第37章 数字类

在C++标准化过程中增加的特征之一是数字类库，这些类对开发数字程序很有帮助。这些类的几个成员函数与从C库中继承来的独立函数类似，区别是这里描述的许多数字函数对valarray类型的对象进行操作（valarray本质上是值的数组），或者是对complex类型的对象（表示数字）进行操作。通过加入数字类，标准C++把编程任务的范畴扩展到了它适用的地方。

37.1 complex类

头文件<complex>定义了complex类，它表示复数。它也定义了一系列函数和对complex类型的对象进行操作的运算符。

complex的模板规范如下所示：

```
template <class T> class complex
```

其中，T规定了用来存储复数组成部分的类型。有三种预定义的complex规范：

```
class complex<float>
class complex<double>
class complex<long double>
```

complex类有下列构造函数：

```
complex(const T &real = T( ), const T &imaginary = T( ));
complex(const complex &ob);
template <class T1> complex(const complex<T1> &ob);
```

第一种形式构造一个complex对象，其实部为real，虚部为imaginary。如果没有指定的话，这些值默认为0。第二种形式创建一个ob的副本。第三种形式创建一个来自ob的complex对象。

下面的操作是为complex对象定义的：

```
+          -          *          /
-=         +=         /=         *=
=          ==         !=
```

非赋值运算符以三种方式被重载。一种用于涉及左边的一个complex对象和右边的标量对象的操作。另一种用于涉及左边的标量和右边的一个complex对象的操作，最后一种用于涉及两个complex对象的操作。例如，下面的操作是可行的：

```
complex_ob + scalar
scalar + complex_ob
complex_ob + complex_ob
```

涉及标量数的操作仅影响实部。

两个成员函数是为complex定义的：real( )和imag( )，如下所示：

```
T real( ) const;
T imag( ) const;
```

real( )函数返回调用对象的实部，imag( )返回虚部。表37.1中所示的函数也是为complex对象定义的。

下面是一个演示 complex 的范例程序。

```
// Demonstrate complex.
#include <iostream>
#include <complex>
using namespace std;

int main()
{
  complex<double> cmpx1(1, 0);
  complex<double> cmpx2(1, 1);

  cout << cmpx1 << " " << cmpx2 << endl;

  complex<double> cmpx3 = cmpx1 + cmpx2;
  cout << cmpx3 << endl;

  cmpx3 += 10;
  cout << cmpx3 << endl;

  return 0;
}
```

它的输出如下：

```
(1,0) (1,1)
(2,1)
(12,1)
```

表37.1 为complex定义的函数

函数	说明
template <class T> T abs(const complex<T> &*ob*);	返回 ob 的绝对值
template <class T> T arg(const complex<T> &*ob*);	返回 ob 的相限角
template <class T> complex<T> conj(const complex<T> &*ob*);	返回 ob 的共轭值
template <class T> complex<T> cos(const complex<T> &*ob*);	返回 ob 的余弦
template <class T> complex<T> cosh(const complex<T> &*ob*);	返回 ob 的双曲余弦
template <class T> complex<T> exp(const complex<T> &*ob*);	返回 e^{ob}
template <class T> T imag(const complex<T> &*ob*);	返回 ob 的虚部

（续表）

函数	说明
template <class T> complex<T> log(const complex<T> &*ob*);	返回 ob 的自然对数
template <class T> complex<T> log10(const complex<T> &*ob*);	返回 ob 的以 10 为底的对数
template <class T> T norm(const complex<T> &*ob*);	返回 ob 平方后的绝对值
template <class T> complex<T> polar(const T &*v*, const T &*theta*=0);	返回一个复数，该复数的绝对值由 v 指定，相角为 theta
template <class T> complex<T> pow(const complex<T> &*b*, int *e*);	返回 b^e
template <class T> complex<T> pow(const complex<T> &*b*, const T &*e*);	返回 b^e
template <class T> complex<T> pow(const complex<T> &*b*, const complex<T> &*e*);	返回 b^e
template <class T> complex<T> pow(const T &*b*, const complex<T> &*e*);	返回 b^e
template <class T> T real(const complex<T> &*ob*);	返回 ob 的实部
template <class T> complex<T> sin(const complex<T> &*ob*);	返回 ob 的正弦
template <class T> complex<T> sinh(const complex<T> &*ob*);	返回 ob 的双曲正弦
template <class T> complex<T> sqrt(const complex<T> &*ob*);	返回 ob 的平方根
template <class T> complex<T> tan(const complex<T> &*ob*);	返回 ob 的正切
template <class T> complex<T> tanh(const complex<T> &*ob*);	返回 ob 的双曲正切

