程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 2 月 26 日

大纲

类与对象 02 讲义中的一页

对象间的运算
和结构变量一样，对象之间可以用“=”进行赋值，但是不能用“==”，“!=”，“>”，“<”“>=”“<=”进行比较，除非这些运算符经过了“重载”。

运算符重载的需求

- C++ 预定义的运算符，只能用于基本数据类型的运算：整型、实型、字符型、逻辑型…
  `+, -,*, /, %, ^, &, ~, !, |, <<, >>, !=, ...`
- 语义上更直观
  `complex_a`和`complex_b`是两个复数对象；求两个复数的和，希望能直接写：
  `complex_a + complex_b`
- 代码上更简洁

运算符重载

- 运算符重载，就是对已有的运算符 (C++ 中预定义的运算符) 赋予多重的含义，使同一运算符作用于不同类型的数据时导致不同类型的行为。

运算符重载

- 运算符重载，就是对已有的运算符 (C++ 中预定义的运算符) 赋予多重的含义，使同一运算符作用于不同类型的数据时导致不同类型的行为。
- 运算符重载的目的是：扩展 C++ 中提供的运算符的适用范围，使之能作用于对象。

运算符重载

- 运算符重载，就是对已有的运算符 (C++ 中预定义的运算符) 赋予多重的含义，使同一运算符作用于不同类型的数据时导致不同类型的行为。
- 运算符重载的目的是：扩展 C++ 中提供的运算符的适用范围，使之能作用于对象。
- 同一个运算符，对不同类型的操作数，所发生的行为不同。
  - `complex_a + complex_b`生成新的复数对象
  - $5 + 4 = 9$

运算符重载的形式

- 运算符重载的实质是函数重载

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 把含运算符的表达式转换成对运算符函数的调用。

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 把含运算符的表达式转换成对运算符函数的调用。
- 把运算符的操作数转换成运算符函数的参数。

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 把含运算符的表达式转换成对运算符函数的调用。
- 把运算符的操作数转换成运算符函数的参数。
- 运算符被多次重载时，根据实参的类型决定调用哪个运算符函数。

运算符重载的形式

- 运算符重载的实质是函数重载
- 可以重载为普通函数，也可以重载为成员函数
- 把含运算符的表达式转换成对运算符函数的调用。
- 把运算符的操作数转换成运算符函数的参数。
- 运算符被多次重载时，根据实参的类型决定调用哪个运算符函数。

```
返回值类型 operator 运算符 (形参表) {
  ......
}
```

运算符重载示例

```
1  #include <iostream>
2  using namespace std;
3
4  class Complex {
5  public:
6    double real, imag;
7    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
8    Complex operator-(const Complex &c);
9  };
10 Complex operator+(const Complex &a, const Complex &b) {
11   return Complex(a.real + b.real, a.imag + b.imag); //返回一个临时对象
12 }
13 Complex Complex::operator-(const Complex &c) {
14   return Complex(real - c.real, imag - c.imag); //返回一个临时对象
15 }
```

运算符重载示例

```cpp
#include <iostream>
using namespace std;

class Complex {
public:
  double real, imag;
  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
  Complex operator-(const Complex &c);
};
Complex operator+(const Complex &a, const Complex &b) {
  return Complex(a.real + b.real, a.imag + b.imag); //返回一个临时对象
}
Complex Complex::operator-(const Complex &c) {
  return Complex(real - c.real, imag - c.imag); //返回一个临时对象
}
```

重载为成员函数时，参数个数为运算符目数减一。

运算符重载示例

```cpp
#include <iostream>
using namespace std;

class Complex {
public:
  double real, imag;
  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
  Complex operator-(const Complex &c);
};
Complex operator+(const Complex &a, const Complex &b) {
  return Complex(a.real + b.real, a.imag + b.imag); //返回一个临时对象
}
Complex Complex::operator-(const Complex &c) {
  return Complex(real - c.real, imag - c.imag); //返回一个临时对象
}
```

重载为成员函数时，参数个数为运算符目数减一。
重载为普通函数时，参数个数为运算符目数。

运算符重载示例

```
17  int main() {
18    Complex a(4, 4), b(1, 1), c;
19    c = a + b; //等价于 c = operator+(a, b);
20    cout << c.real << "," << c.imag << endl;
21    cout << (a-b).real << "," << (a-b).imag << endl; //a-b 等价于 a.operator-(b)
22    return 0;
23  }
```

