第 2 章

C++ 浏 览

本章将首先看一看 C++对数组类型的支持。数组是相同类型元素的集合，例如整型数组，可能代表考试的分数，或者字符串数组，可能代表在文本文件中包含的单词。然后，我们会看一看内置数组类型的缺点，以及怎样通过提供一个基于对象的 Array 类型的类来改善这些缺点。然后将其扩展成一个针对特化之后的 Array 子类型的面向对象层次结构。最后，我们还要比较一下 Array 类型与 C++标准库的 vector 类，并第一次了解泛型算法。沿着这条路，我们将进一步了解 C++对异常处理、模板和名字空间的支持。

2.1　内置数组数据类型

正如第 1 章所介绍的那样，C++为基本算术数据类型（如整数类型）提供了内置的支持。如：

```
// declares an integer object, ival
// initialized to a first value of 1024
int ival = 1024;
```

它也支持双精度和单精度浮点数据类型：

```
// declares a double precision floating point object, dval
// initialized to a first value of 3.14159
double dval = 3.14159;

// declares a single precision floating point object, fval
// also initialized to a first value of 3.14159
```

```
float fval = 3.14159;
```

C++也支持布尔类型以及用来存放字符集中单个元素的字符类型。

C++也为算术数据类型提供了赋值、一般算术运算以及关系运算的内置支持。算术运算如加、减、乘、除，关系运算如等于、不等于、小于、大于。例如：

```
int ival2 = ival + 4096;   // addition(加)
int ival3 = ival2 - ival;  // subtraction(减)

dval = fval * ival; // multiplication(乘)
ival = ival3 / 2;   // division(除)

bool result = ival2 == ival3;   // equality(等于)
result = ival2 + ival != ival3; // inequality(不等于)
result = fval + ival2 < dval;   // less-than(小于)
result = ival > ival2;          // greater-than(大于)
```

另外，标准库还支持基本类抽象的组合，例如字符串、复数。（在 2.7 节之前我们暂时不考虑标准库提供的 vector 类）。

在内置数据类型与标准库类的类型之间是*复合*类型(*compound* types)，特别是指针和数组类型（我们将在 2.2 节中介绍指针类型）。

数组(*array*)是一个顺序容器，它包含单一类型的元素。例如，序列：

```
0 1 1 2 3 5 8 13 21
```

代表菲波拉契数列的前 9 个数。（只要给出最前面两个元素，后面的元素依次可以由前面两个元素相加得出。）

为了定义和初始化一个数组以便存放这些数，我们可以这样写：

```
int fibon[ 9 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

数组对象的名字是 fibon，这是一个包含 9 个元素的整型一*维*(*dimension*)数组。第一个元素为 0，最后一个为 21，通过数组 *下标*(*subscript*)操作符我们可以以索引方式访问数组的元素。例如，为了读取数组的第一个元素，我们可能会这样写：

```
int first_elem = fibon[ 1 ]; // 不正确
```

不幸的是，这是不正确的，虽然它本身并没有语言错误。

在 C++中，数组下标从 0 开始，而不是 1。在位置 1 上的元素实际上是数组的第二个元素。类似地，位置 0 上的元素才是第一个元素，为了访问数组的最后一个元素，我们总是要索引到数组长度-1 的位置处的元素。

```
fibon[ 0 ]; // 第一个元素
fibon[ 1 ]; // 第二个元素
...
fibon[ 8 ]; // 最后一个元素
fibon[ 9 ]; // 喔！...
```

由 9 索引的元素不是数组的元素。fibon 的 9 个元素由下标 0~8 索引。初学者一个常见

的错误就是用位置 1~9 来索引。事实上，这非常普遍，以至于这个错误有了自己专用的名字：*一位偏移*（off-by-one）错误。

通常我们用循环来遍历数组中的元素。例如，下面的程序初始化一个包含 10 个元素的数组，其值分别从 0 到 9。然后在标准输出上以降序输出：

```
int main()
{
  int ia[ 10 ];
  int index;

  for ( index = 0; index < 10; ++index )
        // ia[0] = 0, ia[1] = 1, 等等
        ia[ index ] = index;

  for ( index = 9; index >= 0; --index )
  cout  << ia[ index ] << " ";

  cout << endl;
}
```

两个循环各迭代 10 次。关键字 for 后面的三条语句控制循环。第一条语句向 index 赋值 0，

```
index = 0;
```

它只在循环开始真正工作之前被执行一次。第二条语句

```
index < 10;
```

表示循环的*结束条件*(*stopping condition*)，它开始真正的循环序列。如果它的值为真，则与 for 循环相关联的语句(或一组语句)将被执行。如果它的值为假，则循环终止。在本例中，每次当 index 值小于 10 时，语句

```
ia[ index ] = index;
```

被执行。第三条语句

```
++index
```

是对一个算术对象加一的简短写法，它等价于

```
index = index + 1;
```

它在与 for 循环相关联的语句之后被执行，这里与 for 循环相关联的语句是“把 index 的值赋给以 index 为下标的元素”。这第三条语句的执行完成了 for 循环的一次迭代。序列的每次重复都重新测试条件，条件为假时，循环终止。（第 5 章将详细介绍 for 循环。）第二个循环以相反的顺序输出这些值。

虽然 C++对数组类型提供了内置支持，但是这种支持仅限于“用来读写单个元素”的机制。C++不支持数组的*抽象*(*abstraction*)，不支持对整个数组的操作，我们有时会希望对整个数组进行操作，例如，把一个数组赋值给另外一个数组、对两个数组进行相等比较，或者想知道数组的大小(size)。例如，给出两个数组，我们不能用赋值操作符把一个数组拷贝到

另一个中去：

```
int array0[ 10 ], array1[ 10 ];
...
// 错误：不能直接把一个数组赋值给另一个数组
array0 = array1;
```

如果我们希望把一个数组赋值给另外一个，那么我们必须自己写程序，按顺序拷贝每个元素：

```
for ( int index = 0; index < 10; ++index )
    array0[ index ] = array1[ index ];
```

而且，数组类型本身没有自我意识，它不知道自己的长度，我们必须另外记录数组本身的这些信息。当我们希望把数组作为一个参数传递给一个函数的时候，问题就出现了。在 C++中，数组不同于整数类型和浮点数类型，它不是 C++语言的一等(first-class)公民。数组是从 C 语言中继承来的，它反映了数据与对其进行操作的算法的分离，而这正是过程化程序设计的特征。在本章后面部分，我们将会了解一些不同的策略，通过这些策略使数组具有一些额外的居民特权。

练习 2.1

为什么内置数组类型不支持数组之间的赋值？支持这种操作需要什么信息?

练习 2.2

第一类数组(a first class array)应该支持什么操作？

2.2　动态内存分配和指针

在开始基于对象的设计之前，我们需要暂时偏离一下主题，先来介绍一下 C++程序内存分配的问题。原因是，我们必须首先介绍在程序执行期间怎样申请和访问内存，否则，没有办法真正实现我们的设计(并且展示理想的 C++代码)，这是本小节的目的。

在 C++中，对象可以被静态分配——即编译器在处理程序源代码时分配，或者动态分配——即程序执行时调用运行时刻库函数来分配。这两种内存分配方法的主要区别是效率与灵活性之间的平衡准则不同。由于静态内存分配是在程序执行之前进行的，因而效率比较高。但是，它缺少灵活性，它要求在程序执行之前就知道所需内存的类型和数量。例如，利用静态分配的字符串数组，我们就不可能很容易地处理和存储任意的文本文件。一般来说，存储未知数目的元素需要动态内存分配的灵活性。

到目前为止，所有的内存分配都是静态的。例如，定义

```
int ival = 1024;
```

指示编译器分配足够的存储区存放一个整型值。该存储区与名字 ival 相关联，然后，用数值 1024 初始化该存储区。这些工作都在程序执行之前完成。

有两个值与对象 ival 相关联：一个是它包含的值——本例中为 1024，另一个是存放这个值的存储区的地址。在 C++中，这两个值都可以被访问。当我们写出下面的代码时

```
int ival2 = ival + 1;
```

我们访问 ival 所包含的值，把它加 1，然后用该新值初始化 ival2。在本例中，ival2 有初始值 1025。怎样访问和存储内存地址呢？

C++支持指针类型来存放对象的内存地址值。例如，为了声明一个能存放 ival 内存地址的指针类型，我们这样写：

```
// 一个指向int类型的指针
int *pint;
```

C++预定义一个专门的*取地址*(*address-of*)操作符（&）。当我们把它应用在一个对象上时，它返回对象的地址值。因此，为了将 ival 内存地址值赋给 pint，我们可以这样写

```
int *pint;
pint = &ival; // 把ival的地址赋给pint
```

为了访问 pint 所指向的实际对象，我们必须首先用*解引用*(*dereference*)操作符（*）来*解除 pint 的引用*(*dereference pint*)。例如，下面我们通过 pint 间接地给 ival 加 1：

```
// 通过pint间接地给ival加1
*pint = *pint + 1;
```

它等价于下面直接对 ival 操作的语句：

```
// 直接给ival加1
ival = ival + 1;
```

本例中，使用指针间接地操作 ival 没有什么实际的好处：这样做比直接操作 ival 的效率低，而且又容易出错。我们只是用它来简单地介绍一下指针。在 C++中，指针的一个主要用处是管理和操纵动态分配的内存。

静态与动态内存分配的两个主要区别是：

1．静态对象是有名字的变量，我们直接对其进行操作。而动态对象是没有名字的变量，我们通过指针间接地对它进行操作。稍后我们会看到一个例子。

2．静态对象的分配与释放由编译器自动处理。程序员需要理解这一点，但不需要做任何事情。相反，动态对象的分配与释放，必须由程序员显式地管理，相对来说比较容易出错，它通过 new 和 delete 两个表达式来完成。

对象的动态分配可通过 new 表达式的两个版本之一来完成。第一个版本分配特定类型的单个对象。例如，

```
int *pint = new int( 1024 );
```

分配了一个没有名字的 int 类型的对象，对象初始值为 1024。然后，表达式返回对象在内存中的地址。接着，这个地址被用来初始化指针对象 pint。对于动态分配的内存，唯一的访问方式是通过指针间接地访问。

new 表达式的第二个版本，分配一个特定类型和维数的数组。例如

```
int *pia = new int[ 4 ];
```

分配了一个含有四个整数元素的数组。不幸的是，我们没有办法给动态分配的数组的每个元素显式地指定一个初始值。

分配动态数组时一个常令人迷惑的问题是，返回值只是一个指针，与分配单一动态对象的返回类型相同。例如，pint 与 pia 的不同之处在于，pia 拥有四元素数组的第一个元素的地址，而 pint 只是简单地包含单一对象的地址。当用完了动态分配的对象或对象的数组时，我们必须显式地释放这些内存。我们通过使用 *delete 表达式*(*delete expression*)的两个版本之一可以完成这件事情。释放之后的内存则可以被程序重新使用。单一对象的 delete 表达式形式如下：