37.2　valarray 类

头文件<valarray>定义了支持数字数组的若干类。主类是valarray，它定义了一维数值数组。许多成员运算符和函数为它而定义，当然也有许多非成员函数。虽然这里给出的 valarray 的说明对大多数程序员来讲足够了，但是对数字处理特别感兴趣的程序员将会想要更详细地研究 valarray。还有，尽管 valarray 很大，它的运算很直观。

valarray 类的模板规范是：

```
template <class T> class valarray
```

它定义了下面的构造函数：

```
valarray( );
explicit valarray (size_t num);
valarray(const T &v, size_t num);
valarray(const T *ptr, size_t num);
valarray(const valarray<T> &ob);
valarray(const slice_array<T> &ob);
valarray(const gslice_array<T> &ob);
valarray(const mask_array<T> &ob);
valarray(const indirect_array<T> &ob);
```

其中，第一个构造函数创建了一个空对象。第二个创建了一个长度为 num 的 valarray。第三个创建一个其长度 num 被初始化为 v 的 valarray。第四个创建一个长度为 num 的 valarray，该 valarray 用 ptr 所指的元素进行了初始化。第五个创建 ob 的一个副本。下面的四个构造函数从 valarray 的一个帮助者类中创建了一个 valarray。

下面的运算符是为 valarray 定义的：

```
+          -          *          /
-=         +=         /=         *=
=          ==         !=         <<
>>         <<=        >>=        ^
^=         %          %=         ~
!          |          |=         &
&=         [ ]
```

这些运算符有几个重载的形式，将在所给的几个表中讲述。

valarray定义的成员函数和运算符显示于表37.2中。为valarray定义的非成员运算符函数显示于表 37.3 中。为 valarray 定义的先验函数（transcendental）显示于表 37.4 中。

表 37.2　valarray 定义的成员函数

函数	说明
valarray<T> apply(T *func*(T)) const; valarray<T> apply(T *func*(const T &*ob*)) const;	把 func( )应用到调用数组中并返回包含结果的一个数组
valarray<T> cshift(int *num*) const;	向左旋转调用数组 num 个位置（即，它执行一个向左的圆形移位）。返回一个包含结果的数组
T max( ) const;	返回调用数组中的最大值
T min( ) const	返回调用数组中的最小值

（续表）

函数	说明
valarray<T> &operator=(const valarray<T> &*ob*);	把ob中的元素赋给调用数组中的对应元素。返回一个到调用数组的引用
valarray<T> &operator=(const T &*v*);	给调用数组中的每个元素赋值v。返回一个到调用数组的引用
valarray<T> &operator=(const slice_array<T> &*ob*);	给一个子集赋值。返回一个到调用数组的引用
valarray<T> &operator=(const gslice_array<T> &*ob*);	给一个子集赋值。返回一个到调用数组的引用
valarray<T> &operator=(const mask_array<T> &*ob*);	给一个子集赋值。返回一个到调用数组的引用
valarray<T> &operator=(const indirect_array<T> &*ob*);	给一个子集赋值。返回一个到调用数组的引用
valarray<T> operator+( ) const;	把一元加应用于调用数组的每个元素。返回所得结果的数组
valarray<T> operator-( ) const;	把一元减应用于调用数组的每个元素。返回所得结果的数组
valarray<T> operator~( ) const;	把一元按位非（NOT）应用于调用数组的每个元素。返回所得结果的数组
valarray<T> operator!( ) const;	把一元逻辑非(NOT)应用于调用数组的每个元素。返回所得结果的数组
valarray<T> &operator+=(const T &*v*) const;	把v添加到调用数组的每个元素中。返回一个到调用数组的引用
valarray<T> &operator-=(const T &*v*) const;	从调用数组的每个元素中减去v。返回一个到调用数组的引用
valarray<T> &operator/=(const T &*v*) const;	把调用数组中的每个元素用v除。返回一个到调用数组的引用
valarray<T> &operator*=(const T &*v*) const;	把调用数组中的每个元素乘以v。返回一个到调用数组的引用
valarray<T> &operator%=(const T &*v*) const;	把被v除的余数赋给调用数组中的每个元素。返回一个到调用数组的引用
valarray<T> &operator^=(const T &*v*) const;	与调用数组中的每个元素进行“异或”（XOR）操作。返回一个到调用数组的引用
valarray<T> &operator&=(const T &*v*) const;	与调用数组中的每个元素进行“与”（AND）操作。返回一个到调用数组的引用
valarray<T> &operator|=(const T &*v*) const;	与调用数组中的每个元素进行“或”（OR）操作。返回一个到调用数组的引用
valarray<T> &operator<<=(const T &*v*) const;	左移调用数组中的每个元素v个位置。返回一个到调用数组的引用
valarray<T> &operator>>=(const T &*v*) const;	右移调用数组中的每个元素v个位置。返回一个到调用数组的引用
valarray<T> &operator+=(const valarray<T> &*ob*) const;	把ob和调用数组中的相应元素加到一起。返回一个到调用数组的引用
valarray<T> &operator-=(const valarray<T> &*ob*) const;	从调用数组中与ob对应的元素中减去ob中也存在的元素。返回一个到调用数组的引用
valarray<T> &operator/=(const valarray<T> &*ob*) const;	调用数组中的元素被它们在ob中的对应的元素进行除法操作。返回一个到调用数组的引用

