默认构造函数、赋值运算符及默认析构函数；它是否含有默认的拷贝/移动构造函数则通常要视捕获的数据成员类型而定（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。

14.8.1 节练习

练习 14.38：编写一个类令其检查某个给定的 `string` 对象的长度是否与一个阈值相等。使用该对象编写程序，统计并报告在输入的文件中长度为 1 的单词有多少个、长度为 2 的单词又有多少个、……、长度为 10 的单词又有多少个。

练习 14.39：修改上一题的程序令其报告长度在 1 至 9 之间的单词有多少个、长度在 10 以上的单词又有多少个。

练习 14.40：重新编写 10.3.2 节（第 349 页）的 `biggies` 函数，使用函数对象类替换其中的 `lambda` 表达式。

练习 14.41：你认为 C++11 新标准为什么要增加 `lambda`？对于你自己来说，什么情况下会使用 `lambda`，什么情况下会使用类？

14.8.2 标准库定义的函数对象

标准库定义了一组表示算术运算符、关系运算符和逻辑运算符的类，每个类分别定义了一个执行命名操作的调用运算符。例如，`plus` 类定义了一个函数调用运算符用于对一对运算对象执行`+`的操作；`modulus` 类定义了一个调用运算符执行二元的`%`操作；`equal_to` 类执行`==`，等等。

这些类都被定义成模板的形式，我们可以为其指定具体的应用类型，这里的类型即调用运算符的形参类型。例如，`plus<string>` 令 `string` 加法运算符作用于 `string` 对象；`plus<int>` 的运算对象是 `int`；`plus<Sales_data>` 对 `Sales_data` 对象执行加法运算，以此类推：

```

plus<int> intAdd; // 可执行 int 加法的函数对象
negate<int> intNegate; // 可对 int 值取反的函数对象
// 使用 intAdd::operator(int, int) 求 10 和 20 的和
int sum = intAdd(10, 20); // 等价于 sum = 30
sum = intNegate(intAdd(10, 20)); // 等价于 sum = 30
// 使用 intNegate::operator(int) 生成 -10
// 然后将 -10 作为 intAdd::operator(int, int) 的第二个参数
sum = intAdd(10, intNegate(10)); // sum = 0

```

< 575

表 14.2 所列的类型定义在 `functional` 头文件中。

表 14.2: 标准库函数对象

算术	关系	逻辑
<code>plus<Type></code>	<code>equal_to<Type></code>	<code>logical_and<Type></code>
<code>minus<Type></code>	<code>not_equal_to<Type></code>	<code>logical_or<Type></code>
<code>multiplies<Type></code>	<code>greater<Type></code>	<code>logical_not<Type></code>
<code>divides<Type></code>	<code>greater_equal<Type></code>	
<code>modulus<Type></code>	<code>less<Type></code>	
<code>negate<Type></code>	<code>less_equal<Type></code>	

在算法中使用标准库函数对象

表示运算符的函数对象类常用来替换算法中的默认运算符。如我们所知，在默认情况下排序算法使用 `operator<` 将序列按照升序排列。如果要执行降序排列的话，我们可以传入一个 `greater` 类型的对象。该类将产生一个调用运算符并负责执行待排序类型的大于运算。例如，如果 `svec` 是一个 `vector<string>`，

```
// 传入一个临时的函数对象用于执行两个 string 对象的>比较运算
sort(svec.begin(), svec.end(), greater<string>());
```

则上面的语句将按照降序对 `svec` 进行排序。第三个实参是 `greater<string>` 类型的一个未命名的对象，因此当 `sort` 比较元素时，不再是使用默认的`<`运算符，而是调用给定的 `greater` 函数对象。该对象负责在 `string` 元素之间执行`>`比较运算。

需要特别注意的是，标准库规定其函数对象对于指针同样适用。我们之前曾经介绍过比较两个无关指针将产生未定义的行为（参见 3.5.3 节，第 107 页），然而我们可能会希望通过比较指针的内存地址来 `sort` 指针的 `vector`。直接这么做将产生未定义的行为，因此我们可以使用一个标准库函数对象来实现该目的：

```
vector<string *> nameTable; // 指针的 vector
// 错误：nameTable 中的指针彼此之间没有关系，所以<将产生未定义的行为
sort(nameTable.begin(), nameTable.end(),
    [](string *a, string *b) { return a < b; });
// 正确：标准库规定指针的 less 是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

576 关联容器使用 `less<key_type>` 对元素排序，因此我们可以定义一个指针的 `set` 或者在 `map` 中使用指针作为关键值而无须直接声明 `less`。

14.8.2 节练习

练习 14.42：使用标准库函数对象及适配器定义一条表达式，令其

- (a) 统计大于 1024 的值有多少个。
- (b) 找到第一个不等于 pooh 的字符串。
- (c) 将所有的值乘以 2。

练习 14.43：使用标准库函数对象判断一个给定的 `int` 值是否能被 `int` 容器中的所有元素整除。

14.8.3 可调用对象与 function

C++语言中有几种可调用的对象：函数、函数指针、lambda 表达式（参见 10.3.2 节，第 346 页）、bind 创建的对象（参见 10.3.4 节，第 354 页）以及重载了函数调用运算符的类。

和其他对象一样，可调用的对象也有类型。例如，每个 lambda 有它自己唯一的（未命名）类类型；函数及函数指针的类型则由其返回值类型和实参类型决定，等等。

然而，两个不同类型的可调用对象却可能共享同一种调用形式（call signature）。调用形式指明了调用返回的类型以及传递给调用的实参类型。一种调用形式对应一个函数类型，例如：

```
int(int, int)
```

是一个函数类型，它接受两个 int、返回一个 int。

不同类型可能具有相同的调用形式

对于几个可调用对象共享同一种调用形式的情况，有时我们会希望把它们看成具有相同的类型。例如，考虑下列不同类型的可调用对象：

```
// 普通函数
int add(int i, int j) { return i + j; }
// lambda，其产生一个未命名的函数对象类
auto mod = [] (int i, int j) { return i % j; };
// 函数对象类
struct divide {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
```

上面这些可调用对象分别对其参数执行了不同的算术运算，尽管它们的类型各不相同，但 577 是共享同一种调用形式：

```
int(int, int)
```

我们可能希望使用这些可调用对象构建一个简单的桌面计算器。为了实现这一目的，需要定义一个函数表（function table）用于存储指向这些可调用对象的“指针”。当程序需要执行某个特定的操作时，从表中查找该调用的函数。

在 C++语言中，函数表很容易通过 map 来实现。对于此例来说，我们使用一个表示运算符符号的 string 对象作为关键字；使用实现运算符的函数作为值。当我们需要求给定运算符的值时，先通过运算符索引 map，然后调用找到的那个元素。

假定我们的所有函数都相互独立，并且只处理关于 int 的二元运算，则 map 可以定义成如下的形式：

```
// 构建从运算符到函数指针的映射关系，其中函数接受两个 int、返回一个 int
map<string, int(*)(int,int)> binops;
```

我们可以按照下面的形式将 add 的指针添加到 binops 中：

```
// 正确：add 是一个指向正确类型函数的指针
binops.insert({"+", add}); // {"+", add} 是一个 pair (参见 11.2.3 节，379 页)
```

但是我们不能将 mod 或者 divide 存入 binops：

```
binops.insert({"%", mod});           // 错误: mod 不是一个函数指针
```

问题在于 `mod` 是个 `lambda` 表达式，而每个 `lambda` 有它自己的类类型，该类型与存储在 `binops` 中的值的类型不匹配。

标准库 function 类型

C++ 11

我们可以使用一个名为 `function` 的新的标准库类型解决上述问题，`function` 定义在 `functional` 头文件中，表 14.3 列举出了 `function` 定义的操作。

表 14.3: `function` 的操作

<code>function<T> f;</code>	<code>f</code> 是一个用来存储可调用对象的空 <code>function</code> ，这些可调用对象的调用形式应该与函数类型 <code>T</code> 相同（即 <code>T</code> 是 <code>retType(args)</code> ）
<code>function<T> f(nullptr);</code>	显式地构造一个空 <code>function</code>
<code>function<T> f(obj);</code>	在 <code>f</code> 中存储可调用对象 <code>obj</code> 的副本
<code>f</code>	将 <code>f</code> 作为条件：当 <code>f</code> 含有一个可调用对象时为真；否则为假
<code>f(args)</code>	调用 <code>f</code> 中的对象，参数是 <code>args</code>
定义为 <code>function<T></code> 的成员的类型	
<code>result_type</code>	该 <code>function</code> 类型的可调用对象返回的类型
<code>argument_type</code>	当 <code>T</code> 有一个或两个实参时定义的类型。如果 <code>T</code> 只有一个实参，则 <code>argument_type</code> 是该类型的同义词；如果 <code>T</code> 有两个实参，则 <code>first_argument_type</code> 和 <code>second_argument_type</code> 分别代表两个实参的类型
<code>first_argument_type</code>	
<code>second_argument_type</code>	

`function` 是一个模板，和我们使用过的其他模板一样，当创建一个具体的 `function` 类型时我们必须提供额外的信息。在此例中，所谓额外的信息是指该 `function` 类型能够表示的对象的调用形式。参考其他模板，我们在一对尖括号内指定类型：

```
function<int(int, int)>
```

在这里我们声明了一个 `function` 类型，它可以表示接受两个 `int`、返回一个 `int` 的可调用对象。因此，我们可以用这个新声明的类型表示任意一种桌面计算器用到的类型；

```
function<int(int, int)> f1 = add;           // 函数指针
function<int(int, int)> f2 = divide();       // 函数对象类的对象
function<int(int, int)> f3 = [](int i, int j) // lambda
    { return i * j; };
cout << f1(4,2) << endl;                  // 打印 6
cout << f2(4,2) << endl;                  // 打印 2
cout << f3(4,2) << endl;                  // 打印 8
```

578 使用这个 `function` 类型我们可以重新定义 `map`：

```
// 列举了可调用对象与二元运算符对应关系的表格
// 所有可调用对象都必须接受两个 int、返回一个 int
// 其中的元素可以是函数指针、函数对象或者 lambda
map<string, function<int(int, int)>> binops;
```

我们能把所有可调用对象，包括函数指针、`lambda` 或者函数对象在内，都添加到这个 `map` 中：

```
map<string, function<int(int, int)>> binops = {
    {"+", add},                                // 函数指针
    {"-", std::minus<int>()},                  // 标准库函数对象
    {"/", divide()},                           // 用户定义的函数对象
    {"*", [](int i, int j) { return i * j; }}, // 未命名的 lambda
    {"%", mod} };                            // 命名了的 lambda 对象
```

我们的 map 中包含 5 个元素，尽管其中的可调用对象的类型各不相同，我们仍然能够把所有这些类型都存储在同一个 `function<int (int, int)>` 类型中。

一如往常，当我们索引 map 时将得到关联值的一个引用。如果我们索引 `binops`，将得到 `function` 对象的引用。`function` 类型重载了调用运算符，该运算符接受它自己的实参然后将其传递给存好的可调用对象：

```
binops["+"](10, 5); // 调用 add(10, 5)
binops["-"](10, 5); // 使用 minus<int>对象的调用运算符
binops["/"](10, 5); // 使用 divide 对象的调用运算符
binops["*"](10, 5); // 调用 lambda 函数对象
binops["%"](10, 5); // 调用 lambda 函数对象
```

我们依次调用了 `binops` 中存储的每个操作。在第一个调用中，我们获得的元素存放着一个指向 `add` 函数的指针，因此调用 `binops["+"] (10, 5)` 实际上是使用该指针调用 `add`，并传入 10 和 5。在接下来的调用中，`binops["-"]` 返回一个存放着 `std::minus<int>` 类型对象的 `function`，我们将执行该对象的调用运算符。

重载的函数与 `function`

我们不能（直接）将重载函数的名字存入 `function` 类型的对象中：

```
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data&, const Sales_data&);
map<string, function<int(int, int)>> binops;
binops.insert( {"+", add}); // 错误：哪个 add?
```

解决上述二义性问题的一条途径是存储函数指针（参见 6.7 节，第 221 页）而非函数的名字：

```
int (*fp)(int, int) = add; // 指针所指的 add 是接受两个 int 的版本
binops.insert( {"+", fp}); // 正确：fp 指向一个正确的 add 版本
```

同样，我们也能使用 `lambda` 来消除二义性：

```
// 正确：使用 lambda 来指定我们希望使用的 add 版本
binops.insert( {"+", [](int a, int b) {return add(a, b);}} );
```

`lambda` 内部的函数调用传入了两个 `int`，因此该调用只能匹配接受两个 `int` 的 `add` 版本，而这也正是执行 `lambda` 时真正调用的函数。



新版本标准库中的 `function` 类与旧版本中的 `unary_function` 和 `binary_function` 没有关系，后两个类已经被更通用的 `bind` 函数替代了（参见 10.3.4 节，第 357 页）。

14.8.3 节练习

练习 14.44：编写一个简单的桌面计算器使其能处理二元运算。

14.9 重载、类型转换与运算符

在 7.5.4 节（第 263 页）中我们看到由一个实参调用的非显式构造函数定义了一种隐式的类型转换，这种构造函数将实参类型的对象转换成类类型。我们同样能定义对于类类型的类型转换，通过定义类型转换运算符可以做到这一点。转换构造函数和类型转换运算符共同定义了类类型转换（class-type conversions），这样的转换有时也被称作用户定义的类型转换（user-defined conversions）。

14.9.1 类型转换运算符

类型转换运算符（conversion operator）是类的一种特殊成员函数，它负责将一个类类型的值转换成其他类型。类型转换函数的一般形式如下所示：

```
operator type() const;
```

其中 *type* 表示某种类型。类型转换运算符可以面向任意类型（除了 `void` 之外）进行定义，只要该类型能作为函数的返回类型（参见 6.1 节，第 184 页）。因此，我们不允许转换成数组或者函数类型，但允许转换成指针（包括数组指针及函数指针）或者引用类型。

类型转换运算符既没有显式的返回类型，也没有形参，而且必须定义成类的成员函数。类型转换运算符通常不应该改变待转换对象的内容，因此，类型转换运算符一般被定义成 `const` 成员。



一个类型转换函数必须是类的成员函数；它不能声明返回类型，形参列表也必须为空。类型转换函数通常应该是 `const`。

定义含有类型转换运算符的类

举个例子，我们定义一个比较简单的类，令其表示 0 到 255 之间的一个整数：

```
class SmallInt {
public:
    SmallInt(int i = 0) : val(i)
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

我们的 `SmallInt` 类既定义了向类类型的转换，也定义了从类类型向其他类型的转换。其中，构造函数将算术类型的值转换成 `SmallInt` 对象，而类型转换运算符将 `SmallInt` 对象转换成 `int`：

```
SmallInt si;
```

```
si = 4;           // 首先将 4 隐式地转换成 SmallInt，然后调用 SmallInt::operator=
si + 3;          // 首先将 si 隐式地转换成 int，然后执行整数的加法
```

尽管编译器一次只能执行一个用户定义的类型转换（参见 4.11.2 节，第 144 页），但是隐式的用户定义类型转换可以置于一个标准（内置）类型转换之前或之后（参见 4.11.1 节，第 141 页），并与其一起使用。因此，我们可以将任何算术类型传递给 SmallInt 的构造函数。类似的，我们也能使用类型转换运算符将一个 SmallInt 对象转换成 int，然后再将所得的 int 转换成任何其他算术类型：

```
// 内置类型转换将 double 实参转换成 int
SmallInt si = 3.14;           // 调用 SmallInt(int) 构造函数
// SmallInt 的类型转换运算符将 si 转换成 int
si + 3.14;                   // 内置类型转换将所得的 int 继续转换成 double
```

因为类型转换运算符是隐式执行的，所以无法给这些函数传递实参，当然也就不能在类型转换运算符的定义中使用任何形参。同时，尽管类型转换函数不负责指定返回类型，但实际上每个类型转换函数都会返回一个对应类型的值：

```
class SmallInt;
operator int(SmallInt&);                                // 错误：不是成员函数
class SmallInt {
public:
    int operator int() const;                            // 错误：指定了返回类型
    operator int(int = 0) const;                          // 错误：参数列表不为空
    operator int*() const { return 42; } // 错误：42 不是一个指针
};
```

提示：避免过度使用类型转换函数

和使用重载运算符的经验一样，明智地使用类型转换运算符也能极大地简化类设计者的工作，同时使得使用类更加容易。然而，如果在类类型和转换类型之间不存在明显的映射关系，则这样的类型转换可能具有误导性。

例如，假设某个类表示 Date，我们也许会为它添加一个从 Date 到 int 的转换。然而，类型转换函数的返回值应该是什么？一种可能的解释是，函数返回一个十进制数，依次表示年、月、日，例如，July 30, 1989 可能转换为 int 值 19890730。同时还存在另外一种合理的解释，即类型转换运算符返回的 int 表示的是从某个时间节点（比如 January 1, 1970）开始经过的天数。显然这两种理解都合情合理，毕竟从形式上看它们产生的效果都是越靠后的日期对应的整数值越大，而且两种转换都有实际的用处。

问题在于 Date 类型的对象和 int 类型的值之间不存在明确的一对一映射关系。因此在此例中，不定义该类型转换运算符也许会更好。作为替代的手段，类可以定义一个或多个普通的成员函数以从各种不同形式中提取所需的信息。

类型转换运算符可能产生意外结果

在实践中，类很少提供类型转换运算符。在大多数情况下，如果类型转换自动发生，用户可能会感觉比较意外，而不是感觉受到了帮助。然而这条经验法则存在一种例外情况：对于类来说，定义向 bool 的类型转换还是比较普遍的现象。

在 C++ 标准的早期版本中，如果类想定义一个向 bool 的类型转换，则它常常遇到一个问题：因为 bool 是一种算术类型，所以类类型的对象转换成 bool 后就能被用在任何

需要算术类型的上下文中。这样的类型转换可能引发意想不到的结果，特别是当 `istream` 含有向 `bool` 的类型转换时，下面的代码仍将编译通过：

```
int i = 42;
cin << i; // 如果向 bool 的类型转换不是显式的，则该代码在编译器看来将是合法的！
```

这段程序试图将输出运算符作用于输入流。因为 `istream` 本身并没有定义 `<<`，所以本来代码应该产生错误。然而，该代码能使用 `istream` 的 `bool` 类型转换运算符将 `cin` 转换成 `bool`，而这个 `bool` 值接着会被提升成 `int` 并用作内置的左移运算符的左侧运算对象。这样一来，提升后的 `bool` 值（1 或 0）最终会被左移 42 个位置。这一结果显然与我们的预期大相径庭。

显式的类型转换运算符

C++ 11 为了防止这样的异常情况发生，C++11 新标准引入了显式的类型转换运算符（`explicit conversion operator`）：

```
class SmallInt {
public:
    // 编译器不会自动执行这一类型转换
    explicit operator int() const { return val; }
    // 其他成员与之前的版本一致
};
```

和显式的构造函数（参见 7.5.4 节，第 265 页）一样，编译器（通常）也不会将一个显式的类型转换运算符用于隐式类型转换：

```
SmallInt si = 3;      // 正确：SmallInt 的构造函数不是显式的
si + 3;              // 错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3; // 正确：显式地请求类型转换
```

当类型转换运算符是显式的时，我们也能执行类型转换，不过必须通过显式的强制类型转换才可以。

该规定存在一个例外，即如果表达式被用作条件，则编译器会将显式的类型转换自动应用于它。换句话说，当表达式出现在下列位置时，显式的类型转换将被隐式地执行：

- `if`、`while` 及 `do` 语句的条件部分
- `for` 语句头的条件表达式
- 逻辑非运算符 `(!)`、逻辑或运算符 `(||)`、逻辑与运算符 `(&&)` 的运算对象
- 条件运算符 `(?:)` 的条件表达式。

583 转换为 `bool`

在标准库的早期版本中，IO 类型定义了向 `void*` 的转换规则，以求避免上面提到的问题。在 C++11 新标准下，IO 标准库通过定义一个向 `bool` 的显式类型转换实现同样的目的。

无论我们什么时候在条件中使用流对象，都会使用为 IO 类型定义的 `operator bool`。例如：

```
while (std::cin >> value)
```

`while` 语句的条件执行输入运算符，它负责将数据读入到 `value` 并返回 `cin`。为了对条件求值，`cin` 被 `istream` `operator bool` 类型转换函数隐式地执行了转换。如果 `cin` 的条件状态是 `good`（参见 8.1.2 节，第 280 页），则该函数返回为真；否则该函数返回为假。



向 `bool` 的类型转换通常用在条件部分，因此 `operator bool` 一般定义成 `explicit` 的。

14.9.1 节练习

练习 14.45: 编写类型转换运算符将一个 `Sales_data` 对象分别转换成 `string` 和 `double`，你认为这些运算符的返回值应该是什么？

练习 14.46: 你认为应该为 `Sales_data` 类定义上面两种类型转换运算符吗？应该把它们声明成 `explicit` 的吗？为什么？

练习 14.47: 说明下面这两个类型转换运算符的区别。

```
struct Integral {
    operator const int();
    operator int() const;
};
```

练习 14.48: 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有向 `bool` 的类型转换运算符吗？如果是，解释原因并说明该运算符是否应该是 `explicit` 的；如果不是，也请解释原因。

练习 14.49: 为上一题提到的类定义一个转换目标是 `bool` 的类型转换运算符，先不用在意这么做是否应该。

14.9.2 避免有二义性的类型转换



如果类中包含一个或多个类型转换，则必须确保在类类型和目标类型之间只存在唯一一种转换方式。否则的话，我们编写的代码将很可能会具有二义性。

在两种情况下可能产生多重转换路径。第一种情况是两个类提供相同的类型转换：例如，当 A 类定义了一个接受 B 类对象的转换构造函数，同时 B 类定义了一个转换目标是 A 类的类型转换运算符时，我们就说它们提供了相同的类型转换。

第二种情况是类定义了多个转换规则，而这些转换涉及的类型本身可以通过其他类型转换联系在一起。最典型的例子是算术运算符，对某个给定的类来说，最好只定义最多一个与算术类型有关的转换规则。



通常情况下，不要为类定义相同的类型转换，也不要在类中定义两个及以上以
上转换源或转换目标是算术类型的转换。

584

实参匹配和相同的类型转换

在下面的例子中，我们定义了两种将 B 转换成 A 的方法：一种使用 B 的类型转换运
算符、另一种使用 A 的以 B 为参数的构造函数：

```
// 最好不要在两个类之间构建相同的类型转换
struct B;
struct A {
    A() = default;
    A(const B&);           // 把一个 B 转换成 A
    // 其他数据成员
```

```

};

struct B {
    operator A() const; // 也是把一个 B 转换成 A
    // 其他数据成员
};
A f(const A&);

B b;
A a = f(b); // 二义性错误：含义是 f(B::operator A())
              // 还是 f(A::A(const B&))？

```

因为同时存在两种由 B 获得 A 的方法，所以造成编译器无法判断应该运行哪个类型转换，也就是说，对 f 的调用存在二义性。该调用可以使用以 B 为参数的 A 的构造函数，也可以使用 B 当中把 B 转换成 A 的类型转换运算符。因为这两个函数效果相当、难分伯仲，所以该调用将产生错误。

如果我们确实想执行上述的调用，就不得不显式地调用类型转换运算符或者转换构造函数：

```

A a1 = f(b.operator A()); // 正确：使用 B 的类型转换运算符
A a2 = f(A(b));          // 正确：使用 A 的构造函数

```

值得注意的是，我们无法使用强制类型转换来解决二义性问题，因为强制类型转换本身也面临二义性。

二义性与转换目标为内置类型的多重类型转换

另外如果类定义了一组类型转换，它们的转换源（或者转换目标）类型本身可以通过其他类型转换联系在一起，则同样会产生二义性的问题。最简单也是最困扰我们的例子就是类当中定义了多个参数都是算术类型的构造函数，或者转换目标都是算术类型的类型转换运算符。

例如，在下面的类中包含两个转换构造函数，它们的参数是两种不同的算术类型；同时还包含两个类型转换运算符，它们的转换目标也恰好是两种不同的算术类型：

```

585> struct A {
    A(int = 0);           // 最好不要创建两个转换源都是算术类型的类型转换
    A(double);
    operator int() const; // 最好不要创建两个转换对象都是算术类型的类型转换
    operator double() const;
    // 其他成员
};

void f2(long double);
A a;
f2(a); // 二义性错误：含义是 f(A::operator int())
        // 还是 f(A::operator double())？

long lg;
A a2(lg); // 二义性错误：含义是 A::A(int) 还是 A::A(double)?

```

在对 f2 的调用中，哪个类型转换都无法精确匹配 long double。然而这两个类型转换都可以使用，只要后面再执行一次生成 long double 的标准类型转换即可。因此，在上面的两个类型转换中哪个都不比另一个更好，调用将产生二义性。

当我们试图用 long 初始化 a2 时也遇到了同样问题，哪个构造函数都无法精确匹配 long 类型。它们在使用构造函数前都要求先将实参进行类型转换：

- 先执行 long 到 double 的标准类型转换，再执行 A(double)
- 先执行 long 到 int 的标准类型转换，再执行 A(int)

编译器没办法区分这两种转换序列的好坏，因此该调用将产生二义性。

调用 f2 及初始化 a2 的过程之所以会产生二义性，根本原因是它们所需的标准类型转换级别一致（参见 6.6.1 节，第 219 页）。当我们使用用户定义的类型转换时，如果转换过程包含标准类型转换，则标准类型转换的级别将决定编译器选择最佳匹配的过程：

```
short s = 42;
// 把 short 提升成 int 优于把 short 转换成 double
A a3(s);           // 使用 A::A(int)
```

在此例中，把 short 提升成 int 的操作要优于把 short 转换成 double 的操作，因此编译器将使用 A::A(int) 构造函数构造 a3，其中实参是 s（提升后）的值。



当我们使用两个用户定义的类型转换时，如果转换函数之前或之后存在标准类型转换，则标准类型转换将决定最佳匹配到底是哪个。

提示：类型转换与运算符

< 586

要想正确地设计类的重载运算符、转换构造函数及类型转换函数，必须加倍小心。尤其是当类同时定义了类型转换运算符及重载运算符时特别容易产生二义性。以下的经验规则可能对你有所帮助：

- 不要令两个类执行相同的类型转换：如果 Foo 类有一个接受 Bar 类对象的构造函数，则不要在 Bar 类中再定义转换目标是 Foo 类的类型转换运算符。
- 避免转换目标是内置算术类型的类型转换。特别是当你已经定义了一个转换成算术类型的类型转换时，接下来
 - 不要再定义接受算术类型的重载运算符。如果用户需要使用这样的运算符，则类型转换操作将转换你的类型的对象，然后使用内置的运算符。
 - 不要定义转换到多种算术类型的类型转换。让标准类型转换完成向其他算术类型转换的工作。

一言以蔽之：除了显式地向 bool 类型的转换之外，我们应该尽量避免定义类型转换函数并尽可能地限制那些“显然正确”的非显式构造函数。

重载函数与转换构造函数

当我们调用重载的函数时，从多个类型转换中进行选择将变得更加复杂。如果两个或多个类型转换都提供了同一种可行匹配，则这些类型转换一样好。

举个例子，当几个重载函数的参数分属不同的类类型时，如果这些类恰好定义了同样的转换构造函数，则二义性问题将进一步提升：

```
struct C {
    C(int);
    // 其他成员
```

```

};

struct D {
    D(int);
    // 其他成员
};

void manip(const C&);

void manip(const D&);

manip(10);           // 二义性错误：含义是 manip(C(10)) 还是 manip(D(10))

```

其中 C 和 D 都包含接受 int 的构造函数，两个构造函数各自匹配 manip 的一个版本。因此调用将具有二义性：它的含义可能是把 int 转换成 C，然后调用 manip 的第一个版本；也可能是把 int 转换成 D，然后调用 manip 的第二个版本。

调用者可以显式地构造正确的类型从而消除二义性：

```
manip(C(10));      // 正确：调用 manip(const C&)
```



如果在调用重载函数时我们需要使用构造函数或者强制类型转换来改变实参的类型，则这通常意味着程序的设计存在不足。

重载函数与用户定义的类型转换

当调用重载函数时，如果两个（或多个）用户定义的类型转换都提供了可行匹配，则我们认为这些类型转换一样好。在这个过程中，我们不会考虑任何可能出现的标准类型转换的级别。只有当重载函数能通过同一个类型转换函数得到匹配时，我们才会考虑其中出现的标准类型转换。

例如当我们调用 manip 时，即使其中一个类定义了需要对实参进行标准类型转换的构造函数，这次调用仍然会具有二义性：

```

struct E {
    E(double);
    // 其他成员
};

void manip2(const C&);

void manip2(const E&);

// 二义性错误：两个不同的用户定义的类型转换都能用在此处
manip2(10);      // 含义是 manip2(C(10)) 还是 manip2(E(double(10)))

```

在此例中，C 有一个转换源为 int 的类型转换，E 有一个转换源为 double 的类型转换。对于 manip2(10) 来说，两个 manip2 函数都是可行的：

- manip2(const C&) 是可行的，因为 C 有一个接受 int 的转换构造函数，该构造函数与实参精确匹配。
- manip2(const E&) 是可行的，因为 E 有一个接受 double 的转换构造函数，而且为了使用该函数我们可以利用标准类型转换把 int 转换成所需的类型。

因为调用重载函数所请求的用户定义的类型转换不止一个且彼此不同，所以该调用具有二义性。即使其中一个调用需要额外的标准类型转换而另一个调用能精确匹配，编译器也会将该调用标示为错误。



在调用重载函数时，如果需要额外的标准类型转换，则该转换的级别只有当所有可行函数都请求同一个用户定义的类型转换时才有用。如果所需的用户定义的类型转换不止一个，则该调用具有二义性。

14.9.2 节练习

练习 14.50：在初始化 ex1 和 ex2 的过程中，可能用到哪些类类型的转换序列呢？说明初始化是否正确并解释原因。

```
struct LongDouble {
    LongDouble(double = 0.0);
    operator double();
    operator float();
};

LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;
```

练习 14.51：在调用 calc 的过程中，可能用到哪些类型转换序列呢？说明最佳可行函数是如何被选出来的。

```
void calc(int);
void calc(LongDouble);
double dval;
calc(dval); // 哪个 calc?
```

14.9.3 函数匹配与重载运算符



重载的运算符也是重载的函数。因此，通用的函数匹配规则（参见 6.4 节，第 208 页）同样适用于判断在给定的表达式中到底应该使用内置运算符还是重载的运算符。不过当运算符函数出现在表达式中时，候选函数集的规模要比我们使用调用运算符调用函数时更大。如果 a 是一种类类型，则表达式 a sym b 可能是

```
a.operatorsym(b); // a 有一个 operatorsym 成员函数
operatorsym(a, b); // operatorsym 是一个普通函数
```

和普通函数调用不同，我们不能通过调用的形式来区分当前调用的是成员函数还是非成员函数。

当我们使用重载运算符作用于类类型的运算对象时，候选函数中包含该运算符的普通非成员版本和内置版本。除此之外，如果左侧运算对象是类类型，则定义在该类中的运算符的重载版本也包含在候选函数内。

588

当我们调用一个命名的函数时，具有该名字的成员函数和非成员函数不会彼此重载，这是因为我们用来调用命名函数的语法形式对于成员函数和非成员函数来说是不相同的。当我们通过类类型的对象（或者该对象的指针及引用）进行函数调用时，只考虑该类的成员函数。而当我们在表达式中使用重载的运算符时，无法判断正在使用的是成员函数还是非成员函数，因此二者都应该在考虑的范围内。



表达式中运算符的候选函数集既应该包括成员函数，也应该包括非成员函数。

举个例子，我们为 SmallInt 类定义一个加法运算符：

```
class SmallInt {
    friend
    SmallInt operator+(const SmallInt&, const SmallInt&);

public:
    SmallInt(int = 0); // 转换源为 int 的类型转换
    operator int() const { return val; } // 转换目标为 int 的类型转换

private:
    std::size_t val;
};
```

589 可以使用这个类将两个 SmallInt 对象相加，但如果我们试图执行混合模式的算术运算，就将遇到二义性的问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2; // 使用重载的 operator+
int i = s3 + 0; // 二义性错误
```

第一条加法语句接受两个 SmallInt 值并执行+运算符的重载版本。第二条加法语句具有二义性：因为我们可以把 0 转换成 SmallInt，然后使用 SmallInt 的+；或者把 s3 转换成 int，然后对于两个 int 执行内置的加法运算。



如果我们对同一个类既提供了转换目标是算术类型的类型转换，也提供了重载的运算符，则将会遇到重载运算符与内置运算符的二义性问题。

14.9.3 节练习

练习 14.52：在下面的加法表达式中分别选用了哪个 operator+？列出候选函数、可行函数及为每个可行函数的实参执行的类型转换：

```
struct LongDouble {
    // 用于演示的成员 operator+；在通常情况下+是个非成员
    LongDouble operator+(const SmallInt&);
    // 其他成员与 14.9.2 节（第 521 页）一致
};

LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
```

练习 14.53：假设我们已经定义了如第 522 页所示的 SmallInt，判断下面的加法表达式是否合法。如果合法，使用了哪个加法运算符？如果不合法，应该怎样修改代码才能使其合法？

```
SmallInt s1;
double d = s1 + 3.14;
```

小结

590

一个重载的运算符必须是某个类的成员或者至少拥有一个类类型的运算对象。重载运算符的运算对象数量、结合律、优先级与对应的用于内置类型的运算符完全一致。当运算符被定义为类的成员时，类对象的隐式 `this` 指针绑定到第一个运算对象。赋值、下标、函数调用和箭头运算符必须作为类的成员。

如果类重载了函数调用运算符 `operator()`，则该类的对象被称作“函数对象”。这样的对象常用在标准函数中。`lambda` 表达式是一种简便的定义函数对象类的方式。

在类中可以定义转换源或转换目的是该类型本身的类型转换，这样的类型转换将自动执行。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的类型转换；而非显式的类型转换运算符则定义了从类类型到其他类型的转换。

术语表

调用形式 (call signature) 表示一个可调用对象的接口。在调用形式中包括返回类型以及一个实参类型列表，该列表在一对圆括号内，实参类型之间以逗号分隔。

类类型转换 (class-type conversion) 包括由构造函数定义的从其他类型到类类型的转换以及由类型转换运算符定义的从类类型到其他类型的转换。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的转换；而类型转换运算符则定义了从类类型到某个指定类型的转换。

类型转换运算符 (conversion operator) 是类的成员函数，定义了从类类型到其他类型的转换。类型转换运算符必须是它要转换的类的成员，并且通常被定义为常量成员。这类运算符既没有返回类型，也不接受参数。它们返回一个可变为转换运算符类型的值，也就是说，`operator int` 返回一个 `int`，`operator string` 返回一个 `string`，依此类推。

显式的类型转换运算符 (explicit conversion operator) 由关键字 `explicit` 限定的类

型转换运算符。这样的运算符用于条件中的隐式类型转换。

函数对象 (function object) 定义了重载调用运算符的对象。在需要使用函数的地方都能使用函数对象。

函数表 (function table) 形如 `map` 或 `vector` 的容器，容器中所存的值可以被调用。

函数模板 (function template) 能够表示任意可调用类型的标准库模板。

重载的运算符 (overloaded operator) 重定义了某种内置运算符的含义的函数。重载的运算符函数含有关键字 `operator`，之后是要定义的符号。重载的运算符必须含有至少一个类类型的运算对象。重载运算符的优先级、结合律、运算对象数量都与其内置版本一致。

用户定义的类型转换 (user-defined conversion) 类类型转换的同义词。

第 15 章

面向对象程序设计

内容

15.1 OOP: 概述	526
15.2 定义基类和派生类	527
15.3 虚函数	536
15.4 抽象基类	540
15.5 访问控制与继承	542
15.6 继承中的类作用域	547
15.7 构造函数与拷贝控制	551
15.8 容器与继承	558
15.9 文本查询程序再探	562
小结	575
术语表	575

面向对象程序设计基于三个基本概念：数据抽象、继承和动态绑定。第 7 章已经介绍了数据抽象的知识，本章将介绍继承和动态绑定。

继承和动态绑定对程序的编写有两方面的影响：一是我们可以更容易地定义与其他类相似但不完全相同的新类；二是在使用这些彼此相似的类编写程序时，我们可以在一定程度上忽略掉它们的区别。

592

在很多程序中都存在着一些相互关联但是有细微差别的概念。例如，书店中不同书籍的定价策略可能不同：有的书籍按原价销售，有的则打折销售。有时，我们给那些购买书籍超过一定数量的顾客打折；另一些时候，则只对前多少本销售的书籍打折，之后就调回原价，等等。面向对象的程序设计（OOP）适用于这类应用。



15.1 OOP：概述

面向对象程序设计（object-oriented programming）的核心思想是数据抽象、继承和动态绑定。通过使用数据抽象，我们可以将类的接口与实现分离（见第 7 章）；使用继承，可以定义相似的类型并对其相似关系建模；使用动态绑定，可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。

继承

通过继承（inheritance）联系在一起的类构成一种层次关系。通常在层次关系的根部有一个基类（base class），其他类则直接或间接地从基类继承而来，这些继承得到的类称为派生类（derived class）。基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。

为了对之前提到的不同定价策略建模，我们首先定义一个名为 `Quote` 的类，并将它作为层次关系中的基类。`Quote` 的对象表示按原价销售的书籍。`Quote` 派生出另一个名为 `Bulk_quote` 的类，它表示可以打折销售的书籍。

这些类将包含下面的两个成员函数：

- `isbn()`，返回书籍的 ISBN 编号。该操作不涉及派生类的特殊性，因此只定义在 `Quote` 类中。
- `net_price(size_t)`，返回书籍的实际销售价格，前提是用户购买该书的数量达到一定标准。这个操作显然是类型相关的，`Quote` 和 `Bulk_quote` 都应该包含该函数。

在 C++ 语言中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待。对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明成虚函数（virtual function）。因此，我们可以将 `Quote` 类编写成：

```
class Quote {
public:
    std::string isbn() const;
    virtual double net_price(std::size_t n) const;
};
```

593

派生类必须通过使用类派生列表（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有访问说明符：

```
class Bulk_quote : public Quote {           // Bulk_quote 继承了 Quote
public:
    double net_price(std::size_t) const override;
};
```

因为 `Bulk_quote` 在它的派生列表中使用了 `public` 关键字，因此我们完全可以把

`Bulk_quote` 的对象当成 `Quote` 的对象来使用。

派生类必须在其内部对所有重新定义的虚函数进行声明。派生类可以在这样的函数之前加上 `virtual` 关键字，但是并不是非得这么做。出于 15.3 节（第 538 页）将要解释的原因，C++11 新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，具体措施是在该函数的形参列表之后增加一个 `override` 关键字。

动态绑定

通过使用 **动态绑定**（dynamic binding），我们能用同一段代码分别处理 `Quote` 和 `Bulk_quote` 的对象。例如，当要购买的书籍和购买的数量都已知时，下面的函数负责打印总的费用：

```
// 计算并打印销售给定数量的某种书籍所得的费用
double print_total(ostream &os,
                    const Quote &item, size_t n)
{
    // 根据传入 item 形参的对象类型调用 Quote::net_price
    // 或者 Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn()      // 调用 Quote::isbn
       << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

该函数非常简单：它返回调用 `net_price()` 的结果，并将该结果连同调用 `isbn()` 的结果一起打印出来。

关于上面的函数有两个有意思的结论：因为函数 `print_total` 的 `item` 形参是基类 `Quote` 的一个引用，所以出于 15.2.3 节（第 534 页）将要解释的原因，我们既能使用基类 `Quote` 的对象调用该函数，也能使用派生类 `Bulk_quote` 的对象调用它；又因为 `print_total` 是使用引用类型调用 `net_price` 函数的，所以出于 15.2.1 节（第 528 页）将要解释的原因，实际传入 `print_total` 的对象类型将决定到底执行 `net_price` 的哪个版本：

```
// basic 的类型是 Quote; bulk 的类型是 Bulk_quote
print_total(cout, basic, 20);           // 调用 Quote 的 net_price
print_total(cout, bulk, 20);            // 调用 Bulk_quote 的 net_price
```

第一条调用句将 `Quote` 对象传入 `print_total`，因此当 `print_total` 调用 `net_price` 时，执行的是 `Quote` 的版本；在第二条调用语句中，实参的类型是 `Bulk_quote`，因此执行的是 `Bulk_quote` 的版本（计算打折信息）。因为在上述过程中函数的运行版本由实参决定，即在运行时选择函数的版本，所以动态绑定有时又被称为运行时绑定（run-time binding）。

< 594



在 C++ 语言中，当我们使用基类的引用（或指针）调用一个虚函数时将发生动态绑定。

15.2 定义基类和派生类

定义基类和派生类的方式在很多方面都与我们已知的定义其他类的方式类似，但是也有一些不同之处。本节将介绍在定义有继承关系的类时可能用到的基本特性。



15.2.1 定义基类

我们首先完成 `Quote` 类的定义：

```
class Quote {
public:
    Quote() = default;           // 关于=default 请参见 7.1.4 节（第 237 页）
    Quote(const std::string &book, double sales_price):
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // 返回给定数量的书籍的销售总额
    // 派生类负责改写并使用不同的折扣计算算法
    virtual double net_price(std::size_t n) const
    { return n * price; }
    virtual ~Quote() = default;   // 对析构函数进行动态绑定
private:
    std::string bookNo;          // 书籍的 ISBN 编号
protected:
    double price = 0.0;          // 代表普通状态下不打折的价格
};
```

对于上面这个类来说，新增的部分是在 `net_price` 函数和析构函数之前增加的 `virtual` 关键字以及最后的 `protected` 访问说明符。我们将在 15.7.1 节（第 552 页）详细介绍虚析构函数的知识，现在只需记住作为继承关系中根节点的类通常都会定义一个虚析构函数。



基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此。

成员函数与继承

派生类可以继承其基类的成员，然而当遇到如 `net_price` 这样与类型相关的操作时，

595 派生类必须对其重新定义。换句话说，派生类需要对这些操作提供自己的新定义以覆盖（`override`）从基类继承而来的旧定义。

在 C++ 语言中，基类必须将它的两种成员函数区分开来：一种是基类希望其派生类进行覆盖的函数；另一种是基类希望派生类直接继承而不要改变的函数。对于前者，基类通常将其定义为虚函数（`virtual`）。当我们使用指针或引用调用虚函数时，该调用将被动态绑定。根据引用或指针所绑定的对象类型不同，该调用可能执行基类的版本，也可能执行某个派生类的版本。

基类通过在其成员函数的声明语句之前加上关键字 `virtual` 使得该函数执行动态绑定。任何构造函数之外的非静态函数（参见 7.6 节，第 268 页）都可以是虚函数。关键字 `virtual` 只能出现在类内部的声明语句之前而不能用于类外部的函数定义。如果基类把一个函数声明成虚函数，则该函数在派生类中隐式地也是虚函数。我们将在 15.3 节（第 536 页）介绍更多关于虚函数的知识。

成员函数如果没被声明为虚函数，则其解析过程发生在编译时而非运行时。对于 `isbn` 成员来说这正是我们希望看到的结果。`isbn` 函数的执行与派生类的细节无关，不管作用于 `Quote` 对象还是 `Bulk_quote` 对象，`isbn` 函数的行为都一样。在我们的继承层次关系中只有一个 `isbn` 函数，因此也就不存在调用 `isbn()` 时到底执行哪个版本的疑问。

访问控制与继承

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员。和其他使用基类的代码一样，派生类能访问公有成员，而不能访问私有成员。不过在某些时候基类中还有这样一种成员，基类希望它的派生类有权访问该成员，同时禁止其他用户访问。我们用受保护的（protected）访问运算符说明这样的成员。

我们的 Quote 类希望它的派生类定义各自的 net_price 函数，因此派生类需要访问 Quote 的 price 成员。此时我们将 price 定义成受保护的。与之相反，派生类访问 bookNo 成员的方式与其他用户是一样的，都是通过调用 isbn 函数，因此 bookNo 被定义成私有的，即使是 Quote 派生出来的类也不能直接访问它。我们将在 15.5 节（第 542 页）介绍更多关于受保护成员的知识。

15.2.1 节练习

练习 15.1：什么是虚成员？

练习 15.2：protected 访问说明符与 private 有何区别？

练习 15.3：定义你自己的 Quote 类和 print_total 函数。

15.2.2 定义派生类

596

派生类必须通过使用类派生列表（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有以下三种访问说明符中的一个：public、protected 或者 private。



派生类必须将其继承而来的成员函数中需要覆盖的那些重新声明，因此，我们的 Bulk_quote 类必须包含一个 net_price 成员：

```
class Bulk_quote : public Quote {           // Bulk_quote 继承自 Quote
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    // 覆盖基类的函数版本以实现基于大量购买的折扣政策
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0;                  // 适用折扣政策的最低购买量
    double discount = 0.0;                   // 以小数表示的折扣额
};
```

我们的 Bulk_quote 类从它的基类 Quote 那里继承了 isbn 函数和 bookNo、price 等数据成员。此外，它还定义了 net_price 的新版本，同时拥有两个新增加的数据成员 min_qty 和 discount。这两个成员分别用于说明享受折扣所需购买的最低数量以及一旦该数量达到之后具体的折扣信息。

我们将在 15.5 节（第 543 页）详细介绍派生列表中用到的访问说明符。现在，我们只需知道访问说明符的作用是控制派生类从基类继承而来的成员是否对派生类的用户可见。

如果一个派生是公有的，则基类的公有成员也是派生类接口的组成部分。此外，我们能将公有派生类型的对象绑定到基类的引用或指针上。因为我们在派生列表中使用了

`public`, 所以 `Bulk_quote` 的接口隐式地包含 `isbn` 函数, 同时在任何需要 `Quote` 的引用或指针的地方我们都能使用 `Bulk_quote` 的对象。

大多数类都只继承自一个类, 这种形式的继承被称作“单继承”, 它构成了本章的主题。关于派生列表中含有多个基类的情况将在 18.3 节(第 710 页)中介绍。

派生类中的虚函数

派生类经常(但不总是)覆盖它继承的虚函数。如果派生类没有覆盖其基类中的某个虚函数, 则该虚函数的行为类似于其他的普通成员, 派生类会直接继承其在基类中的版本。

C++ 11 派生类可以在它覆盖的函数前使用 `virtual` 关键字, 但不是非得这么做。我们将在 15.3 节(第 538 页)介绍其原因, C++11 新标准允许派生类显式地注明它使用某个成员函数覆盖了它继承的虚函数。具体做法是在形参列表后面、或者在 `const` 成员函数(参见 7.1.2 节, 第 231 页)的 `const` 关键字后面、或者在引用成员函数(参见 13.6.3 节, 第 483 页)的引用限定符后面添加一个关键字 `override`。

597 派生类对象及派生类向基类的类型转换

一个派生类对象包含多个组成部分: 一个含有派生类自己定义的(非静态)成员的子对象, 以及一个与该派生类继承的基类对应的子对象, 如果有多个基类, 那么这样的子对象也有多个。因此, 一个 `Bulk_quote` 对象将包含四个数据元素: 它从 `Quote` 继承而来的 `bookNo` 和 `price` 数据成员, 以及 `Bulk_quote` 自己定义的 `min_qty` 和 `discount` 成员。

C++ 标准并没有明确规定派生类的对象在内存中如何分布, 但是我们可以认为 `Bulk_quote` 的对象包含如图 15.1 所示的两部分。



在一个对象中, 继承自基类的部分和派生类自定义的部分不一定是连续存储的。图 15.1 只是表示类工作机理的概念模型, 而非物理模型。

图 15.1: Bulk_quote 对象的概念结构

因为在派生类对象中含有与其基类对应的组成部分, 所以我们能把派生类的对象当成基类对象来使用, 而且我们也能将基类的指针或引用绑定到派生类对象中的基类部分上。

```

Quote item;           // 基类对象
Bulk_quote bulk;     // 派生类对象
Quote *p = &item;      // p 指向 Quote 对象
p = &bulk;            // p 指向 bulk 的 Quote 部分
Quote &r = bulk;      // r 绑定到 bulk 的 Quote 部分

```

这种转换通常称为派生类到基类的(derived-to-base)类型转换。和其他类型转换一样, 编译器会隐式地执行派生类到基类的转换(参见 4.11 节, 第 141 页)。

这种隐式特性意味着我们可以把派生类对象或者派生类对象的引用用在需要基类引

用的地方；同样的，我们也可以把派生类对象的指针用在需要基类指针的地方。



在派生类对象中含有与其基类对应的组成部分，这一事实是继承的关键所在。

派生类构造函数

<598

尽管在派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员。和其他创建了基类对象的代码一样，派生类也必须使用基类的构造函数来初始化它的基类部分。



每个类控制它自己的成员初始化过程。

派生类对象的基类部分与派生类对象自己的数据成员都是在构造函数的初始化阶段（参见 7.5.1 节，第 258 页）执行初始化操作的。类似于我们初始化成员的过程，派生类构造函数同样是通过构造函数初始化列表来将实参传递给基类构造函数的。例如，接受四个参数的 Bulk_quote 构造函数如下所示：

```
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) {}
// 与之前一致
};
```

该函数将它的前两个参数（分别表示 ISBN 和价格）传递给 Quote 的构造函数，由 Quote 的构造函数负责初始化 Bulk_quote 的基类部分（即 bookNo 成员和 price 成员）。当（空的）Quote 构造函数体结束后，我们构建的对象的基类部分也就完成初始化了。接下来初始化由派生类直接定义的 min_qty 成员和 discount 成员。最后运行 Bulk_quote 构造函数的（空的）函数体。

除非我们特别指出，否则派生类对象的基类部分会像数据成员一样执行默认初始化。如果想使用其他的基类构造函数，我们需要以类名加圆括号内的实参列表的形式为构造函数提供初始值。这些实参将帮助编译器决定到底应该选用哪个构造函数来初始化派生类对象的基类部分。



首先初始化基类的部分，然后按照声明的顺序依次初始化派生类的成员。

派生类使用基类的成员

派生类可以访问基类的公有成员和受保护成员：

```
// 如果达到了购买书籍的某个最低限量值，就可以享受折扣价格了
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

该函数产生一个打折后的价格：如果给定的数量超过了 min_qty，则将 discount（一 <599

个小于 1 大于 0 的数) 作用于 price。

我们将在 15.6 节 (第 547 页) 进一步讨论作用域, 目前只需要了解派生类的作用域嵌套在基类的作用域之内。因此, 对于派生类的一个成员来说, 它使用派生类成员 (例如 min_qty 和 discount) 的方式与使用基类成员 (例如 price) 的方式没什么不同。

关键概念: 遵循基类的接口

必须明确一点: 每个类负责定义各自的接口。要想与类的对象交互必须使用该类的接口, 即使这个对象是派生类的基类部分也是如此。

因此, 派生类对象不能直接初始化基类的成员。尽管从语法上来说我们可以在派生类构造函数体内给它的公有或受保护的基类成员赋值, 但是最好不要这么做。和使用基类的其他场合一样, 派生类应该遵循基类的接口, 并且通过调用基类的构造函数来初始化那些从基类中继承而来的成员。

继承与静态成员

如果基类定义了一个静态成员 (参见 7.6 节, 第 268 页), 则在整个继承体系中只存在该成员的唯一定义。不论从基类中派生出来多少个派生类, 对于每个静态成员来说都只存在唯一的实例。

```
class Base {
public:
    static void statmem();
};

class Derived : public Base {
    void f(const Derived&);
};
```

静态成员遵循通用的访问控制规则, 如果基类中的成员是 private 的, 则派生类无权访问它。假设某静态成员是可访问的, 则我们既能通过基类使用它也能通过派生类使用它:

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();           // 正确: Base 定义了 statmem
    Derived::statmem();        // 正确: Derived 继承了 statmem
    // 正确: 派生类的对象能访问基类的静态成员
    derived_obj.statmem();     // 通过 Derived 对象访问
    statmem();                 // 通过 this 对象访问
}
```

600> 派生类的声明

派生类的声明与其他类差别不大 (参见 7.3.3 节, 第 250 页), 声明中包含类名但是不包含它的派生列表:

```
class Bulk_quote : public Quote; // 错误: 派生列表不能出现在这里
class Bulk_quote;             // 正确: 声明派生类的正确方式
```

一条声明语句的目的是令程序知晓某个名字的存在以及该名字表示一个什么样的实体, 如一个类、一个函数或一个变量等。派生列表以及与定义有关的其他细节必须与类的主体一起出现。

被用作基类的类

如果我们想将某个类用作基类，则该类必须已经定义而非仅仅声明：

```
class Quote; // 声明但未定义
// 错误: Quote 必须被定义
class Bulk_quote : public Quote { ... };
```

这一规定的原因显而易见：派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类当然要知道它们是什么。因此该规定还有一层隐含的意思，即一个类不能派生它本身。

一个类是基类，同时它也可以是一个派生类：

```
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /* ... */ };
```

在这个继承关系中，Base 是 D1 的直接基类 (direct base)，同时是 D2 的间接基类 (indirect base)。直接基类出现在派生列表中，而间接基类由派生类通过其直接基类继承而来。

每个类都会继承直接基类的所有成员。对于一个最终的派生类来说，它会继承其直接基类的成员；该直接基类的成员又含有其基类的成员；依此类推直至继承链的顶端。因此，最终的派生类将包含它的直接基类的子对象以及每个间接基类的子对象。

防止继承的发生

有时我们会定义这样一种类，我们不希望其他类继承它，或者不想考虑它是否适合作为一个基类。为了实现这一目的，C++11 新标准提供了一种防止继承发生的方法，即在类名后跟一个关键字 final：

```
class NoDerived final { /* */ }; // NoDerived 不能作为基类
class Base { /* */ };
// Last 是 final 的；我们不能继承 Last
class Last final : Base { /* */ }; // Last 不能作为基类
class Bad : NoDerived { /* */ }; // 错误: NoDerived 是 final 的
class Bad2 : Last { /* */ }; // 错误: Last 是 final 的
```

15.2.2 节练习

C++
11

601

练习 15.4：下面哪条声明语句是不正确的？请解释原因。

- class Base { ... };
- (a) class Derived : public Derived { ... };
- (b) class Derived : private Base { ... };
- (c) class Derived : public Base;

练习 15.5：定义你自己的 Bulk_quote 类。

练习 15.6：将 Quote 和 Bulk_quote 的对象传给 15.2.1 节（第 529 页）练习中的 print_total 函数，检查该函数是否正确。

练习 15.7：定义一个类使其实现一种数量受限的折扣策略，具体策略是：当购买书籍的数量不超过一个给定的限量时享受折扣，如果购买量一旦超过了限量，则超出的部分将以原价销售。



15.2.3 类型转换与继承



理解基类和派生类之间的类型转换是理解 C++ 语言面向对象编程的关键所在。

通常情况下，如果我们想把引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致（参见 2.3.1 节，第 46 页和 2.3.2 节，第 47 页），或者对象的类型含有一个可接受的 `const` 类型转换规则（参见 4.11.2 节，第 144 页）。存在继承关系的类是一个重要的例外：我们可以将基类的指针或引用绑定到派生类对象上。例如，我们可以用 `Quote&` 指向一个 `Bulk_quote` 对象，也可以把一个 `Bulk_quote` 对象的地址赋给一个 `Quote*`。

可以将基类的指针或引用绑定到派生类对象上有一层极为重要的含义：当使用基类的引用（或指针）时，实际上我们并不清楚该引用（或指针）所绑定对象的真实类型。该对象可能是基类的对象，也可能是派生类的对象。



和内置指针一样，智能指针类（参见 12.1 节，第 400 页）也支持派生类向基类的类型转换，这意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针内。



静态类型与动态类型

当我们使用存在继承关系的类型时，必须将一个变量或其他表达式的 **静态类型**（static type）与该表达式表示对象的 **动态类型**（dynamic type）区分开来。表达式的静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型；动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直到运行时才可知。

602 >

例如，当 `print_total` 调用 `net_price` 时（参见 15.1 节，第 527 页）：

```
double ret = item.net_price(n);
```

我们知道 `item` 的静态类型是 `Quote&`，它的动态类型则依赖于 `item` 绑定的实参，动态类型直到在运行时调用该函数时才会知道。如果我们传递一个 `Bulk_quote` 对象给 `print_total`，则 `item` 的静态类型将与它的动态类型不一致。如前所述，`item` 的静态类型是 `Quote&`，而在此例中它的动态类型则是 `Bulk_quote`。

如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致。例如，`Quote` 类型的变量永远是一个 `Quote` 对象，我们无论如何都不能改变该变量对应的对象的类型。



基类的指针或引用的静态类型可能与其动态类型不一致，读者一定要理解其中的原因。

不存在从基类向派生类的隐式类型转换……

之所以存在派生类向基类的类型转换是因为每个派生类对象都包含一个基类部分，而基类的引用或指针可以绑定到该基类部分上。一个基类的对象既可以以独立的形式存在，也可以作为派生类对象的一部分存在。如果基类对象不是派生类对象的一部分，则它只含有基类定义的成员，而不含有派生类定义的成员。

因为一个基类的对象可能是派生类对象的一部分，也可能不是，所以不存在从基类向派生类的自动类型转换：

```
Quote base;
Bulk_quote* bulkP = &base;           // 错误：不能将基类转换成派生类
Bulk_quote& bulkRef = base;         // 错误：不能将基类转换成派生类
```

如果上述赋值是合法的，则我们有可能会使用 bulkP 或 bulkRef 访问 base 中本不存在的成员。

除此之外还有一种情况显得有点特别，即使一个基类指针或引用绑定在一个派生类对象上，我们也不能执行从基类向派生类的转换：

```
Bulk_quote bulk;
Quote *itemP = &bulk;                // 正确：动态类型是 Bulk_quote
Bulk_quote *bulkP = itemP;           // 错误：不能将基类转换成派生类
```

编译器在编译时无法确定某个特定的转换在运行时是否安全，这是因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法。如果在基类中含有一个或多个虚函数，我们可以使用 `dynamic_cast`（参见 19.2.1 节，第 730 页）请求一个类型转换，该转换的安全检查将在运行时执行。同样，如果我们已知某个基类向派生类的转换是安全的，则我们可以使用 `static_cast`（参见 4.11.3 节，第 144 页）来强制覆盖掉编译器的检查工作。

……在对象之间不存在类型转换



派生类向基类的自动类型转换只对指针或引用类型有效，在派生类类型和基类类型之间不存在这样的转换。很多时候，我们确实希望将派生类对象转换成它的基类类型，但是这种转换的实际发生过程往往与我们期望的有所差别。

603

请注意，当我们初始化或赋值一个类类型的对象时，实际上是在调用某个函数。当执行初始化时，我们调用构造函数（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页）；而当执行赋值操作时，我们调用赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）。这些成员通常都包含一个参数，该参数的类型是类类型的 `const` 版本的引用。

因为这些成员接受引用作为参数，所以派生类向基类的转换允许我们给基类的拷贝/移动操作传递一个派生类的对象。这些操作不是虚函数。当我们给基类的构造函数传递一个派生类对象时，实际运行的构造函数是基类中定义的那个，显然该构造函数只能处理基类自己的成员。类似的，如果我们将一个派生类对象赋值给一个基类对象，则实际运行的赋值运算符也是基类中定义的那个，该运算符同样只能处理基类自己的成员。

例如，我们的书店类使用了合成版本的拷贝和赋值操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。关于拷贝控制与继承的知识将在 15.7.2 节（第 552 页）做更详细的介绍，现在我们只需要知道合成版本会像其他类一样逐成员地执行拷贝或赋值操作：

```
Bulk_quote bulk;                  // 派生类对象
Quote item(bulk);                 // 使用 Quote::Quote(const Quote&) 构造函数
item = bulk;                      // 调用 Quote::operator=(const Quote&)
```

当构造 `item` 时，运行 `Quote` 的拷贝构造函数。该函数只能处理 `bookNo` 和 `price` 两个成员，它负责拷贝 `bulk` 中 `Quote` 部分的成员，同时忽略掉 `bulk` 中 `Bulk_quote` 部分的成员。类似的，对于将 `bulk` 赋值给 `item` 的操作来说，只有 `bulk` 中 `Quote` 部分的成员被赋值给 `item`。

因为在上述过程中会忽略 `Bulk_quote` 部分，所以我们可以说 `bulk` 的 `Bulk_quote` 部分被切掉（sliced down）了。



当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分将被忽略掉。

15.2.3 节练习

练习 15.8：给出静态类型和动态类型的定义。

练习 15.9：在什么情况下表达式的静态类型可能与动态类型不同？请给出三个静态类型与动态类型不同的例子。

练习 15.10：回忆我们在 8.1 节（第 279 页）进行的讨论，解释第 284 页中将 ifstream 传递给 Sales_data 的 read 函数的程序是如何工作的。

关键概念：存在继承关系的类型之间的转换规则

要想理解在具有继承关系的类之间发生的类型转换，有三点非常重要：

- 从派生类向基类的类型转换只对指针或引用类型有效。
- 基类向派生类不存在隐式类型转换。
- 和任何其他成员一样，派生类向基类的类型转换也可能会由于访问受限而变得不可行。我们将在 15.5 节（第 544 页）详细介绍可访问性的问题。

尽管自动类型转换只对指针或引用类型有效，但是继承体系中的大多数类仍然（显式或隐式地）定义了拷贝控制成员（参见第 13 章）。因此，我们通常能够将一个派生类对象拷贝、移动或赋值给一个基类对象。不过需要注意的是，这种操作只处理派生类对象的基类部分。



15.3 虚函数

如前所述，在 C++ 语言中，当我们使用基类的引用或指针调用一个虚成员函数时会执行动态绑定（参见 15.1 节，第 527 页）。因为我们直到运行时才能知道到底调用了哪个版本的虚函数，所以所有虚函数都必须有定义。通常情况下，如果我们不使用某个函数，则无须为该函数提供定义（参见 6.1.2 节，第 186 页）。但是我们必须为每一个虚函数都提供定义，而不管它是否被用到了，这是因为连编译器也无法确定到底会使用哪个虚函数。

对虚函数的调用可能在运行时才被解析

当某个虚函数通过指针或引用调用时，编译器产生的代码直到运行时才能确定应该调用哪个版本的函数。被调用的函数是与绑定到指针或引用上的对象的动态类型相匹配的那个。

举个例子，考虑 15.1 节（第 527 页）的 print_total 函数，该函数通过其名为 item 的参数来进一步调用 net_price，其中 item 的类型是 Quote&。因为 item 是引用而且 net_price 是虚函数，所以我们到底调用 net_price 的哪个版本完全依赖于运行时绑定到 item 的实参的实际（动态）类型：

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);           // 调用 Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
```

```
print_total(cout, derived, 10);           // 调用 Bulk_quote::net_price
```

在第一条调用语句中，item 绑定到 Quote 类型的对象上，因此当 print_total 调用 net_price 时，运行在 Quote 中定义的版本。在第二条调用语句中，item 绑定到 Bulk_quote 类型的对象上，因此 print_total 调用 Bulk_quote 定义的 net_price。605

必须要搞清楚的一点是，动态绑定只有当我们通过指针或引用调用虚函数时才会发生。

```
base = derived;                         // 把 derived 的 Quote 部分拷贝给 base
base.net_price(20);                    // 调用 Quote::net_price
```

当我们通过一个具有普通类型（非引用非指针）的表达式调用虚函数时，在编译时就会将调用的版本确定下来。例如，如果我们使用 base 调用 net_price，则应该运行 net_price 的哪个版本是显而易见的。我们可以改变 base 表示的对象的值（即内容），但是不会改变该对象的类型。因此，在编译时该调用就会被解析成 Quote 的 net_price。

关键概念：C++的多态性

OOP 的核心思想是多态性（polymorphism）。多态性这个词源自希腊语，其含义是“多种形式”。我们把具有继承关系的多个类型称为多态类型，因为我们能使用这些类型的“多种形式”而无须在意它们的差异。引用或指针的静态类型与动态类型不同这一事实正是 C++ 语言支持多态性的根本所在。

当我们使用基类的引用或指针调用基类中定义的一个函数时，我们并不知道该函数真正作用的对象是什么类型，因为它可能是一个基类的对象也可能是一个派生类的对象。如果该函数是虚函数，则直到运行时才会决定到底执行哪个版本，判断的依据是引用或指针所绑定的对象的真实类型。

另一方面，对非虚函数的调用在编译时进行绑定。类似的，通过对象进行的函数（虚函数或非虚函数）调用也在编译时绑定。对象的类型是确定不变的，我们无论如何都不可能令对象的动态类型与静态类型不一致。因此，通过对象进行的函数调用将在编译时绑定到该对象所属类中的函数版本上。



当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同。

派生类中的虚函数

当我们在派生类中覆盖了某个虚函数时，可以再一次使用 virtual 关键字指出该函数的性质。然而这么做并非必须，因为一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数。

一个派生类的函数如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致。

同样，派生类中虚函数的返回类型也必须与基类函数匹配。该规则存在一个例外，当类的虚函数返回类型是类本身的指针或引用时，上述规则无效。也就是说，如果 D 由 B 派生得到，则基类的虚函数可以返回 B* 而派生类的对应函数可以返回 D*，只不过这样的返回类型要求从 D 到 B 的类型转换是可访问的。15.5 节（第 544 页）将介绍如何确定一个基类的可访问性，在 15.8.1 节（第 561 页）中我们将看到这种虚函数的一个实际例子。606



基类中的虚函数在派生类中隐含地也是一个虚函数。当派生类覆盖了某个虚函数时，该函数在基类中的形参必须与派生类中的形参严格匹配。

final 和 override 说明符

如我们将要在 15.6 节（第 550 页）介绍的，派生类如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，这仍然是合法的行为。编译器将认为新定义的这个函数与基类中原有的函数是相互独立的。这时，派生类的函数并没有覆盖掉基类中的版本。就实际的编程习惯而言，这种声明往往意味着发生了错误，因为我们可能原本希望派生类能覆盖掉基类中的虚函数，但是一不小心把形参列表弄错了。

C++
11

要想调试并发现这样的错误显然非常困难。在 C++11 新标准中我们可以使用 `override` 关键字来说明派生类中的虚函数。这么做的好处是在使得程序员的意图更加清晰的同时让编译器可以为我们发现一些错误，后者在编程实践中显得更加重要。如果我们使用 `override` 标记了某个函数，但该函数并没有覆盖已存在的虚函数，此时编译器将报错：

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};

struct D1 : B {
    void f1(int) const override;           // 正确: f1 与基类中的 f1 匹配
    void f2(int) override;                 // 错误: B 没有形如 f2(int) 的函数
    void f3() override;                  // 错误: f3 不是虚函数
    void f4() override;                  // 错误: B 没有名为 f4 的函数
};
```

在 D1 中，`f1` 的 `override` 说明符是正确的，因为基类和派生类中的 `f1` 都是 `const` 成员，并且它们都接受一个 `int` 返回 `void`，所以 D1 中的 `f1` 正确地覆盖了它从 B 中继承而来的虚函数。

D1 中 `f2` 的声明与 B 中 `f2` 的声明不匹配，显然 B 中定义的 `f2` 不接受任何参数而 D1 的 `f2` 接受一个 `int`。因为这两个声明不匹配，所以 D1 的 `f2` 不能覆盖 B 的 `f2`，它是一个新函数，仅仅是名字恰好与原来的函数一样而已。因为我们使用 `override` 所表达的意思是我们希望能覆盖基类中的虚函数而实际上并未做到，所以编译器会报错。

607

因为只有虚函数才能被覆盖，所以编译器会拒绝 D1 的 `f3`。该函数不是 B 中的虚函数，因此它不能被覆盖。类似的，`f4` 的声明也会发生错误，因为 B 中根本就没有名为 `f4` 的函数。

我们还能把某个函数指定为 `final`，如果我们已经把函数定义成 `final` 了，则之后任何尝试覆盖该函数的操作都将引发错误：

```
struct D2 : B {
    // 从 B 继承 f2() 和 f3()，覆盖 f1(int)
    void f1(int) const final;      // 不允许后续的其他类覆盖 f1(int)
};

struct D3 : D2 {
    void f2();                     // 正确：覆盖从间接基类 B 继承而来的 f2
    void f1(int) const;            // 错误：D2 已经将 f2 声明成 final
};
```

`final` 和 `override` 说明符出现在形参列表（包括任何 `const` 或引用修饰符）以及尾置返回类型（参见 6.3.3 节，第 206 页）之后。

虚函数与默认实参

和其他函数一样，虚函数也可以拥有默认实参（参见 6.5.1 节，第 211 页）。如果某次函数调用使用了默认实参，则该实参值由本次调用的静态类型决定。

换句话说，如果我们通过基类的引用或指针调用函数，则使用基类中定义的默认实参，即使实际运行的是派生类中的函数版本也是如此。此时，传入派生类函数的将是基类函数定义的默认实参。如果派生类函数依赖不同的实参，则程序结果将与我们的预期不符。



如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致。

回避虚函数的机制

在某些情况下，我们希望对虚函数的调用不要进行动态绑定，而是强迫其执行虚函数的某个特定版本。使用作用域运算符可以实现这一目的，例如下面的代码：

```
// 强行调用基类中定义的函数版本而不管 baseP 的动态类型到底是什么  
double undiscounted = baseP->Quote::net_price(42);
```

该代码强行调用 `Quote` 的 `net_price` 函数，而不管 `baseP` 实际指向的对象类型到底是什么。该调用将在编译时完成解析。



通常情况下，只有成员函数（或友元）中的代码才需要使用作用域运算符来回避虚函数的机制。

什么时候我们需要回避虚函数的默认机制呢？通常是当一个派生类的虚函数调用它覆盖的基类的虚函数版本时。在此情况下，基类的版本通常完成继承层次中所有类型都要做的共同任务，而派生类中定义的版本需要执行一些与派生类本身密切相关的操作。



如果一个派生类虚函数需要调用它的基类版本，但是没有使用作用域运算符，则在运行时该调用将被解析为对派生类版本自身的调用，从而导致无限递归。

608

15.3 节练习

练习 15.11：为你的 `Quote` 类体系添加一个名为 `debug` 的虚函数，令其分别显示每个类的数据成员。

练习 15.12：有必要将一个成员函数同时声明成 `override` 和 `final` 吗？为什么？

练习 15.13：给定下面的类，解释每个 `print` 函数的机理：

```
class base {  
public:  
    string name() { return basename; }  
    virtual void print(ostream &os) { os << basename; }  
private:
```

```

        string basename;
    };
    class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};

```

在上述代码中存在问题吗？如果有，你该如何修改它？

练习 15.14：给定上一题中的类以及下面这些对象，说明在运行时调用哪个函数：

base bobj;	base *bp1 = &bobj;	base &br1 = bobj;
derived dobj;	base *bp2 = &dobj;	base &br2 = dobj;
(a) bobj.print();	(b) dobj.print();	(c) bp1->name();
(d) bp2->name();	(e) br1.print();	(f) br2.print();

15.4 抽象基类

假设我们希望扩展书店程序并令其支持几种不同的折扣策略。除了购买量超过一定数量享受折扣外，我们也可能提供另外一种策略，即购买量不超过某个限额时可以享受折扣，但是一旦超过限额就要按原价支付。或者折扣策略还可能是购买量超过一定数量后购买的全部书籍都享受折扣，否则全都不打折。

上面的每个策略都要求一个购买量的值和一个折扣值。我们可以定义一个新的名为 Disc_quote 的类来支持不同的折扣策略，其中 Disc_quote 负责保存购买量的值和折扣值。其他的表示某种特定策略的类（如 Bulk_quote）将分别继承自 Disc_quote，每个派生类通过定义自己的 net_price 函数来实现各自的折扣策略。

在定义 Disc_quote 类之前，首先要确定它的 net_price 函数完成什么工作。显然我们的 Disc_quote 类与任何特定的折扣策略都无关，因此 Disc_quote 类中的 net_price 函数是没有实际含义的。

我们可以在 Disc_quote 类中不定义新的 net_price，此时，Disc_quote 将继承 Quote 中的 net_price 函数。

然而，这样的设计可能导致用户编写出一些无意义的代码。用户可能会创建一个 Disc_quote 对象并为其提供购买量和折扣值，如果将该对象传给一个像 print_total 这样的函数，则程序将调用 Quote 版本的 net_price。显然，最终计算出的销售价格并没有考虑我们在创建对象时提供的折扣值，因此上述操作毫无意义。

纯虚函数

认真思考上面描述的情形我们可以发现，关键问题不仅仅是不知道应该如何定义 net_price，而是我们根本就不希望用户创建一个 Disc_quote 对象。Disc_quote 类表示的是一本打折书籍的通用概念，而非某种具体的折扣策略。

我们可以将 net_price 定义成纯虚（pure virtual）函数从而令程序实现我们的设计意图，这样做可以清晰明了地告诉用户当前这个 net_price 函数是没有实际意义的。和普通的虚函数不一样，一个纯虚函数无须定义。我们通过在函数体的位置（即在声明语句

的分号之前) 书写=0 就可以将一个虚函数说明为纯虚函数。其中, =0 只能出现在类内部的虚函数声明语句处:

```
// 用于保存折扣值和购买量的类, 派生类使用这些数据可以实现不同的价格策略
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Quote(book, price),
        quantity(qty), discount(disc) { }
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0;           // 折扣适用的购买量
    double discount = 0.0;            // 表示折扣的小数值
};
```

和我们之前定义的 Bulk_quote 类一样, Disc_quote 也分别定义了一个默认构造函数和一个接受四个参数的构造函数。尽管我们不能直接定义这个类的对象, 但是 Disc_quote 的派生类构造函数将会使用 Disc_quote 的构造函数来构建各个派生类对象的 Disc_quote 部分。其中, 接受四个参数的构造函数将前两个参数传递给 Quote 的构造函数, 然后直接初始化自己的成员 discount 和 quantity。默认构造函数则对这些成员进行默认初始化。

值得注意的是, 我们也可以为纯虚函数提供定义, 不过函数体必须定义在类的外部。◀ 610也就是说, 我们不能在类的内部为一个=0 的函数提供函数体。

含有纯虚函数的类是抽象基类

含有(或者未经覆盖直接继承)纯虚函数的类是**抽象基类**(abstract base class)。抽象基类负责定义接口, 而后续的其他类可以覆盖该接口。我们不能(直接)创建一个抽象基类的对象。因为 Disc_quote 将 net_price 定义成了纯虚函数, 所以我们不能定义 Disc_quote 的对象。我们可以定义 Disc_quote 的派生类的对象, 前提是这些类覆盖了 net_price 函数:

```
// Disc_quote 声明了纯虚函数, 而 Bulk_quote 将覆盖该函数
Disc_quote discounted;           // 错误: 不能定义 Disc_quote 的对象
Bulk_quote bulk;                 // 正确: Bulk_quote 中没有纯虚函数
```

Disc_quote 的派生类必须给出自己的 net_price 定义, 否则它们仍将是抽象基类。



我们不能创建抽象基类的对象。

派生类构造函数只初始化它的直接基类

接下来可以重新实现 Bulk_quote 了, 这一次我们让它继承 Disc_quote 而非直接继承 Quote:

```
// 当同一书籍的销售量超过某个值时启用折扣
// 折扣的值是一个小于 1 的正的小数值, 以此来降低正常销售价格
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
```

```

Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
// 覆盖基类中的函数版本以实现一种新的折扣策略
double net_price(std::size_t) const override;
};

}

```

这个版本的 `Bulk_quote` 的直接基类是 `Disc_quote`, 间接基类是 `Quote`。每个 `Bulk_quote` 对象包含三个子对象: 一个(空的)`Bulk_quote`部分、一个`Disc_quote`子对象和一个`Quote`子对象。

如前所述, 每个类各自控制其对象的初始化过程。因此, 即使 `Bulk_quote` 没有自己的数据成员, 它也仍然需要像原来一样提供一个接受四个参数的构造函数。该构造函数将它的实参传递给 `Disc_quote` 的构造函数, 随后 `Disc_quote` 的构造函数继续调用 `Quote` 的构造函数。`Quote` 的构造函数首先初始化 `bulk` 的 `bookNo` 和 `price` 成员, 当 `Quote` 的构造函数结束后, 开始运行 `Disc_quote` 的构造函数并初始化 `quantity` 和 `discount` 成员, 最后运行 `Bulk_quote` 的构造函数, 该函数无须执行实际的初始化或其他工作。

611 >

关键概念: 重构

在 `Quote` 的继承体系中增加 `Disc_quote` 类是重构 (refactoring) 的一个典型示例。重构负责重新设计类的体系以便将操作和/或数据从一个类移动到另一个类中。对于面向对象的应用程序来说, 重构是一种很普遍的现象。

值得注意的是, 即使我们改变了整个继承体系, 那些使用了 `Bulk_quote` 或 `Quote` 的代码也无须进行任何改动。不过一旦类被重构 (或以其他方式被改变), 就意味着我们必须重新编译含有这些类的代码了。

15.4 节练习

练习 15.15: 定义你自己的 `Disc_quote` 和 `Bulk_quote`。

练习 15.16: 改写你在 15.2.2 节 (第 533 页) 练习中编写的数量受限的折扣策略, 令其继承 `Disc_quote`。

练习 15.17: 尝试定义一个 `Disc_quote` 的对象, 看看编译器给出的错误信息是什么?