运算符重载示例

```
17  int main() {
18    Complex a(4, 4), b(1, 1), c;
19    c = a + b; //等价于 c = operator+(a, b);
20    cout << c.real << "," << c.imag << endl;
21    cout << (a-b).real << "," << (a-b).imag << endl; //a-b 等价于 a.operator-(b)
22    return 0;
23  }
```

输出

```
5,5
3,3
```

运算符重载示例

```
17  int main() {
18    Complex a(4, 4), b(1, 1), c;
19    c = a + b; //等价于 c = operator+(a, b);
20    cout << c.real << "," << c.imag << endl;
21    cout << (a-b).real << "," << (a-b).imag << endl; //a-b 等价于 a.operator-(b)
22    return 0;
23  }
```

输出

```
5,5
3,3
```

c = a + b; 等价于c = operator+(a, b);

运算符重载示例

```
17  int main() {
18    Complex a(4, 4), b(1, 1), c;
19    c = a + b; //等价于 c = operator+(a, b);
20    cout << c.real << "," << c.imag << endl;
21    cout << (a-b).real << "," << (a-b).imag << endl; //a-b 等价于 a.operator-(b)
22    return 0;
23  }
```

输出

```
5,5
3,3
```

c = a + b; 等价于c = operator+(a, b);

a - b;等价于a.operator-(b);

单选题

如果将 [] 运算符重载成一个类的成员函数，则该重载函数有几个参数?

A. 0
B. 1
C. 2
D. 3

单选题

如果将 [] 运算符重载成一个类的成员函数，则该重载函数有几个参数?

- A 0
- B 1
- C 2
- D 3

答案：B

赋值运算符'=' 重载

有时候希望赋值运算符两边的类型可以不匹配，比如，把一个int类型变量赋值给一个Complex对象，或把一个 char * 类型的字符串赋值给一个字符串对象，此时就需要重载赋值运算符 “=”。

赋值运算符'=' 重载

有时候希望赋值运算符两边的类型可以不匹配，比如，把一个int类型变量赋值给一个Complex对象，或把一个 char * 类型的字符串赋值给一个字符串对象，此时就需要重载赋值运算符“=”。

赋值运算符“=”只能重载为成员函数

运算符重载示例

```
5   class String {
6     char *str;
7   public:
8     String() : str(new char[1]) { str[0] = 0; }
9     const char *c_str() { return str; }
10    String &operator=(const char *s);
11    ~String() { delete[] str; }
12  };
13  String &String::operator=(const char *s) { //重载“=”以使得 obj=“hello”能够成立
14    delete[] str;
15    str = new char[strlen(s) + 1];
16    strcpy(str, s);
17    return *this;
18  }
19  int main() {
20    String s;
21    s = "Good Luck,"; //等价于 s.operator=("Good Luck,");
22    cout << s.c_str() << endl;
23    // String s2 = "hello!";   //这条语句要是不注释掉就会出错
24    s = "Shenzhou 8!"; //等价于 s.operator=("Shenzhou 8!");
25    cout << s.c_str() << endl;
26    return 0;
27  }
```

运算符重载示例

```
5  class String {
6    char *str;
7  public:
8    String() : str(new char[1]) { str[0] = 0; }
9    const char *c_str() { return str; }
10   String &operator=(const char *s);
11   ~String() { delete[] str; }
12 };
13 String &String::operator=(const char *s) { //重载“=”以使得 obj=“hello”能够成立
14   delete[] str;
15   str = new char[strlen(s) + 1];
16   strcpy(str, s);
17   return *this;
18 }
19 int main() {
20   String s;
21   s = "Good Luck,"; //等价于 s.operator=("Good Luck,");
22   cout << s.c_str() << endl;
23   // String s2 = "hello!";   //这条语句要是不注释掉就会出错
24   s = "Shenzhou 8!"; //等价于 s.operator=("Shenzhou 8!");
25   cout << s.c_str() << endl;
26   return 0;
27 }
```

