《面向对象程序设计》朋辈课程 · 第九辑

运算符重载

信息学院 赵家宇

PART 01

运算符重载的引入

回顾：运算符

- 运算符的分类包括：
  - 算术运算符（+、-、×、/、%）
  - 关系运算符（==、>、<）
  - 逻辑运算符（&、|、&&、||）
  - 赋值运算符（=）
- 运算符三大要素：
  - 运算符优先级
  - 运算对象类型和个数
  - 运算符执行过程

回顾：运算符

- 程序设计语言提供了一系列对数据进行操作的操作符，可以支持不同类型数据的操作。例如运算符+可以进行int+int或是float+float甚至是int+double、char+int；运算符/能对整数、单精度数和双精度数进行除法运算，但是表现出来的结果不尽相同。
- 从效果上看，对于用户而言，同一个运算符，能够支持不同类型数据的操作，这实际就是运算符的重载。

回顾：重载

- 经常有此类现象，有着不同的函数名字的多个函数其功能是大致相同的。给这些功能相同的函数起一个名字，只是它们各自的函数体不同，对应着不同的类型的数据；
- C++中引进重载函数，也即允许同一个函数名对应着不同的实现，即各自有自己的函数体；函数重载就是对一个已有的函数赋予新的含义，使之实现新的功能，即一个函数名可以用来代表不同功能的函数。

运算符重载

- 运算符也可以重载，即对已有的运算符赋予多重含义，同一个运算符作用于不同类型的数据导致不同类型的行为；因此在定义类时，希望用户在对对象的操作也能使用传统的运算符。
- 作为用户，我们曾遇见运算符重载的问题，在讨论和字符串字面值的连接中：

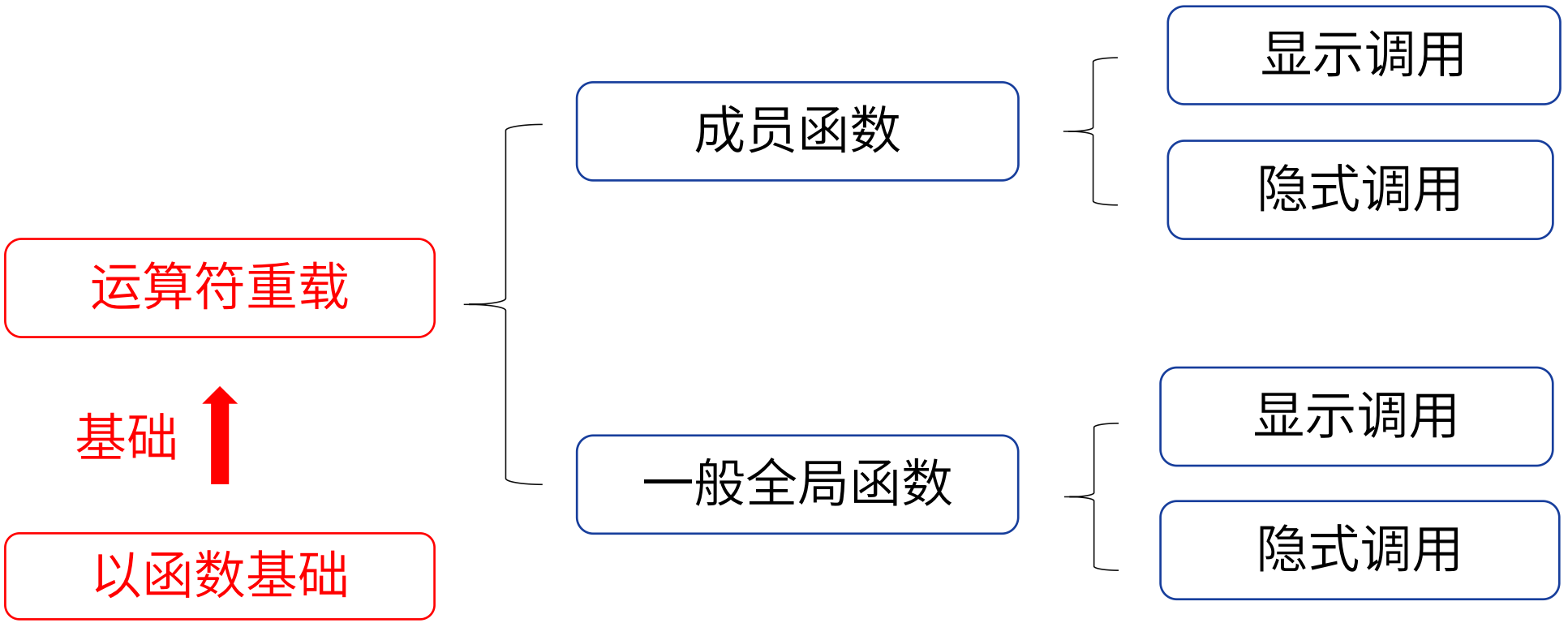
```
char* s1="hello" + ", "  //错误，char[]不支持"+"
string s2="hello " + "world"
/*正确，string通过对"+"运算符重载，且前两部分运算后结果为string，正是它的存在能唤起它调用string类的"+"（而不是调用整型的+），因此能与"world"再做"+"*/
```