```
// 删除单个对象
delete pint;
```

数组形式的 delete 表达式如下：

```
// 删除一个对象数组
delete [] pia;
```

如果忘了删除动态分配的内存，会怎么样呢？那么，程序结束时将会有*内存泄漏*(*memory leak*)。内存泄漏是指一块动态分配的内存，我们不再拥有指向这块内存的指针，因此我们没有办法将它返还给程序以便以后重新使用。（现在大多数系统提供识别内存泄漏的工具，可以向系统管理员咨询。）

对指针类型和动态内存分配的快速浏览可能会引发很多应该回答的问题。但是，动态内存分配和指针操作是 C++实际编程基础的一个方面。我们不想推迟到后面再介绍它。在本章接下去的介绍中，我们将在基于对象的与面向对象的 Array 类的实现中，看到它的用法。8.4 节将详细介绍动态内存分配以及 new 与 delete 表达式的用法。

练习 2.3

说出下面定义的四个对象之间的区别。

```
(a) int ival = 1024;     (c) int *pi2 = new int( 1024 );
(b) int *pi = &ival;     (d) int *pi3 = new int[ 1024 ];
```

练习 2.4

下面的代码段是做什么的？有什么严重错误？（注意指针 pia 的下标操作符的用法是正确的。在 3.9.2 节中我们会解释其理由。）

```
int *pi = new int( 10 );
int *pia = new int[ 10 ];

while ( *pi < 10 ) {
      pia[ *pi ] = *pi;
      *pi = *pi + 1;
}

delete pi;
```

```
delete [] pia;
```

2.3 基于对象的设计

在本节中，我们将使用 C++的类机制设计和实现一个数组抽象。我们最初的实现只支持一个整型数组。以后，我们将用模板机制对这种抽象进行扩展，使其能够支持无限数目的数据类型。

第一步，我们需要决定数组应该提供哪些操作。尽管我们希望能提供所有的操作，但是我们不能一次提供所有的功能。下面是第一步所支持操作的集合：

1．数组类的实现中有内置的自我意识。首先，它知道自己的大小。

2．数组类支持数组之间的赋值、以及两个数组之间的相等和不相等的比较操作。

3．数组类应该支持对其内含的值的下列查询操作：数组中最小值是什么？最大值是什么？某个特殊的值是否在数组中？如果存在，它的第一个位置的索引是什么？

4．数组类支持自排序。为了便于讨论，假定存在一群用户，他们认为数组支持排序的功能很重要。而另外一些人对此没有强烈的看法。

除了支持数组操作，还必须支持数组本身的机制，包括：

5．能够指定长度，以此来创建数组（这个值无需在编译时刻知道）。

6．能够用一组值初始化数组。

7．能够通过一个索引来访问数组中的单个元素。为便于讨论，假设用户强烈要求用数组下标操作符来实现这项功能。

8．能够截获并指出错误的索引值。假设我们认为这很有必要，所以没有询问用户的想法。我们认为这是一个设计良好的数组所必须实现的。

我们与潜在用户的讨论已经引起了极大的热情。现在我们要真正实现它。但是怎样把这个设计转换成 C++代码呢？支持基于对象设计的类的一般形式如下：

```
class classname {
public:
  // the public set of operations
private:
  // the private implementation
};
```

这里，class、public 和 private 是 C++语言的关键字，classname 是用户定义的标识符，它用来命名这个类，以便后面引用该类。我们将前面设计的类命名为 IntArray，等到我们使它支持的数据类型更广泛时，我们将它改名为 Array。

一个类名代表一个新的数据类型，我们可以用它来定义这种类的类型的对象，就像用内置的类型定义对象一样。例如：

```
// 单个IntArray类对象
IntArray myArray;
```

```
// 指向IntArray类对象的指针
IntArray *pArray = new IntArray;
```

类定义包括两个部分：*类头*(*class head*)，由关键字 calss 与相关联的类名构成；*类体*(*class body*)，由花括号括起来，以分号结束。类头本身也用作类的声明。例如：

```
// 在程序中声明IntArray类，但是不提供定义
class IntArray;
```

类体包含成员定义，以及访问标签，如 public 和 private。类的成员包括“该类能执行的操作”和“代表类抽象所必需的数据”。这些操作称为*成员函数*(*member function*)或*方法*(*method*)。对于 IntArray 类来说，它由以下的内容构成：

```
class IntArray {
public:
  // equality and inequality operations: #2b
  bool operator==( const IntArray& ) const;
  bool operator!=( const IntArray& ) const;

  // assignment operator: #2a
  IntArray& operator=( const IntArray& );

  int  size() const; // #1
  void sort();       // #4

  int  min()  const; // #3a
  int  max()  const; // #3b

  // if the value is found within the array,
  // return the index of its first occurrence
  // otherwise, return -1
  int find( int value ) const; // #3c

private:
  // the private implementation
};
```

成员函数右边的数字指的是在前面我们定义的规范表中的条目。我们现在不打算解释参数表中的 const 修饰符或参数表后面的 const 修饰符，现在还没必要如此详细地解释。但是在实际的程序中，它们还是必需的。

通过两个*成员访问操作符*(*member access operator)* 其中之一，我们可以调用一个有名字的成员函数，如 min()。这两个操作符为：对于类对象的点操作符（.），以及对于类对象指针的箭头操作符（->）。例如，为了在数组 myArray 类对象中找到最小值，我们可以这样写

```
// 用myArray数组中的最小元素来初始化min_val
int min_val = myArray.min();
```

为了在动态分配的 IntArray 对象中查找最大值，我们可以这样写

```
int max_val = pArray->max();
```

（是的，我们还没介绍怎样用一个长度与一组值来初始化 IntArray 类对象。有个特殊的被称为*构造函数*(*constructor*)的成员函数可以完成这些工作。我们将会简要地介绍它。）

把操作符应用在这些类对象上的方式与应用在内置数据类型上的对象一样。给出两个 IntArray 对象：

```
IntArray myArray0, myArray1;
```

赋值操作符可以这样应用：

```
// 调用拷贝赋值成员函数：
// myArray0.operator=( myArray1 )
myArray0 = myArray1;
```

等于操作符的调用如下所示：

```
// 调用等于成员函数：
// myArray0.operator==( myArray1 )
if ( myArray0 == myArray1 )
    cout << "!!our assignment operator works!\n";
```

关键字 private 和 public 控制对类成员的访问。出现在类体中公有（public）部分的成员，在一般程序的任何地方都可以访问它们；出现在私有（private）部分的成员只能在该类的成员函数或友元（friend）中被访问。（我们要到 15.2 节才会解释友元。）

一般来说，公有成员提供了该类的*公有接口*(*public interface*)——即实现了这个类的行为的操作集合。它包括该类的所有成员函数，或者只包括其中一个子集。私有成员提供*私有实现*(*private implementation*)——即存储信息的数据。

这种“类的公共接口与私有实现的分离”，被称为*信息隐藏*(*information hiding*)。信息隐藏是软件工程中一个很重要的概念，在后面章节中有详细的介绍。简要说来，它为程序提供了两个主要好处：

1．如果类的私有实现需要修改或扩展，那么，只有相对很小一部分“要求访问这些实现的成员函数”才需要被修改。许多使用该类的用户程序一般无需修改，但是要求重新编译。（6.18 节将演示这个过程。）

2．如果类的私有实现有错误，那么通常需要检查的代码数量只局限在相对较少的需要访问这些实现的成员函数上，而无需检查整个程序。

哪些数据成员是 IntArray 必需的呢？当声明一个 IntArray 对象时，用户会指定数组大小，我们需要存储它。因此，我们将定义一个数据成员来做到这一点。另外，我们需要实际分配并存储底层的数组，这将通过 new 表达式来实现，我们将定义一个指针数据成员来存储 new 表达式返回的地址值：

```
class IntArray {
public:
```

```
    // ...
    int size() const { return _size; }
private:
    //  内部数据
    int _size;
    int *ia;
};
```

由于我们把 _size 放在类的私有区内，因此我们有必要声明一个公有成员函数以便允许用户访问它的值。由于 C++不允许成员函数与数据成员共享同一个名字，所以在这样的情况下，一般习惯是在数据成员名字前面加一个下划线（-）。因此，我们有了公有访问函数 size() 和私有数据成员 _size。在本书以前的版本中，我们在访问函数前加上 *get* 或 *set*。实践证明这样做有些累赘。

尽管这种公有访问函数的用法允许用户读取到相应的值,但是这种实现似乎还有些根本的错误。至少第一眼看上去是这样的。你看出来了吗？考虑下面的语句：

```
IntArray array;
int array_size = array.size();
```

和

```
// 假设_size是public的
int array_size = array._size;
```

将 array_size 初始化为数组的维数。但是，很显然第一个例子需要一个函数调用，而第二个只需直接访问内存就行了。一般来说，函数调用比直接访问内存的开销要大得多。因而信息隐藏是否给程序的执行效率增加了严重的额外负担，或许是阻碍性的负担呢？幸运的是，在一般情况下，回答是：不。

C++提供的解决方案是***内联函数***(*inline function*)机制。内联函数在它的调用点上被展开。一般来说，内联函数不会引入任何函数调用。[1] 例如，在 for 循环的条件子句中的 size()调用：

```
for ( int index = 0; index < array.size(); ++index )
    // ...
```

并没有真的被调用_size 次，而是在编译期间被内联扩展为下面的一般形式：

```
// array.size()的内联扩展
for ( int index = 0; index < array._size; ++index )
    // ...
```

在类定义中被定义的成员函数（如 size()）会自动被当作是内联函数。此外，我们也可以用 inline 关键字显式地要求一个函数被视为内联函数。（7.6 节中有更多关于内联函数的说明。）

[1] 然而，实际并不总是这样的。对于编译器来说，内联函数是一种请求，而不是一种保证。参见 7.6 节的讨论。

到目前为止，我们已经提供了 IntArray 类所要求的操作（前面的 1~4 项），但是还没有提供初始化机制和数组中单个元素的访问方式（5~8 项）。

程序设计中一个常见的错误是使用事先并没有被正确初始化的对象。实际上，这是一个太过于常见的错误，所以 C++为用户定义的类提供了一种自动初始化机制：类 *构造函数*(class *constructor*)。

构造函数是一种特殊的类成员函数，专门用于初始化对象。如果构造函数被定义了，那么在类的每个对象第一次被使用之前，该构造函数被自动应用在对象上。构造函数由谁来定义呢？类的提供者——也就是我们来定义构造函数。为一个类确定必要的构造函数是程序设计不可缺少的一部分。

为了定义一个构造函数，我们只要给它与类相同的名字即可。另外，我们不能给构造函数指定返回值。但是可以给类定义多个构造函数，尽管它们都具有相同的名字，但只要编译器能够根据参数表区分它们就行。

更一般地，C++支持被称为*函数重载*(*function overloading*)的机制。函数重载允许两个或更多个函数使用同一个名字，限制条件是它们的参数表必须不同：参数类型不同，或参数的数目不同。根据不同的参数表，编译器就能够判断出，对某个特定的调用应该选择哪一个版本的重载函数。下面是一组合法的 min()重载函数（这些函数也可以是类成员函数）：

