12

数值
Numerics

本章讲述 C++ 标准程序库的数值相关组件，其中包括复数（complex）、数值数组（value arrays），以及从 C 标准程序库继承而来的全局数值函数。

C++ 标准程序库中，有两个数值组件在本书其它部分已经做过介绍：

1. STL 内含的数值算法，已在 9.11 节, p425 做过介绍。
2. 所有基本数值型别，各种与编译器相关（implementation specific）的表述方式已经在 4.3 节, p59 的 `numeric_limits` 一节讲过。

12.1 复数（Complex Numbers）

C++ 标准程序库提供了一个 template class `complex<>`，用于操作复数。让我们回顾一下，所谓复数就是由实部（real）和虚部（imaginary）组成的数值。虚部的特点是“其平方值为负数”。换言之复数虚部带着 `i`，其中 `i` 是 `-1` 的平方根。

Class `complex` 定义于头文件 `<complex>`：

```
#include <complex>
```

在其中，class `complex` 定义如下：

```
namespace std {
    template <class T>
    class complex;
}
```

其中 template 参数 `T` 被用来作为复数的实部和虚部的标量型别（scalar type）。

此外，C++ 标准程序库还为复数提供了针对标量型别 float,double,long double 的特化版本：

```
namespace std {
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;
}
```

通过这些特化版本，我们可以提供一些特别的优化措施，以及某些更安全的转换（从某个复数型别转换为另一个复数型别）。

12.1.1 Class Complex 运用实例

以下程序展示 class complex 的部分功能，诸如产生复数、以不同表示法打印复数、在复数中执行某些共同操作等等。

```
// num/complex1.cpp

#include <iostream>
#include <complex>
using namespace std;

int main()
{
    /* complex number with real and imaginary parts
    * - real part: 4.0
    * - imaginary part: 3.0
    */
    complex<double> c1(4.0,3.0);

    /* create complex number from polar coordinates
    * - magnitude: 5.0
    * - phase angle: 0.75
    */
    complex<float> c2(polar(5.0,0.75));

    // print complex numbers with real and imaginary parts
    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;
```

```
    // print complex numbers as polar coordinates
    cout << "c1: magnitude: " << abs(c1)
         << " (squared magnitude: " << norm(c1) << ") "
         << " phase angle: " << arg(c1) << endl;
    cout << "c2: magnitude: " << abs(c2)
         << " (squared magnitude: " << norm(c2) << ") "
         << " phase angle: " << arg(c2) << endl;

    // print complex conjugates
    cout << "c1 conjugated: " << conj(c1) << endl;
    cout << "c2 conjugated: " << conj(c2) << endl;

    // print result of a computation
    cout << "4.4 + c1 * 1.8: " << 4.4 + c1 * 1.8 << endl;

    /* print sum of c1 and c2:
    * - note: different types
    */
    cout << "c1 + c2: "
         << c1 + complex<double>(c2.real(),c2.imag()) << endl;

    // add square root of c1 to c1 and print the result
    cout << "c1 += sqrt(c1): " << (c1 += sqrt(c1)) << endl;
}
```

程序可能输出如下（确切的输出还得视“double 型别相关于编译器的某些性质”而定）：

```
c1: (4,3)
c2: (3.65844,3.40819)
c1: magnitude: 5 (squared magnitude: 25) phase angle: 0.643501
c2: magnitude: 5 (squared magnitude: 25) phase angle: 0.75
c1 conjugated:    (4,-3)
c2 conjugated:    (3.65844,-3.40819)
4.4 + c1 * 1.8:   (11.6,5.4)
c1 + c2:          (7.65844,6.40819)
c1 += sqrt(c1):   (6.12132,3.70711)
```

下面是第二个例子。其中有个循环，其内读取两个复数，并计算出以第一个复数为底、以第二个复数为指数的幂次方值（power）：

```
// num/complex2.cpp

#include <iostream>
#include <complex>
#include <cstdlib>
#include <limits>
using namespace std;

int main()
{
    complex<long double> c1, c2;

    while (cin.peek() != EOF) { // 译注：请注意“复数输入方式”的设计

        // read first complex number
        cout << "complex number c1: ";
        cin >> c1;
        if (!cin) {
            cerr << "input error" << endl;
            return EXIT_FAILURE;
        }

        // read second complex number
        cout << "complex number c2: ";
        cin >> c2;
        if (!cin) {
            cerr << "input error" << endl;
            return EXIT_FAILURE;
        }

        if (c1 == c2) {
            cout << "c1 and c2 are equal !" << endl;
        }

        cout << "c1 raised to the c2: " << pow(c1,c2)
             << endl << endl;

        // skip rest of line
        cin.ignore(numeric_limits<int>::max(),'\n');
    }
}
```