PART 02

通过函数实现运算符重载

运算符重载的基本思路

运行机理　　函数声明和定义　　Main函数中应用



用函数实现运算符重载

- 运算符函数的一般格式如下：

  函数类型 operator 运算符名称(形参列表) { 对运算符的重载处理}

- operator是关键字，专用于定义重载运算符函数；“operator 运算符”是函数名，若对于使用字母字符的运算符（例如new等），在operator和运算符之间至少有一个空格，对于其他运算符，空格是可选的；

- 函数与运算符之间对应关系：执行含有重载后的运算符的表达式，系统就会调用“operator 运算符”函数，而操作数是函数的参数，操作结果是函数返回值；

运算符重载的基本思路

运行机理　　函数声明和定义　　Main函数中应用

运算符重载

基础

以函数基础

成员函数
- 显示调用
- 隐式调用

一般全局函数
- 显示调用
- 隐式调用

成员运算符函数

- 在C++中,可以把运算符函数定义成某个类的成员函数,称为成员运算符函数。
- 成员运算符函数的原型在类的内部声明格式和在类外定义成员运算符函数的格式如右；调用成员函数的对象是运算符的一个运算对象。

```
class X {
    //…
    返回类型 operator 运算符(形参表);
}
返回类型 X::operator 运算符(形参表)
{
    //…
}
```

成员运算符函数

```
Complex Complex::operator+(const Complex &x) const{
        Complex temp;
        temp.real=real+x.real;
        temp.imag=imag+x.imag;
        return temp;
}
Complex a(10,20);
Complex b(a);
Complex sum;
sum = a.operator +(b);   //通过函数调用来使用运算符
sum = a+b; //通过+运算，由于操作数是类对象，找寻对应函数
```

- 首先把指定的运算表达式转化为对运算符函数的调用，运算目标转化为运算符函数的实参，然后根据实参的类型来确定需要调用的函数，这个过程是在编译过程中完成的；由于是类对象，则调用重载的函数。

成员运算符函数

- 友元运算符函数的原型在类的内部声明和在类外定义友元运算符函数的格式如右；
- 根据运算符所需要的运算对象，考虑参数格式。

```
class X {
        //…
friend 返回类型 operator运算符(形参表);
     //…
}
返回类型 operator运算符(形参表)
{
       函数体
}
根据运算符所需要的运算对象，考虑参数格式
```

成员运算符函数

```
Class Complex {
   friend Complex operator+ (Complex &x, Complex &y); };
complex operator+ (Complex &x, Complex &y) {
    Complex temp;
    temp.real=x.real+y.real;
    temp.imag=x.imag+y.imag;
    return temp;
}
Complex a(10,20);
Complex b(a);
Complex sum;
sum = a.operator +(b);  //通过函数调用来使用运算符
sum = a+b; //通过+运算，由于操作数是类对象，找寻对应函数
```

- 首先把指定的运算表达式转化为对运算符函数的调用，运算目标转化为运算符函数的实参，然后根据实参的类型来确定需要调用的函数，这个过程是在编译过程中完成的；由于是类对象，则调用重载的函数。

友元运算符函数

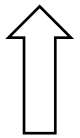
```
bool operator<(const Box& aBox) const {
    return volume() < aBox.volume();
}
 //重载函数定义在类中,且调用其他成员函数
```

```
bool operator<(const Box& aBox,const Box& bBox){
    return aBox.volume() < bBox.volume();
}
//重载运算符函数为一般函数
```

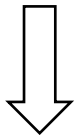
比较下，函数定义之间究竟有什么差别

```
if (box1.operator(box2))

 //翻译成函数调用
```



```
 if (box1<box2) {
       ….
   } //直接使用<运算符
```



```
If(operator(box1,box2))
       ….
   } //翻译成函数调用
```

