

调用时才发生动态绑定，通过对象调用不进行动态绑定。

4. catch 子句的次序必须反映类型层次

将异常类型组织成类层次的时候，用户可以选择应用程序处理异常的粒度级别。例如，只希望清除并退出的应用程序可以定义一个 try 块，该 try 块包围 main 函数中带有如下 catch 代码：

```
catch(exception &e) {
    // do cleanup
    // print a message
    cerr << "Exiting: " << e.what() << endl;
    size_t status_indicator = 42;          // set and return an
    return(status_indicator);              // error indicator
}
```

有更严格实时需求的程序可能需要更好的异常控制，这样的应用程序将清除导致异常的一切并继续执行。

因为 catch 子句按出现次序匹配，所以使用来自继承层次的异常的程序必须将它们的 catch 子句排序，以便派生类型的处理代码出现在其基类类型的 catch 之前。

694



带有因继承而相关的类型的多个 catch 子句，必须从最低派生类型到最高派生类型排序。

习题

习题 17.3 解释下面这个 try 块为什么不正确，并改正它。

```
try {
    // use of the C++ standard library
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
```

17.1.4 重新抛出

有可能单个 catch 不能完全处理一个异常。在进行了一些校正行动之后，catch 可能确定该异常必须由函数调用链中更上层的函数来处理，catch 可以通过重新抛出 (rethrow) 将异常传递给函数调用链中更上层的函数。重新抛出是后面不跟类型或表达式的一个 throw：

```
throw;
```

空 throw 语句将重新抛出异常对象，它只能出现在 catch 或者从 catch 调用的函数中。如果在处理代码不活动时碰到空 throw，就调用 terminate 函数。

虽然重新抛出不指定自己的异常，但仍然将一个异常对象沿链向上传递，被抛出的异常是原来的异常对象，而不是 catch 形参。当 catch 形参是基类类型的时候，我们不知道由重新抛出表达式抛出的实际类型，该类型取决于异常对象的动态类型，而不是 catch 形参的静态类型。例如，来自带基类类型形参 catch 的重新抛出，可能实际上抛出一个派生类型的对象。

一般而言, `catch` 可以改变它的形参。在改变它的形参之后, 如果 `catch` 重新抛出异常, 那么, 只有当异常说明符是引用的时候, 才会传播那些改变。

```
catch (my_error &eObj) {           // specifier is a reference type
    eObj.status = severeErr;        // modifies the exception object
    throw; // the status member of the exception object is severeErr
} catch (other_error eObj) {       // specifier is a nonreference type
    eObj.status = badErr;           // modifies local copy only
    throw; // the status member of the exception rethrown is unchanged
}
```

695

17.1.5 捕获所有异常的处理代码

即使函数不能处理被抛出的异常, 它也可能想要在随抛出异常退出之前执行一些动作。除了为每个可能的异常提供特定 `catch` 子句之外, 因为不可能知道可能被抛出的所有异常, 所以可以使用**捕获所有异常** `catch` 子句 (`catch-all`) 的。捕获所有异常的 `catch` 子句形式为 `(...)`。例如:

```
// matches any exception that might be thrown
catch (...) {
    // place our code here
}
```

捕获所有异常的 `catch` 子句与任意类型的异常都匹配。

`catch (...)` 经常与重新抛出表达式结合使用, `catch` 完成可做的所有局部工作, 然后重新抛出异常:

```
void manip() {
    try {
        // actions that cause an exception to be thrown
    }
    catch (...) {
        // work to partially handle the exception
        throw;
    }
}
```

`catch (...)` 子句可以单独使用, 也可以用在几个 `catch` 子句中间。



如果 `catch (...)` 与其他 `catch` 子句结合使用, 它必须是最后一个, 否则, 任何跟在它后面的 `catch` 子句都将不能被匹配。

17.1.6 函数测试块与构造函数

一般而言, 异常可能发生在程序执行的任何一点。具体而言, 异常可能发生在构造函数中, 或者发生在处理构造函数初始化式的时候。在进入构造函数函数体之前处理构造函数初始化式, 构造函数函数体内部的 `catch` 子句不能处理在处理构造函数初始化式时可能发生的异常。

为了处理来自构造函数初始化式的异常, 必须将构造函数编写为**函数测试块** (`function try block`)。可以使用函数测试块将一组 `catch` 子句与函数联成一个整体。作为例子, 可以将第 16 章的 `Handle` 构造函数包装在一个用来检测 `new` 中失败的测试块当中:

```
template <class T> Handle<T>::Handle(T *p)
try : ptr(p), use(new size_t(1))
{
    // empty function body
} catch(const std::bad_alloc &e)
    { handle_out_of_memory(e); }
```

696

习题

习题 17.4 对于如下的基本 C++ 程序

```
int main() {
    // use of the C++ standard library
}
```

修改 main 函数以捕获由 C++ 标准库中函数抛出的任何异常。处理代码应该在调用 abort 函数（在头文件 cstdlib 中定义）终止 main 函数之前显示与异常相关的错误信息。

习题 17.5 对于下面的异常类型以及 catch 子句，编写一个 throw 表达式，该表达式创建一个可被每个 catch 子句捕获的异常对象。

```
(a) class exceptionType { };
    catch(exceptionType *pet) { }
(b) catch(...) { }
(c) enum mathErr { overflow, underflow, zeroDivide };
    catch(mathErr &ref) { }
(d) typedef int EXCPTYPE;
    catch(EXCPTYPE) { }
```

注意，关键字 try 出现在成员初始化列表之前，并且测试块的复合语句包围了构造函数的函数体。catch 子句既可以处理从成员初始化列表中抛出的异常，也可以处理从构造函数函数体中抛出的异常。



构造函数要处理来自构造函数初始化式的异常，唯一的方法是将构造函数编写为函数测试块。

17.1.7 异常类层次

6.13 节介绍过标准库异常类，但该节没有涵盖的是这些类是因继承而相关的，图 17-1 描绘了这个继承层次。

exception 类型所定义的唯一操作是一个名为 what 的虚成员，该函数返回 const char* 对象，它一般返回用来在抛出位置构造异常对象的信息。因为 what 是虚函数，如果捕获了基类类型引用，对 what 函数的调用将执行适合异常对象的动态类型的版本。

1. 用于书店应用程序的异常类

标准异常类可以用于许多应用程序，此外，应用程序还经常通过从 exception 类或者中间基类派生附加类型来扩充 exception 层次。这些新派生的类可以表示特定于应用程序领域的异常类型。

697

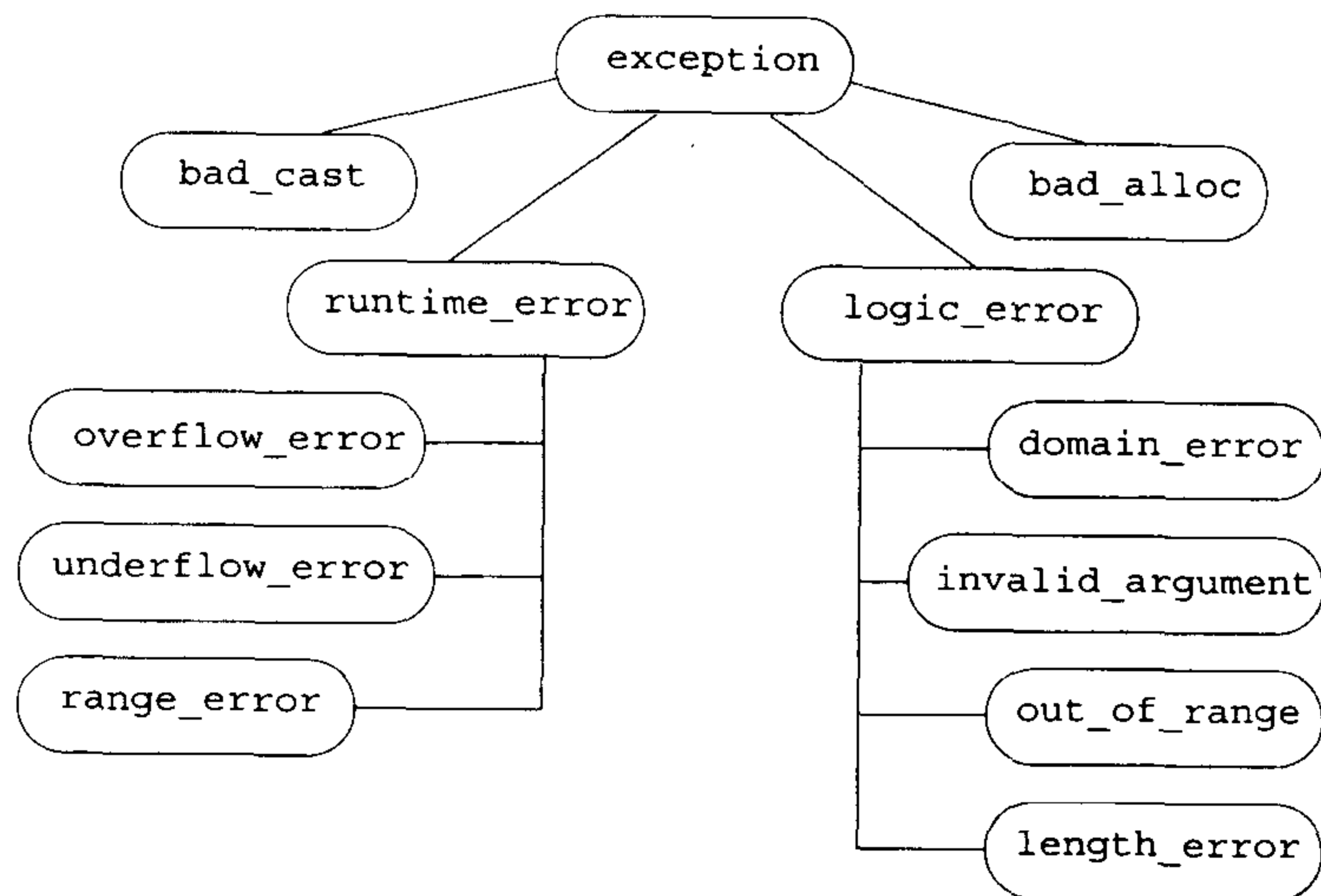


图 17-1 标准 exception 类层次

如果正在建立一个实际的书店应用程序，我们的类也许比本书给出的复杂得多，使它们更为精细的一个方法可能在于它们的异常处理。事实上，我们可能已经定义了自己的异常层次来表示可能出现的特定于应用程序的问题。我们的设计可能包括下面这样的类：

```

// hypothetical exception classes for a bookstore application
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;
    // Section 17.1.10 explains the destructor and why we need one
    virtual ~isbn_mismatch() throw() { }
};

```

698

在这里，通过从标准异常类派生，定义了特定于应用程序的异常类型。像任何层次一样，可以认为异常类按层组织。随着层次的加深，每一层变得更特殊的异常。例如，层次中第一层即最一般的层由 `exception` 类代表，当捕获这一类型的对象时，我们所知道的只是有些地方出错了。

第二层将 `exception` 特化为两个大类：运行时错误和逻辑错误。我们的书店异常类表示更特化的层中的事件。`out_of_stock` 类表示可能在运行时出现问题的特定于应用程序的事情，可以用它发出不能履行某个订单的信号。`isbn_mismatch` 异常是从 `logic_error` 派生的更特殊的异常，原则上，程序可以通过调用 `same_isbn` 检测到不匹配的 ISBN。

2. 使用程序员定义的异常类型

用和使用标准库类相同的方法使用自己的异常类。程序的一个部分抛出某个这些类型的对象，程序的另一部分捕获并处理指出的问题。例如，可以为 `Sales_item` 类定义重载加操作符，以便如果它检测到 ISBN 不匹配就抛出一个 `isbn_mismatch` 类型的错误：

```
// throws exception if both objects do not refer to the same isbn
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    if (!lhs.same_isbn(rhs))
        throw isbn_mismatch("isbn mismatch",
                               lhs.book(), rhs.book());
    Sales_item ret(lhs); // copy lhs into a local object that we'll return
    ret += rhs;          // add in the contents of rhs
    return ret;          // return ret by value
}
```

然后，使用加操作符的代码可以检测这个错误，写出适当的错误消息，并继续：

```
// use hypothetical bookstore exceptions
Sales_item item1, item2, sum;
while (cin >> item1 >> item2) { // read two transactions
    try {
        sum = item1 + item2;      // calculate their sum
        // use sum
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left isbn(" << e.left
              << ") right isbn(" << e.right << ")"
              << endl;
    }
}
```

699

17.1.8 自动资源释放

17.1.2 节介绍过，在发生异常时自动撤销局部对象。运行析构函数这一事实对应用程序的设计具有重要意义，这也是为什么我们鼓励使用标准库类的原因。考虑下面的函数：

```
void f()
{
    vector<string> v; // local vector
    string s;
    while (cin >> s)
        v.push_back(s); // populate the vector
    string *p = new string[v.size()]; // dynamic array
    // remaining processing
    // it is possible that an exception occurs in this code
    // function cleanup is bypassed if an exception occurs
    delete [] p;
} // v destroyed automatically when the function exits
```

这个函数定义了一个局部 `vector` 并动态分配了一个数组。在正常执行的情况下，数组和 `vector` 都在退出函数之前被撤销，函数中最后一个语句释放数组，在函数结束时自动撤销 `vector`。

但是，如果在函数内部发生异常，则将撤销 vector 而不会释放数组。问题在于数组不是自动释放的。在 new 之后但在 delete 之前发生的异常使得数组没有被撤销。不管何时发生异常，都保证运行 vector 析构函数。

用类管理资源分配

对析构函数的运行导致一个重要的编程技术的出现，它使程序更为异常安全的 (exception safe)。异常安全的意味着，即使发生异常，程序也能正确操作。在这种情况下，“安全”来自于保证“如果发生异常，被分配的任何资源都适当地释放”。

通过定义一个类来封装资源的分配和释放，可以保证正确释放资源。这一技术常称为“资源分配即初始化”，简称 RAII。

应该设计资源管理类，以便构造函数分配资源而析构函数释放资源。想要分配资源的时候，就定义该类类型的对象。如果不发生异常，就在获得资源的对象超出作用域的时候释放资源。更为重要的是，如果在创建了对象之后但在它超出作用域之前发生异常，那么，编译器保证撤销该对象，作为展开定义对象的作用域的一部分。

700

下面的类是一个原型例子，其中构造函数分配资源而析构函数释放资源：

```
class Resource {
public:
    Resource(params p): r(allocate(p)) { }
    ~Resource() { release(r); }
    // also need to define copy and assignment
private:
    resource_type *r;           // resource managed by this type
    resource_type *allocate(params p); // allocate this resource
    void release(resource_type*); // free this resource
};
```

Resource 类是分配资源和回收资源的类型，它保存表示该资源的数据成员。Resource 的构造函数分配资源，而析构函数释放它。当使用这个类的时候

```
void fcn()
{
    Resource res(args); // allocates resource_type
    // code that might throw an exception
    // if exception occurs, destructor for res is run automatically
    // ...
} // res goes out of scope and is destroyed automatically
```

自动释放资源。如果函数正常终止，就在 Resource 对象超出作用域时释放资源；如果函数因异常而提早退出，编译器就运行 Resource 的析构函数作为异常处理过程的一部分。



最佳
实践

可能存在异常的程序以及分配资源的程序应该使用类来管理那些资源。如本节所述，使用类管理分配和回收可以保证如果发生异常就释放资源。

习题

习题 17.6 给定下面的函数，解释当发生异常时会发生什么。

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // exception occurs here
    // ...
}
```

习题 17.7 有两种方法可以使上面的代码是异常安全的，描述并实现它们。

701

17.1.9 auto_ptr 类

标准库的 `auto_ptr` 类是上一节中介绍的异常安全的“资源分配即初始化”技术的例子。`auto_ptr` 类（见表 17-1）是接受一个类型形参的模板，它为动态分配的对象提供异常安全。`auto_ptr` 类在头文件 `memory` 中定义。

表 17-1 auto_ptr 类	
<code>auto_ptr<T> ap;</code>	创建名为 <code>ap</code> 的未绑定的 <code>auto_ptr</code> 对象
<code>auto_ptr<T> ap(p);</code>	创建名为 <code>ap</code> 的 <code>auto_ptr</code> 对象， <code>ap</code> 拥有指针 <code>p</code> 指向的对象。该构造函数为 <code>explicit</code>
<code>auto_ptr<T> ap1(ap2);</code>	创建名为 <code>ap1</code> 的 <code>auto_ptr</code> 对象， <code>ap1</code> 保存原来存储在 <code>ap2</code> 中的指针。将所有权转给 <code>ap1</code> ， <code>ap2</code> 成为未绑定的 <code>auto_ptr</code> 对象
<code>ap1 = ap2</code>	将所有权从 <code>ap2</code> 转给 <code>ap1</code> 。删除 <code>ap1</code> 指向的对象并且使 <code>ap1</code> 指向 <code>ap2</code> 指向的对象，使 <code>ap2</code> 成为未绑定的
<code>~ap</code>	析构函数。删除 <code>ap</code> 指向的对象
<code>*ap</code>	返回对 <code>ap</code> 所绑定的对象的引用
<code>ap-></code>	返回 <code>ap</code> 保存的指针
<code>ap.reset(p)</code>	如果 <code>p</code> 与 <code>ap</code> 的值不同，则删除 <code>ap</code> 指向的对象并且将 <code>ap</code> 绑定到 <code>p</code>
<code>ap.release()</code>	返回 <code>ap</code> 所保存的指针并且使 <code>ap</code> 成为未绑定的
<code>ap.get()</code>	返回 <code>ap</code> 保存的指针



`auto_ptr` 只能用于管理从 `new` 返回的一个对象，它不能管理动态分配的数组。正如我们所见，当 `auto_ptr` 被复制或赋值的时候，有不寻常的行为，因此，不能将 `auto_ptr` 存储在标准库容器类型中。

`auto_ptr` 对象只能保存一个指向对象的指针，并且不能用于指向动态分配的数组，使用 `auto_ptr` 对象指向动态分配的数组会导致未定义的运行时行为。

每个 `auto_ptr` 对象绑定到一个对象或者指向一个对象。当 `auto_ptr` 对象指向一个对象的时候，可以说它“拥有”该对象。当 `auto_ptr` 对象超出作用域或者另外撤销的时候，就自动回收 `auto_ptr` 所指向的动态分配对象。

1. 为异常安全的内存分配使用 auto_ptr

如果通过常规指针分配内存，而且在执行 `delete` 之前发生异常，就不会自动释放该内存：

```
void f()
```

```
{
    int *ip = new int(42);           // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip;                       // return the memory before exiting
}
```

如果在 new 和 delete 之间发生异常，并且该异常不被局部捕获，就不会执行 delete，则永不回收该内存。

如果使用一个 auto_ptr 对象来代替，将会自动释放内存，即使提早退出这个块也是这样：

```
void f()
{
    auto_ptr<int> ap(new int(42)); // allocate a new object
    // code that throws an exception that is not caught inside f
} // auto_ptr freed automatically when function ends
```

702 在这个例子中，编译器保证在展开栈越过 f 之前运行 ap 的析构函数。

2. auto_ptr 是可以保存任何类型指针的模板

auto_ptr 类是接受单个类型形参的模板，该类型指定 auto_ptr 可以绑定的对象的类型，因此，可以创建任何类型的 auto_ptr：

```
auto_ptr<string> ap1(new string("Brontosaurus"));
```

3. 将 auto_ptr 绑定到指针

在最常见的情况下，将 auto_ptr 对象初始化为由 new 表达式返回的对象的地址：

```
auto_ptr<int> pi(new int(1024));
```

这个语句将 pi 初始化为由 new 表达式创建的对象地址，这个 new 表达式将对象初始化为 1024。

接受指针的构造函数为 explicit (12.4.4 节) 构造函数，所以必须使用初始化的直接形式来创建 auto_ptr 对象：

```
// error: constructor that takes a pointer is explicit and can't be used implicitly
auto_ptr<int> pi = new int(1024);
auto_ptr<int> pi(new int(1024)); // ok: uses direct initialization
```

703 pi 所指的由 new 表达式创建的对象在超出作用域时自动删除。如果 pi 是局部对象，pi 所指对象在定义 pi 的块的末尾删除；如果发生异常，则 pi 也超出作用域，析构函数将自动运行 pi 的析构函数作为异常处理的一部分；如果 pi 是全局对象，就在程序末尾删除 pi 引用的对象。

4. 使用 auto_ptr 对象

假设希望访问 string 操作。用普通 string 指针，像下面这样做：

```
string *pstr_type = new string("Brontosaurus");
if (pstr_type->empty())
    // oops, something wrong
```

auto_ptr 类定义了解引用操作符 (*) 和箭头操作符 (->) 的重载版本 (14.6 节)，因为 auto_ptr 定义了这些操作符，所以可以用类似于使用内置指针的方式使用 auto_ptr 对象：

```
// normal pointer operations for dereference and arrow
*ap1 = "TRex";           // assigns a new value to the object to which ap1 points
string s = *ap1;         // initializes s as a copy of the object to which ap1 points
```

```
if (ap1->empty()) // runs empty on the string to which ap1 points
```

auto_ptr 的主要目的是，在保证自动删除 auto_ptr 对象引用的对象的同时，支持普通指针式行为。正如我们所见，自动删除该对象这一事实导致在怎样复制和访问它们的地址值方面，auto_ptr 与普通指针明显不同。

5. auto_ptr 对象的复制和赋值是破坏性操作



auto_ptr 和内置指针对待复制和赋值有非常关键的重要区别。当复制 auto_ptr 对象或者将它的值赋给其他 auto_ptr 对象的时候，将基础对象的所有权从原来的 auto_ptr 对象转给副本，原来的 auto_ptr 对象重置为未绑定状态。

复制（或者赋值）普通指针是复制（或者赋值）地址，在复制（或者赋值）之后，两个指针指向同一对象。在复制（或者赋值）auto_ptr 对象之后，原来的 auto_ptr 对象不指向对象而新的 auto_ptr 对象（左边的 auto_ptr 对象）拥有基础对象：

```
auto_ptr<string> ap1(new string("Stegosaurus"));
// after the copy ap1 is unbound
auto_ptr<string> ap2(ap1); // ownership transferred from ap1 to ap2
```

当复制 auto_ptr 对象或者对 auto_ptr 对象赋值的时候，右边的 auto_ptr 对象让出对基础对象的所有职责并重置为未绑定的 auto_ptr 对象。在上例中，删除 string 对象的是 ap2 而不是 ap1，在复制之后，ap1 不再指向任何对象。

704

与其他复制或赋值操作不同，auto_ptr 的复制和赋值改变右操作数，因此，赋值的左右操作数必须都是可修改的左值。

6. 赋值删除左操作数指向的对象

除了将所有权从右操作数转给左操作数之外，赋值还删除左操作数原来指向的对象——假如两个对象不同。通常自身赋值没有效果。

```
auto_ptr<string> ap3(new string("Pterodactyl"));
// object pointed to by ap3 is deleted and ownership transferred from ap2 to ap3;
ap3 = ap2; // after the assignment, ap2 is unbound
```

将 ap2 赋给 ap3 之后：

- 删除了 ap3 指向的对象。
- 将 ap3 置为指向 ap2 所指的對象。
- ap2 是未绑定的 auto_ptr 对象。



因为复制和赋值是破坏性操作，所以不能将 auto_ptr 对象存储在标准容器中。标准库的容器类要求在复制或赋值之后两个对象相等，auto_ptr 不满足这一要求，如果将 ap2 赋给 ap1，则在赋值之后 $ap1 \neq ap2$ ，复制也类似。

7. auto_ptr 的默认构造函数

如果不给定初始式，auto_ptr 对象是未绑定的，它不指向对象：

```
auto_ptr<int> p_auto; // p_auto doesn't refer to any object
```

默认情况下，auto_ptr 的内部指针值置为 0。对未绑定的 auto_ptr 对象解引用，其效果与对未

绑定的指针解引用相同——程序出错并且没有定义会发生什么：

```
*p_auto = 1024; // error: dereference auto_ptr that doesn't point to an object
```

8. 测试 auto_ptr 对象

为了检查指针是否未绑定，可以在条件中直接测试指针，效果是确定指针是否为 0。相反，不能直接测试 auto_ptr 对象：

```
// error: cannot use an auto_ptr as a condition
if (p_auto)
    *p_auto = 1024;
```

705

auto_ptr 类型没有定义到可用作条件的类型的转换，相反，要测试 auto_ptr 对象，必须使用它的 get 成员，该成员返回包含在 auto_ptr 对象中的基础指针：

```
// revised test to guarantee p_auto refers to an object
if (p_auto.get())
    *p_auto = 1024;
```

为了确定 auto_ptr 是否指向一个对象，可以将 get 的返回值与 0 比较。



应该只用 get 询问 auto_ptr 对象或者使用返回的指针值，不能用 get 作为创建其他 auto_ptr 对象的实参。

使用 get 成员初始化其他 auto_ptr 对象违反 auto_ptr 类设计原则：在任意时刻只有一个 auto_ptr 对象保存给定指针，如果两个 auto_ptr 对象保存相同的指针，该指针就会被 delete 两次。

9. reset 操作

auto_ptr 对象与内置指针的另一个区别是，不能直接将一个地址（或者其他指针）赋给 auto_ptr 对象：

```
p_auto = new int(1024); // error: cannot assign a pointer to an auto_ptr
```

相反，必须调用 reset 函数来改变指针：

```
// revised test to guarantee p_auto refers to an object
if (p_auto.get())
    *p_auto = 1024;
else
    // reset p_auto to a new object
    p_auto.reset(new int(1024));
```

要复位 auto_ptr 对象，可以将 0 传给 reset 函数。



调用 auto_ptr 对象的 reset 函数时，在将 auto_ptr 对象绑定到其他对象之前，会删除 auto_ptr 对象所指向的对象（如果存在）。但是，正如自身赋值是没有效果的一样，如果调用该 auto_ptr 对象已经保存的同一指针的 reset 函数，也没有效果，不会删除对象。

习题

习题 17.8 下面 `auto_ptr` 声明中, 哪些是不合法的或者可能导致随后的程序错误? 解释每个声明的问题。

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef auto_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());
```

习题 17.9 假定 `ps` 是一个指向 `string` 的指针, 如果有的话, 下面两个 `assign` (9.6.2 节) 调用有什么不同? 你认为哪个更好, 为什么?

```
(a) ps.get()->assign("Danny");  (b) ps->assign("Danny");
```

17.1.10 异常说明

查看普通函数声明的时候, 不可能确定该函数会抛出什么异常, 但是, 为了编写适当的 `catch` 子句, 了解函数是否抛出异常以及会抛出哪种异常是很有用的。异常说明 (exception specification) 指定, 如果函数抛出异常, 被抛出的异常将是包含在该说明中的一种, 或者是从列出的异常中派生的类型。

706

警告: `auto_ptr` 的缺陷

`auto_ptr` 类模板为处理动态分配的内存提供了安全性和便利性的尺度。要正确地使用 `auto_ptr` 类, 必须坚持该类强加的下列限制:

(1) 不要使用 `auto_ptr` 对象保存指向静态分配对象的指针, 否则, 当 `auto_ptr` 对象本身被撤销的时候, 它将试图删除指向非动态分配对象的指针, 导致未定义的行为。

(2) 永远不要使用两个 `auto_ptr` 对象指向同一对象, 导致这个错误的一种明显方式是, 使用同一指针来初始化或者 `reset` 两个不同的 `auto_ptr` 对象。另一种导致这个错误的微妙方式可能是, 使用一个 `auto_ptr` 对象的 `get` 函数的结果来初始化或者 `reset` 另一个 `auto_ptr` 对象。

(3) 不要使用 `auto_ptr` 对象保存指向动态分配数组的指针。当 `auto_ptr` 对象被删除的时候, 它只释放一个对象——它使用普通 `delete` 操作符, 而不用数组的 `delete[]` 操作符。

(4) 不要将 `auto_ptr` 对象存储在容器中。容器要求所保存的类型定义复制和赋值操作符, 使它们表现得类似于内置类型的操作符: 在复制 (或者赋值) 之后, 两个对象必须具有相同值, `auto_ptr` 类不满足这个要求。

1. 定义异常说明

异常说明跟在函数形参表之后。一个异常说明在关键字 `throw` 之后跟着一个 (可能为空的) 由圆括号括住的异常类型列表:

```
void recoup(int) throw (runtime_error);
```

这个声明指出, `recoup` 是接受 `int` 值的函数, 并返回 `void`。如果 `recoup` 抛出一个异常, 该异常将是 `runtime_error` 对象, 或者是由 `runtime_error` 派生的类型的异常。

空说明列表指出函数不抛出任何异常:

```
void no_problem() throw();
```

异常说明是函数接口的一部分, 函数定义以及该函数的任意声明必须具有相同的异常说明。



如果一个函数声明没有指定异常说明, 则该函数可以抛出任意类型的异常。

2. 违反异常说明

但是, 不可能在编译时知道程序是否抛出异常以及会抛出哪些异常, 只有在运行时才能检测是否违反函数异常说明。

如果函数抛出了没有在其异常说明中列出的异常, 就调用标准库函数 **unexpected**。默认情况下, `unexpected` 函数调用 `terminate` 函数, `terminate` 函数一般会终止程序。



在编译的时候, 编译器不能也不会试图验证异常说明。

即使对函数代码的偶然阅读指明, 它可能抛出异常说明中没有的异常, 编译器也不会给出提示:

```
void f() throw()           // promise not to throw any exception
{
    throw exception();     // violates exception specification
}
```

相反, 编译器会产生代码以便保证: 如果抛出了一个违反异常说明的异常, 就调用 `unexpected` 函数。

3. 确定函数不抛出异常

因为不能在编译时检查异常说明, 异常说明的应用通常是有限的。



异常说明有用的一种重要情况是, 如果函数可以保证不会抛出任何异常。

确定函数将不抛出任何异常, 对函数的用户和编译器都有所帮助: 知道函数不抛出异常会简化编写调用该函数的异常安全的代码的工作, 我们可以知道在调用函数时不必担心异常, 而且, 如果编译器知道不会抛出异常, 它就可以执行被可能抛出异常的代码所抑制的优化。

4. 异常说明与成员函数

像非成员函数一样, 成员函数声明的异常说明跟在函数形参表之后。例如, C++ 标准库中的 `bad_alloc` 类定义为所有成员都有空异常说明, 这些成员承诺不抛出异常:

```
// illustrative definition of library bad_alloc class
class bad_alloc : public exception {
public:
    bad_alloc() throw();
```

```

    bad_alloc(const bad_alloc &) throw();
    bad_alloc & operator=(const bad_alloc &) throw();
    virtual ~bad_alloc() throw();
    virtual const char* what() const throw();
};