表 12.1 列出此程序可能的输入和输出。

表 12.1 complex2.cpp 例中的输入和输出（可能情况）

c1	c2	输出
2	2	c1 raised to c2: (4,0)
(16)	0.5	c1 raised to c2: (4,0)
(8,0)	0.333333333	c1 raised to c2: (2,0)
0.99	(5)	c1 raised to c2: (0.95099,0)
(0,2)	2	c1 raised to c2: (-4,4.89843e-16)
(1.7,0.3)	0	c1 raised to c2: (1,0)
(3,4)	(-4,3)	c1 raised to c2: (4.32424e-05,8.91396e-05)
(1.7,0.3)	(4.3,2.8)	c1 raised to c2: (-4.17622,4.86871)

注意，输入复数时，你可以在括号内（或是不需要括号）只写实部，也可以在括号内以逗号隔开实部和虚部。

12.1.2 复数的各种操作

template class complex 提供一系列操作，细节如下。

创建（Create），复制（Copy）和赋值（Assign）

表 12.2 列举了 complex 的构造函数和赋值操作。我们可透过构造函数传递初值的实部和虚部。如果不提供初值，则分别以实部和虚部标量型别的缺省构造函数进行初始化。

Assignment（赋值）操作符是改变既有复数的唯一途径。复合赋值操作符如 +=, -=, *=, /= 会对第一操作数和第二操作数进行加、减、乘、除等运算，并将结果储存于第一操作数内。

运用辅助函数 polar()，你可以采用极坐标（距原点距离和弧度（radians）相位角）来对一个复数进行初始化：

```
// create a complex number initialized from polar coordinates
std::complex<double> c2(std::polar(4.2,0.75));
```

如果有隐式型别转换，那么创建复数时会有一些问题。例如下面的写法没有问题：

```
std::complex<float> c2(std::polar(4.2,0.75)); // OK
```

可是，换用一个等号就不行了：

```
std::complex<float> c2 = std::polar(4.2,0.75); // ERROR
```

此问题将在下一小节讨论。

表 12.2 Class complex<> 的构造函数和赋值操作

表达式	效果
`complex c`	产生一个复数，实部和虚部都为零：(0 + 0i)。
`complex c(1.3)`	产生一个复数，实部为 1.3，虚部为 0：(1.3 + 0i)。
`complex c(1.3,4.2)`	产生一个复数，实部为 1.3，虚部为 4.2：(1.3 + 4.2i)。
`complex c1(c2)`	产生一个复数，是 c2 的一个副本。
`polar(4.2)`	产生一个极坐标表示法的临时复数，模（magnitude）为 4.2，相位角（phase angle）为 0。
`polar(4.2,0.75)`	产生一个极坐标表示法的临时复数，模（magnitude）为 4.2，相位角（phase angle）为 0.75。
`conj(c)`	产生一个临时复数，是 c 的共轭复数（实部相同而虚部相反）。
`c1 = c2`	将 c2 的值赋值给 c1
`c1 += c2`	将 c2 的值加入 c1
`c1 -= c2`	将 c1 的值减去 c2
`c1 *= c2`	将 c1 的值乘以 c2
`c1 /= c2`	将 c1 的值除以 c2

辅助函数 conj()用来协助我们以某个复数的共轭复数（conjugated complex）产生一个新复数。所谓共轭复数就是将原复数的虚部反相（negated）而后得到的复数：

```
std::complex<double> c1(1.1,5.5);
std::complex<double> c2(conj(c1));  // initialize c2 with
                                    // complex<double>(1.1,-5.5)
```

隐式型别转换（Implicit Type Conversions）

复数的 float,double,long double 等特化版本，遵循以下设计思想：允许安全转换（例如 complex<float> 转为 complex<double>）可以隐式进行，而不安全的转换（例如 complex<long double> 转为 complex<double>）必须显式进行（详见 p542 的声明细节）：

```
std::complex<float> cf;
std::complex<double> cd;
std::complex<long double> cld;
...
std::complex<double> cd1 = cf;       // OK: safe conversion
std::complex<double> cd2 = cld;      // ERROR: no implicit conversion
std::complex<double> cd3(cld);       // OK: explicit conversion
```

