CS 1502H 程序设计思想与方法 (C++)
Chapter 12: 运算符重载

陈东尧
上海交通大学
2025年秋季

本章架构

- 什么是运算符重载
- 几个特殊的运算符重载
- 自动类型转换

回顾：复数类

•定义一个复数类，而复数的**虚部和实部都用有理数**表示，实现复数的加法和输出

方案：利用已有的有理数类，数据成员是两个有理数类对象

```
class Rational {
        int num;
        int den;
        void ReductFraction();
public:
        Rational(int a = 0, int b = 1);
        void add(const Rational &r1, const Rational &r2);
        void multi(const Rational &r1, const Rational &r2);
        void display() ;
};
```

类定义

```
class Complex{
  private:
            Rational real;      //实部
            Rational imag;      //虚部

   public:
      Complex(int r1 = 0, int r2 = 1, int i1= 0, int i2 = 1) :  real(r1, r2), imag(i1, i2) { }
      Complex( Rational r1,  Rational r2 ) :  real(r1), imag(r2) { }
      void add(const  Complex  &c1,  const  Complex  &c2)
      {
          real.add(c1.real, c2.real);
          imag.add(c1.imag, c2.imag);
       }
      void display() {    real.display() ; cout << '+';   imag.display() ; cout << 'i'; }
};
```

类定义

```
class Complex{
  private:
            Rational real;      //实部
            Rational imag;      //虚部

   public:
      Complex(int r1 = 0, int r2 = 1, int i1= 0, int i2 = 1) :  real(r1, r2), imag(i1, i2) { }
      Complex( Rational r1,  Rational r2 ) :  real(r1), imag(r2) { }
      void add(const  Complex  &c1,  const  Complex  &c2)
      {
          real.add(c1.real, c2.real);
          imag.add(c1.imag, c2.imag);
       }
      void display() {    real.display() ; cout << '+';   imag.display() ; cout << 'i'; }
};
```

复数类的使用

```
int main()
{
    Rational r1(1, 2), r2(3,4);
    Complex x1(2,3,4,5) , x2(r1, r2),  x3;

    x3.add( x1 ,  x2);
    x3.dispaly();

    return 0;
}
```

能否有更自然的表示法?
比如**x3=x1+x2**

什么是运算符重载

- **使系统内置的运算符可以用于类类型**

例如：类A的对象a1、a2、a3，可以执行

```
a3 = a1 + a2;
```

运算符重载的限制：

- 不是所有的运算符都能重载
- 重载不能改变运算符的优先级和结合性
- 重载不能改变运算符的操作数个数
- 不能创建新的运算符

运算符重载的方法

- **运算符重载就是写一个函数解释某个运算符在某个类中的含义**
- **关键问题**

  如何使编译器能自动找到重载的这个函数

运算符重载的方法

- **运算符重载就是写一个函数解释某个运算符在某个类中的含义**
- **关键问题**
- 如何使编译器能**自动找到**重载的这个函数
- **解决方案**
- 让**函数名体现出**和某个被重载的运算符的联系
- 重载函数命名

  operator@

其中，@为要重载的运算符

重载“+”运算符，该重载函数名为operator+

重载赋值运算符，函数名为operator=

函数原型

运算符的重载不能改变运算符的运算对象数

形式参数个数（包括成员函数的隐式指针this）及类型

与运算符的运算对象相符

返回值

与运算结果值类型一致

函数原型

运算符的重载不能改变运算符的运算对象数

形式参数个数（包括成员函数的隐式指针this）及类型
与运算符的运算对象相符

返回值
与运算结果值类型一致

重载函数可以是成员函数也可以是全局函数

- **重载为全局函数**
  函数原型与运算符与完全相符
  最好将此函数设为友元函数（为了访问私有成员）

- **重载为成员函数**
  形式参数个数比运算符的运算对象数少1。这是因为成员函数有一个隐含的参数this
  在C++中，把隐含参数this作为运算符的第一个参数。