```
// 一组min()重载函数
// 每个函数都有一个特有的参数表
#include <string>;

int    min( const int *pia, int size );
int    min( int, int );
int    min( const char *str );
char   min( string );
string min( string, string );
```

重载函数在运行时刻的行为与非重载函数完全一样，主要的负担是在编译时刻用来决定应该调用哪个实例所需要的时间。如果 C++不提供函数重载支持，那么我们必须为程序中每个函数都要提供一个独一无二的名字。(第 9 章将详细讨论函数重载的内容。)

我们为 IntArray 类指定了下面三个构造函数：

```
class IntArray {
public:
  explicit IntArray( int sz = DefaultArraySize );
  IntArray( int *array, int array_size  );
  IntArray( const IntArray &rhs );
  // ...

private:
  static const int DefaultArraySize = 12;
//  ...
};
```

构造函数

```
IntArray( int sz = DefaultArraySize );
```

被称为*缺省构造函数*(*default constructor*)**。因为它不需要用户提供任何参数。（我们现在不打算解释这个缺省构造函数声明中出现的关键字** explict**，之所以显示它是为了完整性。）如果程序员提供参数，该值将被传递给构造函数。例如，**

```
IntArray array1( 1024 );
```

将参数 1024 **传递给构造函数。另一方面，如果用户不指定长度，那么构造函数将使用** DefaultArraySize **的值。例如，**

```
IntArray array2;
```

将导致用 DefaultArraySize **的值来调用构造函数。（被声明为** static **的数据成员是一类特殊的共享数据成员，无论这个类的对象被定义了多少个，静态数据成员在程序中只有一份。这是在类的所有对象之间共享数据的一种方式。**13.5 **节有全面的讨论。）**

下面是缺省构造函数的一个简化实现版本，简化到没有考虑出错的可能性。（可能出现什么错误呢？本例中有两个：首先，提供给程序的动态内存不是无限的，因此，new **表达式有可能失败，**2.6 **节中将介绍如何处理这种情况。第二，传递给参数** sz **的值可能是无效的，例如，负数或** 0**，或者太大的值以至于无法存储在** int **类型的变量中。）**

```
IntArray::
IntArray( int sz )
{
  // 设置数据成员
  _size = sz;
  ia = new int[ _size ];

  // 初始化内存
  for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = 0;
}
```

这是我们第一次在类体的外面定义类的成员函数。唯一的语法区别是要指出成员函数属于哪个类，这可以通过*类域操作符*(*class scope operator*)**来实现：**

```
IntArray::
```

双冒号（::**）操作符被称为*域操作符***(*scope operator*)**。当与一个类名相连的时候，像上面例子中那样，它就成为一个类域操作符。我们可以非正式地把域看作是一个可视窗口。全局域的对象在它被定义的整个文件里(一直到文件末尾)都是可见的。在一个函数内被定义的对象是局域的(local scope)，它只在定义它的函数体内可见。每个类维持一个域，在这个域之外，它的成员是不可见的。类域操作符告诉编译器，后面的标识符可在该类的范围内被找到。本例中，**

```
IntArray::
IntArray( int sz )
```

告诉编译器 IntArray()函数被定义为 IntArray 类的成员。（尽管程序域不是我们现在应该关注的事情，但是最终我们还是要理解域的概念。在第 8 章中我们将详细讲解程序域，而类域将在 13.9 节中特别讨论。）

IntArray 类的第二个构造函数用内置的整数数组初始化一个新的 IntArray 类对象。它需要两个参数：一个是实际的数组；另一个参数指明数组的大小。例如：

```
int ia[10] = {0,1,2,3,4,5,6,7,8,9};
IntArray iA3(ia,10);
```

这个构造函数的实现几乎与第一个构造函数相同。（我们又一次没有保护我们的代码会出错。）

```
IntArray::
IntArray( int *array, int sz )
{
 // 设置数据成员
 _size = sz;
 ia = new int[ _size ];

 // 拷贝数据成员
 for ( int ix=0; ix < _size; ++ix )
       ia[ ix ] = array[ ix ];
}
```

最后一个 IntArray 构造函数用另外一个 IntArray 对象来初始化当前的 IntArray 对象，对于下面两种形式，无论是哪一种，它都将被自动调用：

```
IntArray array;

// 等价的初始化方式
IntArray ia1 = array;
IntArray ia2( array );
```

这种构造函数被称为类的*拷贝构造函数*(*copy constructor*)。在后面的章节中我们将会看到更多的示例。下面的实现再次忽略可能出现的运行时刻程序异常：

```
IntArray::
IntArray( const IntArray &rhs )
{ // 拷贝构造函数
 _size = rhs._size;
 ia = new int[ _size ];

 for ( int ix=0; ix < _size; ++ix )
       ia[ ix ] = rhs.ia[ ix ];
}
```

本例引入了一种新的复合类型：*引用*（*reference*），即 IntArray &rhs。引用是一种没有指针语法的指针。（因此，我们写成 rhs._size，而不是 rhs->_size。）与指针一样，引用提供对对象的间接访问。（3.6 节中我们将对指针和引用作更多的介绍。）

注意到所有这三个构造函数都是以相似的方式来实现的。一般来说，当两个或多个函数重复相同的代码时，一般将这部分代码抽取出来，形成独立的函数，以便共享。以后，如果需要改变这份实现，则只需改变一次。而且，这种实现的共享本质更容易为大家所理解。

怎么样把构造函数中的代码抽取出来形成独立的共享函数呢？下面是一种可能的实现：

```
class IntArray {
public:
  // ...
private:
  void init( int sz, int *array );
  // ...
};

void
IntArray::
init( int sz, int *array )
{
  _size = sz;
  ia = new int[ _size ];

  for ( int ix=0; ix < _size; ++ix )
        if ( ! array )
           ia[ ix ] = 0;
        else ia[ ix ] = array[ ix ];
}
```

三个构造函数可被重写为：

```
IntArray::IntArray( int sz ){ init( sz, 0 ); }
IntArray::IntArray( int *array, int sz )
    { init( sz, array ); }

IntArray::IntArray( const IntArray &rhs )
    { init( rhs._size, rhs.ia ); }
```

类机制也支持一个特殊的*析构*成员函数(*destructor* member function)。每个类对象在被程序最后一次使用之后，它的析构函数就会被自动调用。我们通过在类的名字前面加一个波浪线（~）来标识析构函数。一般地，析构函数会释放在类对象被使用和构造过程中所获得的资源。例如，在 IntArray 的析构函数中，它会删除构造时分配的内存。（我们将在第 14 章详细讨论构造函数和析构函数。）下面是我们的实现：

```
class IntArray {
public:
  // 构造函数
```

```
    explicit IntArray( int size = DefaultArraySize );
    IntArray( int *array, int array_size  );
    IntArray( const IntArray &rhs );

    // 析构函数
    ~IntArray() { delete [] ia; }

    // ...

  private:
    //  ...
  };
```

除非用户能够很容易地通过索引来访问单个元素，否则数组类就没有更实际的用处。例如，我们的类需要支持下面的一般用法：

```
IntArray array;
int last_pos = array.size()-1;

int temp = array[ 0 ];
array[ 0 ] = array[ last_pos ];
array[ last_pos ] = temp;
```

我们通过提供“专用于一个类的下标操作符实例”，来支持索引 IntArray 类对象。下面是支持这种用法的一个实现：

```
#include <cassert>
int&
IntArray::
operator[]( int index )
{
  assert( index >= 0 && index < size );
    return ia[ index ];
}
```

更为一般化，C++语言支持*操作符重载*(*operator overloading*)，这样就可以为特定的类(类型)定义新的操作符实例。典型地，类提供一个或多个赋值操作符、等于操作符，可能还有一个或多个关系操作符，以及 iostream 输入和输出操作符。（3.15 节有关于操作符重载的更进一步的说明。在第 15 章我们将详细讨论操作符重载。）

类定义以及相关的常数值或 typedef 名通常都存储在头文件中。并且头文件以类名来命名。因此，例如我们将会创建一对头文件：IntArray.h 和 Matrix.h。所有希望使用 IntArray 类或 Matrix 类的程序都必须包含相关的头文件。

类似地，不在类定义内部定义的类成员函数都存储在与类名同名的程序文本文件中。例如，我们将创建一对程序文本文件：IntArray.C 和 Matrix.C，用来存储相关类的成员函数。

（记住，程序文本文件的后缀因编译系统而不同，你应该检查自己的系统所使用的命名习惯。）这些函数不用随每个使用相关类的程序而重新编译，这些成员函数经过预编译之后被保存在类库中。iostream 库就是这样一个例子。

练习 2.5

C++类的关键特征是接口与实现的分离。接口是一些“用户可以应用到类对象上的操作”的集合。它由三部分构成：这些操作的名字、它们的返回值，以及它们的参数表。一般地，这些就是该类用户所需要知道的全部内容。私有实现包括为支持公有接口所必需的算法和数据。理想情况下，即使类的接口增长了，它也不用变得与以前的版本不相兼容。另一方面，在类的生命周期内其实现可以自由演化。从下面选择一个抽象(指类)，并为该类编写一个公有接口。

```
(a) Matrix    (c) Person    (e) Pointer
(b) Boolean   (d) Date      (f) Point
```

练习 2.6

构造函数和析构函数是程序员提供的函数，它们既不构造也不销毁一个类的对象（编译器自动把它们作用到这些对象上）。因此构造函数(constructor)和析构函数(destructor)这两个词多少有些误导，当我们写：

```
int main() {
  IntArray myArray( 1024 );
  // ...
  return 0;
}
```

在构造函数被应用之前，用于维护 myArray 中数据成员的内存已经被分配了。实际上，编译器在内部把程序转换成如下的代码（注意这不是合法的 C++代码 [2]）：

```
int main() {
  IntArray myArray;

