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。 818

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。

done!

第 19 章
特殊工具与技术

内容

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令 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：

824

```
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：

826

```
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
}
```

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，这样做的好处是可以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条表达式中完成。

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

t1 == t2	如果 type_info 对象 t1 和 t2 表示同一种类型，返回 true；否则返回 false
t1 != t2	如果 type_info 对象 t1 和 t2 表示不同的类型，返回 true；否则返回 false
t.name()	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
t1.before(t2)	返回一个 bool 值，表示 t1 是否位于 t2 之前。before 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：

C++11

```
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
```

835

形参匹配与枚举类型

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
846
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成删除的。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。 852

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，当我们将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;
```