成员函数视角：双目运算符重载

- 对双目运算符而言，成员运算符函数的形参表中仅有一个参数，它作为运算符的右操作数，此时当前对象作为运算符的左操作数，它是通过this指针隐含地传递给函数的。例如：box1<box2;
- 一般而言,如果在类X中采用成员函数重载双目运算符@，成员运算符函数operator@所需的一个操作数由对象aa通过this指针隐含地传递，它的另一个操作数bb在参数表中显示,aa和bb是类X的两个对象，则以下两种函数调用方法是等价的：

```
aa @ bb;              // 隐式调用   box1<box2
aa.operator @(bb);    // 显式调用   box1.operator<box2
```

成员函数视角：单目运算符重载

- 对单目运算符而言,成员运算符函数的参数表中没有参数,此时当前对象作为运算符的一个操作数，例如：-c1;
- 成员运算符函数operator @所需的一个操作数由对象aa通过this指针隐含地传递。因此,在它的参数表中没有参数。一般而言，采用成员函数重载单目运算符时，以下两种方法是等价的：

```
@aa;                    // 隐式调用
aa.operator@();         // 显式调用
```

友元视角：双目运算符重载

- 当用友元函数重载双目运算符时,两个操作数都要传递给运算符函数。

```
bool operator<(const Box& aBox,const Box& bBox)  {
    return aBox.volume()<bBox.volume();
 } //重载运算符函数为一般函数
```

- 调用形式：一般而言,如果在类X中采用友元函数重载双目运算符@,而aa和bb是类X的两个对象,则以下两种函数调用方法是等价的:

```
@aa;                    // 隐式调用
operator@(aa);      // 显式调用
```

成员运算符函数与友元运算符函数的比较

- 对双目运算符而言，成员运算符函数带有一个参数，而友元运算符函数带有两个参数；对单目运算符而言，成员运算符函数不带参数，而友元运算符函数带一个参数。
- 成员运算符函数和友元运算符函数可以用习惯方式调用，也可以用它们专用的方式调用。
- C++的大部分运算符既可说明为成员运算符函数，又可说明为友元运算符函数。究竟选择哪一种运算符好一些，没有定论，这主要取决于实际情况和程序员的习惯。

成员函数与一般函数的抉择

- 之前的基本<运算符重载为比较同类型的两个数据元素，例如对Box1<25.0或是10<Box2型表达式的重载；
- 现在比较自定义类型对象（第一个操作数）和double类型的第二个操作数（重载为成员函数）

```
bool operator<(double aValue) const {
    return volume()<aValue;
 }  //Box对象传送给函数，作为隐式的this指针，double则传送为一个参数
```

成员函数与一般函数的抉择

- 再比较double类型（第一个操作数）和自定义类型对象（第二个操作数）；成员运算符函数总是把this指针提供为作操作数，而在这个表达式中，左操作数为double型，因此不能把运算符实现为成员函数，应该实现为全局运算符函数或友元函数。

```
bool operator<(const double aValue,const Box& aBox) {
    return aValue< aBox.volume();
 }
```

重载运算符的规则

- C++不允许用户自己定义（发明）新的运算符，只能对已有的C++运算符进行重载；重载时，不能修改运算符的优先级、结合性和操作数的个数；
- C++不能重载的运算符有：

.	成员访问运算符	.*	成员指针访问运算符
::	域运算符	sizeof	长度运算符
?:	条件运算符		

- 重载运算符的函数不能有默认的参数；重载的运算符必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象。