```

注意，在 `const` 成员函数声明中，异常说明跟在 `const` 限定符之后。

5. 异常说明与析构函数

17.1.7 节介绍了两个假设的书店应用程序异常类，`isbn_mismatch` 类将析构函数定义为：

```

class isbn_mismatch: public std::logic_error {
public:
    virtual ~isbn_mismatch() throw() { }
};

```

并说明我们会在这里解释这种用法。

`isbn_mismatch` 类从 `logic_error` 类继承而来，`logic_error` 是一个标准异常类，该标准异常类的析构函数包含空 `throw()` 说明符，它们承诺不抛出任何异常。当继承这两个类中的一个时，我们的析构函数也必须承诺不抛出任何异常。

`out_of_stock` 类没有成员，所以它的合成析构函数不做任何可能抛出异常的事情，因此，编译器可以知道合成析构函数将遵守不抛出异常的承诺。

`isbn_mismatch` 类有两个 `string` 类成员，这意味着 `isbn_mismatch` 的合成析构函数调用 `string` 析构函数。C++ 标准保证，`string` 析构函数像任意其他标准库类析构函数一样，不抛出异常。但是，标准库的析构函数没有定义异常说明，在这种情况下，我们知道，但编译器不知道，`string` 析构函数将不抛出异常。我们必须定义自己的析构函数来恢复析构函数不抛出异常的承诺。

6. 异常说明与虚函数

基类中虚函数的异常说明，可以与派生类中对应虚函数的异常说明不同。

709

但是，派生类虚函数的异常说明必须与对应基类虚函数的异常说明同样严格，或者比后者更受限。

这个限制保证，当使用指向基类类型的指针调用派生类虚函数的时候，派生类的异常说明不会增加新的可抛出异常。例如：

```

class Base {
public:
    virtual double f1(double) throw ();
    virtual int f2(int) throw (std::logic_error);
    virtual std::string f3() throw
        (std::logic_error, std::runtime_error);
};
class Derived : public Base {
public:
    // error: exception specification is less restrictive than Base::f1's
    double f1(double) throw (std::underflow_error);
    // ok: same exception specification as Base::f2
    int f2(int) throw (std::logic_error);
    // ok: Derived f3 is more restrictive

```

```
std::string f3() throw ();
};
```

派生类中 `f1` 的声明是错误的，因为它的异常说明在基类 `f1` 版本列出的异常中增加了一个异常。派生类不能在异常说明列表中增加异常，原因在于，继承层次的用户应该能够编写依赖于该说明列表的代码。如果通过基类指针或引用进行函数调用，那么，这些类的用户所涉及的应该只是在基类中指定的异常。

通过将派生类抛出的异常限制为由基类所列出的那些，在编写代码时就可以知道必须处理哪些异常。代码可以依赖于这样一个事实：基类中的异常列表是虚函数的派生类版本可以抛出的异常列表的超集。例如，当调用 `f3` 的时候，我们知道只需要处理 `logic_error` 或 `runtime_error`：

```
// guarantees not to throw exceptions
void compute(Base *pb) throw()
{
    try {
        // may throw exception of type std::logic_error
        // or std::runtime_error
        pb->f3();
    } catch (const logic_error &le) { /* ... */ }
    catch (const runtime_error &re) { /* ... */ }
}
```

710 在确定可能需要捕获什么异常的时候，`compute` 函数使用基类中的异常说明。

17.1.11 函数指针的异常说明

异常说明是函数类型的一部分。这样，也可以在函数指针的定义中提供异常说明：

```
void (*pf)(int) throw(runtime_error);
```

这个声明是说，`pr` 指向接受 `int` 值的函数，该函数返回 `void` 对象，该函数只能抛出 `runtime_error` 类型的异常。如果不提供异常说明，该指针就可以指向能够抛出任意类型异常的具有匹配类型的函数。

在用另一指针初始化带异常说明的函数的指针，或者将后者赋值给函数地址的时候，两个指针的异常说明不必相同，但是，源指针的异常说明必须至少与目标指针的一样严格。

```
void recoup(int) throw(runtime_error);
// ok: recoup is as restrictive as pf1
void (*pf1)(int) throw(runtime_error) = recoup;
// ok: recoup is more restrictive than pf2
void (*pf2)(int) throw(runtime_error, logic_error) = recoup;
// error: recoup is less restrictive than pf3
void (*pf3)(int) throw() = recoup;
// ok: recoup is more restrictive than pf4
void (*pf4)(int) = recoup;
```

第三个初始化是错误的。指针声明指出，`pf3` 指向不抛出任何异常的函数，但是，`recoup` 函数指出它能抛出 `runtime_error` 类型的异常，`recoup` 函数抛出的异常类型超出了 `pf3` 所指定的，对 `pf3` 而言，`recoup` 函数不是有效的初始化式，并且会引发一个编译时错误。

习题

习题 17.10 如果函数有形如 `throw()` 的异常说明，它能抛出什么异常？如果没有异常说明呢？

习题 17.11 如果有，下面哪个初始化是错误的？为什么？

```
void example() throw(string);
(a) void (*pf1)() = example;
(b) void (*pf2)() throw() = example;
```

习题 17.12 下面函数可以抛出哪些异常？

```
(a) void operate() throw(logic_error);
(b) int op(int) throw(underflow_error, overflow_error);
(c) char manip(string) throw();
(d) void process();
```

711

17.2 命名空间

在一个给定作用域中定义的每个名字在该作用域中必须是唯一的，对庞大、复杂的应用程序而言，这个要求可能难以满足。这样的应用程序的全局作用域中一般有许多名字定义。由独立开发的库构成的复杂程序更有可能遇到名字冲突——同样的名字既可能在我们自己的代码中使用，也可能（更常见地）在独立供应商提供的代码中使用。

库倾向于定义许多全局名字——主要是模板名、类型名或函数名。在使用来自多个供应商的库编写应用程序的时候，这些名字中有一些几乎不可避免地会发生冲突，这种名字冲突问题称为**命名空间污染**（`namespace pollution`）问题。

传统上，程序员通过将全局实体的名字设得很长来避免命名空间污染，经常用特定字符序列作为程序中名字的前缀：

```
class cplusplus_primer_Query { ... };
ifstream&
cplusplus_primer_open_file(ifstream&, const string&);
```

这个解决方案很不理想：程序员编写和阅读使用这种长名字的程序非常麻烦。**命名空间**（`namespace`）为防止名字冲突提供了更加可控的机制，命名空间能够划分全局命名空间，这样使用独立开发的库就更加容易了。一个命名空间是一个作用域，通过在命名空间内部定义库中的名字，库的作者（以及用户）可以避免全局名字固有的限制。

17.2.1 命名空间的定义

命名空间定义以关键字 `namespace` 开始，后接命名空间的名字。

```
namespace cplusplus_primer {
    class Sales_item { /* ... */};
    Sales_item operator+(const Sales_item&,
                        const Sales_item&);

    class Query {
    public:
        Query(const std::string&);
```

```

        std::ostream &display(std::ostream&) const;
        // ...
    };
    class Query_base { /* ... */};
}

```

这段代码定义了名为 `cplusplus_primer` 的命名空间，它有四个成员：两个类，一个重载的+操作符，一个函数。

像其他名字一样，命名空间的名字在定义该命名空间的作用域中必须是唯一的。命名空间可以在全局作用域或其他作用域内部定义，但不能在函数或类内部定义。

命名空间名字后面接着由花括号括住的一块声明和定义，可以在命名空间中放入可以出现在全局作用域的任意声明：类、变量（以及它们的初始化）、函数（以及它们的定义）、模板以及其他命名空间。



命名空间作用域不能以分号结束。

1. 每个命名空间是一个作用域

定义在命名空间中的实体称为命名空间成员。像任意作用域的情况一样，命名空间中的每个名字必须引用该命名空间中的唯一实体。因为不同命名空间引入不同作用域，所以不同命名空间可以具有同名成员。

在命名空间中定义的名字可以被命名空间中的其他成员直接访问，命名空间外部的代码必须指出名字定义在哪个命名空间中：

```

cplusplus_primer::Query q =
    cplusplus_primer::Query("hello");
q.display(cout);
// ...

```

如果另一命名空间（如 `AddisonWesley`）也提供 `TextQuery` 类，而且我们想要使用那个类代替 `cplusplus_primer` 中定义的 `TextQuery`，可以通过这样修改代码而实现：

```

AddisonWesley::Query q = AddisonWesley::Query("hello");
q.display(cout);
// ...

```

2. 从命名空间外部使用命名空间成员

当然，总是使用限定名 `namespace_name::member_name` 引用命名空间成员可能非常麻烦。像对 `std` 中定义的命名空间所做的那样，可以编写 `using` 声明（3.1 节）来获得对我们知道将经常使用的名字的直接访问：

```
using cplusplus_primer::Query;
```

在这个 `using` 声明之后，程序可以无须 `cplusplus_primer` 限定符而直接使用名字 `Query`，在 17.2.4 节将介绍简化访问的其他方法。

3. 命名空间可以是不连续的

与其他作用域不同，命名空间可以在几个部分中定义。命名空间由它的分离定义部分的总和构成，命名空间是累积的。一个命名空间的分离部分可以分散在多个文件中，在不同文本文件中

的命名空间定义也是累积的。当然，名字只在声明名字的文件中可见，这一常规限制继续应用，所以，如果命名空间的一个部分需要定义在另一文件中的名字，仍然必须声明该名字。

编写命名空间定义：

```
namespace namespace_name {
    // declarations
}
```

既可以定义新的命名空间，也可以添加到现存命名空间中。

如果名字 `namespace_name` 不是引用前面定义的命名空间，则用该名字创建新的命名空间，否则，这个定义打开一个已存在的命名空间，并将这些新声明加到那个命名空间。

4. 接口和实现的分离

命名空间定义可以不连续意味着，可以用分离的接口文件和实现文件构成命名空间，因此，可以用与管理自己的类和函数定义相同的方法来组织命名空间：

(1) 定义类的命名空间成员，以及作为类接口的一部分的函数声明与对象声明，可以放在头文件中，使用命名空间成员的文件可以包含这些头文件。

(2) 命名空间成员的定义可以放在单独的源文件中。

按这种方式组织命名空间，也满足了不同实体（非内联函数、静态数据成员、变量等）只能在一个程序中定义一次的要求，这个要求同样适用于命名空间中定义的名字。通过将接口和实现分离，可以保证函数和其他我们需要的名字只定义一次，但相同的声明可以在任何使用该实体的地方见到。



最佳
实践

定义多个不相关类型的命名空间应该使用分离的文件，表示该命名空间定义的每个类型。

5. 定义本书的命名空间

使用将接口和实现分离的策略，可以将 `cplusplus_primer` 库定义在几个分离的文件中。本书第一部分建立的 `Sales_item` 的声明及其相关函数可以放在 `Sales_item.h` 中，第 15 章的 `Query` 类的定义以及相关函数放在 `Query.h` 中，以此类推。对应的实现文件可以是 `Sales_item.cc` 和 `Query.cc`：

```
// ---- Sales_item.h ----
namespace cplusplus_primer {
    class Sales_item { /* ... */};
    Sales_item operator+(const Sales_item&,
                        const Sales_item&);
    // declarations for remaining functions in the Sales_item interface
}
// ---- Query.h ----
namespace cplusplus_primer {
    class Query {
    public:
        Query(const std::string&);
        std::ostream &display(std::ostream&) const;
        // ...
    };
};
```

```

    class Query_base { /* ... */};
}
// ---- Sales_item.cc ----
#include "Sales_item.h"
namespace cplusplus_primer {
    // definitions for Sales_item members and overloaded operators
}
// ---- Query.cc ----
#include "Query.h"
namespace cplusplus_primer {
    // definitions for Query members and related functions
}

```

这种程序组织给予开发者和库用户必要的模块性。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。如果允许 `Sales_item.cc` 和 `Query.cc` 文件编译和链接到一个程序而不会导致编译时错误和运行时错误，就可以对用户隐藏实现。库的开发者可以独立工作于每个类型的实现。

使用我们的库的程序可以包含需要的头文件，那些头文件中的名字定义在命名空间 `cplusplus_primer` 内部：

```

// ---- user.cc ----
// defines the cplusplus_primer::Sales_item class
#include "Sales_item.h"
int main()
{
    // ...
    cplusplus_primer::Sales_item trans1, trans2;
    // ...
    return 0;
}

```

715

6. 定义命名空间成员

在命名空间内部定义的函数可以使用同一命名空间中定义的名字的简写形式：

```

namespace cplusplus_primer {
    // members defined inside the namespace may use unqualified names
    std::istream&
    operator>>(std::istream& in, Sales_item& s)
    {
        // ...
    }
}

```

也可以在命名空间定义的外部定义命名空间成员，用类似于在类外部定义类成员的方式：名字的命名空间声明必须在作用域中，并且定义必须指定该名字所属的命名空间：

```

// namespace members defined outside the namespace must use qualified names
cplusplus_primer::Sales_item
cplusplus_primer::operator+(const Sales_item& lhs,
                             const Sales_item& rhs)
{
    Sales_item ret(lhs);
    // ...
}

```

这个定义看起来类似于定义在类外部的类成员函数，返回类型和函数名由命名空间名字限定。一旦看到完全限定的函数名，就处于命名空间的作用域中。因此，形参表和函数体中的命名空间成员引用可以使用非限定名引用 `Sales_item`。

7. 不能在不相关的命名空间中定义成员

虽然可以在命名空间定义的外部定义命名空间成员，对这个定义可以出现的地方仍有些限制，只有包围成员声明的命名空间可以包含成员的定义。例如，`operator+`既可以定义在命名空间 `cplusplus_primer` 中，也可以定义在全局作用域中，但它不能定义在不相关的命名空间中。

8. 全局命名空间

定义在全局作用域的名字（在任意类、函数或命名空间外部声明的名字）是定义在全局命名空间（`global namespace`）中的。全局命名空间是隐式声明的，存在于每个程序中。在全局作用域定义实体的每个文件将那些名字加到全局命名空间。

可以用作用域操作符引用全局命名空间的成员。因为全局命名空间是隐含的，它没有名字，所以记号 `::member_name` 引用全局命名空间的成员。

716

习题

习题 17.13 定义 17.1.7 节描述的书店异常类，作为名为 `Bookstore` 的命名空间的成员。

习题 17.14 在命名空间 `Bookstore` 内部定义 `Sales_item` 及其操作符。定义加操作符抛出一个异常。

习题 17.15 编写一个程序，使用 `Sales_item` 加操作符并处理任何异常。使这个程序成为名为 `MyApp` 的另一命名空间的成员。这个程序应使用上题中在命名空间 `Bookstore` 中定义的异常类。

17.2.2 嵌套命名空间

一个嵌套命名空间即是一个嵌套作用域——其作用域嵌套在包含它的命名空间内部。嵌套命名空间中的名字遵循常规规则：外围命名空间中声明的名字被嵌套命名空间中同一名字的声明所屏蔽。嵌套命名空间内部定义的名字局部于该命名空间。外围命名空间之外的代码只能通过限定名引用嵌套命名空间中的名字。

嵌套命名空间可以改进库中代码的组织：

```
namespace cplusplus_primer {
    // first nested namespace:
    // defines the Query portion of the library
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // second nested namespace:
    // defines the Sales_item portion of the library
    namespace Bookstore {
        class Item_base { /* ... */ };
        class Bulk_item : public Item_base { /* ... */ };
    }
}
```

```

    // ...
}
}

```

命名空间 `cplusplus_primer` 现在包含两个嵌套命名空间：名为 `QueryLib` 的命名空间和名为 `Bookstore` 的命名空间。

当库提供者需要防止库中每个部分的名字与库中其他部分的名字冲突的时候，嵌套命名空间是很有用的。

嵌套命名空间中成员的名字由外围命名空间的名字和嵌套命名空间的名字构成。例如，嵌套命名空间 `QueryLib` 中声明的类的名字是

717

```
cplusplus_primer::QueryLib::Query
```

习题

习题 17.16 将为回答每章中的问题而编写的程序组织到每一章自己的命名空间中，也就是说，命名空间 `chapterrefinheritance` 将包含 `Query` 程序的代码，而 `chapterrefalgs` 将包含 `TextQuery` 代码。使用这个结构，编译 `Query` 代码示例。

习题 17.17 在本书中，我们定义了两个名为 `Sales_item` 的不同类：在第一部分定义和使用的初始简单类，以及在 15.8.1 节定义的与 `Item_base` 继承层次接口的句柄类。在命名空间 `cplusplus_primer` 内部定义两个嵌套命名空间，用于区别这两个类定义。

17.2.3 未命名的命名空间

命名空间可以是未命名的，未命名的命名空间（`unnamed namespace`）在定义时没有给定名字。未命名的命名空间以关键字 `namespace` 开头，接在关键字 `namespace` 后面的是由花括号定界的声明块。



未命名的命名空间与其他命名空间不同，未命名的命名空间的定义局部于特定文件，从不跨越多个文本文件。

未命名的命名空间可以在给定文件中不连续，但不能跨越文件，每个文件有自己的未命名的命名空间。

未命名的命名空间用于声明局部于文件的实体。在未命名的命名空间中定义的变量在程序开始时创建，在程序结束之前一直存在。

未命名的命名空间中定义的名字可直接使用，毕竟，没有命名空间名字来限定它们。不能使用作用域操作符来引用未命名的命名空间的成员。

未命名的命名空间中定义的名字只在包含该命名空间的文件中可见。如果另一文件包含一个未命名的命名空间，两个命名空间不相关。两个命名空间可以定义相同的名字，而这些定义将引用不同的实体。

未命名的命名空间中定义的名字可以在定义该命名空间所在的作用域中找到。如果在文件的最外层作用域中定义未命名的命名空间，那么，未命名的命名空间中的名字必须与全局作用域中

定义的名字不同:

```
int i; // global declaration for i
namespace {
    int i;
}
// error: ambiguous defined globally and in an unnested, unnamed namespace
i = 10;
```

像任意其他命名空间一样, 未命名的命名空间也可以嵌套在另一命名空间内部。如果未命名的命名空间是嵌套的, 其中的名字按常规方法使用外围命名空间名字访问:

```
namespace local {
    namespace {
        int i;
    }
}
// ok: i defined in a nested unnamed namespace is distinct from global i
local::i = 42;
```



如果头文件定义了未命名的命名空间, 那么, 在每个包含该头文件的文件中, 该命名空间中的名字将定义不同的局部实体。

在所有其他方式中, 未命名的命名空间的成员都是普通程序实体。

未命名的命名空间取代文件中的静态声明

在标准 C++ 中引入命名空间之前, 程序必须将名字声明为 static, 使它们局部于一个文件。文件中静态声明的使用从 C 语言继承而来, 在 C 语言中, 声明为 static 的局部实体在声明它的文件之外不可见。



C++ 不赞成文件静态声明。不赞成的特征是在未来版本中可能不支持的特征。应该避免文件静态而使用未命名的命名空间代替。

习题

习题 17.18 为什么在程序中可以定义自己的命名空间? 何时可以使用未命名的命名空间?

习题 17.19 假定有下面的 operator* 的声明, operator* 是嵌套命名空间 cplusplus_primer::MatrixLib 的成员:

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
```

怎样在全局作用域中定义这个操作符? 只需给出操作符定义的原型。

718

719

17.2.4 命名空间成员的使用

像命名空间名 `namespace_name::member_name` 成员名这样引用命名空间的成员无可否认是很麻烦，特别是，命名空间名字很长的时候。幸好，有办法让使用命名空间成员比较容易。我们的程序已经使用了其中的一种方法，就是 `using` 声明（3.1 节），本节将介绍其他方法：命名空间别名和 `using` 指示。



最佳
实践

除了在函数或其他作用域内部，头文件不应该包含 `using` 指示或 `using` 声明。在其顶级作用域包含 `using` 指示或 `using` 声明的头文件，具有将该名字注入包含该头文件的文件中的效果。头文件应该只定义作为其接口的一部分的名字，不要定义在其实现中使用的名字。

1. `using` 声明，扼要重述

本书中使用标准库中名字的程序一般假设进行了适当的 `using` 声明：

```
map<string, vector< pair<size_t, size_t> > > word_map;
```

假定进行了下面的声明：

```
using std::map;
using std::pair;
using std::size_t;
using std::string;
using std::vector;
```

一个 `using` 声明一次只引入一个命名空间成员，它使得无论程序中使用哪些名字，都能够非常明确。

2. `using` 声明的作用域

`using` 声明中引入的名字遵循常规作用域规则。从 `using` 声明点开始，直到包含该 `using` 声明的作用域的末尾，名字都是可见的。外部作用域中定义的同名实体被屏蔽。

简写名字只能在声明它的作用域及其嵌套作用域中使用，一旦该作用域结束了，就必须使用完全限定名。

`using` 声明可以出现在全局作用域、局部作用域或者命名空间作用域中。类作用域中的 `using` 声明局限于被定义类的基类中定义的名字。

3. 命名空间别名

可用命名空间别名（namespace alias）将较短的同义词与命名空间名字相关联。例如，像

```
namespace cplusplus_primer { /* ... */ };
```

这样的长命名空间名字，可以像下面这样与较短的同义词相关联：

```
namespace primer = cplusplus_primer;
```

命名空间别名声明以关键字 `namespace` 开头，接（较短的）命名空间别名名字，再接 `=`，再接原来的命名空间名字和分号。如果原来的命名空间名字是未定义的，就会出错。

命名空间别名也可以引用嵌套的命名空间。除了编写

```
cplusplus_primer::QueryLib::Query tq;
```

之外，我们可以定义和使用 `cplusplus_primer::QueryLib` 的别名：

```
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query tq;
```



一个命名空间可以有許多别名，所有别名以及原来的命名空间名字都可以互换使用。

4. using 指示

像 `using` 声明一样，**using 指示**使我们能够使用命名空间名字的简写形式。与 `using` 声明不同，`using` 指示无法控制使得哪些名字可见——它们都是可见的。

5. using 指示的形式

`using` 指示以关键字 `using` 开头，后接关键字 `namespace`，再接命名空间名字。如果该名字不是已经定义的命名空间名字，就会出错。

`using` 指示使得特定命名空间的所有名字可见，没有限制。短格式名字可从 `using` 指示点开始使用，直到出现 `using` 指示的作用域的末尾。



可以尝试用 `using` 指示编写程序，但在使用多个库的时候，这样做会重新引入名字冲突的所有问题。

6. using 指示与作用域

用 `using` 指示引入的名字的作用域比 `using` 声明的更复杂。`using` 声明将名字直接放入出现 `using` 声明的作用域，好像 `using` 声明是命名空间成员的局部别名一样。因为这种声明是局部化的，冲突的机会最小。

721



`using` 指示不声明命名空间成员名字的别名，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效果。

在最简单的情况下，假定有命名空间 `A` 和函数 `f`，二者都在全局作用域中定义。如果 `f` 有关于 `A` 的 `using` 指示，那么，在 `f` 中，将好像 `A` 中的名字出现在全局作用域中 `f` 的定义之前一样：

```
// namespace A and function f are defined at global scope
namespace A {
    int i, j;
}
void f()
{
    using namespace A;           // injects names from A into the global scope
    cout << i * j << endl;       // uses i and j from namespace A
    //...
}
```



`using` 指示有用的一种情况是，用在命名空间本身的实现文件中。

7. using 指示例子

看一个例子：

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // other declarations
}
int bj = 0; // ok: bj inside blip is hidden inside a namespace
void manip()
{
    // using directive - names in blip "added" to global scope
    using namespace blip;
    // clash between ::bj and blip::bj
    // detected only if bj is used
    ++bi; // sets blip::bi to 17
    ++bj; // error: ambiguous
    // global bj or blip::bj?
    ++::bj; // ok: sets global bj to 1
    ++blip::bj; // ok: sets blip::bj to 16
    int bk = 97; // local bk hides blip::bk
    ++bk; // sets local bk to 98
}
```

722

manip 中的 using 指示使 manip 能够直接访问 blip 中的所有名字：使用它们的简化形式，该函数可以引用这些成员的名字。

blip 的成员看来好像是在定义 blip 和 manip 的作用域中定义的一样。如果在全局作用域中定义 blip，则 blip 的成员看来好像是声明在全局作用域的一样。因为名字在不同的作用域中，manip 内部的局部声明可以屏蔽命名空间的某些成员名字，局部变量 bk 屏蔽命名空间名字 blip::bk，在 manip 内部对 bk 的引用没有二义性，它引用局部变量 bk。

命名空间中的名字可能会与外围作用域中定义的其他名字冲突。例如，对 manip 而言，blip 成员 bj 看来好像声明在全局作用域中，但是，全局作用域存在另一名为 bj 的对象。这种冲突是允许的，但为了使用该名字，必须显式指出想要的是哪个版本，因此，在 manip 内部的 bj 使用是有二义性的：该名字既可引用全局变量又可引用命名空间 blip 的成员。

为了使用像 bj 这样的名字，必须使用作用域操作符指出想要的是哪个名字。可以编写::bj 来获得在全局作用域中定义的变量，要使用 blip 中定义的 bj，必须使用它的限定名字 blip::bj。

习题

习题 17.20 解释 using 声明和 using 指示之间的区别。

习题 17.21 考虑下面代码样本：

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// position 1
```

```

void manip() {
    // position 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    ++::ivar;
}

```

如果命名空间 Exercise 的所有成员的 using 声明放在标为 *position 1* 的地方, 这个代码样本中的声明和表达式的效果是什么? 如果放在 *position 2* 位置呢? 用命名空间 Exercise 的 using 指示代替 using 声明, 回答同一问题。

723

警告: 避免 using 指示

using 指示注入来自一个命名空间的所有名字, 它的使用是靠不住的: 只用一个语句, 命名空间的所有成员名就突然可见了。虽然这个方法看似简单, 但也有它自身的问题。如果应用程序使用许多库, 并且用 using 指示使得这些库中的名字可见, 那么, 全局命名空间污染问题就重新出现。

而且, 当引入库的新版本的时候, 正在工作的程序可能会编译失败。如果新版本引入一个与应用程序正在使用的名字冲突的名字, 就会引发这个问题。

另一个问题是, 由 using 指示引起的二义性错误只能在使用处检测, 这个后来的检测意味着, 可能在特定库引入很久之后才引发冲突, 如果程序开始使用该库的新部分, 就可能引发先前未检测到的冲突。

相对于依赖于 using 指示, 对程序中使用的每个命名空间名字使用 using 声明更好, 这样做减少注入到命名空间¹中的名字数目, 由 using 声明引起的二义性错误在声明点而不是使用点检测, 因此更容易发现和修正。

17.2.5 类、命名空间和作用域

正如我们已经注意到的, 命名空间是作用域。像在任意其他作用域中一样, 名字从声明点开始可见。名字的可见性穿过任意嵌套作用域, 直到引入名字的块的末尾。

对命名空间内部使用的名字的查找遵循常规 C++ 查找规则: 当查找名字的时候, 通过外围作用域向外查找。对命名空间内部使用的名字而言, 外围作用域可能是一个或多个嵌套的命名空间, 最终以全包围的全局命名空间结束。只考虑已经在使用点之前声明的名字, 而该使用点仍在开放的块中:

```

namespace A {
    int i;
    namespace B {
        int i;          // hides A::i within B
        int j;
        int f1()
        {

```

1. 此处原文有误, “命名空间”应为“程序”。——译者注

```

        int j;          // j is local to f1 and hides A::B::j
        return i;       // returns B::i
    }
} // namespace B is closed and names in it are no longer visible
int f2() {
    return j;           // error: j is not defined
}
int j = i;              // initialized from A::i
}

```

724

用非常相似的方式确定类成员定义中使用的名字，只有一个重要区别：如果名字不是局部于成员函数的，就试着在查找更外层作用域之前在类成员中确定名字。

正如 12.3 节所介绍的，类内部所定义的成员可以使用出现在定义文本之后的名字。例如，即使数据成员的定义出现在构造函数定义之后，类定义体内部定义的构造函数也可以初始化那些数据成员。当在类作用域中使用名字的时候，首先在成员本身中查找，然后在类中查找，包括任意基类，只有在查找完类之后，才检查外围作用域。当类包在命名空间中的时候，发生相同的查找：首先在成员中找，然后在类（包括基类）中找，再在外围作用域中找，外围作用域中的一个或多个可以是命名空间：

```

namespace A {
    int i;
    int k;
    class C1 {
    public:
        C1(): i(0), j(0) { } // ok: initializes C1::i and C1::j
        int f1()
        {
            return k;        // returns A::k
        }
        int f2()
        {
            return h;        // error: h is not defined
        }
        int f3();
    private:
        int i;               // hides A::i within C1
        int j;
    };
    int h = i;               // initialized from A::i
}
// member f3 is defined outside class C1 and outside namespace A
int A::C1::f3()
{
    return h;                // ok: returns A::h
}

```

除了成员定义例外，总是向上查找作用域：名字在使用之前必须声明。因此，f2 中的 return 语句将不能编译，它试图引用命名空间 A 中的名字 h，但 h 还没有定义。如果使 A 中的名字在 C1 的定义之前定义，h 的使用就是合法的。类似地，f3 内部对 h 的使用是正确的，因为 f3 定义在已经定义了 A::h 之后。



提示 可以从函数的限定名推断出查找名字时所检查作用域的次序，限定名以相反次序指出被查找的作用域。

725

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，然后查找外围类 `C1` 的作用域。在查找包含 `f3` 定义的作用域之前，最后查找命名空间 `A` 的作用域。

1. 实参相关的查找与类类型形参

考虑下面的简单程序：

```
std::string s;
// ok: calls std::getline(std::istream&, const std::string&)
getline(std::cin, s);
```

这段程序使用了 `std::string` 类型，但它不加限制地引用了 `getline` 函数。为什么可以无须特定 `std::` 限定符或 `using` 声明而使用该函数？

它给出了屏蔽命名空间名字规则的一个重要例外。



接受类类型形参（或类类型指针及引用形参）的函数（包括重载操作符），以及
与类本身定义在同一命名空间中的函数（包括重载操作符），在用类类型对象（或类
类型的引用及指针）作为实参的时候是可见的。

当编译器看到 `getline` 函数的使用 `getline(std::cin, s);` 的时候，它在当前作用域、包含调用的作用域以及定义 `cin` 的类型和 `string` 类型的命名空间中查找匹配的函数。因此，它在命名空间 `std` 中查找并找到由 `string` 类型定义的 `getline` 函数。

如果函数具有类类型形参就使得函数可见，其原因在于，允许无须单独的 `using` 声明就可以使用概念上作为类接口组成部分的非成员函数。能够使用非成员操作对操作符函数特别有用。

例如，考虑下面的简单程序：

```
std::string s;
cin >> s;
```

如果没有查找规则的这个例外，我们将必须编写下面二者之一：

```
using std::operator>>;           // need to allow cin >> s
std::operator>>(std::cin, s);    // ok: explicitly use std::>>
```

这两个声明都不方便使用，而且可能使 `string` 和 IO 库的使用变得更复杂。

726

2. 隐式友元声明与命名空间

回忆一下，当一个类声明友元函数（12.5 节）的时候，函数的声明不必是可见的。如果不存在可见的声明，那么，友元声明具有将该函数或类的声明放入外围作用域的效果。如果类在命名空间内部定义，则没有另外声明的友元函数在同一命名空间中声明。