此外就再也没有其它构造函数可以执行“由他种类型之复数转换而来”的构造行为了。特别提示一点，你不能把一个“实部和虚部为整数”的复数，转换为“实部和虚部为 float 或 double 或 long double”的复数。不过你可以将实部和虚部分开来当做参数，进行相同意义的（转换）操作：

```
std::complex<double> cd;
std::complex<int> ci;
...
std::complex<double> cd4 = ci; // ERROR: no implicit conversion
std::complex<double> cd5(ci);  // ERROR: no explicit conversion
std::complex<double> cd6(ci.real(),ci.imag()); // OK
```

不幸的是，assignment（赋值）操作符允许接受不十分安全的转换——因为它们系以 template 函数的形式被提供出来，可接受任何型别。所以只要数值型别之间可以转换，你就可以赋值一个复数型别[1]：

```
std::complex<double> cd;
std::complex<long double> cld;
std::complex<int> ci;
...
cd = ci;     // OK
cd = cld;    // OK
```

这个问题也发生在 polar()及 conj()。例如下面的写法没有问题：

```
std::complex<float> c2(std::polar(4.2, 0.75));     // OK
```

可是下面的写法就不行：

```
std::complex<float> c2 = std::polar(4.2, 0.75);    // ERROR
```

因为以下表达式：

```
std::polar(4.2, 0.75)
```

[1] 复数特化版本的构造只允许安全隐式转换，而赋值操作却允许任意隐式转换，这可能是 *C++ Standard* 的一个失误。

产出一个临时的 complex<double>，但 C++ 标准程序库之内并未定义从 complex<double>到 complex<float>的隐式转换[2]。

数值的存取（Value Access）

表 12.3 列出各种复数属性的存取函数。

表 12.3 class complex<> 的各种属性（数值）的存取操作

表达式	效果
c.real()	返回实部值（这是一个成员函数）
real(c)	返回实部值（这是一个全局函数）
c.imag()	返回虚部值（这是一个成员函数）
imag(c)	返回虚部值（这是一个全局函数）
abs(c)	返回 c 的绝对值（$\sqrt{c.real\ ()^2 + c.imag\ ()^2}$）
norm(c)	返回 c 绝对值的平方（c.real()^2 + c.imag()^2）
arg(c)	返回 c 的极坐标相位角（φ，相当于 atan2(c.imag(),c.real())）

注意，real()和 imag()只提供读取实部和虚部的能力。如果你只是想改变实部或虚部，仍然必须赋值一个完整复数。例如下面语句将 c 的虚部设定为 3.7：

```
std::complex<double> c;
...
c = std::complex<double>(c.real(),3.7);
```

---

[2] 在

```
X x;
Y y(x); // explicit conversion
```

和

```
X x;
Y y = x; // implicit conversion
```

之间有轻微的不同。前者使用显式转换，从型别 X 产生一个型别 Y 的新对象。后者使用隐式转换，产生一个型别 Y 的新对象。

比较（Comparison）

复数之间的比较很方便，直接检查相等性就行了，如表 12.4。operator== 和 operator!= 被定义为全局函数，如此一来两个操作数之中就可以有一个为标量（scalar value）。如果你使用一个标量作为操作数，它会被解释为复数的实部，相应的虚部则以虚部标量型别的默认构造函数产生出来（通常是 0）。

表 12.4 class complex<> 定义的比较操作

表达式	效果
c1 == c2	判断是否 c1 等于 c2 (c1.real()==c2.real() && c1.imag()==c2.imag())
c == 1.7	判断是否 c1 等于 1.7 (c.real()==1.7 && c.imag()==0.0)
1.7 == c	判断是否 c1 等于 1.7 (c.real()==1.7 && c.imag()==0.0)
c1 != c2	判断是否 c1 和 c2 不同 (c1.real()!=c2.real() || c1.imag()!=c2.imag())
c != 1.7	判断是否 c1 不等于 1.7 (c.real()!=1.7 || c.imag()!=0.0)
1.7 != c	判断是否 c1 不等于 1.7 (c.real()!=1.7 || c.imag()!=0.0)

