函数重载

如果**一个作用域**内**几个函数名字相同**但是**参数列表不同**，成为**函数重载**。

```
#include<iostream>
using namespace std;
int func(int x) {
    return x;
}
double func(double x) {
    return x + 1000;
}
int main() {
    cout << func(2) << endl;
    cout << func(2.0) << endl;
    return 0;
}
```

同函数名，通过参数能够区分的，实现更精细化的处理。编译器如何知道呢？——参数的模式识别。参数不同是指：**参数类型**不同或者**参数个数**不同。（要考虑**默认参数值**）

```
int func(int x) {
    return 2 * x;
}
int func(int x, int y = 2) {
    return y * x;
}
// func(2) 既可以调用第一个版本，也可以调用第二个版本,编译器就懵了
```

重载的意义

1. 通过函数名对函数功能进行提示
2. 通过函数参数列表对函数的用法进行提示
3. 扩展已有的功能

- **通过函数名对函数功能进行提示、扩展已有功能:**

```cpp
#include<iostream>
#include<cmath>
using namespace std;
int ABS(int x) {
    return abs(x);
}
double ABS(double x) {
    return fabs(x);
}
int main() {
    cout << ABS(-2) << endl;
    cout << ABS(-2.3) << endl;
    return 0;
}
```

- **通过函数参数列表对函数的用法进行提示**：

```cpp
//输出全排列
void output_permutation(int ind, int n, int *num, int *buff) { //专业版
    //ind表示存到第几位了
    if(ind == n) {
        for (int i = 0; i < n; i++) {
            cout << buff[i];
        }
        cout << endl;
    }
    for (int i = 1; i <= n; i++) {
        if (num[i]) continue;
        buff[ind] = i;
        num[i] = 1;
        output_permutation(ind + 1, n, num, buff);
        num[i] = 0;
    }
    return ;
}
//主要体现在接口的设计
void output_permutation(int n) { //用户版
    int num[n + 1], buff[n]; // num记录数是否被用过,buff用来暂存数。
    memset(num,  0, sizeof(num));
    memset(buff, 0, sizeof(buff));
    output_permutation(0, n, num, buff);
    return ;
}
output_permutation(4);
```

友元

> 再次辨别类内和类外的含义：这是一个描述**操作环境**的词。外界想要操控某个对象的时候，站在的是类外；类内部实现的时候想要修改对象站在的是类内。

- **友元函数一定放在类内的逻辑：**对方是不是你的朋友只有你说了算，别人说了不算。

```
#include<iostream>
using namespace std;
//输出点类的对象信息
//1. 类内
//2. 类外：友元

class Point {
public:
    Point(int x, int y) : x(x), y(y) {}
    void output() {
        cout << "inner : " << x << " " << y << endl;
    }
    friend void output(Point &); // 类外的output函数是友元
private:
    int x, y;
};

void output(Point &a) {
    cout << "outer : " << a.x << " " << a.y <<endl;
}
int main() {
    Point p(3, 4);
    p.output();
    output(p);
    return 0;
}
```

随堂练：实现一个复数类

要求：

1. 数据的安全性
2. 通过构造函数直接给实部和虚部赋值
3. 完成复数的加减乘除运算

运算符重载（非成员）

C++中重载能够扩展运算符的功能，重载以函数的方式进行。

重点：**运算符重载是扩展运算符的功能，而不是新创造一个运算符**，可以当成一种特殊的函数重载形式。

```
Type operator Sign(type &obj1, type &obj2) {//单目运算符只有一个参数，双目运算符有两个参数。
        Type ret;
        //do something;
        return ret;
}
```

