

程序设计实习（实验班-2024春）

C++面向对象编程：重载

授课教师：姜少峰

助教：冯施源 吴天意

Email: shaofeng.jiang@pku.edu.cn

重载

Overloading

函数重载

多个不同构造函数也可以理解成重载

同名的函数、形参表不同，在C++视为不同的函数（const、引用也算类型区分）

- 必须通过形参表的参数个数、类型来区分，不能仅通过返回值类型区分
- `int f(int a, int b); void f(int a, int b)`是不能同时存在的
- 形参表的形参变量名也不能作为区分依据
- `int f(int a, int b); void f(int x, int y)`也是不能同时存在的

调用时根据调用的参数类型匹配；若都匹配不上则尝试类型转换后匹配

若有模糊则编译错误

类型模糊导致错误的例子

```
float increase(int a, float delta) {  
    return a + delta;  
}  
  
double increase(int a, double delta) {  
    return a + delta;  
}  
  
int main()  
{  
    increase(4, 4); // 错误: 第二个int可以匹配float和double型  
    return 0;  
}
```

const和引用对重载的影响

```
int increase(int a, int delta) {  
    return a + delta;  
}  
  
int increase(int a, int& delta) {  
    return a + delta;  
}  
  
int main()  
{  
    int x = 4;  
    increase(x, 4); // 正确: 第二个参数只能匹配int而不是int&  
    increase(x, x); // 错误: 第二个参数有歧义  
    return 0;  
}
```

```
int increase(int a, int& delta) {  
    return a + delta;  
}  
  
int increase(int a, const int& delta) {  
    return a + delta;  
}  
  
int main()  
{  
    int x = 4;  
    increase(x, x); // 调用int&版本  
    increase(x, 4); // 调用const int&版本  
    return 0;  
}
```

默认参数

这么设计的必要性？

- 在形参表可以给最后连续的几个参数赋默认值，可以是任何合法的表达式
- 匹配有歧义将会报错

```
int increase(int a, int delta = 1, int step = 1) {  
    return a + delta * step;  
}  
  
int main()  
{  
    increase(1);  
    increase(1, 2);  
    increase(1, 2, 3);  
    return 0;  
}
```

没问题

```
int increase(int a, int delta = 1) {  
    return a + delta;  
}  
  
int increase(int a, int delta = 1, int step = 1) {  
    return a + delta * step;  
}  
  
int main()  
{  
    increase(1);  
    increase(1, 2);  
    increase(1, 2, 3);  
    return 0;  
}
```

由于新的increase重载的存在，这里有歧义

默认参数看作重载

默认参数可以理解成一系列重载（即默认参数可用重载实现）

等价于这三个函数

```
int increase(int a, int delta = 1, int step = 1) {  
    return a + delta * step;  
}
```

```
int increase(int a) {  
    return a + 1;  
}  
  
int increase(int a, int delta) {  
    return a + delta;  
}  
  
int increase(int a, int delta, int step) {  
    return a + delta * step;  
}
```

操作符重载

概述

在C++中可对运算符重载：例如想要两个2维点p, q做坐标相加，写成p + q

- 操作符实质是一种函数，一般是`operatorX`作为函数名（例如+就是operator+）
 - 因此可以自由的指定参数类型和返回值，但应尽量保持其“自然”的意义
- C++中除“.” “::” “?:” 均可重载
 - 不可定义新的操作符，不可重载关于基本类型（如int）的操作符
 - 操作符的优先级等属性不可改变

指针，即使是某个class类型的指针，也算基本类型，也不可重载。因此要重载必须至少有一个非指针的class的类型

一般形式

“目”即需要的操作数个数，如+是2目，!是1目

- 考虑一个n目操作符
- 两种可能：以类（非静态）成员函数重载，或以全局函数重载
- 类（非静态）成员函数重载：
 - 形参表长度为n - 1，当前类作为隐含的第一个操作数
 - 返回值 operatorX(形参表n_1)
- 全局函数重载：
 - 形参表长度为n，按顺序对应操作符的所有操作数
 - 返回值 operatorX(形参表n)