15.5 访问控制与继承

每个类分别控制自己的成员初始化过程 (参见 15.2.2 节, 第 531 页), 与之类似, 每个类还分别控制着其成员对于派生类来说是否可访问 (accessible)。

受保护的成员

如前所述, 一个类使用 `protected` 关键字来声明那些它希望与派生类分享但是不想被其他公共访问使用的成员。`protected` 说明符可以看做是 `public` 和 `private` 中和后的产物:

- 和私有成员类似, 受保护的成员对于类的用户来说是不可访问的。

- 和公有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的。

此外，`protected` 还有另外一条重要的性质。

- 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权。

为了理解最后一条规则，请考虑如下的例子：

612

```
class Base {
protected:
    int prot_mem; // protected 成员
};

class Sneaky : public Base {
    friend void clobber(Sneaky&); // 能访问 Sneaky::prot_mem
    friend void clobber(Base&); // 不能访问 Base::prot_mem
    int j; // j 默认是 private
};

// 正确：clobber 能访问 Sneaky 对象的 private 和 protected 成员
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }
// 错误：clobber 不能访问 Base 的 protected 成员
void clobber(Base &b) { b.prot_mem = 0; }
```

如果派生类（及其友元）能访问基类对象的受保护成员，则上面的第二个 `clobber`（接受一个 `Base&`）将是合法的。该函数不是 `Base` 的友元，但是它仍然能够改变一个 `Base` 对象的内容。如果按照这样的思路，则我们只要定义一个形如 `Sneaky` 的新类就能非常简单地规避掉 `protected` 提供的访问保护了。

要想阻止以上的用法，我们就要做出如下规定，即派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员；对于普通的基类对象中的成员不具有特殊的访问权限。

公有、私有和受保护继承

某个类对其继承而来的成员的访问权限受到两个因素影响：一是在基类中该成员的访问说明符，二是在派生类的派生列表中的访问说明符。举个例子，考虑如下的继承关系：

```
class Base {
public:
    void pub_mem(); // public 成员
protected:
    int prot_mem; // protected 成员
private:
    char priv_mem; // private 成员
};

struct Pub_Derv : public Base {
    // 正确：派生类能访问 protected 成员
    int f() { return prot_mem; }
    // 错误：private 成员对于派生类来说是不可访问的
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // private 不影响派生类的访问权限
    int f1() const { return prot_mem; }
};
```

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员没什么影响。对 613 基类成员的访问权限只与基类中的访问说明符有关。Pub_Derv 和 Priv_Derv 都能访问受保护的成员 prot_mem，同时它们都不能访问私有成员 priv_mem。

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限：

```
Pub_Derv d1;           // 继承自 Base 的成员是 public 的
Priv_Derv d2;          // 继承自 Base 的成员是 private 的
d1.pub_mem();          // 正确：pub_mem 在派生类中是 public 的
d2.pub_mem();          // 错误：pub_mem 在派生类中是 private 的
```

Pub_Derv 和 Priv_Derv 都继承了 pub_mem 函数。如果继承是公有的，则成员将遵循其原有的访问说明符，此时 d1 可以调用 pub_mem。在 Priv_Derv 中，Base 的成员是私有的，因此类的用户不能调用 pub_mem。

派生访问说明符还可以控制继承自派生类的新类的访问权限：

```
struct Derived_from_Public : public Pub_Derv {
    // 正确：Base::prot_mem 在 Pub_Derv 中仍然是 protected 的
    int use_base() { return prot_mem; }
};

struct Derived_from_Private : public Priv_Derv {
    // 错误：Base::prot_mem 在 Priv_Derv 中是 private 的
    int use_base() { return prot_mem; }
};
```

Pub_Derv 的派生类之所以能访问 Base 的 prot_mem 成员是因为该成员在 Pub_Derv 中仍然是受保护的。相反，Priv_Derv 的派生类无法执行类的访问，对于它们来说，Priv_Derv 继承自 Base 的所有成员都是私有的。

假设我们之前还定义了一个名为 Prot_Derv 的类，它采用受保护继承，则 Base 的所有公有成员在新定义的类中都是受保护的。Prot_Derv 的用户不能访问 pub_mem，但是 Prot_Derv 的成员和友元可以访问那些继承而来的成员。



派生类向基类转换的可访问性

派生类向基类的转换（参见 15.2.2 节，第 530 页）是否可访问由使用该转换的代码决定，同时派生类的派生访问说明符也会有影响。假定 D 继承自 B：

- 只有当 D 公有地继承 B 时，用户代码才能使用派生类向基类的转换；如果 D 继承 B 的方式是受保护的或者私有的，则用户代码不能使用该转换。
- 不论 D 以什么方式继承 B，D 的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的。
- 如果 D 继承 B 的方式是公有的或者受保护的，则 D 的派生类的成员和友元可以使用 D 向 B 的类型转换；反之，如果 D 继承 B 的方式是私有的，则不能使用。



对于代码中的某个给定节点来说，如果基类的公有成员是可访问的，则派生类向基类的类型转换也是可访问的；反之则不行。

关键概念：类的设计与受保护的成员

不考虑继承的话，我们可以认为一个类有两种不同的用户：普通用户和类的实现者。

其中，普通用户编写的代码使用类的对象，这部分代码只能访问类的公有（接口）成员；实现者则负责编写类的成员和友元的代码，成员和友元既能访问类的公有部分，也能访问类的私有（实现）部分。

如果进一步考虑继承的话就会出现第三种用户，即派生类。基类把它希望派生类能够使用的部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。

和其他类一样，基类应该将其接口成员声明为公有的；同时将属于其实现的部分分成两组：一组可供派生类访问，另一组只能由基类及基类的友元访问。对于前者应该声明为受保护的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；对于后者应该声明为私有的。

友元与继承

就像友元关系不能传递一样（参见 7.3.4 节，第 250 页），友元关系同样也不能继承。基类的友元在访问派生类成员时不具有特殊性，类似的，派生类的友元也不能随意访问基类的成员：

```
class Base {
    // 添加 friend 声明，其他成员与之前的版本一致
    friend class Pal;           // Pal 在访问 Base 的派生类时不具有特殊性
};

class Pal {
public:
    int f(Base b) { return b.prot_mem; } // 正确：Pal 是 Base 的友元
    int f2(Sneaky s) { return s.j; }     // 错误：Pal 不是 Sneaky 的友元
    // 对基类的访问权限由基类本身控制，即使对于派生类的基类部分也是如此
    int f3(Sneaky s) { return s.prot_mem; } // 正确：Pal 是 Base 的友元
};
```

如前所述，每个类负责控制自己的成员的访问权限，因此尽管看起来有点儿奇怪，但 f3 确实是正确的。Pal 是 Base 的友元，所以 Pal 能够访问 Base 对象的成员，这种可访问性包括了 Base 对象内嵌在其派生类对象中的情况。◀615

当一个类将另一个类声明为友元时，这种友元关系只对做出声明的类有效。对于原来那个类来说，其友元的基类或者派生类不具有特殊的访问能力：

```
// D2 对 Base 的 protected 和 private 成员不具有特殊的访问能力
class D2 : public Pal {
public:
    int mem(Base b)
        { return b.prot_mem; }           // 错误：友元关系不能继承
};
```



不能继承友元关系；每个类负责控制各自成员的访问权限。

改变个别成员的可访问性

有时我们需要改变派生类继承的某个名字的访问级别，通过使用 using 声明（参见 3.1 节，第 74 页）可以达到这一目的：

```
class Base {
public:
```

```

        std::size_t size() const { return n; }
protected:
    std::size_t n;
};
class Derived : private Base { // 注意: private 继承
public:
    // 保持对象尺寸相关的成员的访问级别
    using Base::size;
protected:
    using Base::n;
};

```

因为 `Derived` 使用了私有继承，所以继承而来的成员 `size` 和 `n`（在默认情况下）是 `Derived` 的私有成员。然而，我们使用 `using` 声明语句改变了这些成员的可访问性。改变之后，`Derived` 的用户将可以使用 `size` 成员，而 `Derived` 的派生类将能使用 `n`。

通过在类的内部使用 `using` 声明语句，我们可以将该类的直接或间接基类中的任何可访问成员（例如，非私有成员）标记出来。`using` 声明语句中名字的访问权限由该 `using` 声明语句之前的访问说明符来决定。也就是说，如果一条 `using` 声明语句出现在类的 `private` 部分，则该名字只能被类的成员和友元访问；如果 `using` 声明语句位于 `public` 部分，则类的所有用户都能访问它；如果 `using` 声明语句位于 `protected` 部分，则该名字对于成员、友元和派生类是可访问的。



派生类只能为那些它可以访问的名字提供 `using` 声明。

616 >

默认的继承保护级别

在 7.2 节（第 240 页）中我们曾经介绍过使用 `struct` 和 `class` 关键字定义的类具有不同的默认访问说明符。类似的，默认派生运算符也由定义派生类所用的关键字来决定。默认情况下，使用 `class` 关键字定义的派生类是私有继承的；而使用 `struct` 关键字定义的派生类是公有继承的：

```

class Base { /* ... */ };
struct D1 : Base { /* ... */ }; // 默认 public 继承
class D2 : Base { /* ... */ }; // 默认 private 继承

```

人们常常有一种错觉，认为在使用 `struct` 关键字和 `class` 关键字定义的类之间还有更深层次的差别。事实上，唯一的差别就是默认成员访问说明符及默认派生访问说明符；除此之外，再无其他不同之处。



一个私有派生的类最好显式地将 `private` 声明出来，而不要仅仅依赖于默认的设置。显式声明的好处是可以令私有继承关系清晰明了，不至于产生误会。

15.5 节练习

练习 15.18：假设给定了第 543 页和第 544 页的类，同时已知每个对象的类型如注释所示，判断下面的哪些赋值语句是合法的。解释那些不合法的语句为什么不允许：

<code>Base *p = &d1;</code>	<code>// d1 的类型是 Pub_Derv</code>
<code>p = &d2;</code>	<code>// d2 的类型是 Priv_Derv</code>

```

p = &d3;           // d3 的类型是 Prot_Derv
p = &dd1;          // dd1 的类型是 Derived_from_Public
p = &dd2;          // dd2 的类型是 Derived_from_Private
p = &dd3;          // dd3 的类型是 Derived_from_Protected

```

练习 15.19: 假设 543 页和 544 页的每个类都有如下形式的成员函数：

```
void memfcn(Base &b) { b = *this; }
```

对于每个类，分别判断上面的函数是否合法。

练习 15.20: 编写代码检验你对前面两题的回答是否正确。

练习 15.21: 从下面这些一般性抽象概念中任选一个（或者选一个你自己的），将其对应的一组类型组织成一个继承体系：

- (a) 图形文件格式（如 gif、tiff、jpeg、bmp）
- (b) 图形基元（如方格、圆、球、圆锥）
- (c) C++语言中的类型（如类、函数、成员函数）

练习 15.22: 对于你在上一题中选择的类，为其添加合适的虚函数及公有成员和受保护的成员。

15.6 继承中的类作用域



每个类定义自己的作用域（参见 7.4 节，第 253 页），在这个作用域内我们定义类的成员。当存在继承关系时，派生类的作用域嵌套（参见 2.2.4 节，第 43 页）在其基类的作用域之内。如果一个名字在派生类的作用域内无法正确解析，则编译器将继续在外层的基类作用域中寻找该名字的定义。

< 617

派生类的作用域位于基类作用域之内这一事实可能有点儿出人意料，毕竟在我们的程序文本中派生类和基类的定义是相互分离开来的。不过也恰恰因为类作用域有这种继承嵌套的关系，所以派生类才能像使用自己的成员一样使用基类的成员。例如，当我们编写下面的代码时：

```
Bulk_quote bulk;
cout << bulk.isbn();
```

名字 isbn 的解析将按照下述过程所示：

- 因为我们是通过 Bulk_quote 的对象调用 isbn 的，所以首先在 Bulk_quote 中查找，这一步没有找到名字 isbn。
- 因为 Bulk_quote 是 Disc_quote 的派生类，所以接下来在 Disc_quote 中查找，仍然找不到。
- 因为 Disc_quote 是 Quote 的派生类，所以接着查找 Quote；此时找到了名字 isbn，所以我们使用的 isbn 最终被解析为 Quote 中的 isbn。

在编译时进行名字查找

一个对象、引用或指针的静态类型（参见 15.2.3 节，第 532 页）决定了该对象的哪些成员是可见的。即使静态类型与动态类型可能不一致（当使用基类的引用或指针时会发生

这种情况), 但是我们能使用哪些成员仍然是由静态类型决定的。举个例子, 我们可以给 Disc_quote 添加一个新成员, 该成员返回一个存有最小(或最大)数量及折扣价格的 pair (参见 11.2.3 节, 第 379 页):

```
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
    { return {quantity, discount}; }
    // 其他成员与之前的版本一致
};
```

我们只能通过 Disc_quote 及其派生类的对象、引用或指针使用 discount_policy:

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk;           // 静态类型与动态类型一致
Quote *itemP = &bulk;               // 静态类型与动态类型不一致
bulkP->discount_policy();         // 正确: bulkP 的类型是 Bulk_quote*
itemP->discount_policy();         // 错误: itemP 的类型是 Quote*
```

618 尽管在 bulk 中确实含有一个名为 discount_policy 的成员, 但是该成员对于 itemP 却是不可见的。itemP 的类型是 Quote 的指针, 意味着对 discount_policy 的搜索将从 Quote 开始。显然 Quote 不包含名为 discount_policy 的成员, 所以我们无法通过 Quote 的对象、引用或指针调用 discount_policy。

名字冲突与继承

和其他作用域一样, 派生类也能重用定义在其直接基类或间接基类中的名字, 此时定义在内层作用域(即派生类)的名字将隐藏定义在外层作用域(即基类)的名字(参见 2.2.4 节, 第 43 页):

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};

struct Derived : Base {
    Derived(int i): mem(i) { }           // 用 i 初始化 Derived::mem
                                         // Base::mem 进行默认初始化
    int get_mem() { return mem; }        // 返回 Derived::mem
protected:
    int mem;                           // 隐藏基类中的 mem
};
```

get_mem 中 mem 引用的解析结果是定义在 Derived 中的名字, 下面的代码

```
Derived d(42);
cout << d.get_mem() << endl;          // 打印 42
```

的输出结果将是 42。



派生类的成员将隐藏同名的基类成员。

通过作用域运算符来使用隐藏的成员

我们可以通过作用域运算符来使用一个被隐藏的基类成员:

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
    // ...
};
```

作用域运算符将覆盖掉原有的查找规则，并指示编译器从 `Base` 类的作用域开始查找 `mem`。如果使用最新的 `Derived` 版本运行上面的代码，则 `d.get_mem()` 的输出结果将是 0。

Best Practices

除了覆盖继承而来的虚函数之外，派生类最好不要重用其他定义在基类中的名字。

关键概念：名字查找与继承

619

理解函数调用的解析过程对于理解 C++ 的继承至关重要，假定我们调用 `p->mem()`（或者 `obj.mem()`），则依次执行以下 4 个步骤：

- 首先确定 `p`（或 `obj`）的静态类型。因为我们调用的是一个成员，所以该类型必然是类类型。
- 在 `p`（或 `obj`）的静态类型对应的类中查找 `mem`。如果找不到，则依次在直接基类中不断查找直至到达继承链的顶端。如果找遍了该类及其基类仍然找不到，则编译器将报错。
- 一旦找到了 `mem`，就进行常规的类型检查（参见 6.1 节，第 183 页）以确认对于当前找到的 `mem`，本次调用是否合法。
- 假设调用合法，则编译器将根据调用的是否是虚函数而产生不同的代码：
 - 如果 `mem` 是虚函数且我们是通过引用或指针进行的调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据是对象的动态类型。
 - 反之，如果 `mem` 不是虚函数或者我们是通过对象（而非引用或指针）进行的调用，则编译器将产生一个常规函数调用。

一如既往，名字查找先于类型检查

如前所述，声明在内层作用域的函数并不会重载声明在外层作用域的函数（参见 6.4.1 节，第 210 页）。因此，定义派生类中的函数也不会重载其基类中的成员。和其他作用域一样，如果派生类（即内层作用域）的成员与基类（即外层作用域）的某个成员同名，则派生类将在其作用域内隐藏该基类成员。即使派生类成员和基类成员的形参列表不一致，基类成员也仍然会被隐藏掉：

```
struct Base {
    int memfcn();
};

struct Derived : Base {
    int memfcn(int);           // 隐藏基类的 memfcn
};

Derived d; Base b;
b.memfcn();                  // 调用 Base::memfcn
d.memfcn(10);                // 调用 Derived::memfcn
d.memfcn();                  // 错误：参数列表为空的 memfcn 被隐藏了
d.Base::memfcn();            // 正确：调用 Base::memfcn
```

Derived 中的 memfcn 声明隐藏了 Base 中的 memfcn 声明。在上面的代码中前两条调用语句容易理解，第一个通过 Base 对象 b 进行的调用执行基类的版本；类似的，第二个通过 d 进行的调用执行 Derived 的版本；第三条调用语句有点特殊，d.memfcn() 是非法的。

为了解析这条调用语句，编译器首先在 Derived 中查找名字 memfcn；因为 Derived 确实定义了一个名为 memfcn 的成员，所以查找过程终止。一旦名字找到，编译器就不再继续查找了。Derived 中的 memfcn 版本需要一个 int 实参，而当前的调用语句无法提供任何实参，所以该调用语句是错误的。

虚函数与作用域

我们现在可以理解为什么基类与派生类中的虚函数必须有相同的形参列表了（参见 15.3 节，第 537 页）。假如基类与派生类的虚函数接受的实参不同，则我们就无法通过基类的引用或指针调用派生类的虚函数了。例如：

```
class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // 隐藏基类的 fcn，这个 fcn 不是虚函数
    // D1 继承了 Base::fcn() 的定义
    int fcn(int);           // 形参列表与 Base 中的 fcn 不一致
    virtual void f2();       // 是一个新的虚函数，在 Base 中不存在
};

class D2 : public D1 {
public:
    int fcn(int);           // 是一个非虚函数，隐藏了 D1::fcn(int)
    int fcn();               // 覆盖了 Base 的虚函数 fcn
    void f2();               // 覆盖了 D1 的虚函数 f2
};
```

D1 的 fcn 函数并没有覆盖 Base 的虚函数 fcn，原因是它们的形参列表不同。实际上，D1 的 fcn 将隐藏 Base 的 fcn。此时拥有了两个名为 fcn 的函数：一个是 D1 从 Base 继承而来的虚函数 fcn；另一个是 D1 自己定义的接受一个 int 参数的非虚函数 fcn。

通过基类调用隐藏的虚函数

给定上面定义的这些类后，我们来看几种使用其函数的方法：

```
Base bobj; D1 d1obj; D2 d2obj;

Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn();           // 虚调用，将在运行时调用 Base::fcn
bp2->fcn();           // 虚调用，将在运行时调用 Base::fcn
bp3->fcn();           // 虚调用，将在运行时调用 D2::fcn

D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2();             // 错误：Base 没有名为 f2 的成员
d1p->f2();             // 虚调用，将在运行时调用 D1::f2()
d2p->f2();             // 虚调用，将在运行时调用 D2::f2()
```

前三条调用语句是通过基类的指针进行的，因为 `fcn` 是虚函数，所以编译器产生的代码将在运行时确定使用虚函数的那个版本。判断的依据是该指针所绑定对象的真实类型。在 `bp2` 的例子中，实际绑定的对象是 `D1` 类型，而 `D1` 并没有覆盖那个不接受实参的 `fcn`，所以通过 `bp2` 进行的调用将在运行时解析为 `Base` 定义的版本。

接下来的三条调用语句是通过不同类型的指针进行的，每个指针分别指向继承体系中的一个类型。因为 `Base` 类中没有 `fcn()`，所以第一条语句是非法的，即使当前的指针碰巧指向了一个派生类对象也无济于事。

为了完整地阐明上述问题，我们不妨再观察一些对于非虚函数 `fcn(int)` 的调用语句：

```
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42);           // 错误: Base 中没有接受一个 int 的 fcn
p2->fcn(42);           // 静态绑定, 调用 D1::fcn(int)
p3->fcn(42);           // 静态绑定, 调用 D2::fcn(int)
```

在上面的每条调用语句中，指针都指向了 `D2` 类型的对象，但是由于我们调用的是非虚函数，所以不会发生动态绑定。实际调用的函数版本由指针的静态类型决定。

覆盖重载的函数

和其他函数一样，成员函数无论是否是虚函数都能被重载。派生类可以覆盖重载函数的 0 个或多个实例。如果派生类希望所有的重载版本对于它来说都是可见的，那么它就需要覆盖所有的版本，或者一个也不覆盖。

有时一个类仅需覆盖重载集合中的一些而非全部函数，此时，如果我们不得不覆盖基类中的每一个版本的话，显然操作将极其烦琐。

一种好的解决方案是为重载的成员提供一条 `using` 声明语句（参见 15.5 节，第 546 页），这样我们就无须覆盖基类中的每一个重载版本了。`using` 声明语句指定一个名字而不指定形参列表，所以一条基类成员函数的 `using` 声明语句就可以把该函数的所有重载实例添加到派生类作用域中。此时，派生类只需要定义其特有的函数就可以了，而无须为继承而来的其他函数重新定义。

类内 `using` 声明的一般规则同样适用于重载函数的名字（参见 15.5 节，第 546 页）；基类函数的每个实例在派生类中都必须是可访问的。对派生类没有重新定义的重载版本的访问实际上是对 `using` 声明点的访问。

15.6 节练习

练习 15.23: 假设第 550 页的 `D1` 类需要覆盖它继承而来的 `fcn` 函数，你应该如何对其进行修改？如果你修改之后 `fcn` 匹配了 `Base` 中的定义，则该节的那些调用语句将如何解析？

15.7 构造函数与拷贝控制

和其他类一样，位于继承体系中的类也需要控制当其对象执行一系列操作时发生什么样的行为，这些操作包括创建、拷贝、移动、赋值和销毁。如果一个类（基类或派生类）没有定义拷贝控制操作，则编译器将为它合成一个版本。当然，这个合成的版本也可以定义成被删除的函数。



15.7.1 虚析构函数

继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个虚析构函数（参见 15.2.1 节，第 528 页），这样我们就能动态分配继承体系中的对象了。

如前所述，当我们 `delete` 一个动态分配的对象的指针时将执行析构函数（参见 13.1.3 节，第 445 页）。如果该指针指向继承体系中的某个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符的情况（参见 15.2.2 节，第 530 页）。例如，如果我们 `delete` 一个 `Quote*` 类型的指针，则该指针有可能实际指向了一个 `Bulk_quote` 类型的对象。如果这样的话，编译器就必须清楚它应该执行的是 `Bulk_quote` 的析构函数。和其他函数一样，我们通过在基类中将析构函数定义成虚函数以确保执行正确的析构函数版本：

```
class Quote {
public:
    // 如果我们删除的是一个指向派生类对象的基类指针，则需要虚析构函数
    virtual ~Quote() = default;           // 动态绑定析构函数
};
```

和其他虚函数一样，析构函数的虚属性也会被继承。因此，无论 `Quote` 的派生类使用合成的析构函数还是定义自己的析构函数，都将是虚析构函数。只要基类的析构函数是虚函数，就能确保当我们 `delete` 基类指针时将运行正确的析构函数版本：

```
Quote *itemP = new Quote;           // 静态类型与动态类型一致
delete itemP;                      // 调用 Quote 的析构函数
itemP = new Bulk_quote;             // 静态类型与动态类型不一致
delete itemP;                      // 调用 Bulk_quote 的析构函数
```



WARNING 如果基类的析构函数不是虚函数，则 `delete` 一个指向派生类对象的基类指针将产生未定义的行为。

之前我们曾介绍过一条经验准则，即如果一个类需要析构函数，那么它也同样需要拷贝和赋值操作（参见 13.1.4 节，第 447 页）。基类的析构函数并不遵循上述准则，它是一个重要的例外。一个基类总是需要析构函数，而且它能将析构函数设定为虚函数。此时，该析构函数为了成为虚函数而令内容为空，我们显然无法由此推断该基类还需要赋值运算符或拷贝构造函数。

623 > 虚析构函数将阻止合成移动操作

基类需要一个虚析构函数这一事实还会对基类和派生类的定义产生另外一个间接的影响：如果一个类定义了析构函数，即使它通过`=default` 的形式使用了合成的版本，编译器也不会为这个类合成移动操作（参见 13.6.2 节，第 475 页）。

15.7.1 节练习

练习 15.24：哪种类需要虚析构函数？虚析构函数必须执行什么样的操作？



15.7.2 合成拷贝控制与继承

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数、赋值运算符或析构函数类似：它们对类本身的成员依次进行初始化、赋值或销毁的操作。此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化、赋值或销

毁的操作。例如，

- 合成的 Bulk_quote 默认构造函数运行 Disc_quote 的默认构造函数，后者又运行 Quote 的默认构造函数。
- Quote 的默认构造函数将 bookNo 成员默认初始化为空字符串，同时使用类内初始值将 price 初始化为 0。
- Quote 的构造函数完成后，继续执行 Disc_quote 的构造函数，它使用类内初始值初始化 qty 和 discount。
- Disc_quote 的构造函数完成后，继续执行 Bulk_quote 的构造函数，但是它什么具体工作也不做。

类似的，合成的 Bulk_quote 拷贝构造函数使用（合成的） Disc_quote 拷贝构造函数，后者又使用（合成的） Quote 拷贝构造函数。其中， Quote 拷贝构造函数拷贝 bookNo 和 price 成员； Disc_quote 拷贝构造函数拷贝 qty 和 discount 成员。

值得注意的是，无论基类成员是合成的版本（如 Quote 继承体系的例子）还是自定义的版本都没有太大影响。唯一的要求是相应的成员应该可访问（参见 15.5 节，第 542 页）并且不是一个被删除的函数。

在我们的 Quote 继承体系中，所有类都使用合成的析构函数。其中，派生类隐式地使用而基类通过将其虚析构函数定义成`=default`而显式地使用。一如既往，合成的析构函数体是空的，其隐式的析构部分负责销毁类的成员（参见 13.1.3 节，第 444 页）。对于派生类的析构函数来说，它除了销毁派生类自己的成员外，还负责销毁派生类的直接基类；该直接基类又销毁它自己的直接基类，以此类推直至继承链的顶端。

如前所述，Quote 因为定义了析构函数而不能拥有合成的移动操作，因此当我们移动 Quote 对象时实际使用的是合成的拷贝操作（参见 13.6.2 节，第 477 页）。如我们即将看到的那样，Quote 没有移动操作意味着它的派生类也没有。

派生类中删除的拷贝控制与基类的关系

就像其他任何类的情况一样，基类或派生类也能出于同样的原因将其合成的默认构造函数或者任何一个拷贝控制成员定义成被删除的函数（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。此外，某些定义基类的方式也可能导致有的派生类成员成为被删除的函数：

- 如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的函数或者不可访问（参见 15.5 节，第 543 页），则派生类中对应的成员将是被删除的，原因是编译器不能使用基类成员来执行派生类对象基类部分的构造、赋值或销毁操作。
- 如果在基类中有一个不可访问或删除掉的析构函数，则派生类中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分。
- 和过去一样，编译器将不会合成一个删除掉的移动操作。当我们使用`=default`请求一个移动操作时，如果基类中的对应操作是删除的或不可访问的，那么派生类中该函数将是被删除的，原因是派生类对象的基类部分不可移动。同样，如果基类的析构函数是删除的或不可访问的，则派生类的移动构造函数也将是被删除的。

举个例子，对于下面的基类 B 来说：

```
class B {  
public:  
    B();
```

C++
11

624

C++
11

```

B(const B&) = delete;
// 其他成员，不含有移动构造函数
};

class D : public B {
    // 没有声明任何构造函数
};

D d;                      // 正确：D 的合成默认构造函数使用 B 的默认构造函数
D d2(d);                  // 错误：D 的合成拷贝构造函数是被删除的
D d3(std::move(d));       // 错误：隐式地使用 D 的被删除的拷贝构造函数

```

基类 B 含有一个可访问的默认构造函数和一个显式删除的拷贝构造函数。因为我们定义了拷贝构造函数，所以编译器将不会为 B 合成一个移动构造函数(参见 13.6.2 节，第 475 页)。因此，我们既不能移动也不能拷贝 B 的对象。如果 B 的派生类希望它自己的对象能被移动和拷贝，则派生类需要自定义相应版本的构造函数。当然，在这一过程中派生类还必须考虑如何移动或拷贝其基类部分的成员。在实际编程过程中，如果在基类中没有默认、拷贝或移动构造函数，则一般情况下派生类也不会定义相应的操作。

625 移动操作与继承

如前所述，大多数基类都会定义一个虚析构函数。因此在默认情况下，基类通常不含有合成的移动操作，而且在它的派生类中也没有合成的移动操作。

因为基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以当我们确实需要执行移动操作时应该首先在基类中进行定义。我们的 `Quote` 可以使用合成的版本，不过前提是 `Quote` 必须显式地定义这些成员。一旦 `Quote` 定义了自己的移动操作，那么它必须同时显式地定义拷贝操作(参见 13.6.2 节，第 476 页)：

```

class Quote {
public:
    Quote() = default;                                // 对成员依次进行默认初始化
    Quote(const Quote&) = default;                   // 对成员依次拷贝
    Quote(Quote&&) = default;                        // 对成员依次拷贝
    Quote& operator=(const Quote&) = default;        // 拷贝赋值
    Quote& operator=(Quote&&) = default;            // 移动赋值
    virtual ~Quote() = default;
    // 其他成员与之前的版本一致
};

```

通过上面的定义，我们就能对 `Quote` 的对象逐成员地分别进行拷贝、移动、赋值和销毁操作了。而且除非 `Quote` 的派生类中含有排斥移动的成员，否则它将自动获得合成的移动操作。

15.7.2 节练习

练习 15.25: 我们为什么为 `Disc_quote` 定义一个默认构造函数？如果去除掉该构造函数的话会对 `Bulk_quote` 的行为产生什么影响？



15.7.3 派生类的拷贝控制成员

如我们在 15.2.2 节(第 531 页)介绍过的，派生类构造函数在其初始化阶段中不但要初始化派生类自己的成员，还负责初始化派生类对象的基类部分。因此，派生类的拷贝和

移动构造函数在拷贝和移动自有成员的同时，也要拷贝和移动基类部分的成员。类似的，派生类赋值运算符也必须为其基类部分的成员赋值。

和构造函数及赋值运算符不同的是，析构函数只负责销毁派生类自己分配的资源。如前所述，对象的成员是被隐式销毁的（参见 13.1.3 节，第 445 页）；类似的，派生类对象的基类部分也是自动销毁的。



当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。

<626

定义派生类的拷贝或移动构造函数



当为派生类定义拷贝或移动构造函数时（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页），我们通常使用对应的基类构造函数初始化对象的基类部分：

```
class Base { /* ... */ };
class D: public Base {
public:
    // 默认情况下，基类的默认构造函数初始化对象的基类部分
    // 要想使用拷贝或移动构造函数，我们必须在构造函数初始值列表中
    // 显式地调用该构造函数
    D(const D& d): Base(d)           // 拷贝基类成员
        /* D 的成员的初始值 */ { /* ... */ }
    D(D&& d): Base(std::move(d))      // 移动基类成员
        /* D 的成员的初始值 */ { /* ... */ }
};
```

初始值 `Base(d)` 将一个 `D` 对象传递给基类构造函数。尽管从道理上来说，`Base` 可以包含一个参数类型为 `D` 的构造函数，但是在实际编程过程中通常不会这么做。相反，`Base(d)` 一般会匹配 `Base` 的拷贝构造函数。`D` 类型的对象 `d` 将被绑定到该构造函数的 `Base&` 形参上。`Base` 的拷贝构造函数负责将 `d` 的基类部分拷贝给要创建的对象。假如我们没有提供基类的初始值的话：

```
// D 的这个拷贝构造函数很可能是不正确的定义
// 基类部分被默认初始化，而非拷贝
D(const D& d) /* 成员初始值，但是没有提供基类初始值 */
{ /* ... */ }
```

在上面的例子中，`Base` 的默认构造函数将被用来初始化 `D` 对象的基类部分。假定 `D` 的构造函数从 `d` 中拷贝了派生类成员，则这个新构建的对象的配置将非常奇怪：它的 `Base` 成员被赋予了默认值，而 `D` 成员的值则是从其他对象拷贝得来的。



在默认情况下，基类默认构造函数初始化派生类对象的基类部分。如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显式地使用基类的拷贝（或移动）构造函数。

<627

派生类赋值运算符

与拷贝和移动构造函数一样，派生类的赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）也必须显式地为其基类部分赋值：

```
// Base::operator=(const Base&) 不会被自动调用
```

<627

```
D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // 为基类部分赋值
    // 按照过去的方式为派生类的成员赋值
    // 酌情处理自赋值及释放已有资源等情况
    return *this;
}
```

上面的运算符首先显式地调用基类赋值运算符，令其为派生类对象的基类部分赋值。基类的运算符（应该可以）正确地处理自赋值的情况，如果赋值命令是正确的，则基类运算符将释放掉其左侧运算对象的基类部分的旧值，然后利用 `rhs` 为其赋一个新值。随后，我们继续进行其他为派生类成员赋值的工作。

值得注意的是，无论基类的构造函数或赋值运算符是自定义的版本还是合成的版本，派生类的对应操作都能使用它们。例如，对于 `Base::operator=` 的调用语句将执行 `Base` 的拷贝赋值运算符，至于该运算符是由 `Base` 显式定义的还是由编译器合成的无关紧要。

派生类析构函数

如前所述，在析构函数体执行完成后，对象的成员会被隐式销毁（参见 13.1.3 节，第 445 页）。类似的，对象的基类部分也是隐式销毁的。因此，和构造函数及赋值运算符不同的是，派生类析构函数只负责销毁由派生类自己分配的资源：

```
class D: public Base {
public:
    // Base::~Base 被自动调用执行
    ~D() { /* 该处由用户定义清除派生类成员的操作 */ }
};
```

对象销毁的顺序正好与其创建的顺序相反：派生类析构函数首先执行，然后是基类的析构函数，以此类推，沿着继承体系的反方向直至最后。

在构造函数和析构函数中调用虚函数

如我们所知，派生类对象的基类部分将首先被构建。当执行基类的构造函数时，该对象的派生类部分是未被初始化的状态。类似的，销毁派生类对象的次序正好相反，因此当执行基类的析构函数时，派生类部分已经被销毁掉了。由此可知，当我们执行上述基类成员的时候，该对象处于未完成的状态。

为了能够正确地处理这种未完成状态，编译器认为对象的类型在构造或析构的过程中仿佛发生了改变一样。也就是说，当我们构建一个对象时，需要把对象的类和构造函数的类看作是同一个；对虚函数的调用绑定正好符合这种把对象的类和构造函数的类看成同一个的要求；对于析构函数也是同样的道理。上述的绑定不但对直接调用虚函数有效，对间接调用也是有效的，这里的间接调用是指通过构造函数（或析构函数）调用另一个函数。
628>

为了理解上述行为，不妨考虑当基类构造函数调用虚函数的派生类版本时会发生什么情况。这个虚函数可能会访问派生类的成员，毕竟，如果它不需要访问派生类成员的话，则派生类直接使用基类的虚函数版本就可以了。然而，当执行基类构造函数时，它要用到的派生类成员尚未初始化，如果我们允许这样的访问，则程序很可能会崩溃。



如果构造函数或析构函数调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对应的虚函数版本。

15.7.3 节练习

练习 15.26: 定义 `Quote` 和 `Bulk_quote` 的拷贝控制成员，令其与合成的版本行为一致。为这些成员以及其他构造函数添加打印状态的语句，使得我们能够知道正在运行哪个程序。使用这些类编写程序，预测程序将创建和销毁哪些对象。重复实验，不断比较你的预测和实际输出结果是否相同，直到预测完全准确再结束。

15.7.4 继承的构造函数

在 C++11 新标准中，派生类能够重用其直接基类定义的构造函数。尽管如我们所知，这些构造函数并非以常规的方式继承而来，但是为了方便，我们不妨姑且称其为“继承”的。一个类只初始化它的直接基类，出于同样的原因，一个类也只继承其直接基类的构造函数。类不能继承默认、拷贝和移动构造函数。如果派生类没有直接定义这些构造函数，则编译器将为派生类合成它们。

派生类继承基类构造函数的方式是提供一条注明了（直接）基类名的 `using` 声明语句。举个例子，我们可以重新定义 `Bulk_quote` 类（参见 15.4 节，第 541 页），令其继承 `Disc_quote` 类的构造函数：

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // 继承 Disc_quote 的构造函数
    double net_price(std::size_t) const;
};
```

通常情况下，`using` 声明语句只是令某个名字在当前作用域内可见。而当作用于构造函数时，`using` 声明语句将令编译器产生代码。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。换句话说，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。

这些编译器生成的构造函数形如：

```
derived(parms) : base(args) { }
```

其中，`derived` 是派生类的名字，`base` 是基类的名字，`parms` 是构造函数的形参列表，`args` 将派生类构造函数的形参传递给基类的构造函数。在我们的 `Bulk_quote` 类中，继承的构造函数等价于：

```
Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
```

如果派生类含有自己的数据成员，则这些成员将被默认初始化（参见 7.1.4 节，第 238 页）。

继承的构造函数的特点

和普通成员的 `using` 声明不一样，一个构造函数的 `using` 声明不会改变该构造函数的访问级别。例如，不管 `using` 声明出现在哪儿，基类的私有构造函数在派生类中还是一个私有构造函数；受保护的构造函数和公有构造函数也是同样的规则。

而且，一个 `using` 声明语句不能指定 `explicit` 或 `constexpr`。如果基类的构造函数是 `explicit`（参见 7.5.4 节，第 265 页）或者 `constexpr`（参见 7.5.6 节，第 267

C++
11

629

页), 则继承的构造函数也拥有相同的属性。

当一个基类构造函数含有默认实参(参见 6.5.1 节, 第 211 页)时, 这些实参并不会被继承。相反, 派生类将获得多个继承的构造函数, 其中每个构造函数分别省略掉一个含有默认实参的形参。例如, 如果基类有一个接受两个形参的构造函数, 其中第二个形参含有默认实参, 则派生类将获得两个构造函数: 一个构造函数接受两个形参(没有默认实参), 另一个构造函数只接受一个形参, 它对应于基类中最左侧的没有默认值的那个形参。

如果基类含有几个构造函数, 则除了两个例外情况, 大多数时候派生类会继承所有这些构造函数。第一个例外是派生类可以继承一部分构造函数, 而为其他构造函数定义自己的版本。如果派生类定义的构造函数与基类的构造函数具有相同的参数列表, 则该构造函数将不会被继承。定义在派生类中的构造函数将替换继承而来的构造函数。

第二个例外是默认、拷贝和移动构造函数不会被继承。这些构造函数按照正常规则被合成。继承的构造函数不会被作为用户定义的构造函数来使用, 因此, 如果一个类只含有继承的构造函数, 则它也将拥有一个合成的默认构造函数。

15.7.4 节练习

练习 15.27: 重新定义你的 Bulk_quote 类, 令其继承构造函数。



15.8 容器与继承

630 >

当我们使用容器存放继承体系中的对象时, 通常必须采取间接存储的方式。因为不允许在容器中保存不同类型的元素, 所以我们不能把具有继承关系的多种类型的对象直接存放在容器当中。

举个例子, 假定我们想定义一个 vector, 令其保存用户准备购买的几种书籍。显然我们不应该用 vector 保存 Bulk_quote 对象。因为我们不能将 Quote 对象转换成 Bulk_quote (参见 15.2.3 节, 第 534 页), 所以我们将无法把 Quote 对象放置在该 vector 中。

其实, 我们也不应该使用 vector 保存 Quote 对象。此时, 虽然我们可以把 Bulk_quote 对象放置在容器中, 但是这些对象再也不是 Bulk_quote 对象了:

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// 正确: 但是只能把对象的 Quote 部分拷贝给 basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本, 打印 750, 即 15 * $50
cout << basket.back().net_price(15) << endl;
```

basket 的元素是 Quote 对象, 因此当我们向该 vector 中添加一个 Bulk_quote 对象时, 它的派生类部分将被忽略掉 (参见 15.2.3 节, 第 535 页)。



当派生类对象被赋值给基类对象时, 其中的派生类部分将被“切掉”, 因此容器和存在继承关系的类型无法兼容。

在容器中放置（智能）指针而非对象

当我们希望在容器中存放具有继承关系的对象时，我们实际上存放的通常是基类的指针（更好的选择是智能指针（参见 12.1 节，第 400 页））。和往常一样，这些指针所指对象的动态类型可能是基类类型，也可能是派生类类型：

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本；打印 562.5，即在 15*50 中扣除掉折扣金额
cout << basket.back()->net_price(15) << endl;
```

因为 `basket` 存放着 `shared_ptr`，所以我们必须解引用 `basket.back()` 的返回值以获得运行 `net_price` 的对象。我们通过在 `net_price` 的调用中使用 `->` 以达到这个目的。如我们所知，实际调用的 `net_price` 版本依赖于指针所指对象的动态类型。

值得注意的是，我们将 `basket` 定义成 `shared_ptr<Quote>`，但是在第二个 `push_back` 中传入的是一个 `Bulk_quote` 对象的 `shared_ptr`。正如我们可以将一个派生类的普通指针转换成基类指针一样（参见 15.2.2 节，第 530 页），我们也能把一个派生类的智能指针转换成基类的智能指针。在此例中，`make_shared<Bulk_quote>` 返回一个 `shared_ptr<Bulk_quote>` 对象，当我们调用 `push_back` 时该对象被转换成 `shared_ptr<Quote>`。因此尽管在形式上有所差别，但实际上 `basket` 的所有元素的类型都是相同的。

<631

15.8 节练习

练习 15.28： 定义一个存放 `Quote` 对象的 `vector`，将 `Bulk_quote` 对象传入其中。
计算 `vector` 中所有元素总的 `net_price`。

练习 15.29： 再运行一次你的程序，这次传入 `Quote` 对象的 `shared_ptr`。如果这次计算出的总额与之前的程序不一致，解释为什么；如果一致，也请说明原因。

15.8.1 编写 Basket 类



对于 C++ 面向对象的编程来说，一个悖论是我们无法直接使用对象进行面向对象编程。相反，我们必须使用指针和引用。因为指针会增加程序的复杂性，所以我们经常定义一些辅助的类来处理这种复杂情况。首先，我们定义一个表示购物篮的类：

```
class Basket {
public:
    // Basket 使用合成的默认构造函数和拷贝控制成员
    void add_item(const std::shared_ptr<Quote> &sale)
    { items.insert(sale); }
    // 打印每本书的总价和购物篮中所有书的总价
    double total_receipt(std::ostream&) const;
private:
    // 该函数用于比较 shared_ptr，multiset 成员会用到它
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset 保存多个报价，按照 compare 成员排序
```

```
    std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
        items{compare};
};
```

我们的类使用一个 `multiset` (参见 11.2.1 节, 第 377 页) 来存放交易信息, 这样我们就能保存同一本书的多条交易记录, 而且对于一本给定的书籍, 它的所有交易信息都保存在一起 (参见 11.2.2 节, 第 378 页)。

`multiset` 的元素是 `shared_ptr`。因为 `shared_ptr` 没有定义小于运算符, 所以为了对元素排序我们必须提供自己的比较运算符 (参见 11.2.2 节, 第 378 页)。在此例中, 我们定义了一个名为 `compare` 的私有静态成员, 该成员负责比较 `shared_ptr` 所指的对象的 `isbn`。我们初始化 `multiset`, 通过类内初始值调用比较函数 (参见 7.3.1 节, 第 246 页):

632 // multiset 保存多个报价, 按照 compare 成员排序

```
std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
    items{compare};
```

这个声明看起来不太容易理解, 但是从左向右读的话, 我们就能明白它其实是定义了一个指向 `Quote` 对象的 `shared_ptr` 的 `multiset`。这个 `multiset` 将使用一个与 `compare` 成员类型相同的函数来对其中的元素进行排序。`multiset` 成员的名字是 `items`, 我们初始化 `items` 并令其使用我们的 `compare` 函数。

定义 Basket 的成员

`Basket` 类只定义两个操作。第一个成员是我们在类的内部定义的 `add_item` 成员, 该成员接受一个指向动态分配的 `Quote` 的 `shared_ptr`, 然后将这个 `shared_ptr` 放置在 `multiset` 中。第二个成员的名字是 `total_receipt`, 它负责将购物篮的内容逐项打印成清单, 然后返回购物篮中所有物品的总价格:

```
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0; // 保存实时计算出的总价格
    // iter 指向 ISBN 相同的一批元素中的第一个
    // upper_bound 返回一个迭代器, 该迭代器指向这批元素的尾后位置
    for (auto iter = items.cbegin();
        iter != items.cend();
        iter = items.upper_bound(*iter)) {
        // 我们知道在当前的 Basket 中至少有一个该关键字的元素
        // 打印该书籍对应的项目
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl; // 打印最终的总价格
    return sum;
}
```

我们的 `for` 循环首先定义并初始化 `iter`, 令其指向 `multiset` 的第一个元素。条件部分检查 `iter` 是否等于 `items.cend()`: 如果相等, 表明我们已经处理完了所有购买记录, 接下来应该跳出 `for` 循环; 否则, 如果不相等, 则继续处理下一本书籍。

比较有趣的是, `for` 循环中的“递增”表达式。与通常的循环语句依次读取每个元素不同, 我们直接令 `iter` 指向下一个关键字, 调用 `upper_bound` 函数可以令我们跳过与当前关键字相同的所有元素 (参见 11.3.5 节, 第 390 页)。对于 `upper_bound` 函数来说, 它返回的是一个迭代器, 该迭代器指向所有与 `iter` 关键字相等的元素中最后一个元素的

下一位置。因此，我们得到的迭代器或者指向集合的末尾，或者指向下一本书籍。

在 `for` 循环内部，我们通过调用 `print_total`（参见 15.1 节，第 527 页）来打印购物篮中每本书籍的细节：

```
sum += print_total(os, **iter, items.count(*iter));
```

`print_total` 的实参包括一个用于写入数据的 `ostream`、一个待处理的 `Quote` 对象和一个计数值。当我们解引用 `iter` 后将得到一个指向准备打印的对象的 `shared_ptr`。为了得到这个对象，必须解引用该 `shared_ptr`。因此，`**iter` 是一个 `Quote` 对象（或者 `Quote` 的派生类的对象）。我们使用 `multiset` 的 `count` 成员（参见 11.3.5 节，第 388 页）来统计在 `multiset` 中有多少元素的键值相同（即 ISBN 相同）。 633

如我们所知，`print_total` 调用了虚函数 `net_price`，因此最终的计算结果依赖于 `**iter` 的动态类型。`print_total` 函数打印并返回给定书籍的总价格，我们把这个结果添加到 `sum` 当中，最后当循环结束后打印 `sum`。

隐藏指针

`Basket` 的用户仍然必须处理动态内存，原因是 `add_item` 需要接受一个 `shared_ptr` 参数。因此，用户不得不按照如下形式编写代码：

```
Basket bsk;
bsk.add_item(make_shared<Quote>("123", 45));
bsk.add_item(make_shared<Bulk_quote>("345", 45, 3, .15));
```

我们的下一步是重新定义 `add_item`，使得它接受一个 `Quote` 对象而非 `shared_ptr`。新版本的 `add_item` 将负责处理内存分配，这样它的用户就不必再受困于此了。我们将定义两个版本，一个拷贝它给定的对象，另一个则采取移动操作（参见 13.6.3 节，第 481 页）：

```
void add_item(const Quote& sale);           // 拷贝给定的对象
void add_item(Quote&& sale);                // 移动给定的对象
```

唯一的问题是 `add_item` 不知道要分配的类型。当 `add_item` 进行内存分配时，它将拷贝（或移动）它的 `sale` 参数。在某处可能会有一条如下形式的 `new` 表达式：

```
new Quote(sale)
```

不幸的是，这条表达式所做的工作可能是不正确的：`new` 为我们请求的类型分配内存，因此这条表达式将分配一个 `Quote` 类型的对象并且拷贝 `sale` 的 `Quote` 部分。然而，`sale` 实际指向的可能是 `Bulk_quote` 对象，此时，该对象将被迫切掉一部分。

模拟虚拷贝



为了解决上述问题，我们给 `Quote` 类添加一个虚函数，该函数将申请一份当前对象的拷贝。

```
class Quote {
public:
    // 该虚函数返回当前对象的一份动态分配的拷贝
    // 这些成员使用的引用限定符参见 13.6.3 节（第 483 页）
    virtual Quote* clone() const & {return new Quote(*this);}
    virtual Quote* clone() &&
        {return new Quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

634 >

```
class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
        {return new Bulk_quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

因为我们拥有 `add_item` 的拷贝和移动版本，所以我们分别定义 `clone` 的左值和右值版本(参见 13.6.3 节, 第 483 页)。每个 `clone` 函数分配当前类型的一个新对象，其中，`const` 左值引用成员将它自己拷贝给新分配的对象；右值引用成员则将自己移动到新数据中。

我们可以使用 `clone` 很容易地写出新版本的 `add_item`:

```
class Basket {
public:
    void add_item(const Quote& sale)      // 拷贝给定的对象
    { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale)           // 移动给定的对象
    { items.insert(
        std::shared_ptr<Quote>(std::move(sale).clone())); }
    // 其他成员与之前的版本一致
};
```

和 `add_item` 本身一样，`clone` 函数也根据作用于左值还是右值而分为不同的重载版本。在此例中，第一个 `add_item` 函数调用 `clone` 的 `const` 左值版本，第二个函数调用 `clone` 的右值引用版本。在右值版本中，尽管 `sale` 的类型是右值引用类型，但实际上 `sale` 本身(和任何其他变量一样) 是个左值(参见 13.6.1 节, 第 471 页)。因此，我们调用 `move` 把一个右值引用绑定到 `sale` 上。

我们的 `clone` 函数也是一个虚函数。`sale` 的动态类型(通常)决定了到底运行 `Quote` 的函数还是 `Bulk_quote` 的函数。无论我们是拷贝还是移动数据，`clone` 都返回一个新分配对象的指针，该对象与 `clone` 所属的类型一致。我们把一个 `shared_ptr` 绑定到这个对象上，然后调用 `insert` 将这个新分配的对象添加到 `items` 中。注意，因为 `shared_ptr` 支持派生类向基类的类型转换(参见 15.2.2 节, 第 530 页)，所以我们将把 `shared_ptr<Quote>` 绑定到 `Bulk_quote*` 上。

15.8.1 节练习

练习 15.30: 编写你自己的 `Basket` 类，用它计算上一个练习中交易记录的总价格。

15.9 文本查询程序再探

接下来，我们扩展 12.3 节(第 430 页)的文本查询程序，用它作为说明继承的最后一个例子。在上一版的程序中，我们可以查询在文件中某个指定单词的出现情况。我们将在本节扩展该程序使其支持更多更复杂的查询操作。在后面的例子中，我们将针对下面这个小故事展开查询：

```
Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
```

```

A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

```

我们的系统将支持如下查询形式。

- 单词查询，用于得到匹配某个给定 string 的所有行：

```

Executing Query for: Daddy
Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"

```

- 逻辑非查询，使用~运算符得到不匹配查询条件的所有行：

```

Executing Query for: ~(Alice)
~(Alice) occurs 9 times
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.

...

```

- 逻辑或查询，使用 | 运算符返回匹配两个条件中任意一个的行：

```

Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,

```

- 逻辑与查询，使用 & 运算符返回匹配全部两个条件的行：

```

Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.

```

此外，我们还希望能够混合使用这些运算符，比如：

```
fiery & bird | wind
```

在类似这样的例子中，我们将使用 C++ 通用的优先级规则（参见 4.1.2 节，第 121 页）对复杂表达式求值。因此，这条查询语句所得行应该是如下二者之一：在该行中或者 `fiery` 和 `bird` 同时出现，或者出现了 `wind`：

```

Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,

```

636

在输出内容中首先是那条查询语句，我们使用圆括号来表示查询被解释和执行的次序。与之前实现的版本一样，接下来系统将按照查询结果中行号的升序显示结果并且每一行只显示一次。

15.9.1 面向对象的解决方案

我们可能会认为使用 12.3.2 节（第 432 页）的 `TextQuery` 类来表示单词查询，然后

从该类中派生出其他查询是一种可行的方案。

然而，这样的设计实际上存在缺陷。为了理解其中的原因，我们不妨考虑逻辑非查询。单词查询查找一个指定的单词，为了让逻辑非查询按照单词查询的方式执行，我们将不得不定义逻辑非查询所要查找的单词。但是在一般情况下，我们无法得到这样的单词。相反，一个逻辑非查询中含有一个结果值需要取反的查询语句（单词查询或任何其他查询）；类似的，一个逻辑与查询和一个逻辑或查询各包含两个结果值需要合并的查询语句。

由上述观察结果可知，我们应该将几种不同的查询建模成相互独立的类，这些类共享一个公共基类：

```
WordQuery      // Daddy
NotQuery       // ~Alice
OrQuery        // hair | Alice
AndQuery       // hair & Alice
```

这些类将只包含两个操作：

- eval，接受一个 `TextQuery` 对象并返回一个 `QueryResult`，`eval` 函数使用给定的 `TextQuery` 对象查找与之匹配的行。
- rep，返回基础查询的 `string` 表示形式，`eval` 函数使用 `rep` 创建一个表示匹配结果的 `QueryResult`，输出运算符使用 `rep` 打印查询表达式。

关键概念：继承与组合

继承体系的设计本身是一个非常复杂的问题，已经超出了本书的范围。然而，有一条设计准则非常重要也非常基础，每个程序员都应该熟悉它。

当我们令一个类公有地继承另一个类时，派生类应当反映与基类的“是一种 (Is A)”关系。在设计良好的类体系当中，公有派生类的对象应该可以用在任何需要基类对象的地方。

类型之间的另一种常见关系是“有一个 (Has A)”关系，具有这种关系的类暗含成员的意思。

在我们的书店示例中，基类表示的是按规定价格销售的书籍的报价。`Bulk_quote` “是一种” 报价结果，只不过它使用的价格策略不同。我们的书店类都“有一个” 价格成员和 `ISBN` 成员。

抽象基类

如我们所知，在这四种查询之间并不存在彼此的继承关系，从概念上来说它们互为兄弟。因为所有这些类都共享同一个接口，所以我们需要定义一个抽象基类（参见 15.4 节，第 541 页）来表示该接口。我们将所需的抽象基类命名为 `Query_base`，以此来表示它的角色是整个查询继承体系的根节点。

我们的 `Query_base` 类将把 `eval` 和 `rep` 定义成纯虚函数（参见 15.4 节，第 541 页），其他代表某种特定查询类型的类必须覆盖这两个函数。我们将从 `Query_base` 直接派生出 `WordQuery` 和 `NotQuery`。`AndQuery` 和 `OrQuery` 都具有系统中其他类所不具备的一个特殊属性：它们各自包含两个运算对象。为了对这种属性建模，我们定义另外一个名为 `BinaryQuery` 的抽象基类，该抽象基类用于表示含有两个运算对象的查询。`AndQuery` 和 `OrQuery` 继承自 `BinaryQuery`，而 `BinaryQuery` 继承自 `Query_base`。由这些分

析我们将得到如图 15.2 所示的类设计结果：

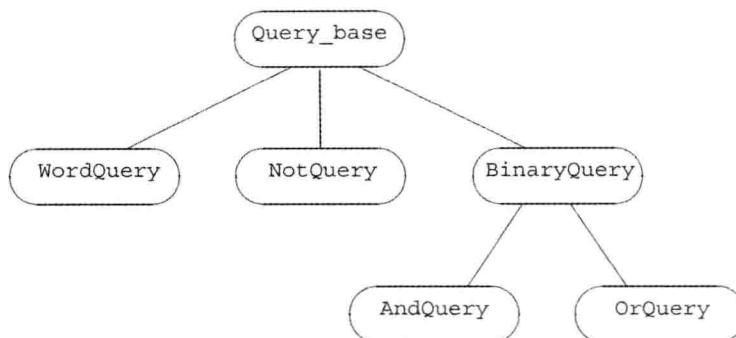


图 15.2: Query_base 继承体系

将层次关系隐藏于接口类中

我们的程序将致力于计算查询结果，而非仅仅构建查询的体系。为了使程序能正常运行，我们必须首先创建查询命令，最简单的办法是编写 C++ 表达式。例如，可以编写下面的代码来生成之前描述的复合查询：

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

如上所述，其隐含的意思是用户层代码将不会直接使用这些继承的类；相反，我们将定义一个名为 `Query` 的接口类，由它负责隐藏整个继承体系。`Query` 类将保存一个 `Query_base` 指针，该指针绑定到 `Query_base` 的派生类对象上。`Query` 类与 `Query_base` 类提供的操作是相同的：`eval` 用于求查询的结果，`rep` 用于生成查询的 `string` 版本，同时 `Query` 也会定义一个重载的输出运算符用于显示查询。

用户将通过 `Query` 对象的操作间接地创建并处理 `Query_base` 对象。我们定义 `Query` 对象的三个重载运算符以及一个接受 `string` 参数的 `Query` 构造函数，这些函数动态分配一个新的 `Query_base` 派生类的对象：

- & 运算符生成一个绑定到新的 `AndQuery` 对象上的 `Query` 对象；
- | 运算符生成一个绑定到新的 `OrQuery` 对象上的 `Query` 对象；
- ~ 运算符生成一个绑定到新的 `NotQuery` 对象上的 `Query` 对象；
- 接受 `string` 参数的 `Query` 构造函数生成一个新的 `WordQuery` 对象。

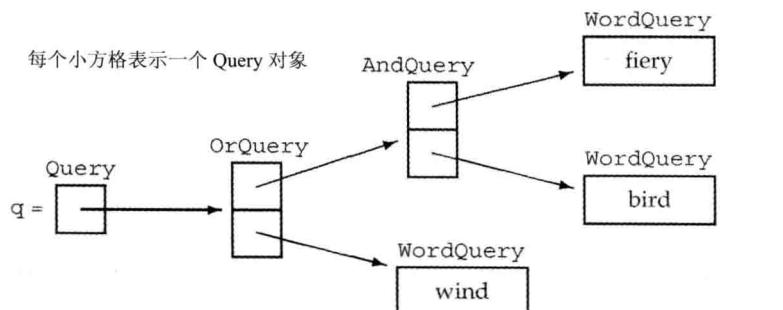


图 15.3: 使用 Query 表达式创建的对象

理解这些类的工作机理

在这个应用程序中，很大一部分工作是构建代表用户查询的对象，对于读者来说认识到这一点非常重要。例如，像上面这样的表达式将生成如图 15.3 所示的一系列相关对象的集合。

一旦对象树构建完成后，对某一条查询语句的求值（或生成表示形式的）过程基本上就转换为沿着箭头方向依次对每个对象求值（或显示）的过程（由编译器为我们组织管理）。

639 例如，如果我们对 q（即树的根节点）调用 eval 函数，则该调用语句将令 q 所指的 OrQuery 对象 eval 它自己。对该 OrQuery 求值实际上是对它的两个运算对象执行 eval 操作：一个运算对象是 AndQuery，另一个是查找单词 wind 的 WordQuery。接下来，对 AndQuery 求值转化为对它的两个 WordQuery 求值，分别生成单词 fiery 和 bird 的查询结果。

对于面向对象编程的新手来说，要想理解一个程序，最困难的部分往往是理解程序的设计思路。一旦你掌握了程序的设计思路，接下来的实现也就水到渠成了。为了帮助读者理解程序设计的过程，我们在表 15.1 中整理了之前那个例子用到的类，并对其进行了简要的描述。

640

表 15.1：概述：Query 程序设计

Query 程序接口类和操作	
TextQuery	该类读入给定的文件并构建一个查找图。这个类包含一个 query 操作，它接受一个 string 实参，返回一个 QueryResult 对象；该 QueryResult 对象表示 string 出现的行（12.3.2 节，第 432 页）
QueryResult	该类保存一个 query 操作的结果（12.3.2 节，第 433 页）
Query	是一个接口类，指向 Query_base 派生类的对象
Query q(s)	将 Query 对象 q 绑定到一个存放着 string s 的新 WordQuery 对象上
q1 & q2	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 AndQuery 对象上
q1 q2	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 OrQuery 对象上
~q	返回一个 Query 对象，该 Query 绑定到一个存放 q 的新 NotQuery 对象上
Query 程序实现类	
Query_base	查询类的抽象基类
WordQuery	Query_base 的派生类，用于查找一个给定的单词
NotQuery	Query_base 的派生类，查询结果是 Query 运算对象没有出现的行的集合
BinaryQuery	Query_base 派生出来的另一个抽象基类，表示有两个运算对象的查询
OrQuery	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的并集
AndQuery	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的交集

15.9.1 节练习

练习 15.31：已知 s1、s2、s3 和 s4 都是 string，判断下面的表达式分别创建了什么样的对象：

- (a) Query(s1) | Query(s2) & ~ Query(s3);
- (b) Query(s1) | (Query(s2) & ~ Query(s3));
- (c) (Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));

15.9.2 Query_base 类和 Query 类

下面我们开始程序的实现过程，首先定义 `Query_base` 类：

```
// 这是一个抽象基类，具体的查询类型从中派生，所有成员都是 private 的
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no; // 用于 eval 函数
    virtual ~Query_base() = default;
private:
    // eval 返回与当前 Query 匹配的 QueryResult
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep 是表示查询的一个 string
    virtual std::string rep() const = 0;
};
```

`eval` 和 `rep` 都是纯虚函数，因此 `Query_base` 是一个抽象基类（参见 15.4 节，第 541 页）。因为我们不希望用户或者派生类直接使用 `Query_base`，所以它没有 `public` 成员。所有对 `Query_base` 的使用都需要通过 `Query` 对象，因为 `Query` 需要调用 `Query_base` 的虚函数，所以我们将 `Query` 声明成 `Query_base` 的友元。

受保护的成员 `line_no` 将在 `eval` 函数内部使用。类似的，析构函数也是受保护的，因为它将（隐式地）在派生类析构函数中使用。

Query 类

`Query` 类对外提供接口，同时隐藏了 `Query_base` 的继承体系。每个 `Query` 对象都含有一个指向 `Query_base` 对象的 `shared_ptr`。因为 `Query` 是 `Query_base` 的唯一接口，所以 `Query` 必须定义自己的 `eval` 和 `rep` 版本。

接受一个 `string` 参数的 `Query` 构造函数将创建一个新的 `WordQuery` 对象，然后将它的 `shared_ptr` 成员绑定到这个新创建的对象上。`&`、`|` 和 `~` 运算符分别创建 `AndQuery`、`OrQuery` 和 `NotQuery` 对象，这些运算符将返回一个绑定到新创建的对象上的 `Query` 对象。为了支持这些运算符，`Query` 还需要另外一个构造函数，它接受指向 `Query_base` 的 `shared_ptr` 并且存储给定的指针。我们将这个构造函数声明为私有的，原因是不希望一般的用户代码能随便定义 `Query_base` 对象。因为这个构造函数是私有的，所以我们需要将三个运算符声明为友元。

在形成了上述设计思路后，`Query` 类本身就比较简单了：

```
// 这是一个管理 Query_base 继承体系的接口类
class Query {
    // 这些运算符需要访问接受 shared_ptr 的构造函数，而该函数是私有的
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const std::string&); // 构建一个新的 WordQuery
    // 接口函数：调用对应的 Query_base 操作
    QueryResult eval(const TextQuery &t) const
        { return q->eval(t); }
    std::string rep() const { return q->rep(); }
```

```

private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};

```

我们首先将创建 `Query` 对象的运算符声明为友元，之所以这么做是因为这些运算符需要访问那个私有构造函数。

在 `Query` 的公有接口部分，我们声明了接受 `string` 的构造函数，不过没有对其进行定义。因为这个构造函数将要创建一个 `WordQuery` 对象，所以我们应该首先定义 `WordQuery` 类，随后才能定义接受 `string` 的 `Query` 构造函数。

另外两个公有成员是 `Query_base` 的接口。其中，`Query` 操作使用它的 `Query_base` 指针来调用各自的 `Query_base` 虚函数。实际调用哪个函数版本将由 `q` 所指的对象类型决定，并且直到运行时才能最终确定下来。



Query 的输出运算符

输出运算符可以很好地解释我们的整个查询系统是如何工作的：

```

std::ostream &
operator<<(std::ostream &os, const Query &query)
{
    // Query::rep 通过它的 Query_base 指针对 rep() 进行了虚调用
    return os << query.rep();
}

```

当我们打印一个 `Query` 时，输出运算符调用 `Query` 类的公有 `rep` 成员。运算符函数通过指针成员虚调用当前 `Query` 所指对象的 `rep` 成员。也就是说，当我们编写如下代码时：

```

Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;

```

输出运算符将调用 `andq` 的 `Query::rep`，而 `Query::rep` 通过它的 `Query_base` 指针虚调用 `Query_base` 版本的 `rep` 函数。因为 `andq` 指向的是一个 `AndQuery` 对象，所以本次的函数调用将运行 `AndQuery::rep`。

15.9.2 节练习

练习 15.32: 当一个 `Query` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

练习 15.33: 当一个 `Query_base` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

642 >

15.9.3 派生类

对于 `Query_base` 的派生类来说，最有趣的部分是这些派生类如何表示一个真实的查询。其中 `WordQuery` 类最直接，它的任务就是保存要查找的单词。

其他类分别操作一个或两个运算对象。`NotQuery` 有一个运算对象，`AndQuery` 和 `OrQuery` 有两个。在这些类当中，运算对象可以是 `Query_base` 的任意一个派生类的对象：一个 `NotQuery` 对象可以被用在 `WordQuery`、`AndQuery`、`OrQuery` 或另一个 `NotQuery` 中。为了支持这种灵活性，运算对象必须以 `Query_base` 指针的形式存储，

这样我们就能把该指针绑定到任何我们需要的具体类上。

然而，实际上我们的类并不存储 `Query_base` 指针，而是直接使用一个 `Query` 对象。就像用户代码可以通过接口类得到简化一样，我们也可以使用接口类来简化我们自己的类。