```
namespace A {
    class C {
        friend void f(const C&); // makes f a member of namespace A
    };
}
```

因为该友元接受类类型实参并与类隐式声明在同一命名空间中，所以使用它时可以无须使用显式命名空间限定符：

```
// f2 defined at global scope
void f2()
{
    A::C cobj;
    f(cobj); // calls A::f
}
```

17.2.6 重载与命名空间

正如我们所见，每个命名空间维持自己的作用域，因此，作为两个不同命名空间的成员的函数不能互相重载。但是，给定命名空间可以包含一组重载函数成员。

一般而言，命名空间内部的函数匹配（7.8.2 节）以与我们已经见过的方式相同的方式进行：

(1) 找到候选函数集。如果一个函数在调用时其声明可见并且与被调用函数同名，这个函数就是候选者。

(2) 从候选集中选择可行函数。如果函数的形参数目与函数调用的实参数目相同，并且每个形参都可用对应实参匹配，这个函数就是可行的。

(3) 从可行集合中选择一个最佳匹配，并产生代码调用该函数。如果可行集合为空，则调用出错，没有匹配；如果可行集合非空且没有最佳匹配，则调用有二义性。

1. 候选函数与命名空间

命名空间对函数匹配有两个影响。一个影响是明显的：using 声明或 using 指示可以将函数加到候选集合。另一个影响则微妙得多。

正如前节所见，有一个或多个类类型形参的函数的名字查找包括定义每个形参类型的命名空间。这个规则还影响怎样确定候选集合，为找候选函数而查找定义形参类（以及定义其基类）的每个命名空间，将那些命名空间中任意与被调用函数名字相同的函数加入候选集合。即使这些函数在调用点不可见，也将之加入候选集合。将那些命名空间中带有匹配名字的函数加入候选集合：

```
namespace NS {
    class Item_base { /* ... */ };
    void display(const Item_base&) { }
}
// Bulk_item's base class is declared in namespace NS
class Bulk_item : public NS::Item_base { };
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

display 函数的实参 book1 具有类类型 Bulk_item。display 调用的候选函数不仅是在调用 display 函数的地方其声明可见的函数，还包括声明 Bulk_item 类及其基类 Item_base 的命名空间中的函数。命名空间 NS 中声明的函数 display(const Item_base&) 被加到候选函数集合中。

2. 重载与 using 声明

using 声明声明一个名字。正如 15.5.3 节所见，没有办法编写 using 声明来引用特定函数

727

声明:

```
using NS::print(int); // error: cannot specify parameter list
using NS::print;      // ok: using declarations specify names only
```

如果命名空间内部的函数是重载的,那么,该函数名字的 using 声明声明了所有具有该名字的函数。如果命名空间 NS 中有用于 int 和 double 的 print 函数,则 NS::print 的 using 声明使得两个函数都在当前作用域中可见。

一个 using 声明包括重载函数的所有版本以保证不违反命名空间的接口。库作者为一个理由提供不同函数,允许用户选择性地忽略重载函数集合中的某些但不是全部函数,可能会导致奇怪的程序行为。

由 using 声明引入的函数,重载出现 using 声明的作用域中的任意其他同名函数的声明。

728

如果 using 声明在已经有同名且带相同形参表的函数的作用域中引入函数,则 using 声明出错,否则,using 定义给定名字的另一重载实例,效果是增大候选函数集合。

3. 重载与 using 指示

using 指示将命名空间成员提升到外围作用域。如果命名空间函数与命名空间所在的作用域中声明的函数同名,就将命名空间成员加到重载集合中:

```
namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}
void print(const std::string &);
// using directive:
using namespace libs_R_us;
// using directive added names to the candidate set for calls to print:
// print(int) from libs_R_us
// print(double) from libs_R_us
// print(const std::string &) declared explicitly
void fooBar(int ival)
{
    print("Value: "); // calls global print(const string &)
    print(ival);      // calls libs_R_us::print(int)
}
```

4. 跨越多个 using 指示的重载

如果存在许多 using 指示,则来自每个命名空间的名字成为候选集合的组成部分:

```
namespace AW {
    int print(int);
}
namespace Primer {
    double print(double);
}
// using directives:
// form an overload set of functions from different namespaces
using namespace AW;
using namespace Primer;
long double print(long double);
int main() {
    print(1); // calls AW::print(int)
    print(3.1); // calls Primer::print(double)
```

```

    return 0;
}

```

729

全局作用域中 `print` 函数的重载集合包含函数 `print(int)`、`print(double)` 和 `print(long double)`，即使这些函数原来在不同的命名空间中声明，它们都是为 `main` 中函数调用考虑的重载集合的组成部分。

习题

习题 17.22 给定下面的代码，如果有，确定哪个函数与 `compute` 函数的调用匹配，列出候选函数与可行函数。如果有，对实参应用什么类型转换序列，以匹配每个可行函数的形参？

```

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);

int main()
{
    compute(0);
    return 0;
}

```

如果 `main` 中的 `using` 声明放在 `compute` 调用之前，会发生什么情况？回答与前面相同的问题。

17.2.7 命名空间与模板

在命名空间内部声明模板影响着怎样声明模板特化（16.6 节）：模板的显式特化必须在定义通用模板的命名空间中声明，否则，该特化将与它所特化的模板不同名。

有两种定义特化的方式：一种是重新打开命名空间并加入特化的定义，可以这样做是因为命名空间定义是不连续的；或者，可以用与在命名空间定义外部定义命名空间成员相同的方式来定义特化：使用由命名空间名字限定的模板名定义特化。



730

为了提供命名空间中所定义模板的自己的特化，必须保证在包含原始模板定义的命名空间中定义特化。

17.3 多重继承与虚继承

大多数应用程序使用单个基类的公用继承，但是，在某些情况下，单继承是不够用的，因为可能无法为问题域建模，或者会对模型带来不必要的复杂性。

在这些情况下，**多重继承**（multiple inheritance）可以更直接地为应用程序建模。多重继承是从多于一个直接基类派生类的能力，多重继承的派生类继承其所有父类的属性。尽管概念简单，

缠绕多个基类的细节可能会带来错综复杂的设计问题或实现问题。

17.3.1 多重继承

本节使用动物园动物层次的一个教学例子。动物园动物在不同抽象级别存在,有个体的动物,由名字区分,如 Ling-Ling、Mowgli 和 Balou; 每个动物属于一个物种,例如, Ling-Ling 是一个大熊猫; 物种又是科的成员,大熊猫是熊科的成员; 每个科又是动物界的成员——在这个例子中,比较受限的是一个特定的动物园。

每个抽象级别包含支持广泛用户的数据和操作。我们将定义一个抽象 ZooAnimal 类保存所有动物园动物的公共信息并提供公用接口, Bear 类将包含 Bear 科的独特信息,以此类推。

除了实际的动物园动物类之外,还有一些辅助类封装不同的抽象,如濒临灭绝的动物。例如,在 Panda 类的实现中, Panda 同时从 Bear 和 Endangered 派生。

1. 定义多个类

为了支持多重继承,扩充派生列表

```
class Bear : public ZooAnimal {
};
```

以支持由逗号分隔的基类列表:

```
class Panda : public Bear, public Endangered {
};
```

派生类为每个基类(显式或隐式地)指定了访问级别——public、protected 或 private。像单继承一样,只有在定义之后,类才可以用作多重继承的基类。对于类可以继承的基类的数目,没有语言强加的限制,但在一个给定派生列表中,一个基类只能出现一次。

2. 多重继承的派生类从每个基类中继承状态

在多重继承下,派生类的对象包含每个基类的基类子对象(15.2.3 节)。当我们编写

```
Panda ying_yang("ying_yang");
```

的时候,对象 ying_yang 包含一个 Bear 类子对象(Bear 类子对象本身包含一个 ZooAnimal 基类子对象)、一个 Endangered 类子对象以及 Panda 类中声明的非 static 数据成员(如果有的话),见图 17-2。

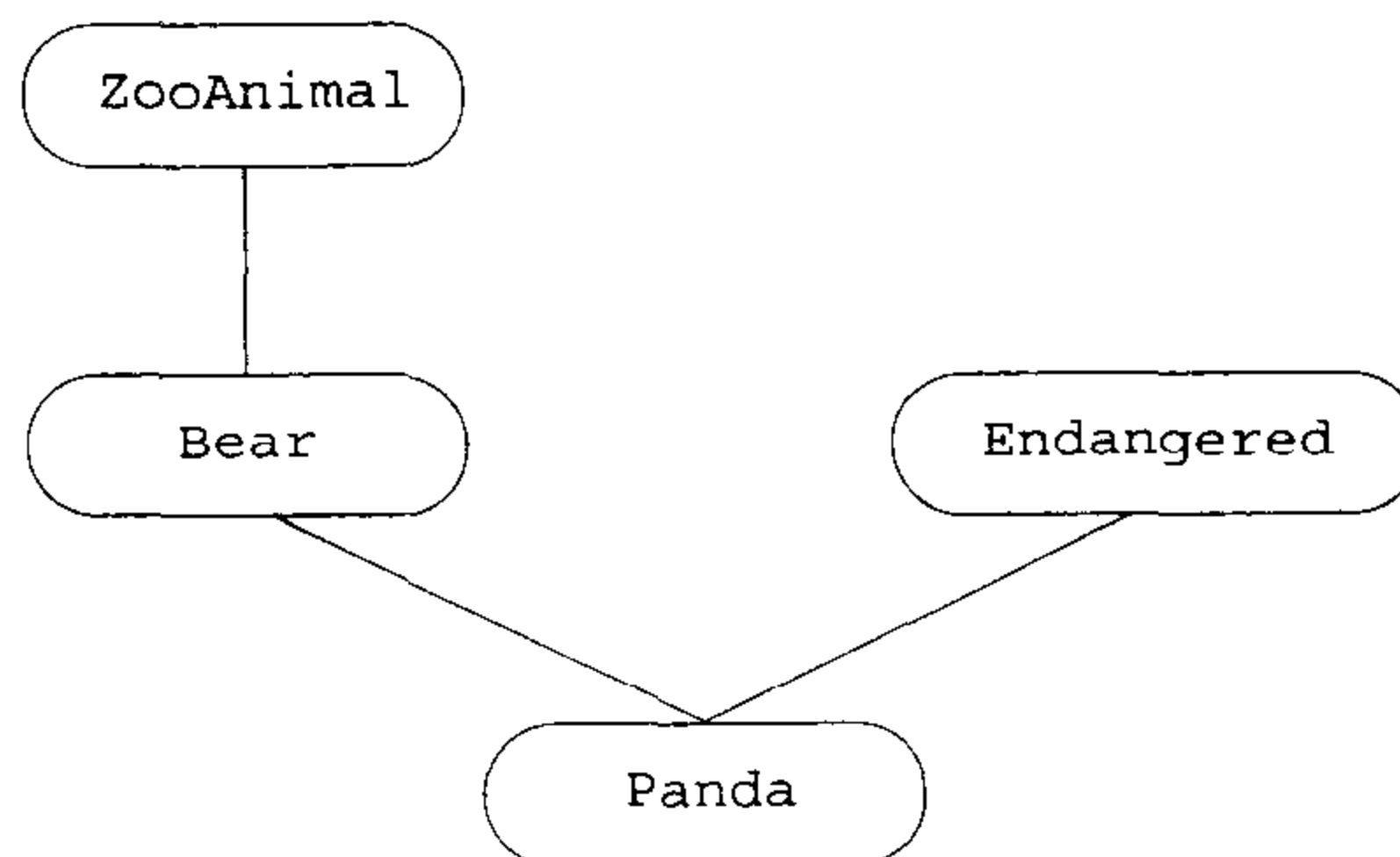


图 17-2 多重继承的 Panda 层次

3. 派生类构造函数初始化所有基类

构造派生类型的对象包括构造和初始化它的所有基类子对象。像继承单个基类（15.4.1 节）的情况一样，派生类的构造函数可以在构造函数初始化式中给零个或多个基类传递值：

```
// explicitly initialize both base classes
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) { }
// implicitly use Bear default constructor to initialize the Bear subobject
Panda::Panda()
    : Endangered(Endangered::critical) { }
```

4. 构造的次序

732

构造函数初始化式只能控制用于初始化基类的值，不能控制基类的构造次序。基类构造函数按照基类构造函数在类派生列表中的出现次序调用。对 Panda 而言，基类初始化的次序是：

- (1) ZooAnimal，从 Panda 的直接基类 Bear 沿层次向上的最终基类。
- (2) Bear，第一个直接基类。
- (3) Endangered，第二个直接基类，它本身没有基类。
- (4) Panda，初始化 Panda 本身的成员，然后运行它的构造函数的函数体。



构造函数调用次序既不受构造函数初始化列表中出现的基类的影响，也不受基类在构造函数初始化列表中的出现次序的影响。

例如，在 Panda 类的默认构造函数中，隐式调用 Bear 类的默认构造函数，它不出现在构造函数初始化列表中，虽然如此，仍在显式列出的 Endangered 类构造函数之前调用 Bear 类的默认构造函数。

5. 析构的次序

总是按构造函数运行的逆序调用析构函数。在我们的例子中，调用析构函数的次序是 ~Panda、~Endangered、~Bear、~ZooAnimal。

习题

习题 17.23 如果有，下面哪些声明是错误的。解释为什么。

- (a) `class CADVehicle : public CAD, Vehicle { ... };`
- (b) `class DoublyLinkedList:`
`public List, public List { ... };`
- (c) `class iostream: public istream, public ostream { ... };`

习题 17.24 给定下面的类层次，其中，每个类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，构造函数的执行次序是什么？

```
MI mi;
```

733

17.3.2 转换与多个基类

在单个基类情况下，派生类的指针或引用可以自动转换为基类的指针或引用，对于多重继承也是如此，派生类的指针或引用可以转换为其任意基类的指针或引用。例如，Panda 指针或引用可以转换为 ZooAnimal、Bear 或 Endangered 的指针或引用。

```
// operations that take references to base classes of type Panda
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang("ying_yang"); // create a Panda object
print(ying_yang); // passes Panda as reference to Bear
highlight(ying_yang); // passes Panda as reference to Endangered
cout << ying_yang << endl; // passes Panda as reference to ZooAnimal
```

在多重继承情况下，遇到二义性转换的可能性更大。编译器不会试图根据派生类转换来区别基类间的转换，转换到每个基类都一样好。例如，如果有 print 函数的重载版本：

```
void print(const Bear&);
void print(const Endangered&);
```

通过 Panda 对象对 print 函数的未限定调用

```
Panda ying_yang("ying_yang");
print(ying_yang); // error: ambiguous
```

导致一个编译时错误，指出该调用是二义性的。

习题

习题 17.25 给定下面的类层次，其中每个类定义了一个默认构造函数，

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : private B { ... };
class D : public X, public C { ... };
```

如果有，下面转换中哪些是不允许的？

```
D *pd = new D;
(a) X *px = pd; (c) B *pb = pd;
(b) A *pa = pd; (d) C *pc = pd;
```

734

1. 多重继承下的虚函数

为了看看多重继承怎样影响虚函数机制，假定我们的类定义表 17-2 中列出的虚成员。

表 17-2 ZooAnimal/Endangered 类中的虚函数

函 数	定义自己版本的类
print	ZooAnimal::ZooAnimal
	Bear::Bear
	Endangered::Endangered
	Panda::Panda
highlight	Endangered::Endangered
	Panda::Panda
toes	Bear::Bear
	Panda::Panda
cuddle	Panda::Panda
析构造函数	ZooAnimal::ZooAnimal
	Endangered::Endangered

2. 基于指针类型或引用类型的查找

像单继承一样，用基类的指针或引用只能访问基类中定义（或继承）的成员，不能访问派生类中引入的成员。

当一个类继承于多个基类的时候，那些基类之间没有隐含的关系，不允许使用一个基类的指针访问其他基类的成员。

作为例子，我们可以使用 Bear、ZooAnimal、Endangered 或 Panda 的指针或引用来访问 Panda 对象。所用指针的类型决定了可以访问哪些操作。如果使用 ZooAnimal 指针，只能使用 ZooAnimal 类中定义的操作，不能访问 Panda 接口的 Bear 特定、Panda 特定和 Endangered 部分。类似地，Bear 指针或引用只知道 Bear 和 ZooAnimal 成员，Endangered 指针或引用局限于 Endangered 成员：

```
Bear *pb = new Panda("ying_yang");
pb->print(cout);      // ok: Panda::print(ostream&)
pb->cuddle();          // error: not part of Bear interface
pb->highlight();       // error: not part of Bear interface
delete pb;            // ok: Panda::~~Panda()
```

如果将 Panda 对象赋给 ZooAnimal 指针，这个调用集合将用完全相同的方式确定。

735

在通过 Endangered 指针或引用使用 Panda 对象的时候，不能访问 Panda 接口的 Panda 特定的部分和 Bear 部分：

```
Endangered *pe = new Panda("ying_yang");
pe->print(cout);      // ok: Panda::print(ostream&)
pe->toes();           // error: not part of Endangered interface
pe->cuddle();          // error: not part of Endangered interface
pe->highlight();       // ok: Endangered::highlight()
delete pe;            // ok: Panda::~~Panda()
```

3. 确定使用哪个虚析构造函数

假定所有根基类都将它们的析构造函数适当定义为虚函数，那么，无论通过哪种指针类型删除对象，虚析构造函数的处理都是一致的：

```
// each pointer points to a Panda
delete pz; // pz is a ZooAnimal*
```

```
delete pb; // pb is a Bear*
delete pp; // pp is a Panda*
delete pe; // pe is a Endangered*
```

假定这些指针每个都指向 Panda 对象，则每种情况下发生完全相同的析构函数调用次序。析构函数调用的次序是构造函数次序的逆序：通过虚机制调用 Panda 析构函数。随着 Panda 析构函数的执行，依次调用 Endangered、Bear 和 ZooAnimal 的析构函数。

习题

习题 17.26 本节给出了一系列通过指向 Panda 对象的 Bear 指针所进行的调用。我们指出，如果指针是 ZooAnimal 指针，将以同样方式确定函数调用。解释为什么。

习题 17.27 假定有两个基类 Base1 和 Base2，其中每一个定义了一个名为 print 的虚成员和一个虚析构函数。从这些基类派生下面的类，其中每一个类都重定义了 print 函数：

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

使用下面的指针确定在每个调用中使用哪个函数：

```
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;
```

习题 17.28 编写与图 17-2 对应的类定义。

736

17.3.3 多重继承派生类的复制控制

多重继承的派生类的逐个成员初始化、赋值和析构（第 13 章），表现得与单继承下的一样，使用基类自己的复制构造函数、赋值操作符或析构函数隐式构造、赋值或撤销每个基类。

假定 Panda 类使用默认复制控制成员。ling_ling 的初始化

```
Panda ying_yang("ying_yang"); // create a Panda object
Panda ling_ling = ying_yang; // uses copy constructor
```

使用默认复制构造函数调用 Bear 复制构造函数，Bear 复制构造函数依次在执行 Bear 复制构造函数之前运行 ZooAnimal 复制构造函数。一旦构造了 ling_ling 的 Bear 部分，就运行 Endangered 复制构造函数来创建对象的那个部分。最后，运行 Panda 复制构造函数。

合成的赋值操作符的行为类似于复制构造函数，它首先对对象的 Bear 部分进行赋值，并通过 Bear 对对象的 ZooAnimal 部分进行赋值，然后，对 Endangered 部分进行赋值，最后对 Panda 部分进行赋值。

合成的析构函数撤销 Panda 对象的每个成员，并且按构造次序的逆序为基类部分调用析构函数。



像单继承(15.4.3节)的情况一样,如果具有多个基类的类定义了自己的析构函数,该析构函数只负责清除派生类。如果派生类定义了自己的复制构造函数或赋值操作符,则类负责复制(赋值)所有的基类子部分。只有派生类使用复制构造函数或赋值操作符的合成版本,才自动复制或赋值基类部分。

17.3.4 多重继承下的类作用域

在多重继承下,类作用域(15.5节)更加复杂,因为多个基类作用域可以包围派生类作用域。通常,成员函数中使用的名字的查找首先在函数本身进行,如果不能在本地找到名字,就继续在成员的类中查找,然后依次查找每个基类。在多重继承下,查找同时检查所有的基类继承子树——在我们的例子中,并行查找 Endangered 子树和 Bear/ZooAnimal 子树。如果在多个子树中找到该名字,则那个名字的使用必须显式指定使用哪个基类;否则,该名字的使用是二义性的。



737

当一个类有多个基类的时候,通过所有直接基类同时进行名字查找。多重继承的派生类有可能从两个或多个基类继承同名成员,对该成员不加以限定的使用是二义性的。

1. 多个基类可能导致二义性

假定 Bear 类和 Endangered 类都定义了名为 print 的成员,如果 Panda 类没有定义该成员,则

```
ying_yang.print(cout);
```

这样的语句将导致编译时错误。

Panda 类的派生(它导致有两个名为 print 的成员)是完全合法的。派生只是导致潜在的二义性,如果没有 Panda 对象调用 print,就可以避免这个二义性。如果每个 print 调用明确指出想要哪个版本——Bear::print 还是 Endangered::print,也可以避免错误。只有在存在使用该成员的二义性尝试的时候,才会出错。

如果只在一个基类子树中找到声明,则标识符得以确定而查找算法结束。例如,Endangered 类可能有一个操作返回给定其对象的估计数目,如果是这样,下面的调用

```
ying_yang.population();
```

可以顺利编译,名字 population 将在基类 Endangered 中找到,并且在 Bear 类或其任意基类中都不会出现。

2. 首先发生名字查找

虽然两个继承的 print 成员的二义性相当明显,但是也许更令人惊讶的是,即使两个继承的函数有不同的形参表,也会产生错误。类似地,即使函数在一个类中是私有的而在另一个类中是公用或受保护的,也是错误的。最后,如果在 ZooAnimal 类中定义了 print 而 Bear 类中没有定义,调用仍是错误的。

名字查找总是以两个步骤发生(7.8.1节):首先编译器找到一个匹配的声明(或者,在这个例子中,找到两个匹配的声明,这导致二义性),然后,编译器才确定所找到的声明是否合法。

3. 避免用户级二义性

可以通过指定使用哪个类解决二义性：

```
ying_yang.Endangered::print(cout);
```

避免潜在二义性最好的方法是，在解决二义性的派生类中定义函数的一个版本。例如，应该给选择使用哪个 print 版本的 Panda 类一个 print 函数：

```
std::ostream& Panda::print(std::ostream &os) const
{
    Bear::print(os);           // print the Bear part
    Endangered::print(os);     // print the Endangered part
    return os;
}
```

738

本节习题的代码

```
struct Base1 {
    void print(int) const;
protected:
    int ival;
    double dval;
    char cval;
private:
    int *id;
};

struct Base2 {
    void print(double) const;
protected:
    double fval;
private:
    double dval;
};

struct Derived : public Base1 {
    void print(std::string) const;
protected:
    std::string sval;
    double dval;
};

struct MI : public Derived, public Base2 {
    void print(std::vector<double>);
protected:
    int *ival;
    std::vector<double> dvec;
};
```

习题

习题 17.29 给定前面给出的类层次以及下面的 MI::foo 成员函数框架，

```
int ival;
double dval;
void MI::foo(double dval) { int id; /* ... */ }
```

- 识别从 MI 中可见的成员名字。有从多个基类中都可见的名字吗？
- 识别从 MI::foo 中可见的成员的集合。

739

习题 17.30 给定前面给出的类层次，为什么下面这个 print 调用是错误的？

```
MI mi;
mi.print(42);
```

修改 MI，使得这个 print 调用可以正确编译和执行。

习题 17.31 使用前面给出的类层次，如果下面赋值中有错误的，识别哪些是错误的：

```
void MI::bar() {
    int sval;
    // exercise questions occur here ...
}
(a) dval = 3.14159;      (d) fval = 0;
(b) cval = 'a';          (e) sval = *ival; (c) id = 1;
```

习题 17.32 使用前面给出的类层次以及下面的 MI::foobar 成员函数框架

```
void MI::foobar(double cval)
{
    int dval;
    // exercise questions occur here ...
}
```

- (1) 将 Base1 的 dval 成员和 Derived 的 dval 成员的和赋给 dval 的局部实例。
- (2) 将 MI::dvec 中最后一个元素赋给 Base2::fval。
- (3) 将 Base1 的 cval 赋给 Derived 中 sval 的第一个字符。

17.3.5 虚继承

在多重继承下，一个基类可以在派生层次中出现多次。实际上，我们的程序已经使用过通过继承层次多次继承同一基类的类。

每个 IO 库类都继承了一个共同的抽象基类，那个抽象基类管理流的条件状态并保存流所读写的缓冲区。istream 和 ostream 类直接继承这个公共基类，库定义了另一个名为 iostream 的类，它同时继承 istream 和 ostream，iostream 类既可以对流进行读又可以对流进行写。IO 继承层次的简化版本如图 17-3 所示。

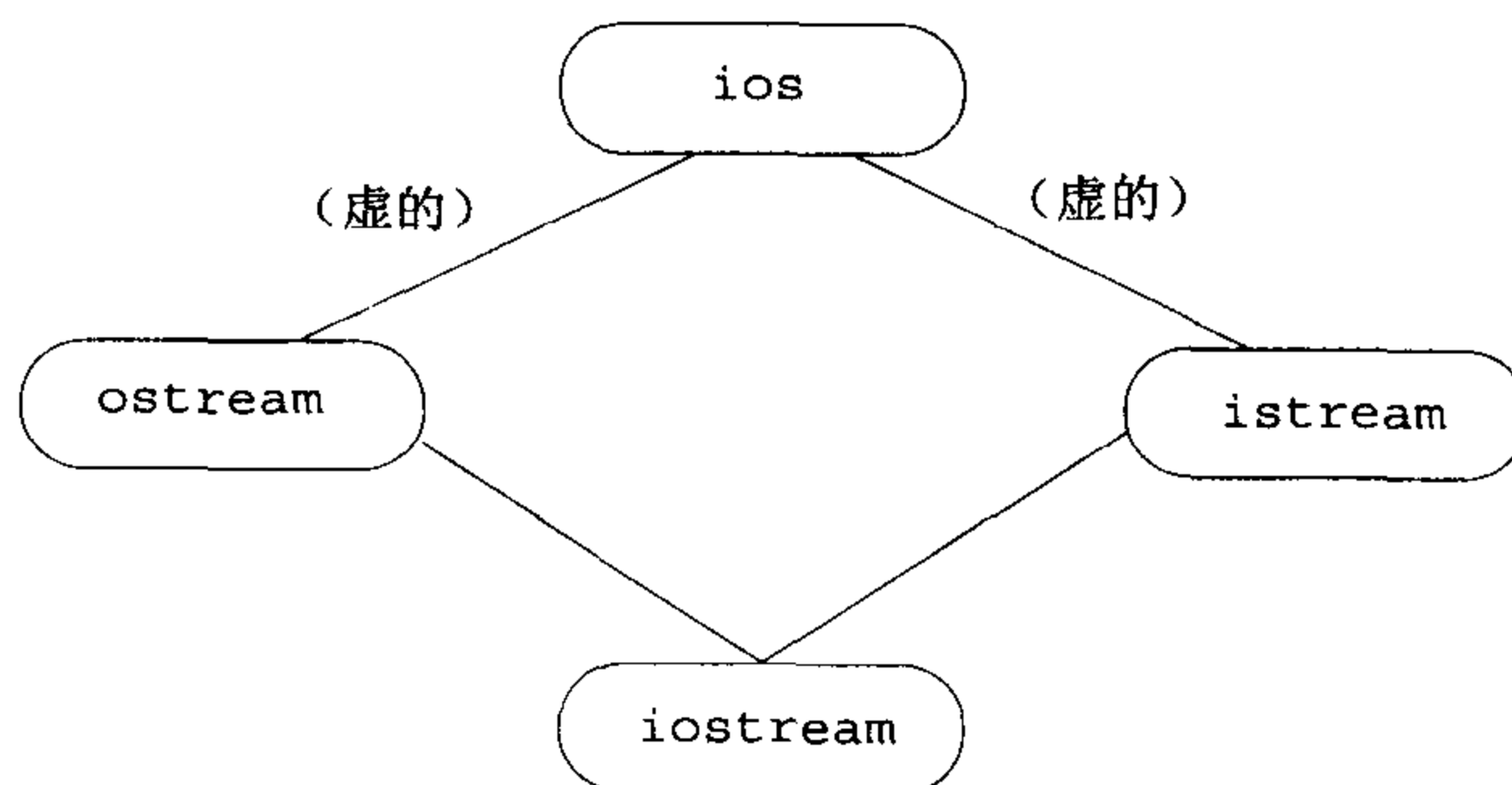


图 17-3 虚继承 iostream 层次（简化的）

像我们知道的那样，多重继承的类从它的每个父类继承状态和动作，如果 IO 类型使用常规继承，则每个 `iostream` 对象可能包含两个 `ios` 子对象：一个包含在它的 `istream` 子对象中，另一个包含在它的 `ostream` 子对象中。从设计角度讲，这个实现正是错误的：`iostream` 类想要对单个缓冲区进行读和写，它希望跨越输入和输出操作符共享条件状态。如果有两个单独的 `ios` 对象，这种共享是不可能的。

在 C++ 中，通过使用**虚继承**（virtual inheritance）解决这类问题。虚继承是一种机制，类通过虚继承指出它希望共享其虚基类的状态。在虚继承下，对给定虚基类，无论该类在派生层次中作为虚基类出现多少次，只继承一个共享的基类子对象。共享的基类子对象称为**虚基类**（virtual base class）。

`istream` 和 `ostream` 类对它们的基类进行虚继承。通过使基类成为虚基类，`istream` 和 `ostream` 指定，如果其他类（如 `iostream`）同时继承它们两个，则派生类中只出现它们的公共基类的一个副本。通过在派生列表中包含关键字 `virtual` 设置虚基类：

```
class istream : public virtual ios { ... };
class ostream : virtual public ios { ... };

// iostream inherits only one copy of its ios base class
class iostream: public istream, public ostream { ... };
```

一个不同的 Panda 类

为了举例说明虚继承，我们将继续使用 Panda 类作为教学例子。在动物学圈子中，对于 Panda 是属于 Raccoon 科还是 Bear 科已经争论了 100 年以上。因为软件设计主要是一种服务行业，所以最现实的解决方案是从二者派生 Panda：

