引用与复制

（OOP）

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/hml/

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 友元
- 静态成员与常量成员
  - 初始化方法和初始化依赖
- 对象的构造与析构时机
  - 常量/静态对象的构造/析构顺序
  - 参数对象的构造/析构顺序
- 对象的new和delete

本讲内容提要

- 6.1 常量引用
- 6.2 拷贝构造函数
- 6.3 右值引用
- 6.4 移动构造函数
- 6.5 赋值运算符
- 6.6 类型转换

回顾：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
- 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

回顾：常量成员和常量对象

- 使用const修饰的数据成员，称为类的常量数据成员，在对象的整个生命周期里不可更改，且只能在构造函数的初始化列表中被设置，不允许在函数体中通过赋值来设置
- 成员函数也能用const来修饰，该成员函数的实现语句不能修改类的数据成员，即不能改变对象状态（内容）
- 若对象被定义为常量，则它只能调用以const修饰的成员函数

```
class Student {
    const int ID; //常量数据成员
public:
    Student(int id) : ID(id) {}  //通过初始化列表设置
    int studentID() const { return ID; } //常量成员函数
};
```

参数中的常量和常量引用

- 最小特权原则：给函数足够的权限去完成相应的任务，但不要给予他多余的权限。
  - 例如函数void add(int& a, int& b)，如果将参数类型定义为int&，则给予该函数在函数体内修改a和b的值的权限

- 如果我们不想给予函数修改权限，则可以在参数中使用常量/常量引用
  - void add(const int& a, const int& b)

此时函数中仅能读取a和b的值，无法对a, b进行任何修改操作。

拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它的参数是语言规定的，是同类对象的常量引用

- 拷贝构造函数示例：

```
class Person {
    int id;
    ...
public:
    Person(const Person& src) { id = src.id;  ... }
    ...
};
```

- 作用：用参数对象的内容初始化当前对象

拷贝构造函数

- 拷贝构造函数被调用的三种常见情况：

1、用一个类对象定义另一个新的类对象

```
Test a; Test b(a);
Test c = a;
```

2、函数调用时以类的对象为形参

```
Func(Test a)
```

3、函数返回类对象

```
Test Func(void)
```

编译器会自动调用“拷贝构造函数”，在已有对象基础上生成新对象。

拷贝构造函数

- 类的新对象被定义后，会调用构造函数或拷贝构造函数。如果调用拷贝构造函数且当前没有给类显式定义拷贝构造函数，编译器将自动合成，且采用位拷贝(Bitcopy)，即直接使用赋值运算符拷贝类的所有数据成员。

