创建与销毁

(OOP)

刘知远

liuzy@tsinghua.edu.cn

http://nlp.csai.tsinghua.edu.cn/~lzy/

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 自定义类与对象：数据成员、成员函数、访问权限，this指针
- 函数重载
- 宏与内联函数

本讲内容提要

- 4.1 构造函数
- 4.2 析构函数
- 4.3 对象的构造与析构时机(局部对象和全局对象)
- 4.4 引用
- 4.5 运算符重载

如何使含有对象的程序更可靠?

- 对用户定义类型进行严格的类型检查
- 隐藏实现，防止受到不必要的干扰
- 对象的初始化和清除，需要自动进行
  - 忘记初始化或清除变量可能会导致程序崩溃。
  - 由类生成的对象是一种新型的变量，也要初始化。
  - 由于隐藏实现（访问权限控制），对象的有些私有数据成员只有类的设计者知道，而且只允许类的成员函数访问。
  - 尽管可以由通过显式调用对象成员函数来初始化对象，但这种做法缺少强制性，因而容易被程序员遗忘。

如何使含有对象的程序更可靠?

- 结论
  - 如何进行初始化和清除(HOW)，应由类设计者决定。
  - 何时进行初始化和清除(WHEN)，应由编译器来决定。

构造函数：对象的“生”

- 对象的“生”（初始化工作）是由编译器在创建对象处自动生成调用构造函数的代码来完成的。
- 构造函数是类的特殊成员函数，它用来确保类的每个对象都能正确地初始化。

构造函数

- 构造函数没有返回值类型，函数名与类名相同
- 类的构造函数可以重载，即可以使用不同的函数参数进行对象初始化

```
class Student {
    int ID;
public:
    Student(int id) { ID = id; }
    Student(int year, int order) {
        ID = year * 10000 + order;
    }
    ...
};
```

构造函数的初始化列表

- 构造函数可以使用初始化列表初始化成员数据
- 该列表在定义构造函数时使用，位置在函数体之前、函数参数列表之后，以冒号作开头。
- 使用“数据成员(初始值)”的形式

```
class Student {
 int ID;//声明
public:
    Student(int id) : ID(id) { }
    Student(int year, int order) {
          ID = year * 10000 + order;
    }
    ...
};
```

构造函数的初始化列表

- 初始化列表的成员是按照声明的顺序初始化的，而不是按照出现在初始化列表中的顺序
- 在下面的代码中，编译器先初始化ID1，再初始化ID2，因此ID1的值将不可预测

```
class Student {
 int ID1; //声明
 int ID2; //声明
public:
    Student(int id) : ID2(id), ID1(ID2) { }
    ...
};
```

构造函数的初始化列表

- 在构造函数的初始化列表中，还可以调用其他构造函数，称为“委派构造函数”

```
class Info {
public:
    Info() { Init(); }
    Info(int i) : Info() { id = i; }
    Info(char c) : Info() { gender = c; }
private:
    void Init() { .... }// 其他初始化
    int id;
    char gender;
    ...
};
```

构造函数

■ 就地初始化

- C++11之前，类中的**非静态成员变量**不能在类定义的时候进行初始化，它们的初始化操作只能通过构造函数来进行。
- C++11支持如下初始化操作：

```
class A {
private:
    int a = 1; //声明+初始化
    double b {2.0}; //声明+初始化
public:
    A() {}
    A(int i):a(i) {}
    A(int i, double j):a(i), b(j) {}
};
```

默认构造函数

- 不带任何参数、或每个形参提供默认实参的构造函数，被称为“默认构造函数”，也称“缺省构造函数”

```
class A {
private:
    int a = 1;
    double b {2.0};
public:
    A() {} //定义默认构造函数，或者 A(int i = 0):a(i) {}
    A(int i):a(i) {}
    A(int i, double j):a(i), b(j) {}
};
```

默认构造函数

- 使用默认构造函数（没有参数）来生成对象时，对象定义的格式为：

```
ClassName a;      //调用默认构造函数
ClassName b = ClassName();
                  //同样调用默认构造函数
```

注意区分：

```
ClassName c();
      //这声明了一个返回值为ClassName的函数
```

默认构造函数

- 在类的构造函数中，除了执行函数体内声明的语句，编译器还会做一些额外操作

```
class A {
public:
    A() { cout << "A()" << endl; }
};
class B {
public:
    A a;
    B() { cout << "B()" << endl;}
};
B b;
```