```
class Panda : public Bear,
              public Raccoon, public Endangered {
};
```

741

虚继承 Panda 层次如图 17-4 所示。如果检查该层次，我们注意到虚继承一个不直观的特征：必须在提出虚派生的任意实际需要之前进行虚派生（在例中，Bear 类和 Raccoon 类的虚派生）。只有在使用 Panda 的声明时，虚继承才是必要的，但如果 Bear 类和 Raccoon 类不是虚派生的，Panda 类的设计者就没有好运气了。

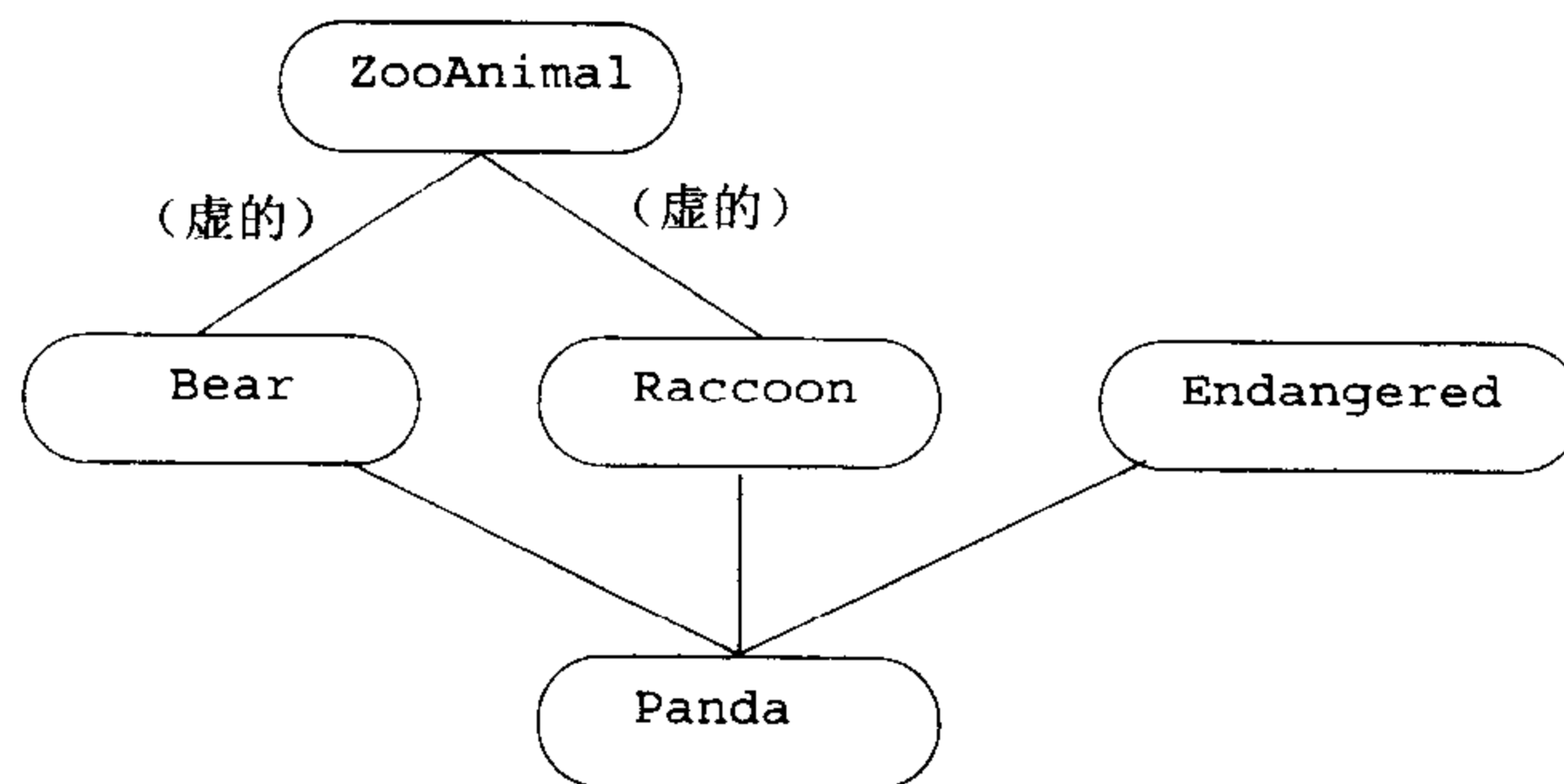


图 17-4 虚继承 Panda 层次

实际上,中间基类指定其继承为虚继承的要求很少引起任何问题。通常,使用虚继承的类层次是一次性由一个人或一个项目设计组设计的,独立开发的类很少需要其基类中的一个为虚基类,而且新基类的开发者不能改变已经存在的层次。

17.3.6 虚基类的声明

通过用关键字 `virtual` 修改声明,将基类指定为通过虚继承派生。例如,下面的声明使 `ZooAnimal` 类成为 `Bear` 类和 `Raccoon` 类的虚基类:

```
// the order of the keywords public and virtual is not significant
class Raccoon : public virtual ZooAnimal { /* ... */ };
class Bear : virtual public ZooAnimal { /* ... */ };
```



指定虚派生只影响从指定了虚基类的类派生的类。除了影响派生类自己的对象之外,它也是关于派生类与自己的未来派生类的关系的一个陈述。

742

`virtual` 说明符陈述了在后代派生类中共享指定基类的单个实例的愿望。

任何可被指定为基类的类也可以被指定为虚基类,虚基类可以包含通常由非虚基类支持的任意类元素。

1. 支持到基类的常规转换

即使基类是虚基类,也照常可以通过基类类型的指针或引用操纵派生类的对象。例如,即使 `Panda` 类将它的 `ZooAnimal` 部分作为虚基类继承,下面所有 `Panda` 的基类转换也能正确执行:

```
void dance(const Bear*);
void rummage(const Raccoon*);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang;
dance(&ying_yang);      // ok: converts address to pointer to Bear
rummage(&ying_yang);    // ok: converts address to pointer to Raccoon
cout << ying_yang;      // ok: passes ying_yang as a ZooAnimal
```

2. 虚基类成员的可见性

使用虚基类的多重继承层次比没有虚继承的引起更少的二义性问题。



可以无二义性地直接访问共享虚基类中的成员。同样,如果只沿一个派生路径重定义来自虚基类的成员,则可以直接访问该重定义成员。在非虚派生情况下,两种访问都可能是二义性的。

假定通过多个派生路径继承名为 `x` 的成员,有下面三种可能性:

(1) 如果在每个路径中 `x` 表示同一虚基类成员,则没有二义性,因为共享该成员的单个实例。
 (2) 如果在某个路径中 `x` 是虚基类的成员,而在另一路径中 `x` 是后代派生类的成员,也没有二义性——特定派生类实例的优先级高于共享虚基类实例。

(3) 如果沿每个继承路径 `x` 表示后代派生类的不同成员,则该成员的直接访问是二义性的。

像非虚多重继承层次一样,这种二义性最好用在派生类中提供覆盖实例的类来解决。

743

习题

习题 17.33 给定下面的类层次，从 VMI 类内部可以无加限定地访问哪些继承成员？哪些继承成员需要限定？解释你的推理。

```
class Base {
public:
    bar(int);
protected:
    int ival;
};
class Derived1 : virtual public Base {
public:
    bar(char);
    foo(char);
protected:
    char cval;
};
class Derived2 : virtual public Base {
public:
    foo(int);
protected:
    int ival;
    char cval;
};
class VMI : public Derived1, public Derived2 { };
```

17.3.7 特殊的初始化语义

通常，每个类只初始化自己的直接基类。在应用于虚基类的时候，这个初始化策略会失败。如果使用常规规则，就可能会多次初始化虚基类。类将沿着包含该虚基类的每个继承路径初始化。在 ZooAnimal 示例中，使用常规规则将导致 Bear 类和 Raccoon 类都试图初始化 Panda 对象的 ZooAnimal 类部分。

为了解决这个重复初始化问题，从具有虚基类的类继承的类对初始化进行特殊处理。在虚派生中，由最低层派生类的构造函数初始化虚基类。在我们的例子中，当创建 Panda 对象的时候，只有 Panda 构造函数控制怎样初始化 ZooAnimal 基类。

虽然由最低层派生类初始化虚基类，但是任何直接或间接继承虚基类的类一般也必须为该基类提供自己的初始化式。只要可以创建虚基类派生类类型的独立对象，该类就必须初始化自己的虚基类，这些初始化式只在创建中间类型的对象时使用。

744

在我们的层次中，可以有 Bear、Raccoon 或 Panda 类型的对象。创建 Panda 对象的时候，它是最低层派生类型并控制共享的 ZooAnimal 基类的初始化；创建 Bear 对象（或 Raccoon 对象）的时候，不涉及更低层的派生类型。在这种情况下，Bear（或 Raccoon）构造函数像平常一样直接初始化它们的 ZooAnimal 基类：

```
Bear::Bear(std::string name, bool onExhibit):
    ZooAnimal(name, onExhibit, "Bear") { }
Raccoon::Raccoon(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Raccoon") { }
```

虽然 ZooAnimal 不是 Panda 的直接基类，但是 Panda 构造函数也初始化 ZooAnimal 基类：

```
Panda::Panda(std::string name, bool onExhibit)
: ZooAnimal(name, onExhibit, "Panda"),
  Bear(name, onExhibit),
  Raccoon(name, onExhibit),
  Endangered(Endangered::critical),
  sleeping_flag(false) { }
```

创建 Panda 对象的时候，这个构造函数初始化 Panda 对象的 ZooAnimal 部分。

1. 怎样构造虚继承的对象

让我们看看虚继承情况下怎样构造对象。

```
Bear winnie("pooh");           // Bear constructor initializes ZooAnimal
Raccoon meeko("meeko");        // Raccoon constructor initializes ZooAnimal
Panda yolo("yolo");             // Panda constructor initializes ZooAnimal
```

当创建 Panda 对象的时候，

- (1) 首先使用构造函数初始化列表中指定的初始化式构造 ZooAnimal 部分。
- (2) 接下来，构造 Bear 部分。忽略 Bear 的用于 ZooAnimal 构造函数初始化列表的初始化式。
- (3) 然后，构造 Raccoon 部分，再次忽略 ZooAnimal 初始化式。
- (4) 最后，构造 Panda 部分。

如果 Panda 构造函数不显式初始化 ZooAnimal 基类，就使用 ZooAnimal 默认构造函数；如果 ZooAnimal 没有默认构造函数，则代码出错。当编译 Panda 构造函数的定义时，编译器将给出一个错误信息。

745

2. 构造函数与析构函数次序



无论虚基类出现在继承层次中任何地方，总是在构造非虚基类之前构造虚基类。

例如，下面毫无规律的 TeddyBear（图 17-5）派生中，有两个虚基类：ToyAnimal 基类和派生 Bear 的间接基类 ZooAnimal：

```
class Character { /* ... */ };
class BookCharacter : public Character { /* ... */ };
class ToyAnimal { /* ... */ };
class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
{ /* ... */ };
```

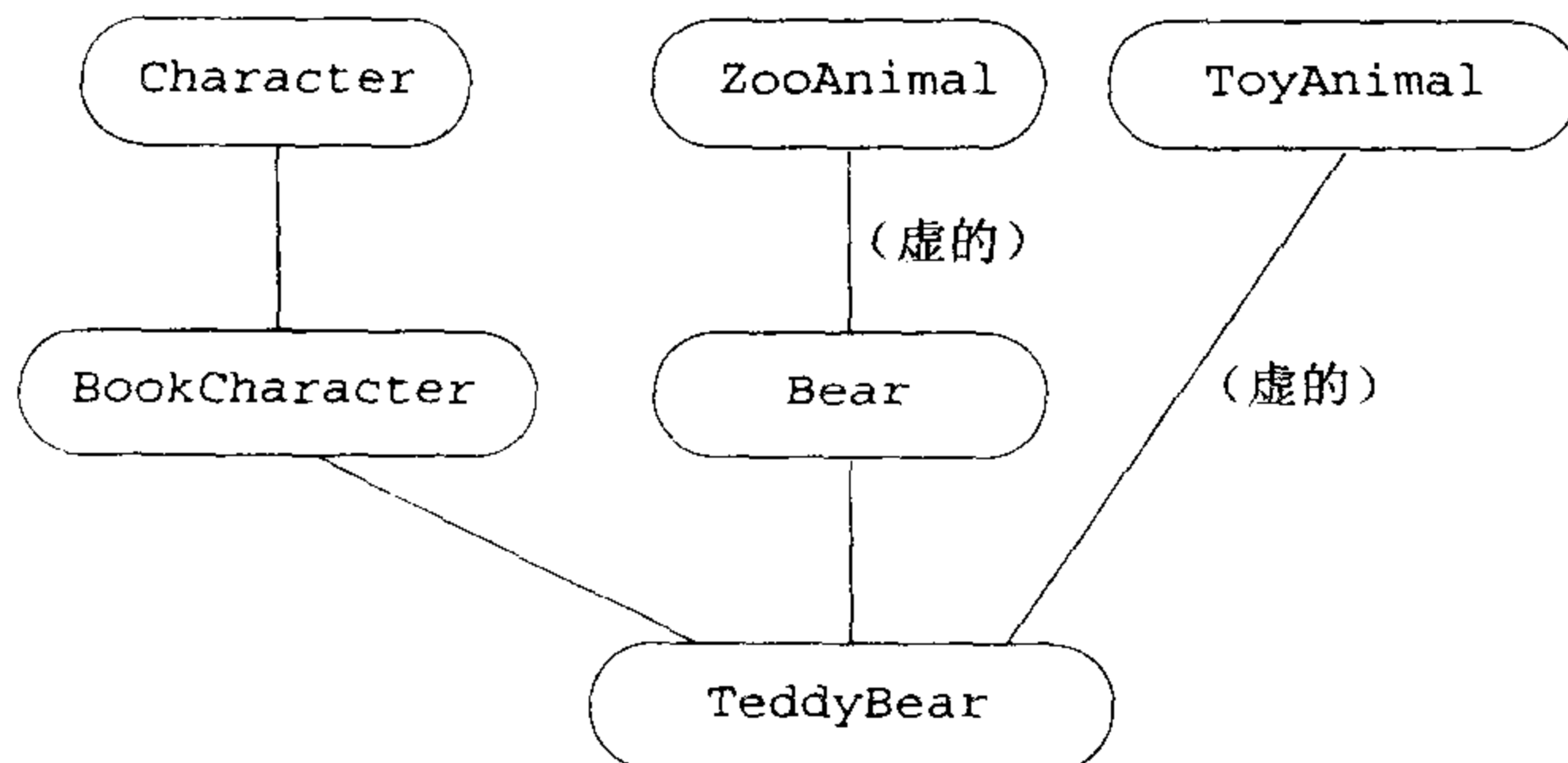


图 17-5 虚继承 TeddyBear 层次

按声明次序检查直接基类，确定是否存在虚基类。例中，首先检查 BookCharacter 的继承子树，然后检查 Bear 的继承子树，最后检查 ToyAnimal 的继承子树。按从根类开始向下到最低层派生类的次序检查每个子树。

TeddyBear 的虚基类的构造次序是先 ZooAnimal 再 ToyAnimal。一旦构造了虚基类，就按声明次序调用非虚基类的构造函数：首先是 BookCharacter，它导致调用 Character 构造函数，然后是 Bear。因此，为了创建 TeddyBear 对象，按下面次序调用构造函数：

746

```
ZooAnimal();           // Bear's virtual base class
ToyAnimal();           // immediate virtual base class
Character();           // BookCharacter's nonvirtual base class
BookCharacter();       // immediate nonvirtual base class
Bear();               // immediate nonvirtual base class
TeddyBear();          // most derived class
```

在这里，由最低层派生类 TeddyBear 指定用于 ZooAnimal 和 ToyAnimal 的初始化式。

在合成复制构造函数中使用同样的构造次序，在合成赋值操作符中也是按这个次序给基类赋值。保证调用基类析构函数的次序与构造函数的调用次序相反。

习题

习题 17.34 有一种情况下派生类不必为虚基类提供初始化式，这种情况是什么？

习题 17.35 给定下面类层次，

```
class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
           public Derived2 { ... };
class Final : public MI, public Class { ... };
```

- 对于 Final 对象的定义，构造函数和析构函数的次序是什么？
- 一个 Final 对象中有几个 Base 子对象？有几个 Class 子对象？
- 下面哪个赋值在编译时有错？

```
Base      *pb;      Class      *pc;
MI        *pmi;     Derived2   *pd2;
(a) pb = new Class;      (c) pmi = pb;
(b) pc = new Final;      (d) pd2 = pmi;
```

习题 17.36 给定前面的层次，并假定 Base 定义了下面三个构造函数，定义从 Base 派生的类，给每个类同样的三个构造函数，每个构造函数使用实参初始化其 Base 部分。

```
struct Base {
    Base();
    Base(std::string);
    Base(const Base&);
protected:
    std::string name;
};
```

747

小结

C++的大部分特征都可以应用于范围很广的问题——从几小时便可解决的问题到大团队花几年时间才能解决的问题。其中有一些特征则最适用于大规模问题的情况，这些特征包括异常处理、命名空间、多重继承和虚继承。

通过异常处理我们能够将程序的错误检测部分与错误处理部分分开。异常处理是在 6.13 节中引入的，本章最终完成了对异常处理的讨论。在抛出异常的时候，会终止当前正在执行的函数并开始查找最近的 catch 子句，在查找 catch 子句的时候，作为异常处理的一部分，将撤销退出函数内部定义的局部变量。这种撤销对象提供了一个重要的编程技术，称为“资源分配即初始化”(RAII)。

命名空间是一种机制，用于管理用独立供应商开发的代码建立的大型复杂应用程序。一个命名空间就是一个作用域，其中可以定义对象、类型、函数、模板和其他命名空间。标准库就定义在名为 std 的命名空间中。

通过 using 声明，当前作用域中就都可以访问某个命名空间中的名字了。当然，也可以通过 using 指示将一个命名空间中的所有名字带入当前作用域，但这种做法很不安全。

从概念上来看，多重继承很简单：派生类可以继承多个直接基类，派生类对象由派生部分和每个基类所贡献的基类部分构成。虽然多重继承概念简单，但细节可能非常复杂，尤其是，继承多个基类引入了新的名字冲突可能性，并且会导致对对象基类部分中的名字的引用出现二义性。

一个类继承多个直接基类的时候，那些类有可能本身还共享另一个基类。在这种情况下，中间类可以选择使用虚继承，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

abort 异常终止程序执行的库函数。通常，由 terminate 调用 abort，程序也可以直接调用 abort。abort 定义在头文件 cstdlib 中。

auto_ptr 一个库类模板，提供对动态分配对象的异常安全的访问。不能将 auto_ptr 对象绑定到数组或者变量指针，auto_ptr 对象的复制和赋值是破坏性操作：将对象的所有权从右操作数转到左操作数。对 auto_ptr 对象进行赋值删除左操作数中的对象，因此，不能将 auto_ptr 对象存储在容器中。

catch-all (捕获所有异常子句) 异常说明符为 (...) 的 catch 子句。这种子句能够捕获任意类型的异常，它通常用于捕获为进行局部清除而

局部检测的异常。异常被重新抛出给程序的其他部分，以处理问题的基本原因。

catch clause (catch 子句) 程序中处理异常的部分，也称异常处理代码。catch 子句由关键字 catch 后接异常说明符和语句块构成。catch 内部的代码完成的必要工作来处理由异常说明符定义的类型异常。

constructor order (构造函数次序) 一般而言，应该按照类派生列表中指定的次序构造基类，派生类构造函数应该通过构造函数初始化列表显式初始化每个基类。构造函数初始化列表中指定基类的次序不影响构造基类的次序。在虚继承中，虚基类在任何其他基类之前构造，

它们按照在派生类型的派生列表中（直接或间接地）出现的次序进行构造，只有最低层派生类型可以初始化虚基类，中间基类中出现的基类构造函数初始化列表被忽略。

destructor order（析构函数次序） 应该按照构造次序的逆序撤销派生类对象——首先撤销派生部分，然后，从最后一个基类开始，撤销类派生列表中指定的类。在多重继承层次中作为基类的类通常应该将它们的析构函数定义为虚函数。

exception handler（异常处理代码） `catch` 子句的另一个名称。

exception handling（异常处理） 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，由程序中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抛出异常，错误处理部分在 `try` 块的 `catch` 子句中处理异常。

exception object（异常对象） 用于在异常的 `throw` 和 `catch` 方之间进行通信的对象。在抛出点创建该对象，该对象是被抛出表达式的副本。异常对象一直存在，直到该异常的最后一个处理代码结束。异常对象的类型是被抛出表达式的类型。

exception safe（异常安全的） 用于描述在抛出异常时表现正确的程序的术语。

exception specification（异常说明） 用于函数声明之上，指出函数抛出什么（如果有）异常类型。在用圆括号括住、以逗号分隔、跟在关键字 `throw` 之后的列表中指定异常类型。空列表表示函数不抛出异常，没有异常说明的函数可以抛出任何异常。

exception specifier（异常说明符） 说明给定 `catch` 子句将处理的异常的类型。异常说明符

的行为像形参表，由异常对象初始化它的单个形参。像参数传递一样，如果异常说明符是非引用类型，就将异常对象复制到 `catch` 中。

file static（文件静态） 用关键字 `static` 声明的局部于文件的名字。在 C 语言和标准版本之前的 C++ 中，文件中的静态声明用于声明只能在单个文件中使用的对象，C++ 不赞成文件静态，已经用未命名的命名空间代替它。

function try block（函数测试块） 是函数体的 `try` 块。关键字 `try` 出现在函数体的左花括号之前，以出现在函数体的右花括号之后的 `catch` 子句作为结束。函数测试块最经常用于包围构造函数定义，以便捕获由构造函数初始化式抛出的异常。

global namespace（全局命名空间） 每个程序中保存所有全局定义的（隐式）命名空间。

multiple inheritance（多重继承） 类有多个直接基类的继承。派生类继承所有基类的成员，通过在类派生列表中指定多个基类而定义多个基类，每个基类需要一个单独的访问标号。

namespace（命名空间） 将一个库或其他程序集合定义的所有名字聚集到单个作用域的机制。与 C++ 中其他作用域不同，命名空间作用域可以在几个部分中定义，在程序的不同部分，命名空间可以是打开的、关闭的和重新打开的。

namespace alias（命名空间别名） 为给定命名空间定义同义词的机制。`namespace N1 = N;` 将 `N1` 定义为名为 `N` 的命名空间的另一名字。一个命名空间可以有多个别名，并且命名空间名字和它的别名可以互换使用。

namespace pollution（命名空间污染） 用来描述类和函数的所有名字放在全局命名空间时发生什么情况的术语。如果名字是全局的，

则使用由多个独立团队编写的代码的大程序经常遇到名字冲突。

raise (引发) 经常用作抛出的同义词。C++ 程序员互换地使用“抛出”异常或“引发”异常。

rethrow (重新抛出) 一个空的 `throw`——没有指定表达式的 `throw`。只有捕获子句或者从 `catch` 直接或间接调用的函数中的重新抛出才有效，其效果是将接到的异常对象重新抛出。

scope operator (作用域操作符) 用于访问命名空间或类中名字的操作符。

stack unwinding (栈展开) 用于描述在查找 `catch` 时退出引起被抛出异常的函数的过程。在进入相应 `catch` 之前，撤销在异常之前构造的局部对象。

terminate 一个库函数。如果没有捕获到异常或者在异常处理过程中发生异常，就调用这个库函数。该函数通常调用 `abort` 函数来结束程序。

throw e 中断当前执行路径的表达式。每个 `throw` 将控制转到可以处理被抛出异常类型的最近的外围 `catch` 子句，表达式 `e` 被复制到异常对象。

try block (测试块) 由关键字 `try` 以及一个或多个 `catch` 子句包围的语句块。如果 `try` 块内部的代码引发一个异常，而一个 `catch` 子句与异常的类型匹配，则由该 `catch` 处理异常；否则，将异常传出 `try` 之外，传给调用链

中更上层的 `catch`。

unexpected 一个库函数，如果被抛出异常违反函数的异常说明，就调用该函数。

unnamed namespace (未命名的命名空间) 没有定义名字的命名空间。未命名的命名空间中定义的名字可以无须使用作用域操作符而直接访问。每个文件都具有自己的未命名的命名空间，文件中的名字在该文件之外不可见。

using declaration (using 声明) 将命名空间中单个名字注入当前作用域的机制。using `std::cout`；使得命名空间 `std` 中的名字 `cout` 在当前作用域中可见，可以无须限定符 `std::` 而使用名字 `cout`。

using directive (using 指示) 使一个命名空间中的所有名字在包含 `using` 指示和命名空间本身的最近作用域中可见的机制。

virtual base class (虚基类) 使用关键字 `virtual` 继承的基类。即使同一类在层次中作为虚基类出现多次，派生类对象中的虚基类部分也只出现一次。在非虚继承中，构造函数只能初始化自己的直接基类，当对一个类进行虚继承的时候，由最低层的派生类初始化那个类，因此最低层的派生类应包含用于其所有虚父类的初始化式。

virtual inheritance (虚继承) 多重继承的形式，这种形式中，派生类共享在层次中被包含多次的基类的一个副本。

第 18 章

特殊工具与技术

目录

18.1 优化内存分配.....	632
18.2 运行时类型识别.....	646
18.3 类成员的指针.....	653
18.4 嵌套类.....	658
18.5 联合：节省空间的类.....	662
18.6 局部类.....	665
18.7 固有的不可移植的特征.....	666
小结.....	672
术语.....	673

本书前四部分讨论了怎样使用 C++ 中有普遍用途的特征，几乎所有 C++ 程序员都有可能在某些时候使用这些特征。此外，C++ 还定义了一些更为特殊的特征，许多程序员从不（或者很少）需要使用本章所介绍的这些特征。

18.1 优化内存分配

C++的内存分配是一种类型化操作：`new`（5.11 节）为特定类型分配内存，并在新分配的内存中构造该类型的一个对象。`new` 表达式自动运行合适的构造函数来初始化每个动态分配的类类型对象。

`new` 基于每个对象分配内存的事实可能会对某些类强加不可接受的运行时开销，这样的类可能需要使用户级的类类型对象分配能够更快一些。这样的类使用的通用策略是，预先分配用于创建新对象的内存，需要时在预先分配的内存中构造每个新对象。

另外一些类希望按最小尺寸为自己的数据成员分配需要的内存。例如，标准库中的 `vector` 类预先分配（9.4 节）额外内存以保存加入的附加元素，将新元素加入到这个保留容量中。将元素保持在连续内存中的时候，预先分配的元素使 `vector` 能够高效地加入元素。

在每种情况下（预先分配内存以保存用户级对象或者保存类的内部数据）都需要将内存分配与对象构造分离开。将内存分配与对象构造分离开的明显的理由是，在预先分配的内存中构造对象很浪费，可能会创建从不使用的对象。当实际使用预先分配的对象的时候，被使用的对象必须重新赋以新值。更微妙的是，如果预先分配的内存必须被构造，某些类就不能使用它。例如，考虑 `vector`，它使用了预先分配策略。如果必须构造预先分配的内存中的对象，就不能有基类型为没有默认构造函数的 `vector`——`vector` 没有办法知道怎样构造这些对象。



本节提出的技术不保证使所有程序更快。即使它们确实能改善性能，也可能带来其他开销，如空间的使用或调试困难。最好将优化推迟到已知程序能够工作，并且运行时测度指出改进内存分配将解决已知的性能问题的时候。

18.1.1 C++中的内存分配

C++中，内存分配和对象构造紧密纠缠，就像对象析构和内存回收一样。使用 `new` 表达式的时候，分配内存，并在该内存中构造一个对象；使用 `delete` 表达式的时候，调用析构函数撤销对象，并将对象所用内存返还给系统。

接管内存分配时，必须处理这两个任务。分配原始内存时，必须在该内存中构造对象；在释放该内存之前，必须保证适当地撤销这些对象。

754



对未构造的内存中的对象进行赋值而不是初始化，其行为是未定义的。对许多类而言，这样做引起运行时崩溃。赋值涉及删除现存对象，如果没有现存对象，赋值操作符中的动作就会有灾难性效果。

C++提供下面两种方法分配和释放未构造的原始内存。

(1) **allocator** 类，它提供可感知类型的内存分配。这个类支持一个抽象接口，以分配内存并随后使用该内存保存对象。

(2) 标准库中的 `operator new` 和 `operator delete`，它们分配和释放需要大小的原始的、

未类型化的内存。

C++还提供不同的方法在原始内存中构造和撤销对象。

(1) `allocator` 类定义了名为 `construct` 和 `destroy` 的成员，其操作正如它们的名字所指出的那样：`construct` 成员在未构造内存中初始化对象，`destroy` 成员在对象上运行适当的析构函数。

(2) 定位 `new` 表达式 (`placement new expression`) 接受指向未构造内存的指针，并在该空间中初始化一个对象或一个数组。

(3) 可以直接调用对象的析构函数来撤销对象。运行析构函数并不释放对象所在的内存。

(4) 算法 `uninitialized_fill` 和 `uninitialized_copy` 像 `fill` 和 `copy` 算法一样执行，除了它们在目的地构造对象而不是给对象赋值之外。



现代 C++ 程序一般应该使用 `allocator` 类来分配内存，它更安全更灵活。但是，在构造对象的时候，用 `new` 表达式比 `allocator::construct` 成员更灵活。有几种情况下必须使用 `new`。

18.1.2 `allocator` 类

`allocator` 类是一个模板，它提供类型化的内存分配以及对象构造与撤销。表 18-1 概述了 `allocator` 类支持的操作。

表 18-1 标准 `allocator` 类与定制算法

<code>allocator<T> a;</code>	定义名为 <code>a</code> 的 <code>allocator</code> 对象，可以分配内存或构造 <code>T</code> 类型的对象
<code>a.allocate(n)</code>	分配原始的未构造内存以保存 <code>T</code> 类型的 <code>n</code> 个对象
<code>a.deallocate(p, n)</code>	释放内存，在名为 <code>p</code> 的 <code>T*</code> 指针中包含的地址处保存 <code>T</code> 类型的 <code>n</code> 个对象。运行调用 <code>deallocate</code> 之前在该内存中构造的任意对象的 <code>destroy</code> 是用户的责任
<code>a.construct(p, t)</code>	在 <code>T*</code> 指针 <code>p</code> 所指内存中构造一个新元素。运行 <code>T</code> 类型的复制构造函数用 <code>t</code> 初始化该对象
<code>a.destroy(p)</code>	运行 <code>T*</code> 指针 <code>p</code> 所指对象的析构函数
<code>uninitialized_copy(b, e, b2)</code>	从迭代器 <code>b</code> 和 <code>e</code> 指出的输入范围将元素复制到从迭代器 <code>b2</code> 开始的未构造的原始内存中。该函数在目的地构造元素，而不是给它们赋值。假定由 <code>b2</code> 指出的目的地足以保存输入范围中元素的副本
<code>uninitialized_fill(b, e, t)</code>	将由迭代器 <code>b</code> 和 <code>e</code> 指出的范围中的对象初始化为 <code>t</code> 的副本。假定该范围是未构造的原始内存。使用复制构造函数构造对象
<code>uninitialized_fill_n(b, e, t, n)</code>	将由迭代器 <code>b</code> 和 <code>e</code> 指出的范围中至多 <code>n</code> 个对象初始化为 <code>t</code> 的副本。假定范围至少为 <code>n</code> 个元素大小。使用复制构造函数构造对象

`allocator` 类将内存分配和对象构造分开。当 `allocator` 对象分配内存的时候，它分配适当大小并排列成保存给定类型对象的空间。但是，它分配的内存是未构造的，`allocator` 的用户必须分别 `construct` 和 `destroy` 放置在该内存中的对象。

1. 使用 `allocator` 管理类成员数据

为了理解可以怎样使用预分配策略以及 `allocator` 类来管理类的内部数据需要，让我们再

想想 vector 类中的内存分配会怎样工作。

回忆一下, vector 类将元素保存在连续的存储中。为了获得可接受的性能, vector 预先分配比所需元素更多的元素 (9.4 节)。每个将元素加到容器中的 vector 成员检查是否有可用空间以容纳另一元素。如果有, 该成员在预分配内存中下一可用位置初始化一个对象; 如果没有自由元素, 就重新分配 vector: vector 获取新的空间, 将现存元素复制到该空间, 增加新元素, 并释放旧空间。

756

vector 所用存储开始是未构造内存, 它还没有保存任何对象。将元素复制或增加到这个预分配空间的时候, 必须使用 allocator 类的 construct 成员构造元素。

为了说明这些概念, 我们将实现 vector 的一小部分。将我们的类命名为 Vector, 以区别于标准类 vector:

```
// pseudo-implementation of memory allocation strategy for a vector-like class
template <class T> class Vector {
public:
    Vector(): elements(0), first_free(0), end(0) { }
    void push_back(const T&);
    // ...
private:
    static std::allocator<T> alloc; // object to get raw memory
    void reallocate(); // get more space and copy existing elements
    T* elements; // pointer to first element in the array
    T* first_free; // pointer to first free element in the array
    T* end; // pointer to one past the end of the array
    // ...
};
```

每个 Vector<T> 类型定义一个 allocator<T> 类型的 static 数据成员, 以便在给定类型的 Vector 中分配和构造元素。每个 Vector 对象在指定类型的内置数组中保存其元素, 并维持该数组的下列三个指针:

- elements, 指向数组的第一个元素。
- first_free, 指向最后一个实际元素之后的那个元素。
- end, 指向数组本身之后的那个元素。

图 18-1 说明了这些指针的含义。

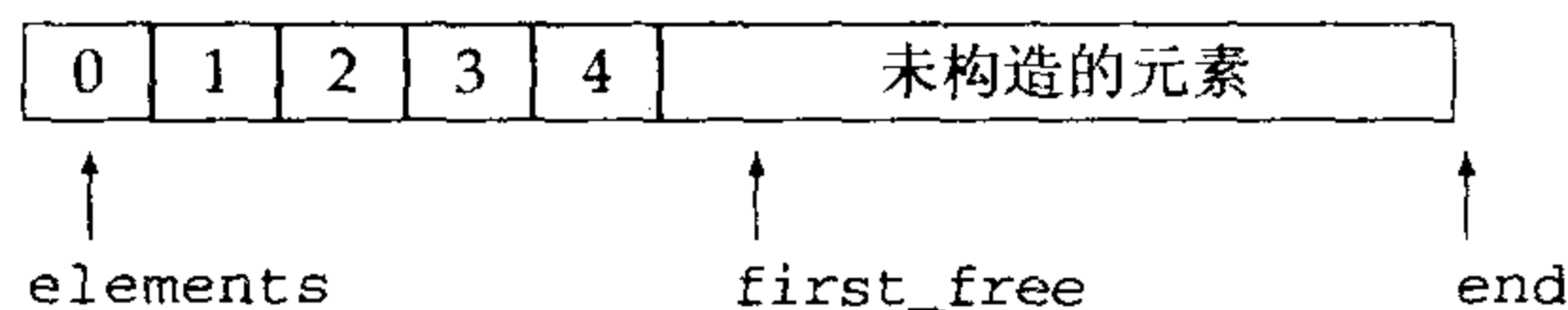


图 18-1 Vector 内存分配策略

可以使用这些指针来确定 Vector 的大小和容量:

- Vector 的 size (实际使用的元素的数目) 等于 first_free-elements。
- Vector 的 capacity (在必须重新分配 Vector 之前, 可以定义的元素总数) 等于 end-elements。
- 自由空间 (在需要重新分配之前, 可以增加的元素的数目) 是 end-first_free。

757

2. 使用 **construct**

`push_back` 成员使用这些指针将新元素加到 `Vector` 末尾:

```
template <class T>
void Vector<T>::push_back(const T& t)
{
    // are we out of space?
    if (first_free == end)
        reallocate(); // gets more space and copies existing elements to it
    alloc.construct(first_free, t);
    ++first_free;
}
```

`push_back` 函数首先确定是否有可用空间, 如果没有, 就调用 `reallocate` 函数, `reallocate` 分配新空间并复制现存元素, 将指针重置为指向新分配的空间。

一旦 `push_back` 函数知道还有空间容纳新元素, 它就请求 `allocator` 对象构造一个新的最后元素。`construct` 函数使用类型 `T` 的复制构造函数将 `t` 值复制到由 `first_free` 指出的元素, 然后, 将 `first_free` 加 1 以指出又有一个元素在用。

3. 重新分配元素与复制元素

`reallocate` 函数所做的工作最多:

```
template <class T> void Vector<T>::reallocate()
{
    // compute size of current array and allocate space for twice as many elements
    std::ptrdiff_t size = first_free - elements;
    std::ptrdiff_t newcapacity = 2 * max(size, 1);
    // allocate space to hold newcapacity number of elements of type T
    T* newelements = alloc.allocate(newcapacity);

    // construct copies of the existing elements in the new space
    uninitialized_copy(elements, first_free, newelements);
    // destroy the old elements in reverse order
    for (T *p = first_free; p != elements; /* empty */ )
        alloc.destroy(--p);

    // deallocate cannot be called on a 0 pointer
    if (elements)
        // return the memory that held the elements
        alloc.deallocate(elements, end - elements);
    // make our data structure point to the new elements
    elements = newelements;
    first_free = elements + size;
    end = elements + newcapacity;
}
```

我们使用一个简单但效果惊人的策略: 每次重新分配时分配两倍内存。函数首先计算当前在用的元素数目, 将该数目翻倍, 并请求 `allocator` 对象来获得所需数量的空间。如果 `Vector` 为空, 就分配两个元素。

如果 `Vector` 保存 `int` 值, `allocate` 函数调用为 `newcapacity` 数目的 `int` 值分配空间; 如果 `Vector` 保存 `string` 对象, 它就为给定数目的 `string` 对象分配空间。

uninitialized_copy 调用使用标准 copy 算法的特殊版本。这个版本希望目的地是原始的未构造内存，它在目的地复制构造每个元素，而不是将输入范围的元素赋值给目的地，使用 T 的复制构造函数从输入范围将每个元素复制到目的地。

for 循环对旧数组中每个对象调用 allocator 的 destroy 成员，它按逆序撤销元素，从数组中最后一个元素开始，以第一个元素结束。destroy 函数运行 T 类型的析构函数来释放旧元素所用的任何资源。

一旦复制和撤销了元素，就释放原来数组所用的空间。在调用 deallocate 之前，必须检查 elements 是否实际指向一个数组。



deallocate 期待指向由 allocate 分配的空间的指针，传给 deallocate 一个零指针是不合法的。

最后，必须重置指针以指向新分配并初始化的数组。将 first_free 和 end 指针分别置为指向最后构造的元素之后的单元以及所分配空间末尾的下一单元。

习题

习题 18.1 实现自己的 Vector 类的版本，包括 vector 成员 reserve (9.4 节)、resize (9.3.5 节) 以及 const 和非 const 下标操作符 (14.5 节)。

习题 18.2 定义一个类型别名，使用对应指针类型作为 Vector 的 iterator。

习题 18.3 为了测试你的 Vector 类，重新实现前面用 vector 编写的程序，用 Vector 代替 vector。

18.1.3 operator new 函数和 operator delete 函数

前几节使用 Vector 类说明了怎样使用 allocator 类来管理用于类的内部数据存储的内存池，下面三节将介绍怎样用更基本的标准库机制实现相同的策略。

759

首先，需要对 new 和 delete 表达式怎样工作有更多的理解。当使用 new 表达式

```
// new expression
string * sp = new string("initialized");
```

的时候，实际上发生三个步骤。首先，该表达式调用名为 **operator new** 的标准库函数，分配足够大的原始的未类型化的内存，以保存指定类型的一个对象；接下来，运行该类型的一个构造函数，用指定初始化式构造对象；最后，返回指向新分配并构造的对象的指针。

```
delete sp;
```

当使用 delete 表达式 delete sp; 删除动态分配对象的时候，发生两个步骤。首先，对 sp 指向的对象运行适当的析构函数；然后，通过调用名为 **operator delete** 的标准库函数释放该对象所用内存。

术语对比: new 表达式与 operator new 函数

标准库函数 `operator new` 和 `operator delete` 的命名容易让人误解。与其他 `operator` 函数（如 `operator=`）不同，这些函数没有重载 `new` 或 `delete` 表达式，实际上，我们不能重定义 `new` 和 `delete` 表达式的行为。

通过调用 `operator new` 函数执行 `new` 表达式获得内存，并接着在该内存中构造一个对象，通过撤销一个对象执行 `delete` 表达式，并接着调用 `operator delete` 函数，以释放该对象使用的内存。



因为 `new`（或 `delete`）表达式与标准库函数同名，所以二者容易混淆。

1. operator new 和 operator delete 接口

`operator new` 和 `operator delete` 函数有两个重载版本，每个版本支持相关的 `new` 表达式和 `delete` 表达式：

```
void *operator new(size_t);           // allocate an object
void *operator new[](size_t);        // allocate an array

void *operator delete(void*);        // free an object
void *operator delete[](void*);      // free an array
```

2. 使用分配操作符函数

虽然 `operator new` 和 `operator delete` 函数的设计意图是供 `new` 表达式使用，但它们通常是标准库中的可用函数。可以使用它们获得未构造内存，它们有点类似 `allocator` 类的 `allocate` 和 `deallocate` 成员。例如，代替使用 `allocator` 对象，可以在 `Vector` 类中使用 `operator new` 和 `operator delete` 函数。在分配新空间时我们曾编写

```
// allocate space to hold newcapacity number of elements of type T
T* newelements = alloc.allocate(newcapacity);
```

这可以重新编写为

```
// allocate unconstructed memory to hold newcapacity elements of type T
T* newelements = static_cast<T*>
    (operator new[](newcapacity * sizeof(T)));
```

类似地，在重新分配由 `Vector` 成员 `elements` 指向的旧空间的时候，我们曾经编写

```
// return the memory that held the elements
alloc.deallocate(elements, end - elements);
```

这可以重新编写为

```
// deallocate the memory that they occupied
operator delete[](elements);
```

这些函数的表现与 `allocator` 类的 `allocate` 和 `deallocate` 成员类似。但是，它们在一个重要方面有不同：它们在 `void*` 指针而不是类型化的指针上进行操作。



最佳
实践

一般而言，使用 allocator 比直接使用 operator new 和 operator delete 函数更为类型安全。

allocate 成员分配类型化的内存，所以使用它的程序可以不必计算以字节为单位的所需内存量，它们也可以避免对 operator new 的返回值进行强制类型转换（5.12.4 节）。类似地，deallocate 释放特定类型的内存，也不必转换为 void*。

18.1.4 定位 new 表达式

标准库函数 operator new 和 operator delete 是 allocator 的 allocate 和 deallocate 成员的低级版本，它们都分配但不初始化内存。

allocator 的成员 construct 和 destroy 也有两个低级选择，这些成员在由 allocator 对象分配的空间中初始化和撤销对象。

类似于 construct 成员，有第三种 new 表达式，称为**定位 new**（placement new）。定位 new 表达式在已分配的原始内存中初始化一个对象，它与 new 的其他版本的不同之处在于，它不分配内存。相反，它接受指向已分配但未构造内存的指针，并在该内存中初始化一个对象。实际上，定位 new 表达式使我们能够在特定的、预分配的内存地址构造一个对象。

定位 new 表达式的形式是：

```
new (place_address) type
new (place_address) type (initializer-list)
```

其中 *place_address* 必须是一个指针，而 *initializer-list* 提供了（可能为空的）初始化列表，以便在构造新分配的对象时使用。

可以使用定位 new 表达式代替 Vector 实现中的 construct 调用。原来的代码

```
// construct a copy t in the element to which first_free points
alloc.construct (first_free, t);
```

可以用等价的定位 new 表达式代替

```
// copy t into element addressed by first_free
new (first_free) T(t);
```

定位 new 表达式比 allocator 类的 construct 成员更灵活。定位 new 表达式初始化一个对象的时候，它可以使用任何构造函数，并直接建立对象。construct 函数总是使用复制构造函数。

例如，可以用下面两种方式之一，从一对迭代器初始化一个已分配但未构造的 string 对象：

```
allocator<string> alloc;
string *sp = alloc.allocate(2); // allocate space to hold 2 strings
// two ways to construct a string from a pair of iterators
new (sp) string(b, e); // construct directly in place
alloc.construct(sp + 1, string(b, e)); // build and copy a temporary
```

定位 new 表达式使用了接受一对迭代器的 string 构造函数，在 sp 指向的空间直接构造 string 对象。当调用 construct 函数的时候，必须首先从迭代器构造一个 string 对象，以获得传递给 construct 的 string 对象，然后，该函数使用 string 的复制构造函数，将那个未命名的临时

string 对象复制到 sp 指向的对象中。

通常, 这些区别是不相干的: 对值型类而言, 在适当的位置直接构造对象与构造临时对象并进行复制之间没有可观察到的区别, 而且性能差别基本没有意义。但对某些类而言, 使用复制构造函数是不可能的 (因为复制构造函数是私有的), 或者是应该避免的, 在这种情况下, 也许有必要使用定位 new 表达式。

习题

习题 18.4 你认为为什么限制 construct 函数只能使用元素类型的复制构造函数?

习题 18.5 为什么定位 new 表达式更灵活?

18.1.5 显式析构函数的调用

正如定位 new 表达式是使用 allocator 类的 construct 成员的低级选择, 我们可以使用析构函数的显式调用作为调用 destroy 函数的低级选择。

762

在使用 allocator 对象的 Vector 版本中, 通过调用 destroy 函数清除每个元素:

```
// destroy the old elements in reverse order
for (T *p = first_free; p != elements; /* empty */ )
    alloc.destroy(--p);
```

对于使用定位 new 表达式构造对象的程序, 显式调用析构函数:

```
for (T *p = first_free; p != elements; /* empty */ )
    p->~T(); // call the destructor
```

在这里直接调用析构函数。箭头操作符对迭代器 p 解引用以获得 p 所指的对象, 然后, 调用析构函数, 析构函数以类名前加~来命名。

显式调用析构函数的效果是适当地清除对象本身。但是, 并没有释放对象所占的内存, 如果需要, 可以重用该内存空间。



调用 operator delete 函数不会运行析构函数, 它只释放指定的内存。

习题

习题 18.6 重新实现 Vector 类, 使用 operator new、operator delete、定位 new 表达式, 并直接调用析构函数。

习题 18.7 运行为原来的 Vector 实现而运行的程序, 测试你的新版本。

习题 18.8 你认为哪个版本更好? 为什么?

18.1.6 类特定的 new 和 delete

前几节介绍了类怎样能够接管自己的内部数据结构的内存管理, 另一种优化内存分配的方法涉及优化 new 表达式的行为。作为例子, 考虑第 16 章的 Queue 类。该类不直接保存它的元素,

763

相反，它使用 `new` 表达式分配 `QueueItem` 类型的对象。

通过预先分配一块原始内存以保存 `QueueItem` 对象，也许有可能改善 `Queue` 的性能。创建新 `QueueItem` 对象的时候，可以在这个预先分配的空间中构造对象。释放 `QueueItem` 对象的时候，将它们放回预先分配对象的块中，而不是将内存真正返回给系统。

这个问题与 `Vector` 的实现之间的区别在于，在这种情况下，我们希望在应用于特定类型的时候优化 `new` 和 `delete` 表达式的行为。默认情况下，`new` 表达式通过调用由标准库定义的 `operator new` 版本分配内存。通过定义自己的名为 `operator new` 和 `operator delete` 的成员，类可以管理用于自身类型的内存。

编译器看到类类型的 `new` 或 `delete` 表达式的时候，它查看该类是否有 `operator new` 或 `operator delete` 成员，如果类定义（或继承）了自己的成员 `new` 和 `delete` 函数，则使用那些函数为对象分配和释放内存；否则，调用这些函数的标准库版本。

优化 `new` 和 `delete` 的行为的时候，只需要定义 `operator new` 和 `operator delete` 的新版本，`new` 和 `delete` 表达式自己照管对象的构造和撤销。

1. 成员 `new` 和 `delete` 函数



如果类定义了这两个成员中的一个，它也应该定义另一个。

类成员 `operator new` 函数必须具有返回类型 `void*` 并接受 `size_t` 类型的形参。由 `new` 表达式用以字节计算的分配内存量初始化函数的 `size_t` 形参。

类成员 `operator delete` 函数必须具有返回类型 `void`。它可以定义为接受单个 `void*` 类型形参，也可以定义为接受两个形参，即 `void*` 和 `size_t` 类型。由 `delete` 表达式用被 `delete` 的指针初始化 `void*` 形参，该指针可以是空指针。如果提供了 `size_t` 形参，就由编译器用第一个形参所指对象的字节大小自动初始化 `size_t` 形参。

除非类是某继承层次的一部分，否则形参 `size_t` 不是必需的。当 `delete` 指向继承层次中类型的指针时，指针可以指向基类对象，也可以指向派生类对象。派生类对象的大小一般比基类对象大。如果基类有 `virtual` 析构函数（15.4.4 节），则传给 `operator delete` 的大小将根据被删除指针所指对象的动态类型而变化；如果基类没有 `virtual` 析构函数，那么，通过基类指针删除指向派生类对象的指针的行为，跟往常一样是未定义的。

764

这些函数隐式地为静态函数（12.6.1 节），不必显式地将它们声明为 `static`，虽然这样做是合法的。成员 `new` 和 `delete` 函数必须是静态的，因为它们要么在构造对象之前使用（`operator new`），要么在撤销对象之后使用（`operator delete`），因此，这些函数没有成员数据可操纵。像任意其他静态成员函数一样，`new` 和 `delete` 只能直接访问所属类的静态成员。

2. 数组操作符 `new[]` 和操作符 `delete[]`

也可以定义成员 `operator new[]` 和 `operator delete[]` 来管理类类型的数组。如果这些 `operator` 函数存在，编译器就使用它们代替全局版本。

类成员 `operator new[]` 必须具有返回类型 `void*`，并且接受的第一个形参类型为 `size_t`。

用表示存储特定类型给定数目元素的数组的字节数值自动初始化操作符的 `size_t` 形参。

成员操作符 `operator delete[]` 必须具有返回类型 `void`，并且第一个形参为 `void*` 类型。用表示数组存储起始位置的值自动初始化操作符的 `void*` 形参。

类的操作符 `delete[]` 也可以有两个形参，第二个形参为 `size_t`。如果提供了附加形参，由编译器用数组所需存储量的字节数自动初始化这个形参。

3. 覆盖类特定的内存分配

如果类定义了自己的成员 `new` 和 `delete`，类的用户就可以通过使用全局作用域确定操作符，强制 `new` 或 `delete` 表达式使用全局的库函数。如果用户编写

```
Type *p = ::new Type;    // uses global operator new
::delete p;              // uses global operator delete
```

那么，即使类定义了自己的类特定的 `operator new`，也调用全局的 `operator new`；`delete` 类似。



如果用 `new` 表达式调用全局 `operator new` 函数分配内存，则 `delete` 表达式也应该调用全局 `operator delete` 函数。

习题

习题 18.9 为 `QueueItem` 类声明成员 `new` 和 `delete`。

765

18.1.7 一个内存分配器基类



像泛型句柄类一样（16.5 节），这个例子表示 C++ 的相当复杂的使用，理解这个例子需要（并论证）良好掌握地继承和模板，将对这个例子的研究推迟到你熟悉了这些特征之后，也许是有用的。

已经看过了怎样声明类特定的成员 `new` 和 `delete`，下面就可以为 `QueueItem` 类实现这些成员，在这样做之前，需要决定怎样改进内置库的 `new` 和 `delete` 函数。一个通用策略是预先分配一块原始内存来保存未构造的对象，创建新元素的时候，可以在一个预先分配的对象中构造；释放元素的时候，将它们放回预先分配对象的块中，而不是将内存实际返还给系统。这种策略常被称为维持一个自由列表（freelist）。可以将自由列表实现为已分配但未构造的对象的链表。

除了为 `QueueItem` 类实现基于自由列表的分配策略，我们注意到 `QueueItem` 不是唯一希望优化其对象分配的类，许多类都可能有一需要。因为这个行为也许通常是有用的，所以我们将定义一个名为 `CachedObj` 的新类来处理自由列表。像 `QueueItem` 这样希望优化其对象分配的类可以使用 `CachedObj` 类，而不用直接实现自己的 `new` 和 `delete` 成员。

`CachedObj` 类有简单的接口：它的工作只是分配和管理已分配但未构造对象的自由列表。这个类将定义一个成员 `operator new`，返回自由列表的下一个元素，并将该元素从自由列表中删除。当自由列表为空的时候，`operator new` 将分配新的原始内存。这个类还定义 `operator delete`，在撤销对象时将元素放回自由列表。

希望为自己的类型使用自由列表分配策略的类将继承 `CachedObj` 类，通过继承，这些类可以使用 `CachedObj` 类的 `operator new` 和 `operator delete` 定义，以及表示自由列表所需的数据成员。因为打算将 `CachedObj` 类作为基类，所以将给它一个 `public` 虚析构函数。



正如我们将看到的，`CachedObj` 只能用于不包含在继承层次中的类型。与成员 `new` 和 `delete` 操作不同，`CachedObj` 类没有办法根据对象的实际类型分配不同大小的对象：它的自由列表保存单一大小的对象。因此，它只能用于不作基类使用的类，如 `QueueItem` 类。

由 `CachedObj` 类定义并被它的派生类继承的数据成员是：

- 指向自由列表表头的 `static` 指针。
- 名为 `next`、从一个 `CachedObj` 对象指向另一个 `CachedObj` 对象的指针。

766

`next` 指针将元素链入自由列表。从 `CachedObj` 类派生的每个类型都将包含自己的类型特定的数据，加上一个从 `CachedObj` 基类继承的指针。每个对象具有由内存分配器使用但被继承类型自己不用的一个额外指针，对象在使用的时候，该指针无意义且不使用；对象可供使用并在自由列表中的时候，就使用 `next` 指针来指向下一个可用的对象。

如果使用 `CachedObj` 类来优化 `Screen` 类的分配，`Screen` 类型的对象（概念上）看起来如图 18-2 所示。

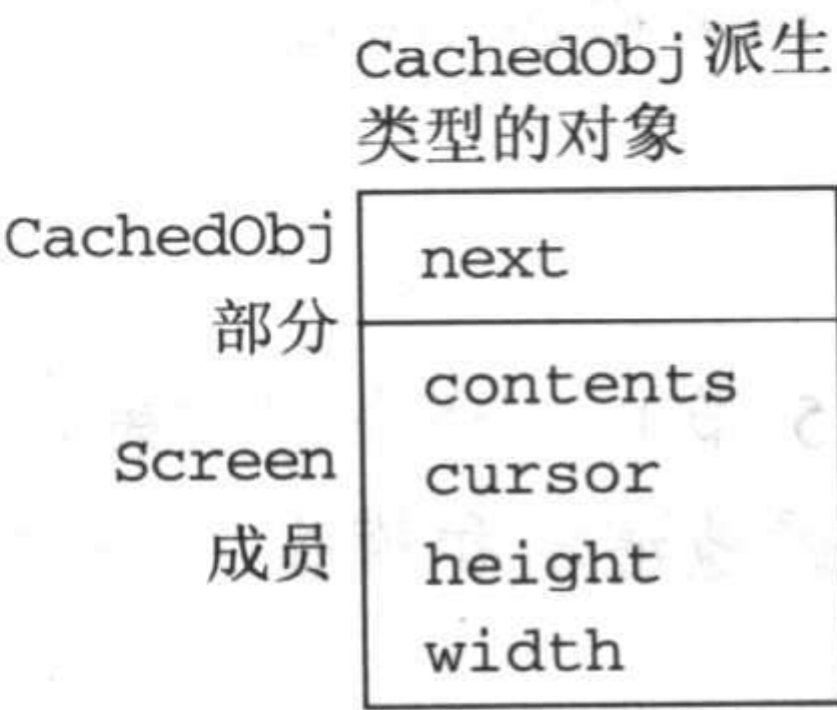


图 18-2 `CachedObj` 派生类举例

1. `CachedObj` 类

剩下的唯一问题是 `CachedObj` 类中的指针使用什么类型。我们希望为任意类型使用自由列表方法，所以 `CachedObj` 类将是一个模板，指针将指向该模板类型的对象：

```
/* memory allocation class: Pre-allocates objects and
 * maintains a freelist of objects that are unused
 * When an object is freed, it is put back on the freelist
 * The memory is only returned when the program exits
 */
template <class T> class CachedObj {
public:
    void *operator new(std::size_t);
    void operator delete(void *, std::size_t);
    virtual ~CachedObj() {}
protected:
    T *next;
private:
```

```

static void add_to_freelist(T*);
static std::allocator<T> alloc_mem;
static T *freeStore;
static const std::size_t chunk;
};

```

767

这个类相当简单，它只提供三个公用成员：`operator new`、`operator delete` 和虚析构函数。`new` 和 `delete` 成员分别从自由列表取走对象和将对象返回到自由列表。

`static` 成员管理自由列表。这些成员声明为 `static`，是因为只为所有给定类型的对象维持一个自由列表。`freeStore` 指针指向自由列表的表头。名为 `chunk` 的成员指定每当自由列表为空时将分配的对象的数目。最后，`add_to_freelist` 函数将对象放在自由列表，`operator new` 使用这个函数将新分配的对象放到自由列表，删除对象的时候，`operator delete` 也使用该函数将对象放回自由列表。

2. 使用 `CachedObj`

使用 `CachedObj` 类，真正复杂的部分是理解模板形参：当继承 `CachedObj` 类的时候，用来实例化 `CachedObj` 类的模板类型将是派生类型本身。为了重用 `CachedObj` 类的自由列表管理而继承 `CachedObj` 类，但是，`CachedObj` 类保存了指向它管理的对象类型的一个指针，该指针的类型是指向 `CachedObj` 的派生类型的指针。

例如，为了优化 `Screen` 类的内存管理，我们将 `Screen` 声明为：

```

class Screen: public CachedObj<Screen> {
    // interface and implementation members of class Screen are unchanged
};

```

这个声明给了 `Screen` 类一个新的基类：形参为 `Screen` 类型的 `CachedObj` 实例。现在除了 `Screen` 类内部定义的其他成员之外，每个 `Screen` 对象还包含附加的名为 `next` 的继承成员。

因为 `QueueItem` 是一个模板类型，从 `CachedObj` 类派生它有点复杂：

```

template <class Type>
class QueueItem: public CachedObj< QueueItem<Type> > {
    // remainder of class declaration and all member definitions unchanged
};

```

这个声明是说，`QueueItem` 是从保存 `QueueItem<Type>` 类型对象的 `CachedObj` 实例派生而来的类模板。例如，如果定义 `int` 值的 `Queue`，就从 `CachedObj<QueueItem<int>>` 派生 `QueueItem<int>` 类。



我们的类不需要其他改变。现在 `QueueItem` 类具有自动内存分配，这个内存分配使用自由列表减少创建新的 `Queue` 元素时需要的分配数目。

768

3. 分配怎样工作

因为我们从 `CachedObj` 类派生 `QueueItem` 类，任何使用 `new` 表达式的分配，如 `Queue::push` 中的调用：

```

// allocate a new QueueItem object
QueueItem<Type> *pt =
    new QueueItem<Type>(val);

```

分配并构造一个新的 QueueItem 对象。每个表达式

(1) 使用 QueueItem<T>::operator new 函数从自由列表分配一个对象。

(2) 为类型 T 使用元素类型的复制构造函数，在该内存中构造一个对象。

类似地，当像 delete pt; 这样删除一个 QueueItem 指针的时候，运行 QueueItem 析构函数清除 pt 指向的对象，并调用该类的 operator delete，将元素所用的内存放回自由列表。

4. 定义 operator new

operator new 成员从自由列表返回一个对象，如果自由列表为空，new 必须首先分配 chunk 数目的新内存：