拷贝构造函数

- 位拷贝示例

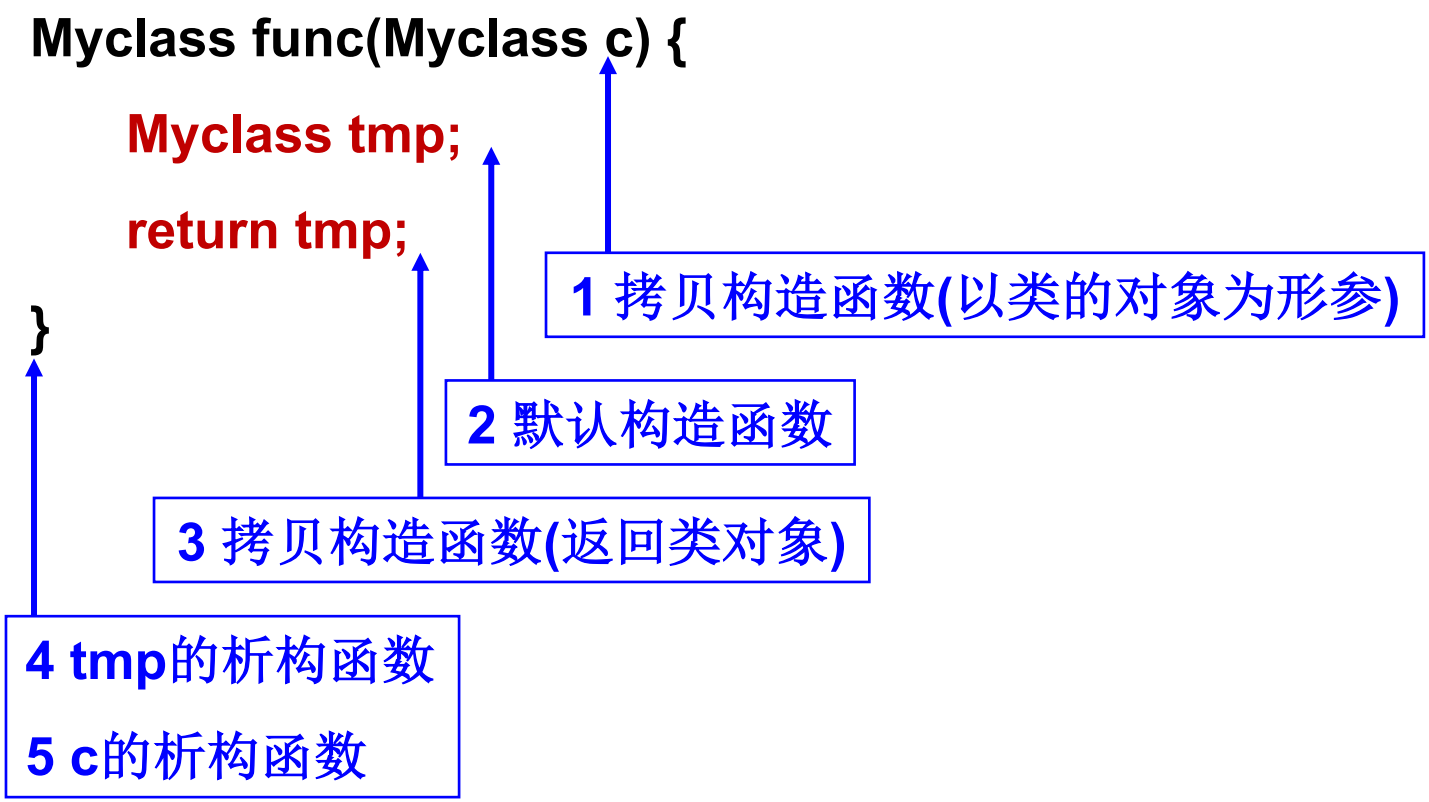
```
class Test {
    int data;
public:
    Test() { } //默认构造函数
    ~Test() { } //析构函数
};
```

  - 上述Test类未显式定义拷贝构造函数，编译器将自动合成。当定义Test类的对象时(Test a; Test b=a;)，自动合成的拷贝构造函数采用位拷贝，即使用赋值运算符初始化b的数据成员b.data=a.data

- 注意：位拷贝在遇到指针类型成员时可能会出错，导致多个指针类型的变量指向同一个地址

拷贝构造函数：执行顺序

- 以下述的func函数为例，调用该函数时，函数中各类构造函数和析构函数的执行顺序如下：



拷贝构造函数：实例1

思考以下代码的运行结果：

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

class Test {
public:
    Test() { //构造函数
        cout << "Test()" << endl;
    }
    Test(const Test& src) { //拷贝构造
        cout << "Test(const Test&)" <<
endl;
    }
    ~Test() { //析构函数
        cout << "~Test()" << endl;
    }
};
```

```
Test copyObj(Test obj) {
    cout << "func()..." <<
endl;
    return Test();
}

int main() {
    cout << "main()..." <<
endl;
    Test t;
    t = copyObj(t);
    return 0;
}
```

拷贝构造函数：实例1

```
Test copyObj(Test obj) {
    cout << "func()..." <<
endl;
    return Test();
}

int main() {
    cout << "main()..." <<
endl;
    Test t;
    t = copyObj(t);
    return 0;
}
```

```
main()...
Test() //main函数内初始化 Test
Test(const Test&)//func参数 拷贝构造
func()...
Test()  //初始化 Test类的对象
Test(const Test&) //返回时拷贝构造
~Test()
~Test()
~Test()
~Test()
```

注意采用编译选项，禁止编译器进行返回值优化：