（续表）

函数	说明
valarray<T> &operator*=(const valarray<T> &*ob*) const;	把ob和调用数组中的对应元素进行乘法操作。返回一个到调用数组的引用
valarray<T> &operator%=(const valarray<T> &*ob*) const;	调用数组中的元素被其ob中对应的元素进行“除”操作并保存余数。返回一个到调用数组的引用
valarray<T> &operator^=(const valarray<T> &*ob*) const;	对ob和调用数组中相对应的元素运用XOR运算符。返回一个到调用数组的引用
valarray<T> &operator&=(const valarray<T> &*ob*) const;	对ob和调用数组中相对应的元素运用AND运算符。返回一个到调用数组的引用
valarray<T> &operator|=(const valarray<T> &*ob*) const;	对ob和调用数组中相对应的元素运用OR运算符。返回一个到调用数组的引用
valarray<T> &operator<<=(const valarray<T> &*ob*) const;	把调用数组中的元素左移ob中对应元素所指定的位置数。返回一个到调用数组的引用
valarray<T> &operator>>=(const valarray<T> &*ob*) const;	把调用数组中的元素右移ob中对应元素所指定的位置数。返回一个到调用数组的引用
T &operator[ ] (size_t *indx*) ;	返回一个到所指定下标处的元素的引用
T operator[ ] (size_t *indx*) const;	返回在所指定下标处的值
slice_array<T> operator[ ](slice *ob*);	返回指定的子集
valarray<T> operator[ ](slice *ob*) const;	返回指定的子集
gslice_array<T> operator[ ](const gslice &*ob*);	返回指定的子集
valarray<T> operator[ ](const gslice &*ob*) const;	返回指定的子集
mask_array<T> operator[ ](valarray<bool> &*ob*);	返回指定的子集
valarray<T> operator[ ](valarray<bool> &*ob*) const;	返回指定的子集
indirect_array<T> operator[ ](const valarray<size_t> &*ob*);	返回指定的子集
valarray<T> operator[ ](const valarray<size_t> &*ob*) const;	返回指定的子集
void resize(size_t *num*, T *v* = T( ));	重新调整调用数组的大小。如果必须增加元素，它们被赋值v
size_t size( ) const;	返回调用数组的大小(即，元素数)
valarray<T> shift(int *num*) const;	把调用数组左移num个位置。返回包含结果的数组
T sum( ) const;	返回存储在调用数组中值的总和

表37.3 为valarray定义的非成员运算符函数

函数	说明
template <class T> valarray<T> operator+(const valarray<T> *ob*, const T &*v*);	把v加到ob的每个元素中。返回包含结果的数组
template <class T> valarray<T> operator+(const T &*v*, const valarray<T> *ob*);	把v加到ob的每个元素中。返回包含结果的数组