```
template <class T>
void *CachedObj<T>::operator new(size_t sz)
{
    // new should only be asked to build a T, not an object
    // derived from T; check that right size is requested
    if (sz != sizeof(T))
        throw std::runtime_error
            ("CachedObj: wrong size object in operator new");
    if (!freeStore) {
        // the list is empty: grab a new chunk of memory
        // allocate allocates chunk number of objects of type T
        T * array = alloc_mem.allocate(chunk);
        // now set the next pointers in each object in the allocated memory
        for (size_t i = 0; i != chunk; ++i)
            add_to_freelist(&array[i]);
    }
    T *p = freeStore;
    freeStore = freeStore->CachedObj<T>::next;
    return p;    // constructor of T will construct the T part of the object
}
```

769

函数首先验证要求它分配正确数量的空间。

这个检查强调了我们的设计意图：CachedObj 类应该只被不是基类的类使用。CachedObj 类在固定大小的自由列表上分配对象，这一事实意味着，继承层次中的类不能使用它来处理内存分配。因继承而相关的类几乎总是定义不同大小的对象，处理这些类的单个分配器，可能必须比这里所实现的这个复杂得多。

operator new 函数接着检查自由列表中是否有对象，如果没有，它就请求 allocator 成员分配 chunk 个新的未构造对象，然后，它通过新分配的对象进行迭代，设置 next 指针。调用了 add_to_freelist 函数之后，自由列表上的每个对象除了将保存下一个可用对象的地址的 next 指针之外，将是未构造的。自由列表如图 18-3 所示。

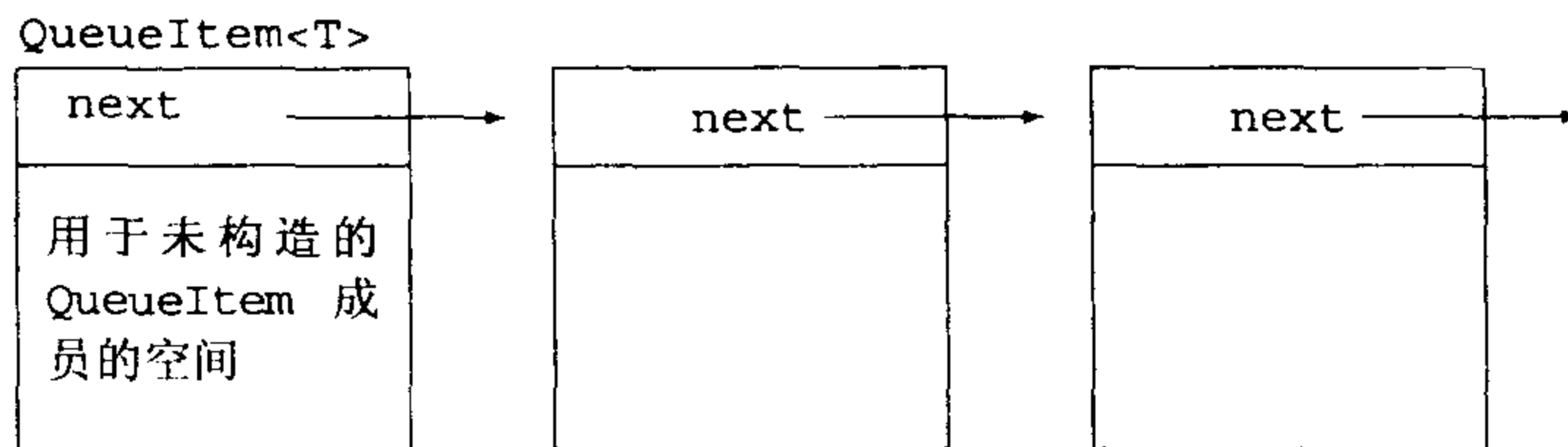


图 18-3 CachedObj 自由列表示例

在有可用对象可以分配的保证之下，operator new 返回自由列表上第一个对象的地址，将 freeStore 指针重置为指向自由列表上下一个元素。被返回的对象是未构造的。因为从 new 表达式调用 operator new，所以 new 表达式将负责构造对象。

5. 定义 operator delete

operator delete 成员只负责管理内存，在析构函数中已经清除了对象本身，delete 表达式在调用 operator delete 之前调用析构函数。operator delete 成员很简单：

```
template <class T>
void CachedObj<T>::operator delete(void *p, size_t)
{
    if (p != 0)
        // put the "deleted" object back at head of freelist
        add_to_freelist(static_cast<T*>(p));
}
```

770

它调用 add_to_freelist 成员将被删除对象放回自由列表。

令人感兴趣的部分是强制类型转换（5.12.4 节）。在删除动态分配的类类型对象时调用 operator delete，编译器将该对象的地址传给 operator delete。但是，指针的形参类型必须是 void*，在调用 add_to_freelist 之前，必须将指针从 void* 强制转换为它的实际类型，本例中，那个类型是指向 T 的指针，它是 CachedObj 派生类型的对象的指针。

6. add_to_freelist 成员

这个成员的任务是设置 next 指针，并且在将对象加到自由列表时更新 freeStore 指针：

```
// puts object at head of the freelist
template <class T>
void CachedObj<T>::add_to_freelist(T *p)
{
    p->CachedObj<T>::next = freeStore;
    freeStore = p;
}
```

唯一复杂的部分是 next 成员的使用。回忆一下，我们打算将 CachedObj 作为基类使用，被分配对象不是 CachedObj 类型的，相反，那些对象是 CachedObj 的派生类型的，因此，类型 T 将是派生类型，指针 p 是指向 T 的指针，不是指向 CachedObj 的指针。如果派生类型有自己的名为 next 的成员，则编写 p->next 将获得派生类的成员！但我們希望在基类——CachedObj 类中设置 next。



为了避免任何与派生类中定义的成员可能的冲突，显式指定我们正在给基类成员 next 赋值。

7. 定义静态数据成员

剩下的是定义静态数据成员：

```
template <class T> allocator< T > CachedObj< T >::alloc_mem;
template <class T> T *CachedObj< T >::freeStore = 0;
template <class T> const size_t CachedObj< T >::chunk = 24;
```

照常，对于类模板的静态成员，每个类型使用不同的静态成员来实例化 CachedObj 类。将 chunk

初始化为任意值，本例中为 24。将 freeStore 指针初始化为 0，指出自由列表开始时为空。
 alloc_mem 成员不需要初始化，但必须记得定义它。

习题

习题 18.10 解释下面每个初始化，指出是否有错误的，如果有，为什么错。

```
class iStack {
public:
    iStack(int capacity): stack(capacity), top{0} { }
private:
    int top;
    vector<int> stack;
};
(a) iStack *ps = new iStack(20);
(b) iStack *ps2 = new const iStack(15);
(c) iStack *ps3 = new iStack[ 100 ];
```

习题 18.11 解释下面 new 和 delete 表达式中发生什么。

```
struct Exercise {
    Exercise();
    ~Exercise();
};
Exercise *pe = new Exercise[20];
delete[] pe;
```

习题 18.12 为 Queue 类或你选择的其他类实现一个类特定的内存分配器。测量性能的改变，看看到底有多大帮助。

18.2 运行时类型识别

通过运行时类型识别 (RTTI)，程序能够使用基类的指针或引用来检索这些指针或引用所指对象的实际派生类型。

通过下面两个操作符提供 RTTI:

- (1) typeid 操作符，返回指针或引用所指对象的实际类型。
- (2) dynamic_cast 操作符，将基类类型的指针或引用安全地转换为派生类型的指针或引用。



这些操作符只为带有一个或多个虚函数的类返回动态类型信息，对于其他类型，返回静态（即编译时）类型的信息。

对于带虚函数的类，在运行时执行 RTTI 操作符，但对于其他类型，在编译时计算 RTTI 操作符。

772

当具有基类的引用或指针，但需要执行不是基类组成部分的派生类操作的时候，需要动态的强制类型转换。通常，从基类指针获得派生类行为最好的方法是通过虚函数。当使用虚函数的时候，编译器自动根据对象的实际类型选择正确的函数。

但是，在某些情况下，不可能使用虚函数。在这些情况下，RTTI 提供了可选的机制。然而，这种机制比使用虚函数更容易出错：程序员必须知道应该将对象强制转换为哪种类型，并且必须

检查转换是否成功执行了。



使用动态强制类型转换要小心。只要有可能，定义和使用虚函数比直接接管类型管理好得多。

18.2.1 `dynamic_cast` 操作符

可以使用 `dynamic_cast` 操作符将基类类型对象的引用或指针转换为同一继承层次中其他类型的引用或指针。与 `dynamic_cast` 一起使用的指针必须是有效的——它必须为 0 或者指向一个对象。

与其他强制类型转换不同，`dynamic_cast` 涉及运行时类型检查。如果绑定到引用或指针的对象不是目标类型的对象，则 `dynamic_cast` 失败。如果转换到指针类型的 `dynamic_cast` 失败，则 `dynamic_cast` 的结果是 0 值；如果转换到引用类型的 `dynamic_cast` 失败，则抛出一个 `bad_cast` 类型的异常。

因此，`dynamic_cast` 操作符一次执行两个操作。它首先验证被请求的转换是否有效，只有转换有效，操作符才实际进行转换。一般而言，引用或指针所绑定的对象的类型在编译时是未知的，基类的指针可以赋值为指向派生类对象，同样，基类的引用也可以用派生类对象初始化，因此，`dynamic_cast` 操作符执行的验证必须在运行时进行。

1. 使用 `dynamic_cast` 操作符

作为例子，假定 `Base` 是至少带一个虚函数的类，并且 `Derived` 类派生于 `Base` 类。如果有一个名为 `basePtr` 的指向 `Base` 的指针，就可以像这样在运行时将它强制转换为指向 `Derived` 的指针：

```
if (Derived *derivedPtr = dynamic_cast<Derived*>(basePtr))
{
    // use the Derived object to which derivedPtr points
} else { // BasePtr points at a Base object
    // use the Base object to which basePtr points
}
```

在运行时，如果 `basePtr` 实际指向 `Derived` 对象，则转换将成功，并且 `derivedPtr` 将被初始化为指向 `basePtr` 所指的 `Derived` 对象；否则，转换的结果是 0，意味着将 `derivedPtr` 置为 0，并且 `if` 中的条件失败。



可以对值为 0 的指针应用 `dynamic_cast`，这样做的结果是 0。

通过检查 `derivedPtr` 的值，`if` 内部的代码知道它是在操作 `Derived` 对象，该代码使用 `Derived` 的操作是安全的。如果 `dynamic_cast` 因 `basePtr` 引用了 `Base` 对象而失败，则 `else` 子句进行适应于 `Base` 的处理来代替。在 `if` 条件内部进行检查的另一好处是，不可能在 `dynamic_cast` 和测试转换结果之间插入代码，因此，不可能在测试转换是否成功之前不经意地使用 `derivedPtr`。第三个好处是，在 `if` 外部不能访问该指针，如果转换失败，则在后面的忘了测试的地方，未绑定的指针是不可用的。



在条件中执行 `dynamic_cast` 保证了转换和其结果测试在一个表达式中进行。

2. 使用 `dynamic_cast` 和引用类型

在前面例子中，使用了 `dynamic_cast` 将基类指针转换为派生类指针，也可以使用 `dynamic_cast` 将基类引用转换为派生类引用，这种 `dynamic_cast` 操作的形式如下：

```
dynamic_cast< Type& >(val)
```

这里，`Type` 是转换的目标类型，而 `val` 是基类类型的对象。

只有当 `val` 实际引用一个 `Type` 类型对象，或者 `val` 是一个 `Type` 派生类型的对象的时候，`dynamic_cast` 操作才将操作数 `val` 转换为想要的 `Type&` 类型。

因为不存在空引用，所以不可能对引用使用用于指针强制类型转换的检查策略，相反，当转换失败的时候，它抛出一个 `std::bad_cast` 异常，该异常在库头文件 `typeinfo` 中定义。

可以重写前面的例子如下，以便使用引用：

```
void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // use the Derived object to which b referred
    } catch (bad_cast) {
        // handle the fact that the cast failed
    }
}
```

774

习题

习题 18.13 给定下面的类层次，其中每个类都定义了 `public` 默认构造函数和虚析构函数，

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };
```

如果有，下面哪些 `dynamic_cast` 失败？

习题 18.14 如果 `D` 和 `B` 都以 `A` 为虚基类，上题最后一个转换中将发生什么？

```
(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);
```

习题 18.15 使用上面习题中定义类层次，重写下面代码片段，以便执行 `dynamic_cast` 将表达式 `*pa` 转换为 `C&` 类型：

```
if (C *pc = dynamic_cast< C* >(pa))
    // use C's members
} else {
    // use A's members
}
```

习题 18.16 解释什么时候可以使用 `dynamic_cast` 代替虚函数。

18.2.2 typeid 操作符

为 RTTI 提供的第二个操作符是 **typeid** 操作符。typeid 操作符使程序能够问一个表达式：你是什么类型？

typeid 表达式形如：

```
typeid(e)
```

这里 e 是任意表达式或者是类型名。

如果表达式的类型是类类型且该类包含一个或多个虚函数，则表达式的动态类型可能不同于它的静态编译时类型。例如，如果表达式对基类指针解引用，则该表达式的静态编译时类型是基类类型；但是，如果指针实际指向派生类对象，则 typeid 操作符将说表达式的类型是派生类型。

typeid 操作符可以与任何类型的表达式一起使用。内置类型的表达式以及常量都可以用作 typeid 操作符的操作数。如果操作数不是类类型或者是没有虚函数的类，则 typeid 操作符指出操作数的静态类型；如果操作数是定义了至少一个虚函数的类类型，则在运行时计算类型。

775

typeid 操作符的结果是名为 type_info 的标准库类型的对象引用，18.2.4 节将更详细地讨论这个类型。要使用 type_info 类，必须包含库头文件 typeinfo。

使用 typeid 操作符

typeid 最常见的用途是比较两个表达式的类型，或者将表达式的类型与特定类型相比较：

```
Base *bp;
Derived *dp;
// compare type at run time of two objects
if (typeid(*bp) == typeid(*dp)) {
    // bp and dp point to objects of the same type
}
// test whether run time type is a specific type
if (typeid(*bp) == typeid(Derived)) {
    // bp actually points to a Derived
}
```

第一个 if 中，比较 bp 所指对象与 dp 所指对象的实际类型，如果它们指向同一类型，则测试成功。类似地，如果 bp 当前指向 Derived 对象，则第二个 if 成功。

注意，typeid 的操作数是表示对象的表达式——测试 *bp，而不是 bp：

```
// test always fails: The type of bp is pointer to Base
if (typeid(bp) == typeid(Derived)) {
    // code never executed
}
```

这个测试将 Base* 类型与 Derived 类型相比较，这两个类型不相等，所以，无论 bp 所指对象的类型是什么，这个测试将总是失败。



只有当 typeid 的操作数是带虚函数的类类型的对象的时候，才返回动态类型信息。测试指针（相对于指针指向的对象）返回指针的静态的、编译时类型。

如果指针 `p` 的值是 0，那么，如果 `p` 的类型是带虚函数的类型，则 `typeid(*p)` 抛出一个 `bad_typeid` 异常；如果 `p` 的类型没有定义任何虚函数，则结果与 `p` 的值是不相关的。正像计算表达式 `sizeof` (5.8 节) 一样，编译器不计算 `*p`，它使用 `p` 的静态类型，这并不要求 `p` 本身是有效指针。

776

习题

- 习题 18.17** 编写一个表达式，动态地将 `Query_base` 对象的指针强制转换为 `AndQuery` 对象的指针。通过使用 `AndQuery` 和其他查询类型的对象测试该转换。显示一个语句指出强制转换是否工作，并确信输出与你的表达式匹配。
- 习题 18.18** 编写相同的强制转换，但将 `Query_base` 对象转换为 `AndQuery` 的引用。重复测试以确信你的转换正确工作。
- 习题 18.19** 编写一个 `typeid` 表达式，看两个 `Query_base` 指针是否指向相同的类型，然后检查该类型是否为 `AndQuery`。

18.2.3 RTTI 的使用

作为说明何时可以使用 RTTI 的例子，考虑一个类层次，我们希望为它实现相等操作符。如果两个对象的给定数据成员集合的值相同，它们就相等。每个派生类型可以增加自己的数据，我们希望在测试相等的时候包含这些数据。

因为确定派生类型的相等与确定基类类型的相等所考虑的值不同，所以对层次中的每一对类型（潜在地）需要一个不同的相等操作符。而且，希望能够使用给定类型作为左操作数或右操作数，所以实际上对每一对类型将需要两个操作符。

如果类层次中只有两个类型，就需要四个函数：

```
bool operator==(const Base&, const Base&)
bool operator==(const Derived&, const Derived&)
bool operator==(const Derived&, const Base&);
bool operator==(const Base&, const Derived&);
```

但是，如果类层次中有几个类型，必须定义的操作符的数目就迅速扩大——仅仅 3 个类型就需要 9 个操作符。如果类层次有 4 个类型，将需要 16 个操作符，以此类推。

也许我们认为可以通过定义一个虚函数集合来解决这个问题，这些虚函数可以在类层次中每一层执行相等测试。给定这些虚函数，可以定义单个相等操作符，操作基类类型的引用，该操作符可以将工作委派给可以完成实际工作的虚操作。

但是，虚函数并不是解决这个问题的好办法。麻烦在于决定 `equal` 操作的形参的类型。虚函数在基类类型和派生类型中必须有相同的形参类型，这意味着，虚 `equal` 操作必须有一个形参是基类的引用。

但是，当比较两个派生类对象的时候，我们希望比较可能特定于派生类的数据成员。如果形参是基类的引用，就只能比较基类中出现的成员，我们不能访问在派生类中但不在基类中出现的

成员。

777

仔细考虑这个问题，我们看到，希望在试图比较不同类型的对象时返回假（false）。有了这个观察，现在可以使用 RTTI 解决我们的问题。

我们将定义单个相等操作符。每个类定义一个虚函数 `equal`，该函数首先将操作数强制转换为正确的类型。如果转换成功，就进行真正的比较；如果转换失败，`equal` 操作就返回 `false`。

1. 类层次

为了使概念更清楚一点，假定类层次是这样的：

```
class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // interface members for Base
protected:
    virtual bool equal(const Base&) const;
    // data and other implementation members of Base
};
class Derived: public Base {
    friend bool operator==(const Base&, const Base&);
public:
    // other interface members for Derived
private:
    bool equal(const Base&) const;
    // data and other implementation members of Derived
};
```

2. 类型敏感的相等操作符

下面看看可以怎样定义整体的相等操作符：

```
bool operator==(const Base &lhs, const Base &rhs)
{
    // returns false if typeid's are different otherwise
    // returns lhs.equal(rhs)
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}
```

如果操作数类型不同，这个操作符就返回假；如果操作数类型相同，它就将实际比较操作数的工作委派给适当的虚函数 `equal`。如果操作数是 `Base` 对象，就调用 `Base::equal`；如果操作数是 `Derived` 对象，就调用 `Derived::equal`。

3. 虚函数 `equal`

层次中的每个类都必须定义自己的 `equal` 版本。派生类中的 `equal` 函数将以相同的方式开始：它们将实参强制转换为类本身的类型。

778

```
bool Derived::equal(const Base &rhs) const
{
    if (const Derived *dp
        = dynamic_cast<const Derived*>(&rhs)) {
        // do work to compare two Derived objects and return result
    } else
        return false;
}
```

这个强制转换应该总是成功——毕竟，只有在测试了两个操作数类型相同之后，才从相等操作符调用该函数。但是，这个强制转换是必要的，以便函数可以访问右操作数的派生类成员。因为操作数是 `Base&`，所以如果想要访问 `Derived` 的成员，就必须首先进行强制转换。

4. 基类 `equal` 函数

这个操作比其他的简单一点：

```
bool Base::equal(const Base &rhs) const
{
    // do whatever is required to compare to Base objects
}
```

使用形参之前不必强制转换，`*this` 和形参都是 `Base` 对象，所以对形参类型也定义了该对象可用的所有操作。

18.2.4 `type_info` 类

`type_info` 类的确切定义随编译器而变化，但是，标准保证所有的实现将至少提供表 18-2 列出的操作。

表 18-2 `type_info` 的操作

<code>t1 == t2</code>	如果两个对象 <code>t1</code> 和 <code>t2</code> 类型相同，就返回 <code>true</code> ；否则，返回 <code>false</code>
<code>t1 != t2</code>	如果两个对象 <code>t1</code> 和 <code>t2</code> 类型不同，就返回 <code>true</code> ；否则，返回 <code>false</code>
<code>t.name()</code>	返回 C 风格字符串，这是类型名字的可显示版本。类型名字用系统相关的方法产生
<code>t1.before(t2)</code>	返回指出 <code>t1</code> 是否出现在 <code>t2</code> 之前的 <code>bool</code> 值。 <code>before</code> 强制的次序与编译器有关

779

因为打算作基类使用，`type_info` 类也提供公用虚析构函数。如果编译器想要提供附加的类型信息，应该在 `type_info` 的派生类中进行。

默认构造函数和复制构造函数以及赋值操作符都定义为 `private`，所以不能定义或复制 `type_info` 类型的对象。程序中创建 `type_info` 对象的唯一方法是使用 `typeid` 操作符。

`name` 函数为 `type_info` 对象所表示的类型的名字返回 C 风格字符串。给定类型所用的值取决于编译器，具体来说，无须与程序中使用的类型名字匹配。对 `name` 返回值的唯一保证是，它为每个类型返回唯一的字符串。虽然如此，仍可以使用 `name` 成员来显示 `type_info` 对象的名字：

```
int iobj;
cout << typeid(iobj).name() << endl
    << typeid(8.16).name() << endl
    << typeid(std::string).name() << endl
    << typeid(Base).name() << endl
    << typeid(Derived).name() << endl;
```

`name` 返回的格式和值随编译器而变化。在我们的机器上执行时，这个程序产生下面的输出：

```
i
d
Ss
4Base
7Derived
```



`type_info` 类随编译器而变。一些编译器提供附加的成员函数，那些函数提供关于程序中所用类型的附加信息。你应该查阅编译器的参考手册来理解所提供的确切的 `type_info` 支持。

习题

习题 18.20 给定下面的类层次，其中每个类都定义了 `public` 默认构造函数及虚析构函数，下面语句显示哪些类型名？

```
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;
```

18.3 类成员的指针

我们知道，给定对象的指针，可以使用箭头 (`->`) 操作符从该对象获得给定成员。有时从成员开始是有用的，也就是说，我们也许希望获得特定成员的指针，然后从一个对象或别的对象获得该成员。

可以通过使用称为**成员指针** (pointer to member) 的特殊种类的指针做到这一点。成员指针包含类的类型以及成员的类型。这一事实影响着怎样定义成员指针，怎样将成员指针绑定到函数或数据成员，以及怎样使用它们。

成员指针只应用于类的非 `static` 成员。`static` 类成员不是任何对象的组成部分，所以不需要特殊语法来指向 `static` 成员，`static` 成员指针是普通指针。

780

18.3.1 声明成员指针

在研究成员指针时，将使用第 12 章的 `Screen` 类的简化版本。

```
class Screen {
public:
    typedef std::string::size_type index;
    char get() const;
    char get(index ht, index wd) const;
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

1. 定义数据成员的指针

`Screen` 类的 `contents` 成员的类型为 `std::string`。`contents` 的完全类型是“`Screen` 类

的成员，其类型是 `std::string`”。因此，可以指向 `contents` 的指针的完全类型是“指向 `std::string` 类型的 `Screen` 类成员的指针”，这个类型可写为

```
string Screen::*
```

可以将指向 `Screen` 类的 `string` 成员的指针定义为

```
string Screen::*ps_Screen;
```

可以用 `contents` 的地址初始化 `ps_Screen`，代码为

```
string Screen::*ps_Screen = &Screen::contents;
```

还可以将指向 `height`、`width` 或 `cursor` 成员的指针定义为

```
Screen::index Screen::*pindex;
```

这是说，`pindex` 是具有 `Screen::index` 类型的 `Screen` 类成员的指针。可以将 `width` 的地址赋给这个指针：

```
pindex = &Screen::width;
```

可以用指针 `pindex` 指向 `width`、`height` 或 `cursor` 中任意一个，因为这三个都是 `index` 类型的 `Screen` 类成员。

2. 定义成员函数的指针

成员函数的指针必须在三个方面与它所指函数的类型相匹配：

- (1) 函数形参的类型和数目，包括成员是否为 `const`。
- (2) 返回类型。
- (3) 所属类的类型。

通过指定函数返回类型、形参表和类来定义成员函数的指针。例如，可引用不接受形参的 `get` 版本的 `Screen` 成员函数的指针具有如下类型：

```
char (Screen::*)() const
```

这个类型指定 `Screen` 类的 `const` 成员函数的指针，不接受形参并返回 `char` 类型的值。这个 `get` 函数版本的指针可以像下面这样定义和初始化：

```
// pmf points to the Screen get member that takes no arguments
char (Screen::*pmf)() const = &Screen::get;
```

也可以将带两个形参的 `get` 函数版本的指针定义为

```
char (Screen::*pmf2)(Screen::index, Screen::index) const;
pmf2 = &Screen::get;
```

调用操作符的优先级高于成员指针操作符，因此，包围 `Screen::*` 的括号是必要的，没有这个括号，编译器就将下面代码当作（无效的）函数声明：

```
// error: non-member function p cannot have const qualifier
char Screen::*p() const;
```

3. 为成员指针使用类型别名

类型别名（`typedef`）可以使成员指针更容易阅读。例如，下面的类型别名将 `Action` 定义为



带两个形参的 get 函数版本的类型的另一名字:

```
// Action is a type name
typedef
char (Screen::*Action)(Screen::index, Screen::index) const;
```

Action 是类型“Screen 类的接受两个 index 类型形参并返回 char 的成员函数的指针”的名字。使用类型别名, 可以将 get 指针的定义简化为

```
Action get = &Screen::get;
```

可以使用成员指针函数类型来声明函数形参和函数返回类型:

```
// action takes a reference to a Screen and a pointer to Screen member function
Screen& action(Screen&, Action = &Screen::get);
```

这个函数声明为接受两个形参: Screen 对象的引用, 以及 Screen 类的接受两个 index 类型形参并返回 char 的成员函数的指针。可以通过传递 Screen 类中适当成员函数的指针或地址调用 action 函数:

```
Screen myScreen;
// equivalent calls:
action(myScreen);           // uses the default argument
action(myScreen, get);      // uses the variable get that we previously defined
action(myScreen, &Screen::get); // pass address explicitly
```

习题

习题 18.21 普通数据指针或函数指针与数据成员指针或函数成员指针之间的区别是什么?

习题 18.22 定义可以表示 Sales_item 类的 isbn 成员的指针的类型。

习题 18.23 定义可以指向 same_isbn 成员的指针。

习题 18.24 编写类型别名, 作为可指向 Sales_item 的 avg_price 成员的指针的同义词。

18.3.2 使用类成员的指针

类似于成员访问操作符.和->, .*和->*是两个新的操作符, 它们使我们能够将成员指针绑定到实际对象。这两个操作符的左操作数必须是类类型的对象或类类型的指针, 右操作数是该类型的成员指针。

783

- 成员指针解引用操作符 (.*) 从对象或引用获取成员。
- 成员指针箭头操作符 (->*) 通过对象的指针获取成员。

1. 使用成员函数的指针

使用成员指针, 可以这样调用不带形参的 get 函数版本:

```
// pmf points to the Screen get member that takes no arguments
char (Screen::*pmf)() const = &Screen::get;
Screen myScreen;
char c1 = myScreen.get();           // call get on myScreen
char c2 = (myScreen.*pmf)();        // equivalent call to get
Screen *pScreen = &myScreen;
c1 = pScreen->get();                 // call get on object to which pScreen points
c2 = (pScreen->*pmf)();               // equivalent call to get
```



因为调用操作符 `()` 比成员指针操作符优先级高，所以调用 `(myScreen.*pmf)()` 和 `(pScreen->*pmf)()` 需要括号。

没有括号，就会将

```
myScreen.*pmf()
```

解释为

```
myScreen.*(pmf())
```

这个代码是说，调用名为 `pmf` 的函数，并将它的返回值绑定到成员对象操作符 `(.*)` 的指针。当然，`pmf` 的类型不支持这样的使用，且会产生编译时错误。

像任何其他函数一样，也可以在通过成员函数指针进行的调用中传递实参：

```
char (Screen::*pmf2)(Screen::index, Screen::index) const;
pmf2 = &Screen::get;
Screen myScreen;
char c1 = myScreen.get(0,0);           // call two-parameter version of get
char c2 = (myScreen.*pmf2)(0,0);       // equivalent call to get
```

2. 使用数据成员的指针

相同的成员指针操作符用于访问数据成员：

```
Screen::index Screen::*pindex = &Screen::width;
Screen myScreen;
// equivalent ways to fetch width member of myScreen
Screen::index ind1 = myScreen.width;           // directly
Screen::index ind2 = myScreen.*pindex;         // dereference to get width
Screen *pScreen;
// equivalent ways to fetch width member of *pScreen
ind1 = pScreen->width;                         // directly
ind2 = pScreen->*pindex;                       // dereference pindex to get width
```

3. 成员指针函数表

函数指针和成员函数指针的一个公共用途是，将它们存储在函数表中。函数表是函数指针的集合，在运行时从中选择给定调用。

对具有几个相同类型成员类而言，可以使用这样的表来从这些成员的集合中选择一个。假定扩展 `Screen` 类以包含几个成员函数，其中每一个在特定方向移动光标：

```
class Screen {
public:
    // other interface and implementation members as before
    Screen& home();           // cursor movement functions
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
```

这些新函数的每一个都不接受形参，并返回调用它的对象的引用。

4. 使用函数指针表

我们可能希望定义一个 `move` 函数，它可以调用这些函数中的任意一个并执行指定动作。为

了支持这个新函数，我们将在 Screen 中增加一个 static 成员，该成员是光标移动函数的指针的数组：

```
class Screen {
public:
    // other interface and implementation members as before
    // Action is pointer that can be assigned any of the cursor movement members
    typedef Screen& (Screen::*Action)();
    static Action Menu[];          // function table
public:
    // specify which direction to move
    enum Directions { HOME, FORWARD, BACK, UP, DOWN };
    Screen& move(Directions);
};
```

785

名为 Menu 的数组将保存指向每个光标移动函数的指针，将在对应于 Directions 中枚举成员的偏移位置保存那些函数，move 函数接受枚举成员并调用适当函数：

```
Screen& Screen::move(Directions cm)
{
    // fetch the element in Menu indexed by cm
    // run that member on behalf of this object
    (this->*Menu[cm])();
    return *this;
}
```

这样计算 move 内部的调用：获取由 cm 索引的 Menu 元素（该元素是 Screen 类成员函数的指针），代表 this 指向的对象调用该元素指向的成员函数。

调用 move 时，传给它一个枚举成员，指出向哪个方向移动光标：

```
Screen myScreen;
myScreen.move(Screen::HOME);    // invokes myScreen.home
myScreen.move(Screen::DOWN);    // invokes myScreen.down
```

5. 定义成员函数指针表

剩下的是定义和初始化表本身：