至此我们已经清楚了所有类的设计思路，接下来依次实现它们。

WordQuery 类

一个 `WordQuery` 查找一个给定的 `string`，它是在给定的 `TextQuery` 对象上实际执行查询的唯一一个操作：

```
class WordQuery: public Query_base {
    friend class Query; // Query 使用 WordQuery 构造函数
    WordQuery(const std::string &s): query_word(s) { }
    // 具体的类：WordQuery 将定义所有继承而来的纯虚函数
    QueryResult eval(const TextQuery &t) const
        { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word; // 要查找的单词
};
```

和 `Query_base` 一样，`WordQuery` 没有公有成员。同时，`Query` 必须作为 `WordQuery` 的友元，这样 `Query` 才能访问 `WordQuery` 的构造函数。

每个表示具体查询的类都必须定义继承而来的纯虚函数 `eval` 和 `rep`。我们在 `WordQuery` 类的内部定义这两个操作：`eval` 调用其 `TextQuery` 参数的 `query` 成员，由 `query` 成员在文件中实际进行查找；`rep` 返回这个 `WordQuery` 表示的 `string`（即 `query_word`）。

定义了 `WordQuery` 类之后，我们就能定义接受 `string` 的 `Query` 构造函数了：

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

这个构造函数分配一个 `WordQuery`，然后令其指针成员指向新分配的对象。

NotQuery 类及~运算符

`~` 运算符生成一个 `NotQuery`，其中保存着一个需要对其取反的 `Query`：

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // 具体的类：NotQuery 将定义所有继承而来的纯虚函数
    std::string rep() const { return "~(" + query.rep() + ")"; }
    QueryResult eval(const TextQuery&) const;
    Query query;
};
inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}
```

643

因为 `NotQuery` 的所有成员都是私有的，所以我们一开始就要把`~`运算符设定为友元。为

了 rep 一个 NotQuery，我们需要将~符号与基础的 Query 连接在一起。我们在输出的结果中加上适当的括号，这样读者就可以清楚地知道查询的优先级了。

值得注意的是，在 NotQuery 自己的 rep 成员中对 rep 的调用最终执行的是一个虚调用：query.rep() 是对 Query 类 rep 成员的非虚调用，接着 Query::rep 将调用 q->rep()，这是一个通过 Query_base 指针进行的虚调用。

~运算符动态分配一个新的 NotQuery 对象，其 return 语句隐式地使用接受一个 shared_ptr<Query_base> 的 Query 构造函数。也就是说，return 语句等价于：

```
// 分配一个新的 NotQuery 对象
// 将所得的 NotQuery 指针绑定到一个 shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp);           // 使用接受一个 shared_ptr 的 Query 构造函数
```

eval 成员比较复杂，因此我们将在类的外部实现它，15.9.4 节（第 573 页）将专门介绍如何定义 eval 函数。

BinaryQuery 类

BinaryQuery 类也是一个抽象基类，它保存操作两个运算对象的查询类型所需的数据：

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
        lhs(l), rhs(r), opSym(s) { }
    // 抽象类: BinaryQuery 不定义 eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                + opSym + " "
                                + rhs.rep() + ")"; }
    Query lhs, rhs;           // 左侧和右侧运算对象
    std::string opSym;        // 运算符的名字
};
```

644 BinaryQuery 中的数据是两个运算对象及相应的运算符符号，构造函数负责接受两个运算对象和一个运算符符号，然后将它们存储在对应的数据成员中。

要想 rep 一个 BinaryQuery，我们需要生成一个带括号的表达式。表达式的内容依次包括左侧运算对象、运算符以及右侧运算对象。就像我们显示 NotQuery 的方法一样，对 rep 的调用最终是对 lhs 和 rhs 所指 Query_base 对象的 rep 函数进行虚调用。



BinaryQuery 不定义 eval，而是继承了该纯虚函数。因此，BinaryQuery 也是一个抽象基类，我们不能创建 BinaryQuery 类型的对象。

AndQuery 类、OrQuery 类及相应的运算符

AndQuery 类和 OrQuery 类以及它们的运算符都非常相似：

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "&") { }
    // 具体的类: AndQuery 继承了 rep 并且定义了其他纯虚函数
    QueryResult eval(const TextQuery&) const;
```

```

};

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}

```

这两个类将各自的运算符定义成友元，并且各自定义了一个构造函数通过运算符创建 BinaryQuery 基类部分。它们继承 BinaryQuery 的 rep 函数，但是覆盖了 eval 函数。

和~运算符一样，&和|运算符也返回一个绑定到新分配对象上的 shared_ptr。在这些运算符中，return 语句负责将 shared_ptr 转换成 Query。

15.9.3 节练习

645

练习 15.34: 针对图 15.3（第 565 页）构建的表达式：

- 列举出在处理表达式的过程中执行的所有构造函数。
- 列举出 cout<<q 所调用的 rep。
- 列举出 q.eval() 所调用的 eval。

练习 15.35: 实现 Query 类和 Query_base 类，其中需要定义 rep 而无须定义 eval。

练习 15.36: 在构造函数和 rep 成员中添加打印语句，运行你的代码以检验你对本节第一个练习中 (a)、(b) 两小题的回答是否正确。

练习 15.37: 如果在派生类中含有 shared_ptr<Query_base>类型的成员而非 Query 类型的成员，则你的类需要做出怎样的改变？

练习 15.38: 下面的声明合法吗？如果不合法，请解释原因；如果合法，请指出该声明的含义。

```

BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");

```

15.9.4 eval 函数

eval 函数是我们这个查询系统的核心。每个 eval 函数作用于各自的运算对象，同时遵循的内在逻辑也有所区别：OrQuery 的 eval 操作返回两个运算对象查询结果的并集，而 AndQuery 返回交集。与它们相比，NotQuery 的 eval 函数更加复杂一些：它需要返回运算对象没有出现的文本行。

为了支持上述 eval 函数的处理，我们需要使用 QueryResult，在它当中定义了 12.3.2 节练习（第 435 页）添加的成员。假设 QueryResult 包含 begin 和 end 成员，它们允许我们在 QueryResult 保存的行号 set 中进行迭代；另外假设 QueryResult 还包含一个名为 get_file 的成员，它返回一个指向待查询文件的 shared_ptr。



我们的 Query 类使用了 12.3.2 节练习(第 435 页)为 QueryResult 定义的成员。

OrQuery::eval

一个 OrQuery 表示的是它的两个运算对象结果的并集，对于每个运算对象来说，我们通过调用 eval 得到它的查询结果。因为这些运算对象的类型是 Query，所以调用 eval 也就是调用 Query::eval，而后者实际上是对潜在的 query_base 对象的 eval 进行虚调用。每次调用完成后，得到的结果是一个 QueryResult，它表示运算对象出现的行号。我们把这些行号组织在一个新 set 中：

646 >

```
// 返回运算对象查询结果 set 的并集
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // 通过 Query 成员 lhs 和 rhs 进行的虚调用
    // 调用 eval 返回每个运算对象的 QueryResult
    auto right = rhs.eval(text), left = lhs.eval(text);
    // 将左侧运算对象的行号拷贝到结果 set 中
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(), left.end());
    // 插入右侧运算对象所得的行号
    ret_lines->insert(right.begin(), right.end());
    // 返回一个新的 QueryResult，它表示 lhs 和 rhs 的并集
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

我们使用接受一对迭代器的 set 构造函数初始化 ret_lines。一个 QueryResult 的 begin 和 end 成员返回行号 set 的迭代器，因此，创建 ret_lines 的过程实际上是拷贝了 left 集合的元素。接下来对 ret_lines 调用 insert，并将 right 的元素插入进来。调用结束后，ret_lines 将包含在 left 或 right 中出现过的所有行号。

eval 函数在最后构建并返回一个表示混合查询匹配的 QueryResult。QueryResult 的构造函数（参见 12.3.2 节，第 434 页）接受三个实参：一个表示查询的 string、一个指向匹配行号 set 的 shared_ptr 和一个指向输入文件 vector 的 shared_ptr。我们调用 rep 生成所需的 string，调用 get_file 获取指向文件的 shared_ptr。因为 left 和 right 指向的是同一个文件，所以使用哪个执行 get_file 函数并不重要。

AndQuery::eval

AndQuery 的 eval 和 OrQuery 很类似，唯一的区别是它调用了一个标准库算法来求得两个查询结果中共有的行：

```
// 返回运算对象查询结果 set 的交集
QueryResult
AndQuery::eval(const TextQuery& text) const
{
```

```

// 通过 Query 运算对象进行的虚调用，以获得运算对象的查询结果 set
auto left = lhs.eval(text), right = rhs.eval(text);
// 保存 left 和 right 交集的 set
auto ret_lines = make_shared<set<line_no>>();
// 将两个范围的交集写入一个目的迭代器中
// 本次调用的目的迭代器向 ret 添加元素
set_intersection(left.begin(), left.end(),
                 right.begin(), right.end(),
                 inserter(*ret_lines, ret_lines->begin()));
return QueryResult(rep(), ret_lines, left.get_file());
}

```

其中我们使用标准库算法 `set_intersection` 来合并两个 `set`，关于 647
`set_intersection` 在附录 A.2.8 (第 779 页) 中有详细的描述。

`set_intersection` 算法接受五个迭代器。它使用前四个迭代器表示两个输入序列 (参见 10.5.2 节, 第 368 页), 最后一个实参表示目的位置。该算法将两个输入序列中共同出现的元素写入到目的位置中。

在上述调用中我们传入一个插入迭代器 (参见 10.4.1 节, 第 357 页) 作为目的位置。当 `set_intersection` 向这个迭代器写入内容时, 实际上是向 `ret_lines` 插入一个新元素。

和 `OrQuery` 的 `eval` 函数一样, `AndQuery` 的 `eval` 函数也在最后构建并返回一个表示混合查询匹配的 `QueryResult`。

NotQuery::eval

`NotQuery` 查找运算对象没有出现的文本行:

```

// 返回运算对象的结果 set 中不存在的行
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // 通过 Query 运算对象对 eval 进行虚调用
    auto result = query.eval(text);
    // 开始时结果 set 为空
    auto ret_lines = make_shared<set<line_no>>();
    // 我们必须在运算对象出现的所有行中进行迭代
    auto beg = result.begin(), end = result.end();
    // 对于输入文件的每一行, 如果该行不在 result 当中, 则将其添加到 ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // 如果我们还没有处理完 result 的所有行
        // 检查当前行是否存在
        if (beg == end || *beg != n)
            ret_lines->insert(n);      // 如果不在 result 当中, 添加这一行
        else if (beg != end)
            ++beg;                  // 否则继续获取 result 的下一行 (如果有的话)
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}

```

和其他 `eval` 函数一样, 我们首先对当前的运算对象调用 `eval`, 所得的结果

`QueryResult` 中包含的是运算对象出现的行号，但我们想要的是运算对象未出现的行号。也就是说，我们需要的是存在于文件中，但是不在 `result` 中的行。

要想得到最终的结果，我们需要遍历不超过输出文件大小的所有整数，并将所有不在 `result` 中的行号放入到 `ret_lines` 中。我们使用 `beg` 和 `end` 分别表示 `result` 的第一个元素和最后一个元素的下一位置。因为遍历的对象是一个 `set`，所以当遍历结束后获得的行号将按照升序排列。

648 循环体负责检查当前的编号是否在 `result` 当中。如果不在，将这个数字添加到 `ret_lines` 中；如果该数字属于 `result`，则我们递增 `result` 的迭代器 `beg`。

一旦处理完所有行号，就返回包含 `ret_lines` 的一个 `QueryResult` 对象；和之前版本的 `eval` 类似，该 `QueryResult` 对象还包含 `rep` 和 `get_file` 的运行结果。

15.9.4 节练习

练习 15.39：实现 `Query` 类和 `Query_base` 类，求图 15.3（第 565 页）中表达式的值并打印相关信息，验证你的程序是否正确。

练习 15.40：在 `OrQuery` 的 `eval` 函数中，如果 `rhs` 成员返回的是空集将发生什么？如果 `lhs` 是空集呢？如果 `lhs` 和 `rhs` 都是空集又将发生什么？

练习 15.41：重新实现你的类，这次使用指向 `Query_base` 的内置指针而非 `shared_ptr`。请注意，做出上述改动后你的类将不能再使用合成的拷贝控制成员。

练习 15.42：从下面的几种改进中选择一种，设计并实现它：

- (a) 按句子查询并打印单词，而不再是按行打印。
- (b) 引入一个历史系统，用户可以按编号查阅之前的某个查询，并可以在其中增加内容或者将其与其他查询组合。
- (c) 允许用户对结果做出限制，比如从给定范围的行中挑出匹配的进行显示。

小结

< 649

继承使得我们可以编写一些新的类，这些新类既能共享其基类的行为，又能根据需要覆盖或添加行为。动态绑定使得我们可以忽略类型之间的差异，其机理是在运行时根据对象的动态类型来选择运行函数的那个版本。继承和动态绑定的结合使得我们能够编写具有特定类型行为但又独立于类型的程序。

在 C++ 语言中，动态绑定只作用于虚函数，并且需要通过指针或引用调用。

在派生类对象中包含有与它的每个基类对应的子对象。因为所有派生类对象都含有基类部分，所以我们能将派生类的引用或指针转换为一个可访问的基类引用或指针。

当执行派生类的构造、拷贝、移动和赋值操作时，首先构造、拷贝、移动和赋值其中的基类部分，然后才轮到派生类部分。析构函数的执行顺序则正好相反，首先销毁派生类，接下来执行基类子对象的析构函数。基类通常都应该定义一个虚析构函数，即使基类根本不需要析构函数也最好这么做。将基类的析构函数定义成虚函数的原因是为了确保当我们删除一个基类指针，而该指针实际指向一个派生类对象时，程序也能正确运行。

派生类为它的每个基类提供一个保护级别。`public` 基类的成员也是派生类接口的一部分；`private` 基类的成员是不可访问的；`protected` 基类的成员对于派生类的派生类是可访问的，但是对于派生类的用户不可访问。

术语表

抽象基类（abstract base class） 含有一个或多个纯虚函数的类，我们无法创建抽象基类的对象。

可访问的（accessible） 能被派生类对象访问的基类成员。可访问性由派生类的派生列表中所用的访问说明符和基类中成员的访问级别共同决定。例如，通过公有继承而来的一个公有成员对于派生类的用户来说是可访问的；而私有继承而来的公有成员是不可访问的。

基类（base class） 可供其他类继承的类。基类的成员也将成为派生类的成员。

类派生列表（class derivation list） 罗列了所有基类，每个基类包含一个可选的访问级别，它定义了派生类继承该基类的方式。如果没有提供访问说明符，则当派生类通过关键字 `struct` 定义时继承是公有的；而当派生类通过关键字 `class` 定义时继承是私有的。

派生类（derived class） 从其他类派生而

来的类。派生类可以覆盖其基类的虚函数，也可以定义自己的新成员。派生类的作用域嵌套在基类作用域当中；派生类的成员能直接访问基类的成员。

派生类向基类的类型转换（derived-to-base conversion） 派生类对象向基类引用或者派生类指针向基类指针的隐式类型转换。

直接基类（direct base class） 派生类直接继承的基类，直接基类在派生类的派生列表中说明。直接基类本身也可以是一个派生类。

动态绑定（dynamic binding） 直到运行时才确定到底执行函数的哪个版本。在 C++ 语言中，动态绑定的意思是在运行时根据引用或指针所绑定对象的实际类型来选择执行虚函数的某一个版本。

动态类型（dynamic type） 对象在运行时的类型。引用所引对象或者指针所指对象的动态类型可能与该引用或指针的静态类型不同。基类的指针或引用可以指向一个

< 650

派生类对象。在这样的情况中，静态类型是基类的引用（或指针），而动态类型是派生类的引用（或指针）。

间接基类 (indirect base class) 不出现在派生类的派生列表中的基类。直接基类以直接或间接方式继承的类是派生类的间接基类。

继承 (inheritance) 由一个已有的类（基类）定义一个新类（派生类）的编程技术。派生类将继承基类的成员。

面向对象编程 (object-oriented programming) 利用数据抽象、继承以及动态绑定等技术编写程序的方法。

覆盖 (override) 派生类中定义的虚函数如果与基类中定义的同名虚函数有相同的形参列表，则派生类版本将覆盖基类的版本。

多态性 (polymorphism) 当用于面向对象编程的范畴时，多态性的含义是指程序能通过引用或指针的动态类型获取类型特定行为的能力。

私有继承 (private inheritance) 在私有继承中，基类的公有成员和受保护成员是派生类的私有成员。

protected 访问说明符 (protected access specifier) `protected` 关键字之后定义的成员能被派生类的成员和友元访问。但是这些成员只对派生类对象是可访问的，对类的普通用户则是不可访问的。

受保护的继承 (protected inheritance) 在受保护的继承中，基类的公有成员和受保护成员是派生类的受保护成员。

公有继承 (public inheritance) 基类的公有接口是派生类公有接口的组成部分。

纯虚函数 (pure virtual) 在类的内部声明虚函数时，在分号之前使用了`=0`。一个纯虚函数不需要（但是可以）被定义。含有纯虚函数的类是抽象基类。如果派生类没有对继承而来的纯虚函数定义自己的版本，则该派生类也是抽象的。

重构 (refactoring) 重新设计程序以便将一些相关的部分搜集到一个单独的抽象中，然后使用新的抽象替换原来的代码。通常情况下，重构类的方式是将数据成员和函数成员移动到继承体系的高级别节点当中，从而避免代码冗余。

运行时绑定 (run-time binding) 参见“动态绑定”。

切掉 (sliced down) 当我们用一个派生类对象初始化基类对象或者为基类对象赋值时发生的情况。对象的派生类部分将被“切掉”，只剩下基类部分赋值给基类对象。

静态类型 (static type) 对象被定义的类型或表达式产生的类型。静态类型在编译时是已知的。

虚函数 (virtual function) 用于定义类型特定行为的成员函数。通过引用或指针对虚函数的调用直到运行时才被解析，依据是引用或指针所绑定对象的类型。

第 16 章

模板与泛型编程

内容

16.1 定义模板	578
16.2 模板实参推断	600
16.3 重载与模板	614
16.4 可变参数模板	618
16.5 模板特例化	624
小结	630
术语表	630

面向对象编程（OOP）和泛型编程都能处理在编写程序时不知道类型的情况。不同之处在于：OOP 能处理类型在程序运行之前都未知的情况；而在泛型编程中，在编译时就能获知类型了。

本书第 II 部分中介绍的容器、迭代器和算法都是泛型编程的例子。当我们编写一个泛型程序时，是独立于任何特定类型来编写代码的。当使用一个泛型程序时，我们提供类型或值，程序实例可在其上运行。

例如，标准库为每个容器提供了单一的、泛型的定义，如 `vector`。我们可以使用这个泛型定义来定义很多类型的 `vector`，它们的差异就在于包含的元素类型不同。

模板是泛型编程的基础。我们不必了解模板是如何定义的就能使用它们，实际上我们已经这样用了。在本章中，我们将学习如何定义自己的模板。

652 模板是 C++ 中泛型编程的基础。一个模板就是一个创建类或函数的蓝图或者说公式。当使用一个 `vector` 这样的泛型类型，或者 `find` 这样的泛型函数时，我们提供足够的信息，将蓝图转换为特定的类或函数。这种转换发生在编译时。在本书第 3 章和第 II 部分中我们已经学习了如何使用模板。在本章中，我们将学习如何定义模板。

16.1 定义模板

假定我们希望编写一个函数来比较两个值，并指出第一个值是小于、等于还是大于第二个值。在实际中，我们可能想要定义多个函数，每个函数比较一种给定类型的值。我们的初次尝试可能定义多个重载函数：

```
// 如果两个值相等，返回 0，如果 v1 小返回 -1，如果 v2 小返回 1
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

这两个函数几乎是相同的，唯一的差异是参数的类型，函数体则完全一样。

如果对每种希望比较的类型都不得不重复定义完全一样的函数体，是非常烦琐且容易出错的。更麻烦的是，在编写程序的时候，我们就要确定可能要 `compare` 的所有类型。如果希望能在用户提供的类型上使用此函数，这种策略就失效了。



16.1.1 函数模板

我们可以定义一个通用的 **函数模板** (function template)，而不是为每个类型都定义一个新函数。一个函数模板就是一个公式，可用来生成针对特定类型的函数版本。`compare` 的模板版本可能像下面这样：

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

653 模板定义以关键字 `template` 开始，后跟一个 **模板参数列表** (template parameter list)，这是一个逗号分隔的一个或多个 **模板参数** (template parameter) 的列表，用小于号 (<) 和大于号 (>) 包围起来。



在模板定义中，模板参数列表不能为空。

模板参数列表的作用很像函数参数列表。函数参数列表定义了若干特定类型的局部变量，但并未指出如何初始化它们。在运行时，调用者提供实参来初始化形参。

类似的，模板参数表示在类或函数定义中用到的类型或值。当使用模板时，我们（隐式地或显式地）指定模板实参（template argument），将其绑定到模板参数上。

我们的 `compare` 函数声明了一个名为 `T` 的类型参数。在 `compare` 中，我们用名字 `T` 表示一个类型。而 `T` 表示的实际类型则在编译时根据 `compare` 的使用情况来确定。

实例化函数模板

当我们调用一个函数模板时，编译器（通常）用函数实参来为我们推断模板实参。即，当我们调用 `compare` 时，编译器使用实参的类型来确定绑定到模板参数 `T` 的类型。例如，在下面的调用中：

```
cout << compare(1, 0) << endl; // T 为 int
```

实参类型是 `int`。编译器会推断出模板实参为 `int`，并将它绑定到模板参数 `T`。

编译器用推断出的模板参数来为我们实例化（*instantiate*）一个特定版本的函数。当编译器实例化一个模板时，它使用实际的模板实参代替对应的模板参数来创建出模板的一个新“实例”。例如，给定下面的调用：

```
// 实例化出 int compare(const int&, const int&)
cout << compare(1, 0) << endl; // T 为 int
// 实例化出 int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T 为 vector<int>
```

编译器会实例化出两个不同版本的 `compare`。对于第一个调用，编译器会编写并编译一个 `compare` 版本，其中 `T` 被替换为 `int`：

```
int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

对于第二个调用，编译器会生成另一个 `compare` 版本，其中 `T` 被替换为 `vector<int>`。这些编译器生成的版本通常被称为模板的实例（*instantiation*）。

模板类型参数

654

我们的 `compare` 函数有一个模板类型参数（type parameter）。一般来说，我们可以将类型参数看作类型说明符，就像内置类型或类类型说明符一样使用。特别是，类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换：

```
// 正确：返回类型和参数类型相同
template <typename T> T foo(T* p)
{
    T tmp = *p; // tmp 的类型将是指针 p 指向的类型
    //...
    return tmp;
}
```

类型参数前必须使用关键字 `class` 或 `typename`:

```
// 错误: U 之前必须加上 class 或 typename
template <typename T, U> T calc(const T&, const U&);
```

在模板参数列表中，这两个关键字的含义相同，可以互换使用。一个模板参数列表中可以同时使用这两个关键字:

```
// 正确: 在模板参数列表中, typename 和 class 没有什么不同
template <typename T, class U> calc (const T&, const U&);
```

看起来用关键字 `typename` 来指定模板类型参数比用 `class` 更为直观。毕竟，我们可以用内置（非类）类型作为模板类型实参。而且，`typename` 更清楚地指出随后的名字是一个类型名。但是，`typename` 是在模板已经广泛使用之后才引入 C++ 语言的，某些程序员仍然只用 `class`。

非类型模板参数

除了定义类型参数，还可以在模板中定义非类型参数（nontype parameter）。一个非类型参数表示一个值而非一个类型。我们通过一个特定的类型名而非关键字 `class` 或 `typename` 来指定非类型参数。

当一个模板被实例化时，非类型参数被一个用户提供的或编译器推断出的值所代替。这些值必须是常量表达式（参见 2.4.4 节，第 58 页），从而允许编译器在编译时实例化模板。

例如，我们可以编写一个 `compare` 版本处理字符串字面常量。这种字面常量是 `const char` 的数组。由于不能拷贝一个数组，所以我们将自己的参数定义为数组的引用（参见 6.2.4 节，第 195 页）。由于我们希望能比较不同长度的字符串字面常量，因此为模板定义了两个非类型的参数。第一个模板参数表示第一个数组的长度，第二个参数表示第二个数组的长度：

```
655 > template<unsigned N, unsigned M>
      int compare(const char (&p1) [N], const char (&p2) [M])
    {
        return strcmp(p1, p2);
    }
```

当我们调用这个版本的 `compare` 时：

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

编译器会使用字面常量的大小来代替 `N` 和 `M`，从而实例化模板。记住，编译器会在一个字符串字面常量的末尾插入一个空字符作为终结符（参见 2.1.3 节，第 36 页），因此编译器会实例化出如下版本：

```
int compare(const char (&p1) [3], const char (&p2) [4])
```

一个非类型参数可以是一个整型，或者是一个指向对象或函数类型的指针或（左值）引用。绑定到非类型整型参数的实参必须是一个常量表达式。绑定到指针或引用非类型参数的实参必须具有静态的生存期（参见第 12 章，第 400 页）。我们不能用一个普通（非 `static`）局部变量或动态对象作为指针或引用非类型模板参数的实参。指针参数也可以用 `nullptr` 或一个值为 0 的常量表达式来实例化。

在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用

非类型参数，例如，指定数组大小。



非类型模板参数的模板实参必须是常量表达式。

inline 和 constexpr 的函数模板

函数模板可以声明为 `inline` 或 `constexpr` 的，如同非模板函数一样。`inline` 或 `constexpr` 说明符放在模板参数列表之后，返回类型之前：

```
// 正确: inline 说明符跟在模板参数列表之后
template <typename T> inline T min(const T&, const T&);

// 错误: inline 说明符的位置不正确
inline template <typename T> T min(const T&, const T&);
```

编写类型无关的代码



我们最初的 `compare` 函数虽然简单，但它说明了编写泛型代码的两个重要原则：

- 模板中的函数参数是 `const` 的引用。
- 函数体中的条件判断仅使用`<`比较运算。

通过将函数参数设定为 `const` 的引用，我们保证了函数可以用于不能拷贝的类型。大多 656 数类型，包括内置类型和我们已经用过的标准库类型（除 `unique_ptr` 和 `IO` 类型之外），都是允许拷贝的。但是，不允许拷贝的类类型也是存在的。通过将参数设定为 `const` 的引用，保证了这些类型可以用我们的 `compare` 函数来处理。而且，如果 `compare` 用于处理大对象，这种设计策略还能使函数运行得更快。

你可能认为既使用`<`运算符又使用`>`运算符来进行比较操作会更为自然：

```
// 期望的比较操作
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

但是，如果编写代码时只使用`<`运算符，我们就降低了 `compare` 函数对要处理的要求。这些类型必须支持`<`，但不必同时支持`>`。

实际上，如果我们真的关心类型无关和可移植性，可能需要用 `less`（参见 14.8.2 节，第 510 页）来定义我们的函数：

```
// 即使用于指针也正确的 compare 版本；参见 14.8.2 节（第 510 页）
template <typename T> int compare(const T &v1, const T &v2)
{
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
```

原始版本存在的问题是，如果用户调用它比较两个指针，且两个指针未指向相同的数组，则代码的行为是未定义的（据查阅资料，`less<T>`的默认实现用的就是`<`，所以这其实并未起到让这种比较有一个良好定义的作用——译者注）。



模板程序应该尽量减少对实参类型的要求。



模板编译

当编译器遇到一个模板定义时，它并不生成代码。只有当我们实例化出模板的一个特定版本时，编译器才会生成代码。当我们使用（而不是定义）模板时，编译器才生成代码，这一特性影响了我们如何组织代码以及错误何时被检测到。

通常，当我们调用一个函数时，编译器只需要掌握函数的声明。类似的，当我们使用一个类类型的对象时，类定义必须是可用的，但成员函数的定义不必已经出现。因此，我们将类定义和函数声明放在头文件中，而普通函数和类的成员函数的定义放在源文件中。

模板则不同：为了生成一个实例化版本，编译器需要掌握函数模板或类模板成员函数的定义。因此，与非模板代码不同，模板的头文件通常既包括声明也包括定义。

657



函数模板和类模板成员函数的定义通常放在头文件中。

关键概念：模板和头文件

模板包含两种名字：

- 那些不依赖于模板参数的名字
- 那些依赖于模板参数的名字

当使用模板时，所有不依赖于模板参数的名字都必须是可见的，这是由模板的提供者来保证的。而且，模板的提供者必须保证，当模板被实例化时，模板的定义，包括类模板的成员的定义，也必须是可见的。

用来实例化模板的所有函数、类型以及与类型关联的运算符的声明都必须是可见的，这是由模板的用户来保证的。

通过组织良好的程序结构，恰当使用头文件，这些要求都很容易满足。模板的设计者应该提供一个头文件，包含模板定义以及在类模板或成员定义中用到的所有名字的声明。模板的用户必须包含模板的头文件，以及用来实例化模板的任何类型的头文件。

大多数编译错误在实例化期间报告

模板直到实例化时才会生成代码，这一特性影响了我们何时才会获知模板内代码的编译错误。通常，编译器会在三个阶段报告错误。

第一个阶段是编译模板本身时。在这个阶段，编译器通常不会发现很多错误。编译器可以检查语法错误，例如忘记分号或者变量名拼错等，但也就这么多了。

第二个阶段是编译器遇到模板使用时。在此阶段，编译器仍然没有很多可检查的。对于函数模板调用，编译器通常会检查实参数目是否正确。它还能检查参数类型是否匹配。对于类模板，编译器可以检查用户是否提供了正确数目的模板实参，但也仅限于此了。

第三个阶段是模板实例化时，只有这个阶段才能发现类型相关的错误。依赖于编译器如何管理实例化，这类错误可能在链接时才报告。

当我们编写模板时，代码不能是针对特定类型的，但模板代码通常对其所使用的类型有一些假设。例如，我们最初的 `compare` 函数中的代码就假定实参类型定义了`<`运算符。

```
if (v1 < v2) return -1; // 要求类型 T 的对象支持<操作
if (v2 < v1) return 1; // 要求类型 T 的对象支持<操作
```

```
return 0; // 返回 int; 不依赖于 T
```

当编译器处理此模板时，它不能验证 `if` 语句中的条件是否合法。如果传递给 `compare` <658> 的实参定义了`<`运算符，则代码就是正确的，否则就是错误的。例如，

```
Sales_data data1, data2;  
cout << compare(data1, data2) << endl; // 错误: Sales_data 未定义<
```

此调用实例化了 `compare` 的一个版本，将 `T` 替换为 `Sales_data`。`if` 条件试图对 `Sales_data` 对象使用`<`运算符，但 `Sales_data` 并未定义此运算符。此实例化生成了一个无法编译通过的函数版本。但是，这样的错误直至编译器在类型 `Sales_data` 上实例化 `compare` 时才会被发现。



保证传递给模板的实参支持模板所要求的操作，以及这些操作在模板中能正确工作，是调用者的责任。

16.1.1 节练习

练习 16.1: 给出实例化的定义。

练习 16.2: 编写并测试你自己版本的 `compare` 函数。

练习 16.3: 对两个 `Sales_data` 对象调用你的 `compare` 函数，观察编译器在实例化过程中如何处理错误。

练习 16.4: 编写行为类似标准库 `find` 算法的模板。函数需要两个模板类型参数，一个表示函数的迭代器参数，另一个表示值的类型。使用你的函数在一个 `vector<int>` 和一个 `list<string>` 中查找给定值。

练习 16.5: 为 6.2.4 节（第 195 页）中的 `print` 函数编写模板版本，它接受一个数组的引用，能处理任意大小、任意元素类型的数组。

练习 16.6: 你认为接受一个数组实参的标准库函数 `begin` 和 `end` 是如何工作的？定义你自己版本的 `begin` 和 `end`。

练习 16.7: 编写一个 `constexpr` 模板，返回给定数组的大小。

练习 16.8: 在第 97 页的“关键概念”中，我们注意到，C++程序员喜欢使用`!=`而不喜欢`<`。解释这个习惯的原因。

16.1.2 类模板



类模板（class template）是用来生成类的蓝图的。与函数模板的不同之处是，编译器不能为类模板推断模板参数类型。如我们已经多次看到的，为了使用类模板，我们必须在模板名后的尖括号中提供额外信息（参见 3.3 节，第 87 页）——用来代替模板参数的模板实参列表。

<659>

定义类模板

作为一个例子，我们将实现 `StrBlob`（参见 12.1.1 节，第 405 页）的模板版本。我们将此模板命名为 `Blob`，意指它不再针对 `string`。类似 `StrBlob`，我们的模板会提供对元素的共享（且核查过的）访问能力。与类不同，我们的模板可以用于更多类型的元素。与标准库容器相同，当使用 `Blob` 时，用户需要指出元素类型。

类似函数模板，类模板以关键字 `template` 开始，后跟模板参数列表。在类模板（及其成员）的定义中，我们将模板参数当作替身，代替使用模板时用户需要提供的类型或值：

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // 构造函数
    Blob();
    Blob(std::initializer_list<T> il);
    // Blob 中的元素数目
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const T &t) { data->push_back(t); }
    // 移动版本，参见 13.6.3 节（第 484 页）
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    // 元素访问
    T& back();
    T& operator[](size_type i); // 在 14.5 节（第 501 页）中定义
private:
    std::shared_ptr<std::vector<T>> data;
    // 若 data[i] 无效，则抛出 msg
    void check(size_type i, const std::string &msg) const;
};
```

我们的 `Blob` 模板有一个名为 `T` 的模板类型参数，用来表示 `Blob` 保存的元素的类型。例如，我们将元素访问操作的返回类型定义为 `T&`。当用户实例化 `Blob` 时，`T` 就会被替换为特定的模板实参类型。

除了模板参数列表和使用 `T` 代替 `string` 之外，此类模板的定义与 12.1.1 节（第 405 页）中定义的类版本及 12.1.6 节（第 422 页）和第 13 章、第 14 章中更新的版本是一样的。

660 实例化类模板

我们已经多次见到，当使用一个类模板时，我们必须提供额外信息。我们现在知道这些额外信息是显式模板实参（explicit template argument）列表，它们被绑定到模板参数。编译器使用这些模板实参来实例化出特定的类。

例如，为了用我们的 `Blob` 模板定义一个类型，必须提供元素类型：

```
Blob<int> ia; // 空 Blob<int>
Blob<int> ia2 = {0,1,2,3,4}; // 有 5 个元素的 Blob<int>
```

`ia` 和 `ia2` 使用相同的特定类型版本的 `Blob`（即 `Blob<int>`）。从这两个定义，编译器会实例化出一个与下面定义等价的类：

```
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    ...
    int& operator[](size_type i);
```

```
private:  
    std::shared_ptr<std::vector<int>> data;  
    void check(size_type i, const std::string &msg) const;  
};
```

当编译器从我们的 Blob 模板实例化出一个类时，它会重写 Blob 模板，将模板参数 T 的每个实例替换为给定的模板实参，在本例中是 int。

对我们指定的每一种元素类型，编译器都生成一个不同的类：

```
// 下面的定义实例化出两个不同的 Blob 类型  
Blob<string> names; // 保存 string 的 Blob  
Blob<double> prices; // 不同的元素类型
```

这两个定义会实例化出两个不同的类。names 的定义创建了一个 Blob 类，每个 T 都被替换为 string。prices 的定义生成了另一个 Blob 类，T 被替换为 double。



一个类模板的每个实例都形成一个独立的类。类型 Blob<string> 与任何其他 Blob 类型都没有关联，也不会对任何其他 Blob 类型的成员有特殊访问权限。

在模板作用域中引用模板类型



为了阅读模板类代码，应该记住类模板的名字不是一个类型名（参见 3.3 节，第 87 页）。类模板用来实例化类型，而一个实例化的类型总是包含模板参数的。

可能令人迷惑的是，一个类模板中的代码如果使用了另外一个模板，通常不将一个实际类型（或值）的名字用作其模板实参。相反的，我们通常将模板自己的参数当作被使用模板的实参。例如，我们的 data 成员使用了两个模板，vector 和 shared_ptr。我们知道，无论何时使用模板都必须提供模板实参。在本例中，我们提供的模板实参就是 Blob 的模板参数。因此，data 的定义如下：

<661

```
std::shared_ptr<std::vector<T>> data;
```

它使用了 Blob 的类型参数来声明 data 是一个 shared_ptr 的实例，此 shared_ptr 指向一个保存类型为 T 的对象的 vector 实例。当我们实例化一个特定类型的 Blob，例如 Blob<string> 时，data 会成为：

```
shared_ptr<vector<string>>
```

如果我们实例化 Blob<int>，则 data 会成为 shared_ptr<vector<int>>，依此类推。

类模板的成员函数

与其他任何类相同，我们既可以在类模板内部，也可以在类模板外部为其定义成员函数，且定义在类模板内的成员函数被隐式声明为内联函数。

类模板的成员函数本身是一个普通函数。但是，类模板的每个实例都有其自己版本的成员函数。因此，类模板的成员函数具有和模板相同的模板参数。因而，定义在类模板之外的成员函数就必须以关键字 template 开始，后接类模板参数列表。

与往常一样，当我们在类外定义一个成员时，必须说明成员属于哪个类。而且，从一个模板生成的类的名字中必须包含其模板实参。当我们定义一个成员函数时，模板实参与模板形参相同。即，对于 StrBlob 的一个给定的成员函数

```
ret-type StrBlob::member-name(parm-list)
```

对应的 Blob 的成员应该是这样的：

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

check 和元素访问成员

我们首先定义 check 成员，它检查一个给定的索引：

```
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

除了类名中的不同之处以及使用了模板参数列表外，此函数与原 StrBlob 类的 check 成员完全一样。

下标运算符和 back 函数用模板参数指出返回类型，其他未变：

```
662> template <typename T>
T& Blob<T>::back()
{
    check(0, "back on empty Blob");
    return data->back();
}
template <typename T>
T& Blob<T>::operator[](size_type i)
{
    // 如果 i 太大, check 会抛出异常, 阻止访问一个不存在的元素
    check(i, "subscript out of range");
    return (*data)[i];
}
```

在原 StrBlob 类中，这些运算符返回 `string&`。而模板版本则返回一个引用，指向用来实例化 Blob 的类型。

`pop_back` 函数与原 StrBlob 的成员几乎相同：

```
template <typename T> void Blob<T>::pop_back()
{
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
```

在原 StrBlob 类中，下标运算符和 back 成员都对 `const` 对象进行了重载。我们将这些成员及 front 成员的定义留作练习。

Blob 构造函数

与其他任何定义在类模板外的成员一样，构造函数的定义要以模板参数开始：

```
template <typename T>
Blob<T>::Blob(): data(std::make_shared<std::vector<T>>()) {}
```

这段代码在作用域 `Blob<T>` 中定义了名为 `Blob` 的成员函数。类似 `StrBlob` 的默认构造

函数（参见 12.1.1 节，第 405 页），此构造函数分配一个空 vector，并将指向 vector 的指针保存在 data 中。如前所述，我们将类模板自己的类型参数作为 vector 的模板实参来分配 vector。

类似的，接受一个 `initializer_list` 参数的构造函数将其类型参数 `T` 作为 `initializer_list` 参数的元素类型：

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il):
    data(std::make_shared<std::vector<T>>(il)) {}
```

类似默认构造函数，此构造函数分配一个新的 vector。在本例中，我们用参数 `il` 来初始化此 vector。

为了使用这个构造函数，我们必须传递给它一个 `initializer_list`，其中的元素必须与 Blob 的元素类型兼容：

```
Blob<string> articles = {"a", "an", "the"};
```

这条语句中，构造函数的参数类型为 `initializer_list<string>`。列表中的每个字符串字面常量隐式地转换为一个 `string`。

类模板成员函数的实例化

663

默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化。例如，下面代码

```
// 实例化 Blob<int> 和接受 initializer_list<int> 的构造函数
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
// 实例化 Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; // 实例化 Blob<int>::operator[](size_t)
```

实例化了 `Blob<int>` 类和它的三个成员函数：`operator[]`、`size` 和接受 `initializer_list<int>` 的构造函数。

如果一个成员函数没有被使用，则它不会被实例化。成员函数只有在被用到时才进行实例化，这一特性使得即使某种类型不能完全符合模板操作的要求（参见 9.2 节，第 294 页），我们仍然能用该类型实例化类。



默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被实例化。

在类代码内简化模板类名的使用

当我们使用一个类模板类型时必须提供模板实参，但这一规则有一个例外。在类模板自己的作用域中，我们可以直接使用模板名而不提供实参：

```
// 若试图访问一个不存在的元素，BlobPtr 抛出一个异常
template <typename T> class BlobPtr {
public:
    BlobPtr(): curr(0) {}
    BlobPtr(Blob<T> &a, size_t sz = 0):
        wptr(a.data), curr(sz) {}
    T& operator*() const
    { auto p = check(curr, "dereference past end");
        return (*p)[curr]; // (*p) 为本对象指向的 vector
```

```

    }
    // 递增和递减
    BlobPtr& operator++(); // 前置运算符
    BlobPtr& operator--();
private:
    // 若检查成功, check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr, 表示底层 vector 可能被销毁
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr; // 数组中的当前位置
};

```

细心的读者可能已经注意到, BlobPtr 的前置递增和递减成员返回 BlobPtr&, 而不是 BlobPtr<T>&。当我们处于一个类模板的作用域中时, 编译器处理模板自身引用时就好像我们已经提供了与模板参数匹配的实参一样。即, 就好像我们这样编写代码一样:

```

BlobPtr<T>& operator++();
BlobPtr<T>& operator--();

```

在类模板外使用类模板名

当我们在类模板外定义其成员时, 必须记住, 我们并不在类的作用域中, 直到遇到类名才表示进入类的作用域 (参见 7.4 节, 第 253 页):

```

// 后置: 递增/递减对象但返回原值
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    // 此处无须检查; 调用前置递增时会进行检查
    BlobPtr ret = *this; // 保存当前值
    ++*this; // 推进一个元素; 前置++检查递增是否合法
    return ret; // 返回保存的状态
}

```

由于返回类型位于类的作用域之外, 我们必须指出返回类型是一个实例化的 BlobPtr, 它所用类型与类实例化所用类型一致。在函数体内, 我们已经进入类的作用域, 因此在定义 ret 时无须重复模板实参。如果不提供模板实参, 则编译器将假定我们使用的类型与成员实例化所用类型一致。因此, ret 的定义与如下代码等价:

```

BlobPtr<T> ret = *this;

```



在一个类模板的作用域内, 我们可以直接使用模板名而不必指定模板实参。

类模板和友元

当一个类包含一个友元声明 (参见 7.2.1 节, 第 241 页) 时, 类与友元各自是否是模板是相互无关的。如果一个类模板包含一个非模板友元, 则友元被授权可以访问所有模板实例。如果友元自身是模板, 类可以授权给所有友元模板实例, 也可以只授权给特定实例。

一对一封好关系

类模板与另一个 (类或函数) 模板间友好关系的最常见的形式是建立对应实例及其友元间的友好关系。例如, 我们的 Blob 类应该将 BlobPtr 类和一个模板版本的 Blob 相

等运算符（最初是在 14.3.1 节（第 498 页）练习中为 `StrBlob` 定义的）定义为友元。

为了引用（类或函数）模板的一个特定实例，我们必须首先声明模板自身。一个模板声明包括模板参数列表：

```
// 前置声明，在 Blob 中声明友元所需要的
template <typename> class BlobPtr;
template <typename> class Blob; // 运算符==中的参数所需要的
template <typename T>
    bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {
    // 每个 Blob 实例将访问权限授予用相同类型实例化的 BlobPtr 和相等运算符
    friend class BlobPtr<T>;
    friend bool operator==(const Blob<T>&, const Blob<T>&);
    // 其他成员定义，与 12.1.1（第 405 页）相同
};
```

< 665

我们首先将 `Blob`、`BlobPtr` 和 `operator==` 声明为模板。这些声明是 `operator==` 函数的参数声明以及 `Blob` 中的友元声明所需要的。

友元的声明用 `Blob` 的模板形参作为它们自己的模板实参。因此，友好关系被限定在用相同类型实例化的 `Blob` 与 `BlobPtr` 相等运算符之间：

```
Blob<char> ca; // BlobPtr<char>和 operator==<char>都是本对象的友元
Blob<int> ia; // BlobPtr<int>和 operator==<int>都是本对象的友元
```

`BlobPtr<char>` 的成员可以访问 `ca`（或任何其他 `Blob<char>` 对象）的非 `public` 部分，但 `ca` 对 `ia`（或任何其他 `Blob<int>` 对象）或 `Blob` 的任何其他实例都没有特殊访问权限。

通用和特定的模板友好关系

一个类也可以将另一个模板的每个实例都声明为自己的友元，或者限定特定的实例为友元：

```
// 前置声明，在将模板的一个特定实例声明为友元时要用到
template <typename T> class Pal;
class C { // C 是一个普通的非模板类
    friend class Pal<C>; // 用类 C 实例化的 Pal 是 C 的一个友元
    // Pal2 的所有实例都是 C 的友元；这种情况无须前置声明
    template <typename T> friend class Pal2;
};

template <typename T> class C2 { // C2 本身是一个类模板
    // C2 的每个实例将相同实例化的 Pal 声明为友元
    friend class Pal<T>; // Pal 的模板声明必须在作用域之内
    // Pal2 的所有实例都是 C2 的每个实例的友元，不需要前置声明
    template <typename X> friend class Pal2;
    // Pal3 是一个非模板类，它是 C2 所有实例的友元
    friend class Pal3; // 不需要 Pal3 的前置声明
};
```

为了让所有实例成为友元，友元声明中必须使用与类模板本身不同的模板参数。

666 令模板自己的类型参数成为友元

C++ 11 在新标准中，我们可以将模板类型参数声明为友元：

```
template <typename Type> class Bar {
    friend Type; // 将访问权限授予用来实例化 Bar 的类型
    //...
};
```

此处我们将用来实例化 Bar 的类型声明为友元。因此，对于某个类型名 Foo，Foo 将成为 Bar<Foo>的友元，Sales_data 将成为 Bar<Sales_data>的友元，依此类推。

值得注意的是，虽然友元通常来说应该是一个类或是一个函数，但我们完全可以用一个内置类型来实例化 Bar。这种与内置类型的友好关系是允许的，以便我们能用内置类型来实例化 Bar 这样的类。

模板类型别名

类模板的一个实例定义了一个类类型，与任何其他类类型一样，我们可以定义一个 `typedef`（参见 2.5.1 节，第 60 页）来引用实例化的类：

```
typedef Blob<string> StrBlob;
```

这条 `typedef` 语句允许我们运行在 12.1.1 节（第 405 页）中编写的代码，而使用的却是用 `string` 实例化的模板版本的 `Blob`。由于模板不是一个类型，我们不能定义一个 `typedef` 引用一个模板。即，无法定义一个 `typedef` 引用 `Blob<T>`。

C++ 11 但是，新标准允许我们为类模板定义一个类型别名：

```
template<typename T> using twin = pair<T, T>;
twin<string> authors; // authors 是一个 pair<string, string>
```

在这段代码中，我们将 `twin` 定义为成员类型相同的 `pair` 的别名。这样，`twin` 的用户只需指定一次类型。

一个模板类型别名是一族类的别名：

```
twin<int> win_loss; // win_loss 是一个 pair<int, int>
twin<double> area; // area 是一个 pair<double, double>
```

就像使用类模板一样，当我们使用 `twin` 时，需要指出希望使用哪种特定类型的 `twin`。

当我们定义一个模板类型别名时，可以固定一个或多个模板参数：

```
template <typename T> using partNo = pair<T, unsigned>;
partNo<string> books; // books 是一个 pair<string, unsigned>
partNo<Vehicle> cars; // cars 是一个 pair<Vehicle, unsigned>
partNo<Student> kids; // kids 是一个 pair<Student, unsigned>
```

这段代码中我们将 `partNo` 定义为一族类型的别名，这族类型是 `second` 成员为 `unsigned` 的 `pair`。`partNo` 的用户需要指出 `pair` 的 `first` 成员的类型，但不能指定 `second` 成员的类型。

667 类模板的 static 成员

与任何其他类相同，类模板可以声明 `static` 成员（参见 7.6 节，第 269 页）：

```
template <typename T> class Foo {
public:
```

```

    static std::size_t count() { return ctr; }
    // 其他接口成员
private:
    static std::size_t ctr;
    // 其他实现成员
};

```

在这段代码中，`Foo` 是一个类模板，它有一个名为 `count` 的 `public static` 成员函数和一个名为 `ctr` 的 `private static` 数据成员。每个 `Foo` 的实例都有其自己的 `static` 成员实例。即，对任意给定类型 `X`，都有一个 `Foo<X>::ctr` 和一个 `Foo<X>::count` 成员。所有 `Foo<X>` 类型的对象共享相同的 `ctr` 对象和 `count` 函数。例如，

```

// 实例化 static 成员 Foo<string>::ctr 和 Foo<string>::count
Foo<string> fs;
// 所有三个对象共享相同的 Foo<int>::ctr 和 Foo<int>::count 成员
Foo<int> fi, fi2, fi3;

```

与任何其他 `static` 数据成员相同，模板类的每个 `static` 数据成员必须有且仅有一个定义。但是，类模板的每个实例都有一个独有的 `static` 对象。因此，与定义模板的成员函数类似，我们将 `static` 数据成员也定义为模板：

```

template <typename T>
size_t Foo<T>::ctr = 0; // 定义并初始化 ctr

```

与类模板的其他任何成员类似，定义的开始部分是模板参数列表，随后是我们定义的成员的类型和名字。与往常一样，成员名包括成员的类名，对于从模板生成的类来说，类名包括模板实参。因此，当使用一个特定的模板实参类型实例化 `Foo` 时，将会为该类类型实例化一个独立的 `ctr`，并将其初始化为 0。

与非模板类的静态成员相同，我们可以通过类类型对象来访问一个类模板的 `static` 成员，也可以使用作用域运算符直接访问成员。当然，为了通过类来直接访问 `static` 成员，我们必须引用一个特定的实例：

```

Foo<int> fi;           // 实例化 Foo<int> 类和 static 数据成员 ctr
auto ct = Foo<int>::count(); // 实例化 Foo<int>::count
ct = fi.count();        // 使用 Foo<int>::count
ct = Foo::count();      // 错误：使用哪个模板实例的 count？

```

类似任何其他成员函数，一个 `static` 成员函数只有在使用时才会实例化。

16.1.2 节练习

668

练习 16.9：什么是函数模板？什么是类模板？

练习 16.10：当一个类模板被实例化时，会发生什么？

练习 16.11：下面 `List` 的定义是错误的。应如何修正它？

```

template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
}

```

```

    void insert(ListItem *ptr, elemType value);
private:
    ListItem *front, *end;
};

```

练习 16.12: 编写你自己版本的 Blob 和 BlobPtr 模板，包含书中未定义的多个 const 成员。

练习 16.13: 解释你为 BlobPtr 的相等和关系运算符选择哪种类型的友好关系？

练习 16.14: 编写 Screen 类模板，用非类型参数定义 Screen 的高和宽。

练习 16.15: 为你的 Screen 模板实现输入和输出运算符。Screen 类需要哪些友元（如果需要的话）来令输入和输出运算符正确工作？解释每个友元声明（如果有的话）为什么是必要的。

练习 16.16: 将 StrVec 类（参见 13.5 节，第 465 页）重写为模板，命名为 Vec。



16.1.3 模板参数

类似函数参数的名字，一个模板参数的名字也没有什么内在含义。我们通常将类型参数命名为 T，但实际上我们可以使用任何名字：

```

template <typename Foo> Foo calc(const Foo& a, const Foo& b)
{
    Foo tmp = a; // tmp 的类型与参数和返回类型一样
    //...
    return tmp; // 返回类型和参数类型一样
}

```

模板参数与作用域

模板参数遵循普通的作用域规则。一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前。与任何其他名字一样，模板参数会隐藏外层作用域中声明的相同名字。但是，与大多数其他上下文不同，在模板内不能重用模板参数名：

```

typedef double A;
template <typename A, typename B> void f(A a, B b)
{
    A tmp = a; // tmp 的类型为模板参数 A 的类型，而非 double
    double B; // 错误：重声明模板参数 B
}

```

正常的名字隐藏规则决定了 A 的 `typedef` 被类型参数 A 隐藏。因此，tmp 不是一个 `double`，其类型是使用 `f` 时绑定到类型参数 A 的类型。由于我们不能重用模板参数名，声明名字为 B 的变量是错误的。

由于参数名不能重用，所以一个模板参数名在一个特定模板参数列表中只能出现一次：

```

// 错误：非法重用模板参数名 V
template <typename V, typename V> //...

```

模板声明

模板声明必须包含模板参数：

```
// 声明但不定义 compare 和 Blob
template <typename T> int compare(const T&, const T&);
template <typename T> class Blob;
```

与函数参数相同，声明中的模板参数的名字不必与定义中相同：

```
// 3个 calc 都指向相同的函数模板
template <typename T> T calc(const T&, const T&); // 声明
template <typename U> U calc(const U&, const U&); // 声明
// 模板的定义
template <typename Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

当然，一个给定模板的每个声明和定义必须有相同数量和种类（即，类型或非类型）的参数。



一个特定文件所需的所有模板的声明通常一起放置在文件开始位置，出现于任何使用这些模板的代码之前，原因我们将在 16.3 节（第 617 页）中解释。

使用类的类型成员

回忆一下，我们用作用域运算符 (::) 来访问 static 成员和类型成员（参见 7.4 节，第 253 页和 7.6 节，第 269 页）。在普通（非模板）代码中，编译器掌握类的定义。因此，它知道通过作用域运算符访问的名字是类型还是 static 成员。例如，如果我们写下 string::size_type，编译器有 string 的定义，从而知道 size_type 是一个类型。

670

但对于模板代码就存在困难。例如，假定 T 是一个模板类型参数，当编译器遇到类似 T::mem 这样的代码时，它不会知道 mem 是一个类型成员还是一个 static 数据成员，直至实例化时才会知道。但是，为了处理模板，编译器必须知道名字是否表示一个类型。例如，假定 T 是一个类型参数的名字，当编译器遇到如下形式的语句时：

```
T::size_type * p;
```

它需要知道我们是正在定义一个名为 p 的变量还是将一个名为 size_type 的 static 数据成员与名为 p 的变量相乘。

默认情况下，C++ 语言假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显式告诉编译器该名字是一个类型。我们通过使用关键字 typename 来实现这一点：

```
template <typename T>
typename T::value_type top(const T& c)
{
    if (!c.empty())
        return c.back();
    else
        return typename T::value_type();
}
```

我们的 top 函数期待一个容器类型的实参，它使用 typename 指明其返回类型并在 c 中没有元素时生成一个值初始化的元素（参见 7.5.3 节，第 262 页）返回给调用者。



当我们希望通知编译器一个名字表示类型时，必须使用关键字 typename，而不能使用 class。

默认模板实参

C++
11

就像我们能为函数参数提供默认实参一样（参见 6.5.1 节，第 211 页），我们也可以提供默认模板实参（default template argument）。在新标准中，我们可以为函数和类模板提供默认实参。而更早的 C++ 标准只允许为类模板提供默认实参。

例如，我们重写 `compare`，默认使用标准库的 `less` 函数对象模板（参见 14.8.2 节，第 509 页）：

```
// compare 有一个默认模板实参 less<T> 和一个默认函数实参 F()
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

671 在这段代码中，我们为模板添加了第二个类型参数，名为 `F`，表示可调用对象（参见 10.3.2 节，第 346 页）的类型；并定义了一个新的函数参数 `f`，绑定到一个可调用对象上。

我们为此模板参数提供了默认实参，并为其对应的函数参数也提供了默认实参。默认模板实参指出 `compare` 将使用标准库的 `less` 函数对象类，它是使用与 `compare` 一样的类型参数实例化的。默认函数实参指出 `f` 将是类型 `F` 的一个默认初始化的对象。

当用户调用这个版本的 `compare` 时，可以提供自己的比较操作，但这并不是必需的：

```
bool i = compare(0, 42); // 使用 less; i 为 -1
// 结果依赖于 item1 和 item2 中的 isbn
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
```

第一个调用使用默认函数实参，即，类型 `less<T>` 的一个默认初始化对象。在此调用中，`T` 为 `int`，因此可调用对象的类型为 `less<int>`。`compare` 的这个实例化版本将使用 `less<int>` 进行比较操作。

在第二个调用中，我们传递给 `compare` 三个实参：`compareIsbn`（参见 11.2.2 节，第 379 页）和两个 `Sales_data` 类型的对象。当传递给 `compare` 三个实参时，第三个实参的类型必须是一个可调用对象，该可调用对象的返回类型必须能转换为 `bool` 值，且接受的实参类型必须与 `compare` 的前两个实参的类型兼容。与往常一样，模板参数的类型从它们对应的函数实参推断而来。在此调用中，`T` 的类型被推断为 `Sales_data`，`F` 被推断为 `compareIsbn` 的类型。

与函数默认实参一样，对于一个模板参数，只有当它右侧的所有参数都有默认实参时，它才可以有默认实参。

模板默认实参与类模板

无论何时使用一个类模板，我们都必须在模板名之后接上尖括号。尖括号指出类必须从一个模板实例化而来。特别是，如果一个类模板为其所有模板参数都提供了默认实参，且我们希望使用这些默认实参，就必须在模板名之后跟一个空尖括号对：

```
template <class T = int> class Numbers { // T 默认为 int
public:
    Numbers(T v = 0) : val(v) {}
```

```

    // 对数值的各种操作
private:
    T val;
};

Numbers<long double> lots_of_precision;
Numbers<> average_precision; // 空<>表示我们希望使用默认类型

```

此例中我们实例化了两个 Numbers 版本：average_precision 是用 int 替代 T 实例化得到的；lots_of_precision 是用 long double 替代 T 实例化而得到的。

16.1.3 节练习

< 672

练习 16.17：声明为 typename 的类型参数和声明为 class 的类型参数有什么不同（如果有的话）？什么时候必须使用 typename？

练习 16.18：解释下面每个函数模板声明并指出它们是否非法。更正你发现的每个错误。

- (a) template <typename T, U, typename V> void f1(T, U, V);
- (b) template <typename T> T f2(int &T);
- (c) inline template <typename T> T foo(T, unsigned int*);
- (d) template <typename T> f4(T, T);
- (e) typedef char Ctype;
template <typename Ctype> Ctype f5(Ctype a);

练习 16.19：编写函数，接受一个容器的引用，打印容器中的元素。使用容器的 size_type 和 size 成员来控制打印元素的循环。

练习 16.20：重写上一题的函数，使用 begin 和 end 返回的迭代器来控制循环。

16.1.4 成员模板

一个类（无论是普通类还是类模板）可以包含本身是模板的成员函数。这种成员被称为成员模板（member template）。成员模板不能是虚函数。

普通（非模板）类的成员模板

作为普通类包含成员模板的例子，我们定义一个类，类似 unique_ptr 所使用的默认删除器类型（参见 12.1.5 节，第 418 页）。类似默认删除器，我们的类将包含一个重载的函数调用运算符（参见 14.8 节，第 506 页），它接受一个指针并对此指针执行 delete。与默认删除器不同，我们的类还将在删除器被执行时打印一条信息。由于希望删除器适用于任何类型，所以我们将调用运算符定义为一个模板：

```

// 函数对象类，对给定指针执行 delete
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr): os(s) { }
    // 与任何函数模板相同，T 的类型由编译器推断
    template <typename T> void operator()(T *p) const
        { os << "deleting unique_ptr" << std::endl; delete p; }
private:
    std::ostream &os;
};

```

673 与任何其他模板相同，成员模板也是以模板参数列表开始的。每个 `DebugDelete` 对象都有一个 `ostream` 成员，用于写入数据；还包含一个自身是模板的成员函数。我们可以用这个类代替 `delete`：

```
double* p = new double;
DebugDelete d; // 可像 delete 表达式一样使用的对象
d(p); // 调用 DebugDelete::operator()(double*)，释放 p
int* ip = new int;
// 在一个临时 DebugDelete 对象上调用 operator()(int*)
DebugDelete()(ip);
```

由于调用一个 `DebugDelete` 对象会 `delete` 其给定的指针，我们也可以将 `DebugDelete` 用作 `unique_ptr` 的删除器。为了重载 `unique_ptr` 的删除器，我们在尖括号内给出删除器类型，并提供一个这种类型的对象给 `unique_ptr` 的构造函数（参见 12.1.5 节，第 418 页）：

```
// 销毁 p 指向的对象
// 实例化 DebugDelete::operator()(int*)(int *)
unique_ptr<int, DebugDelete> p(new int, DebugDelete());
// 销毁 sp 指向的对象
// 实例化 DebugDelete::operator()(string*)(string*)
unique_ptr<string, DebugDelete> sp(new string, DebugDelete());
```

在本例中，我们声明 `p` 的删除器的类型为 `DebugDelete`，并在 `p` 的构造函数中提供了该类型的一个未命名对象。

`unique_ptr` 的析构函数会调用 `DebugDelete` 的调用运算符。因此，无论何时 `unique_ptr` 的析构函数实例化时，`DebugDelete` 的调用运算符都会实例化：因此，上述定义会这样实例化。

```
// DebugDelete 的成员模板实例化样例
void DebugDelete::operator()(int *p) const { delete p; }
void DebugDelete::operator()(string *p) const { delete p; }
```

类模板的成员模板

对于类模板，我们也可以为其定义成员模板。在此情况下，类和成员各自有自己的、独立的模板参数。

例如，我们将为 `Blob` 类定义一个构造函数，它接受两个迭代器，表示要拷贝的元素范围。由于我们希望支持不同类型序列的迭代器，因此将构造函数定义为模板：

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
    //...
};
```

此构造函数有自己的模板类型参数 `It`，作为它的两个函数参数的类型。

与类模板的普通函数成员不同，成员模板是函数模板。当我们在类模板外定义一个成员模板时，必须同时为类模板和成员模板提供模板参数列表。类模板的参数列表在前，后跟成员自己的模板参数列表：

```
template <typename T> // 类的类型参数
template <typename It> // 构造函数的类型参数
Blob<T>::Blob(It b, It e);
```

```
data(std::make_shared<std::vector<T>>(b, e)) { }
```

在此例中，我们定义了一个类模板的成员，类模板有一个模板类型参数，命名为 `T`。而成员自身是一个函数模板，它有一个名为 `It` 的类型参数。

实例化与成员模板

为了实例化一个类模板的成员模板，我们必须同时提供类和函数模板的实参。与往常一样，我们在哪个对象上调用成员模板，编译器就根据该对象的类型来推断类模板参数的实参。与普通函数模板相同，编译器通常根据传递给成员模板的函数实参来推断它的模板实参（参见 16.1.1 节，第 579 页）：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
// 实例化 Blob<int>类及其接受两个 int*参数的构造函数
Blob<int> a1(begin(ia), end(ia));
// 实例化 Blob<int>类的接受两个 vector<long>::iterator 的构造函数
Blob<int> a2(vi.begin(), vi.end());
// 实例化 Blob<string>及其接受两个 list<const char*>::iterator 参数的构造函数
Blob<string> a3(w.begin(), w.end());
```

当我们定义 `a1` 时，显式地指出编译器应该实例化一个 `int` 版本的 `Blob`。构造函数自己的类型参数则通过 `begin(ia)` 和 `end(ia)` 的类型来推断，结果为 `int*`。因此，`a1` 的定义实例化了如下版本：

```
Blob<int>::Blob(int*, int*);
```

`a2` 的定义使用了已经实例化了的 `Blob<int>` 类，并用 `vector<short>::iterator` 替换 `It` 来实例化构造函数。`a3` 的定义（显式地）实例化了一个 `string` 版本的 `Blob`，并（隐式地）实例化了该类的成员模板构造函数，其模板参数被绑定到 `list<const char*>`。

16.1.4 节练习

675

练习 16.21：编写你自己的 `DebugDelete` 版本。

练习 16.22：修改 12.3 节（第 430 页）中你的 `TextQuery` 程序，令 `shared_ptr` 成员使用 `DebugDelete` 作为它们的删除器（参见 12.1.4 节，第 415 页）。

练习 16.23：预测在你的查询主程序中何时会执行调用运算符。如果你的预测和实际不符，确认你理解了原因。

练习 16.24：为你的 `Blob` 模板添加一个构造函数，它接受两个迭代器。

16.1.5 控制实例化



当模板被使用时才会进行实例化（参见 16.1.1 节，第 582 页）这一特性意味着，相同的实例可能出现在多个对象文件中。当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例。

C++
11

在大系统中，在多个文件中实例化相同模板的额外开销可能非常严重。在新标准中，我们可以通过显式实例化（`explicit instantiation`）来避免这种开销。一个显式实例化有如下

形式：

```
extern template declaration;      // 实例化声明
template declaration;           // 实例化定义
```

declaration 是一个类或函数声明，其中所有模板参数已被替换为模板实参。例如，

```
// 实例化声明与定义
extern template class Blob<string>;           // 声明
template int compare(const int&, const int&); // 定义
```

当编译器遇到 `extern` 模板声明时，它不会在本文件中生成实例化代码。将一个实例化声明为 `extern` 就表示承诺在程序其他位置有该实例化的一个非 `extern` 声明（定义）。对于一个给定的实例化版本，可能有多个 `extern` 声明，但必须只有一个定义。

由于编译器在使用一个模板时自动对其实例化，因此 `extern` 声明必须出现在任何使用此实例化版本的代码之前：

```
// Application.cc
// 这些模板类型必须在程序其他位置进行实例化
extern template class Blob<string>;
extern template int compare(const int&, const int&);
Blob<string> sal, sa2; // 实例化会出现在其他位置
// Blob<int>及其接受 initializer_list 的构造函数在本文件中实例化
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); // 拷贝构造函数在本文件中实例化
int i = compare(a1[0], a2[0]); // 实例化出现在其他位置
```

676 文件 `Application.o` 将包含 `Blob<int>` 的实例及其接受 `initializer_list` 参数的构造函数和拷贝构造函数的实例。而 `compare<int>` 函数和 `Blob<string>` 类将不在本文件中进行实例化。这些模板的定义必须出现在程序的其他文件中：

```
// templateBuild.cc
// 实例化文件必须为每个在其他文件中声明为 extern 的类型和函数提供一个（非 extern）
// 的定义
template int compare(const int&, const int&);
template class Blob<string>; // 实例化类模板的所有成员
```

当编译器遇到一个实例化定义（与声明相对）时，它为其生成代码。因此，文件 `templateBuild.o` 将会包含 `compare` 的 `int` 实例化版本的定义和 `Blob<string>` 类的定义。当我们编译此应用程序时，必须将 `templateBuild.o` 和 `Application.o` 链接到一起。



对每个实例化声明，在程序中某个位置必须有其显式的实例化定义。

实例化定义会实例化所有成员

一个类模板的实例化定义会实例化该模板的所有成员，包括内联的成员函数。当编译器遇到一个实例化定义时，它不了解程序使用哪些成员函数。因此，与处理类模板的普通实例化不同，编译器会实例化该类的所有成员。即使我们不使用某个成员，它也会被实例化。因此，我们用来显式实例化一个类模板的类型，必须能用于模板的所有成员。



在一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数。

16.1.5 节练习

练习 16.25：解释下面这些声明的含义：

```
extern template class vector<string>;
template class vector<Sales_data>;
```

练习 16.26：假设 `NoDefault` 是一个没有默认构造函数的类，我们可以显式实例化 `vector<NoDefault>` 吗？如果不可以，解释为什么。

练习 16.27：对下面每条带标签的语句，解释发生了什么样的实例化（如果有的话）。如果一个模板被实例化，解释为什么；如果未实例化，解释为什么没有。

```
template <typename T> class Stack { };
void f1(Stack<char>); // (a)
class Exercise {
    Stack<double> &rsd; // (b)
    Stack<int> si; // (c)
};
int main() {
    Stack<char> *sc; // (d)
    f1(*sc); // (e)
    int iObj = sizeof(Stack< string >); // (f)
}
```

16.1.6 效率与灵活性



对模板设计者所面对的设计选择，标准库智能指针类型（参见 12.1 节，第 400 页）给出了一个很好的展示。

`shared_ptr` 和 `unique_ptr` 之间的明显不同是它们管理所保存的指针的策略——前者给予我们共享指针所有权的能力；后者则独占指针。这一差异对两个类的功能来说是至关重要的。

这两个类的另一个差异是它们允许用户重载默认删除器的方式。我们可以很容易地重载一个 `shared_ptr` 的删除器，只要在创建或 `reset` 指针时传递给它一个可调用对象即可。与之相反，删除器的类型是一个 `unique_ptr` 对象的类型的一部分。用户必须在定义 `unique_ptr` 时以显式模板实参的形式提供删除器的类型。因此，对于 `unique_ptr` 的用户来说，提供自己的删除器就更为复杂。

如何处理删除器的差异实际上就是这两个类功能的差异。但是，如我们将要看到的，这一实现策略上的差异可能对性能有重要影响。

677

在运行时绑定删除器

虽然我们不知道标准库类型是如何实现的，但可以推断出，`shared_ptr` 必须能直接访问其删除器。即，删除器必须保存为一个指针或一个封装了指针的类（如 `function`，参见 14.8.3 节，第 512 页）。

我们可以确定 `shared_ptr` 不是将删除器直接保存为一个成员，因为删除器的类型

直到运行时才会知道。实际上，在一个 `shared_ptr` 的生存期中，我们可以随时改变其删除器的类型。我们可以使用一种类型的删除器构造一个 `shared_ptr`，随后使用 `reset` 赋予此 `shared_ptr` 另一种类型的删除器。通常，类成员的类型在运行时是不能改变的。因此，不能直接保存删除器。

为了考察删除器是如何正确工作的，让我们假定 `shared_ptr` 将它管理的指针保存在一个成员 `p` 中，且删除器是通过一个名为 `del` 的成员来访问的。则 `shared_ptr` 的析构函数必须包含类似下面这样的语句：

```
// del 的值只有在运行时才知道；通过一个指针来调用它  
del ? del(p) : delete p; // del(p) 需要运行时跳转到 del 的地址
```

678 由于删除器是间接保存的，调用 `del(p)` 需要一次运行时的跳转操作，转到 `del` 中保存的地址来执行对应的代码。

在编译时绑定删除器

现在，让我们来考察 `unique_ptr` 可能的工作方式。在这个类中，删除器的类型是类类型的一部分。即，`unique_ptr` 有两个模板参数，一个表示它所管理的指针，另一个表示删除器的类型。由于删除器的类型是 `unique_ptr` 类型的一部分，因此删除器成员的类型在编译时是知道的，从而删除器可以直接保存在 `unique_ptr` 对象中。

`unique_ptr` 的析构函数与 `shared_ptr` 的析构函数类似，也是对其保存的指针调用用户提供的删除器或执行 `delete`：

```
// del 在编译时绑定；直接调用实例化的删除器  
del(p); // 无运行时额外开销
```

`del` 的类型或者是默认删除器类型，或者是用户提供的类型。到底是哪种情况没有关系，应该执行的代码在编译时肯定会知道。实际上，如果删除器是类似 `DebugDelete`（参见 16.1.4 节，第 595 页）之类的东西，这个调用甚至可能被编译为内联形式。

通过在编译时绑定删除器，`unique_ptr` 避免了间接调用删除器的运行时开销。通过在运行时绑定删除器，`shared_ptr` 使用用户重载删除器更为方便。