注意：返回值要根据具体情况指定

举例

这种都是指针的全局版本是不允许的
(因为指针算基本类型) !

```
Point operator+(Point *p, Point *q)
```

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    Point operator+(const Point& p) const {  
        return Point(x + p.x, y + p.y);  
    }  
    Point operator-(const Point& p) const {  
        return Point(x - p.x, y - p.y);  
    }  
    Point operator-() const {  
        return Point(-x, -y);  
    }  
    Point operator*(int c) const { // 伸缩  
        return Point(x * c, y * c);  
    }  
    int operator*(const Point& p) const { // 内积  
        return x * p.x + y * p.y;  
    }  
};
```

单目运算符: 取反

```
int main() {  
    Point x(5, 4), y(1, 1);  
    Point z = x + y;  
    Point zz = x * 10;  
    int t = x * y;  
    y = -x;  
    return 0;  
}
```

相当于调用x.operator+(y)

```
int operator*(const Point& p, const Point& q) {  
    return p.x * q.x + p.y * q.y;  
}
```

内积函数的全局函数版本

Tricky question: 如果类内和全局两个内积operator*都写上会怎样?
全局版的第一个参数去掉const呢? 类内版去掉函数结尾的const呢?

有时全局型定义是必须的

- 例如我们想定义Point + int的操作，意为每个维度都加上这个int值

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    Point operator+(int delta) const {  
        return Point(x + delta, y + delta);  
    }  
};
```

- 如果调用Point p(1, 1); p + 5是没问题的，但是调用5 + p未定义
- 需要全局：

```
Point operator+(int delta, const Point& p) {  
    return Point(p.x + delta, p.y + delta);  
}
```

注意：若x, y都是private的，那么该operator+需在Point类里面声明成友元

一些特殊运算符的重载

赋值运算符的重载

- 作用类似复制构造函数/转换构造函数
- 参数表和返回值应该是什么?
即调用`a.operator=(b)`
- 目标：我们要达到调用`a = b`后，`a`的值改变成`b`的值
- 错误写法：

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    Point operator=(const Point& p) {  
        return p;  
    }  
};
```

返回值不会被使用并复制给当前类

```
int main() {  
    Point p(1, 2), q(2, 3);  
    p = q;  
    cout << p.x << " " << p.y << endl;  
    return 0;  
}
```

会输出1 2，而不是
2 3

赋值运算符的重载 (cont.)

void型返回值及其问题

尝试：在Point类里定义`void operator=(const Point& p)`，拷贝p的内容

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    void operator=(const Point &p) {  
        this->x = p.x; this->y = p.y;  
    }  
};
```

```
int main()  
{  
    Point x(5, 4), y(1, 1);  
    x = y;  
    cout << x.x << " " << x.y << endl;  
    return 0;  
}
```

输出1 1，看似没问题？

但是void型返回值也有不好的地方，例如想使用`a = b = c`这种操作

- 会编译错误：第一次调用`b = c`返回的是void型，不能再赋值给a

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    void operator=(const Point &p) {  
        this->x = p.x; this->y = p.y;  
    }  
};
```

赋值运算符的重载 (cont.)

标准写法

注意：

=, [], (), -> 必须在类里定义，而不能是全局的

- 标准的写法：

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    Point& operator=(const Point& p) {  
        x = p.x; y = p.y;  
        return *this;  
    }  
};
```

- 原则：应兼容一切“自然”的赋值用法（例如int型可用的写法这里也可以用）
- 是否能返回const Point&? 或Point? 不能：

若operator=返回const Point&
这里就会报错（返回Point也会）

```
void modify(Point& p) {  
    p.x += 1;  
}  
int main()  
{  
    Point x(5, 4), y(1, 1);  
    modify(x = y);  
    return 0;  
}
```


赋值运算符的重载 (cont.)