输出：

```
A()
B()
```

- 例如会自动调用成员变量的默认构造函数
  - 先调用成员变量的构造，再执行自己的构造函数

默认构造函数

- 隐式定义的默认构造函数
  - 有时候我们没有手动定义默认构造函数，但我们仍然能够按上述方式定义变量
  - 这是因为编译器帮我们隐式地合成了一个默认构造函数

```
class A {
public:
    int data = 0;
};

A a;
```

等价于

```
class A {
public:
    int data = 0;
    A() {}
};

A a;
```

单选题 1分

设置

关于下列程序的说法，正确的是

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout<<"A()"<<endl; }
    A(int x) { cout << "A(int)" << endl; }
};
class B {
    A a;
public:
    B(int x=1): a(x) {}
};
int main(){
    B b;
}
```

A 输出A(int)

B 输出A()

C 编译错误

提交

默认构造函数

- 隐式定义的默认构造函数
  - 若用户已经定义了其他构造函数，编译器将不会隐式合成默认构造函数

```
class A {
private:
   int a = 1;
   double b {2.0};
public:
   A(int i):a(i) {}
};

A a;  //编译错误
```

默认构造函数

■ 显式声明默认构造函数

- 出于某些需要，我们可以手动指定生成默认版本的构造函数：即便其他构造函数存在，编译器也会定义隐式默认构造函数

```
class A {
private:
    int a = 1;
    double b {2.0};
public:
    A() = default;    // C++11起
    A(int i):a(i) {}
};

A a;
```

默认构造函数

- 显式删除构造函数
  - 有时候，我们需要显式地声明禁用某种构造函数。
  - 如果我们定义类如下，会出现什么问题?

```
class A {
private:
    int a = 1;
    double b {2.0};
    char c = 'c';
public:
    A() = default;
    A(int i):a(i) {}
};

A a('c');
```

默认构造函数

- 显式删除构造函数
  - 按照下面方法生成类对象，编译和执行都不会报错。
  - 此时'c'先被转换成int型值，然后调用构造函数A(int i)。

```
class A {
private:
    int a = 1;
    double b {2.0};
    char c = 'c';
public:
    A() = default;
    A(int i):a(i) {}
};

A a('c');
```

默认构造函数

显式删除构造函数

- 从正确性上讲，这样的代码没有问题，但是从工程的角度讲，这是很危险的行为。因为在开发者看来，用字符初始化应该是未定义的行为。

```
class A {
private:
    int a = 1;
    double b {2.0};
    char c = 'c';
public:
    A() = default;
    A(int i):a(i) {}
};

A a('c');
```

默认构造函数

显式删除构造函数

- 使用delete显式地删除构造函数，避免产生未预期行为的可能性。

```
class A {
private:
    int a = 1;
    double b {2.0};
    char c = 'c';
public:
    A() = default;
    A(int i):a(i) {}
    A(char ch) = delete;
};

A a('c'); // 编译错误
```

默认构造函数

- 默认构造函数
  - 默认构造函数在什么情况下会被隐式定义?
  - 默认构造函数的行为?
  - 参考 https://zh.cppreference.com/w/cpp/language/default_constructor

- 隐式定义的默认构造函数还常出现在继承和虚函数的问题中（以后内容）

对象数组的初始化

- 无参定义对象数组，必须要有默认构造函数

```
A a[50]; // 定义了一个具有50个元素的A类对象数组
```

- 如果构造函数只有一个参数

```
A a[3] = {1, 3, 5}; // 三个实参分别传递给3个数组元素的构造函数
```

- 如果构造函数有多个参数

```
A a[3] = {A(1, 2), A(3, 5), A(0, 7)}; // 构造函数有两个整型参数
```

析构函数：对象的“死”

- 对象的“死”（清除和释放资源）是由编译器在对象作用域结束处自动生成调用析构函数代码来完成的。
  - 当执行到“包含对象定义范围结束处”时，编译器自动调用对象的析构函数。
  - 动态分配的内存是一种典型的需要释放的资源。
- 清除对象占用的资源是无条件的，不需要任何选项。因此，析构函数没有参数，且只有一个（即清除方式唯一）。

析构函数

- 一个类只有一个析构函数，名称是“~类名”，没有函数返回值，没有函数参数。
- 编译器在对象生命期结束时自动调用类的析构函数，以便释放对象占用的资源，或其他后处理