（续表）

函数	说明
template <class T> valarray<T> operator+(const valarray<T> ob1, const valarray<T> &ob2);	把ob1中的每个元素添加到它在ob2的对应元素中。返回包含结果的数组
template <class T> valarray<T> operator–(const valarray<T> ob, const T &v);	从ob的每个元素中减去v。返回包含结果的数组
template <class T> valarray<T> operator–(const T &v, const valarray<T> ob);	从v中减去ob的每个元素。返回包含结果的数组
template <class T> valarray<T> operator–(const valarray<T> ob1, const valarray<T> &ob2);	从其ob1的对应元素中，减去ob2中的每个元素。返回一个包含结果的数组
template <class T> valarray<T> operator*(const valarray<T> ob, const T &v);	把ob中的每个元素乘以v。返回一个包含结果的数组
template <class T> valarray<T> operator*(const T &v, const valarray<T> ob);	把ob中的每个元素乘以v。返回一个包含结果的数组
template <class T> valarray<T> operator*(const valarray<T> ob1, const valarray<T> &ob2);	用ob2中的元素乘以ob1中的对应元素。返回一个包含结果的数组
template <class T> valarray<T> operator/(const valarray<T> ob, const T &v);	用v除以ob中的每个元素。返回一个包含结果的数组
template <class T> valarray<T> operator/(const T &v, const valarray<T> ob);	用ob中的每个元素除以v。返回一个包含结果的数组
template <class T> valarray<T> operator/(const valarray<T> ob1, const valarray<T> &ob2);	用ob2中的每个元素除以ob1中的对应元素。返回一个包含结果的数组
template <class T> valarray<T> operator%(const valarray<T> ob, const T &v);	获得用v除以ob中的每个元素所得结果的余数。返回一个包含结果的数组
template <class T> valarray<T> operator%(const T &v, const valarray<T> ob);	获得用ob中的每个元素除以v所得结果的余数。返回一个包含结果的数组
template <class T> valarray<T> operator%(const valarray<T> ob1, const valarray<T> &ob2);	获得用ob2中的每个元素除以ob1中对应元素所得结果的余数。返回一个包含结果的数组
template <class T> valarray<T> operator^(const valarray<T> ob, const T &v);	把ob中的每个元素和v相“异或”（XOR）。返回一个包含结果的数组

（续表）

函数	说明
template <class T> valarray<T> operator^(const T &v, const valarray<T> ob);	把 ob 中的每个元素和 v 相“异或”（XOR）。返回一个包含结果的数组
template <class T> valarray<T> operator^(const valarray<T> ob1, const valarray<T> &ob2);	把ob1中的每个元素和ob2中的对应元素相“异或”（XOR）。返回一个包含结果的数组
template <class T> valarray<T> operator&(const valarray<T> ob, const T &v);	把 ob 中的每个元素和 v 进行“与”（AND）操作。返回一个包含结果的数组
template <class T> valarray<T> operator&(const T &v, const valarray<T> ob);	把 ob 中的每个元素和 v 进行“与”（AND）操作。返回一个包含结果的数组
template <class T> valarray<T> operator&(const valarray<T> ob1, const valarray<T> &ob2);	把 ob1 中的每个元素与其在 ob2 中的对应元素相“与”（AND）。返回一个包含结果的数组
template <class T> valarray<T> operator|(const valarray<T> ob, const T &v);	把 ob 中的每个元素和 v 相“或”（OR）。返回一个包含结果的数组
template <class T> valarray<T> operator|(const T &v, const valarray<T> ob);	把 ob 中的每个元素和 v 相“或”（OR）。返回一个包含结果的数组
template <class T> valarray<T> operator|(const valarray<T> ob1, const valarray<T> &ob2);	把ob1中的每个元素和其在ob2中的对应元素相“或”（OR）。返回一个包含结果的数组
template <class T> valarray<T> operator<<(const valarray<T> ob, const T &v);	把 ob 中的每个元素左移由 v 指定的位置数。返回一个包含结果的数组
template <class T> valarray<T> operator<<(const T &v, const valarray<T> ob);	把 v 左移由 ob 中的元素指定的位置数。返回一个包含结果的数组
template <class T> valarray<T> operator<<(const valarray<T> ob1, const valarray<T> &ob2);	把 ob1 中的每个元素左移由 ob2 中的对应元素指定的位置数。返回一个包含结果的数组
template <class T> valarray<T> operator>>(const valarray<T> ob, const T &v);	把 ob 中的每个元素右移 v 所指定的位置数。返回一个包含结果的数组
template <class T> valarray<T> operator>>(const T &v, const valarray<T> ob);	把 v 右移 ob 中的元素所指定的位置数。返回一个包含结果的数组
template <class T> valarray<T> operator>>(const valarray<T> ob1, const valarray<T> &ob2);	把ob1中的每个元素右移其在ob2中的对应元素所指定的位置数。返回一个包含结果的数组