重载实例

- 为rational类增加“+”和“*”以及“==”用于比较的重载函数，用以替换现有的add和multi函数

方案一：重载成成员函数

```
class Rational {
private:
    int num;
    int den;
    void ReductFraction();
public:
    Rational(int n = 0, int d = 1)
      {
          num = n;
          den = d;
      }
     Rational operator+(const  Rational &r1) const;
     Rational operator*(const  Rational &r1) const;
    bool operator==(const  Rational &r1) const;
    void display()  const {
          cout << num << '/' << den;
     }
} ;
```

operator可以看做是前一个对象的成员函数

函数实现

```
Rational Rational::operator+(const   Rational  &r1) const
{
    Rational tmp;
    tmp.num = num * r1.den + r1.num * den;
    tmp.den = den * r1.den;
    tmp.ReductFraction();
    return tmp;
}

Rational Rational::operator*(const   Rational  &r1) const
{
    Rational tmp;
    tmp.num = num * r1.num;
    tmp.den = den * r1.den;
    tmp.ReductFraction();
    return tmp;
}

bool Rational::operator==(const   Rational  &r1) const
{   return num == r1.num && den == r1.den;  }
```

operator可以看做是前一个对象的成员函数

方案二：重载成友元函数（特点为有2个操作数）

```
class Rational {
      friend Rational operator+(const  Rational &r1, const Rational &r2);
      friend Rational operator*(const  Rational &r1 , const Rational &r2);
      friend bool operator==(const  Rational &r1 , const Rational &r2);
 private:
   int num;
   int den;
   void ReductFraction();
 public:
   Rational(int n = 0, int d = 1) { num = n; den = d; }
   void display()  const  {  cout  <<  num  <<  '/  '  <<  den;  }
};
```

方案二函数的实现(在类的定义之外)

```
Rational operator+(const Rational &r1, const Rational &r2)
{
  Rational tmp;
  tmp.num = r1.num * r2.den + r2.num * r1.den;
  tmp.den = r1.den * r2.den;
  tmp.ReductFraction();  return tmp;
}

Rational operator*(const Rational &r1, const Rational &r2)
{
  Rational tmp;
  tmp.num = r1.num * r2.num;
  tmp.den = r1.den * r2.den;
  tmp.ReductFraction();  return tmp;
}
```

重载后有理数类的使用

```
int main()
{
    Rational r1(1,6), r2(1,6), r3;

    r3 = r1 + r2;
    r1.display(); cout << " + "; r2.display();
    cout << " = "; r3.display(); cout << endl;
    r3 = r1 * r2;
    r1.display(); cout << " * "; r2.display();
    cout << " = "; r3.display(); cout << endl;

    return 0;
}
```

重载后有理数类的使用

```
int main()
{
    Rational r1(1,6), r2(1,6), r3;

    r3 = r1 + r2;
    r1.display(); cout << " + "; r2.display();
    cout << " = "; r3.display(); cout << endl;
    r3 = r1 * r2;
    r1.display(); cout << " * "; r2.display();
    cout << " = "; r3.display(); cout << endl;

    return 0;
}
```