```
g++ test.cpp --std=c++11 -fno-elide-constructors -o test
```

拷贝构造函数：实例2

思考以下代码的运行结果：

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

class Pointer {
    int *m_arr;
    int m_size;
public:
    Pointer(int i):m_size(i) { //构造
        m_arr = new int[m_size];
        memset(m_arr, 0, m_size*sizeof(int));
    }
    ~Pointer(){delete []m_arr;} //析构
    void set(int index, int value) {
        m_arr[index] = value;
    }
    void print();
};
```

```
void Pointer::print()
{
    cout << "m_arr: ";
    for (int i = 0; i < m_size; ++ i)
    {
        cout << " " << m_arr[i];
    }
    cout << endl;
}

int main() {
    Pointer a(5);
    Pointer b = a; //调用默认的拷贝构造
    a.print();
    b.print();
    b.set(2, 3);
    b.print();
    a.print();
    return 0;
}
```

拷贝构造函数：实例2

```
int main() {
 Pointer a(5);
 Pointer b = a; //调用默认的拷贝构造
 a.print();
 b.print();
 b.set(2, 3);
 b.print();
 a.print();
}
```

```
m_arr: 0, 0, 0, 0, 0 //a.print()
m_arr: 0, 0, 0, 0, 0 //b.print()
m_arr: 0, 0, 3, 0, 0 //b.print()
m_arr: 0, 0, 3, 0, 0 //a.print()
```

以上代码有什么问题？
为什么会有这样的结果？

- 位拷贝会使得对象a, b的指针成员m_arr指向同一个内存地址
- 当类内含指针类型的成员时，为避免指针被重复删除，不应使用默认的拷贝构造函数

拷贝构造函数

- 拷贝构造有什么问题?
  - 当对象很大的时候?
  - 当对象含有指针的时候?
- 频繁的拷贝构造会造成程序效率的显著下降

拷贝构造函数

- 正常情况下，应尽可能避免使用拷贝构造函数
- 解决方法：

（1）使用引用/常量引用传参数或返回对象；

（2）将拷贝构造函数声明为private；

（3）用delete关键字显式地让编译器不生成拷贝构造函数的默认版本。

```
class MyClass
{
  public:
    MyClass()=default;
    MyClass(const MyClass&)=delete;
  ......
}
```

右值引用

- 多数情况下，我们更需要对象的“移动”，而非对象的“拷贝”。C++11为此提供了一种新的构造函数，即移动构造函数。
- 为理解移动构造函数的工作原理，首先要引入C++11的另一个新特性——右值引用。

右值引用

左值和右值

- 左值：可以取地址、有名字的值。
- 右值：不能取地址、没有名字的值;常见于**常值、函数返回值、表达式**

```
int a = 1;
int b = func();
int c = a + b;
```

- 其中a、b、c为左值，1、func函数返回值、a+b的结果为右值。
- 左值可以取地址，并且可以被&引用(左值引用)

```
int *d = &a;        ☺    int &d = a;        ☺

int *e = &(a + b); ☹    int &e = a + b; ☹
```

右值引用

■右值引用

- 虽然右值无法取地址，但可以被&&引用(右值引用)

```
int &&e = a + b;
```
☺

- 右值引用无法绑定左值

```
int &&e = a;
```
☹

■总结

- 左值引用能绑定左值，右值引用能绑定右值
- **例外：常量左值引用能也绑定右值**（为什么这么设计？）

```
const int &e = 3;
```
☺

进一步阅读：
https://www.zhihu.com/question/22111546
https://www.zhihu.com/question/40238995

引用的绑定

- 常见的引用绑定规则

	非常量左值	常量左值	右值
非常量左值引用	√		
常量左值引用	√	√	√
右值引用			√

- 注意：所有的引用（包括右值引用）本身都是左值，结合该规则和上表便可判断各种构造函数、赋值运算符中传递参数和取返回值的引用绑定情况。

进一步阅读：
https://www.justsoftwaresolutions.co.uk/cplusplus/core-c++-lvalues-and-rvalues.html#lvalue-references