  // 伪C++代码 – 应用构造函数
  myArray.IntArray::IntArray( 1024 );
  // ...
  // 伪C++代码 – 应用析构函数
  myArray.IntArray::~IntArray();

  return 0;
}
```

一个类的构造函数主要用来初始化类对象的数据成员。析构函数主要释放类对象在生命

[2] 对于感兴趣的读者，可以参见本书的配套书“Inside the C++ Object Model”，该书讨论了这方面的内容。

期内申请到的所有资源。请定义在练习 2.5 中选择的类所需要的构造函数集。你的类需要析构函数吗？

练习 2.7

在练习 2.5 和练习 2.6 中，你差不多已经定义了使用该类的一个完整的公有接口。（我们还需要定义一个拷贝赋值操作符，但是现在我们忽略这个事实——C++为“从一个类对象向另一个类对象赋值”提供了缺省支持。问题在于，缺省的行为常常是不够的。这将在 14.6 节中讨论。）写一个程序来实践前面两个练习中定义的公有接口。用起来容易还是麻烦？你希望重写这些定义吗？你能在重写的同时保持兼容性吗？

2.4　面向对象的设计

max()与 min()函数的实现没有对数组元素的存储作特殊的假设，因此，我们需要检查数组的每个元素。如果我们要求所有的元素已经被排序，则这两个操作就变得非常简单，只要索引第一个元素和最后一个元素即可。而且，如果已知元素已经排序，那么查找一个元素的存在就会更加高效。但是，对数组进行排序增加了 Array 类实现的复杂性。我们的设计出错了吗？

实际上，我们现在是否犯了错误要相对于我们做出的选择而言。排序的数组是一种特殊的实现：需要时，它完全必要；否则，支持排序数组的额外开销就是一项负担。我们的实现是比较通用的，在大多数情况下是足够的。它支持更广阔范围内的用户。不幸的是，如果用户绝对需要排序数组的行为，那么我们的实现就不能提供支持。对用户来说，他没有办法改写 min()、max()、find()这些函数比较通用的实现。实际上，我们已经选择了一个一般用途的实现，它不适合特殊的环境。

另一方面，我们的实现对另外一类用户而言又针对性太强了：对索引的范围检查为每次访问元素增加了额外的负担。在设计中，我们没有考虑这样的开销(2.3 节中第 8 条)，而是假设：如果结果不正确，那么速度再快也没有价值。但是，这种设计至少对于某一类主要用户：实时虚拟和虚拟现实提供者，就不成立。在这种情况下，数组代表复杂 3D 几何图形的顶点。场景飞快地变化，以至于一些偶然的错误不会被看到。但如果访问速度太慢了，那么实时效果就会被打破。我们实现的数组类虽然比没有范围检查的数组类会更安全，但是在这样的实时应用领域却不够实际。

我们怎样才能支持这三种用户的需要呢？解决的方案已经多多少少体现在代码中了。例如，范围检查局限在下标操作符中。去掉 check_range()的调用，重新命名该数组。现在我们就有了两种实现：一个有范围检查，一个没有范围检查。进一步拷贝一份代码，并把它修改成针对已排序的数组。现在我们就有了对已排序数组的支持：

```
// 未排序，也没有边界检查
class IntArray{ ... };

// 未排序，但支持边界检查
class IntArrayRC{ ... };
```

```
// 已排序，但没有边界检查
class IntSortedArray{ ... };
```

这种方案的缺点是什么呢？

1．我们必须维护三个包含大量重复代码的数组实现。我们更希望把这些公共代码只保留一份，然后由“这三个数组类（以及其他一些我们以后会选择支持的数组类）”共享。（比如，可能会是一个带有边界检查的排序数组。）

2．由于三个数组实现是完全独立的类型，所以我们必须编写独立的函数来操作它们，尽管函数内的一般性操作都是相同的。例如：

```
void process_array( IntArray& );
void process_array( IntArrayRC& );
void process_array( IntSortedArray& );
```

我们希望只编写一个函数，它不但能接受现有的数组类，而且，还能够接受任意将来的数组类，只要这同样的操作集合也能够应用到这些类上。

面向对象的程序设计方法正是为我们提供了这样一种能力。上面第 1 项可由*继承*(*inheritance*)机制提供。当一个 IntArrayRC 类（也就是一个带有范围检查的 IntArray 类）继承了 IntArray 类时，它可以访问 IntArray 的数据成员和成员函数，而不要求我们维护两份代码拷贝。新的类只需提供实现其额外语义所必需的数据成员和成员函数。

在 C++中，被继承的类，如本例中的 IntArray，被称作*基类*(*base class*)。新类被称作从基类*派生*(*derived*)而来，我们把它叫做基类的*派生类*(*derived class*)或*子类型*(*subtype*)。我们说 IntArrayRC 是一种有特殊行为的 IntArray，它支持对索引值的范围检查。子类型与基类共享*公共的接口*(*common interface*)——公有操作的公共集。由于共享公共接口，这就允许子类和基类在程序内部可互换使用，而无需考虑对象的实际类型。从某种意义上来说，公共接口封装了单个子类型中与类型相关的细节。类之间的类型/子类型关系形成了*继承*或*派生层次关系*(*inheritance or derivation hierarchy*)。例如，下面的非成员函数 swap()把指向基类 IntArray 对象的引用作为第一个参数。该函数交换索引 i 和 j 处的元素。

```
#include <IntArray.h>
void swap( IntArray &ia, int i, int j )
{
  int tmp = ia[ i ];
  ia[ i ] = ia[ j ];
  ia[ j ] = tmp;
}
```

下面是 swap()函数的三个合法调用：

```
IntArray       ia;
IntArrayRC       iarc;
IntSortedArray ias;
// ok: ia是一个IntArray
swap( ia, 0, 10 );
```

```
// ok: iarc是IntArray的子类型
swap( iarc, 0, 10 );

// ok: ias也是IntArray的子类型
swap( ias, 0, 10 );

// error: string不是IntArray的子类型
string str( "not an IntArray!" );
swap( str, 0, 10 );
```

三个数组类都提供了自己的下标操作符实现。当然，我们的要求是，当

```
swap( iarc, 0, 10 );
```

被调用时，IntArrayRC 的下标操作符被调用；当

```
swap( ias, 0, 10 );
```

被调用时，IntSortedArray 下标操作符被调用，等等。swap()调用的下标操作符必须潜在地随着每次调用而改变，它必须由被交换元素的数组的实际类型来决定。在 C++中，这可以由***虚拟函数***(*virtual function*)机制来自动完成。

为使 IntArray 类能够被继承，在语法上我们只需要做一点小小的改变：我们必须（可选择的）减少封装的层次，以便允许派生类访问非公有的实现，而且我们也必须显式地指明哪些函数应该是虚拟的。最重要的变化在于我们如何把一个类设计成为基类。

在基于对象的程序设计中，通常类的提供者只有一个，但是类的用户有许多个。提供者设计并且通常也会实现类。用户使用提供者提供的公有接口，行为的分离可通过将类分成公有与私有访问级别而反映出来。

在继承机制下有多个类的提供者：一个提供基类实现（可能还有一些派生类），另外一个或多个提供者在继承层次的生命周期内提供派生类。这种行为也是一种实现行为；子类的提供者经常（但并不总是）需要访问基类的实现。为了提供这种能力，同时还要防止对基类实现的一般性访问，C++提供了另外一个访问级别：保护级别（protected）。在类的保护区域内的数据成员和成员函数，不提供给一般的程序，只提供给派生类。（放在基类的私有区域内的成员只能供该类自己使用，派生类不能使用。）下面是修改过的 IntArray 类：

```
class IntArray {
public:
  // 构造函数
  explicit IntArray( int size = DefaultArraySize );
  IntArray( int *array, int array_size  );
  IntArray( const IntArray &rhs );

  // 虚拟析构函数!
  virtual ~IntArray() { delete [] ia; }

  // 等于和不等于操作:
  bool operator==( const IntArray& ) const;
  bool operator!=( const IntArray& ) const;
```

```
  IntArray& operator=( const IntArray& );
  int size() const { return _size; }

  // 去掉了索引检查功能 ...
  virtual int& operator[](int index) { return ia[index]; }
  virtual void sort();

  virtual int min() const;
  virtual int max() const;
  virtual int find( int value ) const;

protected:
  // 参见13.5节的说明
  static const int DefaultArraySize = 12;
  void init( int sz, int *array );

  int _size;
  int *ia;
};
```

在面向对象与基于对象的设计中，指明一个类的成员是 public **的准则没有变化。重新设计的** IntArray **类将被用作基类，它仍然把构造函数、析构函数、下标操作符、**min()**、**max() **等等声明为公有成员。这些成员继续提供公有接口，但现在接口不只为** IntArray **类服务，同时也为从它派生的整个继承层次服务。**

非公有的成员到底该声明为 protected **还是** private **类成员是新的设计准则。如果希望防止派生类直接访问某个成员，我们就把该成员声明为基类的** private **成员。如果确信某个成员提供了派生类需要直接访问的操作或数据存储，而且通过这个成员，派生类的实现会更有效，则我们把该成员声明为** protected**。对于** IntArray **类，我们已经将全部数据成员设置成** protected**，也就是实际上允许后续派生的类访问** IntArray **的实现细节。**

为了把一个类设计成基类，要做的第二个设计考虑是找出类型相关的成员函数，并把这些成员函数标记为 *virtual***（虚拟的）。**

对于类型相关的成员函数，它的算法由特定的基类或派生类的行为或实现来决定。例如，对每种数组类型，下标操作符的实现是不同的，所以，我们将它声明为 virtual**。**

等于、不等于操作符和 size()**成员函数的实现对于其应用的数组类型来说是独立的，因此，不把它声明成** virtual**。**

对于一个非虚拟函数的调用，编译器在编译时刻选择被调用的函数。而虚拟函数调用的决定则要等到运行时刻。在执行程序内部的每个调用点上，系统根据被调用对象的实际基类或派生类的类型来决定选择哪一个虚拟函数实例。例如，考虑下面的代码：

```
void init( IntArray &ia )
{
  for ( int ix = 0; ix < ia.size(); ++ix )
        ia[ ix ] = ix;
}
```

形式参数 ia 可以引用 IntSortedArray、IntArrayRC 或 IntArray 类的对象（我们将简要介绍这里的派生类）。函数 size()作为非虚拟函数，由编译器处理并内联展开。但是，下标操作符要直到执行循环的每次迭代时才能被处理，因为在编译期间编译器不知道数组 ia 指向的实际类型。

（第 17 章将详细讨论虚拟函数，包括虚拟析构函数的主题，以及使用虚拟函数设计带来的效率问题。[LIPPMAN96a]对虚拟函数的实现与效率有更深入的讨论。）

一旦我们定好了设计方案，C++的实现就很容易了。例如，下面这个完整的 IntArrayRC 派生类定义，被放在一个独立的头文件 IntArrayRC.h 中，该文件包含头文件 IntArray.h，而 IntArray.h 包含有 IntArray 类的定义。

```
#ifndef IntArrayRC_H
#define IntArrayRC_H

#include "IntArray.h"

class IntArrayRC : public IntArray {
public:
   IntArrayRC( int sz = DefaultArraySize );
   IntArrayRC( int *array, int array_size );
   IntArrayRC( const IntArrayRC &rhs );

   virtual int& operator[]( int );

private:
   void check_range( int );
};