```
class ClassRoom {
    int num;
    int* ID_list;
public:
    ClassRoom() : num(0), ID_list(nullptr) {}
    ...
    ~ClassRoom() {  // 析构函数
        if (ID_list) delete[] ID_list; // 释放内存
    }
};
```

析构函数

- 和默认构造函数一样，析构函数除了执行函数体内声明的语句，编译器还会做一些额外操作

```
class A {
public:
    ~A() { cout << "~A()" << endl; }
};
class B {
public:
    A a;
    ~B() { cout << "~B()" << endl;}
};
B b;
```

输出：

```
~B()
~A()
```

- 例如会自动调用成员变量的析构函数
  - 先执行自己的析构函数，再调用成员变量的析构

析构函数

- 隐式定义的析构函数
  - 和构造函数类似，当用户没有自定义析构函数时，编译器会自动合成一个隐式的析构函数

```
class ClassRoom {
    int num;
    int* ID_list;
};
```

等价于

```
class ClassRoom {
    int num;
    int* ID_list;
public:
    ~ClassRoom() {}
};
```

- 注意隐式定义的析构函数不会delete指针成员
  - 因此上述例子可能造成内存泄露

析构函数

- 析构函数
  - 析构函数在什么情况下会被隐式定义?
  - 析构函数的行为?
  - 参考 https://zh.cppreference.com/w/cpp/language/destructor

- 除了用户自定义的代码，析构函数还将自动拓展一些行为，之后会在继承、虚函数的部分介绍（以后内容）

局部对象的构造与析构

- 局部对象
  - 在程序执行到该局部对象的代码时被初始化。
  - 在局部对象生命周期结束、即所在作用域结束后被析构。

- 作用域
  - 该变量能够引用的区域
  - 例如，{}将会形成一个作用域

局部对象的构造与析构

```
class Example {
    int index;
public:
    Example(int i): index(i)
       {cout << index << " is created\n";  }
    ~Example() {  cout << index << " is destroyed\n"; }
};
void create_example(int i) {
    Example e(i); // 只在函数内存在
    cout << "Function is over\n";
}
int main() {
    for(int i = 1; i < 3; i++) {
        Example e(0); // 只在当前循环内存在
        create_example(i);
    }
    return 0;
}
```

```
0 is created
1 is created
Function is over
1 is destroyed
0 is destroyed
0 is created
2 is created
Function is over
2 is destroyed
0 is destroyed
```

全局对象的构造与析构

- 全局对象
  - 在main()函数调用之前进行初始化。
  - 在同一编译单元中，按照定义顺序进行初始化。
    - 编译单元：通常同一编译单元就是同一源文件。
  - **不同编译单元中，对象初始化顺序不确定。**
  - 在main()函数执行完return之后，对象被析构。

全局对象的构造与析构

- 尽量少用全局对象
  - 全局对象的构造顺序不能完全确定，所以全局对象之间不能有依赖关系，否则会出现问题
  - 全局对象会增大代码的耦合性，导致程序难以复用或者测试
  - 使用参数来替代全局对象

```
void foo() {
    input.doSomething();
}
Input input;
int main() {
    foo();
}
```

```
void foo(Input input) {
    input.doSomething();
}
int main() {
    Input input;
    foo(input);
}
```

单选题 1分

设置

下列程序的运行结果是

```
#include <iostream>
using namespace std;
class A {
    const char* s;
public:
    A(const char* str):s(str) {
        cout << s << " A constructing" << endl;
    }
    ~A() { cout << s << " A destructing" << endl; }
};
class B {
public:
    B() { cout << "B constructing" << endl; }
    ~B() { cout << "B destructing" << endl; }
};
A global_obj("global");
int main() {
    cout << "Entering main..." << endl;
    B local_obj;
    cout << "Exiting main..." << endl;
    return 0;
}
```

A

global A constructing
Entering main...
B constructing
Exiting main...
B destructing
global A destructing

B

Entering main...
global A constructing
B constructing
Exiting main...
global A destructing
B destructing

C

global A constructing
Entering main...
B constructing
Exiting main...
global A destructing
B destructing

提交

引用

- 具名变量的别名：类型名 & 引用名 变量名

例：int v0; int & v1 = v0; v1是变量v0的引用，它们在内存中是同一单元的两个不同名字

- 引用必须在定义时进行初始化，且不能修改引用指向
- 被引用变量名可以是结构变量成员，如s.m

引用

- 创建引用

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    cout << "i=" << i << endl; // i=1
    int& j = i;  // j是初始化为的i的int引用
                 // i和j是同一个变量的两个别名
    cout << "j=" << j << endl; // j=1
    i = 2;
    cout << "j=" << j << endl; // j=2
    j = 3;
    cout << "i=" << i << endl; // i=3
    return 0;
}
```