其它的比较操作，例如 operator<，并未被定义出来。为复数定义顺序关系，虽然不是不可能，但不直观，也没什么用。例如复数的大小（magnitude）就不是很好的排序依据，因为两个复数可能大小相同，但非常不一样（1 和 -1 就是例子）。你可以加上一个特别规则以产生合理顺序，例如面对两个复数 c1 和 c2，你可以认为当 |c1| < |c2| 时 c1 < c2，如果两者大小（magnitude）相同则 arg(c1) < arg(c2) 时视为 c1 < c2。不过，这些规则基本上没什么数学意义[3]。

因此，你不能在关联式容器中以 complex 作为元素型别（如果你没有自行定义排序准则的话），因为关联式容器需要对元素进行排序，需要用到函数对象（仿函数）less<>，而后者会调用 operator<（详见 5.10.1 节，p134）。

不过如果你自行定义了一个 operator<，就可以对复数进行排序，并可以在关联式容器中使用复数。注意，请小心，不要污染了标准命名空间（standard namespace）。例如：

[3] 感谢 David Vandevoorde 指出这一点。

```
template <class T>
bool operator< (const std::complex<T>& c1,
                const std::complex<T>& c2)
{
   return std::abs(c1)<std::abs(c2) ||
          (std::abs(c1)==std::abs(c2) &&
           std::arg(c1)<std::arg(c2));
}
```

算术运算

复数支持四种基本运算，以及正负号，见表 12.5。

表 12.5 class complex<> 的算术操作

表达式	效果
c1 + c2	返回 c1 与 c2 的和
c + 1.7	返回 c1 与 1.7 的和
1.7 + c	返回 1.7 与 c1 的和
c1 - c2	返回 c1 与 c2 的差
c - 1.7	返回 c1 与 1.7 的差
c1 * c2	返回 c1 与 c2 的乘积
c * 1.7	返回 c1 与 1.7 的乘积
1.7 * c	返回 c1 与 1.7 的乘积
c1 / c2	返回 c1 与 c2 的商
c / 1.7	返回 c1 与 1.7 的商
1.7 / c	返回 1.7 与 c1 的商
-c	返回 c 的反相（negated value）
+c	返回 c 本身
c1 += c2	等同于 c1 = c1 + c2
c1 -= c2	等同于 c1 = c1 - c2
c1 *= c2	等同于 c1 = c1 * c2
c1 /= c2	等同于 c1 = c1 / c2

输入/输出

Class complex 提供了一般的 I/O 操作符 operator<< 和 operator>>，如表 12.6。

表 12.6 class complex<> 的 I/O 操作

表达式	效果
strm << c	将复数 c 写入 ostream strm 中
strm >> c	从 istream strm 中读取复数 c

output 操作符根据 stream 的当前状态和格式，将复数写出：

```
(realpart,imagpart)
```

它的定义相当于：

```
template <class T, class charT, class traits>
std::basic_ostream<charT,traits>&
operator<< (std::basic_ostream<charT,traits>& strm,
            const std::complex<T>& c)
{
   // temporary value string to do the output with one argument
   std::basic_ostringstream<charT,traits> s;

   s.flags(strm.flags());           // copy stream flags
   s.imbue(strm.getloc());          // copy stream locale
   s.precision(strm.precision());   // copy stream precision

   // prepare the value string
   s << '(' << c.real() << ',' << c.imag() << ')';

   // write the value string
   strm << s.str();

   return strm;
}
```

input 操作符可以接纳下面任何一种格式，从中读取一个复数：

```
(realpart,imagpart)
(realpart)
realpart
```

如果 input stream 内的下一个字符不符合上述所有格式，则设立 ios::failbit，并且可能抛出相应的异常（参见 13.4.4 节，p602）。

可惜的是，你不能设定复数表示式中的实部和虚部间的分隔符。所以如果有的国家以逗号作为小数点（例如德国），I/O 看上去就十分奇异了。一个实部为 4.6，虚部为 2.7 的复数，输出结果是这样：

```
(4,6,2,7)
```

I/O 操作的运用，详见 p532。

超越函数（Transcendental Functions）

表 12.7 列出 complex 的所有超越函数（三角函数、指数等等）。

表 12.7 Class complex<> 的超越函数（Transcendental Functions）

表达式	效果
pow(c,3)	计算幂次方数 c^3
pow(c,1.7)	计算幂次方数 $c^{1.7}$
pow(c1,c2)	计算幂次方数 $c1^{c2}$
pow(1.7,c)	计算幂次方数 1.7^c
exp(c)	计算以 e 为底，c 为指数的幂次方数（e^c）
sqrt(c)	计算 c 的平方根（$\sqrt{c}$）
log(c)	计算 c 的自然对数（ln c）
log10(c)	计算以 10 为底的 c 的对数（lg c）
sin(c)	计算 c 的正弦值（sin c）
cos(c)	计算 c 的余弦值（cos c）
tan(c)	计算 c 的正切值（tan c）
sinh(c)	计算 c 的双曲正弦值（sinh c）
cosh(c)	计算 c 的双曲余弦值（cosh c）
tanh(c)	计算 c 的双曲正切值（tanh c）