PART 03

几种重要的运算符重载

赋值运算符=的重载

- 赋值运算符能把给定类型的对象复制到同一类型的另一个对象中，如果没有提供重载的赋值运算符来复制类的对象，编译器会提供默认版本

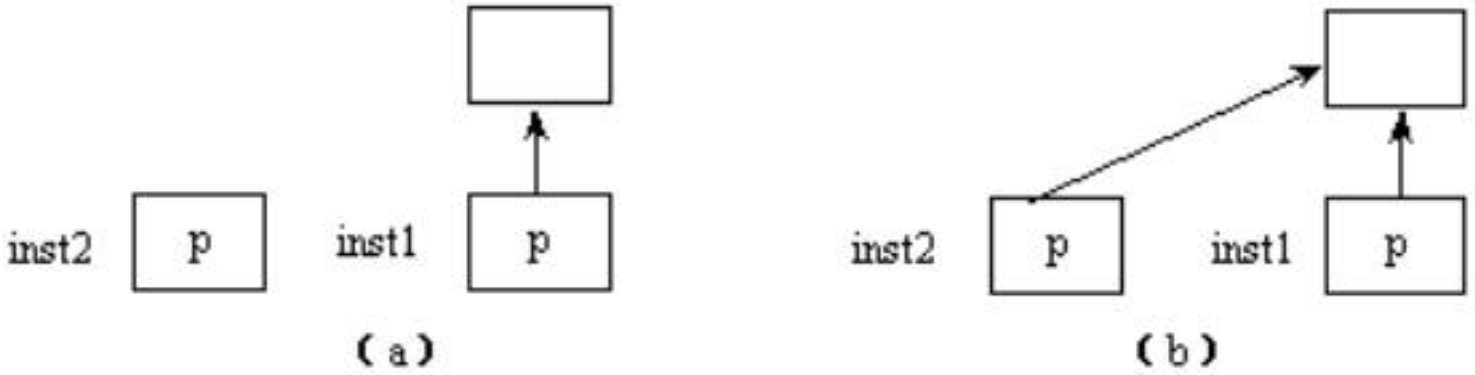
  operator=( );

```
Cube& operator=(const Cube& aBox) {
      side=aBox.side;  //实现对各个成员的赋值操作
      return *this; //this指针指向当前对象，return返回的就是当前对象
   }  //调用形式就是box1=box2;
```

- operator=( )的参数是一个常对象引用，避免对原对象的修改;
- operator=( )的返回类型为对象的引用(引用,可避免不必要的复制操作)

赋值运算符=的重载

- 对于本体与实体不一致的情况，若没有自定义重载赋值运算符，会出现两个对象相互依赖的现象，即：两个对象共享样本对象中自定义的存储区。



赋值运算符=的重载

```
Cube& operator= (const Cube& aBox) {
      side=aBox.side;  //实现对各个成员的赋值操作
      return *this; //this指针指向当前对象，return返回的就是当前对象
 }  //调用形式就是box1=box2;
```

- const意味什么，&意味什么?
- 实参不需要地址&，形参使用时不要*，引用没有占用地址。

赋值运算符=的重载

```
Cube& operator=(const Cube& aBox) {
      side=aBox.side;  //实现对各个成员的赋值操作
      return *this; //this指针指向当前对象，return返回的就是当前对象
   }  //调用形式就是box1=box2;
```

- 为何不是void?

  使用void可以实现load1=load2的基本功能；但是出现load1=load2=load3时，由于load1=load2=load3等价load1=(load2=load3)，假设成员函数operator=()，它等价于load1.operator=(load2.opertaor=(load3))；

- 可以看出，从operator=()返回的内容是另一个operator=()调用的参数，此时返回值为空不合适。

赋值运算符=的重载

```
Complex Complex::operator+=(const Complex& aBox) {
    Complex temp;
    real+=c.real;  image+=c.image;
    temp= *this;
    return temp;
}//有问题
```

- 为何返回时是对象引用而非对象?
- 其核心在于：函数返回时，是否在调用点有临时对象生成。若要返回对象，则需要用返回的对象值初始化内存临时对象（调用拷贝构造函数），内存临时对象作为调用函数的结果。由于返回对象值需要调用拷贝构造函数初始化内存临时对象，因此若对象有动态分配共建，则需定义拷贝构造函数。

赋值运算符=的重载

```
Complex& Complex::operator+=(const Complex& aBox) {
    Complex temp;
    real+=c.real;  image+=c.image; //实现对各个成员的赋值操作
    temp= *this;
    return temp;
}//有问题
```

- 若是返回本类对象（非*this）的引用，会如何?
- 返回的是对象的引用，系统要求执行流程返回到调用点时返回值是存在的。而temp的auto特性导致其在返回前会被撤销，因此产生问题。

赋值运算符=的重载

- 函数必须返回一个对象，该对象就是函数的左操作数；而且，如果为了避免不必要的复制操作，返回类型必须是该对象的引用；
- 赋值操作符无论形参为何种类型，赋值操作符必须定义为成员函数；如果是一般函数（友元），会导致没有*this从而无法正常返回。