（续表）

函数	说明
template <class T> valarray<bool> operator==(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] ==v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator==(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v ==ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator==(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] ==ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator!=(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] != v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator!=(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v != ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator!=(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] !=ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator<(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] < v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator<(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v < ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator<(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] <ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator<=(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] <= v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator<=(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v <= ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator<=(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] <=ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator>(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] > v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator>(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v > ob[i]。返回一个包含结果的布尔数组

（续表）

函数	说明
template <class T> valarray<bool> operator>(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] >ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator>=(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] >=v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator>=(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v >=ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator>=(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] >=ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator&&(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] &&v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator&&(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v &&ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator&&(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] &&ob2[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator||(const valarray<T> *ob*, const T &*v*);	对于每个i，执行ob[i] ||v。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator||(const T &*v*, const valarray<T> *ob*);	对于每个i，执行v ||ob[i]。返回一个包含结果的布尔数组
template <class T> valarray<bool> operator||(const valarray<T> *ob1*, const valarray<T> &*ob2*);	对于每个i，执行ob1[i] ||ob2[i]。返回一个包含结果的布尔数组

表37.4 为valarray定义的先验函数

函数	说明
template<class T> valarray<T> abs(const valarray<T> &*ob*);	获得ob中每个元素的绝对值，并返回一个包含结果的数组
template<class T> valarray<T> acos(const valarray<T> &*ob*);	获得ob中每个元素的反余弦，并返回一个包含结果的数组
template<class T> valarray<T> asin(const valarray<T> &*ob*);	获得ob中每个元素的反正弦，并返回一个包含结果的数组
template<class T> valarray<T> atan(const valarray<T> &*ob*);	获得ob中每个元素的反正切，并返回一个包含结果的数组
template<class T> valarray<T> atan2(const valarray<T> &*ob1*, const valarray<T> &*ob2*);	对于所有的i，获得ob1[i] / ob2[i]的反正切，并返回一个包含结果的数组

（续表）

函数	说明
template<class T> valarray<T> atan2(const T &v, const valarray<T> &ob);	对于所有的i，获得v / ob1[i]的反正切，并返回一个包含结果的数组
template<class T> valarray<T> atan2(const valarray<T> &ob, const T &v);	对于所有的i，获得ob1[i] / v 的反正切，并返回一个包含结果的数组
template<class T> valarray<T> cos(const valarray<T> &ob);	获得ob中每个元素的余弦，并返回一个包含结果的数组
template<class T> valarray<T> cosh(const valarray<T> &ob);	获得ob中每个元素的双曲余弦，并返回一个包含结果的数组
template<class T> valarray<T> exp(const valarray<T> &ob);	计算ob中每个元素的指数函数，并返回一个包含结果的数组
template<class T> valarray<T> log(const valarray<T> &ob);	获得ob中每个元素的自然对数，并返回一个包含结果的数组
template<class T> valarray<T> log10(const valarray<T> &ob);	获得ob中每个元素的对数，并返回一个包含结果的数组
template<class T> valarray<T> pow(const valarray<T> &ob1, const valarray<T> &ob2);	对于所有的i，计算ob1[i]ob2[i]，并返回一个包含结果的数组
template<class T> valarray<T> pow(const T &v, const valarray<T> &ob);	对于所有的i，计算vob[i]，并返回一个包含结果的数组
template<class T> valarray<T> pow(const valarray<T> &ob, const T &v);	对于所有的i，计算ob1[i]v ，并返回一个包含结果的数组
template<class T> valarray<T> sin(const valarray<T> &ob);	获得ob中每个元素的正弦，并返回一个包含结果的数组
template<class T> valarray<T> sinh(const valarray<T> &ob);	获得ob中每个元素的双曲正弦，并返回一个包含结果的数组
template<class T> valarray<T> sqrt(const valarray<T> &ob);	获得ob中每个元素的平方根，并返回一个包含结果的数组
template<class T> valarray<T> tan(const valarray<T> &ob);	获得ob中每个元素的正切，并返回一个包含结果的数组
template<class T> valarray<T> tanh(const valarray<T> &ob);	获得ob中每个元素的双曲正切，并返回一个包含结果的数组