12.1.3 Class complex<> 细部讨论

本节详细探讨 class complex<> 的所有操作函数。以下所有定义中，T 是 class complex<> 的 template 参数，也就是复数的实部和虚部的标量型别。

型别定义

complex::value_type

- 实部和虚部的标量型别

构造、复制、赋值

complex::complex ()

- 缺省构造函数
- 构造一个复数，其中实部和虚部的初值系透过调用实部和虚部的缺省构造函数设定。所以如果是基本型别，实部和虚部的初值为 0（参见 p14 对于基本型别默认值的说明）。

complex::complex (const T& *re*)

- 构造一个复数，实部为 *re*，虚部则透过调用其缺省构造函数设定（基本型别的初值为 0）。
- 此构造函数同时定义了一个从 T 到 complex 的隐式型别转换。

complex::complex (const T& *re*, const T& *im*)

- 构造一个复数，实部初值为 re，虚部初值为 im。

complex **polar** (const T& *rho*)
complex **polar** (const T& *rho*, const T& *theta*)

- 以上两种形式都产生并返回一个复数，其初值以极坐标形式来设定。
- *rho* 是大小（magnitude）。
- *theta* 是以弧度（radians）为单位的相位角（缺省为 0）。

complex **conj** (const *complex*& *cmplx*)

- 产生并返回一个复数；以复数 *cmplx* 的共轭复数为初值。所谓共轭复数是指虚部与原复数的虚部互为反相。

complex::complex (const *complex*& *cmplx*)

- copy 构造函数
- 产生一个新的复数，成为 *cmplx* 的复本。

- 复制实部和虚部。
- 此函数通常同时供应 non-template 和 template 两种形式（参见 p11 对 member templates 的介绍）。因此具备对元素型别的自动转型能力。
- 然而，float,double,long double 等复数特化版本，对于 copy 构造函数有所限制，所以不安全的转换（例如从 double 和 long double 转为 float，或是从 long double 转为 double）就必须显式进行，并且不允许有其它的“元素转型”行为。

```
namespace std {
    template<> class complex<float> {
        public:
        explicit complex(const complex<double>&);
        explicit complex(const complex<long double>&);
        // no other kinds of copy constructors
        ...
    };
    template<> class complex<double> {
        public:
        complex(const complex<float>&);
        explicit complex(const complex<long double>&);
        // no other kinds of copy constructors
        ...
    };
    template<> class complex<long double> {
        public:
        complex(const complex<float>&);
        complex(const complex<double>&);
        // no other kinds of copy constructors
        ...
    };
}
```

关于其确切意义，请参考 p534。

complex& **complex::operator** = (const *complex*& *cmplx*)

- 将复数 *cmplx* 赋值给 *this
- 返回*this
- 此函数通常同时供应 non-template 和 template 两种形式（参见 p11 对 member templates 的介绍）。因此具备对元素型别的自动型别转换能力。（对于 C++ 标准程序库提供的特化版本，这一点也成立）。

```
complex& complex::operator += (const complex& cmplx)
complex& complex::operator -= (const complex& cmplx)
complex& complex::operator *= (const complex& cmplx)
complex& complex::operator /= (const complex& cmplx)
```

- 上述操作分别对*this 和 cmplx 进行加、减、乘、除运算，并将结果存入*this
- 返回 *this
- 此函数通常同时供应 non-template 和 template 两种形式（参见 p11 对 member templates 的介绍）。因此具备对元素型别的自动型别转换能力。（对于 C++ 标准程序库提供的特化版本，这一点也成立）。

注意，赋值操作符是改变既有 complex 的唯一途径。

元素存取

```
T complex::real () const
T real (const complex& cmplx)
T complex::imag () const
T imag (const complex& cmplx)
```

- 上述函数分别返回实部和虚部。
- 注意，返回值并不是一个 reference，所以你不能运用这些函数来改变复数的实部和虚部。若要单独改变实部或虚部，必须赋予一个新的复数值（参见 p536）。

```
T abs (const complex& cmplx)
```