流输入输出运算符的重载

- C++的输入输出都是流来处理的。
- 编译系统在类库中提供：
  - 输入流类istream：管输入的流　　有对象：cin
  - 输出流类ostream：管输出的流　　有对象：cout
- 流插入运算符“<<”和流提取运算符“>>”是在C++类库中提供的，可使用cout<<和cin>>对标准类型数据进行输入输出；

流输入输出运算符的重载

- 问题在于：用户自己定义的类型的对象，是不能直接用<<和>>来输出和输入的。可以通过写一个display或是myprint的成员函数来实现；而更好地方法是通过标准输入输出流进行输出。
- 如果想用标准输入输出流输出和输入自己声明的类型的数据，必须对它们重载；可以发现，运算符重载函数可以以display或是myprint为基础（函数体功能类似）

流输入输出运算符的重载

- 程序中，用户可以自己定义一个ostream的对象的引用s，s可以使用运算符“<<”.

```
#include <iostream>
int main( ) {
    int num=10;
    ostream & scout=cout;   //类对象的引用
    scout<<num<<endl;
    return 0;
}
```

流输入输出运算符的重载

- 定义形式：

```
istream & operator >> (istream &,自定义类 &);
ostream & operator << (ostream &,自定义类 &);
```

- 重载运算符“<<”和“>>”的函数的第一个参数和函数类型都必须是*stream &类型，第二个参数是要进行输入操作的类；只能将重载“<<”和“>>”的函数作为友元函数或普通函数，而不能将它们定义为成员函数；

流输入输出运算符的重载

- 定义形式：

```
istream & operator >> (istream &,自定义类 &);
ostream & operator << (ostream &,自定义类 &);
```

- 返回为何是引用：当需要将运算结果作为左值时，用引用返回cin<<a<<b;
- 参数为何是引用：用户自己定义一个ostream的对象的引用s，则s也可以使用运算符“<<”.

流输入输出运算符的重载

```
#include <iostream>
class complex {
  public:
    complex operator + ( complex &c2);
    friend ostream & operator << (ostream&, complex&);
  private: double real, image;
};
complex complex::operator+ (complex &c2){
    return complex (real+c2.real, image+c2.image);
}
ostream& operator<< (ostream& output, complex& c){
    output<<"("<<c.real<<"+"<<c.image<<"i)"<<endl;
    return output;
}
```

- 重载流插入运算符<<，我们希望“<<”运算符不仅能输出标准数据类型，而且能输出用户自己定义的类对象。
- 调用形式：
  - 隐式调用：cout<<obj;
  - 显示调用：operator<<(cout, Class_obj)

流输入输出运算符的重载

```
#include <iostream>
class complex {
  public:
    friend istream & operator >> (istream&, complex&);
    friend ostream & operator << (ostream&, complex&);
  private: double real, image;
};
istream& operator >> (istream& input, complex& c){
    cout<<“ 请输入复数的实数部分和虚数部分 ”<<endl;
    input>>c.real>>c. image;
    return input;
}
ostream& operator<< (ostream& output, complex& c){
    output<<“(”<<c.real<<“+”<<c.image<<“i)”<<endl;
    return output;
}
```

- 重载流提取运算符>>，我们希望“>>”运算符不仅能输入标准数据类型，而且能输入用户自己定义的类对象。
- 调用形式：
  - 隐式调用：cin>>obj;
  - 显示调用：operator>>(cin, Class_obj)

算术运算符的重载

- “+”运算符，是一个二元运算符，且涉及到创建并返回新对象；

```
inline Box Box::operator+(const Box& aBox) const {
   return box (length>aBox.length?length:aBox.length,
   breadth>aBox.breadth?breadth:aBox.breadth,height+aBox.height);
}//定义为成员函数，且产生一个无名的对象，用值来初始化
```

```
inline Box operator+(const Box& aBox,const Box& bBox) {
    return box (aBox.length>bBox.length?aBox.length:bBox.length,
    aBox.breadth>bBox.breadth?breadth:bBox.breadth,aBox.height+bBox.height);
}   //函数体中，Box对象的尺寸通过公共的成员函数来访问
    //定义为一般函数，且产生一个无名的对象，用值来初始化
```

算术运算符的重载

- +=运算符重载，其作用为给左操作数*this加上右操作数，修改了左操作；
- 函数返回值涉及赋值，需要返回一个引用；函数主体中使用在加号运算符中把Box对象加在一起的方法；

```
Box& Box::operator+= (const Box& right) {
    length = length>right.length?length:right.length;
    breadth = breadth>right.breadth?breadth:right.breadth;
    height += right.height;
    return *this;  // 涉及赋值
}
```