下面的程序演示了valarray的一些功能。

```
// Demonstrate valarray
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;

int main()
{
  valarray<int> v(10);
  int i;
```

```
  for(i=0; i<10; i++) v[i] = i;

  cout << "Original contents: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  v = v.cshift(3);

  cout << "Shifted contents: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  valarray<bool> vb = v < 5;
  cout << "Those elements less than 5: ";
  for(i=0; i<10; i++)
    cout << vb[i] << " ";
  cout << endl << endl;

  valarray<double> fv(5);
  for(i=0; i<5; i++) fv[i] = (double) i;

  cout << "Original contents: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = sqrt(fv);

  cout << "Square roots: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv + fv;
  cout << "Double the square roots: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv - 10.0;
  cout << "After subtracting 10 from each element:\n";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  return 0;
}
```

它的输出如下所示：

```
Original contents: 0 1 2 3 4 5 6 7 8 9
Shifted contents: 3 4 5 6 7 8 9 0 1 2
Those elements less than 5: 1 1 0 0 0 0 0 1 1 1

Original contents: 0 1 2 3 4
Square roots: 0 1 1.41421 1.73205 2
```

```
Double the square roots: 0 2 2.82843 3.4641 4
After subtracting 10 from each element:
-10 -8 -7.17157 -6.5359 -6
```

37.2.1 slice和gslice类

<valarray>头文件定义了两个实用工具类，称为slice和gslice。这两个类封装了数组的一部分，它们和valarray子集的运算符[ ]一起使用。

slice类如下所示：

```
class slice {
public:
  slice();
  slice(size_t start, size_t len, size_t interval);
  size_t start() const;
  size_t size() const;
  size_t stride();
};
```

第一个构造函数创建一个空的部分（slice）。第二个构造函数创建一个规定了起始元素、元素数和元素间的间隔（即步进数）的部分。成员函数返回这些值。

下面是一个演示slice的程序。

```
// Demonstrate slice
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(10), result;
  unsigned int i;

  for(i=0; i<10; i++) v[i] = i;

  cout << "Contents of v: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[slice(0,5,2)];

  cout << "Contents of result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

这个程序的输出如下所示：

```
Contents of v: 0 1 2 3 4 5 6 7 8 9
Contents of result: 0 2 4 6 8
```

可以看到，结果数组由v的5个元素组成，在0处开始，步进数为2。

gslice类如下所示：

```
class gslice {
public:
  gslice();
  gslice()(size_t start, const valarray<size_t> &lens,
           const valarray<size_t> &intervals);
  size_t start() const;
  valarray<size_t> size() const;
  valarray<size_t> stride() const;
};
```

第一个构造函数创建一个空的部分。第二个构造函数创建一个规定了起始元素的部分，一个规定了元素数的数组和一个规定了元素间间隔（即步进数）的数组。长度数和间隔数必须相同。成员函数返回这些参数。我们使用这个类从valarray（它总是一维的）中创建一个多维数组。

下面的程序演示了gslice。

```
// Demonstrate gslice()
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(12), result;
  valarray<size_t> len(2), interval(2);
  unsigned int i;

  for(i=0; i<12; i++) v[i] = i;

  len[0] = 3; len[1] = 3;
  interval[0] = 2; interval[1] = 3;

  cout << "Contents of v: ";
  for(i=0; i<12; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[gslice(0,len,interval)];

  cout << "Contents of result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

输出如下所示：

```
Contents of v: 0 1 2 3 4 5 6 7 8 9 10 11
Contents of result: 0 3 6 2 5 8 4 7 10
```

37.2.2 帮助者类

数字类都依赖下面这些“帮助者”类，你的程序永远不会直接实例化这些类：

slice_array, gslice_array, indirect_array 和 mask_array

37.3 数字算法

头文件<numeric>定义了四种数字算法，可被用来处理容器的内容。下面分别讨论这四种算法。

37.3.1 accumulate

accumulate( )算法计算在一个指定范围内的所有元素的总和，并返回结果。它的原型如下：

```
template <class InIter, class T> T accumulate(InIter start, InIter end, T v);
template <class InIter, class T, class BinFunc>
        T accumulate(InIter start, InIter end, T v, BinFunc func);
```

其中，T是所操作的值。第一种形式计算在start到end这一范围所有元素的和。第二种形式对正在运行中的合计数应用func（即，func规定如何计算总和）。v提供运行中的合计所加的初值。