成员函数，单一操作数：**r1.operator+(r2)**

友元函数，两个操作数：**operator+(r1,r2)**

全局函数 VS成员函数

大多数运算符都可以重载成成员函数或全局函数

必须重载为成员函数

建议重载为成员函数

建议重载为全局函数

全局函数 VS成员函数

大多数运算符都可以重载成成员函数或全局函数

必须重载为成员函数

赋值（=）、下标（[]）、函数调用（（））和成员访问（->）

建议重载为成员函数

建议重载为全局函数

全局函数 vs成员函数

大多数运算符都可以重载成成员函数或全局函数

须重载为成员函数

赋值（=）、下标（[]）、函数调用（（））和成员访问（->）

建议重载为成员函数

只有一个操作数（unary operator）

建议重载为全局函数

全局函数 VS成员函数

大多数运算符都可以重载成成员函数或全局函数

须重载为成员函数

赋值（=）、下标（[]）、函数调用（（））和成员访问（->）

建议重载为成员函数

只有一个操作数（unary operator）

建议重载为全局函数

具有两个操作数（binary operator）的运算符最好这样可以使得应用更加灵活

本章架构

- 什么是运算符重载
- 几个特殊的运算符重载
- 自动类型转换

赋值运算符重载

- **缺省的赋值运算符函数**
- 如果设计者没有自定义赋值运算符函数，编译器为其生成一个
- 在对应的数据成员间赋值

赋值运算符重载

•缺省的赋值运算符函数

•如果设计者没有自定义赋值运算符函数，编译器为其生成一个

•在对应的数据成员间赋值

•缺省的赋值运算符重载函数的问题

•当类含有类型为指针的数据成员时，会带来一些麻烦（为什么？）

•如：对DoubleArray类对象执行array1 = array2

赋值运算符重载

•缺省的赋值运算符函数

•如果设计者没有自定义赋值运算符函数，编译器为其生成一个

•在对应的数据成员间赋值

•缺省的赋值运算符重载函数的问题

•当类含有类型为指针的数据成员时，会带来一些麻烦（为什么？）

•如：对DoubleArray类对象执行array1 = array2

•回忆拷贝构造函数中我们做了什么？（查阅类-第二讲的内容）

回顾：拷贝构造函数 --- DoubleArray类的拷贝构造函数

```
DoubleArray(const DoubleArray &arr)
{   low = arr.low;
    high = arr.high;
    storage = new double [high – low + 1];
    for (int i = 0; i < high –low + 1; ++i)
        storage[i] = arr.storage[i];
}
```

回顾：拷贝构造函数

- 一般情况下，默认的拷贝构造函数足以满足要求，但需要结合具体使用场景。

  例如：希望对DoubleArray类能定义一个和另一个数组完全一样的数组。

  DoubleArray arr2(10,30)，arr1(arr2);

- 默认拷贝构造函数执行：

  arr1.low = arr2.low;　arr1.high = arr2.high;　arr1.storage = arr2.storage;

  操作中，storage是一个指针，意味着使arr1的storage指针和arr2的storage指针指向同一块区间。

回顾：拷贝构造函数 --- 使用同一块空间存在什么问题?

- 一个对象的修改将会影响另一个对象
- 如果两个对象的作用域不同，当一个对象析构时，另一个对象也将丧失它的空间

回顾：拷贝构造函数 --- DoubleArray类的拷贝构造函数

```
DoubleArray(const DoubleArray &arr)
{   low = arr.low;
    high = arr.high;
    storage = new double [high – low + 1];
    for (int i = 0; i < high –low + 1; ++i)
        storage[i] = arr.storage[i];
}
```

赋值运算符“=”的原型

- **赋值运算符只能重载为成员函数**
- **函数原型**

```
X  &X::operator=(const X &source)
{
  // 赋值过程
}
```

DoubleArray类的赋值运算符重载函数

```
DoubleArray &DoubleArray::operator= (const DoubleArray &right)
{
   if (this == &right)
         return *this;
   delete [ ] storage;
   low = right.low;
   high = right.high;
   storage = new double[high - low + 1];
   for (int i=0; i <= high - low; ++i)
        storage[i] = right.storage[i];        //复制数组元素

   return *this;
}
```

处理重复赋值的情况

赋值运算符重载和拷贝构造函数

- 一般来讲，需要拷贝构造函数的类也需要重载赋值运算符
- 定义对象时给对象赋初值调用的是拷贝构造函数
- 程序的语句部分中的赋值表达式调用的是赋值运算符重载函数

下标运算符重载

下标运算符

- 二元运算符，第一个运算数是数组名，第二个运算数是下标值
- 必须用引用返回
- 下标运算符必须重载成成员函数

意义

使DoubleArray那样的类的对象当做数组名，可以用下标变量的形式访问

一元运算符（Unary Operator）是指**只需1个操作数**就能进行运算的符号
二元运算符（Binary Operator）是指**需要2个操作数**来进行运算的符号