#endif
```

IntArrayRC 只需定义不同于 IntArray 实现的哪些方面，或者加上对 IntArray 扩展的实现。

1．它必须提供自己的下标操作符实例，以支持范围检查。

2．它必须提供一个操作来做实际的检查工作。（由于它不是公有接口的一部分，所以我们把它声明为 private。）

3．它必须提供一组自动初始化函数，即自己的构造函数集。

IntArray 的成员函数与数据成员对于 IntArrayRC 来说都是可用的，就如同 IntArrayRC 已经显式地定义了它们一样。这正是下面这句话的含义：

```
class IntArrayRC : public IntArray
```

冒号定义了 IntArrayRC 是从 IntArray 派生而来的。（关键字 public 表明派生类共享基类的公有接口。IntArrayRC 类型的对象可以用在任何“可以使用基类类型对象”的位置上，比如在 swap()例子中。第 18 章会详细解释这一点。）IntArrayRC 可以看作是 IntArray 的扩展，它增加了下标范围检查的额外特性。下面是下标操作符的一个实现：

```
inline int&
IntArrayRC::operator[]( int index )
{
  check_range( index );
  return ia[ index ];
}
```

这里，check_range()被实现为一个内联成员函数。它调用 assert()宏，（关于 assert()宏的讨论见 1.3 节。）

```
#include <cassert>

inline void
IntArrayRC::check_range( int index )
{
  assert( index >= 0 && index < size );
}
```

（我们把 check_range()函数作为一个独立的函数，以便说明私有成员函数并且将范围检查的处理封装起来，方便我们以后改变边界错误的处理方式，或许是用异常处理代替 assert()。）

派生类对象实际上由几部分构成；每个基类是一个类的*子对象*(*subobject*)，它在新定义的派生类中有独立的一部分。派生类对象的初始化过程是这样的，首先自动调用每个基类的构造函数来初始化相关的基类子对象，然后执行派生类的构造函数。从设计的角度来看，派生类的构造函数应该只初始化那些在派生类中被定义的数据成员，而不是基类中的数据成员。

虽然我们引入了与类相关的下标操作符版本，以及一个私有的 check_range()辅助函数，但是我们并没有引入需要初始化的额外数据成员。因此，我们可以合理地假设，继承基类的构造函数已经足够了，我们不需要再提供 IntArrayRC 的构造函数——因为不需要它们做任何事情。

但是，实际上，我们还是需要提供 IntArrayRC 的构造函数，因为基类的构造函数并没有被派生类继承（析构函数和拷贝赋值操作符同样也没有），还因为我们需要某个接口，以便通过这个接口把必要的参数传递给基类 IntArray 的构造函数。

例如，假设我们定义了一个 IntArratRC 对象：

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13 };
IntArrayRC iarc( ia, 8 );
```

怎样才能把 ia 和 8 传递给基类的构造函数呢？（不可否认，如果 IntArray 构造函数被继承了，那么就没有这个问题了，实际上，那样的话我们会有其他更严重的问题，但现在没有足够的篇幅向你证明这一点。）无论如何，派生类构造函数的语法提供了向基类构造函数传递参数的接口。例如，下面是两个必需的 IntArrayRC 构造函数。（第 14 章与第 17 章将对构造函数作更多的讲解，其中包括关于“为什么我们不需要提供 IntArrayRC 拷贝构造函数”的解释）：

```
inline IntArrayRC::IntArrayRC( int sz)
```

```
    : IntArray( sz ) {}

inline IntArrayRC::IntArrayRC( const int *iar, int sz )
    : IntArray( iar, sz ) {}
```

由冒号分割出来的部分被称作*成员初始化列表*(member initialization list)，它提供了一种机制，通过这种机制，我们可以向 IntArray 的构造函数传递参数。两个 IntArrayRC 构造函数的函数体都是空的，因为它们的工作就是把参数传递给相关的 IntArray 构造函数。我们无需提供显式的 IntArrayRC 析构函数，因为派生类没有引入任何需要析构的数据成员。被继承过来的、需要析构的 IntArray 成员都由 IntArray 的析构函数来处理。

头文件 IntArrayRC.h 含有 IntArrayRC 的定义，以及在类定义之外定义的全部内联成员函数的定义。如果我们定义了非内联函数，则把它们放在 IntArrayRC.C ——这个相关联的程序文本文件中。

下面这个小程序实现了 IntArray 与 IntArrayRC 两个类的层次结构：

```
#include <iostream>
#include <IntArray.h>
#include <IntArrayRC.h>

extern void swap(IntArray&,int,int);

int main()
{
 int array[ 4 ] = { 0, 1, 2, 3 };
 IntArray ia1( array, 4 );
 IntArrayRC ia2( array, 4 );

 // 错误：一位偏移(off-by-one)：应该是size-1
 //  IntArray 对象捕捉不到这个错误
 cout << "swap() with IntArray ia1\n";
 swap( ia1, 1, ia1.size() );

 // ok: IntArrayRC对象可以捕捉到这样的错误
 cout << "swap() with IntArrayRC ia2\n";
 swap( ia2, 1, ia2.size() );

 return 0;
}
```

编译并执行这个程序，它产生如下结果：

```
swap() with IntArray ia1
swap() with IntArrayRC ia2
Assertion failed:  index >= 0 && index < size
```

C++支持另外两种形式的继承：多继承(multiple inheritance，或多重继承)，也就是一个

类可以从两个或多个基类派生而来；以及虚拟继承(virtual inheritance)，在这种继承方式下基类的单个实例在多个派生类之间共享。第 18 章将讨论这些内容。面向对象程序设计的另一个较为深入的方面是，在程序执行过程中任意一个点上，我们都能够查询基类的引用或指针所指向的实际类型。这是由运行时刻类型识别（RTTI）设施提供的，我们将在 19.1 节中讨论它。

练习 2.8

一般来说，类型/子类型继承关系反映了一种“*is-a*(*是一种*)”的关系：具有范围检查功能的 ArrayRC 是一种 Array；一本书(Book)是一种图书外借资源(LibraryRentalMaterial)；有声书(AudioBook)是一种书(Book)等等。下面哪些反映出这种“is-a”关系？

a) 成员函数是一种(isA_kindOf)函数
b) 成员函数是一种类
c) 构造函数是一种成员函数
d) 飞机是一种交通工具
e) 摩托车是一种卡车
f) 圆形是一种几何图形
g) 正方形是一种矩形
h) 汽车是一种飞机
i) 借阅者是一种图书馆

练习 2.9

判断以下操作哪些可能是类型相关的，因此可把它们定义为虚拟函数？。哪些可以在所有类之间共享？对单个基类或派生类来说哪些是唯一的？

```
(a) rotate();      (b) print();
(c) size();        (d) dateBorrowed();
(e) rewind();      (f) borrower();
(g) is_late();     (h) is_on_loan();
```

练习 2.10

对于保护（protected）访问级别的使用已经有了一些争论。有人认为，使用保护访问级别允许派生类直接访问基类的成员，这破坏了封装的概念，因此，所有的基类实现细节都应该是私有的（private）。另外一些人认为，如果派生类不能直接访问基类的成员，那么派生类的实现将无法有足够的效率供用户使用；如果没有关键字 protected，类的设计者将被迫把基类成员设置为 public。你怎样认为？

练习 2.11

第二个争论是关于将成员函数显式地声明为 virtual 的必要性。一些人认为，这意味着如果一个类的设计者没有意识到一个函数需要被声明为 virtual，则派生类的设计者就没有办法改写这个关键函数。因此，他们建议把所有成员函数都设置为 virtual 的。另一方面，虚拟函数比非虚拟函数的效率要低一些。[3] 因为它们不能被内联（内联发生在编译时刻，而虚拟函数是在运行时刻被处理的），所以它们可能是运行时刻效率低下的原因之一，尤其是小巧而

[3] 参见[LIPPMAN96a]，其中讨论了虚拟函数性能的问题。

又被频繁调用的、与类型无关的函数，比如 Array(数组)的 size 函数。你又怎样认为呢？

练习 2.12

下面的每个抽象类型都隐式地包含一族抽象子类型。例如，图书馆藏资料(LibraryRentalMaterial)抽象隐式地包含书(Book)、音像(Puppet)、视盘(Video)等。选择其中一个，找出该抽象的子类型层次，并为这个层次指定一个小的公有接口，其中包括构造函数。如果存在的话，指出哪些函数是虚拟的，并且写一小段伪代码程序来练习使用这个公有接口。

```
(a) Points        (b) Employees
(c) Shapes        (d) TelephoneNumbers
(e) BankAccounts (f) CourseOfferings
```

2.5 泛 型 设 计

IntArray 类为预定义的整型数组类型提供了一个有用的替代类型。如果用户希望使用一个 double 或 string 类型的数组，那该怎么办呢？实现一个 double 类型的数组与 IntArray 类的区别只是它要包含的元素的类型不同，代码本身无需改变。

C++的模板设施提供了一种机制，它能够将类或函数定义内部的类型和值*参数化*(*parameterizing*)（我们要到 10.1 节才会讨论值参数）。这些参数在其他方面不变的代码中用作占位符，以后，这些参数会被绑定到实际类型上，可能是内置的类型，也可能是用户定义的类型。例如，在 Array 类模板中，我们把数组所包含的元素的类型参数化。以后，我们*实例化*(*instantiate*)一个特定类型的实例，如 int、double 和 string 类型的 Array(数组)。在程序中我们可以直接使用这三个实例，就好像我们已经显式地为它们编写过代码一样。现在来看一下，怎样把 IntArray 类转换成 Array 类模板。下面是定义：

```
template < class elemType >
class Array {
public:
  // 把元素类型参数化
  explicit Array( int size = DefaultArraySize );
  Array( elemType *array, int array_size  );
  Array( const Array &rhs );

  virtual ~Array() { delete [] ia; }

  bool operator==( const Array& ) const;
  bool operator!=( const Array& ) const;

  Array& operator=( const Array& );
  int size() const { return _size; }

  virtual elemType& operator[](int index){ return ia[index]; }
  virtual void sort();
```

```
  virtual elemType min() const;
  virtual elemType max() const;
  virtual int      find( const elemType &value ) const;

protected:
  static const int DefaultArraySize = 12;

  int _size;
  elemType *ia;
};
```

关键字 template **引入模板，参数由一对尖括号（**<,>**）括起来——本例中，有一个参数** elemType**。关键字** class **表明这个参数代表一个类型。标识符** elemType **代表实际的参数名，它在** Array **类定义中出现了七次，都是作为实际类型的占位符。**