下面是一个演示accumulate( )的例子。

```
// Demonstrate accumulate()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5);
  int i, total;

  for(i=0; i<5; i++) v[i] = i;

  total = accumulate(v.begin(), v.end(), 0);

  cout << "Summation of v is: " << total;

  return 0;
}
```

生成的输出如下：

```
Summation of v is: 10
```

37.3.2 adjacent_difference

adjacent_difference( )算法产生一个新的序列，该序列中每个元素是初始序列中相邻元素的差（结果中的第一个元素与初始序列中的第一个元素相同）。adjacent_difference( )的原型如下所示：

```
template <class InIter, class OutIter>
  OutIter adjacent_difference(InIter start, InIter end, OutIter result);
template <class InIter, class OutIter, class BinFunc>
  OutIter adjacent_difference(InIter start, InIter end, OutIter result,
                              BinFunc func);
```

其中，start 和 end 是指向原始序列开始和结尾处的迭代器，结果序列存储在 result 所指的序列中。在第一种形式中，相邻元素被减，减的方法是用位置 n+1 处的元素中减去位置 n 处的元素。在第二种形式中，二元函数应用于邻接元素。返回指向 result 末尾的迭代器。

下面是一个使用 adjacent_difference( )的例子。

```
// Demonstrate adjacent_difference()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(10), r(10);
  int i;

  for(i=0; i<10; i++) v[i] = i*2;
  cout << "Original sequence: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  adjacent_difference(v.begin(), v.end(), r.begin());

  cout << "Resulting sequence: ";
  for(i=0; i<10; i++)
    cout << r[i] << " ";

  return 0;
}
```

生成的输出如下所示：

```
Original sequence: 0 2 4 6 8 10 12 14 16 18
Resulting sequence: 0 2 2 2 2 2 2 2 2 2
```

可以看到，结果序列包含邻接元素值之间的差。

37.3.3 inner_product

inner_product( )算法生成两个序列中相对应元素积的总和，并返回结果。它的原型如下：

```
template <class InIter1, class InIter2, class T>
   T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v);
template <class InIter1, class InIter2, class T, class BinFunc1, class BinFunc2>
   T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v,
                   BinFunc1 func2, BinFunc2 func2);
```

其中，start1 和 end1 是指向第一个序列开始和结尾处的迭代器，迭代器 start2 是指向第二个序列开始处的迭代器。值 v 提供一个初值，运行中的合计数与此初值相加。在第二种形式中，func1 指定一个决定如何计算合计值的二元函数，func2 指定一个决定如何把两个序列乘在一起的二元函数。

下面是一个演示 inner_product( )的程序。

```
// Demonstrate inner_product()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v1(5), v2(5);
  int i, total;

  for(i=0; i<5; i++) v1[i] = i;
  for(i=0; i<5; i++) v2[i] = i+2;

  total = inner_product(v1.begin(), v1.end(),
                        v2.begin(), 0);

  cout << "Inner product is: " << total;

  return 0;
}
```

输出如下：

```
Inner product is:50
```

37.3.4 partial_sum

partial_sum( )算法计算一个序列值的总和，并在其运行时把当前总计值放到新序列的每个后继元素中（即它创建一个序列，该序列是最初序列的一个运行中的总计值）。结果中的第一个元素与原始序列中的第一个元素相同。partial_sum( )的原型如下所示：

```
template <class InIter, class OutIter>
OutIter partial_sum(InIter start, InIter end, OutIter resuli);
template <class InIter, class OutIter, class BinFunc>
OutIter partial_sum(InIter start, InIter end, OutIter result,
                    BinFunc func);
```

其中，start和end是指向初始序列开始和末尾处的迭代器。迭代器result是指向所生成序列开始处的迭代器。在第二种形式中，func指定了一个二元函数，它决定了如何在运行过程中计算总数。

返回指向result末尾的迭代器。

下面是partial_sum( )的一个例子。

```
// Demonstrate partial_sum()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5), r(5);
  int i;
```

```
  for(i=0; i<5; i++) v[i] = i;
  cout << "Original sequence: ";
  for(i=0; i<5; i++)
    cout << v[i] << " ";
  cout << endl;

  partial_sum(v.begin(), v.end(), r.begin());

  cout << "Resulting sequence: ";
  for(i=0; i<5; i++)
    cout << r[i] << " ";

  return 0;
}
```

下面是它的输出：

```
Original sequence: 0 1 2 3 4
Resulting sequence: 0 1 3 6 10
```