- 单目运算符：++ 、 -- 、 *
- 双目运算符：+ - * /

建立起这么一种认知：`+` 是**运算符**，而`operator+`是运算符的**函数名**。

const关键字在运算符重载中的作用：12.friend.cpp

希望兼容所有参数绑定的情况：正常对象 + 临时匿名对象的绑定。

左值和右值的简单区别：

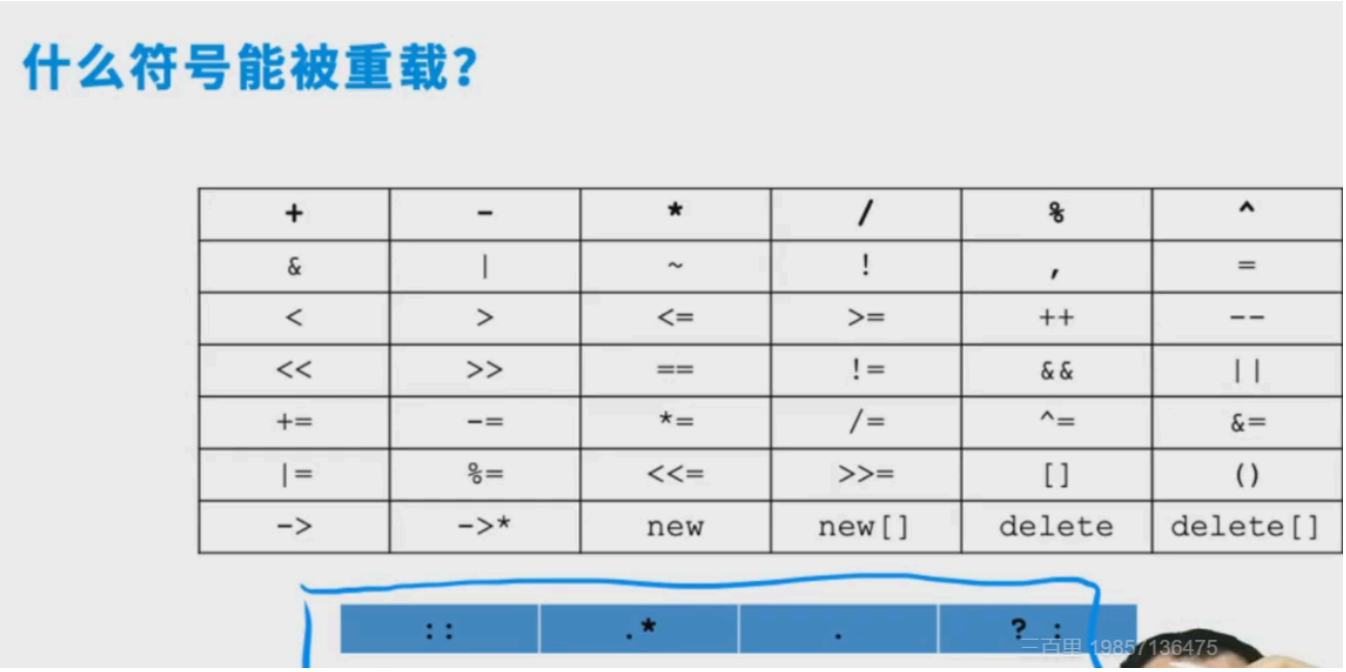
- 左值是常驻值，存储在变量中。
- 右值是临时值，存储在匿名变量中

```cpp
#include<iostream>
#include<cstdlib>
using namespace std;
//输出点类的对象信息
//1. 类内
//2. 类外：友元

class Point {
public:
    Point(int x, int y) : x(x), y(y) {
        z1 = rand();
        z2 = z1 + 1;
        z3 = z2 + 1;
        cout << this << "rand value : " << z1 << endl;
    }
    void output() {
        cout << "inner : " << x << " " << y << endl;
    }
    friend void output(Point &); // 类外的output函数是友元
    friend ostream &operator<<(ostream &, const Point &);
    friend Point operator+(const Point &, const Point &);
    Point operator+(int n) const {// 这里加上const是为了兼容const类型的对象
        return Point(x + n, y + n);
    }
    int z1, z2, z3;
private:
    int x, y;
};

void output(Point &a) {
    cout << "outer : " << a.x << " " << a.y <<endl;
}

ostream &operator<<(ostream &out, const Point &p) {
    out << "Point(" << p.x << ", " << p.y << ")";
    return out;
}
Point operator+(const Point &a, const Point &b) {
    return Point(a.x + b.x, a.y + b.y);
}
int main() {
    srand(time(0));
    Point p(3, 4), q(5, 6);
    p.output();
    output(p);
    cout << p << endl;
```

```
    cout << (p + q) << endl;
    cout << (p + 5) << endl;
    cout << (p + (p + q)) << endl;
    // p + 5 => p.operator+(5)
    const Point cp(7, 8);
    cout << (cp + 6) << endl;

    Point *pp = &p;
    int Point::* px = &Point::z1;
    // 表示这是一个Point类的,int类型的成员属性，指向z1这个属性（是逻辑地址，随对象不同而不同）
    cout << pp->*px << endl;
    px = &Point::z2;
    cout << pp->*px << endl;
    return 0;
}
```