输出：

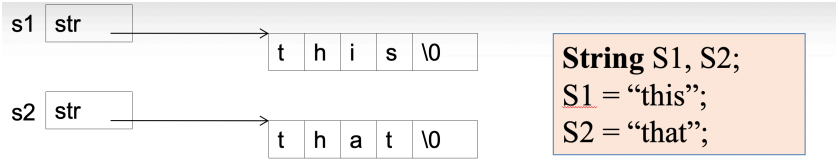
```
Good Luck,
Shenzhou 8!
```

浅拷贝和深拷贝

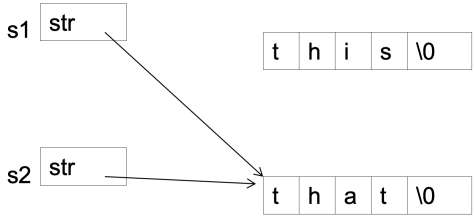
```
5  class String {
6    char *str;
7  public:
8    String() : str(new char[1]) { str[0] = 0; }
9    const char *c_str() { return str; }
10   String &operator=(const char *s);
11   ~String() { delete[] str; }
12 };
13 String &String::operator=(const char *s) { //重载“=”以使得 obj=“hello”能够成立
14   delete[] str;
15   str = new char[strlen(s) + 1];
16   strcpy(str, s);
17   return *this;
18 }
```

```
String S1, S2;
S1 = "this";
S2 = "that";
S1 = S2;
```

浅拷贝和深拷贝



String S1, S2;
S1 = “this”;
S2 = “that”;



S1 = S2;

浅拷贝和深拷贝

- 如不定义自己的赋值运算符，那么`S1 = S2`实际上导致 `S1.str`和 `S2.str` 指向同一地方。

浅拷贝和深拷贝

- 如不定义自己的赋值运算符，那么`S1 = S2`实际上导致 `S1.str`和 `S2.str` 指向同一地方。
- 如果`S1`对象消亡，析构函数将释放 `S1.str`指向的空间，则`S2`消亡时还要释放一次，不妥。

浅拷贝和深拷贝

- 如不定义自己的赋值运算符，那么`S1 = S2`实际上导致 `S1.str`和 `S2.str` 指向同一地方。
- 如果`S1`对象消亡，析构函数将释放 `S1.str`指向的空间，则`S2`消亡时还要释放一次，不妥。
- 另外，如果执行 `S1 = "other";`会导致`S2.str`指向的地方被`delete`

浅拷贝和深拷贝

- 如不定义自己的赋值运算符，那么S1 = S2实际上导致 S1.str和 S2.str 指向同一地方。
- 如果S1对象消亡，析构函数将释放 S1.str指向的空间，则S2消亡时还要释放一次，不妥。
- 另外，如果执行 S1 = "other";会导致S2.str指向的地方被delete
- 因此要在 class String里添加成员函数:

浅拷贝和深拷贝

- 如不定义自己的赋值运算符，那么S1 = S2实际上导致 S1.str和 S2.str 指向同一地方。
- 如果S1对象消亡，析构函数将释放 S1.str指向的空间，则S2消亡时还要释放一次，不妥。
- 另外，如果执行 S1 = "other";会导致S2.str指向的地方被delete
- 因此要在 class String里添加成员函数:

```
1  String &operator=(const String &s) {
2    delete[] str;
3    str = new char[strlen(s.str) + 1];
4    strcpy(str, s.str);
5    return *this;
6  }
```

浅拷贝和深拷贝

- 如不定义自己的赋值运算符，那么S1 = S2实际上导致 S1.str和 S2.str 指向同一地方。
- 如果S1对象消亡，析构函数将释放 S1.str指向的空间，则S2消亡时还要释放一次，不妥。
- 另外，如果执行 S1 = "other";会导致S2.str指向的地方被delete
- 因此要在 class String里添加成员函数:

```
1  String &operator=(const String &s) {
2    delete[] str;
3    str = new char[strlen(s.str) + 1];
4    strcpy(str, s.str);
5    return *this;
6  }
```

这么做就够了吗？还有什么需要改进的地方？

浅拷贝和深拷贝

考虑下面语句：

```
1  String s;
2  s = "Hello";
3  s = s;
```

是否会有问题?

浅拷贝和深拷贝

考虑下面语句：

```
String s;
s = "Hello";
s = s;
```

是否会有问题?

解决办法：

```
String &operator=(const String &s) {
  if (this == &s)
    return *this;
  delete[] str;
  str = new char[strlen(s.str) + 1];
  strcpy(str, s.str);
  return *this;
}
```

对 operator= 返回值类型的讨论

`void` 好不好?
`String` 好不好?
为什么是 `String &`

对 operator= 返回值类型的讨论

`void` 好不好?
`String` 好不好?
为什么是 `String &`
考虑：`a = b = c;`
和 `(a = b) = c;` //会修改 a 的值

对 operator= 返回值类型的讨论

`void` 好不好?
`String` 好不好?
为什么是 `String &`
考虑: `a = b = c;`
和 `(a = b) = c;` //会修改 a 的值
分别等价于:

```
a.operator=(b.operator=(c));
(a.operator=(b)).operator=(c);
```

对 operator= 返回值类型的讨论

`void` 好不好?
`String` 好不好?
为什么是 `String &`
考虑：`a = b = c;`
和 `(a = b) = c;` //会修改 a 的值
分别等价于：

```
a.operator=(b.operator=(c));
(a.operator=(b)).operator=(c);
```

对运算符进行重载的时候，好的风格是应该尽量保留运算符原本的特性

浅拷贝和深拷贝

上面的String类是否就没有问题了?