另外的问题：如果调用`x = x`会怎样？

```
struct PointD {  
    int* coor = NULL; int d;  
    PointD(int _d) : d(_d) {coor = new int[d];}  
    PointD& operator=(const PointD& p) {  
        d = p.d;  
        delete[] coor; coor = new int[d];  
        for (int i = 0; i < d; i++) {coor[i] = p.coor[i];}  
        return *this;  
    }  
};  
int main() {  
    PointD x(3);  
    x = x;  
    return 0;  
}
```

自己的内容复制之前，就delete掉了

如何解决：在operator=函数实现的第一句加上
`if (&p == this) return *this`

这也是实现operator=的标准写法！

测试题

这段程序输出什么？

```
struct X {
    X() {cout << "constructor" << endl;}
    X(const X& x) {cout << "copy constructor" << endl;}
    X& operator=(const X& x) {
        if (this == &x) return *this;
        cout << "operator=" << endl;
        return *this;
    }
};

int main()
{
    X x = X();
    X y = x;
    x = y;
    return 0;
}
```

函数调用操作符()

回忆：
()也必须在类里定义，而不能是全局的

利用()操作符可以让对象类似于函数来进行调用，例如：

```
struct Func {  
    int operator()(int a, int b) const {  
        return a + b;  
    }  
};  
  
void work(const Func& f) {  
    cout << f(1, 2) << endl;  
}  
  
int main() {  
    work(Func());  
    return 0;  
}
```

f本身是对象，但是这里的语法允许其像函数一样使用

函数调用操作符的应用：Function Object

利用这个对象来定义比较函数

- 例如标准库std::sort：接受begin和end后，再接受一个定义()操作符的对象
- 具体来说：operator()需要接受两个待排序的变量，返回bool，代表小于关系

```
struct PointComparator {  
    bool operator()(const Point& p, const Point& q) const {  
        if (p.x != q.x) return p.x < q.x;  
        return p.y < q.y;  
    }  
};  
  
int main()  
{  
    Point x(5, 4), y(1, 1), u(0, 0);  
    Point arr[3] = {x, y, u};  
    std::sort(arr, arr + 3, PointComparator());  
    return 0;  
}
```

对Point类型进行双关键字排序：第一关键字是横坐标，第二是纵坐标

Function Object (cont.)

- 对于排序来说，还可以重载operator<来达到类似的效果，例如

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    bool operator<(const Point& p) const {  
        if (x != p.x) return x < p.x;  
        return y < p.y;  
    }  
};  
int main() {  
    Point x(5, 4), y(1, 1), u(0, 0);  
    Point arr[3] = {x, y, u};  
    std::sort(arr, arr + 3);  
    return 0;  
}
```

注意标准库要求const函数

- 但是：function object可以更加灵活，例如需要定义多种比较器，或需要额外参数

例：到定点距离为关键字

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    int distsq(const Point &p) const {  
        Point delta = *this - p;  
        return delta.x * delta.x + delta.y * delta.y;  
    }  
    Point operator-(const Point& p) const {  
        return Point(x - p.x, y - p.y);  
    }  
};
```

根据到o点距离排序，
有构造函数和成员变量

```
class PointComparatorDist {  
    Point o;  
public:  
    PointComparatorDist(const Point& p) : o(p) {}  
    bool operator()(const Point& p, const Point& q) {  
        return p.distsq(o) < q.distsq(o);  
    }  
};  
  
int main()  
{  
    Point x(5, 4), y(1, 1), u(0, 0);  
    Point arr[3] = {x, y, u};  
    std::sort(arr, arr + 3, PointComparatorDist(Point(0, 1)));  
    for (int i = 0; i < 3; i++) {  
        cout << arr[i].x << " " << arr[i].y << endl;  
    }  
    return 0;  
}
```

创建对象时传入o点坐标
(利用构造函数)

类型转换操作符type()