什么运算符不能被重载？

`::`、`.*`、`.`、`?:` 不能被重载。**sizeof**在C++中本质上还算一个运算符。只不过它也不能被重载。



运算符返回什么样的值，完全取决于功能设计。

运算符重载如何区分前++和后++呢？——如果是后++，参数栏里有一个int，否则就是前++。

```cpp
#include<iostream>
#include<cstdlib>
using namespace std;
#define BEGINS(x) namespace x {
#define ENDS(x) }

BEGINS(haizei)

class Point {
public:
    Point(int x, int y) : x(x), y(y) {
        z1 = rand();
        z2 = z1 + 1;
        z3 = z2 + 1;
        cout << this << "rand value : " << z1 << endl;
    }
    void output() {
        cout << "inner : " << x << " " << y << endl;
    }
    friend void output(Point &); // 类外的output函数是友元
    friend ostream &operator<<(ostream &, const Point &);
    friend Point operator+(const Point &, const Point &);
    Point operator+(int n) const {
        return Point(x + n, y + n);
    }
    Point &operator+=(int x) {
        this->x += x;
        this->y += x;
        return *this;
    }
    Point operator++(int) {//返回加一之前的那个值,所以不传引用
        Point ret(*this);
        x++;
        y++;
        return ret;
    }
    Point &operator++() {
        x++;
        y++;
        return *this;
    }
    int z1, z2, z3;
private:
    int x, y;
};
```

```
void output(Point &a) {
    cout << "outer : " << a.x << " " << a.y <<endl;
}

ostream &operator<<(ostream &out, const Point &p) {
    out << "Point(" << p.x << ", " << p.y << ")";
    return out;
}
Point operator+(const Point &a, const Point &b) {
    return Point(a.x + b.x, a.y + b.y);
}

ENDS(haizei)


BEGINS(test2)
using namespace haizei;

int main(){
    Point p(3, 4);
    (p += 5) += 6;
    cout << p << endl;
    cout << "p++ = " << p++ << endl;
    cout << "p = "   << p   << endl;
    cout << "++p = " << ++p << endl;

    int n = 45;
    cout << "n = " << (++n)++ << endl; // 47 被加了两次，约定俗成是返回该对象，而不是临时对象

    (++p)++;
    cout << "(++p)++ = " << p << endl;
    return 0;
}

ENDS(test2)
int main(){
    // test1::main(); 第一个测试用例
    test2::main();

    return 0;
}
```