右值引用示例

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

void ref(int &x) {
    cout << "left " << x << endl;
}

void ref(int &&x) {
    cout << "right " << x << endl;
}

int main() {
    int a = 1;
    ref(a);
    ref(2); //2是一个常量
    return 0;
}
```

右值引用示例

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

void ref(int &x) {
    cout << "left " << x << endl;
}

void ref(int &&x) {
    cout << "right " << x << endl;
}

int main() {
    int a = 1;
    ref(a);
    ref(2); //2是一个常量
    return 0;
}
```

```
Output：
left 1
right 2
```

int &x代表左值引用参数；
int &&x代表右值引用参数，对2的引用是右值引用。

如果没有定义 ref(int &&x) 函数会发生什么？

编译错误：

```
[Error] invalid initialization of
non-const reference of type 'int&'
from an rvalue of type 'int'
```

右值引用示例

```
#include <iostream>

using namespace std;

void ref(int &x) {
    cout << "left " << x << endl;
}

void ref(int &&x) {
    cout << "right " << x << endl;
    ref(x); //调用哪一个函数？
}

int main() {
    ref(1); //1是一个常量
    return 0;
}
```

右值引用示例

```
#include <iostream>

using namespace std;

void ref(int &x) {
    cout << "left " << x << endl;
}

void ref(int &&x) {
    cout << "right " << x << endl;
    ref(x); //调用哪一个函数？
}

int main() {
    ref(1); //1是一个常量
    return 0;
}
```

Output：
right 1
left 1

ref(1)首先调用ref(int &&x)函数，此时右值引用x为左值，因此ref(x)调用ref(int &x)函数。

移动构造函数

- 右值引用可以延续即将销毁变量的生命周期，用于构造函数可以提升处理效率，在此过程中尽可能少地进行拷贝。
- 使用右值引用作为参数的构造函数叫做移动构造函数。

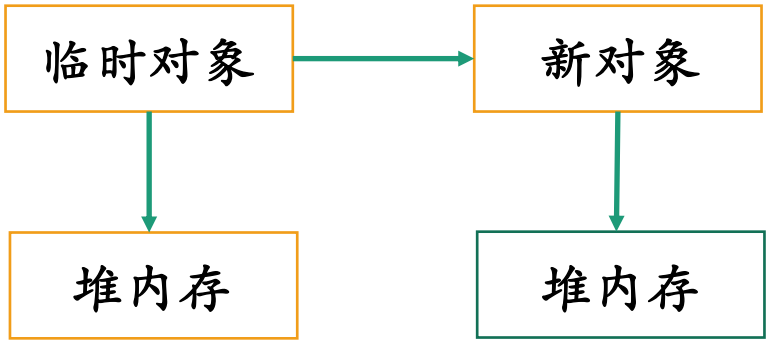
移动构造函数

- 拷贝构造函数
  - `ClassName(const ClassName& VariableName);`

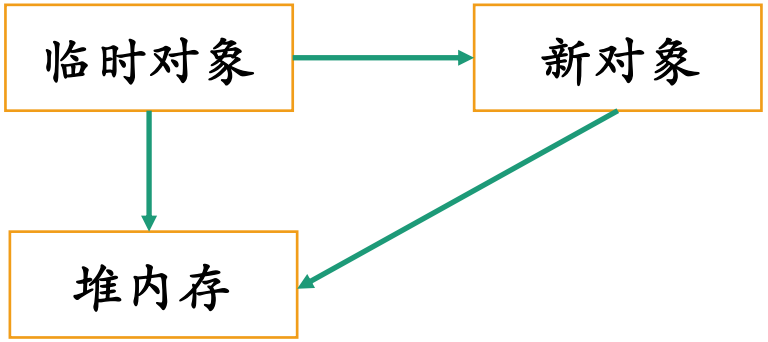
- 移动构造函数
  - `ClassName(ClassName&& VariableName);`