浅拷贝和深拷贝

上面的String类是否就没有问题了?
为 String类编写复制构造函数的时候，会面临和 = 同样的问题，用同样的方法处理。

```
1  String(const String &s) {
2    str = new char[strlen(s.str) + 1];
3    strcpy(str, s.str);
4  }
```

运算符重载为友元函数

一般情况下，将运算符重载为类的成员函数，是较好的选择。
但有时，重载为成员函数不能满足使用要求，重载为普通函数，又不能访问类的私有成员，所以需要将运算符重载为友元。

运算符重载为友元函数

一般情况下，将运算符重载为类的成员函数，是较好的选择。
但有时，重载为成员函数不能满足使用要求，重载为普通函数，又不能访问类的私有成员，所以需要将运算符重载为友元。

```
1  class Complex{
2    double real, imag;
3  public:
4    Complex(double r, double i) : real(r), imag(i) {}
5    Complex operator+(double r);
6  };
7  Complex Complex::operator+(double r) {
8    return Complex(real + r, imag);
9  }
```

运算符重载为友元函数

一般情况下，将运算符重载为类的成员函数，是较好的选择。
但有时，重载为成员函数不能满足使用要求，重载为普通函数，又不能访问类的私有成员，所以需要将运算符重载为友元。

```
1  class Complex{
2    double real, imag;
3  public:
4    Complex(double r, double i) : real(r), imag(i) {}
5    Complex operator+(double r);
6  };
7  Complex Complex::operator+(double r) {
8    return Complex(real + r, imag);
9  }
```

经过上述重载后：

```
1  Complex c;
2  c = c + 5;  //有定义，相当于 c = c.operator+(5);
```

运算符重载为友元函数

一般情况下，将运算符重载为类的成员函数，是较好的选择。
但有时，重载为成员函数不能满足使用要求，重载为普通函数，又不能访问类的私有成员，所以需要将运算符重载为友元。

```
class Complex{
  double real, imag;
public:
  Complex(double r, double i) : real(r), imag(i) {}
  Complex operator+(double r);
};
Complex Complex::operator+(double r) {
  return Complex(real + r, imag);
}
```

经过上述重载后：

```
Complex c;
c = c + 5;  //有定义，相当于 c = c.operator+(5);
```

但是：

```
c = 5 + c; //编译出错
```

运算符重载为友元函数

所以，为了使得上述的表达式能成立，需要将 + 重载为普通函数。

```
1  Complex operator+(double r, const Complex &c) { //能解释 5+c
2    return Complex(c.real + r, c.imag);
3  }
```

运算符重载为友元函数

所以，为了使得上述的表达式能成立，需要将 + 重载为普通函数。

```
Complex operator+(double r, const Complex &c) { //能解释 5+c
  return Complex(c.real + r, c.imag);
}
```

但是普通函数又不能访问私有成员，所以，需要将运算符 + 重载为友元。

```
class Complex {
  double real, imag;
public:
  Complex(double r, double i) : real(r), imag(i) {}
  Complex operator+(double r);
  friend Complex operator+(double r, const Complex &c);
};
```

运算符重载实例: 可变长整型数组

```
69  int main() {            //要编写可变长整型数组类，使之能如下使用:
70    CArray a; //开始里的数组是空的
71    for (int i = 0; i < 5; ++i)
72      a.push_back(i);
73    CArray a2, a3;
74    a2 = a;
75    for (int i = 0; i < a.length(); ++i)
76      cout << a2[i] << " ";
77    a2 = a3;                                  // a2 是空的
78    for (int i = 0; i < a2.length(); ++i) // a2.length() 返回 0
79      cout << a2[i] << " ";
80    cout << endl;
81    a[3] = 100;
82    CArray a4(a);
83    for (int i = 0; i < a4.length(); ++i)
84      cout << a4[i] << " ";
85    return 0;
86  }
```