16.1.6 节练习

练习 16.28：编写你自己版本的 `shared_ptr` 和 `unique_ptr`。

练习 16.29：修改你的 `Blob` 类，用你自己的 `shared_ptr` 代替标准库中的版本。

练习 16.30：重新运行你的一些程序，验证你的 `shared_ptr` 类和修改后的 `Blob` 类。（注意：实现 `weak_ptr` 类型超出了本书范围，因此你不能将 `BlobPtr` 类与你修改后的 `Blob` 一起使用。）

练习 16.31：如果我们将 `DebugDelete` 与 `unique_ptr` 一起使用，解释编译器将删除器处理为内联形式的可能方式。

16.2 模板实参推断

我们已经看到，对于函数模板，编译器利用调用中的函数实参来确定其模板参数。从函数实参来确定模板实参的过程被称为模板实参推断（template argument deduction）。在模

板实参推断过程中，编译器使用函数调用中的实参类型来寻找模板实参，用这些模板实参生成的函数版本与给定的函数调用最为匹配。

16.2.1 类型转换与模板类型参数



与非模板函数一样，我们在一次调用中传递给函数模板的实参被用来初始化函数的形参。如果一个函数形参的类型使用了模板类型参数，那么它采用特殊的初始化规则。只有很有限的几种类型转换会自动地应用于这些实参。编译器通常不是对实参进行类型转换，而是生成一个新的模板实例。

与往常一样，顶层 `const`（参见 2.4.3 节，第 57 页）无论是在形参中还是在实参中，都会被忽略。在其他类型转换中，能在调用中应用于函数模板的包括如下两项。

- `const` 转换：可以将一个非 `const` 对象的引用（或指针）传递给一个 `const` 的引用（或指针）形参（参见 4.11.2 节，第 144 页）。
- 数组或函数指针转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的指针。类似的，一个函数实参可以转换为一个该函数类型的指针（参见 4.11.2 节，第 143 页）。

其他类型转换，如算术转换（参见 4.11.1 节，第 142 页）、派生类向基类的转换（参见 15.2.2 节，第 530 页）以及用户定义的转换（参见 7.5.4 节，第 263 页和 14.9 节，第 514 页），都不能应用于函数模板。

作为一个例子，考虑对函数 `fobj` 和 `oref` 的调用。`fobj` 函数拷贝它的参数，而 `oref` 的参数是引用类型：

```
template <typename T> T fobj(T, T); // 实参被拷贝
template <typename T> T oref(const T&, const T&); // 引用
string s1("a value");
const string s2("another value");
fobj(s1, s2); // 调用 fobj(string, string); const 被忽略
oref(s1, s2); // 调用 oref(const string&, const string&)
               // 将 s1 转换为 const 是允许的
int a[10], b[42];
fobj(a, b); // 调用 f(int*, int*)
oref(a, b); // 错误：数组类型不匹配
```

在第一对调用中，我们传递了一个 `string` 和一个 `const string`。虽然这些类型不严格匹配，但两个调用都是合法的。在 `fobj` 调用中，实参被拷贝，因此原对象是否是 `const` 没有关系。在 `oref` 调用中，参数类型是 `const` 的引用。对于一个引用参数来说，转换为 `const` 是允许的，因此这个调用也是合法的。

在下一对调用中，我们传递了数组实参，两个数组大小不同，因此是不同类型。在 `fobj` 调用中，数组大小不同无关紧要。两个数组都被转换为指针。`fobj` 中的模板类型为 `int*`。但是，`oref` 调用是不合法的。如果形参是一个引用，则数组不会转换为指针（参见 6.2.4 节，第 195 页）。`a` 和 `b` 的类型是不匹配的，因此调用是错误的。



将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有 `const` 转换及数组或函数到指针的转换。



使用相同模板参数类型的函数形参

一个模板类型参数可以用作多个函数形参的类型。由于只允许有限的几种类型转换，因此传递给这些形参的实参必须具有相同的类型。如果推断出的类型不匹配，则调用就是错误的。例如，我们的 `compare` 函数（参见 16.1.1 节，第 578 页）接受两个 `const T&` 参数，其实参必须是相同类型：

```
long lng;
compare(lng, 1024); // 错误：不能实例化 compare(long, int)
```

此调用是错误的，因为传递给 `compare` 的实参类型不同。从第一个函数实参推断出的模板实参为 `long`，从第二个函数实参推断出的模板实参为 `int`。这些类型不匹配，因此模板实参推断失败。

如果希望允许对函数实参进行正常的类型转换，我们可以将函数模板定义为两个类型参数：

```
// 实参类型可以不同，但必须兼容
template <typename A, typename B>
int flexibleCompare(const A& v1, const B& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

现在用户可以提供不同类型的实参了：

```
long lng;
flexibleCompare(lng, 1024); // 正确：调用 flexibleCompare(long, int)
```

当然，必须定义了能比较这些类型的值的`<`运算符。

正常类型转换应用于普通函数实参

函数模板可以有普通类型定义的参数，即，不涉及模板类型参数的类型。这种函数实参不进行特殊处理；它们正常转换为对应形参的类型（参见 6.1 节，第 183 页）。例如，考虑下面的模板：

```
template <typename T> ostream &print(ostream &os, const T &obj)
{
    return os << obj;
}
```

第一个函数参数是一个已知类型 `ostream&`。第二个参数 `obj` 则是模板参数类型。由于 `os` 的类型是固定的，因此当调用 `print` 时，传递给它的实参会进行正常的类型转换：

```
681 print(cout, 42); // 实例化 print(ostream&, int)
ofstream f("output");
print(f, 10); // 使用 print(ostream&, int); 将 f 转换为 ostream&
```

在第一个调用中，第一个实参的类型严格匹配第一个参数的类型。此调用会实例化接受一个 `ostream&` 和一个 `int` 的 `print` 版本。在第二个调用中，第一个实参是一个 `ofstream`，它可以转换为 `ostream&`（参见 8.2.1 节，第 284 页）。由于此参数的类型不依赖于模板参数，因此编译器会将 `f` 隐式转换为 `ostream&`。



如果函数参数类型不是模板参数，则对实参进行正常的类型转换。

16.2.1 节练习

练习 16.32: 在模板实参推断过程中发生了什么？

练习 16.33: 指出在模板实参推断过程中允许对函数实参进行的两种类型转换。

练习 16.34: 对下面的代码解释每个调用是否合法。如果合法， T 的类型是什么？如果不合法，为什么？

```
template <class T> int compare(const T&, const T&);  
(a) compare("hi", "world"); (b) compare("bye", "dad");
```

练习 16.35: 下面调用中哪些是错误的（如果有的话）？如果调用合法， T 的类型是什么？如果调用不合法，问题何在？

```
template <typename T> T calc(T, int);  
template <typename T> T fcn(T, T);  
double d; float f; char c;  
(a) calc(c, 'c'); (b) calc(d, f);  
(c) fcn(c, 'c'); (d) fcn(d, f);
```

练习 16.36: 进行下面的调用会发生什么：

```
template <typename T> f1(T, T);  
template <typename T1, typename T2> f2(T1, T2);  
int i = 0, j = 42, *p1 = &i, *p2 = &j;  
const int *cp1 = &i, *cp2 = &j;  
(a) f1(p1, p2); (b) f2(p1, p2); (c) f1(cp1, cp2);  
(d) f2(cp1, cp2); (e) f1(p1, cp1); (f) f2(p1, cp1);
```

16.2.2 函数模板显式实参

在某些情况下，编译器无法推断出模板实参的类型。其他一些情况下，我们希望允许用户控制模板实例化。当函数返回类型与参数列表中任何类型都不相同时，这两种情况最常出现。

< 682

指定显式模板实参

作为一个允许用户指定使用类型的例子，我们将定义一个名为 `sum` 的函数模板，它接受两个不同类型的参数。我们希望允许用户指定结果的类型。这样，用户就可以选择合适的精度。

我们可以定义表示返回类型的第三个模板参数，从而允许用户控制返回类型：

```
// 编译器无法推断 T1，它未出现在函数参数列表中  
template <typename T1, typename T2, typename T3>  
T1 sum(T2, T3);
```

在本例中，没有任何函数实参的类型可用来推断 $T1$ 的类型。每次调用 `sum` 时调用者都必须为 $T1$ 提供一个显式模板实参（explicit template argument）。

我们提供显式模板实参的方式与定义类模板实例的方式相同。显式模板实参在尖括号中给出，位于函数名之后，实参列表之前：

```
// T1 是显式指定的, T2 和 T3 是从函数实参类型推断而来的
auto val3 = sum<long long>(i, lng); // long long sum(int, long)
```

此调用显式指定 T1 的类型。而 T2 和 T3 的类型则由编译器从 i 和 lng 的类型推断出来。

显式模板实参按由左至右的顺序与对应的模板参数匹配；第一个模板实参与第一个模板参数匹配，第二个实参与第二个参数匹配，依此类推。只有尾部（最右）参数的显式模板实参才可以忽略，而且前提是它们可以从函数参数推断出来。如果我们的 sum 函数按照如下形式编写：

```
// 糟糕的设计：用户必须指定所有三个模板参数
template <typename T1, typename T2, typename T3>
T3 alternative_sum(T2, T1);
```

则我们总是必须为所有三个形参指定实参：

```
// 错误：不能推断前几个模板参数
auto val3 = alternative_sum<long long>(i, lng);
// 正确：显式指定了所有三个参数
auto val2 = alternative_sum<long long, int, long>(i, lng);
```

正常类型转换应用于显式指定的实参

对于用普通类型定义的函数参数，允许进行正常的类型转换（参见 16.2.1 节，第 602 页），出于同样的原因，对于模板类型参数已经显式指定了的函数实参，也进行正常的类型转换：

683

```
long lng;
compare(lng, 1024);           // 错误：模板参数不匹配
compare<long>(lng, 1024);    // 正确：实例化 compare(long, long)
compare<int>(lng, 1024);     // 正确：实例化 compare(int, int)
```

如我们所见，第一个调用是错误的，因为传递给 compare 的实参必须具有相同的类型。如果我们显式指定模板类型参数，就可以进行正常类型转换了。因此，调用 compare<long> 等价于调用一个接受两个 const long& 参数的函数。int 类型的参数被自动转化为 long。在第三个调用中，T 被显式指定为 int，因此 lng 被转换为 int。

16.2.2 节练习

练习 16.37：标准库 max 函数有两个参数，它返回实参中的较大者。此函数有一个模板类型参数。你能在调用 max 时传递给它一个 int 和一个 double 吗？如果可以，如何做？如果不可以，为什么？

练习 16.38：当我们调用 make_share（参见 12.1.1 节，第 401 页）时，必须提供一个显式模板实参。解释为什么需要显式模板实参以及它是如何使用的。

练习 16.39：对 16.1.1 节（第 578 页）中的原始版本的 compare 函数，使用一个显式模板实参，使得可以向函数传递两个字符串字面常量。



16.2.3 尾置返回类型与类型转换

当我们希望用户确定返回类型时，用显式模板实参表示模板函数的返回类型是很有效的。但在其他情况下，要求显式指定模板实参会给用户增添额外负担，而且不会带来什么好处。例如，我们可能希望编写一个函数，接受表示序列的一对迭代器和返回序列中一个

元素的引用:

```
template <typename It>
??? &fcn(It beg, It end)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的引用
}
```

我们并不知道返回结果的准确类型，但知道所需类型是所处理的序列的元素类型:

```
vector<int> vi = {1,2,3,4,5};
Blob<string> ca = { "hi", "bye" };
auto &i = fcn(vi.begin(), vi.end()); // fcn 应该返回 int&
auto &s = fcn(ca.begin(), ca.end()); // fcn 应该返回 string&
```

此例中，我们知道函数应该返回`*beg`，而且知道我们可以用`decltype(*beg)`来获取此表达式的类型。但是，在编译器遇到函数的参数列表之前，`beg`都是不存在的。为了定义此函数，我们必须使用尾置返回类型（参见 6.3.3 节，第 206 页）。由于尾置返回出现在参数列表之后，它可以使用函数的参数:

```
// 尾置返回允许我们在参数列表之后声明返回类型
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的引用
}
```

此例中我们通知编译器`fcn`的返回类型与解引用`beg`参数的结果类型相同。解引用运算符返回一个左值（参见 4.1.1 节，第 121 页），因此通过`decltype`推断的类型为`beg`表示的元素的类型的引用。因此，如果对一个`string`序列调用`fcn`，返回类型将是`string&`。如果是`int`序列，则返回类型是`int&`。

进行类型转换的标准库模板类

有时我们无法直接获得所需要的类型。例如，我们可能希望编写一个类似`fcn`的函数，但返回一个元素的值（参见 6.3.2 节，第 201 页）而非引用。

在编写这个函数的过程中，我们面临一个问题：对于传递的参数的类型，我们几乎一无所知。在此函数中，我们知道唯一可以使用的操作是迭代器操作，而所有迭代器操作都不会生成元素，只能生成元素的引用。

为了获得元素类型，我们可以使用标准库的**类型转换**（type transformation）模板。这些模板定义在头文件`type_traits`中。这个头文件中的类通常用于所谓的模板元程序设计，这一主题已超出本书的范围。但是，类型转换模板在普通编程中也很有用。表 16.1 列出了这些模板，我们将在 16.5 节（第 624 页）中看到它们是如何实现的。

在本例中，我们可以使用`remove_reference`来获得元素类型。`remove_reference`模板有一个模板类型参数和一个名为`type`的（public）类型成员。如果我们用一个引用类型实例化`remove_reference`，则`type`将表示被引用的类型。例如，如果我们实例化`remove_reference<int&>`，则`type`成员将是`int`。类似的，如果我们实例化`remove_reference<string&>`，则`type`成员将是`string`，依此类推。更一般的，给定一个迭代器`beg`:

< 684

C++
11

```
remove_reference<decltype(*beg)>::type
```

将获得 beg 引用的元素的类型： decltype(*beg) 返回元素类型的引用类型。 remove_reference::type 脱去引用，剩下元素类型本身。

组合使用 remove_reference、尾置返回及 decltype，我们就可以在函数中返回元素值的拷贝：

685 // 为了使用模板参数的成员，必须用 typename，参见 16.1.3 节（第 593 页）

```
template <typename It>
auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

注意，type 是一个类的成员，而该类依赖于一个模板参数。因此，我们必须在返回类型的声明中使用 typename 来告知编译器，type 表示一个类型（参见 16.1.3 节，第 593 页）

表 16.1：标准类型转换模板

对 Mod<T>，其中 Mod 为	若 T 为	则 Mod<T>::type 为
remove_reference	X&或 X&&	X
	否则	T
add_const	X&、const X 或函数	T
	否则	const T
add_lvalue_reference	X&	T
	X&&	X&
	否则	T&
add_rvalue_reference	X&或 X&&	T
	否则	T&&
remove_pointer	X*	X
	否则	T
add_pointer	X&或 X&&	X*
	否则	T*
make_signed	unsigned X	X
	否则	T
make_unsigned	带符号类型	unsigned X
	否则	T
remove_extent	X[n]	X
	否则	T
remove_all_extents	X[n1] [n2]...	X
	否则	T

表 16.1 中描述的每个类型转换模板的工作方式都与 remove_reference 类似。每个模板都有一个名为 type 的 public 成员，表示一个类型。此类型与模板自身的模板类型参数相关，其关系如模板名所示。如果不可能（或者不必要）转换模板参数，则 type 成员就是模板参数类型本身。例如，如果 T 是一个指针类型，则 remove_pointer<T>::type 是 T 指向的类型。如果 T 不是一个指针，则无须进行任何

转换，从而 `type` 具有与 `T` 相同的类型。

16.2.3 节练习

< 686

练习 16.40：下面的函数是否合法？如果不合法，为什么？如果合法，对可以传递的实参类型有什么限制（如果有的话）？返回类型是什么？

```
template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

练习 16.41：编写一个新的 `sum` 版本，它的返回类型保证足够大，足以容纳加法结果。

16.2.4 函数指针和实参推断



当我们用一个函数模板初始化一个函数指针或为一个函数指针赋值（参见 6.7 节，第 221 页）时，编译器使用指针的类型来推断模板实参。

例如，假定我们有一个函数指针，它指向的函数返回 `int`，接受两个参数，每个参数都是指向 `const int` 的引用。我们可以使用该指针指向 `compare` 的一个实例：

```
template <typename T> int compare(const T&, const T&);
// pf1 指向实例 int compare(const int&, const int&)
int (*pf1)(const int&, const int&) = compare;
```

`pf1` 中参数的类型决定了 `T` 的模板实参的类型。在本例中，`T` 的模板实参类型为 `int`。指针 `pf1` 指向 `compare` 的 `int` 版本实例。如果不能从函数指针类型确定模板实参，则产生错误：

```
// func 的重载版本；每个版本接受一个不同的函数指针类型
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));
func(compare); // 错误：使用 compare 的哪个实例？
```

这段代码的问题在于，通过 `func` 的参数类型无法确定模板实参的唯一类型。对 `func` 的调用既可以实例化接受 `int` 的 `compare` 版本，也可以实例化接受 `string` 的版本。由于不能确定 `func` 的实参的唯一实例化版本，此调用将编译失败。

我们可以通过使用显式模板实参来消除 `func` 调用的歧义：

```
// 正确：显式指出实例化哪个 compare 版本
func(compare<int>); // 传递 compare(const int&, const int&)
```

此表达式调用的 `func` 版本接受一个函数指针，该指针指向的函数接受两个 `const int&` 参数。



当参数是一个函数模板实例的地址时，程序上下文必须满足：对每个模板参数，能唯一确定其类型或值。

< 687

16.2.5 模板实参推断和引用

为了理解如何从函数调用进行类型推断，考虑下面的例子：

```
template <typename T> void f(T &p);
```

其中函数参数 p 是一个模板类型参数 T 的引用，非常重要的是记住两点：编译器会应用正常的引用绑定规则；const 是底层的，不是顶层的。

从左值引用函数参数推断类型

当一个函数参数是模板类型参数的一个普通（左值）引用时（即，形如 `T&`），绑定规则告诉我们，只能传递给它一个左值（如，一个变量或一个返回引用类型的表达式）。实参可以是 `const` 类型，也可以不是。如果实参是 `const` 的，则 T 将被推断为 `const` 类型：

```
template <typename T> void f1(T&); // 实参必须是一个左值
// 对 f1 的调用使用实参所引用的类型作为模板参数类型
f1(i); // i 是一个 int；模板参数类型 T 是 int
f1(ci); // ci 是一个 const int；模板参数 T 是 const int
f1(5); // 错误：传递给一个&参数的实参必须是一个左值
```

如果一个函数参数的类型是 `const T&`，正常的绑定规则告诉我们可以传递给它任何类型的实参——一个对象（`const` 或非 `const`）、一个临时对象或是一个字面常量值。当函数参数本身是 `const` 时，T 的类型推断的结果不会是一个 `const` 类型。`const` 已经是函数参数类型的一部分；因此，它不会也是模板参数类型的一部分：

```
template <typename T> void f2(const T&); // 可以接受一个右值
// f2 中的参数是 const &；实参中的 const 是无关的
// 在每个调用中，f2 的函数参数都被推断为 const int&
f2(i); // i 是一个 int；模板参数 T 是 int
f2(ci); // ci 是一个 const int，但模板参数 T 是 int
f2(5); // 一个 const &参数可以绑定到一个右值；T 是 int
```

从右值引用函数参数推断类型

当一个函数参数是一个右值引用（参见 13.6.1 节，第 471 页）（即，形如 `T&&`）时，正常绑定规则告诉我们可以传递给它一个右值。当我们这样做时，类型推断过程类似普通左值引用函数参数的推断过程。推断出的 T 的类型是该右值实参的类型：

```
template <typename T> void f3(T&&);
f3(42); // 实参是一个 int 类型的右值；模板参数 T 是 int
```

688 引用折叠和右值引用参数

假定 `i` 是一个 `int` 对象，我们可能认为像 `f3(i)` 这样的调用是不合法的。毕竟，`i` 是一个左值，而通常我们不能将一个右值引用绑定到一个左值上。但是，C++ 语言在正常绑定规则之外定义了两个例外规则，允许这种绑定。这两个例外规则是 `move` 这种标准库设施正确工作的基础。

第一个例外规则影响右值引用参数的推断如何进行。当我们将一个左值（如 `i`）传递给函数的右值引用参数，且此右值引用指向模板类型参数（如 `T&&`）时，编译器推断模板类型参数为实参的左值引用类型。因此，当我们调用 `f3(i)` 时，编译器推断 T 的类型为 `int&`，而非 `int`。

T 被推断为 `int&` 看起来好像意味着 `f3` 的函数参数应该是一个类型 `int&` 的右值引用。

通常，我们不能（直接）定义一个引用的引用（参见 2.3.1 节，第 46 页）。但是，通过类型别名（参见 2.5.1 节，第 60 页）或通过模板类型参数间接定义是可以的。

在这种情况下，我们可以使用第二个例外绑定规则：如果我们间接创建一个引用的引用，则这些引用形成了“折叠”。在所有情况下（除了一个例外），引用会折叠成一个普通的左值引用类型。在新标准中，折叠规则扩展到右值引用。只有一种特殊情况下引用会折叠成右值引用：右值引用的右值引用。即，对于一个给定类型 X ：

- $X\& \&$ 、 $X\& \&&$ 和 $X\&\& \&$ 都折叠成类型 $X\&$
- 类型 $X\&\& \&$ 折叠成 $X\&\&$



引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数。

如果将引用折叠规则和右值引用的特殊类型推断规则组合在一起，则意味着我们可以对一个左值调用 $f3$ 。当我们把一个左值传递给 $f3$ 的（右值引用）函数参数时，编译器推断 T 为一个左值引用类型：

```
f3(i); // 实参是一个左值；模板参数 T 是 int&
f3(ci); // 实参是一个左值；模板参数 T 是一个 const int&
```

当一个模板参数 T 被推断为引用类型时，折叠规则告诉我们函数参数 $T\&\&$ 折叠为一个左值引用类型。例如， $f3(i)$ 的实例化结果可能像下面这样：

```
// 无效代码，只是用于演示目的
void f3<int&>(int& &&); // 当 T 是 int& 时，函数参数为 int& &&
```

$f3$ 的函数参数是 $T\&\&$ 且 T 是 $int\&$ ，因此 $T\&\&$ 是 $int\& \&&$ ，会折叠成 $int\&$ 。因此，即使 $f3$ 的函数参数形式是一个右值引用（即， $T\&\&$ ），此调用也会用一个左值引用类型（即， $int\&$ ）实例化 $f3$ ：

```
void f3<int&>(int&); // 当 T 是 int& 时，函数参数折叠为 int&
```

这两个规则导致了两个重要结果：

- 如果一个函数参数是一个指向模板类型参数的右值引用（如， $T\&\&$ ），则它可以被绑定到一个左值；且
- 如果实参是一个左值，则推断出的模板实参类型将是一个左值引用，且函数参数将被实例化为一个（普通）左值引用参数（ $T\&$ ）

另外值得注意的是，这两个规则暗示，我们可以将任意类型的实参传递给 $T\&\&$ 类型的函数参数。对于这种类型的参数，（显然）可以传递给它右值，而如我们刚刚看到的，也可以传递给它左值。



如果一个函数参数是指向模板参数类型的右值引用（如， $T\&\&$ ），则可以传递给它任意类型的实参。如果将一个左值传递给这样的参数，则函数参数被实例化为一个普通的左值引用（ $T\&$ ）。

编写接受右值引用参数的模板函数

模板参数可以推断为一个引用类型，这一特性对模板内的代码可能有令人惊讶的影响：

```
template <typename T> void f3(T&& val)
{
    T t = val; // 拷贝还是绑定一个引用？
```

```
t = fcn(t); // 赋值只改变 t 还是既改变 t 又改变 val?
if (val == t) { /* ... */ } // 若 T 是引用类型，则一直为 true
}
```

当我们对一个右值调用 `f3` 时，例如字面常量 42，`T` 为 `int`。在此情况下，局部变量 `t` 的类型为 `int`，且通过拷贝参数 `val` 的值被初始化。当我们对 `t` 赋值时，参数 `val` 保持不变。

另一方面，当我们对一个左值 `i` 调用 `f3` 时，则 `T` 为 `int&`。当我们定义并初始化局部变量 `t` 时，赋予它类型 `int&`。因此，对 `t` 的初始化将其绑定到 `val`。当我们对 `t` 赋值时，也同时改变了 `val` 的值。在 `f3` 的这个实例化版本中，`if` 判断永远得到 `true`。

当代码中涉及的类型可能是普通（非引用）类型，也可能是引用类型时，编写正确的代码就变得异常困难（虽然 `remove_reference` 这样的类型转换类可能会有帮助（参见 16.2.3 节，第 605 页））。

在实际中，右值引用通常用于两种情况：模板转发其实参或模板被重载。我们将在 16.2.7 节（第 612 页）中介绍实参转发，在 16.3 节（第 614 页）中介绍模板重载。

目前应该注意的是，使用右值引用的函数模板通常使用我们在 13.6.3 节（第 481 页）中看到的方式来进行重载：

```
template <typename T> void f(T&&);           // 绑定到非 const 右值
template <typename T> void f(const T&);        // 左值和 const 右值
```

与非模板函数一样，第一个版本将绑定到可修改的右值，而第二个版本将绑定到左值或 `const` 右值。

690

16.2.5 节练习

练习 16.42: 对下面每个调用，确定 `T` 和 `val` 的类型：

```
template <typename T> void g(T&& val);
int i = 0; const int ci = i;
(a) g(i); (b) g(ci); (c) g(i * ci);
```

练习 16.43: 使用上一题定义的函数，如果我们调用 `g(i = ci)`，`g` 的模板参数将是什么？

练习 16.44: 使用与第一题中相同的三个调用，如果 `g` 的函数参数声明为 `T`（而不是 `T&&`），确定 `T` 的类型。如果 `g` 的函数参数是 `const T&` 呢？

练习 16.45: 给定下面的模板，如果我们对一个像 42 这样的字面常量调用 `g`，解释会发什么？如果我们对一个 `int` 类型的变量调用 `g` 呢？

```
template <typename T> void g(T&& val) { vector<T> v; }
```



16.2.6 理解 `std::move`

标准库 `move` 函数（参见 13.6.1 节，第 472 页）是使用右值引用的模板的一个很好的例子。幸运的是，我们不必理解 `move` 所使用的模板机制也可以直接使用它。但是，研究 `move` 是如何工作的可以帮助我们巩固对模板的理解和使用。

在 13.6.2 节（第 473 页）中我们注意到，虽然不能直接将一个右值引用绑定到一个左值上，但可以用 `move` 获得一个绑定到左值上的右值引用。由于 `move` 本质上可以接受任

何类型的实参，因此我们不会惊讶于它是一个函数模板。

std::move 是如何定义的

标准库是这样定义 move 的：

```
// 在返回类型和类型转换中也要用到 typename，参见 16.1.3 节（第 593 页）
// remove_reference 是在 16.2.3 节（第 605 页）中介绍的
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    // static_cast 是在 4.11.3 节（第 145 页）中介绍的
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

这段代码很短，但其中有些微妙之处。首先，move 的函数参数 `T&&` 是一个指向模板类型参数的右值引用。通过引用折叠，此参数可以与任何类型的实参匹配。特别是，我们既可以传递给 move 一个左值，也可以传递给它一个右值：

```
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // 正确：从一个右值移动数据
s2 = std::move(s1); // 正确：但在赋值之后，s1 的值是不确定的
```

std::move 是如何工作的

在第一个赋值中，传递给 move 的实参是 string 的构造函数的右值结果——`string("bye!")`。如我们已经见到过的，当向一个右值引用函数参数传递一个右值时，由实参推断出的类型为被引用的类型（参见 16.2.5 节，第 608 页）。因此，在 `std::move(string("bye!"))` 中：

- 推断出的 `T` 的类型为 `string`。
- 因此，`remove_reference` 用 `string` 进行实例化。
- `remove_reference<string>` 的 `type` 成员是 `string`。
- `move` 的返回类型是 `string&&`。
- `move` 的函数参数 `t` 的类型为 `string&&`。

因此，这个调用实例化 `move<string>`，即函数

```
string&& move(string &&t)
```

函数体返回 `static_cast<string&&>(t)`。`t` 的类型已经是 `string&&`，于是类型转换什么都不做。因此，此调用的结果就是它所接受的右值引用。

现在考虑第二个赋值，它调用了 `std::move()`。在此调用中，传递给 move 的实参是一个左值。这样：

- 推断出的 `T` 的类型为 `string&` (`string` 的引用，而非普通 `string`)。
- 因此，`remove_reference` 用 `string&` 进行实例化。
- `remove_reference<string&>` 的 `type` 成员是 `string`。
- `move` 的返回类型仍是 `string&&`。
- `move` 的函数参数 `t` 实例化为 `string& &&`，会折叠为 `string&`。

因此，这个调用实例化 `move<string&>`，即

```
string&& move(string &t)
```



691

这正是我们所寻求的——我们希望将一个右值引用绑定到一个左值。这个实例的函数体返回 `static_cast<string&&>(t)`。在此情况下，`t` 的类型为 `string&`，`cast` 将其转换为 `string&&`。

从一个左值 `static_cast` 到一个右值引用是允许的

通常情况下，`static_cast` 只能用于其他合法的类型转换（参见 4.11.3 节，第 145 页）。但是，这里又有一条针对右值引用的特许规则：虽然不能隐式地将一个左值转换为右值引用，但我们可以用 `static_cast` 显式地将一个左值转换为一个右值引用。

对于操作右值引用的代码来说，将一个右值引用绑定到一个左值的特性允许它们截断左值。有时候，例如在我们的 `StrVec` 类的 `reallocate` 函数（参见 13.6.1 节，第 469 页）中，我们知道截断一个左值是安全的。一方面，通过允许进行这样的转换，C++ 语言认可了这种用法。但另一方面，通过强制使用 `static_cast`，C++ 语言试图阻止我们意外地进行这种转换。

最后，虽然我们可以直接编写这种类型转换代码，但使用标准库 `move` 函数是容易得多的方式。而且，统一使用 `std::move` 使得我们在程序中查找潜在的截断左值的代码变得很容易。

16.2.6 节练习

练习 16.46: 解释下面的循环，它来自 13.5 节（第 469 页）中的 `StrVec::reallocate`:

```
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
```

16.2.7 转发

某些函数需要将其一个或多个实参连同类型不变地转发给其他函数。在此情况下，我们需要保持被转发实参的所有性质，包括实参类型是否是 `const` 的以及实参是左值还是右值。

作为一个例子，我们将编写一个函数，它接受一个可调用表达式和两个额外实参。我们的函数将调用给定的可调用对象，将两个额外参数逆序传递给它。下面是我们的翻转函数的初步模样：

```
// 接受一个可调用对象和另外两个参数的模板
// 对“翻转”的参数调用给定的可调用对象
// flip1 是一个不完整的实现：顶层 const 和引用丢失了
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}
```

这个函数一般情况下工作得很好，但当我们希望用它调用一个接受引用参数的函数时就会出现问题：

```
void f(int v1, int &v2) // 注意 v2 是一个引用
{
    cout << v1 << " " << ++v2 << endl;
}
```

在这段代码中，`f` 改变了绑定到 `v2` 的实参的值。但是，如果我们通过 `flip1` 调用 `f`，`f` 所做的改变就不会影响实参：

```
f(42, i);           // f 改变了实参 i
flip1(f, j, 42); // 通过 flip1 调用 f 不会改变 j
```

问题在于 `j` 被传递给 `flip1` 的参数 `t1`。此参数是一个普通的、非引用的类型 `int`，而非 `int&`。因此，这个 `flip1` 调用会实例化为

```
void flip1(void(*fcn)(int, int&), int t1, int t2);
```

`j` 的值被拷贝到 `t1` 中。`f` 中的引用参数被绑定到 `t1`，而非 `j`，从而其改变不会影响 `j`。

定义能保持类型信息的函数参数

为了通过翻转函数传递一个引用，我们需要重写函数，使其参数能保持给定实参的“左值性”。更进一步，可以想到我们也希望保持参数的 `const` 属性。

通过将一个函数参数定义为一个指向模板类型参数的右值引用，我们可以保持其对应实参的所有类型信息。而使用引用参数（无论是左值还是右值）使得我们可以保持 `const` 属性，因为在引用类型中的 `const` 是底层的。如果我们将函数参数定义为 `T1&&` 和 `T2&&`，通过引用折叠（参见 16.2.5 节，第 608 页）就可以保持翻转实参的左值/右值属性（参见 16.2.5 节，第 608 页）：

```
template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}
```

与较早的版本一样，如果我们调用 `flip2(f, j, 42)`，将传递给参数 `t1` 一个左值 `j`。但是，在 `flip2` 中，推断出的 `T1` 的类型为 `int&`，这意味着 `t1` 的类型会折叠为 `int&`。由于是引用类型，`t1` 被绑定到 `j` 上。当 `flip2` 调用 `f` 时，`f` 中的引用参数 `v2` 被绑定到 `t1`，也就是被绑定到 `j`。当 `f` 递增 `v2` 时，它也同时改变了 `j` 的值。



如果一个函数参数是指向模板类型参数的右值引用（如 `T&&`），它对应的实参的 `const` 属性和左值/右值属性将得到保持。

这个版本的 `flip2` 解决了一半问题。它对于接受一个左值引用的函数工作得很好，但不能用于接受右值引用参数的函数。例如：

```
void g(int &&i, int& j)
{
    cout << i << " " << j << endl;
}
```

如果我们试图通过 `flip2` 调用 `g`，则参数 `t2` 将被传递给 `g` 的右值引用参数。即使我们传递一个右值给 `flip2`：

```
flip2(g, i, 42); // 错误：不能从一个左值实例化 int&
```

传递给 `g` 的将是 `flip2` 中名为 `t2` 的参数。函数参数与其他任何变量一样，都是左值表达式（参见 13.6.1 节，第 471 页）。因此，`flip2` 中对 `g` 的调用将传递给 `g` 的右值引用参数一个左值。

在调用中使用 std::forward 保持类型信息

694 >

我们可以使用一个名为 `forward` 的新标准库设施来传递 `flip2` 的参数，它能保持原始实参的类型。类似 `move`, `forward` 定义在头文件 `utility` 中。与 `move` 不同，`forward` 必须通过显式模板实参来调用（参见 16.2.2 节，第 603 页）。`forward` 返回该显式实参类型的右值引用。即，`forward<T>` 的返回类型是 `T&&`。

C++ 11

通常情况下，我们使用 `forward` 传递那些定义为模板类型参数的右值引用的函数参数。通过其返回类型上的引用折叠，`forward` 可以保持给定实参的左值/右值属性：

```
template <typename Type> intermediary(Type &&arg)
{
    finalFcn(std::forward<Type>(arg));
    // ...
}
```

本例中我们使用 `Type` 作为 `forward` 的显式模板实参类型，它是从 `arg` 推断出来的。由于 `arg` 是一个模板类型参数的右值引用，`Type` 将表示传递给 `arg` 的实参的所有类型信息。如果实参是一个右值，则 `Type` 是一个普通（非引用）类型，`forward<Type>` 将返回 `Type&&`。如果实参是一个左值，则通过引用折叠，`Type` 本身是一个左值引用类型。在此情况下，返回类型是一个指向左值引用类型的右值引用。再次对 `forward<Type>` 的返回类型进行引用折叠，将返回一个左值引用类型。



当用于一个指向模板参数类型的右值引用函数参数 (`T&&`) 时，`forward` 会保持实参类型的所有细节。

使用 `forward`，我们可以再次重写翻转函数：

```
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}
```

如果我们调用 `flip(g, i, 42)`，`i` 将以 `int&` 类型传递给 `g`，`42` 将以 `int&&` 类型传递给 `g`。



与 `std::move` 相同，对 `std::forward` 不使用 `using` 声明是一个好主意。我们将在 18.2.3 节（第 706 页）中解释原因。

16.2.7 节练习

练习 16.47： 编写你自己版本的翻转函数，通过调用接受左值和右值引用参数的函数来测试它。



16.3 重载与模板

函数模板可以被另一个模板或一个普通非模板函数重载。与往常一样，名字相同的函数必须具有不同数量或类型的参数。

695 >

如果涉及函数模板，则函数匹配规则（参见 6.4 节，第 209 页）会在以下几方面受到

影响：

- 对于一个调用，其候选函数包括所有模板实参推断（参见 16.2 节，第 600 页）成功的函数模板实例。
- 候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板。
- 与往常一样，可行函数（模板与非模板）按类型转换（如果对此调用需要的话）来排序。当然，可以用于函数模板调用的类型转换是非常有限的（参见 16.2.1 节，第 601 页）。
- 与往常一样，如果恰有一个函数提供比任何其他函数都更好的匹配，则选择此函数。但是，如果有多个函数提供同样好的匹配，则：
 - 如果同样好的函数中只有一个是非模板函数，则选择此函数。
 - 如果同样好的函数中没有非模板函数，而有多个函数模板，且其中一个模板比其他模板更特例化，则选择此模板。
 - 否则，此调用有歧义。



正确定义一组重载的函数模板需要对类型间的关系及模板函数允许的有限的实参类型转换有深刻的理解。

编写重载模板

作为一个例子，我们将构造一组函数，它们在调试中可能很有用。我们将这些调试函数命名为 `debug_rep`，每个函数都返回一个给定对象的 `string` 表示。我们首先编写此函数的最通用版本，将它定义为一个模板，接受一个 `const` 对象的引用：

```
// 打印任何我们不能处理的类型
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret; // 参见 8.3 节 (第 287 页)
    ret << t; // 使用 T 的输出运算符打印 t 的一个表示形式
    return ret.str(); // 返回 ret 绑定的 string 的一个副本
}
```

此函数可以用来生成一个对象对应的 `string` 表示，该对象可以是任意具备输出运算符的类型。 ◀ 696

接下来，我们将定义打印指针的 `debug_rep` 版本：

```
// 打印指针的值，后跟指针指向的对象
// 注意：此函数不能用于 char*；参见 16.3 节 (第 617 页)
template <typename T> string debug_rep(T *p)
{
    ostringstream ret;
    ret << "pointer: " << p; // 打印指针本身
    if (p)
        ret << " " << debug_rep(*p); // 打印 p 指向的值
    else
        ret << " null pointer"; // 或指出 p 为空
    return ret.str(); // 返回 ret 绑定的 string 的一个副本
}
```

此版本生成一个 `string`，包含指针本身值和调用 `debug_rep` 获得的指针指向的值。注意此函数不能用于打印字符指针，因为 IO 库为 `char*` 值定义了一个 `<<` 版本。此 `<<` 版本假定指针表示一个空字符结尾的字符数组，并打印数组的内容而非地址值。我们将在 16.3

节（第 617 页）介绍如何处理字符指针。

我们可以这样使用这些函数：

```
string s("hi");
cout << debug_rep(s) << endl;
```

对于这个调用，只有第一个版本的 `debug_rep` 是可行的。第二个 `debug_rep` 版本要求一个指针参数，但在此调用中我们传递的是一个非指针对象。因此编译器无法从一个非指针实参实例化一个期望指针类型参数的函数模板，因此实参推断失败。由于只有一个可行函数，所以此函数被调用。

如果我们用一个指针调用 `debug_rep`：

```
cout << debug_rep(&s) << endl;
```

两个函数都生成可行的实例：

- `debug_rep(const string*&)`，由第一个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `string*`。
- `debug_rep(string*)`，由第二个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `string`。

第二个版本的 `debug_rep` 的实例是此调用的精确匹配。第一个版本的实例需要进行普通指针到 `const` 指针的转换。正常函数匹配规则告诉我们应该选择第二个模板，实际上编译器确实选择了这个版本。

697 多个可行模板

作为另外一个例子，考虑下面的调用：

```
const string *sp = &s;
cout << debug_rep(sp) << endl;
```

此例中的两个模板都是可行的，而且两个都是精确匹配：

- `debug_rep(const string*&)`，由第一个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `string*`。
- `debug_rep(const string*)`，由第二个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `const string`。

在此情况下，正常函数匹配规则无法区分这两个函数。我们可能觉得这个调用将是有歧义的。但是，根据重载函数模板的特殊规则，此调用被解析为 `debug_rep(T*)`，即，更特例化的版本。

设计这条规则的原因是，没有它，将无法对一个 `const` 的指针调用指针版本的 `debug_rep`。问题在于模板 `debug_rep(const T&)` 本质上可以用于任何类型，包括指针类型。此模板比 `debug_rep(T*)` 更通用，后者只能用于指针类型。没有这条规则，传递 `const` 的指针的调用永远是有歧义的。



当有多个重载模板对一个调用提供同样好的匹配时，应选择最特例化的版本。

非模板和模板重载

作为下一个例子，我们将定义一个普通非模板版本的 `debug_rep` 来打印双引号包围

的 string:

```
// 打印双引号包围的 string
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
```

现在, 当我们对一个 string 调用 debug_rep 时:

```
string s("hi");
cout << debug_rep(s) << endl;
```

有两个同样好的可行函数:

- `debug_rep<string>(const string&)`, 第一个模板, T 被绑定到 `string*`。
- `debug_rep(const string&)`, 普通非模板函数。

在本例中, 两个函数具有相同的参数列表, 因此显然两者提供同样好的匹配。但是, 编译器会选择非模板版本。698 当存在多个同样好的函数模板时, 编译器选择最特例化的版本, 出于相同的原因, 一个非模板函数比一个函数模板更好。



对于一个调用, 如果一个非函数模板与一个函数模板提供同样好的匹配, 则选择非模板版本。

重载模板和类型转换

还有一种情况我们到目前为止尚未讨论: C 风格字符串指针和字符串字面常量。现在有了一个接受 string 的 debug_rep 版本, 我们可能期望一个传递字符串的调用会匹配这个版本。但是, 考虑这个调用:

```
cout << debug_rep("hi world!") << endl; // 调用 debug_rep(T*)
```

本例中所有三个 debug_rep 版本都是可行的:

- `debug_rep(const T&)`, T 被绑定到 `char[10]`。
- `debug_rep(T*)`, T 被绑定到 `const char`。
- `debug_rep(const string&)`, 要求从 `const char*` 到 `string` 的类型转换。

对给定实参来说, 两个模板都提供精确匹配——第二个模板需要进行一次(许可的)数组到指针的转换, 而对于函数匹配来说, 这种转换被认为是精确匹配(参见 6.6.1 节, 第 219 页)。非模板版本是可行的, 但需要进行一次用户定义的类型转换, 因此它没有精确匹配那么好, 所以两个模板成为可能调用的函数。与之前一样, T* 版本更加特例化, 编译器会选择它。

如果我们希望将字符指针按 string 处理, 可以定义另外两个非模板重载版本:

```
// 将字符指针转换为 string, 并调用 string 版本的 debug_rep
string debug_rep(char *p)
{
    return debug_rep(string(p));
}
string debug_rep(const char *p)
{
    return debug_rep(string(p));
}
```

缺少声明可能导致程序行为异常

值得注意的是，为了使 `char*` 版本的 `debug_rep` 正确工作，在定义此版本时，`debug_rep(const string&)` 的声明必须在作用域中。否则，就可能调用错误的 `debug_rep` 版本：

```
699> template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
// 为了使 debug_rep(char*) 的定义正确工作，下面的声明必须在作用域中
string debug_rep(const string &);
string debug_rep(char *p)
{
    // 如果接受一个 const string& 的版本的声明不在作用域中，
    // 返回语句将调用 debug_rep(const T&) 的 T 实例化为 string 的版本
    return debug_rep(string(p));
}
```

通常，如果使用了一个忘记声明的函数，代码将编译失败。但对于重载函数模板的函数而言，则不是这样。如果编译器可以从模板实例化出与调用匹配的版本，则缺少的声明就不重要了。在本例中，如果忘记了声明接受 `string` 参数的 `debug_rep` 版本，编译器会默默地实例化接受 `const T&` 的模板版本。



Tip 在定义任何函数之前，记得声明所有重载的函数版本。这样就不必担心编译器由于未遇到你希望调用的函数而实例化一个并非你所需的版本。

16.3 节练习

练习 16.48： 编写你自己版本的 `debug_rep` 函数。

练习 16.49： 解释下面每个调用会发生什么：

```
template <typename T> void f(T);
template <typename T> void f(const T* );
template <typename T> void g(T);
template <typename T> void g(T* );
int i = 42, *p = &i;
const int ci = 0, *p2 = &ci;
g(42); g(p); g(ci); g(p2);
f(42); f(p); f(ci); f(p2);
```

练习 16.50： 定义上一个练习中的函数，令它们打印一条身份信息。运行该练习中的代码。如果函数调用的行为与你预期不符，确定你理解了原因。

16.4 可变参数模板

C++ 11

一个可变参数模板（variadic template）就是一个接受可变数目参数的模板函数或模板类。可变数目的参数被称为参数包（parameter packet）。存在两种参数包：模板参数包（template parameter packet），表示零个或多个模板参数；函数参数包（function parameter packet），表示零个或多个函数参数。

我们用一个省略号来指出一个模板参数或函数参数表示一个包。在一个模板参数列表

中，`class...`或`typename...`指出接下来的参数表示零个或多个类型的列表；一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表。在函数参数列表中，如果一个参数的类型是一个模板参数包，则此参数也是一个函数参数包。例如：

```
// Args 是一个模板参数包；rest 是一个函数参数包
// Args 表示零个或多个模板类型参数
// rest 表示零个或多个函数参数
template <typename T, typename... Args>
void foo(const T &t, const Args& ... rest);
```

声明了`foo`是一个可变参数函数模板，它有一个名为`T`的类型参数，和一个名为`Args`的模板参数包。这个包表示零个或多个额外的类型参数。`foo`的函数参数列表包含一个`const &`类型的参数，指向`T`的类型，还包含一个名为`rest`的函数参数包，此包表示零个或多个函数参数。

与往常一样，编译器从函数的实参推断模板参数类型。对于一个可变参数模板，编译器还会推断包中参数的数目。例如，给定下面的调用：

```
int i = 0; double d = 3.14; string s = "how now brown cow";
foo(i, s, 42, d);      // 包中有三个参数
foo(s, 42, "hi");      // 包中有两个参数
foo(d, s);              // 包中有一个参数
foo("hi");              // 空包
```

编译器会为`foo`实例化出四个不同的版本：

```
void foo(const int&, const string&, const int&, const double&);
void foo(const string&, const int&, const char[3]&);
void foo(const double&, const string&);
void foo(const char[3]&);
```

在每个实例中，`T`的类型都是从第一个实参的类型推断出来的。剩下的实参（如果有的话）提供函数额外实参的数目和类型。

sizeof...运算符

当我们需要知道包中有多少元素时，可以使用`sizeof...`运算符。类似`sizeof`（参见4.9节，第139页），`sizeof...`也返回一个常量表达式（参见2.4.4节，第58页），而且不会对其实参求值：

```
template<typename ... Args> void g(Args ... args) {
    cout << sizeof...(Args) << endl; // 类型参数的数目
    cout << sizeof...(args) << endl; // 函数参数的数目
}
```

16.4 节练习

C++
11

练习 16.51：调用本节中的每个`foo`，确定`sizeof...(Args)`和`sizeof...(rest)`分别返回什么。

练习 16.52：编写一个程序验证上一题的答案。

701

16.4.1 编写可变参数函数模板

如 6.2.6 节（第 198 页）所述，我们可以使用一个 `initializer_list` 来定义一个可接受可变数目实参的函数。但是，所有实参必须具有相同的类型（或它们的类型可以转换为同一个公共类型）。当我们既不知道想要处理的实参的数目也不知道它们的类型时，可变参数函数是很有用的。作为一个例子，我们将定义一个函数，它类似较早的 `error_msg` 函数，差别仅在于新函数实参的类型也是可变的。我们首先定义一个名为 `print` 的函数，它在一个给定流上打印给定实参列表的内容。

可变参数函数通常是递归的（参见 6.3.2 节，第 204 页）。第一步调用处理包中的第一个实参，然后用剩余实参调用自身。我们的 `print` 函数也是这样的模式，每次递归调用将第二个实参打印到第一个实参表示的流中。为了终止递归，我们还需要定义一个非可变参数的 `print` 函数，它接受一个流和一个对象：

```
// 用来终止递归并打印最后一个元素的函数
// 此函数必须在可变参数版本的 print 定义之前声明
template<typename T>
ostream &print(ostream &os, const T &t)
{
    return os << t; // 包中最后一个元素之后不打印分隔符
}
// 包中除了最后一个元素之外的其他元素都会调用这个版本的 print
template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
    os << t << ", "; // 打印第一个实参
    return print(os, rest...); // 递归调用，打印其他实参
}
```

第一个版本的 `print` 负责终止递归并打印初始调用中的最后一个实参。第二个版本的 `print` 是可变参数版本，它打印绑定到 `t` 的实参，并调用自身来打印函数参数包中的剩余值。

这段程序的关键部分是可变参数函数中对 `print` 的调用：

```
return print(os, rest...); // 递归调用，打印其他实参
```

我们的可变参数版本的 `print` 函数接受三个参数：一个 `ostream&`，一个 `const T&` 和一个参数包。而此调用只传递了两个实参。其结果是 `rest` 中的第一个实参被绑定到 `t`，剩余实参形成下一个 `print` 调用的参数包。因此，在每个调用中，包中的第一个实参被移除，成为绑定到 `t` 的实参。即，给定：

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

递归会执行如下：

调用	t	rest...
print(cout, i, s, 42)	i	s, 42
print(cout, s, 42)	s	42
print(cout, 42) 调用非可变参数版本的 print		

前两个调用只能与可变参数版本的 `print` 匹配，非可变参数版本是不可行的，因为这两个调用分别传递四个和三个实参，而非可变参数 `print` 只接受两个实参。

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_rep(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

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 ccccccccc 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）：

```
// 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]);
```

< 707

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，接受字符串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>`。710 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：

```
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 (instantiation) 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。

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 (non-type 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，来自动生成一个函数版本。

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...`）

第IV部分

高级主题

内容

第 17 章 标准库特殊设施	635
第 18 章 用于大型程序的工具	683
第 19 章 特殊工具与技术	725

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地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。

第 17 章

标准库特殊设施

内容

17.1 tuple 类型	636
17.2 bitset 类型	640
17.3 正则表达式	645
17.4 随机数	659
17.5 IO 库再探	666
小结	680
术语表	680

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

<code>tuple<T1, T2, ..., Tn> t;</code>	<code>t</code> 是一个 tuple，成员数为 n ，第 i 个成员的类型为 T_i 。 所有成员都进行值初始化（参见 3.3.1 节，第 88 页）
<code>tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);</code>	<code>t</code> 是一个 tuple，成员类型为 $T_1 \dots T_n$ ，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）
<code>make_tuple(v1, v2, ..., vn)</code>	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
<code>t1 == t2</code>	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
<code>t1 != t2</code>	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 <code>t1</code> 的成员和 <code>t2</code> 中的对应成员
<code>t1 < t2</code>	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
<code>get<i>(t)</code>	返回 <code>t</code> 的第 i 个数据成员的引用；如果 <code>t</code> 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
<code>tuple_size<tupleType>::value</code>	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
<code>tuple_element<i, tupleType>::type</code>	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：

```
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
```

< 720

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）。

`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 723→ `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。

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：

```
// 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 0000101111011101111
// 在 64 位机器中，long long OULL 是 64 个 0 比特，因此~OULL 是 64 个 1
bitset<128> bitvec3(~OULL); // 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。



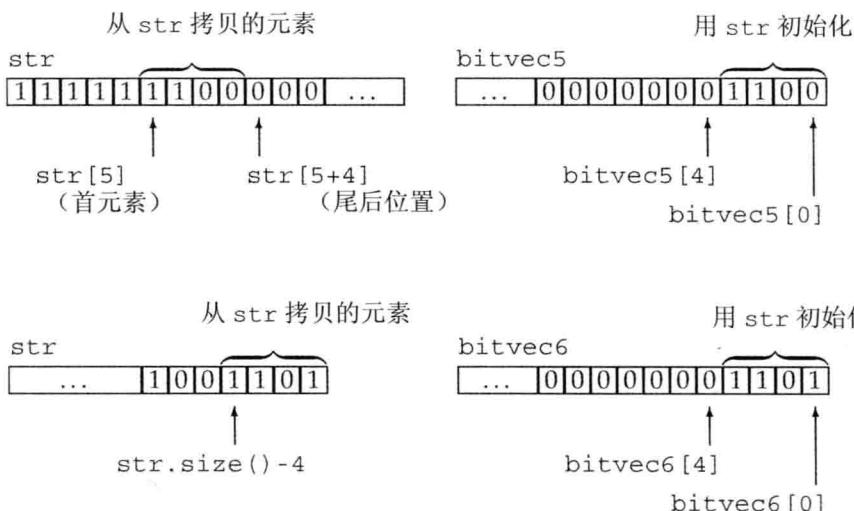
`string` 的下标编号习惯与 `bitset` 恰好相反：`string` 中下标最大的字符（最右字符）用来初始化 `bitset` 中的低位（下标为 0 的二进制位）。当你用一个 `string` 初始化一个 `bitset` 时，要记住这个差别。

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：

- `bitset<64> bitvec(32);`
- `bitset<32> bv(1010101);`
- `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。

表 17.3: bitset 操作

726

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)	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.set()	将位置 pos 处的位复位或将 b 中所有位复位
b.reset(pos)	将位置 pos 处的位复位或将 b 中所有位复位
b.reset()	
b.flip(pos)	改变位置 pos 处的位的状态或改变 b 中每一位的状态
b.flip()	
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()	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_ullong()	
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
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` 替代 `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

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

`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。

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“receipt”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

<code>regex r(re)</code>	<code>re</code> 表示一个正则表达式，它可以是一个 <code>string</code> 、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。 <code>f</code> 是指出对象如何处理的标志。 <code>f</code> 通过下面列出的值来设置。如果未指定 <code>f</code> ，其默认值为 ECMAScript
<code>r1 = re</code>	将 <code>r1</code> 中的正则表达式替换为 <code>re</code> 。 <code>re</code> 表示一个正则表达式，它可以是另一个 <code>regex</code> 对象、一个 <code>string</code> 、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
<code>r1.assign(re, f)</code>	与使用赋值运算符 (<code>=</code>) 效果相同；可选的标志 <code>f</code> 也与 <code>regex</code> 的构造函数中对应的参数含义相同
<code>r.mark_count()</code>	<code>r</code> 中子表达式的数目（我们将在 17.3.3 节（第 654 页）中介绍）
<code>r.flags()</code>	返回 <code>r</code> 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

<code>icase</code>	在匹配过程中忽略大小写
<code>nosubs</code>	不保存匹配的子表达式
<code>optimize</code>	执行速度优先于构造速度
<code>ECMAScript</code>	使用 ECMA-262 指定的语法
<code>basic</code>	使用 POSIX 基本的正则表达式语法
<code>extended</code>	使用 POSIX 扩展的正则表达式语法
<code>awk</code>	使用 POSIX 版本的 <code>awk</code> 语言的语法
<code>grep</code>	使用 POSIX 版本的 <code>grep</code> 的语法
<code>egrep</code>	使用 POSIX 版本的 <code>egrep</code> 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() << "\nerror: " << 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 <code>regex</code> 和 <code>regex_constants::error_type</code> 中	
<code>error_collate</code>	无效的元素校对请求
<code>error_ctype</code>	无效的字符类
<code>error_escape</code>	无效的转义字符或无效的尾置转义
<code>error_backref</code>	无效的向后引用
<code>error_brack</code>	不匹配的方括号 ([或])
<code>error_paren</code>	不匹配的小括号 ((或))
<code>error_brace</code>	不匹配的花括号 ({或})
<code>error_badbrace</code>	{ } 中无效的范围
<code>error_range</code>	无效的字符范围 (如[z-a])
<code>error_space</code>	内存不足，无法处理此正则表达式
<code>error_badrepeat</code>	重复字符 (*、?、+或{}) 之前没有有效的正则表达式
<code>error_complexity</code>	要求的匹配过于复杂
<code>error_stack</code>	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，733 编号从 0 开始。

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 `wchar_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	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	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	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
it1 != it2	

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

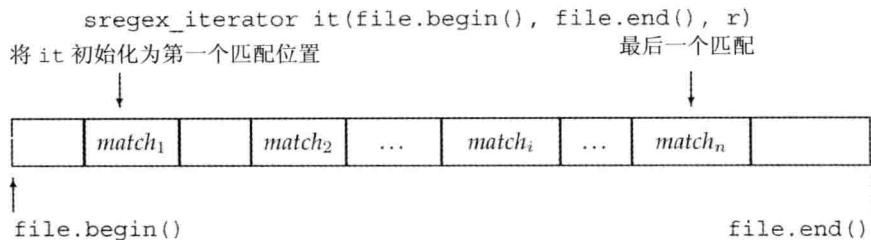
作为一个例子, 我们将扩展之前的程序, 在一个文本文件中查找所有违反 “i 在 e 之前, 除非在 c 之后” 规则的单词。我们假定名为 file 的 string 保存了我们要搜索的输入文件的全部内容。这个版本的程序将使用与前一个版本一样的 pattern, 但会使用一个 sregex_iterator 来进行搜索:

```
// 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
```

735

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

如果我们将最初版本程序中的 `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

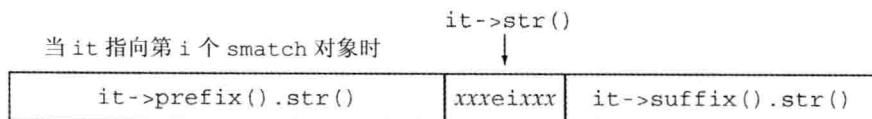
```
hey read or write according to the type
    >>> being <<<
handled. The input operators ignore whi
```

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`。

<code>m.ready()</code>	如果已经通过调用 <code>regex_serach</code> 或 <code>regex_match</code> 设置了 <code>m</code> , 则返回 <code>true</code> ; 否则返回 <code>false</code> 。如果 <code>ready</code> 返回 <code>false</code> , 则对 <code>m</code> 进行操作是未定义的
<code>m.size()</code>	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
<code>m.empty()</code>	若 <code>m.size()</code> 为 0, 则返回 <code>true</code>
<code>m.prefix()</code>	一个 <code>ssub_match</code> 对象, 表示当前匹配之前的序列
<code>m.suffix()</code>	一个 <code>ssub_match</code> 对象, 表示当前匹配之后的部分
<code>m.format(...)</code>	见表 17.12 (第 657 页)
在接受一个索引的操作中, <code>n</code> 的默认值为 0 且必须小于 <code>m.size()</code> 。	
第一个子匹配 (索引为 0) 表示整个匹配。	
<code>m.length(n)</code>	第 <code>n</code> 个匹配的子表达式的大小
<code>m.position(n)</code>	第 <code>n</code> 个子表达式距序列开始的距离
<code>m.str(n)</code>	第 <code>n</code> 个子表达式匹配的 <code>string</code>
<code>m[n]</code>	对应第 <code>n</code> 个子表达式的 <code>ssub_match</code> 对象
<code>m.begin(), m.end()</code>	表示 <code>m</code> 中 <code>sub_match</code> 元素范围的迭代器。与往常一样, <code>cbegin</code> 和 <code>cend</code> 返回 <code>const_iterator</code>

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

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。

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：

- 739
- `\{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}`。

为了验证电话号码，我们需要访问模式的组成部分。例如，我们希望验证区号部分的数字如果用了左括号，那么它是否也在区号后面用了右括号。即，我们不希望出现 (908.555.1800 这样的号码。

为了获得匹配的组成部分，我们需要在定义正则表达式时使用子表达式。每个子表达式用一对括号包围：

```
// 整个正则表达式包含七个子表达式：(ddd)分隔符ddd分隔符dddd
// 子表达式1、3、4和6是可选的；2、5和7保存号码
"(\(\)?(\d{3})(\))?( [-. ])?(\d{3})( [-. ]?) (\d{4})";
```

由于我们的模式使用了括号，而且必须去除反斜线的特殊含义，因此这个模式很难读（也很难写！）。理解此模式的最简单的方法是逐个剥离（括号包围的）子表达式：

1. (\(\)?表示区号部分可选的左括号
2. (\d{3})表示区号
3. (\\))?表示区号部分可选的右括号
4. ([-.])?表示区号部分可选的分隔符
5. (\d{3})表示号码的下三位数字
6. ([-.])?表示可选的分隔符
7. (\d{4})表示号码的最后四位数字

下面的代码读取一个文件，并用此模式查找与完整的电话号码模式匹配的数据。它会调用一个名为 `valid` 的函数来检查号码格式是否合法：740

```
string phone =
    "(\(\)?(\d{3})(\))?( [-. ])?(\d{3})( [-. ]?) (\d{4})";
regex r(phone); // regex对象，用于查找我们的模式
smatch m;
string s;
// 从输入文件中读取每条记录
while (getline(cin, s)) {
    // 对每个匹配的电话号码
    for (sregex_iterator it(s.begin(), s.end(), r), end_it;
         it != end_it; ++it)
        // 检查号码的格式是否合法
        if (valid(*it))
            cout << "valid: " << it->str() << endl;
        else
            cout << "not valid: " << it->str() << endl;
}
```

使用子匹配操作

我们将使用表 17.11 中描述的子匹配操作来编写 `valid` 函数。需要记住的重要一点是，我们的 `pattern` 有七个子表达式。与往常一样，每个 `smatch` 对象会包含八个 `ssub_match` 元素。位置 [0] 的元素表示整个匹配；元素 [1]...[7] 表示每个对应的子表达式。

当调用 `valid` 时，我们知道已经有一个完整的匹配，但不知道每个可选的子表达式是否是匹配的一部分。如果一个子表达式是完整匹配的一部分，则其对应的 `ssub_match` 对象的 `matched` 成员为 `true`。

表 17.11：子匹配操作

注意：这些操作适用于 <code>ssub_match</code> 、 <code>csub_match</code> 、 <code>wssub_match</code> 、 <code>wcsub_match</code> 。
<code>matched</code> 一个 <code>public bool</code> 数据成员，指出此 <code>ssub_match</code> 是否匹配了
<code>first</code> <code>public</code> 数据成员，指向匹配序列首元素和尾后位置的迭代器。如果未匹配，则 <code>first</code> 和 <code>second</code> 是相等的
<code>second</code>
<code>length()</code> 匹配的大小。如果 <code>matched</code> 为 <code>false</code> ，则返回 0
<code>str()</code> 返回一个包含输入中匹配部分的 <code>string</code> 。如果 <code>matched</code> 为 <code>false</code> ，则返回空 <code>string</code>
<code>s = ssub</code> 将 <code>ssub_match</code> 对象 <code>ssub</code> 转化为 <code>string</code> 对象 <code>s</code> 。等价于 <code>s=ssub.str()</code> 。转换运算符不是 <code>explicit</code> 的（参见 14.9.1 节，第 515 页）

741 在一个合法的电话号码中，区号要么是完整括号包围的，要么完全没有括号。因此，`valid` 要做什么工作依赖于号码是否以一个括号开始：

```
bool valid(const smatch& m)
{
    // 如果区号前有一个左括号
    if(m[1].matched)
        // 则区号后必须有一个右括号，之后紧跟剩余号码或一个空格
        return m[3].matched
            && (m[4].matched == 0 || m[4].str() == " ");
    else
        // 否则，区号后不能有右括号
        // 另两个组成部分间的分隔符必须匹配
        return !m[3].matched
            && m[4].str() == m[6].str();
}
```

我们首先检查第一个子表达式（即，左括号）是否匹配了。这个子表达式在 `m[1]` 中。如果匹配了，则号码是以左括号开始的。在此情况下，如果区号后的子表达式也匹配了（意味着区号后有右括号）则整个号码是合法的。而且，如果号码正确使用了括号，则下一个字符必须是一个空格或下一部分的第一个数字。

如果 `m[1]` 未匹配，（即，没有左括号），则区号后的子表达式也不应该匹配。如果它为空，则整个号码是合法的。

17.3.3 节练习

练习 17.19：为什么可以不先检查 `m[4]` 是否匹配了就直接调用 `m[4].str()`？

练习 17.20：编写你自己版本的验证电话号码的程序。

练习 17.21：使用本节中定义的 `valid` 函数重写 8.3.2 节（第 289 页）中的电话号码程序。

练习 17.22：重写你的电话号码程序，使之允许在号码的三个部分之间放置任意多个空白符。

练习 17.23：编写查找邮政编码的正则表达式。一个美国邮政编码可以由五位或九位数字组成。前五位数字和后四位数字之间可以用一个短横线分隔。

17.3.4 使用 regex_replace

正则表达式不仅用在我们希望查找一个给定序列的时候，还用在当我们想将找到的序列替换为另一个序列的时候。例如，我们可能希望将美国的电话号码转换为“ddd.ddd.dddd”的形式，即，区号和后面三位数字用一个点分隔。

当我们希望在输入序列中查找并替换一个正则表达式时，可以调用 `<742> regex_replace`。表 17.12 描述了 `regex_replace`，类似搜索函数，它接受一个输入字符串序列和一个 `regex` 对象，不同的是，它还接受一个描述我们想要的输出形式的字符串。

表 17.12：正则表达式替换操作

<code>m.format(dest, fmt, mft)</code>	使用格式字符串 <code>fmt</code> 生成格式化输出，匹配在 <code>m</code> 中，可选的 <code>match_flag_type</code> 标志在 <code>mft</code> 中。第一个版本写入迭代器 <code>dest</code> 指向的目的位置（参见 10.5.1 节，第 365 页）并接受 <code>fmt</code> 参数，可以是一个 <code>string</code> ，也可以是表示字符数组中范围的一对指针。第二个版本返回一个 <code>string</code> ，保存输出，并接受 <code>fmt</code> 参数，可以是一个 <code>string</code> ，也可以是一个指向空字符结尾的字符数组的指针。 <code>mft</code> 的默认值为 <code>format_default</code>
<code>regex_replace(dest, seq, r, fmt, mft)</code>	遍历 <code>seq</code> ，用 <code>regex_search</code> 查找与 <code>regex</code> 对象 <code>r</code> 匹配的子串。使用格式字符串 <code>fmt</code> 和可选的 <code>match_flag_type</code> 标志来生成输出。第一个版本将输出写入到迭代器 <code>dest</code> 指定的位置，并接受一对迭代器 <code>seq</code> 表示范围。第二个版本返回一个 <code>string</code> ，保存输出，且 <code>seq</code> 既可以是一个 <code>string</code> 也可以是一个指向空字符结尾的字符数组的指针。在所有情况下， <code>fmt</code> 既可以是一个 <code>string</code> 也可以是一个指向空字符结尾的字符数组的指针，且 <code>mft</code> 的默认值为 <code>match_default</code>
<code>regex_replace(seq, r, fmt, mft)</code>	

替换字符串由我们想要的字符组合与匹配的子串对应的子表达式而组成。在本例中，我们希望在替换字符串中使用第二个、第五个和第七个子表达式。而忽略第一个、第三个、第四个和第六个子表达式，因为这些子表达式用来形成号码的原格式而非新格式中的一部分。我们用一个符号\$后跟子表达式的索引号来表示一个特定的子表达式：

```
string fmt = "$2.$5.$7"; // 将号码格式改为 ddd.ddd.dddd
```

可以像下面这样使用我们的正则表达式模式和替换字符串：

```
regex r(phone); // 用来寻找模式的 regex 对象
string number = "(908) 555-1800";
cout << regex_replace(number, r, fmt) << endl;
```

此程序的输出为：

```
908.555.1800
```

只替换输入序列的一部分

正则表达式更有意思的一个用处是替换一个大文件中的电话号码。例如，我们有一个保存人名及其电话号码的文件：

```
743 morgan (201) 555-2368 862-555-0123
      drew (973) 555.0130
      lee (609) 555-0132 2015550175 800.555-0000
```

我们希望将数据转换为下面这样：

```
morgan 201.555.2368 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0000
```

可以用下面的程序完成这种转换：

```
int main()
{
    string phone =
        "(\\" ()? (\\" \d{3}) (\\" )? ([-. ])? (\\" \d{3}) ([-. ])? (\\" \d{4}) ";
    regex r(phone); // 寻找模式所用的 regex 对象
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // 将号码格式改为 ddd.ddd.dddd
    // 从输入文件中读取每条记录
    while (getline(cin, s))
        cout << regex_replace(s, r, fmt) << endl;
    return 0;
}
```

我们读取每条记录，保存到 `s` 中，并将其传递给 `regex_replace`。此函数在输入序列中查找并转换所有匹配子串。

用来控制匹配和格式的标志

就像标准库定义标志来指导如何处理正则表达式一样，标准库还定义了用来在替换过程中控制匹配或格式的标志。表 17.13 列出了这些值。这些标志可以传递给函数 `regex_search` 或 `regex_match` 或是类 `smatch` 的 `format` 成员。

匹配和格式化标志的类型为 `match_flag_type`。这些值都定义在名为 `regex_constants` 的命名空间中。类似用于 `bind` 的 `placeholders`（参见 10.3.4 节，第 355 页），`regex_constants` 也是定义在命名空间 `std` 中的命名空间。为了使用 `regex_constants` 中的名字，我们必须在名字前同时加上两个命名空间的限定符：

```
using std::regex_constants::format_no_copy;
```

此声明指出，如果代码中使用了 `format_no_copy`，则表示我们想要使用命名空间 `std::constants` 中的这个名字。如下所示，我们也可以用另一种形式的 `using` 来代替上面的代码，我们将在 18.2.2 节（第 702 页）中介绍这种形式：

```
using namespace std::regex_constants;
```

表 17.13: 匹配标志

< 744

定义在 <code>regex_constants::match_flag_type</code> 中	
<code>match_default</code>	等价于 <code>format_default</code>
<code>match_not_bol</code>	不将首字符作为行首处理
<code>match_not_eol</code>	不将尾字符作为行尾处理
<code>match_not_bow</code>	不将首字符作为单词首处理
<code>match_not_eow</code>	不将尾字符作为单词尾处理
<code>match_any</code>	如果存在多于一个匹配，则可返回任意一个匹配
<code>match_not_null</code>	不匹配任何空序列
<code>match_continuous</code>	匹配必须从输入的首字符开始
<code>match_prev_avail</code>	输入序列包含第一个匹配之前的内容
<code>format_default</code>	用 ECMA Script 规则替换字符串
<code>format_sed</code>	用 POSIX sed 规则替换字符串
<code>format_no_copy</code>	不输出输入序列中未匹配的部分
<code>format_first_only</code>	只替换子表达式第一次出现

使用格式标志

默认情况下，`regex_replace` 输出整个输入序列。未与正则表达式匹配的部分会原样输出；匹配的部分按格式字符串指定的格式输出。我们可以通过在 `regex_replace` 调用中指定 `format_no_copy` 来改变这种默认行为：

```
// 只生成电话号码：使用新的格式字符串
string fmt2 = "$2.$5.$7 "; // 在最后一部分号码后放置空格作为分隔符
// 通知 regex_replace 只拷贝它替换的文本
cout << regex_replace(s, r, fmt2, format_no_copy) << endl;
```

给定相同的输入，此版本的程序生成

```
201.555.2368 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0000
```

17.3.4 节练习

练习 17.24: 编写你自己版本的重排电话号码格式的程序。

练习 17.25: 重写你的电话号码程序，使之只输出每个人的第一个电话号码。

练习 17.26: 重写你的电话号码程序，使之对多于一个电话号码的人只输出第二个和后续电话号码。

练习 17.27: 编写程序，将九位数字邮政编码的格式转换为 dddd-dddd。

17.4 随机数

< 745

程序通常需要一个随机数源。在新标准出现之前，C 和 C++ 都依赖于一个简单的 C 库函数 `rand` 来生成随机数。此函数生成均匀分布的伪随机整数，每个随机数的范围在 0 和一个系统相关的最大值（至少为 32767）之间。