```
Screen::Action Screen::Menu[] = { &Screen::home,
                                   &Screen::forward,
                                   &Screen::back,
                                   &Screen::up,
                                   &Screen::down,
                                   };
```

习题

习题 18.25 Screen 类的成员 screen 和 cursor 的类型是什么？

习题 18.26 定义一个可以指向 Screen 类 cursor 成员的成员指针，通过该指针获取 Screen::cursor 的值。

习题 18.27 为 Screen 类成员函数的每个可区分类型定义类型别名。

习题 18.28 也可以将成员指针声明为类的数据成员。修改 Screen 类的定义，以便包含与 home 和 end 类型相同的 Screen 成员函数的指针。

习题 18.29 编写一个 `Screen` 构造函数，该构造函数接受指向 `Screen` 成员函数的指针类型形参，成员函数的形参表和返回类型与成员函数 `home` 和 `end` 的相同。

习题 18.30 为该形参提供默认实参，使用该形参对上题中引入的数据成员进行初始化。

习题 18.31 提供一个 `Screen` 成员函数来设置这个成员。

18.4 嵌套类

可以在另一个类内部定义一个类，这样的类是**嵌套类** (nested class)，也称为**嵌套类型** (nested type)。嵌套类最常用于定义执行类，如第 16 章的 `QueueItem` 类。

嵌套类是独立的类，基本上与它们的外围类不相关，因此，外围类和嵌套类的对象是互相独立的。嵌套类型的对象不具备外围类所定义的成员，同样，外围类的成员也不具备嵌套类所定义的成员。

786

嵌套类的名字在其外围类的作用域中可见，但在其他类作用域或定义外围类的作用域中不可见。嵌套类的名字将不会与另一作用域中声明的名字冲突。

嵌套类可以具有与非嵌套类相同种类的成员。像任何其他类一样，嵌套类使用访问标号控制对自己成员的访问。成员可以声明为 `public`、`private` 或 `protected`。外围类对嵌套类的成员没有特殊访问权，并且嵌套类对其外围类的成员也没有特殊访问权。

嵌套类定义了其外围类中的一个类型成员。像任何其他成员一样，外围类决定对这个类型的访问。在外围类的 `public` 部分定义的嵌套类定义了可在任何地方使用的类型，在外围类的 `protected` 部分定义的嵌套类定义了只能由外围类、友元或派生类访问的类型，在外围类的 `private` 部分定义的嵌套类定义了只能被外围类或其友元访问的类型。

18.4.1 嵌套类的实现

在第 16 章中实现的 `Queue` 类定义了一个名为 `QueueItem` 的伙伴执行类，`QueueItem` 类是私有类(它只有 `private` 成员)但它是在全局作用域中定义的。普通用户代码不能使用 `QueueItem` 类的对象：它的所有成员，包括构造函数，均为 `private`。但是，名字 `QueueItem` 是全局可见的，不能定义名为 `QueueItem` 的自有类型或其他实体。

787

一个更好的设计可能是，将 `QueueItem` 类设为 `Queue` 类的 `private` 成员，那样，`Queue` 类(及其友元)可以使用 `QueueItem`，但 `QueueItem` 类类型对普通用户代码不可见。一旦 `QueueItem` 类本身为 `private`，我们就可以使其成员为 `public` 成员——只有 `Queue` 或 `Queue` 的友元可以访问 `QueueItem` 类型，所以不必防止一般程序访问 `QueueItem` 的成员。通过用保留字 `struct` 定义 `QueueItem` 使成员为 `public` 成员。

新的设计如下：

```
template <class Type> class Queue {
    // interface functions to Queue are unchanged
private:
    // public members are ok: QueueItem is a private member of Queue
    // only Queue and its friends may access the members of QueueItem
```

```

struct QueueItem {
    QueueItem(const Type &);
    Type item;           // value stored in this element
    QueueItem *next;     // pointer to next element in the Queue
};
QueueItem *head;        // pointer to first element in Queue
QueueItem *tail;        // pointer to last element in Queue
};

```

因为 QueueItem 类是 private 成员, 所以只有 Queue 类的成员和友元可以使用 QueueItem 类型。使 QueueItem 类成为 private 成员之后, 就可以使 QueueItem 成员为 public, 这样做使我们能够删去 QueueItem 中的友元声明。

1. 嵌套在类模板内部的类是模板

因为 Queue 类是一个模板, 它的成员也隐含地是模板。具体而言, 嵌套类 QueueItem 隐含地是一个类模板。像 Queue 类中任何其他成员一样, QueueItem 的模板形参与其外围类 (Queue 类) 的模板形参相同。

Queue 类的每次实例化用对应于 Type 的适当模板实参产生自己的 QueueItem 类。QueueItem 类模板的实例化与外围 Queue 类模板的实例化之间的映射是一一对应的。

2. 定义嵌套类的成员

这个 QueueItem 类版本中, 我们选择不在类内部定义 QueueItem 构造函数, 相反, 我们单独定义它。唯一复杂的是在哪里定义以及怎样命名。



在其类外部定义的嵌套类成员, 必须定义在定义外围类的同一作用域中。在其类外部定义的嵌套类的成员, 不能定义在外围类内部, 嵌套类的成员不是外围类的成员。

QueueItem 类的构造函数不是 Queue 类的成员, 因此, 不能将它定义在 Queue 类定义体中的任何地方, 它必须与 Queue 类在同一作用域但在 Queue 类的外部定义。为了将成员定义在嵌套类定义体外部, 必须记住, 成员的名字在类外部是不可见的。要定义这个构造函数, 必须指出, QueueItem 是 Queue 类作用域中的嵌套类, 通过用外围 Queue 类的名字限定类名 QueueItem 来做到这一点:

```

// defines the QueueItem constructor
// for class QueueItem nested inside class Queue<Type>
template <class Type>
Queue<Type>::QueueItem::QueueItem(const Type &t):
    item(t), next(0) { }

```

当然, Queue 和 QueueItem 都是类模板, 因此, 这个构造函数也是模板。

这段代码定义了一个函数模板, 以名为 Type 的单个类型形参化为形参。从右至左读函数的名字, 这个函数是 QueueItem 类的构造函数, 它嵌套在 Queue<Type> 类的作用域中。

3. 在外围类外部定义嵌套类

嵌套类通常支持外围类的实现细节。我们可能希望防止外围类的用户看见嵌套类的实现代码。

例如, 我们可能希望将 QueueItem 类的定义放在它自己的文件中, 我们可以在 Queue 类及其成员的实现文件中包含这个文件。正如可以在类定义体外部定义嵌套类的成员一样, 我们也可以在类定义体的外部定义整个嵌套类:

```

template <class Type> class Queue {
    // interface functions to Queue are unchanged
private:
    struct QueueItem; // forward declaration of nested type QueueItem
    QueueItem *head; // pointer to first element in Queue
    QueueItem *tail; // pointer to last element in Queue
};

template <class Type>
struct Queue<Type>::QueueItem {
    QueueItem(const Type &t): item(t), next(0) { }
    Type item; // value stored in this element
    QueueItem *next; // pointer to next element in the Queue
};

```

为了在外围类的外部定义类体，必须用外围类的名字限定嵌套类的名字。注意，我们仍然必须在 Queue 类的定义体中声明 QueueItem 类。

也可以在外围类的定义体中声明然后定义嵌套类。像其他前向声明一样，嵌套类的前向声明使嵌套类能够具有相互引用的成员。

789



在看到在类定义体外部定义的嵌套类的实际定义之前，该类是不完全类型（12.1.4 节），应用所有使用不完全类型的常规限制。

4. 嵌套类静态成员定义

如果 QueueItem 类声明了一个静态成员，它的定义也需要放在外层作用域中。假定 QueueItem 类有一个静态成员，它的定义看起来可能像下面这样：

```

// defines an int static member of QueueItem,
// which is a type nested inside Queue<Type>
template <class Type>
int Queue<Type>::QueueItem::static_mem = 1024;

```

5. 使用外围类的成员

注解

外围作用域的对象与其嵌套类型的对象之间没有联系。

嵌套类中的非静态函数具有隐含的 this 指针，指向嵌套类型的对象。嵌套类型对象只包含嵌套类型的成员，不能使用 this 指针获取外围类的成员。同样，外围类中的非静态成员函数也具有 this 指针，它指向外围类型的对象，该对象只具有外围类中定义的成员。

外围类的非静态数据或函数成员的任何使用都要求通过外围类的指针、引用或对象进行。Queue 类中的 pop 函数不能直接使用 item 或 next：

```

template <class Type>
void Queue<Type>::pop()
{
    // pop is unchecked: popping off an empty Queue is undefined
    QueueItem* p = head; // keep pointer to head so can delete it
    head = head->next; // head now points to next element
    delete p; // delete old head element
}

```

Queue 类型的对象没有名为 item 或 next 的成员。Queue 类的函数成员可以使用 head 和 tail 成员（它们是指向 QueueItem 对象的指针）来获取那些 QueueItem 成员。

6. 使用静态成员或其他类型的成员

嵌套类可以直接引用外围类的静态成员、类型名和枚举成员（2.7 节），当然，引用外围类作用域之外的类型名或静态成员，需要作用域确定操作符。

7. 嵌套模板的实例化

实例化外围类模板的时候，不会自动实例化类模板的嵌套类。像任何成员函数一样，只有当在需要完整类类型的情况下使用嵌套类本身的时候，才会实例化嵌套类。例如，像

```
Queue<int> qi; // instantiates Queue<int> but not QueueItem<int>
```

这样的定义，用 int 类型实例化了 Queue 模板，但没有实例化 QueueItem<int> 类型。成员 head 和 tail 是指向 QueueItem<int> 的指针，这里不需要实例化 QueueItem<int> 来定义那个类的指针。

使 QueueItem 类成为类模板 Queue 的嵌套类并不改变 QueueItem 的实例化。只有在使用 QueueItem<int> 的时候——本例中，只有当 Queue<int> 类的成员函数中对 head 和 tail 解引用的时候，才实例化 QueueItem<int> 类。

18.4.2 嵌套类作用域中的名字查找

对嵌套类中所用名字的名字查找（12.3.1 节）在普通类的名字查找之前进行，现在唯一的区别是可能要查找一个或多个外围类作用域。



当处理类成员声明的时候，所用的任意名字必须在使用之前出现。当处理定义的时候，整个嵌套类和外围类均在作用域中。

作为嵌套类中名字查找的例子，考虑下面的类声明：

```
class Outer {
public:
    struct Inner {
        // ok: reference to incomplete class
        void process(const Outer&);
        Inner2 val; // error: Outer::Inner2 not in scope
    };
    class Inner2 {
public:
        // ok: Inner2::val used in definition
        Inner2(int i = 0): val(i) { }
        // ok: definition of process compiled after enclosing class is complete
        void process(const Outer &out) { out.handle(); }
private:
        int val;
    };
    void handle() const; // member of class Outer
};
```

编译器首先处理 Outer 类成员的声明 Outer::Inner 和 Outer::Inner2。

790

791

791

将名字 `Outer` 作为 `Inner::process` 形参的使用被绑定到外围类，在看到 `process` 的声明时，那个类仍是不完整的，但形参是一个引用，所以这个使用是正确的。

数据成员 `Inner::val` 的声明是错误的，还没有看到 `Inner2` 类型。

`Inner2` 中的声明看来没有问题——它们大多只使用内置类型 `int`。唯一的例外是成员函数 `process`，它的形参确定为不完全类型 `Outer`。因为其形参是一个引用，所以 `Outer` 为不完全类型是无关紧要的。

直到看到了外围类中的其余声明之后，编译器才处理构造函数和 `process` 成员的定义。对 `Outer` 类声明的完成将函数 `handle` 的声明放在作用域中。

当编译器查找 `Inner2` 类中的定义所用的名字时，`Inner2` 类和 `Outer` 类中的所有名字都在作用域中。`val` 的使用（出现在 `val` 的声明之前）是正确的：将该引用绑定到 `Inner2` 类中的数据成员。同样，`Inner2::process` 成员函数体中对 `Outer` 类的 `handle` 的使用也正确，当编译 `Inner2` 类的成员的时候，整个 `Outer` 类在作用域中。

使用作用域操作符控制名字查找

可以使用作用域操作符访问 `handle` 的全局版本：

```
class Inner2 {
public:
    // ...
    // ok: programmer explicitly specifies which handle to call
    void process(const Outer &out) { ::handle(out); }
};
```

习题

习题 18.32 重新实现第 16 章的 `Queue` 类和 `QueueItem` 类，使 `QueueItem` 成为 `Queue` 内部的嵌套类。

习题 18.33 解释 `Queue` 设计的原来版本和嵌套类版本的优缺点。

18.5 联合：节省空间的类

792 联合（union）是一种特殊的类。一个 union 对象可以有多个数据成员，但在任何时刻，只有一个成员可以有值。当将一个值赋给 union 对象的一个成员的时候，其他所有成员都变为未定义的。

为 union 对象分配的存储的量至少与包含其最大数据成员的一样多。像任何类一样，一个 union 定义了一个新的类型。

1. 定义联合

联合提供了便利的办法表示一组相互排斥的值，这些值可以是不同类型的。作为例子，我们可能有一个处理不同种类数值或字符数据的过程。该过程可以定义一个 union 来保存这些值：

```
// objects of type TokenValue have a single member,
// which could be of any of the listed types
union TokenValue {
    char    cval;
```

```

    int    ival;
    double dval;
};

```

一个 union 定义以关键字 union 开始，后接（可选的）union 名字，以及一组以花括号括住的成员声明。这段代码定义了名为 TokenValue 的 union，它可以保存一个 char、int、char 指针或 double 值。本节介绍省略 union 名字意味着什么。

像任何类一样，union 类型定义了与 union 类型的对象相关联的内存是多少。每个 union 对象的大小在编译时是固定的：它至少与 union 的最大数据成员一样大。

2. 没有静态数据成员、引用成员或类数据成员

某些（但不是全部）类特征同样适用于 union。例如，像任何类一样，union 可以指定保护标记使成员成为公用的、私有的或受保护的。默认情况下，union 表现得像 struct：除非另外指定，否则 union 的成员都为 public 成员。

union 也可以定义成员函数，包括构造函数和析构函数。但是，union 不能作为基类使用，所以成员函数不能为虚数。

union 不能具有静态数据成员或引用成员，而且，union 不能具有定义了构造函数、析构函数或赋值操作符的类类型的成员：

```

union illegal_members {
    Screen s;           // error: has constructor
    static int is;      // error: static member
    int &rfi;           // error: reference member
    Screen *ps;         // ok: ordinary built-in pointer type
};

```

这个限制包括了具有带构造函数、析构函数或赋值操作符的成员的类。

793

3. 使用联合类型

union 的名字是一个类型名：

```

TokenValue first_token = {'a'};    // initialized TokenValue
TokenValue last_token;              // uninitialized TokenValue object
TokenValue *pt = new TokenValue;    // pointer to a TokenValue object

```

像其他内置类型一样，默认情况下 union 对象是未初始化的。可以用与显式初始化（12.4.5 节）简单类对象一样的方法显式初始化 union 对象。但是，只能为第一个成员提供初始化式。该初始化式必须括在一对花括号中。first_token 的初始化给它的 cval 成员一个值。

4. 使用联合的成员

可以使用普通成员访问操作符（. 和 ->）访问 union 类型对象的成员：

```

last_token.cval = 'z';
pt->ival = 42;

```

给 union 对象的某个数据成员一个值使得其他数据成员变为未定义的。使用 union 对象时，我们必须总是知道 union 对象中当前存储的是什么类型的值。通过错误的数据成员检索保存在 union 对象中的值，可能会导致程序崩溃或者其他不正确的程序行为。



最佳
实践

避免通过错误成员访问 union 值的最佳办法是，定义一个单独的对象跟踪 union 中存储了什么值。这个附加对象称为 union 的判别式 (discriminant)。

5. 嵌套联合

union 最经常用作嵌套类型，其中判别式是外围类的一个成员：

```
class Token {
public:
    // indicates which kind of value is in val
    enum TokenKind {INT, CHAR, DBL};
    TokenKind tok;
    union {                // unnamed union
        char    cval;
        int     ival;
        double  dval;
    } val;                // member val is a union of the 3 listed types
};
```

这个类中，用枚举对象 tok 指出 val 成员中存储了哪种值，val 成员是一个（未命名的）union，它保存 char、int 或 double 值。

经常使用开关 (switch) 语句 (6.6 节) 测试判别式，然后根据 union 中当前存储的值进行处理：

```
Token token;
switch (token.tok) {
case Token::INT:
    token.val.ival = 42; break;
case Token::CHAR:
    token.val.cval = 'a'; break;
case Token::DBL:
    token.val.dval = 3.14; break;
}
```

6. 匿名联合

不用于定义对象的未命名 union 称为匿名联合 (anonymous union)。匿名 union 的成員的名字出现在外围作用域中。例如，使用匿名 union 重写的 Token 类如下：

```
class Token {
public:
    // indicates which kind of token value is in val
    enum TokenKind {INT, CHAR, DBL};
    TokenKind tok;
    union {                // anonymous union
        char    cval;
        int     ival;
        double  dval;
    };
};
```

因为匿名 union 不提供访问其成员的途径，所以将成员作为定义匿名 union 的作用域的一部分直接访问。重写前面的 switch 以便使用类的匿名 union 版本，如下：

```
Token token;
switch (token.tok) {
case Token::INT:
    token.ival = 42; break;
```

```

case Token::CHAR:
    token.cval = 'a'; break;
case Token::DBL:
    token.dval = 3.14; break;
}

```

 注解

匿名 union 不能有私有成员或受保护成员，也不能定义成员函数。

795

18.6 局部类

可以在函数体内部定义类，这样的类称为**局部类** (local class)。一个局部类定义了一个类型，该类型只在定义它的局部作用域中可见。与嵌套类不同，局部类的成员是严格受限的。

 注解

局部类的所有成员（包括函数）必须完全定义在类定义体内部，因此，局部类远不如嵌套类有用。

实际上，成员完全定义在类中的要求限制了局部类成员函数的复杂性。局部类中的函数很少超过数行代码，超过的话，阅读者会难以理解代码。

类似地，不允许局部类声明 static 数据成员，没有办法定义它们。

1. 局部类不能使用函数作用域中的变量

局部类可以访问的外围作用域中的名字是有限的。局部类只能访问在外围作用域中定义的类型名、static 变量 (7.5.2 节) 和枚举成员，不能使用定义该类的函数中的变量：

```

int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar is local to foo
    class Bar {
    public:
        Loc locVal; // ok: uses local type name
        int barVal;
        void fooBar(Loc l = a) // ok: default argument is Loc::a
        {
            barVal = val; // error: val is local to foo
            barVal = ::val; // ok: uses global object
            barVal = si; // ok: uses static local object
            locVal = b; // ok: uses enumerator
        }
    };
    // ...
}

```

2. 常规保护规则适用于局部类

外围函数对局部类的私有成员没有特殊访问权，当然，局部类可以将外围函数设为友元。

796

 最佳实践

实际上，局部类中 private 成员几乎是不必要的，通常局部类的所有成员都为 public 成员。

可以访问局部类的程序部分是非常有限的。局部类封装在它的局部作用域中，进一步通过信息隐藏进行封装通常是不必要的。

3. 局部类中的名字查找

局部类定义体中的名字查找方式与其他类的相同。类成员声明中所用的名字必须在名字使用之前出现在作用域中，成员定义中所用的名字可以出现在局部类作用域的任何地方。没有确定为类成员的名字首先在外围局部作用域中进行查找，然后在包围函数本身的作用域中查找。

4. 嵌套的局部类

可以将一个类嵌套在局部类内部。这种情况下，嵌套类定义可以出现在局部类定义体之外，但是，嵌套类必须在定义局部类的同一作用域中定义。照常，嵌套类的名字必须用外围类的名字进行限定，并且嵌套类的声明必须出现在局部类的定义中：

```
void foo()
{
    class Bar {
    public:
        // ...
        class Nested;    // declares class Nested
    };
    // definition of Nested
    class Bar::Nested {
        // ...
    };
}
```

嵌套在局部类中的类本身是一个带有所有附加限制的局部类。嵌套类的所有成员必须在嵌套类本身定义体内部定义。

18.7 固有的不可移植的特征

编写可以容易从一个机器移到其他机器的低级程序是 C 程序设计语言的一个特点。将程序移到新机器的过程称为“移植”，所以说 C 程序是**可移植的**。

为了支持低级编程，C 语言定义了一些固有不可移植的特征。算术类型的大小随机器不同而变化的事实（2.1 节），就是我们已经遇到过的一个这样的不可移植特征。本节将讨论 C++ 的另外两个从 C 语言继承来的不可移植特征：位域和 volatile 限定符。这些特征可使与硬件接口的直接通信更容易。

C++ 还增加了另一个不可移植特征（从 C 语言继承来的）：链接指示，它使得可以链接到用其他语言编写的程序。

18.7.1 位域

可以声明一种特殊的类数据成员，称为**位域**（bit-field），来保存特定的位数。当程序需要将二进制数据传递给另一程序或硬件设备的时候，通常使用位域。



位域在内存中的布局是机器相关的。

位域必须是整型数据类型，可以是 `signed` 或 `unsigned`。通过在成员名后面接一个冒号以及指定位数的常量表达式，指出成员是一个位域：

```
typedef unsigned int Bit;
```

```
class File {
    Bit mode: 2;
    Bit modified: 1;
    Bit prot_owner: 3;
    Bit prot_group: 3;
    Bit prot_world: 3;
    // ...
};
```

`mode` 位域有两个位，`modified` 只有一位，其他每个成员有三个位。（如果可能）将类定义体中按相邻次序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，在前面的声明中，5 个位域将存储在一个首先与位域 `mode` 关联的 `unsigned int` 中。位是否压缩到整数以及如何压缩与机器有关。



通常最好将位域设为 `unsigned` 类型。存储在 `signed` 类型中的位域的行为由实现定义。

使用位域

用与类的其他数据成员相同的方式访问位域。例如，作为类的 `private` 成员的位域只能从成员函数的定义和类的友元中访问：

798

```
void File::write()
{
    modified = 1;
    // ...
}
```

```
void File::close()
{
    if (modified)
        // ... save contents
}
```

通常使用内置按位操作符（5.3 节）操纵超过一位的位域：

```
enum { READ = 01, WRITE = 02 }; // File modes

int main() {
    File myFile;

    myFile.mode |= READ; // set the READ bit
    if (myFile.mode & READ) // if the READ bit is on
        cout << "myFile.mode READ is set\n";
}
```

定义了位域成员类通常也定义一组内联成员函数来测试和设置位域的值。例如，`File` 类可以定义成员 `isRead` 和 `isWrite`：

```
inline int File::isRead() { return mode & READ; }
inline int File::isWrite() { return mode & WRITE; }

if (myFile.isRead()) /* ... */
```

有了这些成员函数，现在就可以将位域声明为 File 类的私有成员了。

地址操作符 (&) 不能应用于位域，所以不可能有引用类位域的指针，位域也不能是类的静态成员。

18.7.2 volatile 限定符



volatile 的确切含义与机器相关，只能通过阅读编译器文档来理解。使用 volatile 的程序在移到新的机器或编译器时通常必须改变。

直接处理硬件的程序常具有这样的数据成员，它们的值由程序本身直接控制之外的过程所控制。例如，程序可以包含由系统时钟更新的变量。当可以用编译器的控制或检测之外的方式改变对象值的时候，应该将对象声明为 **volatile**。关键字 volatile 是给编译器的指示，指出对这样的对象不应该执行优化。

799

用与 const 限定符相同的方式使用 volatile 限定符。volatile 限定符是一个对类型的附加修饰符：

```
volatile int display_register;
volatile Task *curr_task;
volatile int ixa[max_size];
volatile Screen bitmap_buf;
```

display_register 是 int 类型的 volatile 对象；curr_task 是 volatile 对象的指针；ixa 是整数的 volatile 数组，该数组的每个元素都被认为是 volatile 的；bitmap_buf 是 volatile Screen 对象，它的每个成员都认为是 volatile 的。

用与定义 const 成员函数相同的方式，类也可以将成员函数定义为 volatile，volatile 对象只能调用 volatile 成员函数。

4.2.5 节介绍了 const 限定符与指针的相互作用，volatile 限定符与指针之间也存在同样的相互作用。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 is a volatile int
int *volatile vip;   // vip is a volatile pointer to int
volatile int *ivp;   // ivp is a pointer to volatile int
// vivp is a volatile pointer to volatile int
volatile int *volatile vivp;
int *ip = &v; // error: must use pointer to volatile
*ivp = &v;    // ok: ivp is pointer to volatile
vivp = &v;    // ok: vivp is volatile pointer to volatile
```

像用 const 一样，只能将 volatile 对象的地址赋给指向 volatile 的指针，或者将指向 volatile 类型的指针复制给指向 volatile 的指针。只有当引用为 volatile 时，我们才可以使用 volatile 对象对引用进行初始化。

合成的复制控制不适用于 `volatile` 对象

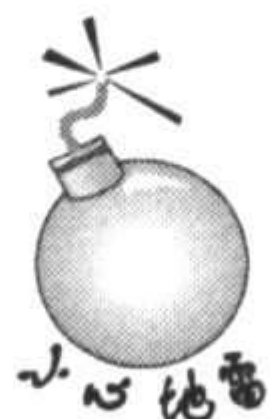
对待 `const` 和 `volatile` 的一个重要区别是,不能使用合成的复制和赋值操作符从 `volatile` 对象进行初始化或赋值。合成的复制控制成员接受 `const` 形参,这些形参是对类类型的 `const` 引用,但是,不能将 `volatile` 对象传递给普通引用或 `const` 引用。

如果类希望允许复制 `volatile` 对象,或者,类希望允许从 `volatile` 操作数或对 `volatile` 操作数进行赋值,它必须定义自己的复制构造函数和/或赋值操作符版本:

```
class Foo {
public:
    Foo(const volatile Foo&);    // copy from a volatile object
    // assign from a volatile object to a non volatile object
    Foo& operator=(volatile const Foo&);
    // assign from a volatile object to a volatile object
    Foo& operator=(volatile const Foo&) volatile;
    // remainder of class Foo
};
```

800

通过将复制控制成员的形参定义为 `const volatile` 引用,我们可以从任何种类的 `Foo` 对象进行复制或赋值:普通 `Foo` 对象、`const Foo` 对象、`volatile Foo` 对象或 `const volatile Foo` 对象。



虽然可以定义复制控制成员来处理 `volatile` 对象,但更深入的问题是复制 `volatile` 对象是否有意义,对该问题的回答与在任意特定程序中使用 `volatile` 的原因密切相关。

18.7.3 链接指示: `extern "C"`

C++程序有时需要调用用其他程序设计语言编写的函数,最常见的一语言是 C 语言。像任何名字一样,必须声明用其他语言编写的函数的名字,该声明必须指定返回类型和形参表。编译器按处理普通 C++函数一样的方式检查对外部语言函数的调用,但是,编译器一般必须产生不同的代码来调用用其他语言编写的函数。C++使用链接指示 (linkage directive) 指出任意非 C++函数所用的语言。

1. 声明非 C++函数

链接指示有两种形式: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,它必须出现在函数的第一次声明上。

作为例子,看看头文件 `cstdlib` 中声明的一些 C 函数。该头文件中的声明形如:

```
// illustrative linkage directives that might appear in the C++ header <cstring>
// single statement linkage directive
extern "C" size_t strlen(const char *);
// compound statement linkage directive
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

801

第一种形式由关键字 `extern` 后接字符串面值，再接“普通”函数声明构成。字符串面值指出编写函数所用的语言。

通过将几个函数的声明放在跟在链接指示之后的花括号内部，可以给它们设定相同的链接。花括号的作用是将应用链接指示的声明聚合起来，忽略了花括号，花括号中声明的函数名就是可见的，就像在花括号之外声明函数一样。

2. 链接指示与头文件

可以将多重声明形式应用于整个头文件。例如，C++的 `cstring` 头文件可以像这样：

```
// compound statement linkage directive
extern "C" {
#include <string.h>      // C functions that manipulate C-style strings
}
```

当将 `#include` 指示放在复合链接指示的花括号中的时候，假定头文件中的所有普通函数声明都是用链接指示的语言编写的函数。链接指示可以嵌套，所以，如果头文件包含了带链接指示的函数，该函数的链接不受影响。



允许将 C++ 从 C 函数库继承而来的函数定义为 C 函数，但不是必须定义为 C 函数——决定是用 C 还是用 C++ 实现 C 函数库，是每个 C++ 实现的事情。

3. 导出 C++ 函数到其他语言

通过对函数定义使用链接指示，使得用其他语言编写的程序可以使用 C++ 函数：

```
// the calc function can be called from C programs
extern "C" double calc(double dparm) { /* ... */ }
```

当编译器为该函数产生代码的时候，它将产生适合于指定语言的代码。



用链接指示定义的函数的每个声明都必须使用相同的链接指示。

4. 链接指示支持的语言

要求编译器支持对 C 语言的链接指示。编译器可以为其他语言提供链接说明。例如，**`extern "Ada"`**、**`extern "FORTRAN"`** 等。



支持什么语言随编译器而变。你必须查阅用户指南，获得关于编译器可以提供的任意非 C 链接说明的进一步信息。

802

对链接到 C 的预处理器支持

有时需要在 C 和 C++ 中编译同一源文件。当编译 C++ 时，自动定义预处理器名字 `__cplusplus` (两个下划线)，所以，可以根据是否正在编译 C++ 有条件地包含代码。

```
#ifdef __cplusplus
// ok: we're compiling C++
extern "C"
#endif
int strcmp(const char*, const char*);
```

5. 重载函数与链接指示

链接指示与函数重载之间的相互作用依赖于目标语言。如果语言支持重载函数，则为该语言实现链接指示的编译器很可能也支持 C++ 的这些函数的重载。

C++ 保证支持的唯一语言是 C。C 语言不支持函数重载，所以，不应该对下面的情况感到惊讶：在一组重载函数中只能为一个 C 函数指定链接指示。用带给定名字的 C 链接声明多于一个函数是错误的：

```
// error: two extern "C" functions in set of overloaded functions
extern "C" void print(const char*);
extern "C" void print(int);
```

在 C++ 程序中，重载 C 函数很常见，但是，重载集合中的其他函数必须都是 C++ 函数：

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };
// the C function can be called from C and C++ programs
// the C++ functions overload that function and are callable from C++
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

可以从 C 程序和 C++ 程序调用 calc 的 C 版本。其余函数是带类型形参的 C++ 函数，只能从 C++ 程序调用。声明的次序不重要。

6. 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。为了声明用其他程序设计语言编写的函数的指针，必须使用链接指示：

```
// pf points to a C function returning void taking an int
extern "C" void (*pf)(int);
```

使用 pf 调用函数的时候，假定该调用是一个 C 函数调用而编译该函数。



C 函数的指针与 C++ 函数的指针具有不同的类型，不能将 C 函数的指针初始化或赋值为 C++ 函数的指针（反之亦然）。

存在这种不匹配的时候，会给出编译时错误：

```
void (*pf1)(int); // points to a C++ function
extern "C" void (*pf2)(int); // points to a C function
pf1 = pf2; // error: pf1 and pf2 have different types
```



一些 C++ 编译器可以接受前面的赋值作为语言扩展，尽管严格说来它是非法的。

7. 应用于整个声明的链接指示

使用链接指示的时候，它应用于函数和任何函数指针，作为返回类型或形参类型使用：

```
// f1 is a C function; its parameter is a pointer to a C function
```

```
extern "C" void f1(void(*) (int));
```

这个声明是说，f1 是一个不返回值的 C 函数，它有一个形参，该形参是不返回值并接受单个形参的函数的指针。链接指示应用于该函数指针以及 f1。调用的时候，必须将 C 函数名字或 C 函数指针传递给它。

因为链接指示应用于一个声明中的所有函数，所以必须使用类型别名，以便将 C 函数的指针传递给 C++ 函数：