运算符重载实例: 可变长整型数组

```
5  class CArray {
6    int size; //数组元素的个数
7    int *ptr; //指向动态分配的数组
8  public:
9    CArray(int s = 0); // s 代表数组元素的个数
10   CArray(const CArray &a);
11   ~CArray();
12   void push_back(int v);              //用于在数组尾部添加一个元素 v
13   CArray &operator=(const CArray &a); //用于数组对象间的赋值
14   int length() { return size; }       //返回数组元素个数
15   int &operator[](int i) { //返回值为 int 不行! 不支持 a[i] = 4
16     //用以支持根据下标访问数组元素, 如 n = a[i] 和 a[i] = 4; 这样的语句
17     return ptr[i];
18   }
19 };
20 CArray::CArray(int s) : size(s) {
21   if (s == 0) ptr = NULL;
22   else ptr = new int[s];
23 }
24 CArray::~CArray() {
25   if (ptr) delete[] ptr;
26 }
```

运算符重载实例: 可变长整型数组

```
27 CArray::CArray(const CArray &a) {
28   if (!a.ptr) {
29     ptr = NULL;
30     size = 0;
31     return;
32   }
33   ptr = new int[a.size];
34   memcpy(ptr, a.ptr, sizeof(int) * a.size);
35   size = a.size;
36 }
37 void CArray::push_back(int v) { //在数组尾部添加一个元素
38   if (ptr) {
39     int *tmpPtr = new int[size + 1];          //重新分配空间
40     memcpy(tmpPtr, ptr, sizeof(int) * size); //拷贝原数组内容
41     delete[] ptr;
42     ptr = tmpPtr;
43   } else { //数组本来是空的
44     ptr = new int[1];
45   }
46   ptr[size++] = v; //加入新的数组元素
47 }
```

运算符重载实例: 可变长整型数组

```
48 CArray &CArray::operator=(const CArray &a) {
49   //赋值号的作用是使“=”左边对象里存放的数组，大小和内容都和右边的对象一样
50   if (ptr == a.ptr) //防止 a=a 这样的赋值导致出错
51     return *this;
52   if (a.ptr == NULL) { //如果 a 里面的数组是空的
53     if (ptr)
54       delete[] ptr;
55     ptr = NULL;
56     size = 0;
57     return *this;
58   }
59   if (size < a.size) { //如果原有空间够大，就不用分配新的空间
60     if (ptr)
61       delete[] ptr;
62     ptr = new int[a.size];
63   }
64   memcpy(ptr, a.ptr, sizeof(int) * a.size);
65   size = a.size;
66   return *this;
67 } // CArray & CArray::operator=( const CArray & a)
```

单选题

在本例的可变长数组类中，重载了哪些运算符或编写了哪些成员函数?

A. =，[]，++
B. =，[]，复制构造函数
C. =，[]，++，复制构造函数
D. =，[]，&，复制构造函数

单选题

在本例的可变长数组类中，重载了哪些运算符或编写了哪些成员函数?

A. =，[]，++
B. =，[]，复制构造函数
C. =，[]，++，复制构造函数
D. =，[]，&，复制构造函数

答案：B

流插入运算符和流提取运算符的重载

```
1  cout << 5 << "this";
```

为什么能够成立?

流插入运算符和流提取运算符的重载

```
1  cout << 5 << "this";
```

为什么能够成立?
cout是什么?

流插入运算符和流提取运算符的重载

```
1  cout << 5 << "this";
```

为什么能够成立?

cout是什么?

“<<”为什么能用在 cout上?

流插入运算符和流提取运算符的重载

```
1  cout << 5 << "this";
```

为什么能够成立?
cout是什么?
“<<”为什么能用在 cout上?

- cout 是在 iostream 中定义的，ostream 类的对象。

流插入运算符和流提取运算符的重载

```
1  cout << 5 << "this";
```

为什么能够成立?
cout是什么?
“<<” 为什么能用在 cout上?

- cout 是在 iostream 中定义的，ostream 类的对象。
- "<<" 能用在cout 上是因为，在iostream里对"<<" 进行了重载。

流插入运算符和流提取运算符的重载

```
1  cout << 5 << "this";
```

为什么能够成立?
cout是什么?
“<<” 为什么能用在 cout上?

- cout 是在 iostream 中定义的，ostream 类的对象。
- "<<" 能用在cout 上是因为，在iostream里对"<<" 进行了重载。
- 考虑, 怎么重载才能使得 cout << 5; 和 cout << "this";都能成立?