在 Array **类的每次实例化中，不论是实例化为** int**、**double **或** string **等等，实例化的实际类型将代替** elemType **参数。下面的例子演示了怎样使用** Array **类模板：**

```
#include <iostream>
#include "Array.h"

int main()
{
  const int array_size = 4;

  // elemType变成了int
  Array<int> ia(array_size);

  // elemType变成了double
  Array<double> da(array_size);

  // elemType变成了char
  Array<char> ca(array_size);

    int ix;
      for ( ix = 0; ix < array_size; ++ix ) {
        ia[ix] = ix;
        da[ix] = ix * 1.75;
        ca[ix] = ix + 'a';
    }

    for ( ix = 0; ix < array_size; ++ix )
        cout << "[ " << ix << " ]  ia: "  << ia[ix]
            << "\tca: " << ca[ix]
            << "\tda: " << da[ix] << endl;

  return 0;
}
```

本例中，我们定义了三个独立的 Array 类模板的实例：

```
Array<int>    ia(array_size);
Array<double> da(array_size);
Array<char>   ca(array_size);
```

这些实例声明就是在类模板名的后面加上一对尖括号，里面写上数组的实际类型。当我们定义类模板对象，如 ia、da 或 ca 时，会发生什么事情呢？编译器必须为相关的对象分配内存。为了做到这一点，形式模板参数被绑定到指定的实际参数类型上。对 ia 来说，Array 类模板通过将 elemType 绑定到类型 int 上，产生如下的类数据成员：

```
// Array<int> ia(array_size);
int _size;
int *ia;
```

结果是一个类，它与我们前面手工编码实现的 IntArray 类等价。对 da 来说，通过将 elemType 绑定到类型 double 上，成员变为：

```
// Array<double> da(array_size);
int _size;
double *ia;
```

类似地，对 ca 来说，通过将 elemType 绑定到类型 char 上，成员变为：

```
// Array<char> ca(array_size);
int _size;
char *ia;
```

类模板的成员函数会怎么样呢？不是所有的成员函数都能自动地随类模板的实例化而被实例化，只有真正被程序使用到的成员函数才会被实例化，这一般发生在程序生成过程中一个独立的阶段。（16.8 节将详细讨论这个过程。）

编译并运行程序，会产生如下结果：

```
[ 0 ]  ia: 0    ca: a    da: 0
[ 1 ]  ia: 1    ca: b    da: 1.75
[ 2 ]  ia: 2    ca: c    da: 3.5
[ 3 ]  ia: 3   ca: d    da: 5.25
```

模板机制也支持面向对象的程序设计。类模板可以作为基类或派生类。下面是一个带有范围检查的 Array 类模板的定义：

```
#include <cassert>
#include "Array.h"

template <class elemType>
class ArrayRC : public Array<elemType> {
public:
```

```
    ArrayRC( int sz = Array<elemType>::DefaultArraySize )
           : Array< elemType >( sz ){};

    ArrayRC( elemType *ia, int sz )
           : Array< elemType >( ia, sz ) {}

    ArrayRC( const ArrayRC &rhs )
           : Array< elemType >( rhs ) {}

    virtual elemType&
    operator[]( int index )
     {
           assert( index >= 0 && index < Array<elemType>::size() );
           return ia[ index ];
    }

private:
    // ...
};
```

每个 ArrayRC 类的实例化过程都会实例化相应的 Array 类模板的实例。例如，下面的定义

```
ArrayRC<int> ia_rc( 10 );
```

引起 Array 类和 ArrayRC 类的一个 int 实例被实例化。ia_rc 同上一节的非模板实例相同。为了说明这一点，我们重写前面的程序来练习 Array 和 ArrayRC 类模板类型。首先，为了支持语句

```
// 现在swap()必须也是一个模板
swap( ia1, 1, ia1.size() );
```

我们必须将 swap()定义成一个函数模板。例如：

```
#include "Array.h"
template <class elemType>
void swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}
```

每个 swap()调用都会根据数组的类型产生适当的实例。下面是重新改写之后的 main()函数，它使用了 Array 和 ArrayRC 类模板：

```
#include <iostream>

#include "Array.h"
#include "ArrayRC.h"
```

```
template <class elemType>
inline void
swap( Array<elemType> &array, int i, int j )
{
 elemType tmp = array[ i ];
 array[ i ] = array[ j ];
 array[ j ] = tmp;
}

int main()
{
 Array<int>   ia1;
 ArrayRC<int> ia2;

 cout << "swap() with Array<int> ia1\n";
 int size = ia1.size();
 swap( ia1, 1, size );

 cout << "swap() with ArrayRC<int> ia2\n";
 size = ia2.size();
 swap( ia2, 1, size );

 return 0;
}
```

程序的输出结果与非模板的 IntArray **类实现相同。**

练习 2.13

给出下列类型声明

```
template <class elemType> class Array;
enum Status { ... };
typedef string *Pstring;
```

如果存在的话，下面哪些对象的定义是错误的？

```
(a) Array< int*& > pri( 1024 );
(b) Array< Array<int> > aai( 1024 );
(c) Array< complex< double > > acd( 1024 );
(d) Array< Status > as( 1024 );
(e) Array< Pstring > aps( 1024 );
```

练习 2.14

重写下面的类定义，使它成为一个类模板：

```
class example1 {
public:
 example1( double min, double max );
```

```
  example1( const double *array, int size );

  double& operator[]( int index );
  bool operator==( const example1& ) const;

  bool insert( const double*, int );
  bool insert( double );

  double min() const { return _min; };
  double max() const { return _max; };

  void min( double );
  void max( double );

  int count( double value ) const;

private:
  int size;
  double *parray;
  double _min;
  double _max;
};
```

练习 2.15
给出如下的类模板

```
template <class elemType>
class Example2 {
public:
  explicit Example2( elemType val = 0 )
            : _val( val ){}

  bool min( elemType value )     { return _val < value; }
  void value( elemType new_val ) { _val = new_val;      }
  void print( ostream &os )      { os << _val;          }

private:
  elemType _val;
};

template<class elemType>
ostream& operator<<( ostream &os, const Example2<elemType> &ex )
         { ex.print( os ); return os; }
```

如下这样写会发生什么事情？

```
(a) Example2< Array<int>* > ex1;
(b) ex1.min( &ex1 );
(c) Example2< int > sa( 1024 ), sb;
```

```
(d) sa = sb;
(e) Example2< string > exs( "Walden" );
(f) cout << "exs: " << exs << endl;
```

练习 2.16

在 Example2 **的定义中，我们这样写**

```
explicit Example2( elemType val = 0 )
          : _val( val ){}
```

其意图是指定一个缺省值，以便用户可以这样写

```
Example2< Type > ex1( value );
Example2< Type > ex2;
```

但是，我们的实现把 Type **限制在一个“不能用** 0 **进行初始化的类型”的子集中。（例如，用** 0 **初始化一个** string **类型，就是一个错误。）**[4] **类似的情况是，如果** Type **不支持输出操作符，那么** print()**调用就会失败，（因此，**Example2 **的输出操作符也会失败。）如果** Type **不支持小于操作符，那么** min()**调用就会失败。**

C++**语言本身并没有提供方法可以指示在实例化模板时** Type **有哪些隐含的限制。在最坏的情况下，当程序编译失败时，程序员发现了这些限制。你认为** C++**语言应该支持限制** Type **的语法吗？如果你认为应该的话，请说明语法，并用它重写** Example2 **的定义；如果认为不需要，请说明理由。**

练习 2.17

在上一个练习中，我们说如果 Type **不支持输出操作符和小于操作符，那么对** print()**和** min()**的调用就会出错。在标准** C++**中，错误的产生不是发生在类模板被创建的时候，而是在** print()**与** min()**被调用的时候。你认为这样的语义正确吗？是否应该在模板定义中标记这个错误？为什么？**

2.6 基于异常的设计

异常（exception**）是指在运行时刻程序出现异常，例如数组下标越界、打开文件失败、可用动态内存耗尽等等。程序员一般有自己的处理异常的风格，这导致了不同的编码习惯，因而很难整合到一个单一的应用程序中。**

异常处理(exception handling)**为“响应运行时刻的程序异常”提供了一个标准的语言级的设施，它支持统一的语法和风格，也允许每个程序员进行微调。异常处理使得我们不需要在程序中处处显式地测试异常状态，从而可以将测试异常状态的代码抽取出来，放入指定的、显式标记的代码块中，因此异常处理设施大大地减少了程序代码的长度和复杂度。**

异常处理机制的主要构成如下：

[4] **通常解决这个问题的做法是：**Example2(elemType nval = elemType()) : _val(nval) {}

1．程序中异常出现的点。一旦识别出程序异常，就会导致*抛出(raising)*异常。当异常被抛出时，正常的程序被挂起，直到异常被处理完毕。在 C++中，异常的抛出由 throw 表达式来执行。例如，下面的程序段中，一个 string 类型的异常被抛出来以便响应打开文件失败异常。

```
if ( ! infile ) {
     string errMsg( "unable to open file: " );
     errMsg += fileName;
     throw errMsg;
}
```

2．程序中异常被处理的点。典型地，程序异常的抛出与处理位于独立的函数或成员函数调用中。找到处理代码通常要涉及到展开*程序调用栈*(*program call stack*)。一旦异常被处理完毕，就恢复正常的程序执行。但不是在发生异常的地方恢复执行过程，而是在处理异常的地方恢复执行过程。C++中，异常的处理由 catch 子句来执行。例如，下面的 catch 子句处理在第 1 项中被抛出的异常：

```
catch( string exceptionMsg ) {
     log_message( exceptionMsg );
     return false;
}
```

catch 子句与 try 块相关联。一个 try 块用一个或多个 catch 子句将一条或多条语句组织起来。例如，下面是函数 stats()：

```
int*
stats( const int *ia, int size )
{
 int *pstats = new int[ 4 ];
 try {
    pstats[ 0 ] = sum_it(  ia, size );
    pstats[ 1 ] = min_val( ia, size );
    pstats[ 2 ] = max_val( ia, size );
 }
 catch( string exceptionMsg )
    { /* 处理异常的代码 */ }

 catch( const statsException &statsExcp )
    { /* 处理异常的代码 */  }

 pstats[ 3 ] = pstats[ 0 ]/size;
 do_something( pstats );

 return pstats;
}
```

在 stats()内部有 4 条语句在 try 块之外，在下面两条语句完成之前，可能会有异常被抛出：

```
(1) int *pstats = new int[ 4 ];
(2) do_something( pstats );
```

在语句（1）中，new 表达式可能会失败。如果发生了这样的情况，标准库将产生 bad_alloc 标准异常。由于 bad_alloc 是在 try 块之外被抛出的，所以在 stats()中并没有试图要处理它，于是函数将终止：pstats 没有被初始化，stats()中后面的语句也不会被执行。异常机制承接了控制权并一直保持控制权直到异常被处理。