DoubleArray类的[ ]重载

另一个操作数为默认的this指针

引用返回，因为下标符号需要为左值

```
DoubleArray & DoubleArray::operator[](int index)
{
    if (index < low || index > high) {
        cout << "下标越界";
        exit(-1);
    }
    return storage[index - low];
}
```

DoubleArray类的使用

- **定义对象**
- DoubleArray array(20, 30);

- **数组输入**

```
for ( i=20;  i<=30;  ++i ) {
        cout << "请输入第" << i << "个元素:";
        cin >> array[i];
}
```

array.storage[i-low]

- **数组输出**

```
for ( i=20;  i<=30; ++i)
        cout << array[i] << '\t';
```

函数调用运算符

函数调用运算符（）

- 是一个**二元运算符**（binary operator）
- 它的第一个运算对象是函数名，第二个参数是形式参数表
- 运算的结果是函数的返回值

函数调用function call运算符

函数调用运算符（）

- 是一个二元运算符（binary operator）
- 它的第一个运算对象是函数名，第二个参数是形式参数表
- 运算的结果是函数的返回值

作用

- 把类的对象当做函数名来使用

参考：https://www.geeksforgeeks.org/cpp/overloading-of-function-call-operator-in-cpp/

函数调用运算符重载

函数调用运算符须重载为成员函数

重载函数的原型

- 函数的返回值　operator() (形式参数表);

函数调用运算符重载实例

- 在DoubleArray类增加一个功能：取数组中的一部分元素形成一个新的数组
- 例如，在一个下标范围为10到20的数组arr中取出下标为第12到15的元素，形成一个下标范围为2到5的数组存放在数组arr1中，可以调用

```
arr1 = arr(12, 15, 2)
```

函数调用运算符重载实例

```
DoubleArray operator()( int start，int end，int lh)
{
   if (start > end || start < low || end > high ) {
         cout << "下标越界";
         exit(-1);
       }

   DoubleArray tmp(lh, lh + end - start);

   for (int i = 0; i < end - start + 1; ++i)
         tmp.storage[i] = storage[start + i - low];

   return tmp;
}
```

++和 -- 重载

++和 -- 重载

- **重载＋＋、－－时遇到的问题**
- 这两个操作符可以是前缀，也可以是后缀
- 前缀和后缀的含义是有区别的。所以，须有两个重载函数
- 两个重载函数有相同的原型

Pre-increment: 前置++
Post-increment : 后置++
Pre-decrement : 前置--
Post-decrement : 后置--

参考：https://www.geeksforgeeks.org/cpp/increment-and-decrement-operator-overloading-in-c/

++和 -- 重载

- **重载＋＋、－－时遇到的问题**
- 这两个操作符可以是前缀，也可以是后缀
- 前缀和后缀的含义是有区别的。所以，须有两个重载函数
- 两个重载函数有相同的原型

- 前缀：一元操作符
- 后缀：二元操作符，认为增加一个整型参数

“++”和“--”重载

- **成员函数重载**

++ob重载为：类名 &ob.operator++( )

ob-- 重载为：类名 ob.operator--(int)

- **友元函数重载**

++ob重载为：类名 & operator++(X &ob)

ob--重载为：类名 operator--(X &ob, int)

++、--重载实例

•在有理数类中增加前缀和后缀的++和--操作，分别实现对有理数加1和减1。

前缀和后缀的含义与内置类型相同。

```
Rational &Rational::operator++()            // 前缀++
{
    num += den;
    return *this;
}
```

```
Rational &Rational::operator++()          // 前缀++
{
    num += den;
    return *this;
}
```

```
Rational Rational::operator++(int)        // 后缀++
{
    Rational tmp = *this;
    num += den;
    return tmp;
}

Rational &Rational::operator--()          // 前缀--
{
   num -= den;
   return *this;
}

Rational Rational::operator--(int)        // 后缀--
{
   Rational tmp = *this;
   num -= den;
   return tmp;
}
```