- 返回 cmplx 的绝对值（模，magnitude）。
- 绝对值计算公式：$\sqrt{cmplx.real()^2 + cmplx.imag()^2}$

```
T norm (const complex& cmplx)
```

- 返回 cmplx 绝对值的平方。
- 计算公式：cmplx.real()^2 + cmplx.imag()^2

```
T arg (const complex& cmplx)
```

- 返回以弧度（radians）为单位的极坐标相位角（φ）
- 相位角计算方法：atan2(cmplx.imag(),cmplx.real())

I/O 操作

ostream& **operator** << (*ostream*& *strm*, const *complex*& *cmplx*)

- 将 *cmplx* 的值以(realpart, imagpart)的格式写入 stream。
- 返回 *strm*。
- 此操作的具体行为见 p539。

istream& **operator** >> (*istream*& *strm*, *complex*& *cmplx*)

- 从 *strm* 中将一个新值读至 *cmplx*。
- 合法的输入格式是：

```
(realpart,imagpart)
(realpart)
realpart
```

- 返回 *strm*。
- 此操作的具体行为请见 p539。

操作符（Operators）

complex **operator** + (const *complex*& *cmplx*)

- 正号。
- 返回 *cmplx*。

complex **operator** - (const *complex*& *cmplx*)

- 负号。
- 将复数 *cmplx* 的实部和虚部都取反相（negated）。

complex ***binary-op*** (const *complex*& *cmplx1*, const *complex*& *cmplx2*)
complex ***binary-op*** (const *complex*& *cmplx*, const T& *value*)
complex ***binary-op*** (const T& *value*, const *complex*& *cmplx*)

- 上述各项操作返回 ***binary-op*** 计算所得的复数。
- 这里的 ***binary-op*** 可以是以下四种运算之一：

```
operator +
operator -
operator *
operator /
```

- 如果传入一个元素型别的标量值（scalar value），它会被视为一个复数的实部，虚部则由其标量型别的缺省初值决定（如果是基本型别，初值为 0）。

```
bool comparison (const complex& cmplx1, const complex& cmplx2)
bool comparison (const complex& cmplx, const T& value)
bool comparison (const T& value, const complex& cmplx)
```

- 返回两个复数的比较结果，或是一个复数与一个标量（scalar value）的比较结果。
- 这里的 *comparison* 可以是下面两种运算之一：

```
operator ==
operator !=
```

- 如果传入一个元素型别的标量值（scalar value），它会被视为一个复数的实部，虚部则由其型别的缺省初值决定（如果是基本型别，初值为 0）。
- 注意，并没有定义 <，<=，>，>= 等等操作符。

超越函数（Transcendental Functions）

```
complex pow (const complex& base, int exp)
complex pow (const complex& base, const T& exp)
complex pow (const complex& base, const complex& exp)
complex pow (const T& base, const complex& exp)
```

- 上述所有形式都是计算“以 *base* 为基底，*exp* 为指数”的幂次方数，定义为 `exp(exp*log(base))`。
- branch cuts 沿着负实数轴进行。
- `pow(0, 0)`的结果由实作版本（implementations）自行定义。

```
complex exp (const complex& cmplx)
```

- 返回“以 e 为基底，*cmplx* 为指数”的幂次方结果。

```
complex sqrt (const complex& cmplx)
```

- 返回位于右半象限的 *cmplx* 平方根
- 如果参数是负实数，则运算结果位于正虚数轴上。
- branch cuts 沿着负实数轴进行。

```
complex log (const complex& cmplx)
```

- 返回 *cmplx* 的自然对数（亦即以 e 为底的对数）。
- 当 *cmplx* 是负实数时，`imag(log(cmplx))`的值为 π（pi）。
- branch cuts 系沿着负实数轴进行。

```
complex log10 (const complex& cmplx)
```

- 返回 *cmplx* 的（以 10 为基底的）对数。
- 相当于 `log(cmplx)/log(10)`。
- branch cuts 系沿着负实数轴进行。

```
complex sin (const complex& cmplx)
complex cos (const complex& cmplx)
complex tan (const complex& cmplx)
complex sinh (const complex& cmplx)
complex cosh (const complex& cmplx)
complex tanh (const complex& cmplx)
```

- 以上各操作函数分别对 *cmplx* 进行复数三角运算（trigonometric operations）。