在语句（2）中，在 do_something()中的语句，以及在 do_something()中被调用的语句，或 do_something()函数中被调用的函数所调用的语句，等等，都可能会抛出异常。在从 do_something()调用开始的函数调用链返回之前，这个异常可能（也可能不）会被捕捉到。如果异常被处理了，那么 stats()继续执行，就像什么也没有发生过一样。如果在 do_something()结束之前，异常没有被处理，那么 stats()也会被终止，因为异常发生在 try 块之外。

（注意，如果 size 等于 0，那么

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

将导致一个除以 0 的除法。尽管这将导致向 pstats[3] 赋一个未定义的数据值，但是对于除以 0 并没有标准异常被抛出。）

try 块内的三条语句会怎么样呢？不同的行为区别如下：如果在 stats()里面 sum_it()、min_val()、max_val()终止之后，被抛出的异常是活动的(有效的)，那么系统不是简单地终止 stats()，而是顺序地检查与 try 块相关联的 catch 子句，试图处理被抛出来的异常。假设 sum_it()抛出如下异常：

```
throw string( "internal error: adump27832" );
```

则 pstats[0]不会被初始化，在 try 块中接下来的两条语句也不会被执行，异常机制意识到 sum_it()是在 try 块中被调用的，它将检查相关的两条 catch()子句。

系统根据被抛出来的异常与 catch 子句中异常类型的匹配情况来选择 catch 子句。本例中，异常是 string 类型，与下面的 catch 子句相匹配：

```
catch( string exceptionMsg )
   { /* code to handle exception */ }
```

系统把控制传递给被选中的 catch 子句体，其中的语句将顺序执行。完成之后，除非在处理该异常的子句中又抛出异常，否则控制将被传回到程序的当前点上。例如，如果我们已经这样写：

```
catch( string exceptionMsg )
{
  cerr << "stats(): exception occurred: "
       << exceptionMsg << endl;
  pstats[0] = pstats[1] = pstats[2] = 0;
}
```

那么，在 catch 子句完成时，控制将被传递给 catch 子句集后面的可执行语句。本例中，语句

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

被执行，然后是 do_something()调用，以及返回 pstats。而调用 stats()的函数根本不知道曾经有异常被抛出。

一段更为合理的异常处理代码可能如下所示：

```
catch( string exceptionMsg )
{
 cerr  << "stats(): exception occurred: "
    << exceptionMsg
    << " unable to stat array "
    << endl;
delete [] pstats;
return 0;
    }
```

在上面的代码中，catch 子句直接把控制返回给外面的调用函数。我们希望外面的函数在把返回值用作索引数组之前，应该测试它是否为 0。

如果 try 块内抛出的异常不能被相关联的 catch 子句处理，那么函数将被终止，然后，异常机制在调用 stats()的函数中查找处理代码。

如果异常机制按照函数被调用的顺序回查每个函数直到 main()函数，仍然没有找到处理代码，那么它将调用标准库函数 terminate()。缺省情况下，terminate()函数结束程序。

一种特殊的、能够处理全部异常的 catch 子句如下：

```
catch( ... )
{
  // handles all exceptions, although it cannot
  // directly access the exception object
}
```

我们可以把它看作是一种捕捉所有异常（catch-all）的 catch 子句。

异常处理机制为统一地处理程序异常提供了语言一级的设施。第 11 章与 19 章将进一步详细讨论。另一本配套的书“ Inside the C++ Object Model ”([LIPPMAN96a])中讨论了实现与性能的话题。Josee Lajoie 在[LIPPMAN96b]中的文章“ Exception Handling: Behind the Scenes ”对此也有讨论。[LIPPMAN96b]中 Tom Cargill 的文章“ Exception Handling: A False Sense of Security ”则对使用异常处理过程中易犯的错误作了很好的讨论。

练习 2.18

下面的函数对可能的非法数据以及可能的操作失败完全没有提供检查。找出程序中所有可能出错的地方。（本练习中，我们不关心可能会抛出的异常。）

```
int *alloc_and_init( string file_name )
{
 ifstream infile( file_name );
```

```
    int elem_cnt;
    infile >> elem_cnt;
    int *pi = allocate_array( elem_cnt );

    int elem;
    int index = 0;
    while ( cin >> elem )
          pi[ index++ ] = elem;

    sort_array( pi, elem_cnt );
    register_data( pi );

    return pi;
}
```

练习 2.19

alloc_and_init()**函数会调用到下面的函数，如果这些函数调用失败了，它们将抛出相应类型的异常：**

```
allocate_array() noMem
sort_array()    int
register_data()  string
```

请在合适的地方插入一个或多个 try **块以及相应的** catch **子句来处理这些异常。在** catch **子句中只需简单地输出错误的出现情况。**

练习 2.20

检查在练习 2.18 **中的函数** alloc_and_int()**中所有可能出现的错误，指出哪些错误会抛出异常。修改该函数（或用练习** 2.18 **的版本，或用练习** 2.19 **的版本）来抛出可被识别的异常（抛出文字串就可以了。）**

2.7 用其他名字来命名数组

把代码分发给其他部门的困难之一是我们不知道全局名字会有什么样的影响。例如，在 Intel **公司，有人写了**

```
class Array { ... };
```

那么他就不能在相同的程序中既使用上面的 Array **类，也使用我们实现的那个** Array **类。名字的可视性使这两份实现代码相互排斥。**

在标准 C++**之前，解决这个问题的传统做法是在全局可见的名字前加一个唯一的字符串前缀。例如，我们可以这样发行数组** Array **类：**

```
class Cplusplus_Primer_Third_Edition_Array { ... };
```

虽然这个名字可能是唯一的（但我们不能保证这一点），但是写起来并不方便。标准

C++的名字空间机制是 C++语言针对这个问题提供的语言一级的解决方案。

名字空间机制允许我们封装名字，否则这些名字有可能会*污染（影响）全局名字空间*(*pollute the global namespace*)。一般来说，只有当我们希望自己的代码被外部软件开发部门使用时，我们才使用名字空间。例如，我们可以这样封装 Array 类：

```
namespace Cplusplus_Primer_3E {
template <class elemType>
  class Array { ... };
// ...
}
```

关键字 namespace 后面的名字标识了一个名字空间，它独立于全局名字空间，我们可以在里面放一些希望声明在函数或类之外的实体。名字空间并不改变其中的声明的意义，只是改变了它们的可视性。在继续讨论之前，我们先扩展我们的可用名字空间集：

```
namespace IBM_Canada_Laboratory {
  template <class elemType>
   class Array { ... };
  class Matrix { ... };
  // ...
}

namespace Disney_Feature_Animation {
  class Point { ... };
  template <class elemType, int size>
   class Array { ... };

  // ...
}
```

如果名字空间内的声明对程序而言不是立即可见的,那么我们怎样访问它们呢？我们可以使用*限定修饰名字符*(*qualified name notation*)，格式如下：

```
namespace_identifier::entity_name;
```

如在

```
Cplusplus_Primer_3E::Array<string> text;
IBM_Canada_Laboratory::Matrix mat;
Disney_Feature_Animation::Point origin( 5000, 5000 );
```

虽然 Disney_Feature_Animation、IBM_Canada_Laboratory 以及 Cplusplus_Primer_3E 都能够唯一地标识相应的名字空间，但是，如果在程序中经常这样使用，则多少有些麻烦。使用名字空间标识符如 P3E、DFA 或 IBM_CL 会更方便一些，但是它们表达的信息相对比较少，同时也增加了名字冲突的可能性。为了提供有意义的名字空间标识符，同时程序员又能很方便地访问在名字空间内定义的实体，C++提供了别名设施。

名字空间别名(*namespace alias*)允许用一个可替代的、短的或更一般的名字与一个现有

的名字空间关联起来。例如：

```
// 提供一个更一般化的别名
namespace LIB = IBM_Canada_Laboratory;

// 提供一个更短的别名
namespace DFA = Disney_Feature_Animation;
```

然后这个别名就可以用作原始名字空间的同义词。例如：

```
#include "IBM_Canada.h"
namespace LIB = IBM_Canada_Laboratory;

int main()
{
    LIB::Array<int> ia(1024);
    // ...
}
```

别名也可以被用来封装正在使用的实际名字空间。例如，在此情形下，我们可以通过改变分配给别名的名字空间，来改变所使用的声明集，而无需改变“通过别名访问这些声明”的实际代码。例如：

```
namespace LIB = Cplusplus_Primer_3E;

int main()
{
    // 在这种情况下，下面的声明无须改变
    LIB::Array<int> ia(1024);
    // ...
}
```

但是，如果要让这项技术在实际工作中发挥作用，那么两个名字空间中的声明必须提供同样的接口。例如，下面的代码就不能工作，因为 Disney 的 Array 类需要一个类型参数和一个数组长度参数：

```
namespace LIB = Disney_Feature_Animation;

int main()
{
    // 不再是一个有效的声明
    LIB::Array<int> ia(1024);
    // ...
}
```

程序员常常希望在访问名字空间内声明的名字时不加限定修饰符。即使我们已经为名字空间标识符提供了较短的别名，在每次访问该名字空间内声明的名字时都要进行限定，还是太麻烦。*using* ***指示符***(*using directive*)使一个名字空间内的所有声明都可见，以便使这些声明能够不加限定就可被使用。例如：

```
#include "IBM_Canada_Laboratory.h"

// 使所有的名字都可见
using namespace IBM_Canada_Laboratory;

int main()
{
    // ok: IBM_Canada_Laboratory::Matrix
    Matrix mat( 4,4 );

    // ok: IBM_Canada_Laboratory::Array
    Array<int> ia( 1024 );
    // ...
}
```

using 与 namespace 都是关键字。被引用的名字空间必须已经被声明了，否则会引起编译错误。

using ***声明***(*using declaration*)提供了选择更为精细的名字可视性机制。它允许使名字空间中的单个声明可见。例如：

```
#include "IBM_Canada_Laboratory.h"

// 只让Matrix可见
using IBM_Canada_Laboratory::Matrix;

int main()
{
    // ok: IBM_Canada_Laboratory::Matrix
    Matrix mat(4,4);

    // 错误: IBM_Canada_Laboratory::Array不可见
    Array<int> ia( 1024 );
    // ...
}
```

为了防止标准 C++库的组件污染用户程序的全局名字空间，所有标准 C++库的组件都被声明在一个被称为 std 的名字空间内。正如在第 1 章中提到的，即使我们在程序文本文件中包含了 C++库头文件，头文件中声明的组件在我们的文本文件中也不是自动可见的。例如，在标准 C++中，下面的代码实例就不能正常编译。

```
#include <string>
 // 错误：string不是可见的
 string current_chapter = "A Tour of C++";
```