流插入运算符和流提取运算符的重载

有可能按以下方式重载成 ostream类的成员函数：

```
1  void ostream::operator<<(int n) {
2    //输出 n 的代码
3    return;
4  }
```

流插入运算符和流提取运算符的重载

有可能按以下方式重载成 ostream类的成员函数：

```
1  void ostream::operator<<(int n) {
2    //输出 n 的代码
3    return;
4  }
```

cout << 5; 即 cout.operator<<(5);

cout << "this"; 即 cout.operator<<("this");

流插入运算符和流提取运算符的重载

有可能按以下方式重载成 ostream类的成员函数：

```
1  void ostream::operator<<(int n) {
2    //输出 n 的代码
3    return;
4  }
```

cout << 5; 即 cout.operator<<(5);

cout << "this"; 即 cout.operator<<("this");

怎么重载才能使得 cout << 5 << "this"; 成立?

流插入运算符和流提取运算符的重载

有可能按以下方式重载成 ostream类的成员函数:

```
1  void ostream::operator<<(int n) {
2    //输出 n 的代码
3    return;
4  }
```

cout << 5; 即 cout.operator<<(5);

cout << "this"; 即 cout.operator<<("this");

怎么重载才能使得 cout << 5 << "this"; 成立?

```
1  ostream &ostream::operator<<(int n) {
2    //输出 n 的代码
3    return *this;
4  }
5  ostream &ostream::operator<<(const char *s) {
6    //输出 s 的代码
7    return *this;
8  }
```

流插入运算符和流提取运算符的重载

有可能按以下方式重载成 ostream类的成员函数：

```
1  void ostream::operator<<(int n) {
2    //输出 n 的代码
3    return;
4  }
```

cout << 5; 即 cout.operator<<(5);

cout << "this"; 即 cout.operator<<("this");

怎么重载才能使得 cout << 5 << "this"; 成立?

```
1  ostream &ostream::operator<<(int n) {
2    //输出 n 的代码
3    return *this;
4  }
5  ostream &ostream::operator<<(const char *s) {
6    //输出 s 的代码
7    return *this;
8  }
```

cout << 5 << "this"; 本质上的函数调用的形式是什么?

cout.operator<<(5).operator<<("this");

流插入运算符和流提取运算符的重载

假定下面程序输出为 5hello, 该补写些什么

```
1  class CStudent {
2  public:
3    int nAge;
4  };
5  int main() {
6    CStudent s;
7    s.nAge = 5;
8    cout << s << "hello";
9    return 0;
10 }
```

流插入运算符和流提取运算符的重载

假定下面程序输出为 5hello, 该补写些什么

```
class CStudent {
public:
  int nAge;
};
int main() {
  CStudent s;
  s.nAge = 5;
  cout << s << "hello";
  return 0;
}
```

```
ostream &operator<<(ostream &o, const CStudent &s) {
  o << s.nAge;
  return o;
}
```

流插入运算符和流提取运算符的重载

假定 c 是 Complex 复数类的对象，
现在希望写“cout << c;”，就能以“a+bi”的形式输出 c 的值，
写“cin >> c;”，就能从键盘接受“a+bi”形式的输入，并且使得 c.real = a,c.imag = b。

```
27  int main() {
28    Complex c;
29    int n;
30    cin >> c >> n;
31    cout << c << "," << n;
32    return 0;
33  }
```

输入：13.2+133i 87
输出：13.2+133i, 87

流插入运算符和流提取运算符的重载

```
5  class Complex {
6    double real, imag;
7  public:
8    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
9    friend ostream &operator<<(ostream &os, const Complex &c);
10   friend istream &operator>>(istream &is, Complex &c);
11 };
12 ostream &operator<<(ostream &os, const Complex &c) {
13   os << c.real << "+" << c.imag << "i"; //以"a+bi" 的形式输出
14   return os;
15 }
16 istream &operator>>(istream &is, Complex &c) {
17   string s;
18   is >> s; //将"a+bi" 作为字符串读入, “a+bi”中间不能有空格
19   int pos = s.find("+", 0);
20   string sTmp = s.substr(0, pos); //分离出代表实部的字符串
21   c.real = atof(sTmp.c_str());
22   // atof 库函数能将 const char* 指针指向的内容转换成 float
23   sTmp = s.substr(pos + 1, s.length() - pos - 2); //分离出代表虚部的字符串
24   c.imag = atof(sTmp.c_str());
25   return is;
26 }
```

单选题

重载"<<" 用于将自定义的对象通过cout输出时，以下说法哪个是正确的?

A 可以将"<<" 重载为 ostream 类的成员函数，返回值类型是 ostream &

B 可以将"<<" 重载为全局函数，第一个参数以及返回值，类型都是 ostream

C 可以将"<<" 重载为全局函数，第一个参数以及返回值，类型都是 ostream &

D 可以将"<<" 重载为 ostream 类的成员函数，返回值类型是 ostream

单选题

重载"<<" 用于将自定义的对象通过cout输出时，以下说法哪个是正确的?