C++
11

`rand` 函数有一些问题：即使不是大多数，也有很多程序需要不同范围的随机数。一些应用需要随机浮点数。一些程序需要非均匀分布的数。而程序员为了解决这些问题而试图转换 `rand` 生成的随机数的范围、类型或分布时，常常会引入非随机性。

定义在头文件 `random` 中的随机数库通过一组协作的类来解决这些问题：随机数引擎类（random-number engines）和随机数分布类（random-number distribution）。表 17.14 描述了这些类。一个引擎类可以生成 `unsigned` 随机数序列，一个分布类使用一个引擎类生成指定类型的、在给定范围内的、服从特定概率分布的随机数。

表 17.14：随机数库的组成

引擎	类型，生成随机 <code>unsigned</code> 整数序列
分布	类型，使用引擎返回服从特定概率分布的随机数



C++ 程序不应该使用库函数 `rand`，而应使用 `default_random_engine` 类和恰当的分布类对象。

17.4.1 随机数引擎和分布

随机数引擎是函数对象类（参见 14.8 节，第 506 页），它们定义了一个调用运算符，该运算符不接受参数并返回一个随机 `unsigned` 整数。我们可以通过调用一个随机数引擎对象来生成原始随机数：

```
default_random_engine e; // 生成随机无符号数
for (size_t i = 0; i < 10; ++i)
    // e() “调用” 对象来生成下一个随机数
    cout << e() << " ";
```

在我们的系统中，此程序生成：

```
16807 282475249 1622650073 984943658 1144108930 470211272 ...
```

在本例中，我们定义了一个名为 `e` 的 `default_random_engine` 对象。在 `for` 循环内，我们调用对象 `e` 来获得下一个随机数。

746

标准库定义了多个随机数引擎类，区别在于性能和随机性质量不同。每个编译器都会指定其中一个作为 `default_random_engine` 类型。此类型一般具有最常用的特性。表 17.15 列出了随机数引擎操作，标准库定义的引擎类型列在附录 A.3.2（第 783 页）中。

表 17.15：随机数引擎操作

<code>Engine e;</code>	默认构造函数；使用该引擎类型默认的种子
<code>Engine e(s);</code>	使用整型值 <code>s</code> 作为种子
<code>e.seed(s)</code>	使用种子 <code>s</code> 重置引擎的状态
<code>e.min()</code>	此引擎可生成的最小值和最大值
<code>e.max()</code>	
<code>Engine::result_type</code>	此引擎生成的 <code>unsigned</code> 整型类型
<code>e.discard(u)</code>	将引擎推进 <code>u</code> 步； <code>u</code> 的类型为 <code>unsigned long long</code>

对于大多数场合，随机数引擎的输出是不能直接使用的，这也是为什么早先我们称之为原始随机数。问题出在生成的随机数的值范围通常与我们需要的不符，而正确转换随机数的范围是极其困难的。

分布类型和引擎

为了得到在一个指定范围内的数，我们使用一个分布类型的对象：

```
// 生成 0 到 9 之间（包含）均匀分布的随机数
uniform_int_distribution<unsigned> u(0,9);
default_random_engine e; // 生成无符号随机整数
for (size_t i = 0; i < 10; ++i)
    // 将 u 作为随机数源
    // 每个调用返回在指定范围内并服从均匀分布的值
    cout << u(e) << " ";
```

此代码生成下面这样的输出

```
0 1 7 4 5 2 0 6 6 9
```

此处我们将 `u` 定义为 `uniform_int_distribution<unsigned>`。此类型生成均匀分布的 `unsigned` 值。当我们定义一个这种类型的对象时，可以提供想要的最小值和最大值。在此程序中，`u(0,9)` 表示我们希望得到 0 到 9 之间（包含）的数。随机数分布类会使用包含的范围，从而我们可以得到给定整型类型的每个可能值。

类似引擎类型，分布类型也是函数对象类。分布类型定义了一个调用运算符，它接受一个随机数引擎作为参数。分布对象使用它的引擎参数生成随机数，并将其映射到指定的分布。

注意，我们传递给分布对象的是引擎对象本身，即 `u(e)`。如果我们将调用写成 `u(e())`，含义就变为将 `e` 生成的下一个值传递给 `u`，会导致一个编译错误。我们传递的是引擎本身，而不是它生成的下一个值，原因是某些分布可能需要调用引擎多次才能得到一个值。



当我们说随机数发生器时，是指分布对象和引擎对象的组合。

比较随机数引擎和 `rand` 函数

对熟悉 C 库函数 `rand` 的读者，值得注意的是：调用一个 `default_random_engine` 对象的输出类似 `rand` 的输出。随机数引擎生成的 `unsigned` 整数在一个系统定义的范围内，而 `rand` 生成的数的范围在 0 到 `RAND_MAX` 之间。一个引擎类型的范围可以通过调用该类型对象的 `min` 和 `max` 成员来获得：

```
cout << "min: " << e.min() << " max: " << e.max() << endl;
```

在我们的系统中，此程序生成下面的输出：

```
min: 1 max: 2147483648
```

引擎生成一个数值序列

随机数发生器有一个特性经常会使新手迷惑：即使生成的数看起来是随机的，但对一个给定的发生器，每次运行程序它都会返回相同的数值序列。序列不变这一事实在调试时非常有用。但另一方面，使用随机数发生器的程序也必须考虑这一特性。

作为一个例子，假定我们需要一个函数生成一个 `vector`，包含 100 个均匀分布在 0 到 9 之间的随机数。我们可能认为应该这样编写此函数：

```
// 几乎肯定是生成随机整数 vector 的错误方法
// 每次调用这个函数都会生成相同的 100 个数!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0, 9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

但是，每次调用这个函数都会返回相同的 vector:

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// 将打印"equal"
cout << ((v1 == v2) ? "equal" : "not equal") << endl;
```

748 此代码会打印 equal，因为 vector v1 和 v2 具有相同的值。

编写此函数的正确方法是将引擎和关联的分布对象定义为 static 的（参见 6.1.1 节，第 185 页）：

```
// 返回一个 vector，包含 100 个均匀分布的随机数
vector<unsigned> good_randVec()
{
    // 由于我们希望引擎和分布对象保持状态，因此应该将它们
    // 定义为 static 的，从而每次调用都生成新的数
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0, 9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

由于 e 和 u 是 static 的，因此它们在函数调用之间会保持住状态。第一次调用会使用 u(e) 生成的序列中的前 100 个随机数，第二次调用会获得接下来 100 个，依此类推。



一个给定的随机数发生器一直会生成相同的随机数序列。一个函数如果定义了局部的随机数发生器，应该将其（包括引擎和分布对象）定义为 static 的。否则，每次调用函数都会生成相同的序列。

设置随机数发生器种子

随机数发生器会生成相同的随机数序列这一特性在调试中很有用。但是，一旦我们的程序调试完毕，我们通常希望每次运行程序都会生成不同的随机结果，可以通过提供一个种子（seed）来达到这一目的。种子就是一个数值，引擎可以利用它从序列中一个新位置重新开始生成随机数。

为引擎设置种子有两种方式：在创建引擎对象时提供种子，或者调用引擎的 seed 成员：

```

default_random_engine e1;           // 使用默认种子
default_random_engine e2(2147483646); // 使用给定的种子值
// e3 和 e4 将生成相同的序列，因为它们使用了相同的种子
default_random_engine e3;           // 使用默认种子值
e3.seed(32767);                  // 调用 seed 设置一个新种子值
default_random_engine e4(32767);    // 将种子值设置为 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i << endl;
    if (e3() != e4())
        cout << "seeded differs at iteration: " << i << endl;
}

```

本例中我们定义了四个引擎。前两个引擎 `e1` 和 `e2` 的种子不同，因此应该生成不同的序列。后两个引擎 `e3` 和 `e4` 有相同的种子，它们将生成相同的序列。

< 749

选择一个好的种子，与生成好的随机数所涉及的其他大多数事情相同，是极其困难的。可能最常用的方法是调用系统函数 `time`。这个函数定义在头文件 `ctime` 中，它返回从一个特定时刻到当前经过了多少秒。函数 `time` 接受单个指针参数，它指向用于写入时间的数据结构。如果此指针为空，则函数简单地返回时间：

```
default_random_engine e1(time(0)); // 稍微随机些的种子
```

由于 `time` 返回以秒计的时间，因此这种方式只适用于生成种子的间隔为秒级或更长的应用。



WARNING

如果程序作为一个自动过程的一部分反复运行，将 `time` 的返回值作为种子的方式就无效了；它可能多次使用的都是相同的种子。

17.4.1 节练习

练习 17.28: 编写函数，每次调用生成并返回一个均匀分布的随机 `unsigned int`。

练习 17.29: 修改上一题中编写的函数，允许用户提供一个种子作为可选参数。

练习 17.30: 再次修改你的程序，此次再增加两个参数，表示函数允许返回的最小值和最大值。

17.4.2 其他随机数分布

随机数引擎生成 `unsigned` 数，范围内的每个数被生成的概率都是相同的。而应用程序常常需要不同类型或不同分布的随机数。标准库通过定义不同随机数分布对象来满足这两方面的要求，分布对象和引擎对象协同工作，生成要求的结果。表 17.16 列出了分布类型所支持的操作。

生成随机实数

程序常需要一个随机浮点数的源。特别是，程序经常需要 0 到 1 之间的随机数。

最常用但不正确的从 `rand` 获得一个随机浮点数的方法是用 `rand()` 的结果除以 `RAND_MAX`，即，系统定义的 `rand` 可以生成的最大随机数的上界。这种方法不正确的原因是随机整数的精度通常低于随机浮点数，这样，有一些浮点值就永远不会被生成了。

使用新标准库设施，可以很容易地获得随机浮点数。我们可以定义一个

< 750

`uniform_real_distribution` 类型的对象，并让标准库来处理从随机整数到随机浮点数的映射。与处理 `uniform_int_distribution` 一样，在定义对象时，我们指定最小值和最大值：

```
default_random_engine e; // 生成无符号随机整数
// 0 到 1(包含) 的均匀分布
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
    cout << u(e) << " ";
```

这段代码与之前生成 `unsigned` 值的程序几乎相同。但是，由于我们使用了一个不同的分布类型，此版本会生成不同的结果：

```
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
```

表 17.16：分布类型的操作

<code>Dist d;</code>	默认构造函数；使 <code>d</code> 准备好被使用。 其他构造函数依赖于 <code>Dist</code> 的类型；参见附录 A.3 节（第 781 页）。 分布类型的构造函数是 <code>explicit</code> 的（参见 7.5.4 节，第 265 页）
<code>d(e)</code>	用相同的 <code>e</code> 连续调用 <code>d</code> 的话，会根据 <code>d</code> 的分布式类型生成一个随机数序列； <code>e</code> 是一个随机数引擎对象
<code>d.min()</code>	返回 <code>d(e)</code> 能生成的最小值和最大值
<code>d.max()</code>	
<code>d.reset()</code>	重建 <code>d</code> 的状态，使得随后对 <code>d</code> 的使用不依赖于 <code>d</code> 已经生成的值

使用分布的默认结果类型

分布类型都是模板，具有单一的模板类型参数，表示分布生成的随机数的类型，对此有一个例外，我们将在 17.4.2 节（第 665 页）中进行介绍。这些分布类型要么生成浮点类型，要么生成整数类型。

每个分布模板都有一个默认模板实参（参见 16.1.3 节，第 594 页）。生成浮点值的分布类型默认生成 `double` 值，而生成整型值的分布默认生成 `int` 值。由于分布类型只有一个模板参数，因此当我们希望使用默认随机数类型时要记得在模板名之后使用空尖括号（参见 16.1.3 节，第 594 页）：

```
// 空<>表示我们希望使用默认结果类型
uniform_real_distribution<> u(0,1); // 默认生成 double 值
```

751 生成非均匀分布的随机数

除了正确生成在指定范围内的数之外，新标准库的另一个优势是可以生成非均匀分布的随机数。实际上，新标准库定义了 20 种分布类型，这些类型列在附录 A.3（第 781）中。

作为一个例子，我们将生成一个正态分布的值的序列，并画出值的分布。由于 `normal_distribution` 生成浮点值，我们的程序使用头文件 `cmath` 中的 `lround` 函数将每个随机数舍入到最接近的整数。我们将生成 200 个数，它们以均值 4 为中心，标准差为 1.5。由于使用的是正态分布，我们期望生成的数中大约 99% 都在 0 到 8 之间（包含）。我们的程序会对这个范围内的每个整数统计有多少个生成的数映射到它：

```
default_random_engine e; // 生成随机整数
normal_distribution<> n(4,1.5); // 均值 4，标准差 1.5
```

```

vector<unsigned> vals(9);           // 9 个元素均为 0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e));      // 舍入到最接近的整数
    if (v < vals.size())          // 如果结果在范围内
        ++vals[v];                // 统计每个数出现了多少次
}
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ":" << string(vals[j], '*') << endl;

```

我们首先定义了随机数发生器对象和一个名为 `vals` 的 `vector`。我们用 `vals` 来统计范围 0...8 中的每个数出现了多少次。与我们使用 `vector` 的大多数组程序不同，此程序按需求大小为 `vals` 分配空间，每个元素都被初始化为 0。

在 `for` 循环中，我们调用 `lround(n(e))` 来将 `n(e)` 返回的值舍入到最接近的整数。获得浮点随机数对应的整数后，我们将它作为计数器 `vector` 的下标。由于 `n(e)` 可能生成范围 0 到 8 之外的数，所以我们首先检查生成的数是否在范围内，然后再将其作为 `vals` 的下标。如果结果确实在范围内，我们递增对应的计数器。

当循环结束时，我们打印 `vals` 的内容，可能会打印出像下面这样的结果：

```

0: ***
1: *****
2: *****
3: *****
4: *****
5: *****
6: *****
7: *****
8: *

```

本例中我们打印一个由星号组成的 `string`，有多少随机数等于此下标我们就打印多少个星号。注意，此图并不是完美对称的。如果打印出的图是完美对称的，我们反倒有理由怀疑随机数发生器的质量了。 <752>

bernoulli_distribution 类

我们注意到有一个分布不接受模板参数，即 `bernoulli_distribution`，因为它是一个普通类，而非模板。此分布总是返回一个 `bool` 值。它返回 `true` 的概率是一个常数，此概率的默认值是 0.5。

作为一个这种分布的例子，我们可以编写一个程序，这个程序与用户玩一个游戏。为了进行这个游戏，其中一个游戏者——用户或是程序——必须先行。我们可以用一个值范围是 0 到 1 的 `uniform_int_distribution` 来选择先行的游戏者，但也可以用伯努利分布来完成这个选择。假定已有一个名为 `play` 的函数来进行游戏，我们可以编写像下面这样的循环来与用户交互：

```

string resp;
default_random_engine e; // e 应保持状态，所以必须在循环外定义！
bernoulli_distribution b; // 默认是 50/50 的机会
do {
    bool first = b(e); // 如果为 true，则程序先行
    cout << (first ? "We go first"
              : "You get to go first") << endl;
}

```

```
// 传递谁先行的指示，进行游戏
cout << ((play(first)) ? "sorry, you lost"
           : "congrats, you won") << endl;
cout << "play again? Enter 'yes' or 'no'" << endl;
} while (cin >> resp && resp[0] == 'y');
```

我们用一个 do while 循环（参见 5.4.4 节，第 169 页）来反复提示用户进行游戏。



由于引擎返回相同的随机数序列（参见 17.4.1 节，第 661 页），所以我们必须在循环外声明引擎对象。否则，每步循环都会创建一个新引擎，从而每步循环都会生成相同的值。类似的，分布对象也要保持状态，因此也应该在循环外定义。

在此程序中使用 bernoulli_distribution 的一个原因是它允许我们调整选择先行一方的概率：

```
bernoulli_distribution b(.55); // 给程序一个微小的优势
```

如果 b 定义如上，则程序有 55/45 的机会先行。

17.4.2 节练习

练习 17.31: 对于本节中的游戏程序，如果在 do 循环内定义 b 和 e，会发生什么？

练习 17.32: 如果我们在循环内定义 resp，会发生什么？

练习 17.33: 修改 11.3.6 节（第 392 页）中的单词转换程序，允许对一个给定单词有多种转换方式，每次随机选择一种进行实际转换。

17.5 IO 库再探

在第 8 章中我们介绍了 IO 库的基本结构及其最常用的部分。在本节中，我们将介绍三个更特殊的 IO 库特性：格式控制、未格式化 IO 和随机访问。

753 17.5.1 格式化输入与输出

除了条件状态外（参见 8.1.2 节，第 279 页），每个 iostream 对象还维护一个格式状态来控制 IO 如何格式化的细节。格式状态控制格式化的某些方面，如整型值是几进制、浮点值的精度、一个输出元素的宽度等。

标准库定义了一组操纵符（manipulator）（参见 1.2 节，第 6 页）来修改流的格式状态，如表 17.7 和表 17.8 所示。一个操纵符是一个函数或是一个对象，会影响流的状态，并能用作输入或输出运算符的运算对象。类似输入和输出运算符，操纵符也返回它所处理的流对象，因此我们可以在一条语句中组合操纵符和数据。

我们已经在程序中使用过一个操纵符——endl，我们将它“写”到输出流，就像它是一个值一样。但 endl 不是一个普通值，而是一个操作：它输出一个换行符并刷新缓冲区。

很多操纵符改变格式状态

操纵符用于两大类输出控制：控制数值的输出形式以及控制补白的数量和位置。大多数改变格式状态的操纵符都是设置/复原成对的；一个操纵符用来将格式状态设置为一个新值，而另一个用来将其复原，恢复为正常的默认格式。



当操纵符改变流的格式状态时，通常改变后的状态对所有后续 IO 都生效。

当我们有一组 IO 操作希望使用相同的格式时，操纵符对格式状态的改变是持久的这一特性很有用。实际上，一些程序会利用操纵符的这一特性对其所有输入或输出重置一个或多个格式规则的行为。在这种情况下，操纵符会改变流这一特性就是满足要求的了。

但是，很多程序（而且更重要的是，很多程序员）期望流的状态符合标准库正常的默认设置。在这些情况下，将流的状态置于一个非标准状态可能会导致错误。因此，通常最好在不再需要特殊格式时尽快将流恢复到默认状态。

控制布尔值的格式

754

操纵符改变对象的格式状态的一个例子是 `boolalpha` 操纵符。默认情况下，`bool` 值打印为 1 或 0。一个 `true` 值输出为整数 1，而 `false` 输出为 0。我们可以通过对流使用 `boolalpha` 操纵符来覆盖这种格式：

```
cout << "default bool values: " << true << " " << false
<< "\nalpha bool values: " << boolalpha
<< true << " " << false << endl;
```

执行这段程序会得到下面的结果：

```
default bool values: 1 0
alpha bool values: true false
```

一旦向 `cout` “写入”了 `boolalpha`，我们就改变了 `cout` 打印 `bool` 值的方式。后续打印 `bool` 值的操作都会打印 `true` 或 `false` 而非 1 或 0。

为了取消 `cout` 格式状态的改变，我们使用 `noboolalpha`：

```
bool bool_val = get_status();
cout << boolalpha      // 设置 cout 的内部状态
<< bool_val
<< noboolalpha;        // 将内部状态恢复为默认格式
```

本例中我们改变了 `bool` 值的格式，但只对 `bool_val` 的输出有效。一旦完成此值的打印，我们立即将流恢复到初始状态。

指定整型值的进制

默认情况下，整型值的输入输出使用十进制。我们可以使用操纵符 `hex`、`oct` 和 `dec` 将其改为十六进制、八进制或是改回十进制：

```
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;
```

当编译并执行这段程序时，会得到如下输出：

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

注意，类似 `boolalpha`，这些操纵符也会改变格式状态。它们会影响下一个和随后所有的整型输出，直至另一个操纵符又改变了格式为止。



操纵符 `hex`、`oct` 和 `dec` 只影响整型运算对象，浮点值的表示形式不受影响。

755 在输出中指出进制

默认情况下，当我们打印出数值时，没有可见的线索指出使用的是几进制。例如，20 是十进制的 20 还是 16 的八进制表示？当我们按十进制打印数值时，打印结果会符合我们的期望。如果需要打印八进制值或十六进制值，应该使用 `showbase` 操纵符。当对流应用 `showbase` 操纵符时，会在输出结果中显示进制，它遵循与整型常量中指定进制相同的规范：

- 前导 `0x` 表示十六进制。
- 前导 `0` 表示八进制。
- 无前导字符串表示十进制。

我们可以使用 `showbase` 修改前一个程序：

```
cout << showbase; // 当打印整型值时显示进制
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase; // 恢复流状态
```

修改后的程序的输出会更清楚地表明底层值到底是什么：

```
default: 20 1024
in octal: 024 02000
in hex: 0x14 0x400
in decimal: 20 1024
```

操纵符 `noshowbase` 恢复 `cout` 的状态，从而不再显示整型值的进制。

默认情况下，十六进制值会以小写打印，前导字符也是小写的 `x`。我们可以通过使用 `uppercase` 操纵符来输出大写的 `X` 并将十六进制数字 `a-f` 以大写输出：

```
cout << uppercase << showbase << hex
<< "printed in hexadecimal: " << 20 << " " << 1024
<< nouppercase << noshowbase << dec << endl;
```

这条语句生成如下输出：

```
printed in hexadecimal: 0X14 0X400
```

我们使用了操纵符 `nouppercase`、`noshowbase` 和 `dec` 来重置流的状态。

控制浮点数格式

我们可以控制浮点数输出三个种格式：

- 以多高精度（多少个数字）打印浮点值
- 数值是打印为十六进制、定点十进制还是科学记数法形式
- 对于没有小数部分的浮点值是否打印小数点

<756

默认情况下，浮点值按六位数字精度打印；如果浮点值没有小数部分，则不打印小数点；根据浮点数的值选择打印成定点十进制或科学记数法形式。标准库会选择一种可读性更好的格式：非常大和非常小的值打印为科学记数法形式，其他值打印为定点十进制形式。

指定打印精度

默认情况下，精度会控制打印的数字的总数。当打印时，浮点值按当前精度舍入而非截断。因此，如果当前精度为四位数字，则 3.14159 将打印为 3.142；如果精度为三位数字，则打印为 3.14。

我们可以通过调用 IO 对象的 precision 成员或使用 setprecision 操纵符来改变精度。precision 成员是重载的（参见 6.4 节，第 206 页）。一个版本接受一个 int 值，将精度设置为此值，并返回旧精度值。另一个版本不接受参数，返回当前精度值。setprecision 操纵符接受一个参数，用来设置精度。



操纵符 setprecision 和其他接受参数的操纵符都定义在头文件 iomanip 中。

下面的程序展示了控制浮点值打印精度的不同方法：

```
// cout.precision 返回当前精度值
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
// cout.precision(12) 将打印精度设置为 12 位数字
cout.precision(12);
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
// 另一种设置精度的方法是使用 setprecision 操纵符
cout << setprecision(3);
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
```

编译并执行这段程序，会得到如下输出：

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

此程序调用标准库 sqrt 函数，它定义在头文件 cmath 中。sqrt 函数是重载的，不同版本分别接受一个 float、double 或 long double 参数，返回实参的平方根。

<757

表 17.17：定义在 iostream 中的操纵符

boolalpha	将 true 和 false 输出为字符串
* noboolalpha	将 true 和 false 输出为 1, 0
showbase	对整型值输出表示进制的前缀
* noshowbase	不生成表示进制的前缀
showpoint	对浮点值总是显示小数点

续表

* noshowpoint	只有当浮点值包含小数部分时才显示小数点
showpos	对非负数显示+
* noshowpos	对非负数不显示+
uppercase	在十六进制值中打印 0x，在科学记数法中打印 E
* nouppercase	在十六进制值中打印 0x，在科学记数法中打印 e
* dec	整型值显示为十进制
hex	整型值显示为十六进制
oct	整型值显示为八进制
left	在值的右侧添加填充字符
right	在值的左侧添加填充字符
internal	在符号和值之间添加填充字符
fixed	浮点值显示为定点十进制
scientific	浮点值显示为科学记数法
hexfloat	浮点值显示为十六进制（C++11 新特性）
defaultfloat	重置浮点数格式为十进制（C++11 新特性）
unitbuf	每次输出操作后都刷新缓冲区
* nounitbuf	恢复正常缓冲区刷新方式
* skipws	输入运算符跳过空白符
noskipws	输入运算符不跳过空白符
flush	刷新 ostream 缓冲区
ends	插入空字符，然后刷新 ostream 缓冲区
endl	插入换行，然后刷新 ostream 缓冲区

* 表示默认流状态

指定浮点数记数法



除非你需要控制浮点数的表示形式（如，按列打印数据或打印表示金额或百分比的数据），否则由标准库选择记数法是最好的方式。

通过使用恰当的操纵符，我们可以强制一个流使用科学记数法、定点十进制或是十六进制记数法。操纵符 `scientific` 改变流的状态来使用科学记数法。操纵符 `fixed` 改变流的状态来使用定点十进制。

在新标准库中，通过使用 `hexfloat` 也可以强制浮点数使用十六进制格式。新标准库还提供另一个名为 `defaultfloat` 的操纵符，它将流恢复到默认状态——根据要打印的值选择记数法。

这些操纵符也会改变流的精度的默认含义。在执行 `scientific`、`fixed` 或 `hexfloat` 后，精度值控制的是小数点后面的数字位数，而默认情况下精度值指定的是数字的总位数——既包括小数点之后的数字也包括小数点之前的数字。使用 `fixed` 或 `scientific` 令我们可以按列打印数值，因为小数点距小数部分的距离是固定的：

```
cout << "default format: " << 100 * sqrt(2.0) << '\n'
<< "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
<< "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
<< "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
<< "use defaults: " << defaultfloat << 100 * sqrt(2.0)
```

```
<< "\n\n";
```

此程序会生成下面的输出：

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcp+7
use defaults: 141.421
```

默认情况下，十六进制数字和科学记数法中的 e 都打印成小写形式。我们可以用 uppercase 操纵符打印这些字母的大写形式。

打印小数点

默认情况下，当一个浮点值的小数部分为 0 时，不显示小数点。showpoint 操纵符强制打印小数点：

```
cout << 10.0 << endl;           // 打印 10
cout << showpoint << 10.0      // 打印 10.0000
<< noshowpoint << endl;       // 恢复小数点的默认格式
```

操纵符 noshowpoint 恢复默认行为。下一个输出表达式将有默认行为，即，当浮点值的小数部分为 0 时不输出小数点。

输出补白

当按列打印数据时，我们常常需要非常精细地控制数据格式。标准库提供了一些操纵符帮助我们完成所需的控制：

- setw 指定下一个数字或字符串值的最小空间。
- left 表示左对齐输出。
- right 表示右对齐输出，右对齐是默认格式。
- internal 控制负数的符号的位置，它左对齐符号，右对齐值，用空格填满所有中间空间。 ◀759
- setfill 允许指定一个字符代替默认的空格来补白输出。



setw 类似 endl，不改变输出流的内部状态。它只决定下一个输出的大小。

下面程序展示了如何使用这些操纵符：

```
int i = -16;
double d = 3.14159;
// 补白第一列，使用输出中最小 12 个位置
cout << "i: " << setw(12) << i << "next col" << '\n'
     << "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，左对齐所有列
cout << left
     << "i: " << setw(12) << i << "next col" << '\n'
     << "d: " << setw(12) << d << "next col" << '\n'
     << right; // 恢复正常对齐
// 补白第一列，右对齐所有列
cout << right
     << "i: " << setw(12) << i << "next col" << '\n'
```

```

<< "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，但补在域的内部
cout << internal
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，用#作为补白字符
cout << setfill('#')
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << setfill(' ');
// 恢复正常的补白字符

```

执行这段程序，会得到下面的输出：

```

i:          -16next col
d:      3.14159next col
i: -16      next col
d: 3.14159      next col
i:          -16next col
d:      3.14159next col
i: -      16next col
d:      3.14159next col
i: -#####16next col
d: #####3.14159next col

```

760

表 17.18：定义在 iomanip 中的操纵符

setfill(ch)	用 ch 填充空白
setprecision(n)	将浮点精度设置为 n
setw(w)	读或写值的宽度为 w 个字符
setbase(b)	将整数输出为 b 进制

控制输入格式

默认情况下，输入运算符会忽略空白符（空格符、制表符、换行符、换纸符和回车符）。下面的循环

```

char ch;
while (cin >> ch)
    cout << ch;

```

当给定下面输入序列时

```

a b      c
d

```

循环会执行 4 次，读取字符 a 到 d，跳过中间的空格以及可能的制表符和换行符。此程序的输出是

```

abcd

```

操纵符 noskipws 会令输入运算符读取空白符，而不是跳过它们。为了恢复默认行为，我们可以使用 skipws 操纵符：

```

cin >> noskipws; // 设置 cin 读取空白符
while (cin >> ch)
    cout << ch;

```

```
cin >> skipws; // 将 cin 恢复到默认状态，从而丢弃空白符
```

给定与前一个程序相同的输入，此循环会执行 7 次，从输入中既读取普通字符又读取空白符。此循环的输出为

```
a b      c  
d
```

17.5.1 节练习

练习 17.34: 编写一个程序，展示如何使用表 17.17 和表 17.18 中的每个操纵符。

练习 17.35: 修改第 670 页中的程序，打印 2 的平方根，但这次打印十六进制数字的大写形式。

练习 17.36: 修改上一题中的程序，打印不同的浮点数，使它们排成一列。

17.5.2 未格式化的输入/输出操作

761

到目前为止，我们的程序只使用过格式化 IO (formatted IO) 操作。输入和输出运算符 (<<和>>) 根据读取或写入的数据类型来格式化它们。输入运算符忽略空白符，输出运算符应用补白、精度等规则。

标准库还提供了一组低层操作，支持未格式化 IO (unformatted IO)。这些操作允许我们将一个流当作一个无解释的字节序列来处理。

单字节操作

有几个未格式化操作每次一个字节地处理流。这些操作列在表 17.19 中，它们会读取而不是忽略空白符。例如，我们可以使用未格式化 IO 操作 get 和 put 来读取和写入一个字符：

```
char ch;  
while (cin.get(ch))  
    cout.put(ch);
```

此程序保留输入中的空白符，其输出与输入完全相同。它的执行过程与前一个使用 noskipws 的程序完全相同。

表 17.19: 单字节低层 IO 操作

is.get(ch)	从 istream is 读取下一个字节存入字符 ch 中。返回 is
os.put(ch)	将字符 ch 输出到 ostream os。返回 os
is.get()	将 is 的下一个字节作为 int 返回
is.putback(ch)	将字符 ch 放回 is。返回 is
is.unget()	将 is 向后移动一个字节。返回 is
is.peek()	将下一个字节作为 int 返回，但不从流中删除它

将字符放回输入流

有时我们需要读取一个字符才能知道还未准备好处理它。在这种情况下，我们希望将字符放回流中。标准库提供了三种方法退回字符，它们有着细微的差别：

- peek 返回输入流中下一个字符的副本，但不会将它从流中删除，peek 返回的值仍然留在流中。

- `unget` 使得输入流向后移动，从而最后读取的值又回到流中。即使我们不知道最后从流中读取什么值，仍然可以调用 `unget`。
- `putback` 是更特殊版本的 `unget`：它退回从流中读取的最后一个值，但它接受一个参数，此参数必须与最后读取的值相同。

762 一般情况下，在读取下一个值之前，标准库保证我们可以退回最多一个值。即，标准库不保证在中间不进行读取操作的情况下能连续调用 `putback` 或 `unget`。

从输入操作返回的 int 值

函数 `peek` 和无参的 `get` 版本都以 `int` 类型从输入流返回一个字符。这有些令人吃惊，可能这些函数返回一个 `char` 看起来会更自然。

这些函数返回一个 `int` 的原因是：可以返回文件尾标记。我们使用 `char` 范围中的每个值来表示一个真实字符，因此，取值范围中没有额外的值可以用来表示文件尾。

返回 `int` 的函数将它们要返回的字符先转换为 `unsigned char`，然后再将结果提升到 `int`。因此，即使字符集中有字符映射到负值，这些操作返回的 `int` 也是正值（参见 2.1.2 节，第 32 页）。而标准库使用负值表示文件尾，这样就可以保证与任何合法字符的值都不同。头文件 `cstdio` 定义了一个名为 `EOF` 的 `const`，我们可以用它来检测从 `get` 返回的值是否是文件尾，而不必记忆表示文件尾的实际数值。对我们来说重要的是，用一个 `int` 来保存从这些函数返回的值：

```
int ch; // 使用一个 int，而不是一个 char 来保存 get() 的返回值
// 循环读取并输出输入中的所有数据
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

此程序与第 673 页中的程序完成相同的工作，唯一的不同是用来读取输入的 `get` 版本不同。

多字节操作

一些未格式化 IO 操作一次处理大块数据。如果速度是要考虑的重点问题的话，这些操作是很重要的，但类似其他低层操作，这些操作也容易出错。特别是，这些操作要求我们自己分配并管理用来保存和提取数据的字符数组（参见 12.2 节，第 423 页）。表 17.20 列出了多字节操作。

表 17.20：多字节低层 IO 操作

<code>is.get(sink, size, delim)</code>	从 <code>is</code> 中读取最多 <code>size</code> 个字节，并保存在字符数组中，字符数组的起始地址由 <code>sink</code> 给出。读取过程直至遇到字符 <code>delim</code> 或读取了 <code>size</code> 个字节或遇到文件尾时停止。如果遇到了 <code>delim</code> ，则将其留在输入流中，不读取出来存入 <code>sink</code>
<code>is.getline(sink, size, delim)</code>	与接受三个参数的 <code>get</code> 版本类似，但会读取并丢弃 <code>delim</code>
<code>is.read(sink, size)</code>	读取最多 <code>size</code> 个字节，存入字符数组 <code>sink</code> 中。返回 <code>is</code>
<code>is.gcount()</code>	返回上一个未格式化读取操作从 <code>is</code> 读取的字节数
<code>os.write(source, size)</code>	将字符数组 <code>source</code> 中的 <code>size</code> 个字节写入 <code>os</code> 。返回 <code>os</code>

续表

```
is.ignore(size, delim)
```

读取并忽略最多 size 个字符，包括 delim。与其他未格式化函数不同，ignore 有默认参数：size 的默认值为 1，delim 的默认值为文件尾

get 和 getline 函数接受相同的参数，它们的行为类似但不相同。在两个函数中，sink 都是一个 char 数组，用来保存数据。两个函数都一直读取数据，直至下面条件之一发生：

- 已读取了 size-1 个字符
- 遇到了文件尾
- 遇到了分隔符

两个函数的差别是处理分隔符的方式：get 将分隔符留作 istream 中的下一个字符，而 getline 则读取并丢弃分隔符。无论哪个函数都不会将分隔符保存在 sink 中。



WARNING

一个常见的错误是本想从流中删除分隔符，但却忘了做。

< 763

确定读取了多少个字符

某些操作从输入读取未知个数的字节。我们可以调用 gcount 来确定最后一个未格式化输入操作读取了多少个字符。应该在任何后续未格式化输入操作之前调用 gcount。特别是，将字符退回流的单字符操作也属于未格式化输入操作。如果在调用 gcount 之前调用了 peek、unget 或 putback，则 gcount 的返回值为 0。

小心：低层函数容易出错

一般情况下，我们主张使用标准库提供的高层抽象。返回 int 的 IO 操作很好地解释了原因。

一个常见的编程错误是将 get 或 peek 的返回值赋予一个 char 而不是一个 int。这样做是错误的，但编译器却不能发现这个错误。最终会发生什么依赖于程序运行于哪台机器以及输入数据是什么。例如，在一台 char 被实现为 unsigned char 的机器上，下面的循环永远不会停止：

```
char ch; // 此处使用 char 就是引入灾难!
// 从 cin.get 返回的值被转换为 char，然后与一个 int 比较
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

问题出在当 get 返回 EOF 时，此值会被转换为一个 unsigned char。转换得到的值与 EOF 的 int 值不再相等，因此循环永远也不会停止。这种错误很可能在调试时发现。

在一台 char 被实现为 signed char 的机器上，我们不能确定循环的行为。当一个越界的值被赋予一个 signed 变量时会发生什么完全取决于编译器。在很多机器上，这个循环可以正常工作，除非输入序列中有一个字符与 EOF 值匹配。虽然在普通数据中这种字符不太可能出现，但低层 IO 通常用于读取二进制值的场合，而这些二进制值不能直接映射到普通字符和数值。例如，在我们的机器上，如果输入中包含有一个值为 '\377' 的字符，则循环会提前终止。因为在我们的机器上，将 -1 转换为一个 signed char，就会得到 '\377'。如果输入中有这个值，则它会被（过早）当作文件尾指示符。

当我们读写有类型的值时，这种错误就不会发生。如果你可以使用标准库提供的类型更加安全、更高层的操作，就应该使用它们。

17.5.2 节练习

练习 17.37: 用未格式化版本的 `getline` 逐行读取一个文件。测试你的程序，给它一个文件，既包含空行又包含长度超过你传递给 `getline` 的字符数组大小的行。

练习 17.38: 扩展上一题中你的程序，将读入的每个单词打印到它所在的行。

17.5.3 流随机访问

各种流类型通常都支持对流中数据的随机访问。我们可以重定位流，使之跳过一些数据，首先读取最后一行，然后读取第一行，依此类推。标准库提供了一对函数，来定位(`seek`)到流中给定的位置，以及告诉(`tell`)我们当前位置。



随机 IO 本质上是依赖于系统的。为了理解如何使用这些特性，你必须查询系统文档。

虽然标准库为所有流类型都定义了 `seek` 和 `tell` 函数，但它们是否会被做有意义的事情依赖于流绑定到哪个设备。在大多数系统中，绑定到 `cin`、`cout`、`cerr` 和 `clog` 的流不支持随机访问——毕竟，当我们向 `cout` 直接输出数据时，类似向回跳十个位置这种操作是没有意义的。对这些流我们可以调用 `seek` 和 `tell` 函数，但在运行时会出错，将流置于一个无效状态。



WARNING 由于 `istream` 和 `ostream` 类型通常不支持随机访问，所以本节剩余内容只适用于 `fstream` 和 `sstream` 类型。

764

765

seek 和 tell 函数

为了支持随机访问，`IO` 类型维护一个标记来确定下一个读写操作要在哪里进行。它们还提供了两个函数：一个函数通过将标记 `seek` 到一个给定位置来重定位它；另一个函数 `tell` 我们标记的当前位置。标准库实际上定义了两对 `seek` 和 `tell` 函数，如表 17.21 所示。一对用于输入流，另一对用于输出流。输入和输出版本的差别在于名字的后缀是 `g` 还是 `p`。`g` 版本表示我们正在“获得”（读取）数据，而 `p` 版本表示我们正在“放置”（写入）数据。

表 17.21: `seek` 和 `tell` 函数

<code>tellg()</code>	返回一个输入流中 (<code>tellg</code>) 或输出流中 (<code>tellp</code>) 标记的当前位置
<code>tellp()</code>	
<code>seekg(pos)</code>	在一个输入流或输出流中将标记重定位到给定的绝对地址。 <code>pos</code> 通常是前一个 <code>tellg</code> 或 <code>tellp</code> 返回的值
<code>seekp(pos)</code>	
<code>seekp(off, from)</code>	在一个输入流或输出流中将标记定位到 <code>from</code> 之前或之后 <code>off</code> 个字符， <code>from</code> 可以是下列值之一 <ul style="list-style-type: none"> • <code>beg</code>, 偏移量相对于流开始位置 • <code>cur</code>, 偏移量相对于流当前位置 • <code>end</code>, 偏移量相对于流结尾位置
<code>seekg(off, from)</code>	

从逻辑上讲，我们只能对 `istream` 和派生自 `istream` 的类型 `ifstream` 和 `istringstream`（参见 8.1 节，第 278 页）使用 `g` 版本，同样只能对 `ostream` 和派生自 `ostream` 的类型 `ofstream` 和 `ostringstream` 使用 `p` 版本。一个 `iostream`、

`fstream` 或 `stringstream` 既能读又能写关联的流，因此对这些类型的对象既能使用 g 版本又能使用 p 版本。

只有一个标记

标准库区分 `seek` 和 `tell` 函数的“放置”和“获得”版本这一特性可能会导致误解。即使标准库进行了区分，但它在一个流中只维护单一的标记——并不存在独立的读标记和写标记。

当我们处理一个只读或只写的流时，两种版本的区别甚至是不明显的。我们可以对这些流只使用 g 或只使用 p 版本。如果我们试图对一个 `ifstream` 流调用 `tellp`，编译器会报告错误。类似的，编译器也不允许我们对一个 `ostringstream` 调用 `seekg`。

`fstream` 和 `stringstream` 类型可以读写同一个流。在这些类型中，有单一的缓冲区用于保存读写的数据，同样，标记也只有一个，表示缓冲区中的当前位置。标准库将 g 和 p 版本的读写位置都映射到这个单一的标记。



由于只有单一的标记，因此只要我们在读写操作间切换，就必须进行 `seek` 操作来重定位标记。

< 766

重定位标记

`seek` 函数有两个版本：一个移动到文件中的“绝对”地址；另一个移动到一个给定位置的指定偏移量：

```
// 将标记移动到一个固定位置  
seekg(new_position); // 将读标记移动到指定的 pos_type 类型的位置  
seekp(new_position); // 将写标记移动到指定的 pos_type 类型的位置  
  
// 移动到给定起始点之前或之后指定的偏移位置  
seekg(offset, from); // 将读标记移动到距 from 偏移量为 offset 的位置  
seekp(offset, from); // 将写标记移动到距 from 偏移量为 offset 的位置
```

`from` 的可能值如表 17.21 所示。

参数 `new_position` 和 `offset` 的类型分别是 `pos_type` 和 `off_type`，这两个类型都是机器相关的，它们定义在头文件 `istream` 和 `ostream` 中。`pos_type` 表示一个文件位置，而 `off_type` 表示距当前位置的一个偏移量。一个 `off_type` 类型的值可以是正的也可以是负的，即，我们可以在文件中向前移动或向后移动。

访问标记

函数 `tellg` 和 `tellp` 返回一个 `pos_type` 值，表示流的当前位置。`tell` 函数通常用来记住一个位置，以便稍后再定位回来：

```
// 记住当前写位置  
ostringstream writeStr; // 输出 stringstream  
ostringstream::pos_type mark = writeStr.tellp();  
// ...  
if (cancelEntry)  
    // 回到刚才记住的位置  
    writeStr.seekp(mark);
```

读写同一个文件

我们来考察一个编程实例。假定已经给定了一个要读取的文件，我们要在此文件的末尾写入新的一行，这一行包含文件中每行的相对起始位置。例如，给定下面文件：

```
abcd
efg
hi
j
```

程序应该生成如下修改过的文件：

767 →

```
abcd
efg
hi
j
5 9 12 14
```

注意，我们的程序不必输出第一行的偏移——它总是从位置 0 开始。还要注意，统计偏移量时必须包含每行末尾不可见的换行符。最后，注意输出的最后一个数是我们的输出开始那行的偏移量。在输出中包含了这些偏移量后，我们的输出就与文件的原始内容区分开来了。我们可以读取结果文件中最后一个数，定位到对应偏移量，即可得到我们的输出的起始地址。

我们的程序将逐行读取文件。对每一行，我们将递增计数器，将刚刚读取的一行的长度加到计数器上，则此计数器即为下一行的起始地址：

```
int main()
{
    // 以读写方式打开文件，并定位到文件尾
    // 文件模式参数参见 8.2.2 节（第 286 页）
    fstream inOut("copyOut",
                  fstream::ate | fstream::in | fstream::out);
    if (!inOut) {
        cerr << "Unable to open file!" << endl;
        return EXIT_FAILURE; // EXIT_FAILURE 参见 6.3.2 节（第 204 页）
    }
    // inOut 以 ate 模式打开，因此一开始就定义到其文件尾
    auto end_mark = inOut.tellg(); // 记住原文件尾位置
    inOut.seekg(0, fstream::beg); // 重定位到文件开始
    size_t cnt = 0; // 字节数累加器
    string line; // 保存输入中的每行
    // 继续读取的条件：还未遇到错误且还在读取原数据
    while (inOut && inOut.tellg() != end_mark
           && getline(inOut, line)) { // 且还可获取一行输入
        cnt += line.size() + 1; // 加 1 表示换行符
        auto mark = inOut.tellg(); // 记住读取位置
        inOut.seekp(0, fstream::end); // 将写标记移动到文件尾
        inOut << cnt; // 输出累计的长度
        // 如果不是最后一行，打印一个分隔符
        if (mark != end_mark) inOut << " ";
        inOut.seekg(mark); // 恢复读位置
    }
    inOut.seekp(0, fstream::end); // 定位到文件尾
```

```
    inout << "\n"; // 在文件尾输出一个换行符
    return 0;
}
```

我们的程序用 `in`、`out` 和 `ate` 模式（参见 8.2.2 节，第 286 页）打开 `fstream`。前两个模式指出我们想读写同一个文件。指定 `ate` 会将读写标记定位到文件尾。与往常一样，我们检查文件是否成功打开，如果失败就退出（参见 6.3.2 节，第 203 页）。 768

由于我们的程序向输入文件写入数据，因此不能通过文件尾来判断是否停止读取，而是应该在达到原数据的末尾时停止。因此，我们必须首先记住原文件尾的位置。由于我们是以 `ate` 模式打开文件的，因此 `inout` 已经定位到文件尾了。我们将当前位置（即，原文件尾）保存在 `end_mark` 中。记住文件尾位置之后，我们 `seek` 到距文件起始位置偏移量为 0 的地方，即，将读标记重定位到文件起始位置。

`while` 循环的条件由三部分组成：首先检查流是否合法；如果合法，通过比较当前读位置（由 `tellg` 返回）和记录在 `end_mark` 中的位置来检查是否读完了原数据；最后，假定前两个检查都已成功，我们调用 `getline` 读取输入的下一行，如果 `getline` 成功，则执行 `while` 循环体。

循环体首先将当前位置记录在 `mark` 中。我们保存当前位置是为了在输出下一个偏移量后再退回来。接下来调用 `seekp` 将写标记重定位到文件尾。我们输出计数器的值，然后调用 `seekg` 回到记录在 `mark` 中的位置。回退到原位置后，我们就准备好继续检查循环条件了。

每步循环都会输出下一行的偏移量。因此，最后一步循环负责输出最后一行的偏移量。但是，我们还需要在文件尾输出一个换行符。与其他写操作一样，在输出换行符之前我们调用 `seekp` 来定位到文件尾。

17.5.3 节练习

练习 17.39：对本节给出的 `seek` 程序，编写你自己的版本。

769

小结

本章介绍了一些特殊 IO 操作和四个标准库类型: `tuple`、`bitset`、正则表达式和随机数。

`tuple` 是一个模板, 允许我们将多个不同类型的成员捆绑成单一对象。每个 `tuple` 包含指定数量的成员, 但对一个给定的 `tuple` 类型, 标准库并未限制我们可以定义的成员数量上限。

`bitset` 允许我们定义指定大小的二进制位集合。标准库不限制一个 `bitset` 的大小必须与整型类型的大小匹配, `bitset` 的大小可以更大。除了支持普通的位运算符 (参见 4.8 节, 第 136 页) 外, `bitset` 还定义了一些命名的操作, 允许我们操纵 `bitset` 中特定位的状态。

正则表达式库提供了一组类和函数: `regex` 类管理用某种正则表达式语言编写的正则表达式。匹配类保存了某个特定匹配的相关信息。这些类被函数 `regex_search` 和 `regex_match` 所用。这两个函数接受一个 `regex` 对象和一个字符序列, 检查 `regex` 中的正则表达式是否匹配给定的字符序列。`regex` 迭代器类型是迭代器适配器, 它们使用 `regex_search` 遍历输入序列, 返回每个匹配的子序列。标准库还定义了一个 `regex_replace` 函数, 允许我们用指定内容替换输入序列中与正则表达式匹配的部分。

随机数库由一组随机数引擎类和分布类组成。随机数引擎返回一个均匀分布的整型值序列。标准库定义了多个引擎, 它们具有不同的性能特点。`default_random_engine` 是适合于大多数普通情况的引擎。标准库还定义了 20 个分布类型。这些分布类型使用一个引擎来生成指定类型的随机数, 这些随机数的值都在给定范围内, 且分布满足指定的概率分布。

术语表

bitset 标准库类, 保存二进制位集合, 大小在编译时已知, 并提供检测和设置集合中二进制位的操作。

cmatch csub_match 对象的容器, 保存一个 `regex` 与一个 `const char*` 输入序列匹配的相关信息。容器首元素描述了整个匹配结果。后续元素描述了子表达式的匹配结果。

cregex_iterator 类似 `sregex_iterator`, 唯一的差别是此迭代器遍历一个 `char` 数组。

csub_match 保存一个正则表达式与一个 `const char*` 匹配结果的类型。可以表示整个匹配或子表达式的匹配。

默认随机数引擎 (default random engine) 用于普通用途的随机数引擎的类型别名。

770

格式化 IO (formatted IO) 读写操作, 利用要读写的对象的类型来定义操作的行为。格式化输入操作执行适合要读取的类型的转换操作, 如将 ASCII 码字符串转换为算术类型以及 (默认地) 忽略空白符。格式化输出操作将类型转换为可打印的字符表示形式、补白输出, 还可能执行其他与输出类型相关的转换。

get 模板函数, 返回给定 `tuple` 的指定成员。例如, `get<0>(t)` 返回 `tuplet` 的第一个成员。

高位 (high-order) `bitset` 中下标最大的那些位。

低位 (low-order) `bitset` 中下标最小的那些位。

操纵符 (manipulator) “操纵”流的类函数对象。操纵符可用作重载的 IO 运算符<<和>>的右侧运算对象。大多数操纵符会改变流对象的内部状态。这种操纵符通常是一对的——一个改变状态，另一个恢复到流的默认状态。

随机数分布 (random-number distribution) 标准库类型，根据其名字所指出的概率分布转换随机数引擎的输出值。例如，`uniform_int_distribution<T>`生成类型为 `T` 的均匀分布的整数，而 `normal_distribution<T>` 生成正态分布的值，依此类推。

随机数引擎 (random-number engine) 标准库类型，生成随机的无符号数。引擎的设计意图是只用作随机数分布的输入。

随机数发生器 (random-number generator) 一个随机数引擎类型和一个分布类型的组合。

regex 管理正则表达式的类。

regex_error 异常类型，当正则表达式中存在语法错误时抛出此异常。

regex_match 确定整个输入序列是否与给定 `regex` 对象匹配的函数。

regex_replace 使用一个 `regex` 对象来匹配输入序列并用给定格式替换匹配的子表达式的函数。

regex_search 使用一个 `regex` 对象在给定输入序列中查找匹配的子序列的函数。

正则表达式 (regular expression) 一种描述字符序列的方式。

种子 (seed) 提供给随机数引擎的值，使引擎移动到生成的随机数序列中一个新的点。

smatch ssub_match 对象的容器，提供一个 `regex` 与一个 `string` 输入序列匹配的相关信息。容器首元素描述了整个匹配结果。后续元素描述了子表达式的匹配结果。

sregex_iterator 迭代器，使用给定的 `regex` 对象遍历一个 `string` 来查找匹配子串。其构造函数通过调用 `regex_search` 将迭代器定位到第一个匹配。递增迭代器的操作会调用 `regex_search`，从给定 `string` 中当前匹配之后的位置开始查找匹配。解引用迭代器返回一个描述当前匹配的 `smatch` 对象。

ssub_match 保存正则表达式与 `string` 匹配结果的类型。可以描述整个匹配或子表达式的匹配。

子表达式 (subexpression) 正则表达式模式中用括号包围的组成部分。

tuple 模板，生成的类型保存指定类型的未命名成员。标准库没有限制一个 `tuple` 最多可以包含多少个成员。

未格式化 IO (unformatted IO) 将流当作无差别的字节流来处理的操作。未格式化操作给用户增加了很多管理 IO 的负担。

第 18 章

用于大型程序的工具

内容

18.1 异常处理.....	684
18.2 命名空间.....	695
18.3 多重继承与虚继承.....	710
小结	722
术语表.....	722

C++语言能解决的问题规模千变万化，有的小到一个程序员几小时就能完成，有的则是含有几千几万行代码的庞大系统，需要几百个程序员协同工作好几年。本书之前介绍的内容对各种规模的编程问题都适用。

除此之外，C++语言还包含其他一些特征，当我们编写比较复杂的、小组和个人难以管理的系统时，这些特征最为有用。本章的主题即是向读者介绍这些特征，它们包括异常处理、命名空间和多重继承。

772 与仅需几个程序员就能开发完成的系统相比，大规模编程对程序设计语言的要求更高。大规模应用程序的特殊要求包括：

- 在独立开发的子系统之间协同处理错误的能力。
- 使用各种库（可能包含独立开发的库）进行协同开发的能力。
- 对比较复杂的应用概念建模的能力。

本章介绍的三种 C++ 语言特性正好能满足上述要求，它们是：异常处理、命名空间和多重继承。

18.1 异常处理

异常处理（exception handling）机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并做出相应的处理。异常使得我们能够将问题的检测与解决过程分离开来。程序的一部分负责检测问题的出现，然后解决该问题的任务传递给程序的另一部分。检测环节无须知道问题处理模块的所有细节，反之亦然。

在 5.6 节（第 173 页）我们曾介绍过一些有关异常处理的基本概念和机理，本节将继续扩展这些知识。对于程序员来说，要想有效地使用异常处理，必须首先了解当抛出异常时发生了什么，捕获异常时发生了什么，以及用来传递错误的对象的意义。

18.1.1 抛出异常

在 C++ 语言中，我们通过抛出（throwing）一条表达式来引发（raised）一个异常。被抛出的表达式的类型以及当前的调用链共同决定了哪段处理代码（handler）将被用来处理该异常。被选中的处理代码是在调用链中与抛出对象类型匹配的最近的处理代码。其中，根据抛出对象的类型和内容，程序的异常抛出部分将会告知异常处理部分到底发生了什么错误。

当执行一个 throw 时，跟在 throw 后面的语句将不再被执行。相反，程序的控制权从 throw 转移到与之匹配的 catch 模块。该 catch 可能是同一个函数中的局部 catch，也可能位于直接或间接调用了发生异常的函数的另一个函数中。控制权从一处转移到另一处，这有两个重要的含义：

- 沿着调用链的函数可能会提早退出。
- 一旦程序开始执行异常处理代码，则沿着调用链创建的对象将被销毁。

因为跟在 throw 后面的语句将不再被执行，所以 throw 语句的用法有点类似于 return 语句：它通常作为条件语句的一部分或者作为某个函数的最后（或者唯一）一条语句。

773 栈展开

当抛出一个异常后，程序暂停当前函数的执行过程并立即开始寻找与异常匹配的 catch 子句。当 throw 出现在一个 try 语句块（try block）内时，检查与该 try 块关联的 catch 子句。如果找到了匹配的 catch，就使用该 catch 处理异常。如果这一步没找到匹配的 catch 且该 try 语句嵌套在其他 try 块中，则继续检查与外层 try 匹配的 catch 子句。如果还是找不到匹配的 catch，则退出当前的函数，在调用当前函数的外层函数中继续寻找。

如果对抛出异常的函数的调用语句位于一个 try 语句块内，则检查与该 try 块关联

的 catch 子句。如果找到了匹配的 catch，就使用该 catch 处理异常。否则，如果该 try 语句嵌套在其他 try 块中，则继续检查与外层 try 匹配的 catch 子句。如果仍然没有找到匹配的 catch，则退出当前这个主调函数，继续在调用了刚刚退出的这个函数的其他函数中寻找，以此类推。

上述过程被称为栈展开 (stack unwinding) 过程。栈展开过程沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的 catch 子句为止；或者也可能一直没找到匹配的 catch，则退出主函数后查找过程终止。

假设找到了一个匹配的 catch 子句，则程序进入该子句并执行其中的代码。当执行完这个 catch 子句后，找到与 try 块关联的最后一个 catch 子句之后的点，并从这里继续执行。

如果没找到匹配的 catch 子句，程序将退出。因为异常通常被认为是妨碍程序正常执行的事件，所以一旦引发了某个异常，就不能对它置之不理。当找不到匹配的 catch 时，程序将调用标准库函数 **terminate**，顾名思义，**terminate** 负责终止程序的执行过程。



一个异常如果没有被捕获，则它将终止当前的程序。

栈展开过程中对象被自动销毁

在栈展开过程中，位于调用链上的语句块可能会提前退出。通常情况下，程序在这些块中创建了一些局部对象。我们已经知道，块退出后它的局部对象也将随之销毁，这条规则对于栈展开过程同样适用。如果在栈展开过程中退出了某个块，编译器将负责确保在这个块中创建的对象能被正确地销毁。如果某个局部对象的类型是类类型，则该对象的析构函数将被自动调用。与往常一样，编译器在销毁内置类型的对象时不需要做任何事情。

如果异常发生在构造函数中，则当前的对象可能只构造了一部分。有的成员已经初始化了，而另外一些成员在异常发生前也许还没有初始化。即使某个对象只构造了一部分，我们也要确保已构造的成员能被正确地销毁。

类似的，异常也可能发生在数组或标准库容器的元素初始化过程中。与之前类似，如果在异常发生前已经构造了一部分元素，则我们应该确保这部分元素被正确地销毁。

析构函数与异常

< 774

析构函数总是会被执行的，但是函数中负责释放资源的代码却可能被跳过，这一特点对于我们如何组织程序结构有重要影响。如我们在 12.1.4 节（第 415 页）介绍过的，如果一个块分配了资源，并且在负责释放这些资源的代码前面发生了异常，则释放资源的代码将不会被执行。另一方面，类对象分配的资源将由类的析构函数负责释放。因此，如果我们使用类来控制资源的分配，就能确保无论函数正常结束还是遭遇异常，资源都能被正确地释放。

析构函数在栈展开的过程中执行，这一事实影响着我们编写析构函数的方式。在栈展开的过程中，已经引发了异常但是我们还没有处理它。如果异常抛出后没有被正确捕获，则系统将调用 **terminate** 函数。因此，出于栈展开可能使用析构函数的考虑，析构函数不应该抛出不能被它自身处理的异常。换句话说，如果析构函数需要执行某个可能抛出异常的操作，则该操作应该被放置在一个 try 语句块当中，并且在析构函数内部得到处理。

在实际的编程过程中，因为析构函数仅仅是释放资源，所以它不太可能抛出异常。所有标准库类型都能确保它们的析构函数不会引发异常。



在栈展开的过程中，运行类类型的局部对象的析构函数。因为这些析构函数是自动执行的，所以它们不应该抛出异常。一旦在栈展开的过程中析构函数抛出了异常，并且析构函数自身没能捕获到该异常，则程序将被终止。

异常对象

异常对象 (exception object) 是一种特殊的对象，编译器使用异常抛出表达式来对异常对象进行拷贝初始化 (参见 13.1.1 节，第 441 页)。因此，`throw` 语句中的表达式必须拥有完全类型 (参见 7.3.3 节，第 250 页)。而且如果该表达式是类类型的话，则相应的类必须含有一个可访问的析构函数和一个可访问的拷贝或移动构造函数。如果该表达式是数组类型或函数类型，则表达式将被转换成与之对应的指针类型。

异常对象位于由编译器管理的空间中，编译器确保无论最终调用的是哪个 `catch` 子句都能访问该空间。当异常处理完毕后，异常对象被销毁。

如我们所知，当一个异常被抛出时，沿着调用链的块将依次退出直至找到与异常匹配的处理代码。如果退出了某个块，则同时释放块中局部对象使用的内存。因此，抛出一个指向局部对象的指针几乎肯定是一种错误的行为。出于同样的原因，从函数中返回指向局部对象的指针也是错误的 (参见 6.3.2 节，第 202 页)。如果指针所指的对象位于某个块中，而该块在 `catch` 语句之前就已经退出了，则意味着在执行 `catch` 语句之前局部对象已经被销毁了。

当我们抛出一条表达式时，该表达式的静态编译时类型 (参见 15.2.3 节，第 534 页) 决定了异常对象的类型。读者必须牢记这一点，因为很多情况下程序抛出的表达式类型来自于某个继承体系。如果一条 `throw` 表达式解引用一个基类指针，而该指针实际指向的是派生类对象，则抛出的对象将被切掉一部分 (参见 15.2.3 节，第 535 页)，只有基类部分被抛出。

775



抛出指针要求在任何对应的处理代码存在的地方，指针所指的对象都必须存在。

18.1.1 节练习

练习 18.1： 在下列 `throw` 语句中异常对象的类型是什么？

(a) <code>range_error r("error");</code>	(b) <code>exception *p = &r;</code>
<code>throw r;</code>	<code>,</code>
	<code>throw *p;</code>

如果将 (b) 中的 `throw` 语句写成了 `throw p` 将发生什么情况？

练习 18.2： 当在指定的位置发生了异常时将出现什么情况？

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // 此处发生异常
}
```

练习 18.3: 要想让上面的代码在发生异常时能正常工作，有两种解决方案。请描述这两种方法并实现它们。

18.1.2 捕获异常

catch 子句 (catch clause) 中的异常声明 (exception declaration) 看起来像是只包含一个形参的函数形参列表。像在形参列表中一样，如果 catch 无须访问抛出的表达式的话，则我们可以忽略捕获形参的名字。

声明的类型决定了处理代码所能捕获的异常类型。这个类型必须是完全类型 (参见 7.3.3 节, 第 250 页)，它可以是左值引用，但不能是右值引用 (参见 13.6.1 节, 第 471 页)。

当进入一个 catch 语句后，通过异常对象初始化异常声明中的参数。和函数的参数类似，如果 catch 的参数类型是非引用类型，则该参数是异常对象的一个副本，在 catch 语句内改变该参数实际上改变的是局部副本而非异常对象本身；相反，如果参数是引用类型，则和其他引用参数一样，该参数是异常对象的一个别名，此时改变参数也就是改变异常对象。

catch 的参数还有一个特性也与函数的参数非常类似：如果 catch 的参数是基类类型，则我们可以使用其派生类类型的异常对象对其进行初始化。此时，如果 catch 的参数是非引用类型，则异常对象将被切掉一部分 (参见 15.2.3 节, 第 535 页)，这与将派生类对象以值传递的方式传给一个普通函数差不多。另一方面，如果 catch 的参数是基类的引用，则该参数将以常规方式绑定到异常对象上。

最后一点需要注意的是，异常声明的静态类型将决定 catch 语句所能执行的操作。如果 catch 的参数是基类类型，则 catch 无法使用派生类特有的任何成员。



通常情况下，如果 catch 接受的异常与某个继承体系有关，则最好将该 catch 的参数定义成引用类型。

<776

查找匹配的处理代码

在搜寻 catch 语句的过程中，我们最终找到的 catch 未必是异常的最佳匹配。相反，挑选出来的应该是第一个与异常匹配的 catch 语句。因此，越是专门的 catch 越应该置于整个 catch 列表的前端。

因为 catch 语句是按照其出现的顺序逐一进行匹配的，所以当程序使用具有继承关系的多个异常时必须对 catch 语句的顺序进行组织和管理，使得派生类异常的处理代码出现在基类异常的处理代码之前。

与实参和形参的匹配规则相比，异常和 catch 异常声明的匹配规则受到更多限制。此时，绝大多数类型转换都不被允许，除了一些极细小的差别之外，要求异常的类型和 catch 声明的类型是精确匹配的：

- 允许从非常量向常量的类型转换，也就是说，一条非常量对象的 throw 语句可以匹配一个接受常量引用的 catch 语句。
- 允许从派生类向基类的类型转换。
- 数组被转换成指向数组 (元素) 类型的指针，函数被转换成指向该函数类型的指针。

除此之外，包括标准算术类型转换和类类型转换在内，其他所有转换规则都不能在匹配

catch 的过程中使用。



如果在多个 catch 语句的类型之间存在着继承关系，则我们应该把继承链最底端的类（most derived type）放在前面，而将继承链最顶端的类（least derived type）放在后面。

重新抛出

有时，一个单独的 catch 语句不能完整地处理某个异常。在执行了某些校正操作之后，当前的 catch 可能会决定由调用链更上一层的函数接着处理异常。一条 catch 语句通过重新抛出（rethrowing）的操作将异常传递给另外一个 catch 语句。这里的重新抛出仍然是一条 throw 语句，只不过不包含任何表达式：

```
throw;
```

空的 throw 语句只能出现在 catch 语句或 catch 语句直接或间接调用的函数之内。如果在处理代码之外的区域遇到了空 throw 语句，编译器将调用 terminate。

一个重新抛出语句并不指定新的表达式，而是将当前的异常对象沿着调用链向上传递。

很多时候，catch 语句会改变其参数的内容。如果在改变了参数的内容后 catch 语句重新抛出异常，则只有当 catch 异常声明是引用类型时我们对参数所做的改变才会被保留并继续传播：

```
catch (my_error &eObj) {           // 引用类型
    eObj.status = errCodes::severeErr; // 修改了异常对象
    throw;                          // 异常对象的 status 成员是 severeErr
} catch (other_error eObj) {        // 非引用类型
    eObj.status = errCodes::badErr;  // 只修改了异常对象的局部副本
    throw;                          // 异常对象的 status 成员没有改变
}
```

捕获所有异常的处理代码

有时我们希望不论抛出的异常是什么类型，程序都能统一捕获它们。要想捕获所有可能的异常是比较有难度的，毕竟有些情况下我们也不知道异常的类型到底是什么。即使我们知道所有的异常类型，也很难为所有类型提供唯一一个 catch 语句。为了一次性捕获所有异常，我们使用省略号作为异常声明，这样的处理代码称为捕获所有异常（catch-all）的处理代码，形如 catch(...)。一条捕获所有异常的语句可以与任意类型的异常匹配。

catch(...) 通常与重新抛出语句一起使用，其中 catch 执行当前局部能完成的工作，随后重新抛出异常：

```
void manip() {
    try {
        // 这里的操作将引发并抛出一个异常
    }
    catch (...) {
        // 处理异常的某些特殊操作
        throw;
    }
}
```

`catch(...)`既能单独出现，也能与其他几个 `catch` 语句一起出现。



如果 `catch(...)` 与其他几个 `catch` 语句一起出现，则 `catch(...)` 必须在最后的位置。出现在捕获所有异常语句后面的 `catch` 语句将永远不会被匹配。

18.1.2 节练习

练习 18.4：查看图 18.1（第 693 页）所示的继承体系，说明下面的 `try` 块有何错误并修改它。

```
try {
    // 使用 C++ 标准库
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
```

练习 18.5：修改下面的 `main` 函数，使其能捕获图 18.1（第 693 页）所示的任何异常类型：

```
int main() {
    // 使用 C++ 标准库
}
```

处理代码应该首先打印异常相关的错误信息，然后调用 `abort`（定义在 `cstdlib` 头文件中）终止 `main` 函数。

练习 18.6：已知下面的异常类型和 `catch` 语句，书写一个 `throw` 表达式使其创建的异常对象能被这些 `catch` 语句捕获：

- (a) class exceptionType { };
 catch(exceptionType *pet) { }
- (b) catch(...) { }
- (c) typedef int EXCPTYPE;
 catch(EXCPTYPE) { }

18.1.3 函数 `try` 语句块与构造函数

通常情况下，程序执行的任何时刻都可能发生异常，特别是异常可能发生在处理构造函数初始值的过程中。构造函数在进入其函数体之前首先执行初始值列表。因为在初始值列表抛出异常时构造函数体内的 `try` 语句块还未生效，所以构造函数体内的 `catch` 语句无法处理构造函数初始值列表抛出的异常。

要想处理构造函数初始值抛出的异常，我们必须将构造函数写成函数 `try` 语句块（也称为函数测试块，function try block）的形式。函数 `try` 语句块使得一组 `catch` 语句既能处理构造函数体（或析构函数体），也能处理构造函数的初始化过程（或析构函数的析构过程）。举个例子，我们可以把 `Blob` 的构造函数（参见 16.1.2 节，第 586 页）置于一个函数 `try` 语句块中：

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il) try :
```

```

        data(std::make_shared<std::vector<T>>(il)) {
    /* 空函数体 */
} catch(const std::bad_alloc &e) { handle_out_of_memory(e); }

```

注意：关键字 `try` 出现在表示构造函数初始值列表的冒号以及表示构造函数体（此例为空）的花括号之前。与这个 `try` 关联的 `catch` 既能处理构造函数体抛出的异常，也能处理成员初始化列表抛出的异常。

还有一种情况值得读者注意，在初始化构造函数的参数时也可能发生异常，这样的异常不属于函数 `try` 语句块的一部分。函数 `try` 语句块只能处理构造函数开始执行后发生的异常。和其他函数调用一样，如果在参数初始化的过程中发生了异常，则该异常属于调用表达式的一部分，并将在调用者所在的上下文中处理。



处理构造函数初始值异常的唯一方法是将构造函数写成函数 `try` 语句块。

18.1.3 节练习

练习 18.7：根据第 16 章的介绍定义你自己的 `Blob` 和 `BlobPtr`，注意将构造函数写成函数 `try` 语句块。

18.1.4 noexcept 异常说明

对于用户及编译器来说，预先知道某个函数不会抛出异常显然大有裨益。首先，知道函数不会抛出异常有助于简化调用该函数的代码；其次，如果编译器确认函数不会抛出异常，它就能执行某些特殊的优化操作，而这些优化操作并不适用于可能出错的代码。

在 C++11 新标准中，我们可以通过提供 **`noexcept` 说明** (`noexcept` specification) 指定某个函数不会抛出异常。其形式是关键字 `noexcept` 紧跟在函数的参数列表后面，用以标识该函数不会抛出异常：

```

void recoup(int) noexcept;           // 不会抛出异常
void alloc(int);                   // 可能抛出异常

```

这两条声明语句指出 `recoup` 将不会抛出任何异常，而 `alloc` 可能抛出异常。我们说 `recoup` 做了**不抛出说明** (**nonthrowing specification**)。

对于一个函数来说，`noexcept` 说明要么出现在该函数的所有声明语句和定义语句中，要么一次也不出现。该说明应该在函数的尾置返回类型（参见 6.3.3 节，第 206 页）之前。我们也可以在函数指针的声明和定义中指定 `noexcept`。在 `typedef` 或类型别名中则不能出现 `noexcept`。在成员函数中，`noexcept` 说明符需要跟在 `const` 及引用限定符之后，而在 `final`、`override` 或虚函数的 `=0` 之前。

违反异常说明

读者需要清楚的一个事实是编译器并不会在编译时检查 `noexcept` 说明。实际上，如果一个函数在说明了 `noexcept` 的同时又含有 `throw` 语句或者调用了可能抛出异常的其他函数，编译器将顺利编译通过，并不会因为这种违反异常说明的情况而报错（不排除个别编译器会对这种用法提出警告）：

```

// 尽管该函数明显违反了异常说明，但它仍然可以顺利编译通过
void f() noexcept           // 承诺不会抛出异常
{

```

```

    throw exception();           // 违反了异常说明
}

```

因此可能出现这样一种情况：尽管函数声明了它不会抛出异常，但实际上还是抛出了。一旦一个 `noexcept` 函数抛出了异常，程序就会调用 `terminate` 以确保遵守不在运行时抛出异常的承诺。上述过程对是否执行栈展开未作约定，因此 `noexcept` 可以用在两种情况下：一是我们确认函数不会抛出异常，二是我们根本不知道该如何处理异常。

指明某个函数不会抛出异常可以令该函数的调用者不必再考虑如何处理异常。无论是函数确实不抛出异常，还是程序被终止，调用者都无须为此负责。



通常情况下，编译器不能也不必在编译时验证异常说明。

WARNING

向后兼容：异常说明

早期的 C++ 版本设计了一套更加详细的异常说明方案，该方案使得我们可以指定某个函数可能抛出的异常类型。函数可以指定一个关键字 `throw`，在后面跟上括号括起来的异常类型列表。`throw` 说明符所在的位置与新版本 C++ 中 `noexcept` 所在的位置相同。