引用

- 函数参数可以是引用类型，表示函数的形式参数与实际参数是同一个变量，改变形参将改变实参。如调用以下函数将交换实参的值：

```
void swap(int& a, int& b)
{  int tmp = b; b = a; a = tmp; }
```

- 函数返回值可以是引用类型，但不得是函数的临时变量

比较：参数中的值、引用

```
void swap(int a, int b)
{
    int tmp =  a;
    a =  b;
    b = tmp;
}

swap(a, b);
```

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

swap(&a, &b);
```

```
void swap(int &a, int &b)
{
    int tmp =  a;
    a =  b;
    b = tmp;
}

swap(a,  b);
```

引用

- 把引用作为返回值

```
old a[0]=1
new a[0]=2
old a[1]=3
new a[1]=4
old a[2]=5
new a[2]=6
```

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

int a[3] = {1,3,5};
int& get(int i) { return a[i]; } // 返回a[i]的引用
int main() {
    for(int i = 0; i < 3; i++) {
        cout << "old a[" << i << "]=" << get(i) << endl;
        get(i) += 1;
        cout << "new a[" << i << "]=" << get(i) << endl;
    }
    return 0;
}
```

引用

和指针的区别

- 不存在空引用。引用必须连接到一块合法的内存。
- 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
- 引用必须在创建时被初始化。指针可以在任何时间被初始化。

反思：为什么要“引用”？

- The easiest way to think about a reference is as a fancy pointer. You never have to wonder whether it’s been initialized(the compiler enforces it) and how to dereference it(the compiler does it).

  —《Thinking in C++》

- 引用的优势：更灵活地支持运算符重载
- 引用的特性：创建时必须初始化、初始化后便不能指向其他对象，不存在空引用

单选题 1分

设置

右侧代码的输出结果是

A 1_4

B 4_7

C 4_4

D 运行错误

```
class Int {
public:
    int data;
    Int() { data = 1; }
    Int(int i): data(i) {}
};
void fuc1(Int& a, Int b) {
    a.data += b.data;
}
Int& fuc2(Int& a, Int b) {
    fuc1(a, b);
    Int tmp(a.data + b.data);
    return tmp;
}
int main() {
    Int a, b(3);
    Int& f = fuc2(a, b);
    cout << a.data << "_";
    cout << f.data << endl;
    return 0;
}
```

提交

类的运算符重载

- 为什么需要类运算符重载?
  - 用户自定义类，没有对常用的运算符进行定义，比如想要表示两个类对象相加，无法采用a+b这种方式。
  - 可以采取定义一个add函数的方式，解决这种问题。

```
A add(A a, A b) {
    return A(a.data + b.data);
}
```

  - 但这种实现方式，在调用的时候，会和基础类型差别很大，缺少编程的一致性。需要过多地区分自定义类和基础类别，调用起来也不方便。
  - 因此，我们引入运算符重载

运算符重载

- 运算符重载需要按规则声明执行该运算的函数
  - 例如 + 对应 operator+
- 运算重载一般有两种方式（注意参数不同）
  - 全局函数的运算符重载

```
A operator+(A a, A b) {…}
```

  - 成员函数的运算符重载

```
class A{
   int data;
public:
   A operator+(A b) {…};
}
```

运算符重载

```
class A {
public:
    int data;
    A(int i) { data = i; }
    A& operator+=(A& a) { data += a.data; return *this;}
    // A operator+(A& a) {
    //     A new_a(data + a.data);
    //     return new_a;
    // }
};
A operator+(A& a1, A& a2) {
    A new_a(a1.data + a2.data);
    return new_a;
}
int main() {
    A a1(2), a2(3);
    a1 += a2;
    cout << a1.data << endl; // 5
    cout << (a1 + a2).data << endl; // 8
    return 0;
}
```