A 可以将"<<" 重载为 ostream 类的成员函数，返回值类型是 ostream &

B 可以将"<<" 重载为全局函数，第一个参数以及返回值，类型都是 ostream

C 可以将"<<" 重载为全局函数，第一个参数以及返回值，类型都是 ostream &

D 可以将"<<" 重载为 ostream 类的成员函数，返回值类型是 ostream

答案：C

重载类型转换运算符

```
4   class Complex {
5     double real, imag;
6   public:
7     Complex(double r = 0, double i = 0) : real(r), imag(i) { }
8     operator double() { //重载强制类型转换运算符 double
9       return real;
10    }
11  };
12  int main() {
13    Complex c(1.2, 3.4);
14    cout << (double)c << endl; //输出 1.2
15    double n = 2 + c;          //等价于 double n=2+c.operator double()
16    cout << n;                 //输出 3.2
17  }
```

重载类型转换运算符

```
4   class Complex {
5     double real, imag;
6   public:
7     Complex(double r = 0, double i = 0) : real(r), imag(i) { }
8     operator double() { //重载强制类型转换运算符 double
9       return real;
10    }
11  };
12  int main() {
13    Complex c(1.2, 3.4);
14    cout << (double)c << endl; //输出 1.2
15    double n = 2 + c;          //等价于 double n=2+c.operator double()
16    cout << n;                 //输出 3.2
17  }
```

类型强制转换运算符被重载时不能写返回值类型，实际上其返回值类型就是该类型强制转换运算符代表的类型

自增，自减运算符的重载

自增运算符++、自减运算符--有前置/后置之分，为了区分所重载的是前置运算符还是后置运算符，C++ 规定：

- 前置运算符作为一元运算符重载

```
1  //重载为成员函数:
2  T& operator++();
3  T& operator--();
4  //重载为全局函数:
5  T& operator++(T&);
6  T& operator--(T&);
```

自增，自减运算符的重载

自增运算符++、自减运算符--有前置/后置之分，为了区分所重载的是前置运算符还是后置运算符，C++ 规定：

- 前置运算符作为一元运算符重载

```
1 //重载为成员函数:
2 T& operator++();
3 T& operator--();
4 //重载为全局函数:
5 T& operator++(T&);
6 T& operator--(T&);
```

- 后置运算符作为二元运算符重载，多写一个没用的参数:

```
1 //重载为成员函数:
2 T operator++(int);
3 T operator--(int);
4 //重载为全局函数:
5 T operator++(T&, int);
6 T operator--(T&, int);
```

自增，自减运算符的重载

```
34 int main() {
35   CDemo d(5);
36   cout << (d++) << ","; //等价于 d.operator++(0);
37   cout << d << ",";
38   cout << (++d) << ","; //等价于 d.operator++();
39   cout << d << endl;
40   cout << (d--) << ","; //等价于 operator--(d,0);
41   cout << d << ",";
42   cout << (--d) << ","; //等价于 operator--(d);
43   cout << d << endl;
44   return 0;
45 }
```

自增，自减运算符的重载

```
34 int main() {
35   CDemo d(5);
36   cout << (d++) << ","; //等价于 d.operator++(0);
37   cout << d << ",";
38   cout << (++d) << ","; //等价于 d.operator++();
39   cout << d << endl;
40   cout << (d--) << ","; //等价于 operator--(d,0);
41   cout << d << ",";
42   cout << (--d) << ","; //等价于 operator--(d);
43   cout << d << endl;
44   return 0;
45 }
```

输出结果：

```
5,6,7,7
7,6,5,5
```

如何编写 CDemo?

自增，自减运算符的重载

```
4  class CDemo {
5  private:
6    int n;
7  public:
8    CDemo(int i = 0) : n(i) {}
9    CDemo &operator++();   //用于前置形式
10   CDemo operator++(int); //用于后置形式
11   operator int() { return n; }
12   friend CDemo &operator--(CDemo &);
13   friend CDemo operator--(CDemo &, int);
14 };
15 CDemo &CDemo::operator++() { //前置 ++
16   n++;
17   return *this;
18 }
19 CDemo CDemo::operator++(int k) {   //后置 ++
20   CDemo tmp(*this); //记录修改前的对象
21   n++;
22   return tmp; //返回修改前的对象
23 } // s++ 即为: s.operator++(0);
24 CDemo &operator--(CDemo &d) { //前置--
25   d.n--;
26   return d;
27 } //--s 即为: operator--(s);
28 CDemo operator--(CDemo &d, int) { //后置--
29   CDemo tmp(d);
30   d.n--;
31   return tmp;
32 } // s--即为: operator--(s, 0);
```

单选题

如何区分自增运算符重载的前置形式和后置形式?