上述使用 `throw` 的异常说明方案在 C++11 新版本中已经被取消了。然而尽管如此，它还有一个重要的用处。如果函数被设计为是 `throw()` 的，则意味着该函数将不会抛出异常：

```

void recoup(int) noexcept;          // recoup 不会抛出异常
void recoup(int) throw();           // 等价的声明

```

上面的两条声明语句是等价的，它们都承诺 `recoup` 不会抛出异常。

异常说明的实参

`noexcept` 说明符接受一个可选的实参，该实参必须能转换为 `bool` 类型：如果实参是 `true`，则函数不会抛出异常；如果实参是 `false`，则函数可能抛出异常：

```

void recoup(int) noexcept(true);    // recoup 不会抛出异常
void alloc(int) noexcept(false);     // alloc 可能抛出异常

```

noexcept 运算符

`noexcept` 说明符的实参常常与 **noexcept 运算符** (`noexcept operator`) 混合使用。`noexcept` 运算符是一个一元运算符，它的返回值是一个 `bool` 类型的右值常量表达式，用于表示给定的表达式是否会抛出异常。和 `sizeof` (参见 4.9 节，第 139 页) 类似，`noexcept` 也不会求其运算对象的值。

例如，因为我们声明 `recoup` 时使用了 `noexcept` 说明符，所以下面的表达式的返回值为 `true`：

`noexcept(recoup(i))` // 如果 `recoup` 不抛出异常则结果为 `true`；否则结果为 `false`

更普通的形式是：

`noexcept(e)`

当 `e` 调用的所有函数都做了不抛出说明且 `e` 本身不含有 `throw` 语句时，上述表达式为 `true`；否则 `noexcept(e)` 返回 `false`。

C++
11

< 781

我们可以使用 `noexcept` 运算符得到如下的异常说明：

```
void f() noexcept(noexcept(g())); // f 和 g 的异常说明一致
```

如果函数 `g` 承诺了不会抛出异常，则 `f` 也不会抛出异常；如果 `g` 没有异常说明符，或者 `g` 虽然有异常说明符但是允许抛出异常，则 `f` 也可能抛出异常。



`noexcept` 有两层含义：当跟在函数参数列表后面时它是异常说明符；而当作为 `noexcept` 异常说明的 `bool` 实参出现时，它是一个运算符。

异常说明与指针、虚函数和拷贝控制

尽管 `noexcept` 说明符不属于函数类型的一部分，但是函数的异常说明仍然会影响函数的使用。

函数指针及该指针所指的函数必须具有一致的异常说明。也就是说，如果我们为某个指针做了不抛出异常的声明，则该指针将只能指向不抛出异常的函数。相反，如果我们显式或隐式地说明了指针可能抛出异常，则该指针可以指向任何函数，即使是承诺了不抛出异常的函数也可以：

```
// recoup 和 pf1 都承诺不会抛出异常
void (*pf1)(int) noexcept = recoup;
// 正确：recoup 不会抛出异常，pf2 可能抛出异常，二者之间互不干扰
void (*pf2)(int) = recoup;

pf1 = alloc;      // 错误：alloc 可能抛出异常，但是 pf1 已经说明了它不会抛出异常
pf2 = alloc;      // 正确：pf2 和 alloc 都可能抛出异常
```

如果一个虚函数承诺了它不会抛出异常，则后续派生出来的虚函数也必须做出同样的承诺；与之相反，如果基类的虚函数允许抛出异常，则派生类的对应函数既可以允许抛出异常，也可以不允许抛出异常：

```
class Base {
public:
    virtual double f1(double) noexcept; // 不会抛出异常
    virtual int f2() noexcept(false);   // 可能抛出异常
    virtual void f3();                 // 可能抛出异常
};

782> class Derived : public Base {
public:
    double f1(double);               // 错误：Base::f1 承诺不会抛出异常
    int f2() noexcept(false);        // 正确：与 Base::f2 的异常说明一致
    void f3() noexcept;              // 正确：Derived 的 f3 做了更严格的限定，
                                    // 这是允许的
};
```

当编译器合成拷贝控制成员时，同时也生成一个异常说明。如果对所有成员和基类的所有操作都承诺了不会抛出异常，则合成的成员是 `noexcept` 的。如果合成成员调用的任意一个函数可能抛出异常，则合成的成员是 `noexcept(false)`。而且，如果我们定义了一个析构函数但是没有为它提供异常说明，则编译器将合成一个。合成的异常说明将与假设由编译器为类合成析构函数时所得的异常说明一致。

18.1.4 节练习

练习 18.8: 回顾你之前编写的各个类，为它们的构造函数和析构函数添加正确的异常说明。如果你认为某个析构函数可能抛出异常，尝试修改代码使得该析构函数不会抛出异常。

18.1.5 异常类层次

标准库异常类（参见 5.6.3 节，第 176 页）构成了图 18.1 所示的继承体系（参见第 15 章）。

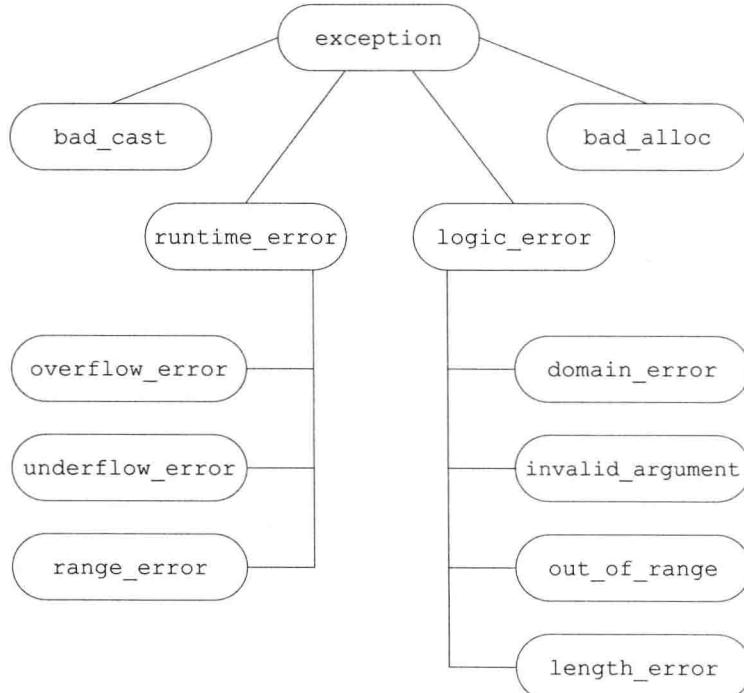


图 18.1: 标准 exception 类层次

类型 `exception` 仅仅定义了拷贝构造函数、拷贝赋值运算符、一个虚析构函数和一个名为 `what` 的虚成员。其中 `what` 函数返回一个 `const char*`，该指针指向一个以 `unll` 结尾的字符数组，并且确保不会抛出任何异常。

类 `exception`、`bad_cast` 和 `bad_alloc` 定义了默认构造函数。类 `runtime_error` 和 `logic_error` 没有默认构造函数，但是有一个可以接受 C 风格字符串或者标准库 `string` 类型实参的构造函数，这些实参负责提供关于错误的更多信息。在这些类中，`what` 负责返回用于初始化异常对象的信息。因为 `what` 是虚函数，所以当我们捕获基类的引用时，对 `what` 函数的调用将执行与异常对象动态类型对应的版本。

书店应用程序的异常类

实际的应用程序通常会自定义 `exception`（或者 `exception` 的标准库派生类）的派生类以扩展其继承体系。这些面向应用的异常类表示了与应用相关的异常条件。

如果我们构建的是一个真实的书店应用程序，则其中的类将比本书之前所示的复杂得多。复杂性的一个方面就是如何处理异常。实际上，我们很可能需要建立一个自己的异常

类体系，用它来表示与应用相关的各种问题。我们设计的异常类可能如下所示：

```
// 为某个书店应用程序设定的异常类
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):
        std::runtime_error(s) { }

};

class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
        std::logic_error(s) { }

    isbn_mismatch(const std::string &s,
        const std::string &lhs, const std::string &rhs):
        std::logic_error(s), left(lhs), right(rhs) { }

    const std::string left, right;
};

};
```

784 由上可知，我们的面向应用的异常类继承自标准异常类。和其他继承体系一样，异常类也可以看作按照层次关系组织的。层次越低，表示的异常情况就越特殊。例如，在异常类继承体系中位于最顶层的通常是 exception，exception 表示的含义是某处出错了，至于错误的细节则未作描述。

继承体系的第二层将 exception 划分为两个大的类别：运行时错误和逻辑错误。运行时错误表示的是只有在程序运行时才能检测到的错误；而逻辑错误一般指的是我们可以在程序代码中发现的错误。

我们的书店应用程序进一步细分上述异常类别。名为 out_of_stock 的类表示在运行时可能发生的错误，比如某些顺序无法满足；名为 isbn_mismatch 的类表示 logic_error 的一个特例，程序可以通过比较对象的 isbn() 结果来阻止或处理这一错误。

使用我们自己的异常类型

我们使用自定义异常类的方式与使用标准异常类的方式完全一样。程序在某处抛出异常类型的对象，在另外的地方捕获并处理这些出现的问题。举个例子，我们可以为 Sales_data 类定义一个复合加法运算符，当检测到参与加法的两个 ISBN 编号不一致时抛出名为 isbn_mismatch 的异常：

```
// 如果参与加法的两个对象并非同一书籍，则抛出一个异常
Sales_data&
Sales_data::operator+=(const Sales_data& rhs)
{
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong ISBNs", isbn(), rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

使用了复合加法运算符的代码将能检测到这一错误，进而输出一条相应的错误信息并继续完成其他任务：

```

// 使用之前设定的书店程序异常类
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) {           // 读取两条交易信息
    try {
        sum = item1 + item2;             // 计算它们的和
        // 此处使用 sum
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left isbn(" << e.left
            << ") right isbn(" << e.right << ")" << endl;
    }
}

```

18.1.5 节练习

785

练习 18.9: 定义本节描述的书店程序异常类，然后为 `Sales_data` 类重新编写一个复合赋值运算符并令其抛出一个异常。

练习 18.10: 编写程序令其对两个 ISBN 编号不相同的对象执行 `Sales_data` 的加法运算。为该程序编写两个不同的版本：一个处理异常，另一个不处理异常。观察并比较这两个程序的行为，用心体会当出现了一个未被捕获的异常时程序会发生什么情况。

练习 18.11: 为什么 `what` 函数不应该抛出异常？

18.2 命名空间

大型程序往往会使用多个独立开发的库，这些库又会定义大量的全局名字，如类、函数和模板等。当应用程序用到多个供应商提供的库时，不可避免地会发生某些名字相互冲突的情况。多个库将名字放置在全局命名空间中将引发**命名空间污染**（namespace pollution）。

传统上，程序员通过将其定义的全局实体名字设得很长来避免命名空间污染问题，这样的名字中通常包含表示名字所属库的前缀部分：

```

class cplusplus_primer_Query { ... };
string cplusplus_primer_make_plural(size_t, string&);

```

这种解决方案显然不太理想：对于程序员来说，书写和阅读这么长的名字费时费力且过于繁琐。

命名空间（namespace）为防止名字冲突提供了更加可控的机制。命名空间分割了全局命名空间，其中每个命名空间是一个作用域。通过在某个命名空间中定义库的名字，库的作者（以及用户）可以避免全局名字固有的限制。

18.2.1 命名空间定义

一个命名空间的定义包含两部分：首先是关键字 `namespace`，随后是命名空间的名字。在命名空间名字后面是一系列由花括号括起来的声明和定义。只要能出现在全局作用域中的声明就能置于命名空间内，主要包括：类、变量（及其初始化操作）、函数（及其定义）、模板和其他命名空间：

```

namespace cplusplus_primer {
    class Sales_data { /* ... */ };
}

```

```

Sales_data operator+(const Sales_data&,
                      const Sales_data&);
class Query { /* ... */ };
class Query_base { /* ... */ };
} // 命名空间结束后无须分号，这一点与块类似

```

786 上面的代码定义了一个名为 `cplusplus_primer` 的命名空间，该命名空间包含四个成员：三个类和一个重载的+运算符。

和其他名字一样，命名空间的名字也必须在定义它的作用域内保持唯一。命名空间既可以定义在全局作用域内，也可以定义在其他命名空间中，但是不能定义在函数或类的内部。



命名空间作用域后面无须分号。

每个命名空间都是一个作用域

和其他作用域类似，命名空间中的每个名字都必须表示该空间内的唯一实体。因为不同命名空间的作用域不同，所以在不同命名空间内可以有相同名字的成员。

定义在某个命名空间中的名字可以被该命名空间内的其他成员直接访问，也可以被这些成员内嵌作用域中的任何单位访问。位于该命名空间之外的代码则必须明确指出所用的名字属于哪个命名空间：

```

cplusplus_primer::Query q =
    cplusplus_primer::Query("hello");

```

如果其他命名空间（比如说 `AddisonWesley`）也提供了一个名为 `Query` 的类，并且我们希望使用这个类替代 `cplusplus_primer` 中定义的同名类，则可以按照如下方式修改代码：

```
AddisonWesley::Query q = AddisonWesley::Query("hello");
```

命名空间可以是不连续的

如我们在 16.5 节（第 626 页）介绍过的，命名空间可以定义在几个不同的部分，这一点与其他作用域不太一样。编写如下的命名空间定义：

```

namespace nsp {
// 相关声明
}

```

可能是定义了一个名为 `nsp` 的新命名空间，也可能是为已经存在的命名空间添加一些新成员。如果之前没有名为 `nsp` 的命名空间定义，则上述代码创建一个新的命名空间；否则，上述代码打开已经存在的命名空间定义并为其添加一些新成员的声明。

命名空间的定义可以不连续的特性使得我们可以将几个独立的接口和实现文件组成一个命名空间。此时，命名空间的组织方式类似于我们管理自定义类及函数的方式：

- 命名空间的一部分成员的作用是定义类，以及声明作为类接口的函数及对象，则这些成员应该置于头文件中，这些头文件将被包含在使用了这些成员的文件中。
- 命名空间成员的定义部分则置于另外的源文件中。

787 在程序中某些实体只能定义一次：如非内联函数、静态数据成员、变量等，命名空间中定义的名字也需要满足这一要求，我们可以通过上面的方式组织命名空间并达到目的。这种接口和实现分离的机制确保我们所需的函数和其他名字只定义一次，而只要是用到这些实

体的地方都能看到对于实体名字的声明。



定义多个类型不相关的命名空间应该使用单独的文件分别表示每个类型(或关联类型构成的集合)。

定义本书的命名空间

通过使用上述接口与实现分离的机制,我们可以将 `cplusplus_primer` 库定义在几个不同的文件中。`Sales_data` 类的声明及其函数将置于 `Sales_data.h` 头文件中,第 15 章介绍的 `Query` 类将置于 `Query.h` 头文件中,以此类推。对应的实现文件将分别是 `Sales_data.cc` 和 `Query.cc`:

```
// ---- Sales_data.h ----
// #include 应该出现在打开命名空间的操作之前
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* ... */;
        Sales_data operator+(const Sales_data&,
                               const Sales_data&);
    // Sales_data 的其他接口函数的声明
}
// ---- Sales_data.cc ----
// 确保#include 出现在打开命名空间的操作之前
#include "Sales_data.h"

namespace cplusplus_primer {
// Sales_data 成员及重载运算符的定义
}
```

程序如果想使用我们定义的库,必须包含必要的头文件,这些头文件中的名字定义在命名空间 `cplusplus_primer` 内:

```
// ---- user.cc ----
// Sales_data.h 头文件的名字位于命名空间 cplusplus_primer 中
#include "Sales_data.h"
int main()
{
    using cplusplus_primer::Sales_data;
    Sales_data transl, trans2;
    // ...
    return 0;
}
```

这种程序的组织方式提供了开发者和库用户所需的模块性。每个类仍组织在自己的接口和实现文件中,一个类的用户不必编译与其他类相关的名字。我们对用户隐藏了实现细节,同时允许文件 `Sales_data.cc` 和 `user.cc` 被编译并链接成一个程序而不会产生任何编译时错误或链接时错误。库的开发者可以分别实现每一个类,相互之间没有干扰。

有一点需要注意,在通常情况下,我们不把`#include` 放在命名空间内部。如果我们这么做了,隐含的意思是把头文件中所有的名字定义成该命名空间的成员。例如,如果 `Sales_data.h` 在包含 `string` 头文件前就已经打开了命名空间 `cplusplus_primer`,则程序将出错,因为这么做意味着我们试图将命名空间 `std` 嵌套在命名空间 `cplusplus_primer` 中。

定义命名空间成员

假定作用域中存在合适的声明语句，则命名空间中的代码可以使用同一命名空间定义的名字的简写形式：

```
#include "Sales_data.h"
namespace cplusplus_primer {           // 重新打开命名空间 cplusplus_primer
// 命名空间中定义的成员可以直接使用名字，此时无须前缀
std::istream&
operator>>(std::istream& in, Sales_data& s) { /* ... */ }
```

也可以在命名空间定义的外部定义该命名空间的成员。命名空间对于名字的声明必须在作用域内，同时该名字的定义需要明确指出其所属的命名空间：

```
// 命名空间之外定义的成员必须使用含有前缀的名字
cplusplus_primer::Sales_data
cplusplus_primer::operator+(const Sales_data& lhs,
                           const Sales_data& rhs)
{
    Sales_data ret(lhs);
    // ...
}
```

和定义在类外部的类成员一样，一旦看到含有完整前缀的名字，我们就可以确定该名字位于命名空间的作用域内。在命名空间 `cplusplus_primer` 内部，我们可以直接使用该命名空间的其他成员，比如在上面的代码中，可以直接使用 `Sales_data` 定义函数的形参。

尽管命名空间的成员可以定义在命名空间外部，但是这样的定义必须出现在所属命名空间的外层空间中。换句话说，我们可以在 `cplusplus_primer` 或全局作用域中定义 `Sales_data operator+`，但是不能在一个不相关的作用域中定义这个运算符。

模板特例化

789 模板特例化必须定义在原始模板所属的命名空间中（参见 16.5 节，第 626 页）。和其他命名空间名字类似，只要我们在命名空间中声明了特例化，就能在命名空间外部定义它了：

```
// 我们必须将模板特例化声明成 std 的成员
namespace std {
    template <> struct hash<Sales_data>;
}
// 在 std 中添加了模板特例化的声明后，就可以在命名空间 std 的外部定义它了
template <> struct std::hash<Sales_data>
{
    size_t operator()(const Sales_data& s) const
    { return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue); }
    // 其他成员与之前的版本一致
};
```

全局命名空间

全局作用域中定义的名字（即在所有类、函数及命名空间之外定义的名字）也就是定义在全局命名空间（global namespace）中。全局命名空间以隐式的方式声明，并且在所有

程序中都存在。全局作用域中定义的名字被隐式地添加到全局命名空间中。

作用域运算符同样可以用于全局作用域的成员，因为全局作用域是隐式的，所以它并没有名字。下面的形式

```
::member_name
```

表示全局命名空间中的一个成员。

嵌套的命名空间

嵌套的命名空间是指定义在其他命名空间中的命名空间：

```
namespace cplusplus_primer {
    // 第一个嵌套的命名空间：定义了库的 Query 部分
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // 第二个嵌套的命名空间：定义了库的 Sales_data 部分
    namespace Bookstore {
        class Quote { /* ... */ };
        class Disc_quote : public Quote { /* ... */ };
        // ...
    }
}
```

上面的代码将命名空间 `cplusplus_primer` 分割为两个嵌套的命名空间，分别是 `QueryLib` 和 `Bookstore`。

嵌套的命名空间同时是一个嵌套的作用域，它嵌套在外层命名空间的作用域中。嵌套的命名空间中的名字遵循的规则与往常类似：内层命名空间声明的名字将隐藏外层命名空间声明的同名成员。在嵌套的命名空间中定义的名字只在内层命名空间中有效，外层命名空间中的代码要想访问它必须在名字前添加限定符。例如，在嵌套的命名空间 `QueryLib` 中声明的类名是

```
cplusplus_primer::QueryLib::Query
```

内联命名空间

C++11 新标准引入了一种新的嵌套命名空间，称为内联命名空间（inline namespace）。和普通的嵌套命名空间不同，内联命名空间中的名字可以被外层命名空间直接使用。也就是说，我们无须在内联命名空间的名字前添加表示该命名空间的前缀，通过外层命名空间的名字就可以直接访问它。

定义内联命名空间的方式是在关键字 `namespace` 前添加关键字 `inline`：

```
inline namespace FifthEd {
    // 该命名空间表示本书第 5 版的代码
}
namespace FifthEd {           // 隐式内联
    class Query_base { /* ... */ };
    // 其他与 Query 有关的声明
}
```

790

C++
11

关键字 `inline` 必须出现在命名空间第一次定义的地方，后续再打开命名空间的时候可以写 `inline`，也可以不写。

当应用程序的代码在一次发布和另一次发布之间发生了改变时，常常会用到内联命名空间。例如，我们可以把本书当前版本的所有代码都放在一个内联命名空间中，而之前版本的代码都放在一个非内联命名空间中：

```
namespace FourthEd {
    class Item_base { /* ... */;
    class Query_base { /* ... */;
    // 本书第4版用到的其他代码
}
```

命名空间 `cplusplus_primer` 将同时使用这两个命名空间。例如，假定每个命名空间都定义在同名的头文件中，则我们可以把命名空间 `cplusplus primer` 定义成如下形式：

```
namespace cplusplus_primer {  
#include "FifthEd.h"  
#include "FourthEd.h"  
}
```

因为 `FifthEd` 是内联的，所以形如 `cplusplus_primer::` 的代码可以直接获得 `FifthEd` 的成员。如果我们想使用早期版本的代码，则必须像其他嵌套的命名空间一样加上完整的外层命名空间名字，比如 `cplusplus_primer::FourthEd::Query_base`。

未命名的命名空间

未命名的命名空间 (unnamed namespace) 是指关键字 `namespace` 后紧跟花括号括起来的一系列声明语句。未命名的命名空间中定义的变量拥有静态生命周期：它们在第一次使用前创建，并且直到程序结束才销毁。

一个未命名的命名空间可以在某个给定的文件内不连续，但是不能跨越多个文件。每个文件定义自己的未命名的命名空间，如果两个文件都含有未命名的命名空间，则这两个空间互相关联。在这两个未命名的命名空间中可以定义相同的名字，并且这些定义表示的是不同实体。如果一个头文件定义了未命名的命名空间，则该命名空间中定义的名字将在每个包含了该头文件的文件中对应不同实体。



和其他命名空间不同，未命名的命名空间仅在特定的文件内部有效，其作用范围不会横跨多个不同的文件。

定义在未命名的命名空间中的名字可以直接使用，毕竟我们找不到什么命名空间的名字来限定它们；同样的，我们也不能对未命名的命名空间的成员使用作用域运算符。

未命名的命名空间中定义的名字的作用域与该命名空间所在的作用域相同。如果未命名的命名空间定义在文件的最外层作用域中，则该命名空间中的名字一定要与全局作用域中的名字有所区别：

```
int i; // i 的全局声明
namespace {
    int i;
}
```

```
// 二义性: i 的定义既出现在全局作用域中, 又出现在未嵌套的未命名的命名空间中
i = 10;
```

其他情况下, 未命名的命名空间中的成员都属于正确的程序实体。和所有命名空间类似, 一个未命名的命名空间也能嵌套在其他命名空间当中。此时, 未命名的命名空间中的成员可以通过外层命名空间的名字来访问:

```
namespace local {
    namespace {
        int i;
    }
}
// 正确: 定义在嵌套的未命名的命名空间中的 i 与全局作用域中的 i 不同
local::i = 42;
```

未命名的命名空间取代文件中的静态声明

< 792

在标准 C++ 引入命名空间的概念之前, 程序需要将名字声明成 `static` 的以使得其对于整个文件有效。在文件中进行静态声明的做法是从 C 语言继承而来的。在 C 语言中, 声明为 `static` 的全局实体在其所在的文件外不可见。

 在文件中进行静态声明的做法已经被 C++ 标准取消了, 现在的做法是使用未命名的命名空间。

18.2.1 节练习

练习 18.12: 将你为之前各章练习编写的程序放置在各自的命名空间中。也就是说, 命名空间 `chapter15` 包含 `Query` 程序的代码, 命名空间 `chapter10` 包含 `TextQuery` 的代码; 使用这种结构重新编译 `Query` 代码示例。

练习 18.13: 什么时候应该使用未命名的命名空间?

练习 18.14: 假设下面的 `operator*` 声明的是嵌套的命名空间 `mathLib::MatrixLib` 的一个成员:

```
namespace mathLib {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
```

请问你应该如何在全局作用域中声明该运算符?

18.2.2 使用命名空间成员

像 `namespace_name::member_name` 这样使用命名空间的成员显然非常烦琐, 特别是当命名空间的名字很长时尤其如此。幸运的是, 我们可以通过一些其他更简便的方法使用命名空间的成员。之前的程序已经使用过其中一种方法, 即 `using` 声明 (参见 3.1 节, 第 74 页)。本节还将介绍另外几种方法, 如命名空间的别名以及 `using` 指示等。

命名空间的别名

命名空间的别名（namespace alias）使得我们可以为命名空间的名字设定一个短得多的同义词。例如，一个很长的命名空间的名字形如

```
namespace cplusplus_primer { /* ... */ };
```

我们可以为其设定一个短得多的同义词：

```
namespace primer = cplusplus_primer;
```

793 命名空间的别名声明以关键字 namespace 开始，后面是别名所用的名字、= 符号、命名空间原来的名字以及一个分号。不能在命名空间还没有定义前就声明别名，否则将产生错误。

命名空间的别名也可以指向一个嵌套的命名空间：

```
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query q;
```



一个命名空间可以有好几个同义词或别名，所有别名都与命名空间原来的名字等价。

using 声明：扼要概述

一条 **using** 声明（using declaration）语句一次只引入命名空间的一个成员。它使得我们可以清楚地知道程序中所用的到底是哪个名字。

using 声明引入的名字遵守与过去一样的作用域规则：它的有效范围从 **using** 声明的地方开始，一直到 **using** 声明所在的作用域结束为止。在此过程中，外层作用域的同名实体将被隐藏。未加限定的名字只能在 **using** 声明所在的作用域以及其内层作用域中使用。在有效作用域结束后，我们就必须使用完整的经过限定的名字了。

一条 **using** 声明语句可以出现在全局作用域、局部作用域、命名空间作用域以及类的作用域中。在类的作用域中，这样的声明语句只能指向基类成员（参见 15.5 节，第 546 页）。

using 指示

using 指示（using directive）和 **using** 声明类似的地方是，我们可以使用命名空间名字的简写形式；和 **using** 声明不同的地方是，我们无法控制哪些名字是可见的，因为所有名字都是可见的。

using 指示以关键字 **using** 开始，后面是关键字 **namespace** 以及命名空间的名字。如果这里所用的名字不是一个已经定义好的命名空间的名字，则程序将发生错误。**using** 指示可以出现在全局作用域、局部作用域和命名空间作用域中，但是不能出现在类的作用域中。

using 指示使得某个特定的命名空间中所有的名字都可见，这样我们就无须再为它们添加任何前缀限定符了。简写的名字从 **using** 指示开始，一直到 **using** 指示所在的作用域结束都能使用。



如果我们提供了一个对 **std** 等命名空间的 **using** 指示而未做任何特殊控制的话，将重新引入由于使用了多个库而造成的名字冲突问题。

using 指示与作用域

using 指示引入的名字的作用域远比 using 声明引入的名字的作用域复杂。如我们所知，using 声明的名字的作用域与 using 声明语句本身的作用域一致，从效果上看就好像 using 声明语句为命名空间的成员在当前作用域内创建了一个别名一样。

using 指示所做的绝非声明别名这么简单。相反，它具有将命名空间成员提升到包含命名空间本身和 using 指示的最近作用域的能力。794

using 声明和 using 指示在作用域上的区别直接决定了它们工作方式的不同。对于 using 声明来说，我们只是简单地令名字在局部作用域内有效。相反，using 指示是令整个命名空间的所有内容变得有效。通常情况下，命名空间中会含有一些不能出现在局部作用域中的定义，因此，using 指示一般被看作是出现在最近的外层作用域中。

在最简单的情况下，假定我们有一个命名空间 A 和一个函数 f，它们都定义在全局作用域中。如果 f 含有一个对 A 的 using 指示，则在 f 看来，A 中的名字仿佛是出现在全局作用域中 f 之前的位置一样：

```
// 命名空间 A 和函数 f 定义在全局作用域中
namespace A {
    int i, j;
}
void f()
{
    using namespace A;           // 把 A 中的名字注入到全局作用域中
    cout << i * j << endl;      // 使用命名空间 A 中的 i 和 j
    // ...
}
```

using 指示示例

让我们看一个简单的示例：

```
namespace blip {
    int i = 16, j = 15, k = 23;
    // 其他声明
}
int j = 0;           // 正确：blip 的 j 隐藏在命名空间中
void manip()
{
    // using 指示，blip 中的名字被“添加”到全局作用域中
    using namespace blip; // 如果使用了 j，则将在::j 和 blip::j 之间产生冲突
    ++i;                // 将 blip::i 设定为 17
    ++j;                // 二义性错误：是全局的 j 还是 blip::j？
    +++:j;              // 正确：将全局的 j 设定为 1
    ++blip::j;           // 正确：将 blip::j 设定为 16
    int k = 97;           // 当前局部的 k 隐藏了 blip::k
    ++k;                // 将当前局部的 k 设定为 98
}
```

manip 的 using 指示使得程序可以直接访问 blip 的所有名字，也就是说，manip 的代码可以使用 blip 中名字的简写形式。

blip 的成员看起来好像是定义在 blip 和 manip 所在的作用域一样。假定 manip 795

定义在全局作用域中，则 blip 的成员也好像是定义在全局作用域中一样。

当命名空间被注入到它的外层作用域之后，很有可能该命名空间中定义的名字会与其外层作用域中的成员冲突。例如在 manip 中，blip 的成员 j 就与全局作用域中的 j 产生了冲突。这种冲突是允许存在的，但是要想使用冲突的名字，我们就必须明确指出名字的版本。manip 中所有未加限定的 j 都会产生二义性错误。

为了使用像 j 这样的名字，我们必须使用作用域运算符来明确指出所需的版本。我们使用 ::j 来表示定义在全局作用域中的 j，而使用 blip::j 来表示定义在 blip 中的 j。

因为 manip 的作用域和命名空间的作用域不同，所以 manip 内部的声明可以隐藏命名空间中的某些成员名字。例如，局部变量 k 隐藏了命名空间的成员 blip::k。在 manip 内使用 k 不存在二义性，它指的就是局部变量 k。

头文件与 using 声明或指示

头文件如果在其顶层作用域中含有 using 指示或 using 声明，则会将名字注入到所有包含了该头文件的文件中。通常情况下，头文件应该只负责定义接口部分的名字，而不定义实现部分的名字。因此，头文件最多只能在它的函数或命名空间内使用 using 指示或 using 声明（参见 3.1 节，第 75 页）。

提示：避免 using 指示

using 指示一次性注入某个命名空间的所有名字，这种用法看似简单实则充满了风险：只使用一条语句就突然将命名空间中所有成员的名字变得可见了。如果应用程序使用了多个不同的库，而这些库中的名字通过 using 指示变得可见，则全局命名空间污染的问题将重新出现。

而且，当引入库的新版本后，正在工作的程序很可能会编译失败。如果新版本引入了一个与应用程序正在使用的名字冲突的名字，就会出现这个问题。

另一个风险是由 using 指示引发的二义性错误只有在使用了冲突名字的地方才能被发现。这种延后的检测意味着可能在特定库引入很久之后才爆发冲突。直到程序开始使用该库的新部分后，之前一直未被检测到的错误才会出现。

相比于使用 using 指示，在程序中对命名空间的每个成员分别使用 using 声明效果更好，这么做可以减少注入到命名空间中的名字数量。using 声明引起的二义性问题在声明处就能发现，无须等到使用名字的地方，这显然对检测并修改错误大有益处。



using 指示也并非一无是处，例如在命名空间本身的实现文件中就可以使用 using 指示。

18.2.2 节练习

练习 18.15：说明 using 指示与 using 声明的区别。

练习 18.16：假定在下面的代码中标记为“位置 1”的地方是对于命名空间 Exercise 中所有成员的 using 声明，请解释代码的含义。如果这些 using 声明出现在“位置 2”又会怎样呢？将 using 声明变为 using 指示，重新回答之前的问题。

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
```

```

        const int limit = 1000;
    }
int ivar = 0;
// 位置 1
void manip() {
    // 位置 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    ++::ivar;
}

```

练习 18.17：实际编写代码检验你对上一题的回答是否正确。

18.2.3 类、命名空间与作用域

对命名空间内部名字的查找遵循常规的查找规则：即由内向外依次查找每个外层作用域。外层作用域也可能是一个或多个嵌套的命名空间，直到最外层的全局命名空间查找过程终止。只有位于开放的块中且在使用点之前声明的名字才被考虑：

```

namespace A {
    int i;
    namespace B {
        int i;                      // 在 B 中隐藏了 A::i
        int j;
        int f1()
        {
            int j;                // j 是 f1 的局部变量，隐藏了 A::B::j
            return i;              // 返回 B::i
        }
    } // 命名空间 B 结束，此后 B 中定义的名字不再可见
    int f2() {
        return j;                // 错误：j 没有被定义
    }
    int j = i;                  // 用 A::i 进行初始化
}

```

对于位于命名空间中的类来说，常规的查找规则仍然适用：当成员函数使用某个名字时，首先在该成员中进行查找，然后在类中查找（包括基类），接着在外层作用域中查找，这时一个或几个外层作用域可能就是命名空间：

```

namespace A {
    int i;
    int k;
    class C1 {
public:
    C1(): i(0), j(0) { }      // 正确：初始化 C1::i 和 C1::j
    int f1() { return k; }     // 返回 A::k
    int f2() { return h; }     // 错误：h 未定义
    int f3();
private:
    int i;                    // 在 C1 中隐藏了 A::i
}

```

```

        int j;
    };
    int h = i; // 用 A::i 进行初始化
}
// 成员 f3 定义在 C1 和命名空间 A 的外部
int A::C1::f3() { return h; } // 正确：返回 A::h

```

除了类内部出现的成员函数定义之外（参见 7.4.1 节，第 254 页），总是向上查找作用域。名字必须先声明后使用，因此 f2 的 return 语句无法通过编译。该语句试图使用命名空间 A 的名字 h，但此时 h 尚未定义。如果 h 在 A 中定义的位置位于 C1 的定义之前，则上述语句将合法。类似的，因为 f3 的定义位于 A::h 之后，所以 f3 对于 h 的使用是合法的。



可以从函数的限定名推断出查找名字时检查作用域的次序，限定名以相反次序指出被查找的作用域。

限定符 A::C1::f3 指出了查找类作用域和命名空间作用域的相反次序。首先查找函数 f3 的作用域，然后查找外层类 C1 的作用域，最后检查命名空间 A 的作用域以及包含着 f3 定义的作用域。



实参相关的查找与类类型形参

考虑下面这个简单的程序：

```
std::string s;
std::cin >> s;
```

如我们所知，该调用等价于（参见 14.1 节，第 491 页）：

```
operator>>(std::cin, s);
```

798 operator>> 函数定义在标准库 string 中，string 又定义在命名空间 std 中。但是我们不用 std:: 限定符和 using 声明就可以调用 operator>>。

对于命名空间中名字的隐藏规则来说有一个重要的例外，它使得我们可以直接访问输出运算符。这个例外是，当我们给函数传递一个类类型的对象时，除了在常规的作用域查找外还会查找实参类所属的命名空间。这一例外对于传递类的引用或指针的调用同样有效。

在此例中，当编译器发现对 operator>> 的调用时，首先在当前作用域中寻找合适的函数，接着查找输出语句的外层作用域。随后，因为>>表达式的形参是类类型的，所以编译器还会查找 cin 和 s 的类所属的命名空间。也就是说，对于这个调用来说，编译器会查找定义了 istream 和 string 的命名空间 std。当在 std 中查找时，编译器找到了 string 的输出运算符函数。

查找规则的这个例外允许概念上作为类接口一部分的非成员函数无须单独的 using 声明就能被程序使用。假如该例外不存在，则我们将不得不为输出运算符专门提供一个 using 声明：

```
using std::operator>>; // 要想使用 cin >> s 就必须有该 using 声明
```

或者使用函数调用的形式以把命名空间的信息包含进来：

```
std::operator>>(std::cin, s); // 正确：显式地使用 std::>>
```

在没有使用运算符语法的情况下，上述两种声明都显得比较笨拙且无形中增加了使用 IO 标准库的难度。

查找与 std::move 和 std::forward

很多甚至是绝大多数 C++ 程序员从来都没有考虑过与实参相关的查找问题。通常情况下，如果在应用程序中定义了一个标准库中已有的名字，则将出现以下两种情况中的一种：要么根据一般的重载规则确定某次调用应该执行函数的那个版本；要么应用程序根本就不会执行函数的标准库版本。

接下来考虑标准库 move 和 forward 函数。这两个都是模板函数，在标准库的定义中它们都接受一个右值引用的函数形参。如我们所知，在函数模板中，右值引用形参可以匹配任何类型（参见 16.2.6 节，第 611 页）。如果我们的应用程序也定义了一个接受单一形参的 move 函数，则不管该形参是什么类型，应用程序的 move 函数都将与标准库的版本冲突。forward 函数也是如此。

因此，move（以及 forward）的名字冲突要比其他标准库函数的冲突频繁得多。而且，因为 move 和 forward 执行的是非常特殊的类型操作，所以应用程序专门修改函数原有行为的概率非常小。

对于 move 和 forward 来说，冲突很多但是大多数是无意的，这一特点解释了为什么我们建议最好使用它们的带限定语的完整版本的原因（参见 12.1.5 节，第 417 页）。通过书写 std::move 而非 move，我们就能明确地知道想要使用的是函数的标准库版本。

友元声明与实参相关的查找

799



回顾我们曾经讨论过的，当类声明了一个友元时，该友元声明并没有使得友元本身可见（参见 7.2.1 节，第 242 页）。然而，一个另外的未声明的类或函数如果第一次出现在友元声明中，则我们认为它是最近的外层命名空间的成员。这条规则与实参相关的查找规则结合在一起将产生意想不到的效果：

```
namespace A {
    class C {
        // 两个友元，在友元声明之外没有其他的声明
        // 这些函数隐式地成为命名空间 A 的成员
        friend void f2();           // 除非另有声明，否则不会被找到
        friend void f(const C&);   // 根据实参相关的查找规则可以被找到
    };
}
```

此时，f 和 f2 都是命名空间 A 的成员。即使 f 不存在其他声明，我们也能通过实参相关的查找规则调用 f：

```
int main()
{
    A::C cobj;
    f(cobj);           // 正确：通过在 A::C 中的友元声明找到 A::f
    f2();             // 错误：A::f2 没有被声明
}
```

因为 f 接受一个类类型的实参，而且 f 在 C 所属的命名空间进行了隐式的声明，所以 f 能被找到。相反，因为 f2 没有形参，所以它无法被找到。

18.2.3 节练习

练习 18.18: 已知有下面的 swap 的典型定义（参见 13.3 节，第 457 页），当 mem1 是一个 string 时程序使用 swap 的哪个版本？如果 mem1 是 int 呢？说明在这两种情况下名字查找的过程。

```
void swap(T v1, T v2)
{
    using std::swap;
    swap(v1.mem1, v2.mem1);
    // 交换类型 T 的其他成员
}
```

练习 18.19: 如果对 swap 的调用形如 std::swap(v1.mem1, v2.mem1) 将发生什么情况？

800> 18.2.4 重载与命名空间

命名空间对函数的匹配过程有两方面的影响（参见 6.4 节，第 209 页）。其中一个影响非常明显：using 声明或 using 指示能将某些函数添加到候选函数集。另外一个影响则比较微妙。



与实参相关的查找与重载

在上一节中我们了解到，对于接受类类型实参的函数来说，其名字查找将在实参类所属的命名空间中进行。这条规则对于我们如何确定候选函数集同样也有影响。我们将在每个实参类（以及实参类的基类）所属的命名空间中搜寻候选函数。在这些命名空间中所有与被调用函数同名的函数都将被添加到候选集当中，即使其中某些函数在调用语句处不可见也是如此：

```
namespace NS {
    class Quote { /* ... */ };
    void display(const Quote&) { /* ... */ }
}
// Bulk_item 的基类声明在命名空间 NS 中
class Bulk_item : public NS::Quote { /* ... */ };
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

我们传递给 display 的实参属于类类型 Bulk_item，因此该调用语句的候选函数不仅应该在调用语句所在的作用域中查找，而且也应该在 Bulk_item 及其基类 Quote 所属的命名空间 NS 中声明的函数 display(const Quote&) 也将被添加到候选函数集中。

重载与 using 声明

要想理解 using 声明与重载之间的交互关系，必须首先明确一条：using 声明语句声明的是一个名字，而非一个特定的函数（参见 15.6 节，第 551 页）：

```
using NS::print(int);      // 错误：不能指定形参列表
using NS::print;           // 正确：using 声明只声明一个名字
```

当我们为函数书写 using 声明时，该函数的所有版本都被引入到当前作用域中。

一个 using 声明囊括了重载函数的所有版本以确保不违反命名空间的接口。库的作者为某项任务提供了好几个不同的函数，允许用户选择性地忽略重载函数中的一部分但不是全部有可能导致意想不到的程序行为。

一个 using 声明引入的函数将重载该声明语句所属作用域中已有的其他同名函数。801如果 using 声明出现在局部作用域中，则引入的名字将隐藏外层作用域的相关声明。如果 using 声明所在的作用域中已经有一个函数与新引入的函数同名且形参列表相同，则该 using 声明将引发错误。除此之外，using 声明将为引入的名字添加额外的重载实例，并最终扩充候选函数集的规模。

重载与 using 指示

using 指示将命名空间的成员提升到外层作用域中，如果命名空间的某个函数与该命名空间所属作用域的函数同名，则命名空间的函数将被添加到重载集合中：

```
namespace libs_R_us {  
    extern void print(int);  
    extern void print(double);  
}  
// 普通的声明  
void print(const std::string &);  
// 这个using 指示把名字添加到print调用的候选函数集  
using namespace libs_R_us;  
// print调用此时的候选函数包括：  
// libs_R_us 的print(int)  
// libs_R_us 的print(double)  
// 显式声明的print(const std::string &)  
void fooBar(int ival)  
{  
    print("Value: ");           // 调用全局函数print(const string &)  
    print(ival);                // 调用libs_R_us::print(int)  
}
```

与 using 声明不同的是，对于 using 指示来说，引入一个与已有函数形参列表完全相同的函数并不会产生错误。此时，只要我们指明调用的是命名空间中的函数版本还是当前作用域的版本即可。

跨越多个 using 指示的重载

如果存在多个 using 指示，则来自每个命名空间的名字都会成为候选函数集的一部分：

```
namespace AW {  
    int print(int);  
}  
namespace Primer {  
    double print(double);  
}  
// using 指示从不同的命名空间中创建了一个重载函数集合802  
using namespace AW;  
using namespace Primer;  
long double print(long double);  
int main() {
```

```

    print(1);           // 调用 AW::print(int)
    print(3.1);         // 调用 Primer::print(double)
    return 0;
}

```

在全局作用域中，函数 `print` 的重载集合包括 `print(int)`、`print(double)` 和 `print(long double)`，尽管它们的声明位于不同作用域中，但它们都属于 `main` 函数中 `print` 调用的候选函数集。

18.2.4 节练习

练习 18.20：在下面的代码中，确定哪个函数与 `compute` 调用匹配。列出所有候选函数和可行函数，对于每个可行函数的实参与形参的匹配过程来说，发生了哪种类型转换？

```

namespace primerLib {
    void compute();
    void compute(const void *);
}

using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);
void f()
{
    . compute(0);
}

```

如果将 `using` 声明置于 `main` 函数中 `compute` 的调用点之前将发生什么情况？重新回答之前的那些问题。

18.3 多重继承与虚继承

多重继承（multiple inheritance）是指从多个直接基类（参见 15.2.2 节，第 533 页）中产生派生类的能力。多重继承的派生类继承了所有父类的属性。尽管概念上非常简单，但是多个基类相互交织产生的细节可能会带来错综复杂的设计问题与实现问题。

为了探讨有关多重继承的问题，我们将以动物园中动物的层次关系作为教学实例。动物园中的动物存在于不同的抽象级别上。有个体的动物，如 Ling-Ling、Mowgli 和 Balou 等，它们以名字进行区分；每个动物属于一个物种，例如 Ling-Ling 是一只大熊猫；物种又是科的成员，大熊猫是熊科的成员；每个科是动物界的成员，在这个例子中动物界是指一个动物园中所有动物的总和。

我们将定义一个抽象类 `ZooAnimal`，用它来保存动物园中动物共有的信息并提供公共接口。类 `Bear` 将存放 `Bear` 科特有的信息，以此类推。

除了类 `ZooAnimal` 之外，我们的应用程序还包含其他一些辅助类，这些类负责封装不同的抽象，如濒临灭绝的动物。以类 `Panda` 的实现为例，`Panda` 是由 `Bear` 和 `Endangered` 共同派生而来的。

18.3.1 多重继承

在派生类的派生列表中可以包含多个基类:

```
class Bear : public ZooAnimal {
class Panda : public Bear, public Endangered { /* ... */ };
```

每个基类包含一个可选的访问说明符 (参见 15.5 节, 第 543 页)。一如往常, 如果访问说明符被忽略掉了, 则关键字 `class` 对应的默认访问说明符是 `private`, 关键字 `struct` 对应的是 `public` (参见 15.5 节, 第 546 页)。

和只有一个基类的继承一样, 多重继承的派生列表也只能包含已经被定义过的类, 而且这些类不能是 `final` 的 (参见 15.2.2 节, 第 533 页)。对于派生类能够继承的基类个数, C++ 没有进行特殊规定; 但是在某个给定的派生列表中, 同一个基类只能出现一次。

多重继承的派生类从每个基类中继承状态

在多重继承关系中, 派生类的对象包含有每个基类的子对象 (参见 15.2.2 节, 第 530 页)。如图 18.2 所示, 在 `Panda` 对象中含有一个 `Bear` 部分 (其中又含有一个 `ZooAnimal` 部分)、一个 `Endangered` 部分以及在 `Panda` 中声明的非静态数据成员。



图 18.2: Panda 对象的概念结构

派生类构造函数初始化所有基类

804

构造一个派生类的对象将同时构造并初始化它的所有基类子对象。与从一个基类进行的派生一样 (参见 15.2.2 节, 第 531 页), 多重继承的派生类的构造函数初始值也只能初始化它的直接基类:

```
// 显式地初始化所有基类
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) {}

// 隐式地使用 Bear 的默认构造函数初始化 Bear 子对象
Panda::Panda()
    : Endangered(Endangered::critical) {}
```

派生类的构造函数初始值列表将实参分别传递给每个直接基类。其中基类的构造顺序与派生列表中基类的出现顺序保持一致, 而与派生类构造函数初始值列表中基类的顺序无关。一个 `Panda` 对象按照如下次序进行初始化:

- `ZooAnimal` 是整个继承体系的最终基类, `Bear` 是 `Panda` 的直接基类, `ZooAnimal` 是 `Bear` 的基类, 所以首先初始化 `ZooAnimal`。
- 接下来初始化 `Panda` 的第一个直接基类 `Bear`。

- 然后初始化 Panda 的第二个直接基类 Endangered。
- 最后初始化 Panda。

继承的构造函数与多重继承

C++
11

在 C++11 新标准中，允许派生类从它的一个或几个基类中继承构造函数（参见 15.7.4 节，第 557 页）。但是如果从多个基类中继承了相同的构造函数（即形参列表完全相同），则程序将产生错误：

```
struct Basel {
    Basel() = default;
    Basel(const std::string&);
    Basel(std::shared_ptr<int>);

};

struct Base2 {
    Base2() = default;
    Base2(const std::string&);
    Base2(int);

};

// 错误：D1 试图从两个基类中都继承 D1::D1(const string&)
struct D1: public Basel, public Base2 {
    using Basel::Basel;           // 从 Basel 继承构造函数
    using Base2::Base2;           // 从 Base2 继承构造函数
};
```

如果一个类从它的多个基类中继承了相同的构造函数，则这个类必须为该构造函数定义它自己的版本：

805

```
struct D2: public Basel, public Base2 {
    using Basel::Basel;           // 从 Basel 继承构造函数
    using Base2::Base2;           // 从 Base2 继承构造函数
    // D2 必须自定义一个接受 string 的构造函数
    D2(const string &s): Basel(s), Base2(s) { }
    D2() = default;              // 一旦 D2 定义了它自己的构造函数，则必须出现
};
```

析构函数与多重继承

和往常一样，派生类的析构函数只负责清除派生类本身分配的资源，派生类的成员及基类都是自动销毁的。合成的析构函数体为空。

析构函数的调用顺序正好与构造函数相反，在我们的例子中，析构函数的调用顺序是 ~Panda、~Endangered、~Bear 和 ~ZooAnimal。

多重继承的派生类的拷贝与移动操作

与只有一个基类的继承一样，多重继承的派生类如果定义了自己的拷贝/赋值构造函数和赋值运算符，则必须在完整的对象上执行拷贝、移动或赋值操作（参见 15.7.2 节，第 553 页）。只有当派生类使用的是合成版本的拷贝、移动或赋值成员时，才会自动对其基类部分执行这些操作。在合成的拷贝控制成员中，每个基类分别使用自己的对应成员隐式地完成构造、赋值或销毁等工作。

例如，假设 Panda 使用了合成版本的成员 ling_ling 的初始化过程：

```
Panda ying_yang("ying_yang");
```

```
Panda ling_ling = ying_yang;           // 使用拷贝构造函数
```

将调用 Bear 的拷贝构造函数，后者又在执行自己的拷贝任务之前先调用 ZooAnimal 的拷贝构造函数。一旦 ling_ling 的 Bear 部分构造完成，接着就会调用 Endangered 的拷贝构造函数来创建对象相应的部分。最后，执行 Panda 的拷贝构造函数。合成的移动构造函数的工作机理与之类似。

合成的拷贝赋值运算符的行为与拷贝构造函数很相似。它首先赋值 Bear 部分（并且通过 Bear 赋值 ZooAnimal 部分），然后赋值 Endangered 部分，最后是 Panda 部分。移动赋值运算符的工作机理与之类似。

18.3.1 节练习

练习 18.21：解释下列声明的含义，在它们当中存在错误吗？如果有，请指出来并说明错误的原因。

- (a) class CADVehicle : public CAD, Vehicle { ... };
- (b) class DblList: public List, public List { ... };
- (c) class iostream: public istream, public ostream { ... };

练习 18.22：已知存在如下所示的类的继承体系，其中每个类都定义了一个默认构造函数：

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

对于下面的定义来说，构造函数的执行顺序是怎样的？

```
MI mi;
```

18.3.2 类型转换与多个基类

在只有一个基类的情况下，派生类的指针或引用能自动转换成一个可访问基类的指针或引用（参见 15.2.2 节，第 530 页；参见 15.5 节，第 544 页）。多个基类的情况与之类似。我们可以令某个可访问基类的指针或引用直接指向一个派生类对象。例如，一个 ZooAnimal、Bear 或 Endangered 类型的指针或引用可以绑定到 Panda 对象上：

```
// 接受 Panda 的基类引用的一系列操作
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang("ying_yang");
print(ying_yang);           // 把一个 Panda 对象传递给一个 Bear 的引用
highlight(ying_yang);       // 把一个 Panda 对象传递给一个 Endangered 的引用
cout << ying_yang << endl; // 把一个 Panda 对象传递给一个 ZooAnimal 的引用
```

编译器不会在派生类向基类的几种转换中进行比较和选择，因为在它看来转换到任意一种基类都一样好。例如，如果存在如下所示的 print 重载形式：

```
void print(const Bear&);
void print(const Endangered&);
```

则通过 Panda 对象对不带前缀限定符的 print 函数进行调用将产生编译错误：

```
Panda ying_yang("ying_yang");
print(ying_yang); // 二义性错误
```

基于指针类型或引用类型的查找

与只有一个基类的继承一样，对象、指针和引用的静态类型决定了我们能够使用哪些成员（参见 15.6 节，第 547 页）。如果我们使用一个 ZooAnimal 指针，则只有定义在 ZooAnimal 中的操作是可以使用的，Panda 接口中的 Bear、Panda 和 Endangered 特有的部分都不可见。类似的，一个 Bear 类型的指针或引用只能访问 Bear 及 ZooAnimal 的成员，一个 Endangered 的指针或引用只能访问 Endangered 的成员。

举个例子，已知我们的类已经定义了表 18.1 列出的虚函数，考虑下面的这些函数调用：

```
Bear *pb = new Panda("ying_yang");
pb->print(); // 正确: Panda::print()
pb->curl(); // 错误: 不属于 Bear 的接口
pb->highlight(); // 错误: 不属于 Bear 的接口
delete pb; // 正确: Panda::~Panda()
```

当我们通过 Endangered 的指针或引用访问一个 Panda 对象时，Panda 接口中 Panda 特有的部分以及属于 Bear 的部分都是不可见的：

```
Endangered *pe = new Panda("ying_yang");
pe->print(); // 正确: Panda::print()
pe->toes(); // 错误: 不属于 Endangered 的接口
pe->curl(); // 错误: 不属于 Endangered 的接口
pe->highlight(); // 正确: Panda::highlight()
delete pe; // 正确: Panda::~Panda()
```

表 18.1：在 ZooAnimal/Endangered 中定义的虚函数

函数	含有自定义版本的类
print	ZooAnimal::ZooAnimal Bear::Bear Endangered::Endangered Panda::Panda
highlight	Endangered::Endangered Panda::Panda
toes	Bear::Bear Panda::Panda
curl	Panda::Panda
析构函数	ZooAnimal::ZooAnimal Endangered::Endangered

18.3.2 节练习

练习 18.23：使用练习 18.22 的继承体系以及下面定义的类 D，同时假定每个类都定义了默认构造函数，请问下面的哪些类型转换是不被允许的？

```

class D : public X, public C { ... };
D *pd = new D;
(a) X *px = pd;          (b) A *pa = pd;
(c) B *pb = pd;          (d) C *pc = pd;

```

练习 18.24: 在第 714 页, 我们使用一个指向 Panda 对象的 Bear 指针进行了一系列调用, 假设我们使用的是一个指向 Panda 对象的 ZooAnimal 指针将发生什么情况, 请对这些调用语句逐一进行说明。

练习 18.25: 假设我们有两个基类 Base1 和 Base2, 它们各自定义了一个名为 print 的虚成员和一个虚析构函数。从这两个基类中我们派生出下面的类, 它们都重新定义了 print 函数:

```

class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };

```

通过下面的指针, 指出在每个调用中分别使用了哪个函数:

```

Base1 *pb1 = new MI;
Base2 *pb2 = new MI;
D1 *pd1 = new MI;
D2 *pd2 = new MI;
(a) pb1->print();      (b) pd1->print();      (c) pd2->print();
(d) delete pb2;         (e) delete pd1;        (f) delete pd2;

```

18.3.3 多重继承下的类作用域

在只有一个基类的情况下, 派生类的作用域嵌套在直接基类和间接基类的作用域中(参见 15.6 节, 第 547 页)。查找过程沿着继承体系自底向上进行, 直到找到所需的名字。派生类的名字将隐藏基类的同名成员。

在多重继承的情况下, 相同的查找过程在所有直接基类中同时进行。如果名字在多个基类中都被找到, 则对该名字的使用将具有二义性。

在我们的例子中, 如果我们通过 Panda 的对象、指针或引用使用了某个名字, 则程序会并行地在 Endangered 和 Bear/ZooAnimal 这两棵子树中查找该名字。如果名字在超过一棵子树中被找到, 则该名字的使用具有二义性。对于一个派生类来说, 从它的几个基类中分别继承名字相同的成员是完全合法的, 只不过在使用这个名字时必须明确指出它的版本。



当一个类拥有多个基类时, 有可能出现派生类从两个或更多基类中继承了同名成员的情况。此时, 不加前缀限定符直接使用该名字将引发二义性。

例如, 如果 ZooAnimal 和 Endangered 都定义了名为 max_weight 的成员, 并且 Panda 没有定义该成员, 则下面的调用是错误的:

```
double d = ying_yang.max_weight();
```

Panda 在派生的过程中拥有了两个名为 max_weight 的成员, 这是完全合法的。派生仅仅是产生了潜在的二义性, 只要 Panda 对象不调用 max_weight 函数就能避免二义性错误。另外, 如果每次调用 max_weight 时都指出所调用的版本

< 808 >

< 809 >

(`ZooAnimal::max_weight` 或者 `Endangered::max_weight`)，也不会发生二义性。只有当要调用哪个函数含糊不清时程序才会出错。

在上面的例子中，派生类继承的两个 `max_weight` 会产生二义性，这一点显而易见。一种更复杂的情况是，有时即使派生类继承的两个函数形参列表不同也可能发生错误。此外，即使 `max_weight` 在一个类中是私有的，而在另一个类中是公有的或受保护的同样也可能发生错误。最后一种情况，假如 `max_weight` 定义在 `Bear` 中而非 `ZooAnimal` 中，上面的程序仍然是错误的。

和往常一样，先查找名字后进行类型检查（参见 6.4.1 节，第 210 页）。当编译器在两个作用域中同时发现了 `max_weight` 时，将直接报告一个调用二义性的错误。

要想避免潜在的二义性，最好的办法是在派生类中为该函数定义一个新版本。例如，我们可以为 `Panda` 定义一个 `max_weight` 函数从而解决二义性问题：

```
double Panda::max_weight() const
{
    return std::max(ZooAnimal::max_weight(),
                    Endangered::max_weight());
}
```

18.3.3 节练习

```
struct Basel {
    void print(int) const; // 默认情况下是公有的
protected:
    int ival;
    double dval;
    char cval;
private:
    int *id;
};

struct Base2 {
    void print(double) const; // 默认情况下是公有的
protected:
    double fval;
private:
    double dval;
};

struct Derived : public Basel {
    void print(std::string) const; // 默认情况下是公有的
protected:
    std::string sval;
    double dval;
};

struct MI : public Derived, public Base2 {
    void print(std::vector<double>); // 默认情况下是公有的
protected:
    int *ival;
    std::vector<double> dvec;
};
```

练习 18.26: 已知如上所示的继承体系，下面对 print 的调用为什么是错误的？适当修改 MI，令其对 print 的调用可以编译通过并正确执行。

```
MI mi;
mi.print(42);
```

练习 18.27: 已知如上所示的继承体系，同时假定为 MI 添加了一个名为 foo 的函数：

```
int ival;
double dval;
void MI::foo(double cval)
{
    int dval;
    // 练习中的问题发生在此处
}
```

- (a) 列出在 MI::foo 中可见的所有名字。
- (b) 是否存在某个可见的名字是继承自多个基类的？
- (c) 将 Base1 的 dval 成员与 Derived 的 dval 成员求和后赋给 dval 的局部实例。
- (d) 将 MI::dvec 的最后一个元素的值赋给 Base2::fval。
- (e) 将从 Base1 继承的 cval 赋给从 Derived 继承的 sval 的第一个字符。

18.3.4 虚继承

< 810

尽管在派生列表中同一个基类只能出现一次，但实际上派生类可以多次继承同一个类。派生类可以通过它的两个直接基类分别继承同一个间接基类，也可以直接继承某个基类，然后通过另一个基类再一次间接继承该类。

举个例子，IO 标准库的 `istream` 和 `ostream` 分别继承了一个共同的名为 `base_ios` 的抽象基类。该抽象基类负责保存流的缓冲内容并管理流的条件状态。`iostream` 是另外一个类，它从 `istream` 和 `ostream` 直接继承而来，可以同时读写流的内容。因为 `istream` 和 `ostream` 都继承自 `base_ios`，所以 `iostream` 继承了 `base_ios` 两次，一次是通过 `istream`，另一次是通过 `ostream`。

在默认情况下，派生类中含有继承链上每个类对应的子部分。如果某个类在派生过程中出现了多次，则派生类中将包含该类的多个子对象。

这种默认的情况对某些形如 `iostream` 的类显然是行不通的。一个 `iostream` 对象肯定希望在同一个缓冲区中进行读写操作，也会要求条件状态能同时反映输入和输出操作的情况。假如在 `iostream` 对象中真的包含了 `base_ios` 的两份拷贝，则上述的共享行为就无法实现了。

< 811

在 C++ 语言中我们通过虚继承（virtual inheritance）的机制解决上述问题。虚继承的目的是令某个类做出声明，承诺愿意共享它的基类。其中，共享的基类子对象称为虚基类（virtual base class）。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。

另一个 Panda 类

在过去，科学界对于大熊猫属于 Raccoon 科还是 Bear 科争论不休。为了如实地反映这种争论，我们可以对 Panda 类进行修改，令其同时继承 Bear 和 Raccoon。此时，为了避免赋予 Panda 两份 ZooAnimal 的子对象，我们将 Bear 和 Raccoon 继承

ZooAnimal 的方式定义为虚继承。图 18.3 描述了新的继承体系。

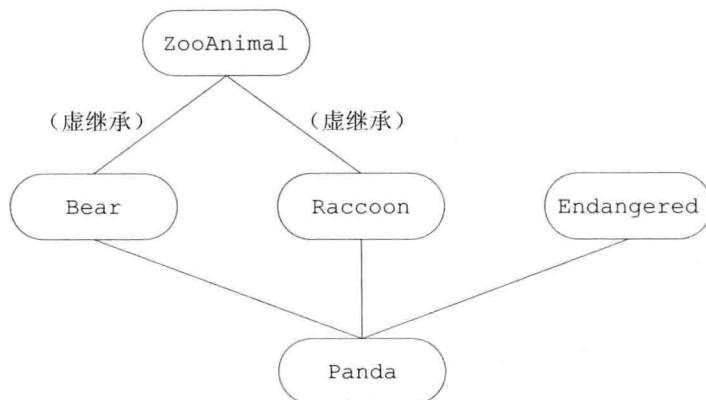


图 18.3: Panda 的虚继承层次

观察这个新的继承体系，我们将发现虚继承的一个不太直观的特征：必须在虚派生的真实需求出现前就已经完成虚派生的操作。例如在我们的类中，当我们定义 Panda 时才出现了对虚派生的需求，但是如果 Bear 和 Raccoon 不是从 ZooAnimal 虚派生得到的，那么 Panda 的设计者就显得不太幸运了。

在实际的编程过程中，位于中间层次的基类将其继承声明为虚继承一般不会带来什么问题。通常情况下，使用虚继承的类层次是由一个人或一个项目组一次性设计完成的。对于一个独立开发的类来说，很少需要基类中的某一个是虚基类，况且新基类的开发者也无法改变已存在的类体系。



虚派生只影响从指定了虚基类的派生类中进一步派生出的类，它不会影响派生类本身。

812

使用虚基类

我们指定虚基类的方式是在派生列表中添加关键字 `virtual`:

```
// 关键字 public 和 virtual 的顺序随意
class Raccoon : public virtual ZooAnimal { /* ... */ };
class Bear : virtual public ZooAnimal { /* ... */ };
```

通过上面的代码我们将 ZooAnimal 定义为 Raccoon 和 Bear 的虚基类。

`virtual` 说明符表明了一种愿望，即在后续的派生类当中共享虚基类的同一份实例。至于什么样的类能够作为虚基类并没有特殊规定。

如果某个类指定了虚基类，则该类的派生仍按常规方式进行:

```
class Panda : public Bear,
              public Raccoon, public Endangered {
};
```

Panda 通过 Raccoon 和 Bear 继承了 ZooAnimal，因为 Raccoon 和 Bear 继承 ZooAnimal 的方式都是虚继承，所以在 Panda 中只有一个 ZooAnimal 基类部分。

支持向基类的常规类型转换

不论基类是不是虚基类，派生类对象都能被可访问基类的指针或引用操作。例如，下面这些从 Panda 向基类的类型转换都是合法的：

```
void dance(const Bear&);
void rummage(const Raccoon&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang;
dance(ying_yang);           // 正确：把一个 Panda 对象当成 Bear 传递
rummage(ying_yang);        // 正确：把一个 Panda 对象当成 Raccoon 传递
cout << ying_yang;         // 正确：把一个 Panda 对象当成 ZooAnimal 传递
```

虚基类成员的可见性

因为在每个共享的虚基类中只有唯一一个共享的子对象，所以该基类的成员可以被直接访问，并且不会产生二义性。此外，如果虚基类的成员只被一条派生路径覆盖，则我们仍然可以直接访问这个被覆盖的成员。但是如果成员被多余一个基类覆盖，则一般情况下派生类必须为该成员自定义一个新的版本。

例如，假定类 B 定义了一个名为 x 的成员，D1 和 D2 都是从 B 虚继承得到的，D 继承了 D1 和 D2，则在 D 的作用域中，x 通过 D 的两个基类都是可见的。如果我们通过 D 的对象使用 x，有三种可能性：

- 如果在 D1 和 D2 中都没有 x 的定义，则 x 将被解析为 B 的成员，此时不存在二义性，一个 D 的对象只含有 x 的一个实例。
- 如果 x 是 B 的成员，同时是 D1 和 D2 中某一个的成员，则同样没有二义性，派生类的 x 比共享虚基类 B 的 x 优先级更高。◀813
- 如果在 D1 和 D2 中都有 x 的定义，则直接访问 x 将产生二义性问题。

与非虚的多重继承体系一样，解决这种二义性问题最好的方法是在派生类中为成员自定义新的实例。

18.3.4 节练习

练习 18.28：已知存在如下的继承体系，在 VMI 类的内部哪些继承而来的成员无须前缀限定符就能直接访问？哪些必须有限定符才能访问？说明你的原因。

```
struct Base {
    void bar(int);           // 默认情况下是公有的
protected:
    int ival;
};

struct Derived1 : virtual public Base {
    void bar(char);          // 默认情况下是公有的
    void foo(char);
protected:
    char cval;
};

struct Derived2 : virtual public Base {
    void foo(int);           // 默认情况下是公有的
protected:
    int ival;
```

```

    char cval;
};

class VMI : public Derived1, public Derived2 { };

```

18.3.5 构造函数与虚继承

在虚派生中，虚基类是由最低层的派生类初始化的。以我们的程序为例，当创建 Panda 对象时，由 Panda 的构造函数独自控制 ZooAnimal 的初始化过程。

为了理解这一规则，我们不妨假设当以普通规则处理初始化任务时会发生什么情况。在此例中，虚基类将会在多条继承路径上被重复初始化。以 ZooAnimal 为例，如果应用普通规则，则 Raccoon 和 Bear 都会试图初始化 Panda 对象的 ZooAnimal 部分。

当然，继承体系中的每个类都可能在某个时刻成为“最低层的派生类”。只要我们能创建虚基类的派生类对象，该派生类的构造函数就必须初始化它的虚基类。例如在我们的继承体系中，当创建一个 Bear（或 Raccoon）的对象时，它已经位于派生的最低层，因此 Bear（或 Raccoon）的构造函数将直接初始化其 ZooAnimal 基类部分：

```

Bear::Bear(std::string name, bool onExhibit):
    ZooAnimal(name, onExhibit, "Bear") { }
Raccoon::Raccoon(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Raccoon") { }

```

而当创建一个 Panda 对象时，Panda 位于派生的最低层并由它负责初始化共享的 ZooAnimal 基类部分。即使 ZooAnimal 不是 Panda 的直接基类，Panda 的构造函数也可以初始化 ZooAnimal：

```

Panda::Panda(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Panda"),
    Bear(name, onExhibit),
    Raccoon(name, onExhibit),
    Endangered(Endangered::critical),
    sleeping_flag(false) { }

```

虚继承的对象的构造方式

含有虚基类的对象的构造顺序与一般的顺序稍有区别：首先使用提供给最低层派生类构造函数的初始值初始化该对象的虚基类子部分，接下来按照直接基类在派生列表中出现的次序依次对其进行初始化。

例如，当我们创建一个 Panda 对象时：

- 首先使用 Panda 的构造函数初始值列表中提供的初始值构造虚基类 ZooAnimal 部分。
- 接下来构造 Bear 部分。
- 然后构造 Raccoon 部分。
- 然后构造第三个直接基类 Endangered。
- 最后构造 Panda 部分。

如果 Panda 没有显式地初始化 ZooAnimal 基类，则 ZooAnimal 的默认构造函数将被调用。如果 ZooAnimal 没有默认构造函数，则代码将发生错误。



虚基类总是先于非虚基类构造，与它们在继承体系中的次序和位置无关。

构造函数与析构函数的次序

一个类可以有多个虚基类。此时，这些虚的子对象按照它们在派生列表中出现的顺序从左向右依次构造。例如，在下面这个稍显杂乱的 TeddyBear 派生关系中有两个虚基类：ToyAnimal 是直接虚基类，ZooAnimal 是 Bear 的虚基类：

```
class Character { /* ... */ };
class BookCharacter : public Character { /* ... */ };
class ToyAnimal { /* ... */ };
class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
{ /* ... */ };
```

编译器按照直接基类的声明顺序对其进行检查，以确定其中是否含有虚基类。如果有，则先构造虚基类，然后按照声明的顺序逐一构造其他非虚基类。因此，要想创建一个 TeddyBear 对象，需要按照如下次序调用这些构造函数：

ZooAnimal();	// Bear 的虚基类
ToyAnimal();	// 直接虚基类
Character();	// 第一个非虚基类的间接基类
BookCharacter();	// 第一个直接非虚基类
Bear();	// 第二个直接非虚基类
TeddyBear();	// 最底层的派生类

合成的拷贝和移动构造函数按照完全相同的顺序执行，合成的赋值运算符中的成员也按照该顺序赋值。和往常一样，对象的销毁顺序与构造顺序正好相反，首先销毁 TeddyBear 部分，最后销毁 ZooAnimal 部分。

18.3.5 节练习

练习 18.29: 已知有如下所示的类继承关系：

```
class Class { ... };
class Base : public Class { ... };
class D1 : virtual public Base { ... };
class D2 : virtual public Base { ... };
class MI : public D1, public D2 { ... };
class Final : public MI, public Class { ... };
```

- (a) 当作用于一个 Final 对象时，构造函数和析构函数的执行次序分别是什么？
- (b) 在一个 Final 对象中有几个 Base 部分？几个 Class 部分？
- (c) 下面的哪些赋值运算将造成编译错误？