- 同一运算符（+）只能采用一种实现（紫色或蓝色）

可以重载的运算符

- 双目算术运算符
  - +（加），-(减)，*(乘)，/(除)，%（取模）
- 关系运算符
  - ==(等于)，!=（不等于)，<（小于)，>（大于>，<=(小于等于)，>=(大于等于)
- 逻辑运算符
  - ||(逻辑或)，&&(逻辑与)，!(逻辑非)
- 单目运算符
  - +（正)，-(负)，*(指针)，&(取地址)
- 自增自减运算符
  - ++(自增)，--(自减)

可以重载的运算符

- 位运算符
  - |（按位或），&（按位与），~(按位取反），^(按位异或)，，<<（左移），>>(右移)
- 赋值运算符
  - =，+=，-=，*=，/=，%=，&=，|=，^=，<<=，>>=
- 空间申请与释放
  - new，delete，new[]，delete[]
- 其他运算符
  - ()(函数调用)，->(成员访问)，,(逗号)，[](下标)

前缀与后缀的++、--

- 前缀运算符重载声明
  - ClassName& operator++();
  - ClassName& operator--();
- 后缀运算符重载声明
  - ClassName operator++(int dummy);
    - ++a ⇔ operator++(a)
    - a++ ⇔ operator++(a,int)
  - ClassName operator--(int dummy);
- 通过在函数体中没有使用的哑元参数dummy来区分前缀与后缀的同名重载

```
int fun(int, int a){ return a/10*10; }
```

++前缀、后缀语义

前缀语义:

```
int a = ++b; //先完成b+1操作，再赋值
```

后缀语义:

```
int a = b++; //先完成赋值，再b+1操作
```

前缀运算符++重载示例

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

class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test& operator++ () {
    ++data;
    return *this;
  }
};

int main() {
  Test test(1);
  ++test;
  return 0;
}
```

单选题 1分

设置

类比前缀运算符++重载，填入下列代码(1)处能实现后缀运算符++重载的是：

```
class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test operator++ (int) {
     // (1)
  }
};
```

A
```
++data;
return Test(data);
```

B
```
return Test(++data);
```

C
```
Test test(data);
++data;
return test;
```

提交

后缀运算符++重载示例

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

class Test {
public:
  int data = 1;
  Test(int d) {data = d;}
  Test operator++ (int) {
    Test test(data);
    ++data;
    return test;
  }
};

int main() {
  Test test(1);
  test++;
  return 0;
}
```

这里定义了一个哑元参数int，但是没有任何变量名，所以函数实现中永远不会用到该变量

http://blog.csdn.net/megustas_jjc/article/details/53583672

前缀与后缀的++、--

- 也可以使用全局重载的方式

```
class A {
public:
    int data;
    A() { data = 0; }
    A(int i) { data = i; }
};

A operator++(A& a) {
    ++a.data;
    return a;
}
```

```
A operator++(A& a, int) { //哑元
    A new_a(a.data);
    ++a.data;
    return new_a;
}

int main() {
    A a(1);
    cout << (++a).data << endl; // 2
    cout << (a++).data << endl; // 2
    cout << a.data << endl; // 3
    return 0;
}
```

函数运算符 ( ) 重载

- 在自定义类中也可以重载函数运算符()，它使对象看上去象是一个函数名

```
ReturnType operator() (Parameters) {
    ...
}

ClassName Obj;
Obj(real_parameters); //注意不是调用构造函数!
// → Obj.operator() (real_parameters);
```

函数运算符（）重载示例

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

class Test {
public:
  int operator() (int a, int b) {
    cout << "operator() called. " << a << ' ' << b << endl;
    return a + b;
  }
};

int main() {
  Test sum;
  int s = sum(3, 4); /// sum对象看上去象是一个函数，故也称“函数对象”
  cout << "a + b = " << s << endl;

  int t = sum.operator()(5, 6);
  return 0;
}
```

数组下标运算符[ ]重载

- 函数声明形式

  返回类型 operator[] (参数);

- 如果返回类型是引用，则数组运算符调用可以出现在等号左边，接受赋值，即

  Obj[index] = value;

- 如果返回类型不是引用，则只能出现在等号右边

  Var = Obj[index];

数组下标运算符重载示例