在<string>头文件中的所有声明都包含在名字空间 std 中。正如第 1 章所提到的，我们可以用“ 在#include 预处理器指示符后面加上 using 指示符 ”的办法，使得 C++头文件<string>

中的、在名字空间 std 中声明的组件对于我们的程序都是可见的：

```
#include <string>
  using namespace std;

  // ok: string是可见的
  string current_chapter = "A Tour of C++";
```

为了使在名字空间 std 中声明的名字在我们的程序中可见，指示符 using 通常被看作是一种比较差的选择方案。在上面的例子中，指示符 using 使头文件<string>中声明的、并且位于名字空间 std 中的所有组件在程序文本文件中都是可见的，这又将全局名字空间污染问题带回来了。而这个问题正是 std 名字空间首先要努力避免的，它增加了“ C++标准库组件的名字 ”与“ 我们程序中声明的全局名字 ”冲突的机会。

现在我们对名字空间机制已经有了一些了解，我们知道有另外两种机制可代替指示符 using，使我们能够引用到隐藏在名字空间std 中的名字 string。我们可以使用限定的名字，例如：

```
#include <string>
  // ok: 使用限定的名字
  std::string current_chapter = "A Tour of C++";
```

或如下使用 using 声明：

```
#include <string>
  using std::string;

  // ok: 上面的using声明使string可见
  string current_chapter = "A Tour of C++";
```

为了使用名字空间中声明的名字，建议使用带有精细选择功能的 using 声明代替 using 指示符。这正是本书的代码示例中没有出现 using 指示符的另一个原因。理想情况下，每一个代码示例对它所用到的每个库组件都应该有一个 using 声明。为了限制例子代码的长度，也因为本书的许多例子是在不支持名字空间的情况下被编译的，所以 using 声明就没有显示出来。8.6 节将进一步讨论怎样对标准 C++库的组件使用 using 声明。

在接下来的四章中，我们将讲述另外四个类的设计。第 3 章讲的是 String(字符串)类的设计与实现，第 4 章介绍整数 Stack(栈)类的设计。第 5 章是 List(列表)类，第 6 章对第 4 章定义的 Stack(栈)类进行重新设计。名字空间机制允许我们把每个类放在单独的头文件中，但是仍然能把它们的名字封装到单个 Cplusplus_Primer_3E 名字空间中。在第 8 章中我们将讨论这项技术，并对名字空间作更多的介绍。

练习 2.21

给出如下名字空间定义：

```
namespace Exercise {
 template <class elemType>
   class Array { ... };
```

```
  template <class Etype>
    void print( Array< Etype > );

  class String { ... };
  template <class listType>
    class List { ... };
}
```

以及下面的程序：

```
int main() {
  const int size = 1024;
  Array< String > as( size );
  List< int > il( size );

  // ...

  Array< String > *pas = new Array<String>(as);
  List <int> *pil = new List<int>(il);

  print( *pas );
}
```

因为类型名被封装在名字空间中，所以当前程序编译失败。把程序修改为：

1．**用限定名字修饰符来访问名字空间** Exercise **中的类型定义。**

2．**使用** using **声明来访问类型定义。**

3．**用名字空间别名机制。**

4．**用** using **指示符。**

2.8　标准数组——向量(vector)

正如我们已经看到的，尽管 C++**内置的数组支持容器的机制，但是它不支持容器抽象的语义。为了在这样的层次上编写程序，在标准** C++**之前，我们必须要么从某个途径获得这样的类，要么自己实现这样的类。在标准** C++**中，数组类是** C++**标准库的一部分，现在它不叫数组，而是叫做向量。**

当然，向量是一个类模板，所以我们这样写：

```
vector<int>   ivec( 10 );
vector<string> svec( 10 );
```

上面的代码分别定义了一个包含 10 **个整型对象的向量和一个包含** 10 **个字符串对象的向量。**

在我们实现的 Array 类模板与 vector 类模板的实现之间有两个主要区别。第一个区别是 vector 类模板支持‘ 向现有的数组元素赋值 ’的概念以及‘ 插入附加元素 ’的概念——即 vector 数组可以在运行时刻动态增长（如果程序员希望使用这个特性的话）。第二个区别是更加广泛，代表了设计方法的重要转变。vector 类不是提供一个巨大的“ 可以适用于向量 ”的成员操作集，如 sort()、min()、max()、find()等等，而是只提供了一个最小集：如等于、小于操作符、size()、empty()等操作。而一些通用的操作如 sort()、min()、max()、find()等等，则是作为独立的*泛型算法*(*generic algorithms*)被提供的。

要定义一个向量，我们必须包含相关的头文件：

```
#include < vector >
```

下面都是 vector 对象的合法定义：

```
#include < vector >

// 创建vector对象的各种方法
vector<int> vec0;     // 空的vector

const int size  = 8;
const int value = 1024;

// size为8的vector
// 每个元素都被初始化为0
vector<int> vec1( size );

// size为8的vector
// 每个元素都被初始化为1024
vector<int> vec2( size, value );

// vec3的size为4
// 被初始化为ia的4个值
int ia[4] = { 0, 1, 1, 2 };
vector<int> vec3( ia, ia+4 );

// vec4是vec2的拷贝
vector<int> vec4( vec2 );
```

既然定义了向量，我们就需要遍历里边的元素。与 Array 类模板一样，标准 vector 类模板也支持使用下标操作符，例如：

```
#include <vector>
extern int getSize();

void mumble()
{
    int size = getSize();
    vector< int > vec( size );

    for ( int ix = 0; ix < size; ++ix )
```

```
        vec[ ix ] = ix;

    // ...
}
```

另外一种遍历方法是使用*迭代器对*(*iterator pair*)来标记向量的起始处和结束处。迭代器是一个支持指针类型抽象的类对象。vector 类模板提供了一对操作 begin()和 end()，它们分别返回指向向量开始处和结束处再往后 1 个的迭代器。一对迭代器合起来可以标记出待遍历元素的范围。例如，下面的代码是前面代码段的一个等价实现：

```
#include < vector >
extern int getSize();

void mumble()
{
    int size = getSize();
    vector< int > vec( size );

    vector< int >::iterator iter = vec.begin();

    for ( int ix = 0; iter != vec.end(); ++iter, ++ix )
       *iter = ix;

    // ...
}
```

iter 的定义

```
vector< int >::iterator iter = vec.begin();
```

将其初始值指向 vec 的第一个元素。iterator 是 vector 类模板中用 typedef 定义的类型，而这里的 vector 类实例包含 int 类型的元素。下面的代码使迭代器指向 vector 的下一个元素：

```
++iter
```

下面的代码解除迭代器的引用，以便访问实际的元素：

```
*iter
```

能够应用到向量上的操作惊人地多。但是它们并不是作为 vector 类模板的成员函数提供的，而是以一个独立的泛型算法集的形式，由标准库提供。下面是一组可供使用的泛型算法的示例：

- ***搜索***(*search*)**算法：**find()、find_if()、search()、binary_search()、count()**和** count_if()。
- ***分类排序***(*sorting*)**与通用*排序***(*ordering*)**算法：**sort()、partial_sort()、merge()、partition()、rotate()、reverse()**和** random_shuffle()。
- ***删除***(*deletion*)**算法：**unique()**和** remove()。
- ***算术***(*numeric*)**算法：**accumulate()、partial_sum()、inner_product()**和** adjacent_difference()。

- ***生成***(*generation*)**和** ***变异***(*mutation*)**算法：**generate()、fill()、transformation()、copy()**和** for_each()**。**
- ***关系***(*Relational*)**算法：**equal()、min()**和** max()**。**

泛型算法接受一对迭代器，它们标记了要遍历元素的范围。例如，ivec **是一个包含某种类型元素的、某个长度的向量，要“**sort()**”它的全部元素，我们只需简单地这样写：**

```
sort( ivec.begin(), ivec.end() );
```

只想“sort()**”**ivec **向量的前面一半，可以这样写：**

```
sort( ivec.begin(), ivec.begin()+ivec.size()/2 );
```

泛型算法还能接受指向内置数组的指针对，例如，已知数组

```
int ia[7] = { 10, 7, 9, 5, 3, 7, 1 };
```

我们可以如下“sort()**”整个数组：**

```
sort( ia, ia+7 );
```

我们还可以只“sort()**”前四个元素：**

```
sort( ia, ia+4 );
```

要使用这些算法，我们必须包含与它们相关的头文件：

```
#include <algorithm>
```

下面的代码显示了怎样把各种各样的泛型算法应用到 vector **类对象上：**

```
class object:
#include <vector>
#include <algorithm>
#include <iostream>

int ia[ 10 ] = {
    51, 23, 7, 88, 41, 98, 12, 103, 37, 6 };

int main()
{
    vector< int > vec( ia, ia+10 );

    // sort the array
    sort( vec.begin(), vec.end() );

    // grab value to search for
    int search_value;
    cin >> search_value;

    // search for an element
    vector<int>::iterator found;
    found = find( vec.begin(), vec.end(), search_value );
    if ( found != vec.end() )
```

```
            cout << "search_value found!\n";
        else cout << "search_value not found!\n";

    // reverse the array
    reverse( vec.begin(), vec.end() );

    // ...
}
```

标准库还提供了对 *map* 关联数组的支持，即数组元素可以被整数值之外的其他东西索引。例如，我们可以这样来支持一个电话目录：这个电话目录是电话号码的数组，但是它的元素可以由该号码所属人的姓名来索引：

```
#include <map>
#include <string>
#include "TelephoneNumber.h"

map< string, telephoneNum > telephone_directory;
```

在第 6 章我们将看到 vector、map 以及标准 C++库支持的其他容器类型，我们将通过一个文本查询系统的实现来说明这些类型的用法。第 12 章会讲解泛型算法。附录按字母顺序提供了每个算法的解释及其用法。

本章大致地讲述了 C++为数据抽象（基于对象的程序设计）、面向对象的程序设计、泛型程序设计（模板、容器类型以及泛型算法）、大型程序设计（异常处理与名字空间）提供的支持。本书余下的部分将更详细地介绍这些内容，逐步讲解 C++中基本的，但又非常先进的特性。

练习 2.22

解释每个 vector 定义的结果：

```
string pals[] = {
      "pooh", "tigger", "piglet", "eeyore", "kanga" };

(a) vector<string> svec1( pals, pals+5 );
(b) vector<int>    ivec1( 10 );
(c) vector<int>    ivec2( 10, 10 );
(d) vector<string> svec2( svec1 );
(e) vector<double> dvec;
```

练习 2.23

已知下列函数声明，请实现 min()的函数体，它查找并返回 vec 的最小元素。要求首先使用“索引 vec 中元素的 for 循环”来实现 min()，然后，使用“通过迭代器遍历 vec 的 for 循环”来实现 min()：

```
template <class elemType>
elemType
min( const vector<elemType> &vec );
```