```
Base *pb;           Class *pc;           MI *pmi;           D2 *pd2;
(a) pb = new Class;      (b) pc = new Final;
(c) pmi = pb;          (d) pd2 = pmi;
```

练习 18.30: 在 Base 中定义一个默认构造函数、一个拷贝构造函数和一个接受 int 形参的构造函数。在每个派生类中分别定义这三种构造函数，每个构造函数应该使用它的实参初始化其 Base 部分。

816 小结

C++语言可以用于解决各种类型的问题，既有几个小时就可以解决的小问题，也有一个大团队工作数年才能解决的超大规模问题。C++的某些特性特别适合于处理超大规模问题，这些特性包括：异常处理、命名空间以及多重继承或虚继承。

异常处理使得我们可以将程序的错误检测部分与错误处理部分分隔开来。当程序抛出一个异常时，当前正在执行的函数暂时中止，开始查找最邻近的与异常匹配的 `catch` 语句。作为异常处理的一部分，如果查找 `catch` 语句的过程中退出了某些函数，则函数中定义的局部变量也随之销毁。

命名空间是一种管理大规模复杂应用程序的机制，这些应用可能是由多个独立的供应商分别编写的代码组合而成的。一个命名空间是一个作用域，我们可以在其中定义对象、类型、函数、模板以及其他命名空间。标准库定义在名为 `std` 的命名空间中。

从概念上来说，多重继承非常简单：一个派生类可以从多个直接基类继承而来。在派生类对象中既包含派生类部分，也包含与每个基类对应的基类部分。虽然看起来很简单，但实际上多重继承的细节非常复杂。特别是对多个基类的继承可能会引入新的名字冲突，并造成来自于基类部分的名字的二义性问题。

如果一个类是从多个基类直接继承而来的，那么有可能这些基类本身又共享了另一个基类。在这种情况下，中间类可以选择使用虚继承，从而声明愿意与层次中虚继承同一基类的其他类共享虚基类。用这种方法，后代派生类中将只有一个共享虚基类的副本。

术语表

捕获所有异常 (catch-all) 异常声明形如 (...) 的 `catch` 子句。一条捕获所有异常的子句可以捕获任意类型的异常。常用于捕获局部检测的异常，该异常将重新抛出到程序的其他部分并最终解决问题。

catch 子句 (catch clause) 程序中负责处理异常的部分。`catch` 子句包含关键字 `catch`，后面是异常声明以及一个语句块。`catch` 子句的代码负责处理异常声明中定义的异常。

构造函数顺序 (constructor order) 在非虚继承中，基类的构造顺序与其在派生列表中出现的顺序一致。在虚继承中，首先构造虚基类。虚基类的构造顺序与其在派生类的派生列表中出现的顺序一致。只有最低层的派生类才能初始化虚基类。虚基类的初始值如果出现在中间基类中，则这些初始值将被忽略。

异常声明 (exception declaration) `catch` 子句中指定其能够处理的异常类型的部分。异常声明的行为与形参列表类似，其中的唯一一个形参通过异常对象进行初始化。如果异常说明符是非引用类型，则异常对象将被拷贝给 `catch`。

异常处理 (exception handling) 管理运行时异常的语言级支持。代码中一个独立开发的部分可以检测并引发异常，由程序的另一个独立开发的部分处理该异常。也就是说，程序的错误检测部分负责抛出异常，而错误处理部分在 `try` 语句块的 `catch` 子句中处理异常。

异常对象 (exception object) 用于在异常的 `throw` 和 `catch` 之间进行通信的对象。在抛出点创建该对象，该对象是被抛出的表达式的副本。在该异常的最后一段处理代码完成之前异常对象都一直存在。异常对象的类型是被抛出的表达式的静态类型。

文件中的静态声明 (file static) 使用关键字 `static` 声明的仅对当前文件有效的名字。在 C 语言和之前的 C++ 版本中，文件中的静态声明用于声明只能在当前文件中使用的名字。该特性在当前的 C++ 版本中已经被未命名的命名空间替换了。

函数 try 语句块 (function try block) 用于捕获构造函数初始化过程发生的异常。关键字 `try` 出现在表示构造函数初始值列表开始的冒号之前（或者当初始值列表为空时出现在函数体的左侧花括号之前），并以函数体右侧花括号之后的一个或几个 `catch` 子句作为结束。

全局命名空间 (global namespace) 是每个程序的隐式命名空间，用于存放全局定义。

处理代码 (handler) 是“`catch` 子句”的同义词。

内联的命名空间 (inline namespace) 内联命名空间中的名字可以看成是外层命名空间的成员。

多重继承 (multiple inheritance) 有多个直接基类的类。派生类继承所有基类的成员。可以为每个基类分别设定访问说明符。

命名空间 (namespace) 将库或者其他程序集定义的名字放在同一个作用域中的机制。和 C++ 的其他作用域不同，命名空间作用域可以定义成几个部分。我们可以打开并关闭命名空间，然后在程序的另一个地方重新打开并关闭该命名空间。

命名空间的别名 (namespace alias) 为某个给定的命名空间定义同义词的机制：

```
namespace N1 = N;
```

将 `N1` 定义成命名空间 `N` 的另一个名字。命名空间可以含有多个别名，命名空间的原名和别名是等价的。

命名空间污染 (namespace pollution) 当所有类和函数的名字都放置于全局命名空间时将造成命名空间污染。如果来自于多个独立供应商的代码都含有全局名字，则使用这些代码的大型程序很可能会面临命

名空间污染的问题。

noexcept 运算符 (noexcept operator) 该运算符返回一个 `bool` 值，用于表示给定的表达式是否会抛出异常。该表达式不会被求值，运算的结果是一个常量表达式。当提供的表达式不含 `throw` 并且只调用了做出不抛出说明的函数时，结果为 `true`；否则结果为 `false`。

noexcept 说明 (noexcept specification) 表示函数是否会抛出异常的关键字。当 `noexcept` 跟在函数的形参列表之后时，它可以连接一个括号括起来的常量表达式，前提是该表达式可以转换成 `bool` 值。如果忽略了该表达式，或者表达式的值为 `true`，则函数不会抛出异常。如果表达式的值是 `false` 或者函数没有异常声明，则其可能抛出异常。

不抛出说明 (nonthrowing specification) 该异常说明用于承诺某个函数不会抛出异常。如果一个做了不抛出说明的函数实际抛出了异常，将调用 `terminate`。不抛出说明符是不含实参或者含有一个值为 `true` 的实参的 `noexcept`。

引发 (raise) 常常作为抛出的同义词。C++ 程序员认为抛出异常和引发异常基本上是等价的。

重新抛出 (rethrow) 不指定表达式的 `throw`。重新抛出只有在 `catch` 子句内部或者被 `catch` 直接或间接调用了的函数内时才有效。它的效果是将其接受的异常重新抛出。

栈展开 (stack unwinding) 在搜寻 `catch` 时依次退出函数的过程。异常发生前构造的局部对象将在进入相应的 `catch` 前被销毁。

terminate 是一个标准库函数，当异常未被捕获或者在处理异常的过程中发生了另一个异常时，`terminate` 负责结束程序的执行。

throw e 该表达式将中断当前的执行路径，`throw` 语句将控制权传递给最近的能够处

理该异常的 catch 子句。表达式 e 将被拷贝给异常对象。

try 语句块 (try block) 含有关键字 try 以及一个或多个 catch 子句的语句块。如果 try 语句块中的代码引发了一个异常，并且某个 catch 可以匹配该异常，则异常将被这个 catch 处理。否则，异常被传递到 try 语句块之外并继续沿着调用链寻找与之匹配的 catch。

未命名的命名空间 (unnamed namespace) 定义时未指定名字的命名空间。对于定义在未命名的命名空间中的名字，我们可以不用作用域运算符就直接访问它们。每个文件有一个独有的未命名的命名空间，其中的名字在文件外不可见。

using 声明 (using declaration) 是一种将命名空间中的某个名字注入当前作用域的机制：

```
using std::cout;
```

上述语句使得命名空间 std 中的名字 cout 在当前作用域可见。之后，我们就可以直接使用 cout 而无须前缀 std:: 了。

using 指示 (using directive) 是具有如下形式的声明：

```
using NS;
```

上述语句使得命名空间 NS 的所有名字在 using 指示所在的作用域以及 NS 所在的作用域都变得可见。

虚基类 (virtual base class) 在派生列表中使用了关键字 virtual 的基类。在派生类对象中，虚基类部分只有一份，即使该虚基类在继承体系中出现了多次也是如此。对于非虚继承而言，构造函数只能初始化它的直接基类。但是对于虚继承来说，虚基类将被最低层的派生类初始化，因此最低层的派生类应该含有它的所有虚基类的初始值。

虚继承 (virtual inheritance) 是多重继承的一种形式，基类被继承了多次，但是派生类共享该基类的唯一一份副本。

作用域运算符 (:: operator) 用于访问命名空间或类中的名字。

第 19 章

特殊工具与技术

内容

19.1 控制内存分配	726
19.2 运行时类型识别	730
19.3 枚举类型	736
19.4 类成员指针	739
19.5 嵌套类	746
19.6 union：一种节省空间的类	749
19.7 局部类	754
19.8 固有的不可移植的特性	755
小结	762
术语表	762

本书的前三部分讨论了 C++ 语言的基本要素，这些要素绝大多数程序员都会用到。此外，C++ 还定义了一些非常特殊的性质，对于很多程序员来说，他们一般很少会用到本章介绍的内容。

820

C++语言的设计者希望它能处理各种各样的问题。因此，C++的某些特征可能对于一些特殊的应用非常重要，而在另外一些情况下没什么作用。本章将介绍C++语言的几种未被广泛使用的特征。

19.1 控制内存分配

某些应用程序对内存分配有特殊的需求，因此我们无法将标准内存管理机制直接应用于这些程序。它们常常需要自定义内存分配的细节，比如使用关键字 `new` 将对象放置在特定的内存空间中。为了实现这一目的，应用程序需要重载 `new` 运算符和 `delete` 运算符以控制内存分配的过程。

19.1.1 重载 new 和 delete

尽管我们说能够“重载 `new` 和 `delete`”，但是实际上重载这两个运算符与重载其他运算符的过程大不相同。要想真正掌握重载 `new` 和 `delete` 的方法，首先要对 `new` 表达式和 `delete` 表达式的工作机理有更多了解。

当我们使用一条 `new` 表达式时：

```
// new 表达式
string *sp = new string("a value");    // 分配并初始化一个 string 对象
string *arr = new string[10];           // 分配 10 个默认初始化的 string 对象
```

实际执行了三步操作。第一步，`new` 表达式调用一个名为 `operator new`（或者 `operator new[]`）的标准库函数。该函数分配一块足够大的、原始的、未命名的内存空间以便存储特定类型的对象（或者对象的数组）。第二步，编译器运行相应的构造函数以构造这些对象，并为其传入初始值。第三步，对象被分配了空间并构造完成，返回一个指向该对象的指针。

当我们使用一条 `delete` 表达式删除一个动态分配的对象时：

```
delete sp;                           // 销毁 *sp，然后释放 sp 指向的内存空间
delete [] arr;                      // 销毁数组中的元素，然后释放对应的内存空间
```

实际执行了两步操作。第一步，对 `sp` 所指的对象或者 `arr` 所指的数组中的元素执行对应的析构函数。第二步，编译器调用名为 `operator delete`（或者 `operator delete[]`）的标准库函数释放内存空间。

如果应用程序希望控制内存分配的过程，则它们需要定义自己的 `operator new` 函数和 `operator delete` 函数。即使在标准库中已经存在这两个函数的定义，我们仍旧可以定义自己的版本。编译器不会对这种重复的定义提出异议，相反，编译器将使用我们自定义的版本替换标准库定义的版本。

821



当自定义了全局的 `operator new` 函数和 `operator delete` 函数后，我们就担负起了控制动态内存分配的职责。这两个函数必须是正确的：因为它们是程序整个处理过程中至关重要的一部分。

应用程序可以在全局作用域中定义 `operator new` 函数和 `operator delete` 函数，也可以将它们定义为成员函数。当编译器发现一条 `new` 表达式或 `delete` 表达式后，将

在程序中查找可供调用的 `operator` 函数。如果被分配（释放）的对象是类类型，则编译器首先在类及其基类的作用域中查找。此时如果该类含有 `operator new` 成员或 `operator delete` 成员，则相应的表达式将调用这些成员。否则，编译器在全局作用域查找匹配的函数。此时如果编译器找到了用户自定义的版本，则使用该版本执行 `new` 表达式或 `delete` 表达式；如果没找到，则使用标准库定义的版本。

我们可以使用作用域运算符 `::new` 表达式或 `::delete` 表达式忽略定义在类中的函数，直接执行全局作用域中的版本。例如，`::new` 只在全局作用域中查找匹配的 `operator new` 函数，`::delete` 与之类似。

operator new 接口和 operator delete 接口

标准库定义了 `operator new` 函数和 `operator delete` 函数的 8 个重载版本。其中前 4 个版本可能抛出 `bad_alloc` 异常，后 4 个版本则不会抛出异常：

```
// 这些版本可能抛出异常
void *operator new(size_t);                                // 分配一个对象
void *operator new[](size_t);                             // 分配一个数组
void *operator delete(void*) noexcept;                  // 释放一个对象
void *operator delete[](void*) noexcept;                // 释放一个数组

// 这些版本承诺不会抛出异常，参见 12.1.2 节（第 409 页）
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
```

类型 `nothrow_t` 是定义在 `new` 头文件中的一个 `struct`，在这个类型中不包含任何成员。`new` 头文件还定义了一个名为 `nothrow` 的 `const` 对象，用户可以通过这个对象请求 `new` 的非抛出版本（参见 12.1.2 节，第 408 页）。与析构函数类似，`operator delete` 也不允许抛出异常（参见 18.1.1 节，第 685 页）。当我们重载这些运算符时，必须使用 `noexcept` 异常说明符（参见 18.1.4 节，第 690 页）指定其不抛出异常。

应用程序可以自定义上面函数版本中的任意一个，前提是自定义的版本必须位于全局作用域或者类作用域中。当我们将上述运算符函数定义成类的成员时，它们是隐式静态的（参见 7.6 节，第 270 页）。我们无须显式地声明 `static`，当然这么做也不会引发错误。因为 `operator new` 用在对象构造之前而 `operator delete` 用在对象销毁之后，所以这两个成员（`new` 和 `delete`）必须是静态的，而且它们不能操纵类的任何数据成员。

822

对于 `operator new` 函数或者 `operator new[]` 函数来说，它的返回类型必须是 `void*`，第一个形参的类型必须是 `size_t` 且该形参不能含有默认实参。当我们为一个对象分配空间时使用 `operator new`；为一个数组分配空间时使用 `operator new[]`。当编译器调用 `operator new` 时，把存储指定类型对象所需的字节数传给 `size_t` 形参；当调用 `operator new[]` 时，传入函数的则是存储数组中所有元素所需的空间。

如果我们想要自定义 `operator new` 函数，则可以为它提供额外的形参。此时，用到这些自定义函数的 `new` 表达式必须使用 `new` 的定位形式（参见 12.1.2 节，第 409 页）将实参传给新增的形参。尽管在一般情况下我们可以自定义具有任何形参的 `operator new`，但是下面这个函数却无论如何不能被用户重载：

```
void *operator new(size_t, void*);                         // 不允许重新定义这个版本
```

这种形式只供标准库使用，不能被用户重新定义。

对于 operator delete 函数或者 operator delete[] 函数来说，它们的返回类型必须是 void，第一个形参的类型必须是 void*。执行一条 delete 表达式将调用相应的 operator 函数，并用指向待释放内存的指针来初始化 void* 形参。

当我们将 operator delete 或 operator delete[] 定义成类的成员时，该函数可以包含另外一个类型为 size_t 的形参。此时，该形参的初始值是第一个形参所指对象的字节数。size_t 形参可用于删除继承体系中的对象。如果基类有一个虚析构函数（参见 15.7.1 节，第 552 页），则传递给 operator delete 的字节数将因待删除指针所指对象的动态类型不同而有所区别。而且，实际运行的 operator delete 函数版本也由对象的动态类型决定。

术语：new 表达式与 operator new 函数

标准库函数 operator new 和 operator delete 的名字容易让人误解。和其他 operator 函数不同（比如 operator=），这两个函数并没有重载 new 表达式或 delete 表达式。实际上，我们根本无法自定义 new 表达式或 delete 表达式的行为。

一条 new 表达式的执行过程总是先调用 operator new 函数以获取内存空间，然后在得到的内存空间中构造对象。与之相反，一条 delete 表达式的执行过程总是先销毁对象，然后调用 operator delete 函数释放对象所占的空间。

我们提供新的 operator new 函数和 operator delete 函数的目的在于改变内存分配的方式，但是不管怎样，我们都不能改变 new 运算符和 delete 运算符的基本含义。

823 ➤ malloc 函数与 free 函数

当你定义了自己的全局 operator new 和 operator delete 后，这两个函数必须以某种方式执行分配内存与释放内存的操作。也许你的初衷仅仅是使用一个特殊定制的内存分配器，但是这两个函数还应该同时满足某些测试的目的，即检验其分配内存的方式是否与常规方式类似。

为此，我们可以使用名为 **malloc** 和 **free** 的函数，C++ 从 C 语言中继承了这些函数，并将其定义在 cstdlib 头文件中。

malloc 函数接受一个表示待分配字节数的 size_t，返回指向分配空间的指针或者返回 0 以表示分配失败。**free** 函数接受一个 void*，它是 **malloc** 返回的指针的副本，**free** 将相关内存返回给系统。调用 **free(0)** 没有任何意义。

如下所示是编写 operator new 和 operator delete 的一种简单方式，其他版本与之类似：

```
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept { free(mem); }
```

19.1.1 节练习

练习 19.1: 使用 `malloc` 编写你自己的 `operator new(size_t)` 函数, 使用 `free` 编写 `operator delete(void *)` 函数。

练习 19.2: 默认情况下, `allocator` 类使用 `operator new` 获取存储空间, 然后使用 `operator delete` 释放它。利用上一题中的两个函数重新编译并运行你的 `StrVec` 程序 (参见 13.5 节, 第 465 页)。

19.1.2 定位 new 表达式

尽管 `operator new` 函数和 `operator delete` 函数一般用于 `new` 表达式, 然而它们毕竟是标准库的两个普通函数, 因此普通的代码也可以直接调用它们。

在 C++ 的早期版本中, `allocator` 类 (参见 12.2.2 节, 第 427 页) 还不是标准库的一部分。应用程序如果想把内存分配与初始化分离开来的话, 需要调用 `operator new` 和 `operator delete`。这两个函数的行为与 `allocator` 的 `allocate` 成员和 `deallocate` 成员非常类似, 它们负责分配或释放内存空间, 但是不会构造或销毁对象。

与 `allocator` 不同的是, 对于 `operator new` 分配的内存空间来说我们无法使用 `construct` 函数构造对象。相反, 我们应该使用 `new` 的定位 `new` (placement `new`) 形式 (参见 12.1.2 节, 第 409 页) 构造对象。如我们所知, `new` 的这种形式为分配函数提供了额外的信息。我们可以使用定位 `new` 传递一个地址, 此时定位 `new` 的形式如下所示:

```
new (place_address) type  
new (place_address) type (initializers)  
new (place_address) type [size]  
new (place_address) type [size] { braced initializer list }
```

其中 `place_address` 必须是一个指针, 同时在 `initializers` 中提供一个 (可能为空的) 以逗号分隔的初始值列表, 该初始值列表将用于构造新分配的对象。

当仅通过一个地址值调用时, 定位 `new` 使用 `operator new(size_t, void*)` “分配” 它的内存。这是一个我们无法自定义的 `operator new` 版本 (参见 19.1.1 节, 第 727 页)。该函数不分配任何内存, 它只是简单地返回指针实参; 然后由 `new` 表达式负责在指定的地址初始化对象以完成整个工作。事实上, 定位 `new` 允许我们在一个特定的、预先分配的内存地址上构造对象。



当只传入一个指针类型的实参时, 定位 `new` 表达式构造对象但是不分配内存。

尽管在很多时候使用定位 `new` 与 `allocator` 的 `construct` 成员非常相似, 但在它们之间也有一个重要的区别。我们传给 `construct` 的指针必须指向同一个 `allocator` 对象分配的空间, 但是传给定位 `new` 的指针无须指向 `operator new` 分配的内存。实际上如我们将在 19.6 节 (第 753 页) 介绍的, 传给定位 `new` 表达式的指针甚至不需要指向动态内存。

显式的析构函数调用

就像定位 `new` 与使用 `allocate` 类似一样, 对析构函数的显式调用也与使用 `destroy` 很类似。我们既可以通过对象调用析构函数, 也可以通过对象的指针或引用调

用析构函数，这与调用其他成员函数没什么区别：

```
string *sp = new string("a value"); // 分配并初始化一个 string 对象
sp->~string();
```

在这里我们直接调用了一个析构函数。箭头运算符解引用指针 sp 以获得 sp 所指的对象，然后我们调用析构函数，析构函数的形式是波浪线 (~) 加上类型的名字。

和调用 `destroy` 类似，调用析构函数可以清除给定的对象但是不会释放该对象所在的空间。如果需要的话，我们可以重新使用该空间。



调用析构函数会销毁对象，但是不会释放内存。

825 > 19.2 运行时类型识别

运行时类型识别 (run-time type identification, RTTI) 的功能由两个运算符实现：

- `typeid` 运算符，用于返回表达式的类型。
- `dynamic_cast` 运算符，用于将基类的指针或引用安全地转换成派生类的指针或引用。

当我们将这两个运算符用于某种类型的指针或引用，并且该类型含有虚函数时，运算符将使用指针或引用所绑定对象的动态类型（参见 15.2.3 节，第 534 页）。

这两个运算符特别适用于以下情况：我们想使用基类对象的指针或引用执行某个派生类操作并且该操作不是虚函数。一般来说，只要有可能我们应该尽量使用虚函数。当操作被定义成虚函数时，编译器将根据对象的动态类型自动地选择正确的函数版本。

然而，并非任何时候都能定义一个虚函数。假设我们无法使用虚函数，则可以使用一个 RTTI 运算符。另一方面，与虚成员函数相比，使用 RTTI 运算符蕴含着更多潜在的风险：程序员必须清楚地知道转换的目标类型并且必须检查类型转换是否被成功执行。



使用 RTTI 必须要加倍小心。在可能的情况下，最好定义虚函数而非直接接管类型管理的重任。

19.2.1 `dynamic_cast` 运算符

`dynamic_cast` 运算符 (`dynamic_cast` operator) 的使用形式如下所示：

```
dynamic_cast<type*>(e)
dynamic_cast<type&>(e)
dynamic_cast<type&&>(e)
```

其中，`type` 必须是一个类类型，并且通常情况下该类型应该含有虚函数。在第一种形式中，`e` 必须是一个有效的指针（参见 2.3.2 节，第 47 页）；在第二种形式中，`e` 必须是一个左值；在第三种形式中，`e` 不能是左值。

在上面的所有形式中，`e` 的类型必须符合以下三个条件中的任意一个：`e` 的类型是目标 `type` 的公有派生类、`e` 的类型是目标 `type` 的公有基类或者 `e` 的类型就是目标 `type` 的类型。如果符合，则类型转换可以成功。否则，转换失败。如果一条 `dynamic_cast` 语句的转换目标是指针类型并且失败了，则结果为 0。如果转换目标是引用类型并且失败了，

则 `dynamic_cast` 运算符将抛出一个 `bad_cast` 异常。

指针类型的 `dynamic_cast`

举个简单的例子，假定 `Base` 类至少含有一个虚函数，`Derived` 是 `Base` 的公有派生类。如果有一个指向 `Base` 的指针 `bp`，则我们可以在运行时将它转换成指向 `Derived` 的指针，具体代码如下：

```
if (Derived *dp = dynamic_cast<Derived*>(bp))
{
    // 使用 dp 指向的 Derived 对象
} else { // bp 指向一个 Base 对象
    // 使用 bp 指向的 Base 对象
}
```

826

如果 `bp` 指向 `Derived` 对象，则上述的类型转换初始化 `dp` 并令其指向 `bp` 所指的 `Derived` 对象。此时，`if` 语句内部使用 `Derived` 操作的代码是安全的。否则，类型转换的结果为 0，`dp` 为 0 意味着 `if` 语句的条件失败，此时 `else` 子句执行相应的 `Base` 操作。



我们可以对一个空指针执行 `dynamic_cast`，结果是所需类型的空指针。

值得注意的一点是，我们在条件部分定义了 `dp`，这样做的好处是在一个操作中同时完成类型转换和条件检查两项任务。而且，指针 `dp` 在 `if` 语句外部是不可访问的。一旦转换失败，即使后续的代码忘了做相应判断，也不会接触到这个未绑定的指针，从而确保程序是安全的。



在条件部分执行 `dynamic_cast` 操作可以确保类型转换和结果检查在同一表达式中完成。

引用类型的 `dynamic_cast`

引用类型的 `dynamic_cast` 与指针类型的 `dynamic_cast` 在表示错误发生的方式上略有不同。因为不存在所谓的空引用，所以对于引用类型来说无法使用与指针类型完全相同的错误报告策略。当对引用的类型转换失败时，程序抛出一个名为 `std::bad_cast` 的异常，该异常定义在 `typeinfo` 标准库头文件中。

我们可以按照如下的形式改写之前的程序，令其使用引用类型：

```
void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // 使用 b 引用的 Derived 对象
    } catch (bad_cast) {
        // 处理类型转换失败的情况
    }
}
```

19.2.1 节练习

练习 19.3：已知存在如下的类继承体系，其中每个类分别定义了一个公有的默认构造函

数和一个虚析构函数：

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };
```

下面的哪个 `dynamic_cast` 将失败？

- (a) `A *pa = new C;`
`B *pb = dynamic_cast< B*>(pa);`
- (b) `B *pb = new B;`
`C *pc = dynamic_cast< C*>(pb);`
- (c) `A *pa = new D;`
`B *pb = dynamic_cast< B*>(pa);`

练习 19.4： 使用上一个练习定义的类改写下面的代码，将表达式 `*pa` 转换成类型 `C&`：

```
if (C *pc = dynamic_cast< C*>(pa))
}
    // 使用 C 的成员
} else {
    // 使用 A 的成员
}
```

练习 19.5： 在什么情况下你应该使用 `dynamic_cast` 替代虚函数？

19.2.2 typeid 运算符

为 RTTI 提供的第二个运算符是 **typeid** 运算符 (typeid operator)，它允许程序向表达式提问：你的对象是什么类型？

827

`typeid` 表达式的形式是 `typeid(e)`，其中 `e` 可以是任意表达式或类型的名字。`typeid` 操作的结果是一个常量对象的引用，该对象的类型是标准库类型 `type_info` 或者 `type_info` 的公有派生类型。`type_info` 类定义在 `typeinfo` 头文件中，19.2.4 节（第 735 页）将介绍更多关于 `type_info` 的细节。

`typeid` 运算符可以作用于任意类型的表达式。和往常一样，顶层 `const`（参见 2.4.3 节，第 57 页）被忽略，如果表达式是一个引用，则 `typeid` 返回该引用所引对象的类型。不过当 `typeid` 作用于数组或函数时，并不会执行向指针的标准类型转换（参见 4.11.2 节，第 143 页）。也就是说，如果我们对数组 `a` 执行 `typeid(a)`，则所得的结果是数组类型而非指针类型。

当运算对象不属于类类型或者是一个不包含任何虚函数的类时，`typeid` 运算符指示的是运算对象的静态类型。而当运算对象是定义了至少一个虚函数的类的左值时，`typeid` 的结果直到运行时才会求得。

使用 typeid 运算符

通常情况下，我们使用 `typeid` 比较两条表达式的类型是否相同，或者比较一条表达式的类型是否与指定类型相同：

828

```
Derived *dp = new Derived;
Base *bp = dp;                                // 两个指针都指向 Derived 对象
// 在运行时比较两个对象的类型
```

```
if (typeid(*bp) == typeid(*dp)) {  
    // bp 和 dp 指向同一类型的对象  
}  
// 检查运行时类型是否是某种指定的类型  
if (typeid(*bp) == typeid(Derived)) {  
    // bp 实际指向 Derived 对象  
}
```

在第一个 if 语句中，我们比较 bp 和 dp 所指的对象的动态类型是否相同。如果相同，则条件成功。类似的，当 bp 当前所指的是一个 Derived 对象时，第二个 if 语句的条件满足。

注意， typeid 应该作用于对象，因此我们使用 *bp 而非 bp：

```
// 下面的检查永远是失败的：bp 的类型是指向 Base 的指针  
if (typeid(bp) == typeid(Derived)) {  
    // 此处的代码永远不会执行  
}
```

这个条件比较的是类型 Base* 和 Derived。尽管指针所指的对象类型是一个含有虚函数的类，但是指针本身并不是一个类类型的对象。类型 Base* 将在编译时求值，显然它与 Derived 不同，因此不论 bp 所指的对象到底是什么类型，上面的条件都不会满足。



当 typeid 作用于指针时（而非指针所指的对象），返回的结果是该指针的静态编译时类型。

typeid 是否需要运行时检查决定了表达式是否会被求值。只有当类型含有虚函数时，编译器才会对表达式求值。反之，如果类型不含有虚函数，则 typeid 返回表达式的静态类型；编译器无须对表达式求值也能知道表达式的静态类型。

如果表达式的动态类型可能与静态类型不同，则必须在运行时对表达式求值以确定返回的类型。这条规则适用于 typeid(*p) 的情况。如果指针 p 所指的类型不含有虚函数，则 p 不必非得是一个有效的指针。否则， *p 将在运行时求值，此时 p 必须是一个有效的指针。如果 p 是一个空指针，则 typeid(*p) 将抛出一个名为 bad_typeid 的异常。

19.2.2 节练习

练习 19.6：编写一条表达式将 Query_base 指针动态转换为 AndQuery 指针（参见 15.9.1 节，第 564 页）。分别使用 AndQuery 的对象以及其他类型的对象测试转换是否有效。打印一条表示类型转换是否成功的信息，确保实际输出的结果与期望的一致。

练习 19.7：编写与上一个练习类似的转换，这一次将 Query_base 对象转换为 AndQuery 的引用。重复上面的测试过程，确保转换能正常工作。

练习 19.8：编写一条 typeid 表达式检查两个 Query_base 对象是否指向同一种类型。再检查该类型是否是 AndQuery。

19.2.3 使用 RTTI

在某些情况下 RTTI 非常有用，比如当我们想为具有继承关系的类实现相等运算符时（参见 14.3.1 节，第 497 页）。对于两个对象来说，如果它们的类型相同并且对应的数据成

员取值相同，则我们说这两个对象是相等的。在类的继承体系中，每个派生类负责添加自己的数据成员，因此派生类的相等运算符必须把派生类的新成员考虑进来。

829 一种容易想到的解决方案是定义一套虚函数，令其在继承体系的各个层次上分别执行相等性判断。此时，我们可以为基类的引用定义一个相等运算符，该运算符将它的工作委托给虚函数 equal，由 equal 负责实际的操作。

遗憾的是，上述方案很难奏效。虚函数的基类版本和派生类版本必须具有相同的形参类型（参见 15.3 节，第 537 页）。如果我们想定义一个虚函数 equal，则该函数的形参必须是基类的引用。此时，equal 函数将只能使用基类的成员，而不能比较派生类独有的成员。

要想实现真正有效的相等比较操作，我们需要首先清楚一个事实：即如果参与比较的两个对象类型不同，则比较结果为 false。例如，如果我们试图比较一个基类对象和一个派生类对象，则==运算符应该返回 false。

基于上述推论，我们就可以使用 RTTI 解决问题了。我们定义的相等运算符的形参是基类的引用，然后使用 typeid 检查两个运算对象的类型是否一致。如果运算对象的类型不一致，则==返回 false；类型一致才调用 equal 函数。每个类定义的 equal 函数负责比较类型自己的成员。这些运算符接受 Base& 形参，但是在进行比较操作前先把运算对象转换成运算符所属的类类型。

类的层次关系

为了更好地解释上述概念，我们定义两个示例类：

```
class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // Base 的接口成员
protected:
    virtual bool equal(const Base&) const;
    // Base 的数据成员和其他用于实现的成员
};

830 class Derived: public Base {
public:
    // Derived 的其他接口成员
protected:
    bool equal(const Base&) const;
    // Derived 的数据成员和其他用于实现的成员
};
```

类型敏感的相等运算符

接下来介绍我们是如何定义整体的相等运算符的：

```
bool operator==(const Base &lhs, const Base &rhs)
{
    // 如果 typeid 不相同，返回 false；否则虚调用 equal
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}
```

在这个运算符中，如果运算对象的类型不同则返回 false。否则，如果运算对象的类型相同，则运算符将其工作委托给虚函数 equal。当运算对象是 Base 的对象时，调用 Base::equal；当运算对象是 Derived 的对象时，调用 Derived::equal。

虚 equal 函数

继承体系中的每个类必须定义自己的 equal 函数。派生类的所有函数要做的第一件事都是相同的，那就是将实参的类型转换为派生类类型：

```
bool Derived::equal(const Base &rhs) const
{
    // 我们清楚这两个类型是相等的，所以转换过程不会抛出异常
    auto r = dynamic_cast<const Derived&>(rhs);
    // 执行比较两个 Derived 对象的操作并返回结果
}
```

上面的类型转换永远不会失败，因为毕竟我们只有在验证了运算对象的类型相同之后才会调用该函数。然而这样的类型转换是必不可少的，执行了类型转换后，当前函数才能访问右侧运算对象的派生类成员。

基类 equal 函数

下面这个操作比其他的稍微简单一点：

```
bool Base::equal(const Base &rhs) const
{
    // 执行比较 Base 对象的操作
}
```

无须事先转换形参的类型。`*this` 和形参都是 `Base` 对象，因此当前对象可用的操作对于形参类型同样有效。

19.2.4 type_info 类

831

`type_info` 类的精确定义随着编译器的不同而略有差异。不过，C++ 标准规定 `type_info` 类必须定义在 `typeinfo` 头文件中，并且至少提供表 19.1 所列的操作。

表 19.1: `type_info` 的操作

<code>t1 == t2</code>	如果 <code>type_info</code> 对象 <code>t1</code> 和 <code>t2</code> 表示同一种类型，返回 <code>true</code> ；否则返回 <code>false</code>
<code>t1 != t2</code>	如果 <code>type_info</code> 对象 <code>t1</code> 和 <code>t2</code> 表示不同的类型，返回 <code>true</code> ；否则返回 <code>false</code>
<code>t.name()</code>	返回一个 C 风格字符串，表示类型名字的可打印形式。类型名字的生成方式因系统而异
<code>t1.before(t2)</code>	返回一个 <code>bool</code> 值，表示 <code>t1</code> 是否位于 <code>t2</code> 之前。 <code>before</code> 所采用的顺序关系是依赖于编译器的

除此之外，因为 `type_info` 类一般是作为一个基类出现，所以它还应该提供一个公有的虚析构函数。当编译器希望提供额外的类型信息时，通常在 `type_info` 的派生类中完成。

`type_info` 类没有默认构造函数，而且它的拷贝和移动构造函数以及赋值运算符都被定义成删除的（参见 13.1.6 节，第 450 页）。因此，我们无法定义或拷贝 `type_info` 类型的对象，也不能为 `type_info` 类型的对象赋值。创建 `type_info` 对象的唯一途径是使用 `typeid` 运算符。

`type_info` 类的 `name` 成员函数返回一个 C 风格字符串，表示对象的类型名字。对

于某种给定的类型来说，`name` 的返回值因编译器而异并且不一定与在程序中使用的名字一致。对于 `name` 返回值的唯一要求是，类型不同则返回的字符串必须有所区别。例如：

```
int arr[10];
Derived d;
Base *p = &d;

cout << typeid(42).name() << ", "
<< typeid(arr).name() << ", "
<< typeid(Sales_data).name() << ", "
<< typeid(std::string).name() << ", "
<< typeid(p).name() << ", "
<< typeid(*p).name() << endl;
```

在作者的计算机上运行该程序，输出结果如下：

```
i, A10_i, 10Sales_data, Ss, P4Base, 7Derived
```



`type_info` 类在不同的编译器上有所区别。有的编译器提供了额外的成员函数以提供程序中所用类型的额外信息。读者应该仔细阅读你所用编译器的使用手册，从而获取关于 `type_info` 的更多细节。

832

19.2.4 节练习

练习 19.9：编写与本节最后一个程序类似的代码，令其打印你的编译器为一些常见类型所起的名字。如果你得到的输出结果与本书类似，尝试编写一个函数将这些字符串翻译成人们更容易读懂的形式。

练习 19.10：已知存在如下的类继承体系，其中每个类定义了一个默认公有的构造函数和一个虚析构函数。下面的语句将打印哪些类型名字？

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };

(a) A *pa = new C;
    cout << typeid(pa).name() << endl;
(b) C cobj;
    A& ra = cobj;
    cout << typeid(&ra).name() << endl;
(c) B *px = new B;
    A& ra = *px;
    cout << typeid(ra).name() << endl;
```

19.3 枚举类型

枚举类型（enumeration）使我们可以将一组整型常量组织在一起。和类一样，每个枚举类型定义了一种新的类型。枚举属于字面值常量类型（参见 7.5.6 节，第 267 页）。

C++包含两种枚举：限定作用域的和不限定作用域的。C++11 新标准引入了限定作用域的枚举类型（scoped enumeration）。定义限定作用域的枚举类型的一般形式是：首先是关键字 `enum class`（或者等价地使用 `enum struct`），随后是枚举类型名字以及用花括

号括起来的以逗号分隔的枚举成员（enumerator）列表，最后是一个分号：

```
enum class open_modes {input, output, append};
```

我们定义了一个名为 `open_modes` 的枚举类型，它包含三个枚举成员：`input`、`output` 和 `append`。

定义不限定作用域的枚举类型（unscoped enumeration）时省略掉关键字 `class`（或 `struct`），枚举类型的名字是可选的：

```
enum color {red, yellow, green};           // 不限定作用域的枚举类型
// 未命名的、不限定作用域的枚举类型
enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

如果 `enum` 是未命名的，则我们只能在定义该 `enum` 时定义它的对象。和类的定义类似，我们需要在 `enum` 定义的右侧花括号和最后的分号之间提供逗号分隔的声明列表（参见 2.6.1 节，第 64 页）。

枚举成员

< 833

在限定作用域的枚举类型中，枚举成员的名字遵循常规的作用域准则，并且在枚举类型的作用域外是不可访问的。与之相反，在不限定作用域的枚举类型中，枚举成员的作用域与枚举类型本身的作用域相同：

```
enum color {red, yellow, green};           // 不限定作用域的枚举类型
enum stoplight {red, yellow, green};       // 错误：重复定义了枚举成员
enum class peppers {red, yellow, green};   // 正确：枚举成员被隐藏了
color eyes = green; // 正确：不限定作用域的枚举类型的枚举成员位于有效的作用域中
peppers p = green; // 错误：peppers 的枚举成员不在有效的作用域中
                           // color::green 在有效的作用域中，但是类型错误
color hair = color::red;                  // 正确：允许显式地访问枚举成员
peppers p2 = peppers::red;                // 正确：使用 pappers 的 red
```

默认情况下，枚举值从 0 开始，依次加 1。不过我们也能为一个或几个枚举成员指定专门的值：

```
enum class intTypes {
    charTyp = 8, shortTyp = 16, intTyp = 16,
    longTyp = 32, long_longTyp = 64
};
```

由枚举成员 `intTyp` 和 `shortTyp` 可知，枚举值不一定唯一。如果我们没有显式地提供初始值，则当前枚举成员的值等于之前枚举成员的值加 1。

枚举成员是 `const`，因此在初始化枚举成员时提供的初始值必须是常量表达式（参见 2.4.4 节，第 58 页）。也就是说，每个枚举成员本身就是一条常量表达式，我们可以在任何需要常量表达式的地方使用枚举成员。例如，我们可以定义枚举类型的 `constexpr` 变量：

```
constexpr intTypes charbits = intTypes::charTyp;
```

类似的，我们也可以将一个 `enum` 作为 `switch` 语句的表达式，而将枚举值作为 `case` 标签（参见 5.3.2 节，第 160 页）。出于同样的原因，我们还能将枚举类型作为一个非类型模板形参使用（参见 16.1.1 节，第 580 页）；或者在类的定义中初始化枚举类型的静态数据成员（参见 7.6 节，第 270 页）。

和类一样，枚举也定义新的类型

只要 enum 有名字，我们就能定义并初始化该类型的成员。要想初始化 enum 对象或者为 enum 对象赋值，必须使用该类型的一个枚举成员或者该类型的另一个对象：

```
open_modes om = 2;           // 错误：2 不属于类型 open_modes
om = open_modes::input;      // 正确：input 是 open_modes 的一个枚举成员
```

834 一个不限定作用域的枚举类型的对象或枚举成员自动地转换成整型。因此，我们可以在任何需要整型值的地方使用它们：

```
int i = color::red; // 正确：不限定作用域的枚举类型的枚举成员隐式地转换成 int
int j = peppers::red; // 错误：限定作用域的枚举类型不会进行隐式转换
```

指定 enum 的大小

尽管每个 enum 都定义了唯一的类型，但实际上 enum 是由某种整数类型表示的。在 C++11 新标准中，我们可以在 enum 的名字后加上冒号以及我们想在该 enum 中使用的类型：

```
enum intValues : unsigned long long {
    charTyp = 255, shortTyp = 65535, intTyp = 65535,
    longTyp = 4294967295UL,
    long_longTyp = 18446744073709551615ULL
};
```

如果我们没有指定 enum 的潜在类型，则默认情况下限定作用域的 enum 成员类型是 int。对于不限定作用域的枚举类型来说，其枚举成员不存在默认类型，我们只知道成员的潜在类型足够大，肯定能够容纳枚举值。如果我们指定了枚举成员的潜在类型（包括对限定作用域的 enum 的隐式指定），则一旦某个枚举成员的值超出了该类型所能容纳的范围，将引发程序错误。

指定 enum 潜在类型的能力使得我们可以控制不同实现环境中使用的类型，我们将可以确保在一种实现环境中编译通过的程序所生成的代码与其他实现环境中生成的代码一致。

枚举类型的前置声明

C++11 在 C++11 新标准中，我们可以提前声明 enum。enum 的前置声明（无论隐式地还是显示地）必须指定其成员的大小：

```
// 不限定作用域的枚举类型 intValues 的前置声明
enum intValues : unsigned long long; // 不限定作用域的，必须指定成员类型
enum class open_modes; // 限定作用域的枚举类型可以使用默认成员类型 int
```

因为不限定作用域的 enum 未指定成员的默认大小，因此每个声明必须指定成员的大小。对于限定作用域的 enum 来说，我们可以不指定其成员的大小，这个值被隐式地定义成 int。

和其他声明语句一样，enum 的声明和定义必须匹配，这意味着在该 enum 的所有声明和定义中成员的大小必须一致。而且，我们不能在同一个上下文中先声明一个不限定作用域的 enum 名字，然后再声明一个同名的限定作用域的 enum：

```
// 错误：所有的声明和定义必须对该 enum 是限定作用域的还是不限定作用域的保持一致
enum class intValues;
enum intValues; // 错误：intValues 已经被声明成限定作用域的 enum
enum intValues : long; // 错误：intValues 已经被声明成 int
```

形参匹配与枚举类型

<835

要想初始化一个 enum 对象，必须使用该 enum 类型的另一个对象或者它的一个枚举成员（参见 19.3 节，第 737 页）。因此，即使某个整型值恰好与枚举成员的值相等，它也不能作为函数的 enum 实参使用：

```
// 不限定作用域的枚举类型，潜在类型因机器而异
enum Tokens {INLINE = 128, VIRTUAL = 129};
void ff(Tokens);
void ff(int);
int main() {
    Tokens curTok = INLINE;
    ff(128);                      // 精确匹配 ff(int)
    ff(INLINE);                    // 精确匹配 ff(Tokens)
    ff(curTok);                   // 精确匹配 ff(Tokens)
    return 0;
}
```

尽管我们不能直接将整型值传给 enum 形参，但是可以将一个不限定作用域的枚举类型的对象或枚举成员传给整型形参。此时，enum 的值提升成 int 或更大的整型，实际提升的结果由枚举类型的潜在类型决定：

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

枚举类型 Tokens 只有两个枚举成员，其中较大的那个值是 129。该枚举类型可以用 unsigned char 来表示，因此很多编译器使用 unsigned char 作为 Tokens 的潜在类型。不管 Tokens 的潜在类型到底是什么，它的对象和枚举成员都提升成 int。尤其是，枚举成员永远不会提升成 unsigned char，即使枚举值可以用 unsigned char 存储也是如此。

19.4 类成员指针

成员指针（pointer to member）是指可以指向类的非静态成员的指针。一般情况下，指针指向一个对象，但是成员指针指示的是类的成员，而非类的对象。类的静态成员不属于任何对象，因此无须特殊的指向静态成员的指针，指向静态成员的指针与普通指针没有什么区别。

成员指针的类型囊括了类的类型以及成员的类型。当初始化一个这样的指针时，我们令其指向类的某个成员，但是不指定该成员所属的对象；直到使用成员指针时，才提供成员所属的对象。

为了解释成员指针的原理，不妨使用 7.3.1 节（第 243 页）的 Screen 类：

<836

```
class Screen {
public:
    typedef std::string::size_type pos;
    char get_cursor() const { return contents[cursor]; }
    char get() const;
```

```

    char get(pos ht, pos wd) const;
private:
    std::string contents;
    pos cursor;
    pos height, width;
};

```

19.4.1 数据成员指针

和其他指针一样，在声明成员指针时我们也使用*来表示当前声明的名字是一个指针。与普通指针不同的是，成员指针还必须包含成员所属的类。因此，我们必须在*之前添加 `classname::` 以表示当前定义的指针可以指向 `classname` 的成员。例如：

```
// pdata 可以指向一个常量（非常量）Screen 对象的 string 成员
const string Screen::*pdata;
```

上述语句将 `pdata` 声明成“一个指向 `Screen` 类的 `const string` 成员的指针”。常量对象的数据成员本身也是常量，因此将我们的指针声明成指向 `const string` 成员的指针意味着 `pdata` 可以指向任何 `Screen` 对象的一个成员，而不管该 `Screen` 对象是否是常量。作为交换条件，我们只能使用 `pdata` 读取它所指的成员，而不能向它写入内容。

当我们初始化一个成员指针（或者向它赋值）时，需指定它所指的成员。例如，我们可以令 `pdata` 指向某个非特定 `Screen` 对象的 `contents` 成员：

```
pdata = &Screen::contents;
```

其中，我们将取地址运算符作用于 `Screen` 类的成员而非内存中的一个该类对象。

当然，在 C++11 新标准中声明成员指针最简单的方法是使用 `auto` 或 `decltype`：

```
auto pdata = &Screen::contents;
```

使用数据成员指针

读者必须清楚的一点是，当我们初始化一个成员指针或为成员指针赋值时，该指针并没有指向任何数据。成员指针指定了成员而非该成员所属的对象，只有当解引用成员指针时我们才提供对象的信息。

837 与成员访问运算符`.`和`->`类似，也有两种成员指针访问运算符：`.*`和`->*`，这两个运算符使得我们可以解引用指针并获得该对象的成员：

```

Screen myScreen, *pScreen = &myScreen;
// .*解引用 pdata 以获得 myScreen 对象的 contents 成员
auto s = myScreen.*pdata;
// ->*解引用 pdata 以获得 pScreen 所指对象的 contents 成员
s = pScreen->*pdata;

```

从概念上来说，这些运算符执行两步操作：它们首先解引用成员指针以得到所需的成员；然后像成员访问运算符一样，通过对对象（`.*`）或指针（`->*`）获取成员。

返回数据成员指针的函数

常规的访问控制规则对成员指针同样有效。例如，`Screen` 的 `contents` 成员是私有的，因此之前对于 `pdata` 的使用必须位于 `Screen` 类的成员或友元内部，否则程序将发生错误。

因为数据成员一般情况下是私有的，所以我们通常不能直接获得数据成员的指针。如果一个像 Screen 这样的类希望我们可以访问它的 contents 成员，最好定义一个函数，令其返回值是指向该成员的指针：

```
class Screen {  
public:  
    // data 是一个静态成员，返回一个成员指针  
    static const std::string Screen::*data()  
    { return &Screen::contents; }  
    // 其他成员与之前的版本一致  
};
```

我们为 Screen 类添加了一个静态成员，令其返回指向 contents 成员的指针。显然该函数的返回类型与最初的 pdata 指针类型一致。从右向左阅读函数的返回类型，可知 data 返回的是一个指向 Screen 类的 const string 成员的指针。函数体对 contents 成员使用了取地址运算符，因此函数将返回指向 Screen 类 contents 成员的指针。

当我们调用 data 函数时，将得到一个成员指针：

```
// data() 返回一个指向 Screen 类的 contents 成员的指针  
const string Screen::*pdata = Screen::data();
```

一如往常，pdata 指向 Screen 类的成员而非实际数据。要想使用 pdata，必须把它绑定到 Screen 类型的对象上：

```
// 获得 myScreen 对象的 contents 成员  
auto s = myScreen.*pdata;
```

19.4.1 节练习

< 838

练习 19.11：普通的数据指针与指向数据成员的指针有何区别？

练习 19.12：定义一个成员指针，令其可以指向 Screen 类的 cursor 成员。通过该指针获得 Screen::cursor 的值。

练习 19.13：定义一个类型，使其可以表示指向 Sales_data 类的 bookNo 成员的指针。

19.4.2 成员函数指针

我们也可以定义指向类的成员函数的指针。与指向数据成员的指针类似，对于我们来说要想创建一个指向成员函数的指针，最简单的方法是使用 auto 来推断类型：

```
// pmf 是一个指针，它可以指向 Screen 的某个常量成员函数  
// 前提是该函数不接受任何实参，并且返回一个 char  
auto pmf = &Screen::get_cursor;
```

和指向数据成员的指针一样，我们使用 `classname::*` 的形式声明一个指向成员函数的指针。类似于任何其他函数指针（参见 6.7 节，第 221 页），指向成员函数的指针也需要指定目标函数的返回类型和形参列表。如果成员函数是 const 成员（参见 7.1.2 节，第 231 页）或者引用成员（参见 13.6.3 节，第 483 页），则我们必须将 const 限定符或引用限定符包含进来。

和普通的函数指针类似，如果成员存在重载的问题，则我们必须显式地声明函数类型以明确指出我们想要使用的是哪个函数（参见 6.7 节，第 211 页）。例如，我们可以声明一

个指针，令其指向含有两个形参的 get：

```
char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
pmf2 = &Screen::get;
```

出于优先级的考虑，上述声明中 Screen::*两端的括号必不可少。如果没有这对括号的话，编译器将认为该声明是一个（无效的）函数声明：

```
// 错误：非成员函数 p 不能使用 const 限定符
char Screen::*p(Screen::pos, Screen::pos) const;
```

这个声明试图定义一个名为 p 的普通函数，并且返回 Screen 类的一个 char 成员。因为它声明的是一个普通函数，所以不能使用 const 限定符。

和普通函数指针不同的是，在成员函数和指向该成员的指针之间不存在自动转换规则：

```
// pmf 指向一个 Screen 成员，该成员不接受任何实参且返回类型是 char
pmf = &Screen::get;           // 必须显式地使用取地址运算符
pmf = Screen::get;           // 错误：在成员函数和指针之间不存在自动转换规则
```

839 使用成员函数指针

和使用指向数据成员的指针一样，我们使用 .* 或者 ->* 运算符作用于指向成员函数的指针，以调用类的成员函数：

```
Screen myScreen, *pScreen = &myScreen;
// 通过 pScreen 所指的对象调用 pmf 所指的函数
char c1 = (pScreen->*pmf)();
// 通过 myScreen 对象将实参 0, 0 传给含有两个形参的 get 函数
char c2 = (myScreen.*pmf2)(0, 0);
```

之所以 (myScreen->*pmf)() 和 (pScreen.*pmf2)(0, 0) 的括号必不可少，原因是调用运算符的优先级要高于指针指向成员运算符的优先级。

假设去掉括号的话，

```
myScreen.*pmf()
```

其含义将等同于下面的式子：

```
myScreen.*(pmf())
```

这行代码的意思是调用一个名为 pmf 的函数，然后使用该函数的返回值作为指针指向成员运算符 (.*) 的运算对象。然而 pmf 并不是一个函数，因此代码将发生错误。



因为函数调用运算符的优先级较高，所以在声明指向成员函数的指针并使用这样的指针进行函数调用时，括号必不可少：(C::*p)(parms) 和 (obj.*p)(args)。

使用成员指针的类型别名

使用类型别名或 `typedef`（参见 2.5.1 节，第 60 页）可以让成员指针更容易理解。例如，下面的类型别名将 Action 定义为两参数 get 函数的同义词：

```
// Action 是一种可以指向 Screen 成员函数的指针，它接受两个 pos 实参，返回一个 char
using Action =
char (Screen::*)(Screen::pos, Screen::pos) const;
```

Action 是某类型的另外一个名字，该类型是“指向 Screen 类的常量成员函数的指针，其中这个成员函数接受两个 pos 形参，返回一个 char”。通过使用 Action，我们可以简化指向 get 的指针定义：

```
Action get = &Screen::get; // get 指向 Screen 的 get 成员
```

和其他函数指针类似，我们可以将指向成员函数的指针作为某个函数的返回类型或形参类型。其中，指向成员的指针形参也可以拥有默认实参：

```
// action 接受一个 Screen 的引用，和一个指向 Screen 成员函数的指针
Screen& action(Screen&, Action = &Screen::get);
```

action 是包含两个形参的函数，其中一个形参是 Screen 对象的引用，另一个形参是指向 Screen 成员函数的指针，成员函数必须接受两个 pos 形参并返回一个 char。当我们调用 action 时，只需将 Screen 的一个符合要求的函数的指针或地址传入即可： ◀840

```
Screen myScreen;
// 等价的调用：
action(myScreen); // 使用默认实参
action(myScreen, get); // 使用我们之前定义的变量 get
action(myScreen, &Screen::get); // 显式地传入地址
```



通过使用类型别名，可以令含有成员指针的代码更易读写。

成员指针函数表

对于普通函数指针和指向成员函数的指针来说，一种常见的用法是将其存入一个函数表当中（参见 14.8.3 节，第 511 页）。如果一个类含有几个相同类型的成员，则这样一张表可以帮助我们从这些成员中选择一个。假定 Screen 类含有几个成员函数，每个函数负责将光标向指定的方向移动：

```
class Screen {
public:
    // 其他接口和实现成员与之前一致
    Screen& home(); // 光标移动函数
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
```

这几个新函数有一个共同点：它们都不接受任何参数，并且返回值是发生光标移动的 Screen 的引用。

我们希望定义一个 move 函数，使其可以调用上面的任意一个函数并执行对应的操作。为了支持这个新函数，我们将在 Screen 中添加一个静态成员，该成员是指向光标移动函数的指针的数组：

```
class Screen {
public:
    // 其他接口和实现成员与之前一致
    // Action 是一个指针，可以用任意一个光标移动函数对其赋值
    using Action = Screen& (Screen::*)(());
    // 指定具体要移动的方向，其中 enum 参见 19.3 节（第 736 页）
```

```

enum Directions { HOME, FORWARD, BACK, UP, DOWN };
Screen& move(Directions);
private:
    static Action Menu[]; // 函数表
};

```

841> 数组 Menu 依次保存每个光标移动函数的指针，这些函数将按照 Directions 中枚举成员对应的偏移量存储。move 函数接受一个枚举成员并调用相应的函数：

```

Screen& Screen::move(Directions cm)
{
    // 运行 this 对象中索引值为 cm 的元素
    return (this->*Menu[cm])(); // Menu[cm] 指向一个成员函数
}

```

move 中的函数调用的原理是：首先获取索引值为 cm 的 Menu 元素，该元素是指向 Screen 成员函数的指针。我们根据 this 所指的对象调用该元素所指的成员函数。

当我们调用 move 函数时，给它传入一个表示光标移动方向的枚举成员：

```

Screen myScreen;
myScreen.move(Screen::HOME); // 调用 myScreen.home
myScreen.move(Screen::DOWN); // 调用 myScreen.down

```

剩下的工作就是定义并初始化函数表本身了：

```

Screen::Action Screen::Menu[] = { &Screen::home,
                                  &Screen::forward,
                                  &Screen::back,
                                  &Screen::up,
                                  &Screen::down,
};

```

19.4.2 节练习

练习 19.14：下面的代码合法吗？如果合法，代码的含义是什么？如果不合法，解释原因。

```

auto pmf = &Screen::get_cursor;
pmf = &Screen::get;

```

练习 19.15：普通函数指针和指向成员函数的指针有何区别？

练习 19.16：声明一个类型别名，令其作为指向 Sales_data 的 avg_price 成员的指针的同义词。

练习 19.17：为 Screen 的所有成员函数类型各定义一个类型别名。

842> 19.4.3 将成员函数用作可调用对象

如我们所知，要想通过一个指向成员函数的指针进行函数调用，必须首先利用 .* 运算符或 ->* 运算符将该指针绑定到特定的对象上。因此与普通的函数指针不同，成员指针不是一个可调用对象，这样的指针不支持函数调用运算符（参见 10.3.2 节，第 346 页）。

因为成员指针不是可调用对象，所以我们不能直接将一个指向成员函数的指针传递给算法。举个例子，如果我们想在一个 string 的 vector 中找到第一个空 string，显然

不能使用下面的语句：

```
auto fp = &string::empty; // fp 指向 string 的 empty 函数
// 错误，必须使用.*或->*调用成员指针
find_if(svec.begin(), svec.end(), fp);
```

`find_if` 算法需要一个可调用对象，但我们提供给它的是一个指向成员函数的指针 `fp`。因此在 `find_if` 的内部将执行如下形式的代码，从而导致无法通过编译：

```
// 检查对当前元素的断言是否为真
if (fp(*it)) // 错误：要想通过成员指针调用函数，必须使用->*运算符
```

显然该语句试图调用的是传入的对象，而非函数。

使用 `function` 生成一个可调用对象

从指向成员函数的指针获取可调用对象的一种方法是使用标准库模板 `function`（参见 14.8.3 节，第 511 页）：

```
function<bool (const string&)> fcn = &string::empty;
find_if(svec.begin(), svec.end(), fcn);
```

我们告诉 `function` 一个事实：即 `empty` 是一个接受 `string` 参数并返回 `bool` 值的函数。通常情况下，执行成员函数的对象将被传给隐式的 `this` 形参。当我们想要使用 `function` 为成员函数生成一个可调用对象时，必须首先“翻译”该代码，使得隐式的形参变成显式的。

当一个 `function` 对象包含有一个指向成员函数的指针时，`function` 类知道它必须使用正确的指向成员的指针运算符来执行函数调用。也就是说，我们可以认为在 `find_if` 当中含有类似于如下形式的代码：

```
// 假设 it 是 find_if 内部的迭代器，则*it 是给定范围内的一个对象
if (fcn(*it)) // 假设 fcn 是 find_if 内部的一个可调用对象的名字
```

其中，`function` 将使用正确的指向成员的指针运算符。从本质上来看，`function` 类将函数调用转换成了如下形式：

```
// 假设 it 是 find_if 内部的迭代器，则*it 是给定范围内的一个对象
if (((*it).*p)()) // 假设 p 是 fcn 内部的一个指向成员函数的指针
```

当我们定义一个 `function` 对象时，必须指定该对象所能表示的函数类型，即可调用对象的形式。如果可调用对象是一个成员函数，则第一个形参必须表示该成员是在哪个（一般是隐式的）对象上执行的。同时，我们提供给 `function` 的形式中还必须指明对象是否是以指针或引用的形式传入的。

以定义 `fcn` 为例，我们想在 `string` 对象的序列上调用 `find_if`，因此我们要求 `function` 生成一个接受 `string` 对象的可调用对象。又因为我们的 `vector` 保存的是 `string` 的指针，所以必须指定 `function` 接受指针：

```
vector<string*> pvec;
function<bool (const string*)> fp = &string::empty;
// fp 接受一个指向 string 的指针，然后使用->*调用 empty
find_if(pvec.begin(), pvec.end(), fp);
```

843

使用 `mem_fn` 生成一个可调用对象

通过上面的介绍可知，要想使用 `function`，我们必须提供成员的调用形式。我们也

C++ 11 可以采取另外一种方法，通过使用标准库功能 `mem_fn` 来让编译器负责推断成员的类型。和 `function` 一样，`mem_fn` 也定义在 `functional` 头文件中，并且可以从成员指针生成一个可调用对象；和 `function` 不同的是，`mem_fn` 可以根据成员指针的类型推断可调用对象的类型，而无须用户显式地指定：

```
find_if(svec.begin(), svec.end(), mem_fn(&string::empty));
```

我们使用 `mem_fn(&string::empty)` 生成一个可调用对象，该对象接受一个 `string` 实参，返回一个 `bool` 值。

`mem_fn` 生成的可调用对象可以通过对象调用，也可以通过指针调用：

```
auto f = mem_fn(&string::empty); // f 接受一个 string 或者一个 string*
f(*svec.begin()); // 正确：传入一个 string 对象，f 使用.*调用 empty
f(&svec[0]); // 正确：传入一个 string 的指针，f 使用->*调用 empty
```

实际上，我们可以认为 `mem_fn` 生成的可调用对象含有一对重载的函数调用运算符：一个接受 `string*`，另一个接受 `string&`。

使用 `bind` 生成一个可调用对象

出于完整性的考虑，我们还可以使用 `bind`（参见 10.3.4 节，第 354 页）从成员函数生成一个可调用对象：

```
// 选择范围中的每个 string，并将其 bind 到 empty 的第一个隐式实参上
auto it = find_if(svec.begin(), svec.end(),
                  bind(&string::empty, _1));
```

和 `function` 类似的地方是，当我们使用 `bind` 时，必须将函数中用于表示执行对象的隐式形参转换成显式的。和 `mem_fn` 类似的地方是，`bind` 生成的可调用对象的第一个实参既可以是 `string` 的指针，也可以是 `string` 的引用：

```
auto f = bind(&string::empty, _1);
f(*svec.begin()); // 正确：实参是一个 string，f 使用.*调用 empty
f(&svec[0]); // 正确：实参是一个 string 的指针，f 使用->*调用 empty
```

19.4.3 节练习

练习 19.18：编写一个函数，使用 `count_if` 统计在给定的 `vector` 中有多少个空 `string`。

练习 19.19：编写一个函数，令其接受 `vector<Sales_data>` 并查找平均价格高于某个值的第一个元素。

19.5 嵌套类

一个类可以定义在另一个类的内部，前者称为 **嵌套类**（nested class）或 **嵌套类型**（nested type）。嵌套类常用于定义作为实现部分的类，比如我们在文本查询示例中使用的 `QueryResult` 类（参见 12.3 节，第 430 页）。

844 嵌套类是一个独立的类，与外层类基本没什么关系。特别是，外层类的对象和嵌套类的对象是相互独立的。在嵌套类的对象中不包含任何外层类定义的成员；类似的，在外层类的对象中也不包含任何嵌套类定义的成员。

嵌套类的名字在外层类作用域中是可见的，在外层类作用域之外不可见。和其他嵌套的名字一样，嵌套类的名字不会和别的作用域中的同一个名字冲突。

嵌套类中成员的种类与非嵌套类是一样的。和其他类类似，嵌套类也使用访问限定符来控制外界对其成员的访问权限。外层类对嵌套类的成员没有特殊的访问权限，同样，嵌套类对外层类的成员也没有特殊的访问权限。

嵌套类在其外层类中定义了一个类型成员。和其他成员类似，该类型的访问权限由外层类决定。位于外层类 `public` 部分的嵌套类实际上定义了一种可以随处访问的类型；位于外层类 `protected` 部分的嵌套类定义的类型只能被外层类及其友元和派生类访问；位于外层类 `private` 部分的嵌套类定义的类型只能被外层类的成员和友元访问。

声明一个嵌套类

我们为 12.3.2 节（第 432 页）的 `TextQuery` 类定义了一个名为 `QueryResult` 的配套类，这两个类密切相关。`QueryResult` 类的主要作用是表示 `TextQuery` 对象上 `query` 操作的结果，显然将 `QueryResult` 用作其他目的没有任何意义。为了充分体现这种紧密的相关性，我们可以把 `QueryResult` 定义成 `TextQuery` 的成员。

```
class TextQuery {
public:
    class QueryResult; // 嵌套类稍后定义
    // 其他成员与 12.3.2 节（第 432 页）一致
};
```

我们只需对原来的 `TextQuery` 类做一处改动，即将 `QueryResult` 声明成嵌套类。因为 `QueryResult` 是一个类型成员（参见 7.4.1 节，第 254 页），所以我们必须对它先声明后使用，尤其是必须先声明 `QueryResult`，再将它作为 `query` 成员的返回类型。类的其他成员没有任何变化。

在外层类之外定义一个嵌套类

<845>

我们在 `TextQuery` 内声明了 `QueryResult`，但是没有给出它的定义。和成员函数一样，嵌套类必须声明在类的内部，但是可以定义在类的内部或者外部。

当我们在外层类之外定义一个嵌套类时，必须以外层类的名字限定嵌套类的名字：

```
// QueryResult 是 TextQuery 的成员，下面的代码负责定义 QueryResult
class TextQuery::QueryResult {
    // 位于类的作用域内，因此我们不必对 QueryResult 形参进行限定
    friend std::ostream&
        print(std::ostream&, const QueryResult&);

public:
    // 无须定义 QueryResult::line_no
    // 嵌套类可以直接使用外层类的成员，无须对该成员的名字进行限定
    QueryResult(std::string,
                std::shared_ptr<std::set<line_no>>,
                std::shared_ptr<std::vector<std::string>>);
    // 其他成员与 12.3.2 节（第 432 页）一致
};
```

和原来的类相比唯一的改动是，我们无须在 `QueryResult` 内定义 `line_no` 成员了。因为该成员属于 `TextQuery`，所以 `QueryResult` 可以直接访问它而不必再定义一次。



在嵌套类在其外层类之外完成真正的定义之前，它都是一个不完全类型（参见 7.3.3 节，第 250 页）。

定义嵌套类的成员

在这个版本的 `QueryResult` 类中，我们并没有在类的内部定义其构造函数。要想为其定义构造函数，必须指明 `QueryResult` 是嵌套在 `TextQuery` 的作用域之内的。具体做法是使用外层类的名字限定嵌套类的名字：

```
// QueryResult 类嵌套在 TextQuery 类中
// 下面的代码为 QueryResult 类定义名为 QueryResult 的成员
TextQuery::QueryResult::QueryResult(string s,
                                     shared_ptr<set<line_no>> p,
                                     shared_ptr<vector<string>> f):
    sought(s), lines(p), file(f) { }
```

从右向左阅读函数的名字可知我们定义的是 `QueryResult` 类的构造函数，而 `QueryResult` 类是嵌套在 `TextQuery` 类中的。该构造函数除了把实参值赋给对应的数据成员之外，没有做其他工作。

嵌套类的静态成员定义

如果 `QueryResult` 声明了一个静态成员，则该成员的定义将位于 `TextQuery` 的作用域之外。例如，假设 `QueryResult` 有一个静态成员，则该成员的定义将形如：

```
// QueryResult 类嵌套在 TextQuery 类中,
// 下面的代码为 QueryResult 定义一个静态成员
int TextQuery::QueryResult::static_mem = 1024;
```

嵌套类作用域中的名字查找

名字查找的一般规则（参见 7.4.1 节，第 254 页）在嵌套类中同样适用。当然，因为嵌套类本身是一个嵌套作用域，所以还必须查找嵌套类的外层作用域。这种作用域嵌套的性质正好可以说明为什么我们不在 `QueryResult` 的嵌套版本中定义 `line_no`。原来的 `QueryResult` 类定义了该成员，从而使其成员可以避免使用 `TextQuery::line_no` 的形式。然而 `QueryResult` 的嵌套类版本本身就是定义在 `TextQuery` 中的，所以我们不需要再使用 `typedef`。嵌套的 `QueryResult` 无须说明 `line_no` 属于 `TextQuery` 就可以直接使用它。

如我们所知，嵌套类是其外层类的一个类型成员，因此外层类的成员可以像使用任何其他类型成员一样使用嵌套类的名字。因为 `QueryResult` 嵌套在 `TextQuery` 中，所以 `TextQuery` 的 `query` 成员可以直接使用名字 `QueryResult`：

```
// 返回类型必须指明 QueryResult 是一个嵌套类
TextQuery::QueryResult
TextQuery::query(const string &sought) const
{
    // 如果我们没有找到 sought，则返回 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而非下标以避免向 wm 中添加单词
    auto loc = wm.find(sought);
    if (loc == wm.end())
```

```

        return QueryResult(sought, nodata, file);           // 没有找到
    else
        return QueryResult(sought, loc->second, file);
}

```

和过去一样，返回类型不在类的作用域中（参见 7.4 节，第 253 页），因此我们必须指明函数的返回值是 `TextQuery::QueryResult` 类型。不过在函数体内部我们可以直接访问 `QueryResult`，比如上面的 `return` 语句就是这样。

嵌套类和外层类是相互独立的

尽管嵌套类定义在其外层类的作用域中，但是读者必须谨记外层类的对象和嵌套类的对象没有任何关系。嵌套类的对象只包含嵌套类定义的成员；同样，外层类的对象只包含外层类定义的成员，在外层类对象中不会有任何嵌套类的成员。

说得再具体一些，`TextQuery::query` 的第二条 `return` 语句

```
return QueryResult(sought, loc->second, file);
```

使用了 `TextQuery` 对象的数据成员，而 `query` 正是用它们来初始化 `QueryResult` 对象的。因为在一个 `QueryResult` 对象中不包含其外层类的成员，所以我们必须使用上述成员构造我们返回的 `QueryResult` 对象。

< 847

19.5 节练习

练习 19.20：将你的 `QueryResult` 类嵌套在 `TextQuery` 中，然后重新运行 12.3.2 节（第 435 页）中使用了 `TextQuery` 的程序。

19.6 union：一种节省空间的类

联合（union）是一种特殊的类。一个 `union` 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当我们给 `union` 的某个成员赋值之后，该 `union` 的其他成员就变成未定义的状态了。分配给一个 `union` 对象的存储空间至少要能容纳它的最大的数据成员。和其他类一样，一个 `union` 定义了一种新类型。

类的某些特性对 `union` 同样适用，但并非所有特性都如此。`union` 不能含有引用类型的成员，除此之外，它的成员可以是绝大多数类型。在 C++11 新标准中，含有构造函数或析构函数的类类型也可以作为 `union` 的成员类型。`union` 可以为其成员指定 `public`、`protected` 和 `private` 等保护标记。默认情况下，`union` 的成员都是公有的，这一点与 `struct` 相同。

`union` 可以定义包括构造函数和析构函数在内的成员函数。但是由于 `union` 既不能继承自其他类，也不能作为基类使用，所以在 `union` 中不能含有虚函数。

定义 `union`

`union` 提供了一种有效的途径使得我们可以方便地表示一组类型不同的互斥值。举个例子，假设我们需要处理一些不同种类的数字数据和字符数据，则在此过程中可以定义一个 `union` 来保存这些值：

```
// Token 类型的对象只有一个成员，该成员的类型可能是下列类型中的任意一种
union Token {
```

```
// 默认情况下成员是公有的
char    cval;
int     ival;
double  dval;
};
```

在定义一个 union 时，首先是关键字 union，随后是该 union 的（可选的）名字以及花括号内的一组成员声明。上面的代码定义了一个名为 Token 的 union，它可以保存一个值，这个值的类型可能是 char、int 或 double 中的一种。

848 使用 union 类型

union 的名字是一个类型名。和其他内置类型一样，默认情况下 union 是未初始化的。我们可以像显式地初始化聚合类（参见 7.5.5 节，第 266 页）一样使用一对花括号内的初始值显式地初始化一个 union：

```
Token first_token = {'a'};           // 初始化 cval 成员
Token last_token;                   // 未初始化的 Token 对象
Token *pt = new Token;              // 指向一个未初始化的 Token 对象的指针
```

如果提供了初始值，则该初始值被用于初始化第一个成员。因此，first_token 的初始化过程实际上是给 cval 成员赋了一个初值。

我们使用通用的成员访问运算符访问一个 union 对象的成员：

```
last_token.cval = 'z';
pt->ival = 42;
```

为 union 的一个数据成员赋值会令其他数据成员变成未定义的状态。因此，当我们使用 union 时，必须清楚地知道当前存储在 union 中的值到底是什么类型。如果我们使用错误的数据成员或者为错误的数据成员赋值，则程序可能崩溃或出现异常行为，具体的情况根据成员的类型而有所不同。