```
#include <iostream>   // cout
#include <string>     // strcmp
using namespace std;

char week_name[7][4] = {  "mon", "tu", "wed",
                          "thu", "fri", "sat", "sun"};

class WeekTemp{
    int temp[7];
public:
    int& operator[] (const char* name) // 字符串作下标
注意返回值{
        for (int i = 0; i < 7; i++) {
            if (strcmp(week_name[i], name) == 0)
                return temp[i];
        }
    }
};
```

数组下标运算符重载示例

```
/// 关于数组下标运算符重载的测试
int main()
{
    WeekTemp beijing;
    beijing["mon"] = -3;
    beijing["tu"] = -1;
    cout    << "Monday Temperature: "
            << beijing["mon"] << endl;

    return 0;
}
```

运行输出结果

Monday Temperature: -3

只能成员函数重载的运算符

- =,[],(),->只能通过成员函数来重载

```
class cls {
public:
    int data;
    cls(int i) : data(i) {}
};
cls& operator[](cls& c1, cls& c2) { return c2; }
```

编译错误：error: ‘cls& operator[](cls&, cls&)’ must be a nonstatic member function

只能成员函数重载的运算符

- 为什么这么做?
  - 当没有自定义operator=时，编译器会自动合成一个默认版本的赋值操作
  - 在类内定义operator=，编译器则不会自动合成
  - 如果允许使用全局函数重载，可能会对是否自动合成产生干扰

假设能全局重载的例子

cls.h:

```
class Cls {
public:
    int data;
    Cls(int _data): data(_data) {}
};
```

main.cpp:

```
int main(){
    Cls a, b(3);
    a = b; //?
    return 0;
}
```

func.cpp:

```
Cls& operator=(Cls &a, Cls b)
{
    cout << "operator=" << endl;
    a.data = b.data;
    return a;
}
```

调用自动合成运算？还是全局重载运算？

C++标准禁止了operator=的全局重载

单选题 1分

设置

关于运算符重载，下列说法不正确的是

A sum 为自定义类的一个变量，可以通过 sum.operator()(5,6); 去调用()运算符重载函数。

B 通过重载[]运算符： int operator[](const char* name); 使得我们可以使用 Beijing[“mon”]= -3进行赋值。

C ClassName& operator++(); 为前缀自增运算符的重载声明。

D 运算符 () 必须作为成员函数重载。

提交

对象输入输出 —— 流运算符重载

- 用户自定义的类，虽然可以像内置类型那样定义变量（对象），但想要使用流运算符输入、输出对象，则还需要为类定义流运算符重载。
- 如：

```
Test obj;
cin >> obj;
...
cout << obj;
...
```

流运算符重载函数的声明

```
istream& operator>> (istream& in, Test& dst );

ostream& operator<< (ostream& out, const Test& src );
```

- 函数名为：operator>> 和 operator<<
- 不修改istream和ostream类的情况下，只能使用全局函数重载
- 返回值为：istream& 和 ostream&，均为引用
- 参数分别：流对象的引用、目标对象的引用。对于输出流，目标对象还是常量。

流运算符重载示例

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

class Test {
    int id;
public:
    Test(int i) : id(i) { cout << "obj_" << id << " created\n"; }

    friend istream& operator>> (istream& in, Test& dst);
    friend ostream& operator<< (ostream& out, const Test& src);
};
istream& operator>> (istream& in, Test& dst) {
        in >> dst.id;
        return in;
}
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id << endl;
    return out;
}

int main() {
    Test obj(1);
    cout << obj;  // operator<<(cout,obj)
    cin >> obj;  // operator>>(cin,obj)
    cout << obj;
    return 0;
}
```

思考题:
为什么形参和返回值都是引用?

为什么要引用?

```
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id << endl;
    return out;
}


//测试代码
cout << obj << obj2 << obj3 << endl;

//等价于
ostream& out = operator<<(cout, obj); //return cout;
ostream& out1 = operator<<(out, obj2); //return cout;
ostream& out2 = operator<<(out1, obj3); //return cout;
```

课后阅读

- 《c++编程思想》
  - 初始化与清除，p156-p169

结束