```
// FC is a pointer to C function
extern "C" typedef void FC(int);
// f2 is a C++ function with a parameter that is a pointer to a C function
void f2(FC *);
```

习题

习题 18.34 解释下面这些声明，并指出它们是否合法：

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

804

小结

C++ 讲述了几个专门针对一些特定问题的特殊设施。

类的自定义内存管理有两种方式：定义自己的内部内存分配，以简化自己的数据成员的分配；定义自己的、类特定的 operator new 和 operator delete 函数，在分配类类型的新对象时使用它们。

一些程序需要在运行时直接询问对象的动态类型。运行时类型识别 (RTTI) 为这类程序设计提供了语言级支持。RTTI 只适用于定义了虚函数的类，没有定义虚函数的类型的类型信息是可用的但反映静态类型。

普通对象的指针都是有类型的。定义类成员的指针的时候，指针类型必须也封装指针所指向的类的类型。可以将成员指针绑定到具有相同类型的任意类成员，引用成员指针的时候，必须指定从中获取成员的对象。

C++ 还定义了另外几个聚集类型：

- 嵌套类，它是在另一个类的作用域中定义的类，这样的类经常定义为其外围类的具体实现类。
- 联合，是只能包含简单数据成员的一种特殊类。union 类型的对象在任意时刻只能为它的一个数据成员定义值。联合经常嵌套在其他类类型内部。
- 局部类，是局部于函数而定义的非常简单的类。局部类的所有成员必须定义在类定义体中，局部类没有静态数据成员。

C++ 还支持几种固有的不可移植的特征，包括位域和 volatile（它们可使与硬件接口更容易）以及链接指示（它使得与用其他语言编写的程序接口更容易）。

术语

allocator 类 标准库类，支持原始未构造内存的类特定的分配。allocator 类是一个类模板，定义了成员函数，对 allocator 的模板形参类型的对象进行 allocate、deallocate、construct 和 destroy。

anonymous union (匿名联合) 不用于定义对象的未命名联合。直接引用匿名联合的成员。这些联合不能具有成员函数，也不能具有私有或受保护成员。

bit-field (位域) 有符号或无符号整型类成员，它指定了分配给成员的位数。如果可能，将类中以连续次序定义的位域压缩到公共整型值中。

delete 表达式 delete 表达式撤销特定类型的动态分配对象，并释放该对象所用的内存。delete[] 表达式撤销特定类型动态分配数组的元素，并释放数组使用的内存。这些表达式使用库函数或类特定的 operator delete 函数的对应版本来释放保存对象或数组的原始内存。

discriminant (判别式) 一种编程技术，使用对象来确定任意给定时刻联合中保存的实际类型。

dynamic_cast 执行从基类类型到派生类型的带检查强制转换的操作符。基类类型必须定义至少一个 virtual 函数。这个操作符检查引用或指针所绑定对象的动态类型。如果对象类型与转换的类型（或者从该类型派生的类型）相同，则进行转换；否则，指针转换返回 0 指针，引用的转换抛出一个异常。

freelist (自由列表) 一种内存管理技术，涉及预先分配未构造内存以保存在需要时创建的对象。释放对象的时候，将它们的内存放回自由列表，而不是返还给系统。

linkage directive (链接指示) 一种机制，用

于允许从 C++ 程序调用以不同语言编写的函数。所有编译器都支持调用 C 和 C++ 函数，是否支持任何其他语言随编译器而定。

local class (局部类) 在函数内部定义的类。局部类只在定义它的函数内部可见，类的所有成员必须定义在类定义体内部。局部类不能有静态成员，局部类成员不能访问外围函数中定义的局部变量，它们可以使用外围函数中定义的类型名、静态变量和枚举成员。

member operator new and delete (成员操作符 new 和 delete) 类成员函数，覆盖由全局库函数 operator new 和 operator delete 执行的默认内存分配。可以定义这些函数的对象形式 (new) 和数组形式 (new[])。成员 new 和 delete 函数隐式声明为 static，这些操作符分配（释放）内存，它们由 new (delete) 表达式自动使用，new (delete) 表达式处理对象的初始化和撤销。

nested class (嵌套类) 在其他类内部定义的类。嵌套类定义在其外围作用域内部：嵌套类名字必须在定义它们的类作用域中唯一，但可以在外围类之外的作用域中重用。在外围类之外访问嵌套类需要使用作用域操作符指定包含嵌套类的作用域。

nested type (嵌套类型) 嵌套类的同义词。

new expression (new 表达式) new 表达式分配和构造特定类型的对象。new[] 表达式分配和构造对象数组。这些表达式使用库函数 operator new 的对应版本分配原始内存，并在该内存中构造特定类型的对象或数组。

operator delete 释放由 operator new 分配的未类型化的未构造内存的库函数。库函数 operator delete[] 释放由 operator new[] 分配的用于保存数组的内存。

operator new 一个库函数，它分配给定大小未类型化的未构造内存。库函数 `operator new[]` 为数组分配原始内存。这些库函数提供了比库类 `allocator` 更基本的分配机制。现代 C++ 程序应使用 `allocator` 类而不是这些库函数。

placement new expression (定位 new 表达式) 在特定内存中构造对象的 `new` 的形式。它不进行分配，相反，它接受指定在何处构造对象的实参。它是对 `allocator` 类的 `construct` 成员所提供的行为的低级模拟。

pointer to member (成员指针) 封装类类型以及指针所指向的成员类型的指针。成员指针的定义必须指定类名以及指针可以指向的成员的类型：

```
T C::*pmem = &C::member;
```

这个语句将 `pmem` 定义为指针，它可以指向名为 `C` 的类的类型为 `T` 的成员，并将 `pmem` 初始化为 `C` 中名为 `member` 的成员。对该指针解引用的时候，它必须是绑定到类型 `C` 的对象或指向类型 `C` 的指针：

```
classobj.*pmem;  
classptr->*pmem;
```

从 `classptr` 所指对象的 `classobj` 对象获取 `member`。

portable (可移植的) 一个用来描述用相对较少的努力就可以移到新机器的程序的术语。

run-time type identification (运行时类型识别) 用来描述语言和库设施的术语，这种设施允许在运行时获得引用或指针的动态类型。RTTI 操作符 `typeid` 和 `dynamic_cast` 只为带虚函数的类类型提供动态类型，应用于其他类型的时候，返回引用或指针的静态类型。

typeid 一元操作符，接受一个表达式，并返回描述表达式类型的名为 `type_info` 的库类型的对象引用。如果表达式是具有虚函数的类型的对象，就返回表达式的动态类型；如果该类型是没有定义虚函数的引用、指针或其他类型，就返回引用、指针或对象的静态类型。

type_info 描述类型的库类型。`type_info` 类是固有机相关，但任何库都必须定义带有表 18-2 (18.2.4 节) 中所列出成员的 `type_info`。`type_info` 对象不能复制。

union (联合) 类形式的聚合类型，它可以定义多个数据成员，但在任意点只有一个成员可以具有值。联合的成员必须为简单类型：内置类型，复合类型，没有定义构造函数、析构函数或赋值操作符的类类型。联合可以有成员函数，包括构造函数和析构函数。联合不能作基类使用。

volatile 类型限定符，告诉编译器可以在程序的直接控制之外改变一个变量。它是告诉编译器不能执行某些优化的信号。

附录

标准库

目录

A.1 标准库名字和头文件	676
A.2 算法简介	677
A.3 再谈 IO 库	688

本附录提供了标准库更多有用的细节。首先将标准库中的名字集中在一起，表 A-1 列出每个名字以及定义该名字的头文件。

第 11 章讨论过标准库算法，那一章举例说明了怎样使用一些比较常用的算法，并描述了算法库的体系结构。本附录中将列出所有算法，按它们执行的操作种类组织。

本附录的最后考察了另一些 IO 库的功能，包括格式控制、未格式化的 IO 和文件随机访问。每个 IO 类型定义了格式状态和控制这些状态的相关函数集合。使用这些格式状态我们能够更好地控制输入和输出的工作。我们已经做过的 IO 都是格式化的——输入和输出例程了解使用的类型，并据此格式化输入和输出数据。还有未格式化的 IO 函数，它们在 char 级别处理流，不解释数据。第 8 章介绍过 fstream 类型能读写同一文件，本附录将介绍怎样做到这一点。

A.1 标准库名字和头文件

本书的示例程序大多没有给出编译程序所需要的实际#include指示。为了方便读者阅读，表 A-1 列出了示例程序使用过的标准库名字以及包含标准库名字的头文件。

表 A-1 标准库名字和头文件

名字	头文件	名字	头文件
abort	<cstdlib>	ios_base	<ios_base>
accumulate	<numeric>	isalpha	<cctype>
allocator	<memory>	islower	<cctype>
auto_ptr	<memory>	ispunct	<cctype>
back_inserter	<iterator>	isspace	<cctype>
bad_alloc	<new>	istream	<iostream>
bad_cast	<typeinfo>	istream_iterator	<iterator>
bind2nd	<functional>	istringstream	<sstream>
bitset	<bitset>	isupper	<cctype>
boolalpha	<iostream>	left	<iostream>
cerr	<iostream>	less_equal	<functional>
cin	<iostream>	list	<list>
copy	<algorithm>	logic_error	<stdexcept>
count	<algorithm>	lower_bound	<algorithm>
count_if	<algorithm>	make_pair	<utility>
cout	<iostream>	map	<map>
dec	<iostream>	max	<algorithm>
deque	<deque>	min	<algorithm>
endl	<iostream>	multimap	<map>
ends	<iostream>	multiset	<set>
equal_range	<algorithm>	negate	<functional>
exception	<exception>	noboolalpha	<iostream>
fill	<algorithm>	noshowbase	<iostream>
fill_n	<algorithm>	noshowpoint	<iostream>
find	<algorithm>	noskipws	<iostream>
find_end	<algorithm>	not1	<functional>
find_first_of	<algorithm>	nounitbuf	<iostream>
fixed	<iostream>	nouppercase	<iostream>
flush	<iostream>	nth_element	<algorithm>
for_each	<algorithm>	oct	<iostream>
front_inserter	<iterator>	ofstream	<fstream>
fstream	<fstream>	ostream	<iostream>
getline	<string>	ostream_iterator	<iterator>
hex	<iostream>	ostringstream	<sstream>
ifstream	<fstream>	out_of_range	<stdexcept>
inner_product	<numeric>	pair	<utility>
inserter	<iterator>	partial_sort	<algorithm>
internal	<iostream>	plus	<functional>

(续)

名字	头文件	名字	头文件
priority_queue	<queue>	sqrt	<cmath>
ptrdiff_t	<cstdint>	stable_sort	<algorithm>
queue	<queue>	stack	<stack>
range_error	<stdexcept>	strcmp	<cstring>
replace	<algorithm>	strcpy	<cstring>
replace_copy	<algorithm>	string	<string>
reverse_iterator	<iterator>	stringstream	<sstream>
right	<iostream>	strlen	<cstring>
runtime_error	<stdexcept>	strncpy	<cstring>
scientific	<iostream>	terminate	<exception>
set	<set>	tolower	<cctype>
set_difference	<algorithm>	toupper	<cctype>
set_intersection	<algorithm>	type_info	<typeinfo>
set_union	<algorithm>	unexpcted	<exception>
setfill	<iomanip>	uninitialized_copy	<memory>
setprecision	<iomanip>	unitbuf	<iostream>
setw	<iomanip>	unique	<algorithm>
showbase	<iostream>	unique_copy	<algorithm>
showpoint	<iostream>	upper_bound	<algorithm>
size_t	<cstdint>	uppercase	<iostream>
skipws	<iostream>	vector	<vector>
sort	<algorithm>		

A.2 算法简介

第 11 章介绍了一般算法并列出了它们的底层体系结构。标准库定义了 100 多个算法，学习如何使用它们需要理解它们的结构，而不是记住每个算法的细节。本节描述每个算法，其中，按算法执行行为的类型组织这些算法。

A.2.1 查找对象的算法

find 和 count 算法在输入范围中查找指定值。find 返回元素的迭代器，count 返回匹配元素的数目。

1. 简单查找算法

这些算法要求输入迭代器。find 和 count 算法查找特定元素，find 算法返回引用第一个匹配元素的迭代器，count 算法返回元素在输入序列中出现次数的计数。

```
find(beg, end, val)
count(beg, end, val)
```

在输入范围中查找等于 val 的元素，使用基础类型的相等 (==) 操作符。find 返回第一个匹配元素的迭代器，如果不存在匹配元素就返回 end。count 返回 val 出现次数的计数。

```
find_if(beg, end, unaryPred)
```

count_if(beg, end, unaryPred)

在 unaryPred 为真的输入范围中查找。谓词必须接受一个形参，形参类型为输入范围的 value_type，并且返回可以用作条件的类型。

find_if 返回第一个使 unaryPred 为真的元素的迭代器，如果不存在这样的元素就返回 end。count_if 对每个元素应用 unaryPred，并返回使 unaryPred 为真的元素的数目。

2. 查找许多值中的一个的算法

这些算法要求两对前向迭代器。它们在第一个范围中查找与第二个范围中任意元素相等的第一个（或最后一个）元素。beg1 和 end1 的类型必须完全匹配，beg2 和 end2 的类型也必须完全匹配。

不要求 beg1 和 beg2 的类型完全匹配，但是，必须有可能对这两个序列的元素类型进行比较。例如，如果第一个序列是 list<string>，则第二个可以是 vector<char*>。

每个算法都是重载的。默认情况下，使用元素类型的==操作符测试元素，或者，可以指定一个谓词，该谓词接受两个形参，并返回表示这两个元素间的测试成功或失败的 bool 值。

find_first_of(beg1, end1, beg2, end2)

返回第二个范围的任意元素在第一个范围的首次出现的迭代器，如果找不到匹配就返回 end1。

find_first_of(beg1, end1, beg2, end2, binaryPred)

使用 binaryPred 比较来自两个序列的元素，返回第一个范围中第一个这种元素的迭代器：当对该元素和来自第二个范围的一个元素应用 binaryPred 的时候，binaryPred 为真。如果不存在这样的元素，就返回 end1。

find_end(beg1, end1, beg2, end2)

find_end(beg1, end1, beg2, end2, binaryPred)

与 find_first_of 相像的操作符，只不过它查找来自第二个序列的任意元素的最后一次出现。

作为例子，如果第一个序列是 0,1,1,2,2,4,0,1 而第二个序列是 1,3,5,7,9，则 find_end 返回表示输入范围中最后一个元素的迭代器，而 find_first_of 将返回第二个元素的迭代器——本例中，它返回输入序列中的第一个 1。

3. 查找子序列的算法

这些算法要求前向迭代器。它们查找子序列而不是单个元素。如果找到了子序列，就返回子序列中第一个元素的迭代器；如果找不到子序列，就返回输入范围的 end 迭代器。

每个函数都是重载的。默认情况下，使用相等操作符(==)比较元素；第二个版本允许程序员提供一个谓词代替==进行测试。

adjacent_find(beg, end)

adjacent_find(beg, end, binaryPred)

返回重复元素的第一个相邻对。如果没有相邻的重复元素，就返回 end。在第一种情况下，使用==找到重复元素，第二种情况下，重复元素是使 binaryPred 为真的那些元素。

search(beg1, end1, beg2, end2)

search(beg1, end1, beg2, end2, binaryPred)

返回输入范围中第二个范围作为子序列出现的第一个位置。如果找不到子序列，就返回 end1。beg1 和 beg2 的类型可以不同，但必须是兼容的：必须能够比较两个序列中的元素。

```
search_n(beg, end, count, val)
search_n(beg, end, count, val, binaryPred)
```

返回 count 个相等元素的子串的开头迭代器。如果不存在这样的子串，就返回 end。第一个版本查找给定 val 的 count 次出现，第二个版本查找使 binaryPred 为真的 count 次出现。

A.2.2 其他只读算法

这些算法要求用于前两个实参的输入迭代器。equal 和 mismatch 算法还接受一个附加输入迭代器，该迭代器表示第二个范围。第二个序列中的元素至少与第一个序列一样多，如果第二个序列元素较多，就忽略多余元素；如果第二个序列元素较少，就会出错并导致未定义的运行时行为。

照常，表示输入范围的迭代器的类型必须完全匹配。beg2 的类型必须与 beg1 的类型兼容，即必须能够比较两个序列中的元素。

equal 和 mismatch 函数是重载的：一个版本使用元素相等操作符 (==) 测试元素对，另一个使用谓词。

```
for_each(beg, end, f)
```

对其输入范围中的每个元素应用函数（或函数对象（14.8 节））f。如果 f 有返回值，就忽略该返回值。迭代器是输入迭代器，所以 f 不能写元素。通常，用有副作用的函数使用 for_each。例如，f 可以显示范围中的值。

813

```
mismatch(beg1, end1, beg2)
mismatch(beg1, end1, beg2, binaryPred)
```

比较两个序列中的元素，返回一对表示第一个不匹配元素的迭代器。如果所有元素都匹配，则返回的 pair 是 end1，以及 beg2 中偏移量为第一个序列长度的迭代器。

```
equal(beg1, end1, beg2)
equal(beg1, end1, beg2, binaryPred)
```

确定两个序列是否相等。如果输入范围中的每个元素都与从 beg2 开始的序列中的对应元素相等，就返回 true。

例如，给定序列 meet 和 meat，对 mismatch 的调用将返回一个 pair 对象，其中包含指向第一个序列中第二个 e 的迭代器，以及指向第二个序列中元素 a 的迭代器。如果，第二个序列是 meeting，并调用 equal，则返回的将是 end1 和表示第二个范围中元素 i 的迭代器。

A.2.3 二分查找算法

虽然可以与前向迭代器一起使用这些算法，它们还是提供了与随机访问迭代器一起使用的特殊版本，它们的速度更快。

这些算法执行二分查找，这意味着输入序列必须是已排序的。这些算法的表现类似于同名的关联容器成员（10.5.2 节）。equal_range、lower_bound 和 upper_bound 算法返回一个迭代器，该迭代器指向容器中的位置，可以将给定元素插入到这个位置而仍然保持容器的排序。如果元素比容器中任意其他元素都大，则返回的迭代器会是超出末端迭代器。

每个算法提供两个版本：第一个使用元素类型的小于操作符 (<) 测试元素，第二个使用指定的比较关系。

```
lower_bound(beg, end, val)
lower_bound(beg, end, val, comp)
```

返回第一个这种位置的迭代器：可以将 val 插入到该位置而仍然保持顺序。

```
upper_bound(beg, end, val)
upper_bound(beg, end, val, comp)
```

返回最后一个这种位置的迭代器：可以将 val 插入到该位置而仍然保持顺序。

```
equal_range(beg, end, val)
equal_range(beg, end, val, comp)
```

返回一个表示子范围的迭代器对，可以将 val 插入到该子范围而仍然保持顺序。

```
binary_search(beg, end, val)
binary_search(beg, end, val, comp)
```

返回一个 bool 值，表示序列是否包含与 val 相等的元素。如果 $x < y$ 和 $x > y$ 都获得假值，就认为两个值 x 和 y 相等。

814

A.2.4 写容器元素的算法

许多算法写容器元素。可以根据所操作的迭代器种类，以及是写输入范围的元素还是写到特定目的地，来区分这些算法。

最简单的算法读序列中的元素，只要求输入迭代器。那些写回输入序列的算法要求前向迭代器。一些算法反向读取序列，所以要求双向迭代器。写至单独目的地的算法，照常假定目的地足以保存输入。

1. 只写元素不读元素的算法

这些算法要求表示目的地的输出迭代器。它们接受指定数量的第二个实参并将该数目的元素写到目的地。

```
fill_n(dest, cnt, val)
generate_n(dest, cnt, Gen)
```

将 cnt 个值写到 dest。fill_n 函数写 val 值的 cnt 个副本，generate_n 对发生器 Gen() 进行 cnt 次计算。发生器是一个函数（或函数对象（14.8 节）），每次调用它都期待产生一个不同的返回值。

2. 使用输入迭代器写元素的算法

这些操作每一个读一个输入序列，并写到由 dest 表示的输出序列。它们要求 dest 是一个输出迭代器，而表示输入范围的迭代器必须是输入迭代器。调用者负责保证 dest 可以保存给定输入序列所需数量的元素。这些算法返回 dest，dest 增量至指向所写最后元素的下一位置。

```
copy(beg, end, dest)
```

将输入范围复制到从迭代器 dest 开始的序列。

```
transform(beg, end, dest, unaryOp)
transform(beg, end, beg2, dest, binaryOp)
```

对输入范围中每个元素应用指定操作，将结果写到 dest。第一个版本对输入范围中每个元素应用一元操作。第二个版本对元素对应用二元操作，它从由 beg 和 end 表示的序列接受二元操作的第一个实参，从开始于 beg2 的第二个序列接受第二个实参。程序员必须保证开始于 beg2 的序列具有至少与第一个序列一样多的元素。

```
replace_copy(beg, end, dest, old_val, new_val)
replace_copy_if(beg, end, dest, unaryPred, new_val)
```

将每个元素复制到 dest, 用 new_val 代替指定元素。第一个版本代替那些 ==old_val 的元素, 第二个版本代替那些使 unaryPred 为真的元素。

815

```
merge(beg1, end1, beg2, end2, dest)
merge(beg1, end1, beg2, end2, dest, comp)
```

两个输入序列都必须是已排序的。将合并后的序列写至 dest。第一个版本使用 < 操作符比较元素, 第二个版本使用给定的比较关系。

3. 使用前向迭代器写元素的算法

这些算法要求前向迭代器, 因为它们修改输入序列中的元素。

```
swap(elem1, elem2)
iter_swap(iter1, iter2)
```

这些函数的形参是引用, 所以实参必须是可写的。交换指定元素或由给定迭代器表示的元素。

```
swap_ranges(beg1, end1, beg2)
```

用开始于 beg2 的第二个序列中的元素交换输入范围中的元素。范围必须不重叠。程序员必须保证开始于 beg2 的序列至少与输入序列一样大。返回 beg2, beg2 增量到指向被交换的最后一个元素之后的元素。

```
fill(beg, end, val)
generate(beg, end, Gen)
```

将新值赋给输入序列中的每个元素。fill 赋 val 值, generate 执行 Gen() 来创建新值。

```
replace(beg, end, old_val, new_val)
replace_if(beg, end, unaryPred, new_val)
```

用 new_val 代替每个匹配元素。第一个版本使用 == 将元素和 old_val 比较, 第二个版本对每个元素执行 unaryPred, 代替使 unaryPred 为真的那些元素。

4. 使用双向迭代器写元素的算法

这些算法要求在序列中往回走的能力, 所以它们要求双向迭代器。

```
copy_backward(beg, end, dest)
```

按逆序将元素复制到输出迭代器 dest。返回 dest, dest 增量至指向被复制的最后一个元素的下一位置。

```
inplace_merge(beg, mid, end)
inplace_merge(beg, mid, end, comp)
```

将同一序列中的两个相邻子序列合并为一个有序序列: 将从 beg 到 mid 和从 mid 到 end 的子序列合并为从 beg 到 end 的序列。第一个版本使用 < 比较元素, 第二个版本使用指定的比较关系。返回 void。

816

A.2.5 划分与排序算法

排序和划分算法为容器元素排序提供不同的策略。

partition 将输入范围中的元素划分为两组, 第一组由满足给定谓词的元素构成, 第二组由不满足谓词的元素构成。例如, 可以根据元素是否为奇数划分容器中的元素, 或者, 根据单词

是否以大写字母开头，诸如此类。

每个排序和划分算法都提供稳定和不稳定版本，稳定算法维持相等元素的相对次序。例如，给定序列

```
{ "psheW", "Honey", "tiger", "Pooh" }
```

基于单词是否以大写字母开头的稳定算法，产生维持两个单词类的相对次序的序列：

```
{ "Honey", "Pooh", "psheW", "tiger" }
```

稳定算法完成更多工作，因此相比于不稳定算法，可能运行较慢且使用更多内存。

1. 划分算法

这些算法要求双向迭代器。

```
stable_partition(beg, end, unaryPred)
partition(beg, end, unaryPred)
```

使用 unaryPred 划分输入序列。使 unaryPred 为真的元素放在序列开头，使 unaryPred 为假的元素放在序列末尾。返回一个迭代器，该迭代器指向使 unaryPred 为真的最后元素的下一位置。

2. 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版本，一个版本使用元素操作符 < 比较元素，另一个版本接受一个指定比较关系的额外形参。这些算法返回 void，除了一个例外，partial_sort_copy 返回目的地迭代器。

partial_sort 和 nth_element 算法只完成序列排序的部分工作，经常用它们解决通过对整个序列排序来处理的问题。因为这些操作做的工作较少，所以它们一般比排序整个输入范围要快一些。

```
sort(beg, end)
stable_sort(beg, end)
sort(beg, end, comp)
stable_sort(beg, end, comp)
```

817 对整个范围进行排序。

```
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
```

对 mid-beg 个元素进行排序，也就是说，如果 mid-beg 等于 42，则该函数将有序次序中的最小值元素放在序列中前 42 个位置。partial_sort 完成之后，从 beg 到 mid（但不包括 mid）范围内的元素是有序的，已排序范围内没有元素大于 mid 之后的元素。未排序元素之间的次序是未指定的。

例如，有一个赛跑成绩的集合，我们想要知道前三名的成绩但并不关心其他名次的次序，可以这样对这个序列进行排序：

```
partial_sort(scores.begin(),
              scores.begin() + 3, scores.end());
```

```
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
```

对输入序列中的元素进行排序，将已排序序列中适当数目的元素放入由迭代器 destBeg 和 destEnd 表示的序列。如果目的地范围与输入范围一样大，或者比输入范围大，则将整个输入范围排序且

有序序列从 `destBeg` 开始。如果目的地较小，则只复制适当数目的有序元素。

返回目的地中的迭代器，指向已排序的最后一个元素之后。如果目的地序列比输入范围小或者与输入范围大小相等，返回的迭代器将是 `destEnd`。

```
nth_element(beg, nth, end)
nth_element(beg, nth, end, comp)
```

实参 `nth` 必须是一个迭代器，定位输入序列中的一个元素。运行 `nth_element` 之后，该迭代器表示的元素的值就是：如果整个序列是已排序的，这个位置上应放置的值。容器中的元素也围绕 `nth` 划分：`nth` 之前的元素都小于或等于 `nth` 所表示的值，`nth` 之后的元素都大于或等于它。可以使用 `nth_element` 查找与中值最接近的值：

```
nth_element(scores.begin(),
            scores.begin() + scores.size()/2, scores.end());
```

A.2.6 通用重新排序操作

有几个算法用指定方法对元素进行重新排序。最前面的两个 `remove` 和 `unique` 对容器进行重新排序，以便序列中的第一部分满足一些标准，它们返回标志这个子序列的末尾的迭代器。其他算法，如 `reverse`、`rotate` 和 `random_shuffle`，重新安排整个序列。

这些算法“就地”操作，它们在输入序列本身中重新安排元素。三个重新排序算法提供“复制”版本，这些算法是 `remove_copy`、`rotate_copy` 和 `unique_copy`，将重新排序之后的序列写至目的地，而不是直接重新安排元素。

818

1. 使用前向迭代器的重新排序算法

这些算法对输入序列进行重新排序。它们要求迭代器至少是前向迭代器。

```
remove(beg, end, val)
remove_if(beg, end, unaryPred)
```

通过用要保存的元素覆盖元素而从序列中“移去”元素。被移去的元素是 `==val` 或使 `unaryPred` 为真的那些元素。返回一个迭代器，该迭代器指向未移去的最后一个元素的下一位置。

例如，如果输入序列是 `helloworld` 而 `val` 是 `o`，则 `remove` 调用将通过将序列左移两次覆盖两个元素，即字母 `'o'`。新序列将是 `hellwrlld`，返回的迭代器将指向第一个 `d` 之后的元素。

```
unique(beg, end)
unique(beg, end, binaryPred)
```

“移去”匹配元素的每个连续组，除了第一个之外。返回一个迭代器，该迭代器指向最后一个单一元素的下一位置。第一个版本使用 `==` 确定两个元素是否相同，第二个版本使用谓词测试相邻元素。

例如，如果输入序列是 `boohiss`，则调用 `unique` 之后，第一个序列将包含 `bohiss`。返回的迭代器指向第一个 `s` 之后的元素，序列中剩余的两个元素的值是未指定的。

```
rotate(beg, mid, end)
```

围绕由 `mid` 表示的元素旋转元素。`mid` 处的元素成为第一个元素，从 `mid+1` 到 `end` 的元素其次，后面是从 `beg` 到 `mid`¹ 的范围。返回 `void`。

1. 此处原文 `mid` 有误。——译者注

例如，给定输入序列 `hissboo`，如果 `mid` 表示字符 `b`，则旋转将序列重新排序为 `boohiss`。

2. 使用双向迭代器的重新排序算法

因为这些算法向后处理输入序列，所以它们要求双向迭代器。

```
reverse(beg, end)
reverse_copy(beg, end, dest)
```

颠倒序列中的元素。`reverse` 就地操作，它将重新安排的元素写回输入序列。`reverse_copy` 将元素按逆序复制到输出迭代器 `dest`。照常，程序员必须保证可以安全地使用 `dest`。`reverse` 返回 `void`，`reverse_copy` 返回一个迭代器，该迭代器指向复制到目的地的最后一个元素的下一位置。

3. 写至输出迭代器的重新排序算法

这些算法要求输入序列的前向迭代器以及目的地的输出迭代器。

前面的每个通用重新排序算法都有一个 `_copy` 版本，这些 `_copy` 版本执行相同的重新排序，但是将重新排序之后的元素写至指定目的地序列，而不是改变输入序列。除 `rotate_copy`（它要求前向迭代器）之外，其他的都由输入迭代器指定输入范围。`dest` 迭代器必须是输出迭代器，而且，程序员也必须保证可以安全地写目的地。这些算法返回 `dest` 迭代器，`dest` 迭代器增量至指向被复制的最后元素的下一位置。

```
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred)
```

除了与 `val` 匹配或使 `unaryPred` 返回真的元素之外，其他元素都复制到 `dest`。

```
unique_copy(beg, end, dest)
unique_copy(beg, end, dest, binaryPred)
```

将唯一元素复制到 `dest`。

```
rotate_copy(beg, mid, end, dest)
```

除了保持输入序列不变并将旋转后的序列写至 `dest` 之外，与 `rotate` 很像。返回 `void`。

4. 使用随机访问迭代器的重新排序算法

因为这些算法按随机次序重新安排元素，所以它们要求随机访问迭代器。

```
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
```

打乱输入序列中的元素。第二个版本接受随机数发生器，该函数必须接受并返回迭代器的 `difference_type` 值。两个版本都返回 `void`。

A.2.7 排列算法

考虑下面的三个字符的序列：`abc`。这个序列有 6 种可能的排列：`abc`、`acb`、`bac`、`bca`、`cab` 和 `cba`。基于小于操作符按字典序列出这些排列，即，`abc` 是第一个排列，因为它的第一个元素小于或等于其他每个排列中的首元素，而且，它的第二个元素小于首元素相同的任意排列中的第二个元素。类似地，`acb` 是下一个排列，因为它以 `a` 开头，`a` 小于其余任意排列中的首元素。以 `b` 开头的那些排列出现在以 `c` 开头的那些之前。

对于任意给定排列而言，可以指出哪个排列出现在它之前以及哪个出现在它之后。给定排列