算术运算符的重载

- 可以使用一个运算符实现另外一个运算符，例如可以由+=实现+:

```
Box& Box::operator+= (const Box& right) {
    length = length>right.length?length:right.length;
    breadth = breadth>right.breadth?breadth:right.breadth;
    height += right.height;
    return *this;
}
Box Box::operator+(const Box& aBox) const {
    return Box(*this)+=aBox;
} // Box(*this)调用拷贝构造函数
```

算术运算符的重载

- 重载单目运算符中的++和--，只有一个操作数；++和--运算符有两种使用方式：前置自增运算符和后置自增运算符，在重载时如何区分?
- 解决：在自增（自减）运算符重载函数中，增加一个int型虚拟形参，就是后置自增运算符函数。

```
class Object {
  public:
    Object & operator++ ( );  //前置 ++a
    const Object operator++( int ) //后置 a++
};
```

算术运算符的重载

- 前缀形式的返回类型一般是当前对象递增后的引用；后缀形式，先在修改之前创建原对象的副本，再将执行递增后的原对象返回。后缀运算符的返回值常声明为const。

```
class Object {
  public:
    Object & operator++ ( );  //前置 ++a
    const Object operator++( int ) //后置 a++
};
```

算术运算符的重载

```
class Increase {
    public:
        Increase & operator++();              //前增量
        Increase operator++(int);             //后增量
    private: int value;
};
Increase & Increase::operator++() {
    value++;                                  //先增量
        return *this;                         //再返回原对象
}
Increase Increase::operator++(int) {
    Increase temp(*this);    //临时对象存放原有对象值
        value++;             //原有对象增量修改
        return temp;         //返回原有对象值
}
```

```
void main() {
   Increase n(20);
   n.display();
   (n++).display(); //显示临时对象值
   n.display();     //显示原有对象
    ++n;
   n.display();
   ++(++n);
    n.display();
    (n++)++;//第二次操作对临时对象进行
   n.display();
}
```

此程序的运行结果为:

20 20 21 22 24 25

算术运算符的重载

```
class Increase {
   public:
      friend Increase & operator++(Increase & );
      friend Increase operator++(Increase &,int);
   private: int value;
};
Increase & operator++(Increase & a) {
    a.value++;                    //前增量
    return a;                     //再返回原对象
} //前增量，由于要修改第一操作数，形参必须声明为引用
Increase operator++(Increase& a, int) {
    Increase temp(a); //通过拷贝构造函数保存原有对象值
    a.value++;          //原有对象增量修改
    return temp;        //返回原有对象值
}
```

```
void main() {
   Increase n(20);
   n.display();
   (n++).display(); //显示临时对象值
   n.display();     //显示原有对象
    ++n;
   n.display();
   ++(++n);
    n.display();
   (n++)++;//第二次操作对临时对象进行
   n.display();
}
此程序的运行结果为:
20 20 21 22 24 25
```

算术运算符的重载

- “( )”“[ ]”不能用友元函数重载，只能采用成员函数重载。
- 重载函数调用运算符( )，对应的运算符重载函数为operator( )(…);
- 设xobj为类X的一个对象，则表达式xobj(arg1,arg2)可被解释为

  xobj.operator( )(arg1,arg2)

- 重载下标运算符[ ]：掌握连续空间下，可判断下标是否超界；对应的运算符重载函数为operator[ ](…)。
- 设xobj为类X的一个对象，则表达式xobj[arg]可被解释为

  xobj.operator[](arg)

算术运算符的重载

```
class Array {
  int *p; int size;
  public:
     Array(int num) { p=new int[(num>6)?num:6];}
     ~Array( ){ delete[] p; }
     int & operator[](int idx) {
         if(idx>=size) expend(idx-size+1);
         return p[idx]; }
     }//p[idx]非对象,[ ]是之前下标
     void expend(int offset){
         int *pi = new int[size+offset];
         for(int num=0;num<size;num++) pi[num]=p[num];
         delete []p; p=pi; size+=offset;
     }
     void contract(int offset) { size=size-offest; }
};
```

```
void main( ){
    int num=0;
    Array a_Array(10);
    for(;num<10;num++)
        a_Array[num]=num;
    a_Array[10]=10;
    for(num=0;num<=10;num++)
        cout<<a_Array[num];
}
```

输出:

```
0 1 2 3 4 5 6 7 8 9
```

感谢倾听

给个好评！