匿名 union

匿名 union (anonymous union) 是一个未命名的 union，并且在右花括号和分号之间没有任何声明（参见 2.6.1 节，第 65 页）。一旦我们定义了一个匿名 union，编译器就自动地为该 union 创建一个未命名的对象：

```
union {                                // 匿名 union
    char cval;
    int ival;
    double dval;
}; // 定义一个未命名的对象，我们可以直接访问它的成员
cval = 'c';                            // 为刚刚定义的未命名的匿名 union 对象赋一个新值
ival = 42;                             // 该对象当前保存的值是 42
```

在匿名 union 的定义所在的作用域内该 union 的成员都是可以直接访问的。



匿名 union 不能包含受保护的成员或私有成员，也不能定义成员函数。

含有类类型成员的 union

C++ 的早期版本规定，在 union 中不能含有定义了构造函数或拷贝控制成员的类类型

成员。C++11 新标准取消了这一限制。不过，如果 union 的成员类型定义了自己的构造函数和/或拷贝控制成员，则该 union 的用法要比只含有内置类型成员的 union 复杂得多。

< 849

C++
11

当 union 包含的是内置类型的成员时，我们可以使用普通的赋值语句改变 union 保存的值。但是对于含有特殊类类型成员的 union 就没这么简单了。如果我们想将 union 的值改为类类型成员对应的值，或者将类类型成员的值改为一个其他值，则必须分别构造或析构该类类型的成员：当我们将 union 的值改为类类型成员对应的值时，必须运行该类型的构造函数；反之，当我们将类类型成员的值改为一个其他值时，必须运行该类型的析构函数。

当 union 包含的是内置类型的成员时，编译器将按照成员的次序依次合成默认构造函数或拷贝控制成员。但是如果 union 含有类类型的成员，并且该类型自定义了默认构造函数或拷贝控制成员，则编译器将为 union 合成对应的版本并将其声明为删除的（参见 13.1.6 节，第 450 页）。

例如，string 类定义了五个拷贝控制成员以及一个默认构造函数。如果 union 含有 string 类型的成员，并且没有自定义默认构造函数或某个拷贝控制成员，则编译器将合成缺少的成员并将其声明为删除的。如果在某个类中含有一个 union 成员，而且该 union 含有删除的拷贝控制成员，则该类与之对应的拷贝控制操作也将是删除的。

使用类管理 union 成员

对于 union 来说，要想构造或销毁类类型的成员必须执行非常复杂的操作，因此我们通常把含有类类型成员的 union 内嵌在另一个类当中。这个类可以管理并控制与 union 的类类型成员有关的状态转换。举个例子，我们为 union 添加一个 string 成员，并将我们的 union 定义成匿名 union，最后将它作为 Token 类的一个成员。此时，Token 类将可以管理 union 的成员。

为了追踪 union 中到底存储了什么类型的值，我们通常会定义一个独立的对象，该对象称为 union 的判别式（discriminant）。我们可以使用判别式辨认 union 存储的值。为了保持 union 与其判别式同步，我们将判别式也作为 Token 的成员。我们的类将定义一个枚举类型（参见 19.3 节，第 736 页）的成员来追踪其 union 成员的状态。

在我们的类中定义的函数包括默认构造函数、拷贝控制成员以及一组赋值运算符，这些赋值运算符可以将 union 的某种类型的值赋给 union 成员：

```
class Token {
public:
    // 因为 union 含有一个 string 成员，所以 Token 必须定义拷贝控制成员
    // 定义移动构造函数和移动赋值运算符的任务留待本节练习完成
    Token(): tok(INT), ival{0} { }
    Token(const Token &t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token&);
    // 如果 union 含有一个 string 成员，则我们必须销毁它，参见 19.1.2 节（第 729 页）
    ~Token() { if (tok == STR) sval~string(); }
    // 下面的赋值运算符负责设置 union 的不同成员
    Token &operator=(const std::string&);
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
private:
```

< 850

```

enum {INT, CHAR, DBL, STR} tok; // 判别式
union { // 匿名 union
    char    cval;
    int     ival;
    double  dval;
    std::string sval;
}; // 每个 Token 对象含有一个该未命名 union 类型的未命名成员
// 检查判别式，然后酌情拷贝 union 成员
void copyUnion(const Token&);
};

```

我们的类定义了一个嵌套的、未命名的、不限定作用域的枚举类型（参见 19.3 节，第 736 页），并将其作为 `tok` 成员的类型。其中，`tok` 的声明位于枚举类型定义的右侧花括号之后，以及表示该枚举类型定义结束的分号之前，因此，`tok` 的类型就是当前这个未命名的 `enum` 类型（参见 2.6.1 节，第 65 页）。

我们使用 `tok` 作为判别式。当 `union` 存储的是一个 `int` 值时，`tok` 的值是 `INT`；当 `union` 存储的是一个 `string` 值时，`tok` 的值是 `STR`；以此类推。

类的默认构造函数初始化判别式以及 `union` 成员，令其保存 `int` 值 0。

因为我们的 `union` 含有一个定义了析构函数的成员，所以必须为 `union` 也定义一个析构函数以销毁 `string` 成员。和普通的类类型成员不一样，作为 `union` 组成部分的类成员无法自动销毁。因为析构函数不清楚 `union` 存储的值是什么类型，所以它无法确定应该销毁哪个成员。

我们的析构函数检查被销毁的对象中是否存储着 `string` 值。如果有，则类的析构函数显式地调用 `string` 的析构函数（参见 19.1.2 节，第 729 页）释放该 `string` 使用的内存；反之，如果 `union` 存储的值是内置类型，则类的析构函数什么也不做。

管理判别式并销毁 string

类的赋值运算符将负责设置 `tok` 并为 `union` 的相应成员赋值。和析构函数一样，这些运算符在为 `union` 赋新值前必须首先销毁 `string`：

```

Token &Token::operator=(int i)
{
    if (tok == STR) sval~string(); // 如果当前存储的是 string，释放它
    ival = i;                      // 为成员赋值
    tok = INT;                     // 更新判别式
    return *this;
}

```

如果 `union` 的当前值是 `string`，则我们必须先调用 `string` 的析构函数销毁这个 `string`，然后再为 `union` 赋新值。清除了 `string` 成员之后，我们将给定的值赋给与运算符形参类型相匹配的成员。在此例中，形参类型是 `int`，所以我们赋值给 `ival`。随后更新判别式并返回结果。

851 `double` 和 `char` 版本的赋值运算符与 `int` 赋值运算符非常相似，读者可以在本节的练习中尝试使用这两个运算符。`string` 版本与其他几个有所区别，原因是 `string` 版本必须管理与 `string` 类型有关的转换：

```

Token &Token::operator=(const std::string &s)
{

```

```

    if (tok == STR)           // 如果当前存储的是 string, 可以直接赋值
        sval = s;
    else
        new(&sval) string(s); // 否则需要先构造一个 string
    tok = STR;               // 更新判别式
    return *this;
}

```

在此例中, 如果 union 当前存储的是 string, 则我们可以使用普通的 string 赋值运算符直接为其赋值。如果 union 当前存储的不是 string, 则我们找不到一个已存在的 string 对象供我们调用赋值运算符。此时, 我们必须先利用定位 new 表达式(参见 19.1.2 节, 第 729 页)在内存中为 sval 构造一个 string, 然后将该 string 初始化为 string 形参的副本, 最后更新判别式并返回结果。

管理需要拷贝控制的联合成员

和依赖于类型的赋值运算符一样, 拷贝构造函数和赋值运算符也需要先检验判别式以明确拷贝所采用的方式。为了完成这一任务, 我们定义一个名为 copyUnion 的成员。

当我们在拷贝构造函数中调用 copyUnion 时, union 成员将被默认初始化, 这意味着编译器会初始化 union 的第一个成员。因为 string 不是第一个成员, 所以显然 union 成员保存的不是 string。在赋值运算符中情况有些不一样, 有可能 union 已经存储了一个 string。我们将在赋值运算符中直接处理这种情况。copyUnion 假设如果它的形参存储了 string, 则它一定会构造自己的 string:

```

void Token::copyUnion(const Token &t)
{
    switch (t.tok) {
        case Token::INT: ival = t.ival; break;
        case Token::CHAR: cval = t.cval; break;
        case Token::DBL: dval = t.dval; break;
        // 要想拷贝一个 string 可以使用定位 new 表达式构造它, 参见 19.1.2 节(第 729 页)
        case Token::STR: new(&sval) string(t.sval); break;
    }
}

```

该函数使用一个 switch 语句(参见 5.3.2 节, 第 159 页)检验判别式。对于内置类型来说, 我们把值直接赋给对应的成员; 如果拷贝的是一个 string, 则需要构造它。

赋值运算符必须处理 string 成员的三种可能情况: 左侧运算对象和右侧运算对象都是 string、两个运算对象都不是 string、只有一个运算对象是 string:

```

Token &Token::operator=(const Token &t)
{
    // 如果此对象的值是 string 而 t 的值不是, 则我们必须释放原来的 string
    if (tok == STR && t.tok != STR) sval~string();
    if (tok == STR && t.tok == STR)
        sval = t.sval; // 无须构造一个新 string
    else
        copyUnion(t); // 如果 t.tok 是 STR, 则需要构造一个 string
    tok = t.tok;
    return *this;
}

```

如果作为左侧运算对象的 union 的值是 string 但右侧运算对象的值不是，则我们必须先释放原来的 string 再给 union 成员赋一新值。如果两侧运算对象的值都是 string，则我们可以使用普通的 string 赋值运算符完成拷贝。否则，我们调用 copyUnion 进行赋值。在 copyUnion 内部，如果右侧运算对象是 string，则我们在左侧运算对象的 union 成员内构造一个新 string；如果两端都不是 string，则直接执行普通的赋值操作就可以了。

19.6 节练习

练习 19.21：编写你自己的 Token 类。

练习 19.22：为你的 Token 类添加一个 Sales_data 类型的成员。

练习 19.23：为你的 Token 类添加移动构造函数和移动赋值运算符。

练习 19.24：如果我们将一个 Token 对象赋给它自己将发生什么情况？

练习 19.25：编写一系列赋值运算符，令其分别接受 union 中各种类型的值。

19.7 局部类

类可以定义在某个函数的内部，我们称这样的类为 **局部类** (local class)。局部类定义的类型只在定义它的作用域内可见。和嵌套类不同，局部类的成员受到严格限制。



局部类的所有成员（包括函数在内）都必须完整定义在类的内部。因此，局部类的作用与嵌套类相比相差很远。

853

在实际编程的过程中，因为局部类的成员必须完整定义在类的内部，所以成员函数的复杂性不可能太高。局部类的成员函数一般只有几行代码，否则我们就很难读懂它了。

类似的，在局部类中也不允许声明静态数据成员，因为我们没法定义这样的成员。

局部类不能使用函数作用域中的变量

局部类对其外层作用域中名字的访问权限受到很多限制，局部类只能访问外层作用域定义的类型名、静态变量（参见 6.1.1 节，第 185 页）以及枚举成员。如果局部类定义在某个函数内部，则该函数的普通局部变量不能被该局部类使用：

```
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar 是 foo 的局部类
    struct Bar {
        Loc locVal;           // 正确：使用一个局部类型名
        int barVal;
    };
    void fooBar(Loc l = a)   // 正确：默认实参是 Loc::a
    {
        barVal = val;       // 错误：val 是 foo 的局部变量
    }
}
```

```
    barVal = ::val;           // 正确：使用一个全局对象
    barVal = si;             // 正确：使用一个静态局部对象
    locVal = b;              // 正确：使用一个枚举成员
}
};

// ...
}
```

常规的访问保护规则对局部类同样适用

外层函数对局部类的私有成员没有任何访问特权。当然，局部类可以将外层函数声明为友元；或者更常见的情况是局部类将其成员声明成公有的。在程序中有权访问局部类的代码非常有限。局部类已经封装在函数作用域中，通过信息隐藏进一步封装就显得没什么必要了。

局部类中的名字查找

局部类内部的名字查找次序与其他类相似。在声明类的成员时，必须先确保用到的名字位于作用域中，然后再使用该名字。定义成员时用到的名字可以出现在类的任意位置。如果某个名字不是局部类的成员，则继续在外层函数作用域中查找；如果还没有找到，则在外层函数所在的作用域中查找。◀ 854

嵌套的局部类

可以在局部类的内部再嵌套一个类。此时，嵌套类的定义可以出现在局部类之外。不过，嵌套类必须定义在与局部类相同的作用域中。

```
void foo()
{
    class Bar {
        public:
            // ...
            class Nested; // 声明 Nested 类
    };
    // 定义 Nested 类
    class Bar::Nested {
        // ...
    };
}
```

和往常一样，当我们在类的外部定义成员时，必须指明该成员所属的作用域。因此在上面的例子中，`Bar::Nested` 的意思是 `Nested` 是定义在 `Bar` 的作用域内的一个类。

局部类内的嵌套类也是一个局部类，必须遵循局部类的各种规定。嵌套类的所有成员都必须定义在嵌套类内部。

19.8 固有的不可移植的特性

为了支持低层编程，C++ 定义了一些固有的不可移植（nonportable）的特性。所谓不可移植的特性是指因机器而异的特性，当我们把含有不可移植特性的程序从一台机器转移到另一台机器上时，通常需要重新编写该程序。算术类型的大小在不同机器上不一样（参见 2.1.1 节，第 30 页），这是我们使用过的不可移植特性的一个典型示例。

本节将介绍 C++ 从 C 语言继承而来的另外两种不可移植的特性：位域和 `volatile` 限定符。此外，我们还将介绍链接指示，它是 C++ 新增的一种不可移植的特性。

19.8.1 位域

类可以将其（非静态）数据成员定义成位域（bit-field），在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。



位域在内存中的布局是与机器相关的。

855

位域的类型必须是整型或枚举类型（参见 19.3 节，第 736 页）。因为带符号位域的行为是由具体实现确定的，所以在通常情况下我们使用无符号类型保存一个位域。位域的声明形式是在成员名字之后紧跟一个冒号以及一个常量表达式，该表达式用于指定成员所占的二进制位数：

```
typedef unsigned int Bit;
class File {
    Bit mode: 2;                      // mode 占 2 位
    Bit modified: 1;                   // modified 占 1 位
    Bit prot_owner: 3;                // prot_owner 占 3 位
    Bit prot_group: 3;                // prot_group 占 3 位
    Bit prot_world: 3;                // prot_world 占 3 位
    // File 的操作和数据成员
public:
    // 文件类型以八进制的形式表示，参见 2.1.3 节（第 35 页）
    enum modes { READ = 01, WRITE = 02, EXECUTE = 03 };
    File &open(modes);
    void close();
    void write();
    bool isRead() const;
    void setWrite();
};
```

`mode` 位域占 2 个二进制位，`modified` 只占 1 个，其他成员则各占 3 个。如果可能的话，在类的内部连续定义的位域压缩在同一整数的相邻位，从而提供存储压缩。例如在之前的声明中，五个位域可能会存储在同一个 `unsigned int` 中。这些二进制位是否能压缩到一个整数中以及如何压缩是与机器相关的。

取地址运算符（`&`）不能作用于位域，因此任何指针都无法指向类的位域。



通常情况下最好将位域设为无符号类型，存储在带符号类型中的位域的行为将因具体实现而定。

使用位域

访问位域的方式与访问类的其他数据成员的方式非常相似：

```
void File::write()
{
    modified = 1;
```

```

    // ...
}

void File::close()
{
    if (modified)
        // ..... 保存内容
}

```

通常使用内置的位运算符（参见 4.8 节，第 136 页）操作超过 1 位的位域：

```

File &File::open(File::modes m)
{
    mode |= READ;           // 按默认方式设置 READ
    // 其他处理
    if (m & WRITE)         // 如果打开了 READ 和 WRITE
        // 按照读/写方式打开文件
    return *this;
}

```

< 856

如果一个类定义了位域成员，则它通常也会定义一组内联的成员函数以检验或设置位域的值：

```

inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }

```

19.8.2 volatile 限定符



volatile 的确切含义与机器有关，只能通过阅读编译器文档来理解。要想让使用了 **volatile** 的程序在移植到新机器或新编译器后仍然有效，通常需要对该程序进行某些改变。

直接处理硬件的程序常常包含这样的数据元素，它们的值由程序直接控制之外的过程控制。例如，程序可能包含一个由系统时钟定时更新的变量。当对象的值可能在程序的控制或检测之外被改变时，应该将该对象声明为 **volatile**。关键字 **volatile** 告诉编译器不应对这样的对象进行优化。

volatile 限定符的用法和 **const** 很相似，它起到对类型额外修饰的作用：

```

volatile int display_register;      // 该 int 值可能发生改变
volatile Task *curr_task;          // curr_task 指向一个 volatile 对象
volatile int iax[max_size];        // iax 的每个元素都是 volatile
volatile Screen bitmapBuf;         // bitmapBuf 的每个成员都是 volatile

```

const 和 **volatile** 限定符互相没什么影响，某种类型可能既是 **const** 的也是 **volatile** 的，此时它同时具有二者的属性。

就像一个类可以定义 **const** 成员函数一样，它也可以将成员函数定义成 **volatile** 的。只有 **volatile** 的成员函数才能被 **volatile** 的对象调用。

2.4.2 节（第 56 页）描述了 **const** 限定符和指针的相互作用，在 **volatile** 限定符和指针之间也存在类似的关系。我们可以声明 **volatile** 指针、指向 **volatile** 对象的指针以及指向 **volatile** 对象的 **volatile** 指针：

```

volatile int v;                  // v 是一个 volatile int

```

< 857

```

int *volatile vip;    // vip 是一个 volatile 指针，它指向 int
volatile int *ivp;    // ivp 是一个指针，它指向一个 volatile int
// vivp 是一个 volatile 指针，它指向一个 volatile int
volatile int *volatile vivp;

int *ip = &v;          // 错误：必须使用指向 volatile 的指针
ivp = &v;              // 正确：ivp 是一个指向 volatile 的指针
vivp = &v;             // 正确：vivp 是一个指向 volatile 的 volatile 指针

```

和 `const` 一样，我们只能将一个 `volatile` 对象的地址（或者拷贝一个指向 `volatile` 类型的指针）赋给一个指向 `volatile` 的指针。同时，只有当某个引用是 `volatile` 的时，我们才能使用一个 `volatile` 对象初始化该引用。

合成的拷贝对 `volatile` 对象无效

`const` 和 `volatile` 的一个重要区别是我们不能使用合成的拷贝/移动构造函数及赋值运算符初始化 `volatile` 对象或从 `volatile` 对象赋值。合成的成员接受的形参类型是（非 `volatile`）常量引用，显然我们不能把一个非 `volatile` 引用绑定到一个 `volatile` 对象上。

如果一个类希望拷贝、移动或赋值它的 `volatile` 对象，则该类必须自定义拷贝或移动操作。例如，我们可以将形参类型指定为 `const volatile` 引用，这样我们就能利用任意类型的 `Foo` 进行拷贝或赋值操作了：

```

class Foo {
public:
    Foo(const volatile Foo&); // 从一个 volatile 对象进行拷贝
    // 将一个 volatile 对象赋值给一个非 volatile 对象
    Foo& operator=(volatile const Foo&);
    // 将一个 volatile 对象赋值给一个 volatile 对象
    Foo& operator=(volatile const Foo&) volatile;
    // Foo 类的剩余部分
};

```

尽管我们可以为 `volatile` 对象定义拷贝和赋值操作，但是一个更深层次的问题是拷贝 `volatile` 对象是否有意义呢？不同程序使用 `volatile` 的目的各不相同，对上述问题的回答与具体的使用目的密切相关。

19.8.3 链接指示：`extern "C"`

C++程序有时需要调用其他语言编写的函数，最常见的是调用 C 语言编写的函数。像所有其他名字一样，其他语言中的函数名字也必须在 C++中进行声明，并且该声明必须指定返回类型和形参列表。对于其他语言编写的函数来说，编译器检查其调用的方式与处理普通 C++函数的方式相同，但是生成的代码有所区别。C++使用链接指示（linkage directive）指出任意非 C++函数所用的语言。



要想把 C++代码和其他语言（包括 C 语言）编写的代码放在一起使用，要求我们必须有权访问该语言的编译器，并且这个编译器与当前的 C++编译器是兼容的。

声明一个非 C++ 的函数

链接指示可以有两种形式：单个的或复合的。链接指示不能出现在类定义或函数定义的内部。同样的链接指示必须在函数的每个声明中都出现。

举个例子，接下来的声明显示了 `cstring` 头文件的某些 C 函数是如何声明的：

```
// 可能出现在 C++头文件<cstring>中的链接指示
// 单语句链接指示
extern "C" size_t strlen(const char *);
// 复合语句链接指示
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

链接指示的第一种形式包含一个关键字 `extern`，后面是一个字符串字面值常量以及一个“普通的”函数声明。

其中的字符串字面值常量指出了编写函数所用的语言。编译器应该支持对 C 语言的链接指示。此外，编译器也可能会支持其他语言的链接指示，如 `extern "Ada"`、`extern "FORTRAN"` 等。

链接指示与头文件

我们可以令链接指示后面跟上花括号括起来的若干函数的声明，从而一次性建立多个链接。花括号的作用是将适用于该链接指示的多个声明聚合在一起，否则花括号就会被忽略，花括号中声明的函数名字就是可见的，就好像在花括号之外声明的一样。

多重声明的形式可以应用于整个头文件。例如，C++ 的 `cstring` 头文件可能形如：

```
// 复合语句链接指示
extern "C" {
#include <string.h>           // 操作 C 风格字符串的 C 函数
}
```

当一个 `#include` 指示被放置在复合链接指示的花括号中时，头文件中的所有普通函数声明都被认为是由链接指示的语言编写的。链接指示可以嵌套，因此如果头文件包含带自带链接指示的函数，则该函数的链接不受影响。



C++从 C 语言继承的标准库函数可以定义成 C 函数，但并非必须：决定使用 C 还是 C++实现 C 标准库，是每个 C++实现的事情。

<859

指向 `extern "C"` 函数的指针

编写函数所用的语言是函数类型的一部分。因此，对于使用链接指示定义的函数来说，它的每个声明都必须使用相同的链接指示。而且，指向其他语言编写的函数的指针必须与函数本身使用相同的链接指示：

```
// pf 指向一个 C 函数，该函数接受一个 int 返回 void
extern "C" void (*pf)(int);
```

当我们使用 `pf` 调用函数时，编译器认定当前调用的是一个 C 函数。

指向 C 函数的指针与指向 C++ 函数的指针是不一样的类型。一个指向 C 函数的指针

不能用在执行初始化或赋值操作后指向 C++ 函数，反之亦然。就像其他类型不匹配的问题一样，如果我们试图在两个链接指示不同的指针之间进行赋值操作，则程序将发生错误：

```
void (*pf1)(int); // 指向一个 C++ 函数
extern "C" void (*pf2)(int); // 指向一个 C 函数
pf1 = pf2; // 错误：pf1 和 pf2 的类型不同
```



有的 C++ 编译器会接受之前的这种赋值操作并将其作为对语言的扩展，尽管从严格意义上来看它是非法的。

链接指示对整个声明都有效

当我们使用链接指示时，它不仅对函数有效，而且对作为返回类型或形参类型的函数指针也有效：

```
// f1 是一个 C 函数，它的形参是一个指向 C 函数的指针
extern "C" void f1(void(*)(int));
```

这条声明语句指出 f1 是一个不返回任何值的 C 函数。它有一个类型是函数指针的形参，其中的函数接受一个 int 形参返回为空。这个链接指示不仅对 f1 有效，对函数指针同样有效。当我们调用 f1 时，必须传给它一个 C 函数的名字或者指向 C 函数的指针。

因为链接指示同时作用于声明语句中的所有函数，所以如果我们希望给 C++ 函数传入一个指向 C 函数的指针，则必须使用类型别名（参见 2.5.1 节，第 60 页）：

```
860 // f1 是一个指向 C 函数的指针
extern "C" typedef void FC(int);
// f2 是一个 C++ 函数，该函数的形参是指向 C 函数的指针
void f2(FC *);
```

导出 C++ 函数到其他语言

通过使用链接指示对函数进行定义，我们可以令一个 C++ 函数在其他语言编写的程序中可用：

```
// calc 函数可以被 C 程序调用
extern "C" double calc(double dparam) { /* ... */ }
```

编译器将为该函数生成适合于指定语言的代码。

值得注意的是，可被多种语言共享的函数的返回类型或形参类型受到很多限制。例如，我们不太可能把一个 C++ 类的对象传给 C 程序，因为 C 程序根本无法理解构造函数、析构函数以及其他类特有的操作。

对链接到 C 的预处理器的支持

有时需要在 C 和 C++ 中编译同一个源文件，为了实现这一目的，在编译 C++ 版本的程序时预处理器定义 `__cplusplus`（两个下画线）。利用这个变量，我们可以在编译 C++ 程序的时候有条件地包含进来一些代码：

```
#ifdef __cplusplus
// 正确：我们正在编译 C++ 程序
extern "C"
#endif
int strcmp(const char*, const char*);
```

重载函数与链接指示

链接指示与重载函数的相互作用依赖于目标语言。如果目标语言支持重载函数，则为该语言实现链接指示的编译器很可能也支持重载这些 C++ 的函数。

C 语言不支持函数重载，因此也就不难理解为什么一个 C 链接指示只能用于说明一组重载函数中的某一个了：

```
// 错误：两个 extern "C" 函数的名字相同
extern "C" void print(const char*);
extern "C" void print(int);
```

如果在一组重载函数中有一个是 C 函数，则其余的必定都是 C++ 函数：

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };
// C 函数可以在 C 或 C++ 程序中调用
// C++ 函数重载了该函数，可以在 C++ 程序中调用
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

861

C 版本的 calc 函数可以在 C 或 C++ 程序中调用，而使用了类类型形参的 C++ 函数只能在 C++ 程序中调用。上述性质与声明的顺序无关。

19.8.3 节练习

练习 19.26：说明下列声明语句的含义并判断它们是否合法：

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

862 小结

C++为解决某些特殊问题设置了一系列特殊的处理机制。

有的程序需要精确控制内存分配过程，它们可以通过在类的内部或在全局作用域中自定义 `operator new` 和 `operator delete` 来实现这一目的。如果应用程序为这两个操作定义了自己的版本，则 `new` 和 `delete` 表达式将优先使用应用程序定义的版本。

有的程序需要在运行时直接获取对象的动态类型，运行时类型识别（RTTI）为这种程序提供了语言级别的支持。RTTI 只对定义了虚函数的类有效；对没有定义虚函数的类，虽然也可以得到其类型信息，但只是静态类型。

当我们定义指向类成员的指针时，在指针类型中包含了该指针所指成员所属类的类型信息。成员指针可以绑定到该类当中任意一个具有指定类型的成员上。当我们解引用成员指针时，必须提供获取成员所需的对象。

C++定义了另外几种聚集类型：

- 嵌套类，定义在其他类的作用域中，嵌套类通常作为外层类的实现类。
- `union`，是一种特殊的类，它可以定义几个数据成员但是在任意时刻只有一个成员有值，`union` 通常嵌套在其他类的内部。
- 局部类，定义在函数的内部，局部类的所有成员都必须定义在类内，局部类不能含有静态数据成员。

C++支持几种固有的不可移植的特性，其中位域和 `volatile` 使得程序更容易访问硬件；链接指示使得程序更容易访问用其他语言编写的代码。

术语表

匿名 union (anonymous union) 未命名的 `union`，不能用于定义对象。匿名 `union` 的成员也是外层作用域的成员。匿名 `union` 不能包含成员函数，也不能包含私有成员或受保护的成员。

位域 (bit-field) 特殊的类成员，该成员含有一个整型值以指定为其分配的二进制位数。如果可能的话，在类中连续定义的位域将被压缩在一个普通的整数值当中。

判别式 (discriminant) 是一种使用一个对象判断 `union` 的当前值类型的编程技术。

dynamic_cast 是一个运算符，执行从基类向派生类的带检查的强制类型转换。当基类中至少含有一个虚函数时，该运算符负责检查指针或引用所绑定的对象的动态类型。如果对象类型与目标类型（或其派生类）一致，则类型转换完成。否则，指针

转换将返回一个值为 0 的指针；引用转换将抛出一个异常。

枚举类型 (enumeration) 将一组整型常量命名后聚合在一起形成的类型。

枚举成员 (enumerator) 是枚举类型的成员。枚举成员是常量，可以用在任何需要整型常量的地方。

free 是定义在 `cstdlib` 中的低层函数，负责释放内存。`free` 只能释放由 `malloc` 分配的内存。

链接指示 (linkage directive) 支持 C++ 程序调用其他语言编写的函数的一种机制。所有编译器都应支持调用 C++ 和 C 函数，至于是否支持其他语言则由编译器决定。

局部类 (local class) 定义在函数中的类。局部类只有在其外层函数内可见。局部类

的所有成员都必须定义在类的内部。局部类不能含有静态成员。局部类成员不能访问外层函数的非静态变量，只能访问类型名字、静态变量或枚举成员。

malloc 是定义在 `cstdlib` 中的低层函数，负责分配内存。**malloc** 分配的内存必须由 `free` 释放。

mem_fn 是一个标准库类模板，根据指向成员函数的指针生成一个可调用对象。

嵌套类 (nested class) 定义在其他类内部的类，嵌套类定义在它的外层作用域中：在外层类的作用域中嵌套类的名字必须唯一，在外层类之外可以被重用。在外层类之外访问嵌套类需要用作用域运算符指明嵌套类所属的范围。

嵌套类型 (nested type) “嵌套类”的同义词。

不可移植 (nonportable) 固有的与机器有关的特性，当程序转移到其他机器或编译器上时需要修改代码。

operator delete 是一个标准库函数，用于释放由 `operator new` 分配的未指明类型的、未构造的内存空间。相应的，`operator delete[]` 释放由 `operator new[]` 为数组分配的内存。

operator new 是一个标准库函数，用于分配一个给定大小的、未指明类型的、未构造的内存空间。标准库函数 `operator new[]` 为数组分配原始内存。与 `allocator` 类相比，这两个标准库函数提供的内存分配机制更低级。现代的 C++ 程序应该使用 `allocator` 而不是这两个函数。

定位 new 表达式 (placement new expression) 是 `new` 的一种特殊形式，在给定的内存中构造对象。它不分配内存，而是根据实参指定在哪儿构造对象。它是对 `allocator` 类的 `construct` 成员的行为的一种低级模拟。

成员指针 (pointer to member) 其中既包含类类型，也包含指针所指的成员类型。

成员指针的定义必须同时指定类的名字以及指针所指的成员类型：

```
T C::*pmem = &C::member;
```

该语句将 `pmem` 定义为一个指针，它可以指向类 `C` 的成员，并且该成员的类型是 `T`，然后初始化 `pmem` 令其指向类 `C` 的名为 `member` 的成员。要使用该指针，我们必须提供 `C` 的一个对象或指针：

```
classobj.*pmem;
```

```
classptr->*pmem;
```

从 `classptr` 所指的对象 `classobj` 中获取 `member`。 ◀ 864

运行时类型识别 (run-time type identification) 是 C++ 的一种特性，允许在运行时获取指针或引用的动态类型。RTTI 运算符包括 `typeid` 和 `dynamic_cast`，为含有虚函数的类的指针或引用提供动态类型。当作用于其他类型时，返回的结果是指针或引用的静态类型。

限定作用域的枚举类型 (scoped enumeration) 是一种新的枚举类型，它的枚举成员不能被外层作用域直接访问。

typeid 运算符 (typeid operator) 是一个一元运算符，返回标准库类型 `type_info` 的引用，表示给定表达式的类型。当表达式是某个含有虚函数的类型的对象时，返回表达式的动态类型；此类表达式在运行时求值。如果表达式的类型是指针、引用或其他未定义虚函数的类型，则返回指针、引用或对象的静态类型；此类表达式不会被求值。

type_info typeid 运算符返回的标准库类型。`type_info` 的细节因机器而异，但是必须提供一组操作，其中名为 `name` 的函数负责返回一个表示类型名字的字符串。`type_info` 对象不能被拷贝、移动或赋值。

联合 (union) 是一种和类有些相似的类型。可以包含多个数据成员，但是同一时刻只能有一个成员有值。联合可以有包括

构造函数和析构函数在内的成员函数。联合不能被用作基类。在 C++11 新标准中，联合可以含有类类型的成员，前提是这些类自定义了拷贝控制成员。对于这样的联合来说，如果它们没有定义自己的拷贝控制成员，则编译器将为它们生成删除的版本。

不限定作用域的枚举类型（unscoped enumeration） 该枚举类型的枚举成员在枚举类型的外层作用域中可以访问。

volatile 是一种类型限定符，告诉编译器变量可能在程序的直接控制之外发生改变。它起到一种标示的作用，令编译器不对代码进行优化操作。

附录 A

标准库

内容

A.1 标准库名字和头文件	766
A.2 算法概览	770
A.3 随机数	781

本附录介绍了标准库中算法和随机数部分的一些额外细节。这里还提供了一个我们使用过的所有标准库名字的列表，列表中给出了每个名字所在的头文件。

在第 10 章中我们使用过一些较常用的算法，并且描述了算法之下的架构。在本附录中，我们将列出所有标准库算法，按它们执行的操作的种类来组织。

在 17.4 节（第 660 页）中我们描述了随机数库的架构，并使用了几个分布类型。库中定义了若干随机数引擎和 20 种不同的分布。在本附录中，我们将列出所有引擎和分布类型。

866 A.1 标准库名字和头文件

本书中大多数代码没有给出编译程序所需的实际#*include* 指令。为了方便读者，表A.1列出了本书程序用到的标准库名字以及它们所在的头文件。

表 A.1：标准库名字和头文件

名字	头文件
abort	<cstdlib>
accumulate	<numeric>
allocator	<memory>
array	<array>
auto_ptr	<memory>
back_inserter	<iterator>
bad_alloc	<new>
bad_array_new_length	<new>
bad_cast	<typeinfo>
begin	<iterator>
bernoulli_distribution	<random>
bind	<functional>
bitset	<bitset>
boolalpha	<iostream>
cerr	<iostream>
cin	<iostream>
cmatch	<regex>
copy	<algorithm>
count	<algorithm>
count_if	<algorithm>
cout	<iostream>
cref	<functional>
csub_match	<regex>
dec	<iostream>
default_float_engine	<iostream>
default_random_engine	<random>
deque	<deque>
domain_error	<stdexcept>
end	<iterator>
endl	<iostream>
ends	<iostream>
equal_range	<algorithm>
exception	<exception>
fill	<algorithm>
fill_n	<algorithm>
find	<algorithm>

续表

名字	头文件
find_end	<algorithm>
find_first_of	<algorithm>
find_if	<algorithm>
fixed	<iostream>
flush	<iostream>
for_each	<algorithm>
forward	<utility>
forward_list	<forward_list>
free	<cstdlib>
front_inserter	<iterator>
fstream	<fstream>
function	<functional>
get	<tuple>
getline	<string>
greater	<functional>
hash	<functional>
hex	<iostream>
hexfloat	<iostream>
ifstream	<fstream>
initializer_list	<initializer_list>
inserter	<iterator>
internal	<iostream>
ios_base	<ios_base>
isalpha	<cctype>
islower	<cctype>
isprint	<cctype>
ispunct	<cctype>
isspace	<cctype>
istream	<iostream>
istream_iterator	<iterator>
istringstream	<sstream>
isupper	<cctype>
left	<iostream>
less	<functional>
less_equal	<functional>
list	<list>
logic_error	<stdexcept>
lower_bound	<algorithm>
lround	<cmath>
make_move_iterator	<iterator>
make_pair	<utility>

867

续表

868

名字	头文件
make_shared	<memory>
make_tuple	<tuple>
malloc	<cstdlib>
map	<map>
max	<algorithm>
max_element	<algorithm>
mem_fn	<functional>
min	<algorithm>
move	<utility>
multimap	<map>
multiset	<set>
negate	<functional>
noboolalpha	<iostream>
normal_distribution	<random>
noshowbase	<iostream>
noshowpoint	<iostream>
noskipws	<iostream>
not1	<functional>
nothrow	<new>
nothrow_t	<new>
nounitbuf	<iostream>
nouppercase	<iostream>
nth_element	<algorithm>
oct	<iostream>
ofstream	<fstream>
ostream	<iostream>
ostream_iterator	<iterator>
ostringstream	<sstream>
out_of_range	<stdexcept>
pair	<utility>
partial_sort	<algorithm>
placeholders	<functional>
placeholders::_1	<functional>
plus	<functional>
priority_queue	<queue>
ptrdiff_t	<cstddef>
queue	<queue>
rand	<random>
random_device	<random>
range_error	<stdexcept>
ref	<functional>

续表

名字	头文件
regex	<regex>
regex_constants	<regex>
regex_error	<regex>
regex_match	<regex>
regex_replace	<regex>
regex_search	<regex>
remove_pointer	<type_traits>
remove_reference	<type_traits>
replace	<algorithm>
replace_copy	<algorithm>
reverse_iterator	<iterator>
right	<iostream>
runtime_error	<stdexcept>
scientific	<iostream>
set	<set>
set_difference	<algorithm>
set_intersection	<algorithm>
set_union	<algorithm>
setfill	<iomanip>
setprecision	<iomanip>
setw	<iomanip>
shared_ptr	<memory>
showbase	<iostream>
showpoint	<iostream>
size_t	<cstddef>
skipws	<iostream>
smatch	<regex>
sort	<algorithm>
sqrt	<cmath>
sregex_iterator	<regex>
ssub_match	<regex>
stable_sort	<algorithm>
stack	<stack>
stoi	<string>
strcmp	<cstring>
strcpy	<cstring>
string	<string>
stringstream	<sstream>
strlen	<cstring>
strncpy	<cstring>
strtod	<string>

续表

870

名字	头文件
swap	<utility>
terminate	<exception>
time	<ctime>
tolower	<cctype>
toupper	<cctype>
transform	<algorithm>
tuple	<tuple>
tuple_element	<tuple>
tuple_size	<tuple>
type_info	<typeinfo>
unexpected	<exception>
uniform_int_distribution	<random>
uniform_real_distribution	<random>
uninitialized_copy	<memory>
uninitialized_fill	<memory>
unique	<algorithm>
unique_copy	<algorithm>
unique_ptr	<memory>
unitbuf	<iostream>
unordered_map	<unordered_map>
unordered_multimap	<unordered_map>
unordered_multiset	<unordered_set>
unordered_set	<unordered_set>
upper_bound	<algorithm>
uppercase	<iostream>
vector	<vector>
weak_ptr	<memory>

A.2 算法概览

标准库定义了超过 100 个算法。要想高效使用这些算法需要了解它们的结构而不是单纯记忆每个算法的细节。因此，我们在第 10 章中关注标准库算法架构的描述和理解。在本节中，我们将简要描述每个算法，在下面的描述中，

- beg 和 end 是表示元素范围的迭代器（参见 9.2.1 节，第 296 页）。几乎所有算法都对一个由 beg 和 end 表示的序列进行操作。
- beg2 是表示第二个输入序列开始位置的迭代器。end2 表示第二个序列的末尾位置（如果有的话）。如果没有 end2，则假定 beg2 表示的序列与 beg 和 end 表示的序列一样大。beg 和 beg2 的类型不必匹配，但是，必须保证对两个序列中的元素都可以执行特定操作或调用给定的可调用对象。
- dest 是表示目的序列的迭代器。对于给定输入序列，算法需要生成多少元素，目的序列必须保证能保存同样多的元素。

- `unaryPred` 和 `binaryPred` 是一元和二元谓词（参见 10.3.1 节，第 344 页），分别接受一个和两个参数，都是来自输入序列的元素，两个谓词都返回可用作条件的类型。
- `comp` 是一个二元谓词，满足关联容器中对关键字序的要求（参见 11.2.2 节，第 378 页）。
- `unaryOp` 和 `binaryOp` 是可调用对象（参见 10.3.2 节，第 346 页），可分别使用来自输入序列的一个和两个实参来调用。

A.2.1 查找对象的算法

这些算法在一个输入序列中搜索一个指定值或一个值的序列。

每个算法都提供两个重载的版本，第一个版本使用底层类型的相等运算符（`==`）来比较元素；第二个版本使用用户给定的 `unaryPred` 和 `binaryPred` 比较元素。

简单查找算法

这些算法查找指定值，要求输入迭代器（input iterator）。

```
find(beg, end, val)
find_if(beg, end, unaryPred)
find_if_not(beg, end, unaryPred)
count(beg, end, val)
count_if(beg, end, unaryPred)
```

`find` 返回一个迭代器，指向输入序列中第一个等于 `val` 的元素。

`find_if` 返回一个迭代器，指向第一个满足 `unaryPred` 的元素。

`find_if_not` 返回一个迭代器，指向第一个令 `unaryPred` 为 `false` 的元素。上述三个算法在未找到元素时都返回 `end`。

`count` 返回一个计数器，指出 `val` 出现了多少次；`count_if` 统计有多少个元素满足 `unaryPred`。

```
all_of(beg, end, unaryPred)
any_of(beg, end, unaryPred)
none_of(beg, end, unaryPred)
```

这些算法都返回一个 `bool` 值，分别指出 `unaryPred` 是否对所有元素都成功、对任意一个元素成功以及对所有元素都不成功。如果序列为空，`any_of` 返回 `false`，而 `all_of` 和 `none_of` 返回 `true`。

查找重复值的算法

下面这些算法要求前向迭代器（forward iterator），在输入序列中查找重复元素。

```
adjacent_find(beg, end)
adjacent_find(beg, end, binaryPred)
```

返回指向第一对相邻重复元素的迭代器。如果序列中无相邻重复元素，则返回 `end`。

```
search_n(beg, end, count, val)
search_n(beg, end, count, val, binaryPred)
```

返回一个迭代器，从此位置开始有 `count` 个相等元素。如果序列中不存在这样的子序列，

则返回 end。

872 ◀ 查找子序列的算法

在下面的算法中，除了 `find_first_of` 之外，都要求两个前向迭代器。`find_first_of` 用输入迭代器表示第一个序列，用前向迭代器表示第二个序列。这些算法搜索子序列而不是单个元素。

```
search(beg1, end1, beg2, end2)
search(beg1, end1, beg2, end2, binaryPred)
```

返回第二个输入范围（子序列）在第一个输入范围中第一次出现的位置。如果未找到子序列，则返回 `end1`。

```
find_first_of(beg1, end1, beg2, end2)
find_first_of(beg1, end1, beg2, end2, binaryPred)
```

返回一个迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置。如果未找到匹配元素，则返回 `end1`。

```
find_end(beg1, end1, beg2, end2)
find_end(beg1, end1, beg2, end2, binaryPred)
```

类似 `search`，但返回的是最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围中未找到它，则返回 `end1`。

A.2.2 其他只读算法

这些算法要求前两个实参都是输入迭代器。

`equal` 和 `mismatch` 算法还接受一个额外的输入迭代器，表示第二个范围的开始位置。这两个算法都提供两个重载的版本。第一个版本使用底层类型的相等运算符（`==`）比较元素，第二个版本则用用户指定的 `unaryPred` 或 `binaryPred` 比较元素。

```
for_each(beg, end, unaryOp)
```

对输入序列中的每个元素应用可调用对象（参见 10.3.2 节，第 346 页）`unaryOp`。`unaryOp` 的返回值（如果有的话）被忽略。如果迭代器允许通过解引用运算符向序列中的元素写入值，则 `unaryOp` 可能修改元素。

```
mismatch(beg1, end1, beg2)
mismatch(beg1, end1, beg2, binaryPred)
```

比较两个序列中的元素。返回一个迭代器的 `pair`（参见 11.2.3 节，第 379 页），表示两个序列中第一个不匹配的元素。如果所有元素都匹配，则返回的 `pair` 中第一个迭代器为 `end1`，第二个迭代器指向 `beg2` 中偏移量等于第一个序列长度的位置。

```
equal(beg1, end1, beg2)
equal(beg1, end1, beg2, binaryPred)
```

确定两个序列是否相等。如果输入序列中每个元素都与从 `beg2` 开始的序列中对应元素相等，则返回 `true`。

873 ◀ A.2.3 二分搜索算法

这些算法都要求前向迭代器，但这些算法都经过了优化，如果我们提供随机访问迭代

器 (random-access iterator) 的话, 它们的性能会好得多。从技术上讲, 无论我们提供什么类型的迭代器, 这些算法都会执行对数次的比较操作。但是, 当使用前向迭代器时, 这些算法必须花费线性次数的迭代器操作来移动到序列中要比较的元素。

这些算法要求序列中的元素已经是有序的。它们的行为类似关联容器的同名成员 (参见 11.3.5 节, 第 389 页)。`equal_range`、`lower_bound` 和 `upper_bound` 算法返回迭代器, 指向给定元素在序列中的正确插入位置——插入后还能保持有序。如果给定元素比序列中的所有元素都大, 则会返回尾后迭代器。

每个算法都提供两个版本: 第一个版本用元素类型的小于运算符 (`<`) 来检测元素; 第二个版本则使用给定的比较操作。在下列算法中, “`x 小于 y`” 表示 `x < y` 或 `comp(x, y)` 成功。

```
lower_bound(beg, end, val)
lower_bound(beg, end, val, comp)
```

返回一个迭代器, 表示第一个小于等于 `val` 的元素, 如果不存在这样的元素, 则返回 `end`。

```
upper_bound(beg, end, val)
upper_bound(beg, end, val, comp)
```

返回一个迭代器, 表示第一个大于 `val` 的元素, 如果不存在这样的元素, 则返回 `end`。

```
equal_range(beg, end, val)
equal_range(beg, end, val, comp)
```

返回一个 `pair` (参见 11.2.3 节, 第 379 页), 其 `first` 成员是 `lower_bound` 返回的迭代器, `second` 成员是 `upper_bound` 返回的迭代器。

```
binary_search(beg, end, val)
binary_search(beg, end, val, comp)
```

返回一个 `bool` 值, 指出序列中是否包含等于 `val` 的元素。对于两个值 `x` 和 `y`, 当 `x` 不小于 `y` 且 `y` 也不小于 `x` 时, 认为它们相等。

A.2.4 写容器元素的算法

很多算法向给定序列中的元素写入新值。这些算法可以从不同角度加以区分: 通过表示输入序列的迭代器类型来区分; 或者通过是写入输入序列中元素还是写入给定目的位置来区分。

只写不读元素的算法

874

这些算法要求一个输出迭代器 (output iterator), 表示目的位置。`_n` 结尾的版本接受第二个实参, 表示写入的元素数目, 并将给定数目的元素写入到目的位置中。

```
fill(beg, end, val)
fill_n(dest, cnt, val)
generate(beg, end, Gen)
generate_n(dest, cnt, Gen)
```

给输入序列中每个元素赋予一个新值。`fill` 将值 `val` 赋予元素; `generate` 执行生成器对象 `Gen()` 生成新值。生成器是一个可调用对象 (参见 10.3.2 节, 第 346 页), 每次调用会生成一个不同的返回值。`fill` 和 `generate` 都返回 `void`。`_n` 版本返回一个迭代器, 指向写入到输出序列的最后一个元素之后的位置。

使用输入迭代器的写算法

这些算法读取一个输入序列，将值写入到一个输出序列中。它们要求一个名为 dest 的输出迭代器，而表示输入范围的迭代器必须是输入迭代器。

```
copy(beg, end, dest)
copy_if(beg, end, dest, unaryPred)
copy_n(beg, n, dest)
```

从输入范围将元素拷贝到 dest 指定的目的序列。copy 拷贝所有元素，copy_if 拷贝那些满足 unaryPred 的元素，copy_n 拷贝前 n 个元素。输入序列必须有至少 n 个元素。

```
move(beg, end, dest)
```

对输入序列中的每个元素调用 std:: move (参见 13.6.1 节，第 472 页)，将其移动到迭代器 dest 开始的序列中。

```
transform(beg, end, dest, unaryOp)
transform(beg, end, beg2, dest, binaryOp)
```

调用给定操作，并将结果写到 dest 中。第一个版本对输入范围内每个元素应用一元操作。第二个版本对两个输入序列中的元素应用二元操作。

```
replace_copy(beg, end, dest, old_val, new_val)
replace_copy_if(beg, end, dest, unaryPred, new_val)
```

将每个元素拷贝到 dest，将指定的元素替换为 new_val。第一个版本替换那些 ==old_val 的元素。第二个版本替换那些满足 unaryPred 的元素。

```
merge(beg1, end1, beg2, end2, dest)
merge(beg1, end1, beg2, end2, dest, comp)
```

两个输入序列必须都是有序的。将合并后的序列写入到 dest 中。第一个版本用<运算符比较元素；第二个版本则使用给定比较操作。

875 使用前向迭代器的写算法

这些算法要求前向迭代器，由于它们是向输入序列写入元素，迭代器必须具有写入元素的权限。

```
iter_swap(iter1, iter2)
swap_ranges(beg1, end1, beg2)
```

交换 iter1 和 iter2 所表示的元素，或将输入范围内所有元素与 beg2 开始的第二个序列中所有元素进行交换。两个范围不能有重叠。iter_swap 返回 void，swap_ranges 返回递增后的 beg2，指向最后一个交换元素之后的位置。

```
replace(beg, end, old_val, new_val)
replace_if(beg, end, unaryPred, new_val)
```

用 new_val 替换每个匹配元素。第一个版本使用==比较元素与 old_val，第二个版本替换那些满足 unaryPred 的元素。

使用双向迭代器的写算法

这些算法需要在序列中有反向移动的能力，因此它们要求双向迭代器 (bidirectional iterator)。

```
copy_backward(beg, end, dest)
```

move_backward(beg, end, dest)

从输入范围中拷贝或移动元素到指定目的位置。与其他算法不同，`dest` 是输出序列的尾后迭代器（即，目的序列恰在 `dest` 之前结束）。输入范围中的尾元素被拷贝或移动到目的序列的尾元素，然后是倒数第二个元素被拷贝/移动，依此类推。元素在目的序列中的顺序与在输入序列中相同。如果范围为空，则返回值为 `dest`；否则，返回值表示从 `*beg` 中拷贝或移动的元素。

inplace_merge(beg, mid, end)
inplace_merge(beg, mid, end, comp)

将同一个序列中的两个有序子序列合并为单一的有序序列。`beg` 到 `mid` 间的子序列和 `mid` 到 `end` 间的子序列被合并，并被写入到原序列中。第一个版本使用<比较元素，第二个版本使用给定的比较操作，返回 `void`。

A.2.5 划分与排序算法

对于序列中的元素进行排序，排序和划分算法提供了多种策略。

每个排序和划分算法都提供稳定和不稳定版本（参见 10.3.1 节，第 345 页）。稳定算法保证保持相等元素的相对顺序。由于稳定算法会做更多工作，可能比不稳定版本慢得多并消耗更多内存。

划分算法

< 876

一个划分算法将输入范围中的元素划分为两组。第一组包含那些满足给定谓词的元素，第二组则包含不满足谓词的元素。例如，对于一个序列中的元素，我们可以根据元素是否是奇数或者单词是否以大写字母开头等来划分它们。这些算法都要求双向迭代器。

is_partitioned(beg, end, unaryPred)

如果所有满足谓词 `unaryPred` 的元素都在不满足 `unaryPred` 的元素之前，则返回 `true`。若序列为空，也返回 `true`。

partition_copy(beg, end, dest1, dest2, unaryPred)

将满足 `unaryPred` 的元素拷贝到 `dest1`，并将不满足 `unaryPred` 的元素拷贝到 `dest2`。返回一个迭代器 `pair` (11.2.3 节，第 379 页)，其 `first` 成员表示拷贝到 `dest1` 的元素的末尾，`second` 表示拷贝到 `dest2` 的元素的末尾。输入序列与两个目的序列都不能重叠。

partition_point(beg, end, unaryPred)

输入序列必须是已经用 `unaryPred` 划分过的。返回满足 `unaryPred` 的范围的尾后迭代器。如果返回的迭代器不是 `end`，则它指向的元素及其后的元素必须都不满足 `unaryPred`。

stable_partition(beg, end, unaryPred)
partition(beg, end, unaryPred)

使用 `unaryPred` 划分输入序列。满足 `unaryPred` 的元素放置在序列开始，不满足的元素放在序列尾部。返回一个迭代器，指向最后一个满足 `unaryPred` 的元素之后的位置，如果所有元素都不满足 `unaryPred`，则返回 `beg`。

排序算法

这些算法要求随机访问迭代器。每个排序算法都提供两个重载的版本。一个版本用元

素的`<`运算符来比较元素，另一个版本接受一个额外参数来指定排序关系（11.2.2 节，第 378 页）。`partial_sort_copy` 返回一个指向目的位置的迭代器，其他排序算法都返回 `void`。

`partial_sort` 和 `nth_element` 算法都只进行部分排序工作，它们常用于不需要排序整个序列の場合。由于这些算法工作量更少，它们通常比排序整个输入序列的算法更快。

```
sort(beg, end)
stable_sort(beg, end)
sort(beg, end, comp)
stable_sort(beg, end, comp)
```

排序整个范围。

877 ➤ `is_sorted(beg, end)`
`is_sorted(beg, end, comp)`
`is_sorted_until(beg, end)`
`is_sorted_until(beg, end, comp)`

`is_sorted` 返回一个 `bool` 值，指出整个输入序列是否有序。`is_sorted_until` 在输入序列中查找最长初始有序子序列，并返回子序列的尾后迭代器。

```
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
```

排序 `mid-beg` 个元素。即，如果 `mid-beg` 等于 42，则此函数将值最小的 42 个元素有序放在序列前 42 个位置。当 `partial_sort` 完成后，从 `beg` 开始直至 `mid` 之前的范围中的元素就都已排好序了。已排序范围中的元素都不会比 `mid` 后的元素更大。未排序区域中元素的顺序是未指定的。

```
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
```

排序输入范围中的元素，并将足够多的已排序元素放到 `destBeg` 和 `destEnd` 所指示的序列中。如果目的范围的大小大于等于输入范围，则排序整个输入序列并存入从 `destBeg` 开始的范围。如果目的范围大小小于输入范围，则只拷贝输入序列中与目的范围一样多的元素。

算法返回一个迭代器，指向目的范围中已排序部分的尾后迭代器。如果目的序列的大小小于或等于输入范围，则返回 `destEnd`。

```
nth_element(beg, nth, end)
nth_element(beg, nth, end, comp)
```

参数 `nth` 必须是一个迭代器，指向输入序列中的一个元素。执行 `nth_element` 后，此迭代器指向的元素恰好是整个序列排好序后此位置上的值。序列中的元素会围绕 `nth` 进行划分：`nth` 之前的元素都小于等于它，而之后的元素都大于等于它。

A.2.6 通用重排操作

这些算法重排输入序列中元素的顺序。前两个算法 `remove` 和 `unique`，会重排序列，使得排在序列第一部分的元素满足某种标准。它们返回一个迭代器，标记子序列的末尾。其他算法，如 `reverse`、`rotate` 和 `random_shuffle` 都重排整个序列。

这些算法的基本版本都进行“原址”操作，即，在输入序列自身内部重排元素。三个

重排算法提供“拷贝”版本。这些_copy 版本完成相同的重排工作，但将重排后的元素写入到一个指定目的序列中，而不是改变输入序列。这些算法要求输出迭代器来表示目的序列。

使用前向迭代器的重排算法

< 878

这些算法重排输入序列。它们要求迭代器至少是前向迭代器。

```
remove(beg, end, val)
remove_if(beg, end, unaryPred)
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred)
```

从序列中“删除”元素，采用的办法是用保留的元素覆盖要删除的元素。被删除的是那些`==val` 或满足 `unaryPred` 的元素。算法返回一个迭代器，指向最后一个删除元素的尾后位置。

```
unique(beg, end)
unique(beg, end, binaryPred)
unique_copy(beg, end, dest)
unique_copy_if(beg, end, dest, binaryPred)
```

重排序列，对相邻的重复元素，通过覆盖它们来进行“删除”。返回一个迭代器，指向不重复元素的尾后位置。第一个版本用`==`确定两个元素是否相同，第二个版本使用谓词检测相邻元素。

```
rotate(beg, mid, end)
rotate_copy(beg, mid, end, dest)
```

围绕 `mid` 指向的元素进行元素转动。元素 `mid` 成为首元素，随后是 `mid+1` 到 `end` 之前的元素，再接着是 `beg` 到 `mid` 之前的元素。返回一个迭代器，指向原来在 `beg` 位置的元素。

使用双向迭代器的重排算法

由于这些算法要反向处理输入序列，它们要求双向迭代器。

```
reverse(beg, end)
reverse_copy(beg, end, dest)
```

翻转序列中的元素。`reverse` 返回 `void`，`reverse_copy` 返回一个迭代器，指向拷贝到目的序列的元素的尾后位置。

使用随机访问迭代器的重排算法

由于这些算法要随机重排元素，它们要求随机访问迭代器。

```
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
shuffle(beg, end, Uniform_rand)
```

混洗输入序列中的元素。第二个版本接受一个可调用对象参数，该对象必须接受一个正整数值，并生成 0 到此值的包含区间内的一个服从均匀分布的随机整数。`shuffle` 的第三个参数必须满足均匀分布随机数生成器的要求（参见 17.4 节，第 659 页）。所有版本都返回 `void`。

< 879

A.2.7 排列算法

排列算法生成序列的字典序排列。对于一个给定序列，这些算法通过重排它的一个排列来生成字典序中下一个或前一个排列。算法返回一个 `bool` 值，指出是否还有下一个或前一个排列。

为了理解什么是下一个或前一个排列，考虑下面这个三字符的序列：abc。它有六种可能的排列：abc、acb、bac、bca、cab 及 cba。这些排列是按字典序递增序列出的。即，abc 是第一个排列，这是因为它的第一个元素小于或等于任何其他排列的首元素，并且它的第二个元素小于任何其他首元素相同的排列。类似的，acb 排在下一位，原因是它以 a 开头，小于任何剩余排列的首元素。同理，以 b 开头的排列也都排在以 c 开头的排列之前。

对于任意给定的排列，基于单个元素的一个特定的序，我们可以获得它的前一个和下一个排列。给定排列 bca，我们知道其前一个排列为 bac，下一个排列为 cab。序列 abc 没有前一个排列，而 cba 没有下一个排列。

这些算法假定序列中的元素都是唯一的，即，没有两个元素的值是一样的。

为了生成排列，必须既向前又向后处理序列，因此算法要求双向迭代器。

```
is_permutation(beg1, end1, beg2)
is_permutation(beg1, end1, beg2, binaryPred)
```

如果第二个序列的某个排列和第一个序列具有相同数目的元素，且元素都相等，则返回 `true`。第一个版本用`==`比较元素，第二个版本使用给定的 `binaryPred`。

```
next_permutation(beg, end)
next_permutation(beg, end, comp)
```

如果序列已经是最后一个排列，则 `next_permutation` 将序列重排为最小的排列，并返回 `false`。否则，它将输入序列转换为字典序中下一个排列，并返回 `true`。第一个版本使用元素的`<`运算符比较元素，第二个版本使用给定的比较操作。

```
prev_permutation(beg, end)
prev_permutation(beg, end, comp)
```

类似 `next_permutation`，但将序列转换为前一个排列。如果序列已经是最小的排列，则将其重排为最大的排列，并返回 `false`。

880 A.2.8 有序序列的集合算法

集合算法实现了有序序列上的一般集合操作。这些算法与标准库 `set` 容器不同，不要与 `set` 上的操作相混淆。这些算法提供了普通顺序容器（`vector`、`list` 等）或其他序列（如输入流）上的类集合行为。

这些算法顺序处理元素，因此要求输入迭代器。他们还接受一个表示目的序列的输出迭代器，唯一的例外是 `includes`。这些算法返回递增后的 `dest` 迭代器，表示写入 `dest` 的最后一个元素之后的位置。

每种算法都有重载版本，第一个使用元素类型的`<`运算符，第二个使用给定的比较操作。

```
includes(beg, end, beg2, end2)
includes(beg, end, beg2, end2, comp)
```

如果第二个序列中每个元素都包含在输入序列中，则返回 `true`。否则返回 `false`。

```
set_union(beg, end, beg2, end2, dest)
set_union(beg, end, beg2, end2, dest, comp)
```

对两个序列中的所有元素，创建它们的有序序列。两个序列都包含的元素在输出序列中只出现一次。输出序列保存在 `dest` 中。

```
set_intersection(beg, end, beg2, end2, dest)
set_intersection(beg, end, beg2, end2, dest, comp)
```

对两个序列都包含的元素创建一个有序序列。结果序列保存在 `dest` 中。

```
set_difference(beg, end, beg2, end2, dest)
set_difference(beg, end, beg2, end2, dest, comp)
```

对出现在第一个序列中，但不在第二个序列中的元素，创建一个有序序列。

```
set_symmetric_difference(beg, end, beg2, end2, dest)
set_symmetric_difference(beg, end, beg2, end2, dest, comp)
```

对只出现在一个序列中的元素，创建一个有序序列。

A.2.9 最小值和最大值

这些算法使用元素类型的`<`运算符或给定的比较操作。第一组算法对值而非序列进行操作。第二组算法接受一个序列，它们要求输入迭代器。

```
min(val1, val2)
min(val1, val2, comp)
min(initializer_list)
min(initializer_list, comp)
max(val1, val2)
max(val1, val2, comp)
max(initializer_list)
max(initializer_list, comp)
```

881

返回 `val1` 和 `val2` 中的最小值/最大值，或 `initializer_list` 中的最小值/最大值。两个实参的类型必须完全一致。参数和返回类型都是 `const` 的引用，意味着对象不会被拷贝。

```
minmax(val1, val2)
minmax(val1, val2, comp)
minmax(initializer_list)
minmax(initializer_list, comp)
```

返回一个 `pair`(参见 11.2.3 节, 第 379 页), 其 `first` 成员为提供的值中的较小者, `second` 成员为较大者。`initializer_list` 版本返回一个 `pair`, 其 `first` 成员为 `list` 中的最小值, `second` 为最大值。

```
min_element(beg, end)
min_element(beg, end, comp)
max_element(beg, end)
max_element(beg, end, comp)
minmax_element(beg, end)
minmax_element(beg, end, comp)
```

`min_element` 和 `max_element` 分别返回指向输入序列中最小和最大元素的迭代器。`minmax_element` 返回一个 `pair`, 其 `first` 成员为最小元素, `second` 成员为最大元素。

字典序比较

此算法比较两个序列, 根据第一对不相等的元素的相对大小来返回结果。算法使用元素类型的<运算符或给定的比较操作。两个序列都要求用输入迭代器给出。

```
lexicographical_compare(beg1, end1, beg2, end2)
lexicographical_compare(beg1, end1, beg2, end2, comp)
```

如果第一个序列在字典序中小于第二个序列, 则返回 `true`。否则, 返回 `false`。如果一个序列比另一个短, 且所有元素都与较长序列的对应元素相等, 则较短序列在字典序中更小。如果序列长度相等, 且对应元素都相等, 则在字典序中任何一个都不大于另外一个。

A.2.10 数值算法

数值算法定义在头文件 `numeric` 中。这些算法要求输入迭代器; 如果算法输出数据, 则使用输出迭代器表示目的位置。

882 > `accumulate(beg, end, init)`
`accumulate(beg, end, init, binaryOp)`

返回输入序列中所有值的和。和的初值从 `init` 指定的值开始。返回类型与 `init` 的类型相同。第一个版本使用元素类型的+运算符, 第二个版本使用指定的二元操作。

```
inner_product(beg1, end1, beg2, init)
inner_product(beg1, end1, beg2, init, binOp1, binOp2)
```

返回两个序列的内积, 即, 对应元素的积的和。两个序列一起处理, 来自两个序列的元素相乘, 乘积被累加起来。和的初值由 `init` 指定, `init` 的类型确定了返回类型。

第一个版本使用元素类型的乘法 (*) 和加法 (+) 运算符。第二个版本使用给定的二元操作, 使用第一个操作代替加法, 第二个操作代替乘法。

```
partial_sum(beg, end, dest)
partial_sum(beg, end, dest, binaryOp)
```

将新序列写入 `dest`, 每个新元素的值都等于输入范围中当前位置和之前位置上所有元素之和。第一个版本使用元素类型的+运算符; 第二个版本使用指定的二元操作。算法返回递增后的 `dest` 迭代器, 指向最后一个写入元素之后的位置。

```
adjacent_difference(beg, end, dest)
adjacent_difference(beg, end, dest, binaryOp)
```

将新序列写入 `dest`, 每个新元素(除了首元素之外)的值都等于输入范围中当前位置和前一个位置元素之差。第一个版本使用元素类型的-运算符, 第二个版本使用指定的二元操作。

```
iota(beg, end, val)
```

将 `val` 赋予首元素并递增 `val`。将递增后的值赋予下一个元素, 继续递增 `val`, 然后将递增后的值赋予序列中的下一个元素。继续递增 `val` 并将其新值赋予输入序列中的后续元素。

A.3 随机数

标准库定义了一组随机数引擎类和适配器，使用不同数学方法生成伪随机数。标准库还定义了一组分布模板，根据不同的概率分布生成随机数。引擎和分布类型的名字都与它们的数学性质相对应。

这些类如何生成随机数的细节已经大大超出了本书的范围。在本节中，我们将列出这些引擎和分布类型，但读者需要查询其他资料来学习如何使用这些类型。

A.3.1 随机数分布

<883

除了总是生成 `bool` 类型的 `bernoulli_distribution` 外，其他分布类型都是模板。每个模板都接受单个类型参数，它指出了分布生成的结果类型。

分布类与我们已经用过的其他类模板不同，它们限制了我们可以为模板类型指定哪些类型。一些分布模板只能用来生成浮点数，而其他模板只能用来生成整数。

在下面的描述中，我们通过将类型说明为 `template_name<RealT>` 来指出分布生成浮点数。对这些模板，我们可以用 `float`、`double` 或 `long double` 代替 `RealT`。类似的，`IntT` 表示要求一个内置整型类型，但不包括 `bool` 类型或任何 `char` 类型。可以用来代替 `IntT` 的类型是 `short`、`int`、`long`、`long long`、`unsigned short`、`unsigned int`、`unsigned long` 或 `unsigned long long`。

分布模板定义了一个默认模板类型参数（参见 17.4.2 节，第 664 页）。整型分布的默认参数是 `int`，生成浮点数的模板的默认参数是 `double`。

每个分布的构造函数都有这种分布特定的参数。某些参数指出了分布的范围。这些范围与迭代器范围不同，都是包含的。

均匀分布

```
uniform_int_distribution<IntT> u(m, n);  
uniform_real_distribution<RealT> u(x, y);
```

生成指定类型的，在给定包含范围内的值。`m`（或 `x`）是可以返回的最小值；`n`（或 `y`）是最大值。`m` 默认为 0；`n` 默认为类型 `IntT` 对象可以表示的最大值。`x` 默认为 0.0，`y` 默认为 1.0。

伯努利分布

```
bernoulli_distribution b(p);
```

以给定概率 `p` 生成 `true`；`p` 的默认值为 0.5。

```
binomial_distribution<IntT> b(t, p);
```

分布是按采样大小为整型值 `t`，概率为 `p` 生成的；`t` 的默认值为 1，`p` 的默认值为 0.5。

```
geometric_distribution<IntT> g(p);
```

每次试验成功的概率为 `p`；`p` 的默认值为 0.5。

```
negative_binomial_distribution<IntT> nb(k, p);
```

`k`（整型值）次试验成功的概率为 `p`；`k` 的默认值为 1，`p` 的默认值为 0.5。

泊松分布

`poisson_distribution<IntT> p(x);`

均值为 double 值 x 的分布。

884 `exponential_distribution<RealT> e(lam);`

指数分布，参数 lambda 通过浮点值 lam 给出；lam 的默认值为 1.0。

`gamma_distribution<RealT> g(a, b);`

alpha（形状参数）为 a, beta（尺度参数）为 b；两者的默认值均为 1.0。

`weibull_distribution<RealT> w(a, b);`

形状参数为 a, 尺度参数为 b 的分布；两者的默认值均为 1.0。

`extreme_value_distribution<RealT> e(a, b);`

a 的默认值为 0.0, b 的默认值为 1.0。

正态分布

`normal_distribution<RealT> n(m, s);`

均值为 m, 标准差为 s; m 的默认值为 0.0, s 的默认值为 1.0。

`lognormal_distribution<RealT> ln(m, s);`

均值为 m, 标准差为 s; m 的默认值为 0.0, s 的默认值为 1.0。

`chi_squared_distribution<RealT> c(x);`

自由度为 x; 默认值为 1.0。

`cauchy_distribution<RealT> c(a, b);`

位置参数 a 和尺度参数 b 的默认值分别为 0.0 和 1.0。

`fisher_f_distribution<RealT> f(m, n);`

自由度为 m 和 n; 默认值均为 1。

`student_t_distribution<RealT> s(n);`

自由度为 n; n 的默认值均为 1。

抽样分布

`discrete_distribution<IntT> d(i, j);`

`discrete_distribution<IntT> d(il);`

i 和 j 是一个权重序列的输入迭代器, il 是一个权重的花括号列表。权重必须能转换为 double。

`piecewise_constant_distribution<RealT> pc(b, e, w);`

b、e 和 w 是输入迭代器。

`piecewise_linear_distribution<RealT> pl(b, e, w);`

b、e 和 w 是输入迭代器。

A.3.2 随机数引擎

标准库定义了三个类，实现了不同的算法来生成随机数。标准库还定义了三个适配器，可以修改给定引擎生成的序列。引擎和引擎适配器类都是模板。与分布的参数不同，这些引擎的参数更为复杂，且需深入了解特定引擎使用的数学知识。我们在这里列出所有引擎，以便读者对它们有所了解，但介绍如何生成这些类型超出了本书的范围。 ◀885

标准库还定义了几个从引擎和适配器类型构造的类型。`default_random_engine` 类型是一个参数化的引擎类型的类型别名，参数化所用的变量的目的是在通常情况下获得好的性能。标准库还定义了几个类，它们都是一个引擎或适配器的完全特例化版本。标准库定义的引擎和特例化版本如下：

`default_random_engine`

某个其他引擎类型的类型别名，目的是用于大多数情况。

`linear_congruential_engine`

`minstd_rand0` 的乘数为 16807，模为 2147483647，增量为 0。

`minstd_rand` 的乘数为 48271，模为 2147483647，增量为 0。

`mersenne_twister_engine`

`mt19937` 为 32 位无符号梅森旋转生成器。

`mt19937_64` 为 64 位无符号梅森旋转生成器。

`subtract_with_carry_engine`

`ranlux24_base` 为 32 位无符号借位减法生成器。

`ranlux48_base` 为 64 位无符号借位减法生成器。

`discard_block_engine`

引擎适配器，将其底层引擎的结果丢弃。用要使用的底层引擎、块大小和旧块大小来参数化。

`ranlux24` 使用 `ranlux24_base` 引擎，块大小为 223，旧块大小为 23。

`ranlux48` 使用 `ranlux48_base` 引擎，块大小为 389，旧块大小为 11。

`independent_bits_engine`

引擎适配器，生成指定位数的随机数。用要使用的底层引擎、结果的位数以及保存生成的二进制位的无符号整型类型来参数化。指定的位数必须小于指定的无符号类型所能保存的位数。

`shuffle_order_engine`

引擎适配器，返回的就是底层引擎生成的数，但返回的顺序不同。用要使用的底层引擎和要混洗的元素数目来参数化。

`knuth_b` 使用 `minstd_rand0` 和表大小 256。