关于后缀中的形参int

在考虑重载++和--运算符时，必须注意一个问题。作为内置运算符，++和--有两种用法，它们既可以作为前缀使用，也可以作为后缀使用。而且这两种用法的结果是不一样的：作为前缀使用时，返回的是修改以后的对象值；而作为后缀使用时，返回的是修改以前的对象值。为了与内置类型一致，重载后的++和--处理前缀

重载自增自减运算符

we need two different function definitions to distinguish between them. This is achieved by passing a **dummy int parameter in the postfix version**

但问题是，处理++的两个重载函数的形式参数个数和返回类型是完全相同的，处理--的两个重载函数的形式参数个数和返回类型也是完全相同的。普通的函数重载无法区分这两个函数。

为了解决这个问题，C++规定后缀运算符重载函数接受一个额外的（即无用的）int 型的形式参数。使用后缀运算符时，编译器用 0 作为这个参数的值。当编译器看到一个前缀表示的++或--时，调用正常重载的这个函数。如果看到的是一个后缀表示的++或--，则调用有一个额外参数的重载函数。这样就把前缀和后缀的重载区分开了。

```cpp
#include<iostream>
using namespace std;
//1. Q: 为什么前缀++要作为返回引用？A: 用于作为左值使用
//2. Q: 为什么后缀++要有int作为形参？A: int在这里作为区分用，和前缀分开
class Age{
    private:
        int i;
    public:
        Age& operator++() //前置++
        {
            ++i;
            return *this;
        }
        Age operator++(int) //后置++
        {
            Age tmp = *this; //通过拷贝值传递
            ++i;
            return tmp;
        }
        Age& operator=(int i) //赋值操作
        {
            this->i = i;
            return *this;
        }
};
```

代码演示

overloading_increment_decrement.cpp

```cpp
#include<iostream>
using namespace std;
//1. Q: 为什么前缀++要作为返回引用？A: 用于作为左值使用
//2. Q: 为什么后缀++要有int作为形参？A: int在这里作为区分用，和前缀分开
class Age{
    private:
        int i;
    public:
        Age& operator++() //前置++
        {
            ++i;
            return *this;
        }
        Age operator++(int) //后置++
        {
            Age tmp = *this; //通过拷贝值传递
            ++i;
            return tmp;
        }
        Age& operator=(int i) //赋值操作
        {
            this->i = i;
            return *this;
        }
};
```

代码演示

overloading_increment_decrement.cpp

该形参也可以有实际用处，详见👇

代码演示

overloading_increment_decrement_expand.cpp

本章架构

- 什么是运算符重载
- 几个特殊的运算符重载
- 自动类型转换

自定义类型转换运算符

class类型能否和其他的class类型或内置类型互相转换?

内置类型到class类型的转换

- **利用构造函数进行转换**
- 需要类有一个构造函数，它只有一个参数，是某个内置类型
- 例如，对于Rational类的对象 r，如果执行r=2
  - 此时，我们要隐式地调用Rational的构造函数构造一个num=2，den= 1的Rational类的对象

类型转换函数

- **重载成成员函数**
- **格式**

```
operator 目标类型名 ( ) const
{       …
   return (结果为目标类型的表达式);
}
```

- **类型转换函数的特点**
- 无参数，无返回值
- 是const函数

Rational类到double的转换

转换函数的定义

```
operator double ( ) const
{
    return (double(num)/den);
}
```

应用

- 如r是有理数，值为（1，3），x是double型变量
- 可以执行x = r
- 执行后，x的值为0.333333

经过运算符重载后的DoubleArray类

希望包含的功能：

1. 赋值运算符重载
2. 下标运算符可以兼容不同的下标范围
3. 输出的时候可以自动先输出文字提示
4. 输入的时候可以自动先给出输入要求
5. 判断是否相等
6. 括号可以自动取数组中的一部分元素形成一个新的数组