- 例如有Point p，想利用(int)p把p转化成int（此处type()的type = int）

- 规则：

()必须在类里定义、而不能是全局的规则依然适用！

- 函数名operator type()，无返回值，形参为空

- 一般定义为const成员函数

- 类型转换时会自动调用

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    operator int() const {  
        return x + y;  
    }  
};  
int main() {  
    Point x(5, 4);  
    cout << (int)x << endl;  
    return 0;  
}
```

数组下标操作符[]

回忆：

[]也必须在类里定义，而不能是全局的

- 例如当对象本身维护了一个类似数组的结构，然后想通过下标来访问
 - 例如有一个d维欧氏点PointD类，对象PointD pd，想用pd[i]访问第i维
- 标准写法：

```
struct PointD {  
    int* coor = NULL; int d;  
    PointD(int _d) : d(_d) {coor = new int[d];}  
    int& operator[] (int i) {  
        return coor[i];  
    }  
};
```

注意此处返回引用

- 返回引用的好处：允许pd[i] = 1这种改写操作，否则只能读取pd[i]

流操作符>>和<<

以<<为例

- `cout << a << b << c`等价于`((cout.op<<(a)).op<<(b)).op<<(c)`
- 为使上述操作成立，应该如何设计`operator<<`的形参表和返回值？

成员函数版本： `ostream& operator<<(int)`

此处需要是引用：原因有很多，例如每次<<可能修改流状态，以及流对应的资源未必可以复制

全局版本： `ostream& operator<<(ostream&, int)`

自增减运算符

以自增为例

- 比较特殊：分为前置++a以及后置a++
- 为进行区分，C++规定前置为一元运算符
 - 成员函数：T& operator++(), 全局函数：T& operator++(T&)
- 后置为二元
 - 成员函数：T operator++(int), 全局函数：T operator++(T&, int)

这个引用的必要性：

```
++ ++ ++ a;
```

不存在a++ ++;的用法，
因此不需要&

需要多一个没用的int参数（仅作为区分）！

关于左值和右值

关于左值和右值

回答“为什么无法做/不允许做`a++ ++`”

一种比较直观（但不完全准确）的定义：

一个判别标准是支持取地址操作`&`

- 左值：具有存储空间的值（存在于某个变量中）
- 右值：其他值，主要是临时值，例如
 - 常量：10, “abc”
 - 函数返回的非引用值，如`int f() {return 1;}`，然后调用`f()`得到的值就是右值

关于左值和右值 (cont.)

注意：

=, [], (), -> 必须在类里定义，而不能是全局的

赋值号左边的永远必须是左值

现在可以理解这项规定的（其中一项）原因：
需要确保第一操作数是左值

- 例如 `111 = a` 是不行的；`int f() {return 1;}` 后，调用 `f() = 2` 也是不行的

一般规则：左值可以直接转化为右值，但右值无法转化成左值

- 例如 `c = a + b`，`+` 两边都应该是右值，但变量（左值）`a` 和 `b` 可自动转化成右值

引用&是左值

C++引用&类型：

- 定义时初始化的值必须为左值：因此`int &a = 5`是不行的，因为5不是左值
- 引用类型会被当作左值：因此可以有之前`pd[i] = 5`这样的用法

即使是通常为右值的地方，如函数返回值

回忆：`int& operator[] (int i);`

- 这同时引入了一个问题：`int g() {return 1;}`无法用于`void f(int &a)`的参数

`f(g());`编译错误，因为`g()`返回右值

const引用与右值

- 但是加const, 即`int f(const int& a)`后`f(g())`就可以使用了

内部定义了临时变量转化成左值,
之后再定义const引用

- 思考: 复制构造的两个版本`X(X&)`和`X(const X&)`的区别?
 - 想要这种用法: `X x = func();`应该采用哪种?

a++ VS ++a

a++返回的是右值

因为a++先返回a原本值，然后才+1，原本值已经没有存储空间了，是右值

- 解释：因为a++先返回a原本值才+1，但原本值已经无存储空间了，是右值
- 因此a++ ++的写法是不行的（右值无法定义++）

++a返回的是左值

- 解释：自增后返回的是a现在的引用，是左值
- 当然C++也可以把这里设计成右值，但这样无法使用++ ++a这样的写法

右值引用

- 动机:

- 设调用String x = func(), 这种用法会调用复制构造, 复制过程产生很大代价
- 然而: func()返回的是右值, 在Stringx = func()调用之后就会被销毁
- 因此: 可以把func()返回值内的arr“**移动**”给x, 然后销毁, 少发生一次复制