数组对象

本质上就是重载了`[]`运算符的对象

```cpp
#include<iostream>
#include<cstdlib>
using namespace std;
//输出点类的对象信息
//1. 类内
//2. 类外：友元

#define BEGINS(x) namespace x {
#define ENDS(x) }

BEGINS(haizei)
class Point;
ENDS(haizei)
//1. Point类的声明

BEGINS(test3)
class Array {
public:
    Array(int n = 100) : n(n), data(new int[n]) {}
    int &operator[](int ind) {
        return data[ind];
    }
    const int &operator[](int ind) const {
        return data[ind];
    }//返回值是const类型的保证不会被修改。
    int operator[](const char *s) {
        int ind = stoi(s);
        return data[ind];
    }
    int &operator[](const haizei::Point &a);
private:
    int n;
    int *data;
};
ENDS(test3)
//2. Array类的具体声明：需要用到Point的简单声明

BEGINS(haizei)
class Point {
public:
    Point(int x, int y) : x(x), y(y) {
        z1 = rand();
        z2 = z1 + 1;
        z3 = z2 + 1;
        cout << this << "rand value : " << z1 << endl;
    }
```

```
    void output() {
        cout << "inner : " << x << " " << y << endl;
    }
    friend void output(Point &); // 类外的output函数是友元
    friend ostream &operator<<(ostream &, const Point &);
    friend Point operator+(const Point &, const Point &);
    friend int &test3::Array::operator[](const Point &);
    Point operator+(int n) const {
        return Point(x + n, y + n);
    }
    Point &operator+=(int x) {
        this->x += x;
        this->y += x;
        return *this;
    }
    Point operator++(int) {//返回加一之前的那个值,所以不传引用
        Point ret(*this);
        x++;
        y++;
        return ret;
    }
    Point &operator++() {
        x++;
        y++;
        return *this;
    }
    int z1, z2, z3;

private:
    int x, y;
};
void output(Point &a) {
    cout << "outer : " << a.x << " " << a.y <<endl;
}
ostream &operator<<(ostream &out, const Point &p) {
    out << "Point(" << p.x << ", " << p.y << ")";
    return out;
}
Point operator+(const Point &a, const Point &b) {
    return Point(a.x + b.x, a.y + b.y);
}
ENDS(haizei)
//3. Point类的具体声明：需要用到Array类的具体声明

BEGINS(test1)
using namespace haizei;
```

```cpp
int main() {
    srand(time(0));
    Point p(3, 4), q(5, 6);
    p.output();
    output(p);
    cout << p << endl;
    cout << (p + q) << endl;
    cout << (p + 5) << endl;
    cout << (p + (p + q)) << endl;
    // p + 5 => p.operator+(5)
    const Point cp(7, 8);
    cout << (cp + 6) << endl;

    Point *pp = &p;
    int Point::* px = &Point::z1;
    // 表示这是一个Point类的,int类型的成员属性，指向z1这个属性（是逻辑地址，随对象不同而不同）
    cout << pp->*px << endl;
    px = &Point::z2;
    cout << pp->*px << endl;
    return 0;
}
ENDS(test1)

BEGINS(test2)
using namespace haizei;

int main(){
    Point p(3, 4);
    (p += 5) += 6;
    cout << p << endl;
    cout << "p++ = " << p++ << endl;
    cout << "p = "   << p   << endl;
    cout << "++p = " << ++p << endl;

    int n = 45;
    cout << "n = " << (++n)++ << endl; // 47 被加了两次，约定俗成是返回该对象，而不是临时对象

    (++p)++;
    cout << "(++p)++ = " << p << endl;
    return 0;
}
ENDS(test2)


BEGINS(test3)
using namespace haizei;
```

```cpp
int &Array::operator[](const haizei::Point &a) {
    return data[a.x + a.y];
}
//4. Array::operator[]的定义：需要用到Point类的属性

int main(){
    Array arr;
    for (int i = 0; i < 10; i++) {
        arr[i] = i;//返回值得是引用，相关位置变量的引用
    }
    cout << arr["0"] << " " << arr["1"] << endl;
    for (int i = 0; i < 10; i++) {
        cout<<arr[i] << " ";
    }
    cout<<endl;

    const Array arr2;
    for (int i = 0; i < 10; i++) {
        cout << arr2[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < 10; i++) {
        cout << &arr2[i] << " ";
    }
    cout << endl;
    return 0;
}
ENDS(test3)

int main(){
    // test1::main(); 第一个测试用例
    //test2::main();
    test3::main();
    return 0;
}
```