经过运算符重载后的DoubleArray类

希望包含的功能：

1. 赋值运算符重载
   ➔ =重载
2. 下标运算符可以兼容不同的下标范围
   ➔ []重载
3. 输出的时候可以自动先输出文字提示
   ➔ <<重载
4. 输入的时候可以自动先给出输入要求
   ➔ >>重载
5. 判断是否相等
   ➔ ==重载
6. 括号可以自动取数组中的一部分元素形成一个新的数组
   ➔ ()重载

经过运算符重载后的DoubleArray类 — 头文件

```
#ifndef _array_h
#define _array_h
#include<iostream>
using namespace std;

class DoubleArray{
        friend ostream &operator<<(ostream &os,   const DoubleArray &obj);
        friend istream &operator>>(istream &is, DoubleArray &obj);
        friend bool operator==(const DoubleArray &obj1,  const DoubleArray &obj2);
private:
    int low;
    int high;
    double *storage;
public:
     DoubleArray(int lh = 0, int rh = 0): low(lh),  high(rh) {  storage = new double [high - low + 1];  }
     DoubleArray(const  DoubleArray  &arr);
     DoubleArray &operator=(const  DoubleArray  &right);
     double  & operator[ ](int index);
     const double & operator[ ](int index) const;
     DoubleArray  operator( )(int start,  int end,  int lh);
     ~DoubleArray()    {  delete [ ] storage;   }
};
#endif
```

代码演示

rational-overloading.h

rational-overloading.cpp

rational-main.cpp

(需联合编译)

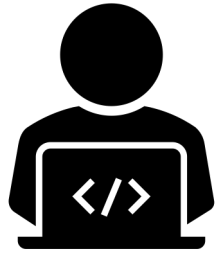
经过运算符重载后的DoubleArray类 — 头文件

```
#ifndef _array_h
#define _array_h
#include<iostream>
using namespace std;

class DoubleArray{
        friend ostream &operator<<(ostream &os,   const DoubleArray &obj);
        friend istream &operator>>(istream &is, DoubleArray &obj);
        friend bool operator==(const DoubleArray &obj1,  const DoubleArray &obj2);
private:
    int low;
    int high;
    double *storage;
public:
     DoubleArray(int lh = 0, int rh = 0): low(lh),  high(rh) {  storage = new double [high - low + 1];  }
     DoubleArray(const  DoubleArray  &arr);
     DoubleArray &operator=(const  DoubleArray  &right);
     double  & operator[ ](int index);
     const double & operator[ ](int index) const;
     DoubleArray  operator( )(int start,  int end,  int lh);
     ~DoubleArray()    {  delete [ ] storage;   }
};
#endif
```

1. 赋值运算符重载➔ =重载
2. 下标运算符可以兼容不同的下标范围 ➔ []重载
3. 输出的时候可以自动先输出文字提示➔ >>重载
4. 输入的时候可以自动先给出输入要求➔ <<重载
5. 判断是否相等➔ ==重载
6. 括号可以自动取数组中的一部分元素形成一个新的数组➔ ()重载

DoubleArray.cpp

```
//文件名：DoubleArray.cpp
//DoubleArray类的实现
#include <cassert>
#include "DoubleArray.h“

DoubleArray::DoubleArray(const  DoubleArray  &arr)
{
      low = arr.low;
      high = arr.high;
      storage = new double [high - low + 1];
      for (int i = 0; i < high -low + 1; ++i)
          storage[i] = arr.storage[i];
}
```