- A 后置形式比前置形式多一个 `int` 类型的参数
- B 重载时，前置形式的函数名是 `++operator`，后置形式的函数名是 `operator++`
- C 无法区分，使用时不管前置形式还是后置形式，都调用相同的重载函数
- D 前置形式比后置形式多了一个`int`类型的参数

单选题

如何区分自增运算符重载的前置形式和后置形式?

- A 后置形式比前置形式多一个 `int` 类型的参数
- B 重载时，前置形式的函数名是 `++operator`，后置形式的函数名是 `operator++`
- C 无法区分，使用时不管前置形式还是后置形式，都调用相同的重载函数
- D 前置形式比后置形式多了一个`int`类型的参数

答案：A

->重载

```cpp
#include <iostream>
using namespace std;

class A {
private:
  int x;
public:
  A() : x(5) {}
  int getX() {
    return x;
  }
  A *operator->() {
    return this;
  }
};

int main() {
  A a;
  cout << a->getX() << endl;//a.operator->()->getX()
  return 0;
}
```

->重载

```
4  class client {
5  public:
6    int a;
7    client(int x) : a(x) {}
8  };
9
10 class proxy {
11   client *target;
12 public:
13   proxy(client *t) : target(t) {}
14   client *operator->() const { return target; }
15 };
16
17 class proxy2 {
18   proxy *target;
19 public:
20   proxy2(proxy *t) : target(t) {}
21   proxy &operator->() const { return *target; }
22 };
23
24 int main() {
25     client x(3);
26     proxy y(&x);
27     proxy2 z(&y);
28     cout << x.a << y->a << z->a; // print "333"
29     return 0;
30 }
```

https://stackoverflow.com/questions/8777845/overloading-member-access-operators

运算符重载的注意事项

1. C++ 不允许定义新的运算符；

运算符重载的注意事项

1. C++ 不允许定义新的运算符；
2. 重载后运算符的含义应该符合日常习惯；
   - `complex_a + complex_b`
   - `word_a > word_b`
   - `date_b = date_a + n`

运算符重载的注意事项

1. C++ 不允许定义新的运算符；
2. 重载后运算符的含义应该符合日常习惯；
   - `complex_a + complex_b`
   - `word_a > word_b`
   - `date_b = date_a + n`
3. 运算符重载不改变运算符的优先级；

运算符重载的注意事项

1. C++ 不允许定义新的运算符；
2. 重载后运算符的含义应该符合日常习惯；
   - `complex_a + complex_b`
   - `word_a > word_b`
   - `date_b = date_a + n`
3. 运算符重载不改变运算符的优先级；
4. 以下运算符不能被重载：`"."`, `".*"`, `"::"`, `"?:"`, `sizeof`

运算符重载的注意事项

1. C++ 不允许定义新的运算符；
2. 重载后运算符的含义应该符合日常习惯；
   - `complex_a + complex_b`
   - `word_a > word_b`
   - `date_b = date_a + n`
3. 运算符重载不改变运算符的优先级；
4. 以下运算符不能被重载：`"."`，`".*"`，`"::"`，`"?:"`，`sizeof`
5. 重载运算符`()`，`[]`，`->`或者赋值运算符`=`时，运算符重载函数必须声明为类的成员函数。

运算符重载的注意事项

1. C++ 不允许定义新的运算符；
2. 重载后运算符的含义应该符合日常习惯；
   - `complex_a + complex_b`
   - `word_a > word_b`
   - `date_b = date_a + n`
3. 运算符重载不改变运算符的优先级；
4. 以下运算符不能被重载：`"."`，`".*"`，`"::"`，`"?:"`，`sizeof`
5. 重载运算符()，[]，->或者赋值运算符=时，运算符重载函数必须声明为类的成员函数。
6. 重载运算符是为了让它能作用于对象，因此重载运算符，不允许操作数都不是对象 (有一个操作数是枚举类型也可以)。

```
1  void operator+(double a, char *p) {//此重载不成立
2  }
```

注：What are the pointer-to-member operators ->* and .* in C++?

https://stackoverflow.com/questions/6586205/what-are-the-pointer-to-member-operators-and-in-c