```
struct String {  
    char* arr; int len = 0;  
    String() {arr = new char[10000];}  
    String(const String& s) : String() {  
        len = s.len;  
        for (int i = 0; i < s.len; i++)  
            arr[i] = s.arr[i];  
    }  
};
```

我们想要这样一种复制构造: 只是把参数的内容移动过来, 而不进行复制

```
String(/*某种特殊类型*/ s) {  
    len = s.len;  
    arr = s.arr;  
    s.arr = NULL;  
}
```

难点不在于内部实现上: 难点在于如何准确区分何时调用这种构造, 因为很多时候我们不想移动, 而是仍然想要深复制

“右值引用”类型 (C++11)
就是用于区分String x = func()时才调用的“特殊类型”

注意：

String&&型只能匹配到右值
因此若只有String&&复制构造而没有String&的
则会导致String x = y这种用法编译错误

右值引用类型声明的语法：

除此之外和引用效果相同，
即是变量的“别名”且可修改

X&& x

区别于普通（左值）引用，
这里加两个&

X&&作为参数会优先匹配右值，即：

- String(const String&)以及String(String&&)同时存在时
- 调用String x = func()会匹配到String(String&&)版本
- 调用String x = y会匹配到String(const String&)版本

变量y是左值

接受右值引用的复制构造和复制操作符示例

```
struct String {  
    char* arr; int len = 0;  
    String(char* str) {  
        arr = new char[10000]; strcpy(arr, str); len = strlen(str);  
    }  
    String(String&& s) {  
        cout << "rvalue ref" << endl;  
        len = s.len;  
        arr = s.arr;  
        s.arr = NULL;  
    }  
    String& operator=(String&& s) {  
        cout << "rvalue assign" << endl;  
        len = s.len;  
        arr = s.arr;  
        s.arr = NULL;  
        return *this;  
    }  
};
```

这种不复制而仅进行移动操作的实现叫做“move semantics”（移动语义）

把s的数据指针arr设置成NULL很重要！
因为s是右值，之后可能马上被析构，需要确保arr不要被删除

std::move

- std::move(x)返回x的右值引用版本（而不改变x本身）
- 用于：明确知道可以使用右值、且要故意使用右值复制构造/赋值操作符时

```
void naive_swap(String& s, String& t) {  
    String tmp = s;  
    s = t;  
    t = tmp;  
}
```

朴素的swap会发生
3次深复制

```
void move_swap(String& s, String& t) {  
    String tmp = std::move(s);  
    s = std::move(t);  
    t = std::move(tmp);  
}
```

若String没有实现右值版本的
复制构造/赋值，仍可正常调用
正常版本，因为编译器会尝试
将右值引用强制转化成左值

转化成右值之后，所有复制都调用
右值版本，不发生任何深复制！

右值引用是左值还是右值？

- 规则：只有无名字的右值引用才是右值，其余的右值引用依然算左值

- 什么是无名字的右值引用？例如String&& func();

否则可能会导致尚不能销毁的右值调用了右值引用版本的复制/赋值
换句话说，只有无名字右值才能在任何语境下都安全当作右值

```
struct Homework {  
    String title;  
    String content;  
    Homework(Homework&& h) {  
        title = std::move(h.title);  
        content = std::move(h.title);  
    }  
};
```

h在这里当作左值，因此需要调用std::move才能成为右值引用，从而调用右值引用版本的复制构造/赋值符

右值引用小结

- 右值引用主要用例是复制构造和赋值操作符避免不必要的复制
 - 用于其他函数上虽然可行，但是一般不提倡
- C++11的标准库已经全部重写支持右值引用，因此诸如swap等操作都十分高效
- 右值引用还有很多其他细节和需要注意的，这里不再展开
 - 可以参考《Effective Modern C++》一书
- 请回顾整个本节，用左/右值去解释各种运算符重载的标准写法的原因