代码演示

rational-overloading.h
rational-overloading.cpp
rational-main.cpp
(需联合编译)

operator=

```
DoubleArray &DoubleArray::operator= (const  DoubleArray  & a)
{
    if (this == &a) return *this;

    delete [] storage;
    low = a.low;  high = a.high;
    storage = new double[high - low + 1];
    for (int i=0; i <= high - low; ++i)
       storage[i] = a.storage[i];

    return *this;
}
```

operator[]

```
double  & DoubleArray::operator[ ](int index)
{
     assert( index >= low && index <= high);
     cout <<  “ lvalue”  << endl;
     return storage[index - low];
}

const  double  & DoubleArray::operator[ ]( int index ) const
{
    assert( index >= low && index <= high );
    cout <<  “rvalue”  << endl;
    return storage[index - low];
}
```

operator<<

```
ostream  &operator<<(ostream  &os,  const DoubleArray  &obj)
{
    os << "数组内容为：  \n";
    for ( int i=obj.low;  i<=obj.high;  ++i)
        os << obj[i] << '\t';
     os << endl;
     return os;
}
```

operator>>

```
istream  &operator>>( istream  &is,  DoubleArray  &obj)
{
      cout << "请输入数组元素[" << obj.low   << ", " << obj.high << "]:\n";
      for ( int  i=obj.low;  i<=obj.high ; ++i)
           is >> obj[i] ;
       return  is;
}
```

operator==

```
bool operator==(const  DoubleArray  &obj1，const DoubleArray  &obj2)
{
    if ( obj1.low != obj2.low  || obj1.high != obj2.high )
        return false;
    for ( int  i = obj1.low;  i<=obj1.high;  ++i)
        if (obj1[i] != obj2[i])   return false;
    return true;
}
```

operator()

```
DoubleArray  DoubleArray::operator( )  ( int start,  int end,  int lh)
{
   assert (start <= end && start >= low  && end <= high );

   DoubleArray tmp(lh, lh + end - start);

   for (int i = 0; i < end - start + 1; ++i)
            tmp.storage[i] = storage[start + i - low];

   return tmp;
}
```

Main函数

```
int main()
{
   DoubleArray array1(20,30), array2;
   cin >> array1;
   cout << "array1 "; cout << array1;
   array2 = array1;
   cout << "执行 array2 = array1, array2 " << array2;
   cout << "array1 == array2 是 “ << ((array1 == array2) ? "true" : "false") << endl;
   array2[25] = 0;
   cout << "执行array[25] = 0后, array1 == array2 是 “
        << ((array1 == array2) ? "true" : "false") << endl;
   array2 = array1(22, 25, 2);
   cout << "执行array2 = array1(22, 25, 2)后, array2 的值为: “ << array2;
   return 0;
}
```

Main函数

```
int main()
{
    DoubleArray array1(20,30), array2;
    cin >> array1;
    cout << "array1 "; cout << array1;
    array2 = array1;
    cout << "执行 array2 = array1, array2 " << array2;
    cout << "array1 == array2 是 “ << ((array1 == array2) ? "true" : "false") << endl;
    array2[25] = 0;
    cout << "执行array[25] = 0后, array1 == array2 是 “
         << ((array1 == array2) ? "true" : "false") << endl;
    array2 = array1(22, 25, 2);
    cout << "执行array2 = array1(22, 25, 2)后, array2 的值为: “ << array2;
    return 0;
}
```

```
请输入数组元素[20，30]:
1 2 3 4 5 6 7 8 9 10 11
array1的内容为:
1  2  3  4  5  6  7  8  9  10  11
执行 array2 = array1，array2的内容为:
1  2  3  4  5  6  7  8  9  10  11
array1 == array2是true
lvalue
执行array2[25] = 0后，array1 == array2是false
执行array2 = array1(22, 25, 2)后, array2 的值为: 3  4  5  6
```

执行结果

请输入数组元素[20，30]:
1 2 3 4 5 6 7 8 9 10 11
array1的内容为:
1 2 3 4 5 6 7 8 9 10 11
执行 array2 = array1，array2的内容为:
1 2 3 4 5 6 7 8 9 10 11
array1 == array2是true
lvalue
执行array2[25] = 0后，array1 == array2是false
执行array2 = array1(22, 25, 2)后, array2 的值为: 3 4 5 6

小结

- 运算符重载的作用
- 如何选择用成员函数或全局函数
- 如何写一个重载函数
- 介绍了一种区分++和—的前后缀应用的方法
- 通过运算符重载实现类类型和内置类型及其他类类型之间的转换