移动构造函数

拷贝构造函数



移动构造函数



- 移动构造函数与拷贝构造函数最主要的差别就是类中堆内存是重新开辟并拷贝，还是直接将指针指向那块地址。

- 对于一些即将析构的临时类，移动构造函数直接利用了原来临时对象中的堆内存，新的对象无需开辟内存，临时对象无需释放内存，从而大大提高计算效率。

移动构造函数：实例

```
class Test {
public:
    int * buf; //// only for demo.
    Test() {
            buf = new int[10]; //申请一块内存
            cout << "Test(): this->buf @ " << hex << buf << endl;
    }
    ~Test() {
            cout << "~Test(): this->buf @ " << hex << buf << endl;
            if (buf) delete buf;
    }
    Test(const Test& t) : buf(new int[10]) {
            for(int i=0; i<10; i++)
                    buf[i] = t.buf[i]; //拷贝数据
            cout << "Test(const Test&) called. this->buf @ "
                    << hex << buf << endl;
    }
    Test(Test&& t) : buf(t.buf) { //直接复制地址，避免拷贝
            cout << "Test(Test&&) called. this->buf @ "
                    << hex << buf << endl;
            t.buf = nullptr; //将t.buf改为nullptr，使其不再指向原来内存区域
    }
};
```

移动构造函数：实例

```
Test GetTemp() {
    Test tmp;
    cout << "GetTemp(): tmp.buf @ "
         << hex << tmp.buf << endl;
    return tmp;
}
void fun(Test t) {
    cout << "fun(Test t): t.buf @ "
         << hex << t.buf << endl;
}
int main() {
    Test a = GetTemp();
    cout << "main() : a.buf @ " << hex << a.buf << endl;
    fun(a);
    return 0;
}
```

移动构造函数：实例

```
Test GetTemp() {
    Test tmp;
    cout << "GetTemp(): tmp.buf @ "
    << hex << tmp.buf << endl;
    return tmp;
}
void fun(Test t) {
    cout << "fun(Test t): t.buf @ "
    << hex << t.buf << endl;
}
int main() {
    Test a = GetTemp();
    cout << “main() : a.buf @ ” <<
         hex << a.buf << endl;
    fun(a);
    return 0;
}
```

```
Test(): this->buf @ 0x7fa908c04b90
GetTemp(): tmp.buf @ 0x7fa908c04b90
main() : a.buf @ 0x7fa908c04b90
Test(const Test&) called.
     this->buf @ 0x7fa908c04ba0
fun(Test t): t.buf @ 0x7fa908c04ba0
~Test(): this->buf @ 0x7fa908c04ba0
~Test(): this->buf @ 0x7fa908c04b90
```

编译指令:

```
g++ test.cpp --std=c++11 -o test
```

Q:为什么没有调用移动构造函数?
也少调用了几次拷贝构造函数?

A:编译器进行了返回值优化。

移动构造函数：实例

```
Test GetTemp() {
    Test tmp;
    cout << "GetTemp(): tmp.buf @ "
    << hex << tmp.buf << endl;
    return tmp;
}
void fun(Test t) {
    cout << "fun(Test t): t.buf @ "
    << hex << t.buf << endl;
}
int main() {
    Test a = GetTemp();
    cout << “main() : a.buf @ ” <<
         hex << a.buf << endl;
    fun(a);
    return 0;
}
```

```
Test(): this->buf @ 0x7fa908c04b90
GetTemp(): tmp.buf @ 0x7fa908c04b90
main() : a.buf @ 0x7fa908c04b90
Test(const Test&) called.
     this->buf @ 0x7fa908c04ba0
fun(Test t): t.buf @ 0x7fa908c04ba0
~Test(): this->buf @ 0x7fa908c04ba0
~Test(): this->buf @ 0x7fa908c04b90
```

编译指令：

```
g++ test.cpp --std=c++11 -o test
```

返回值优化的两个条件：

1) return 的值类型与函数签名的返回值类型相同；
2) return的是一个局部对象。

*返回值优化的进一步说明可参考：
https://www.zhihu.com/question/27000013/answer/34846612

移动构造函数：实例

```
Test GetTemp() {
    Test tmp;
    cout << "GetTemp(): tmp.buf @ "
    << hex << tmp.buf << endl;
    return tmp;
}
void fun(Test t) {
    cout << "fun(Test t): t.buf @ "
    << hex << t.buf << endl;
}
int main()
{
    Test a = GetTemp();
    cout << “main() : a.buf @ ” <<
        hex << a.buf << endl;
    fun(a);
    return 0;
}
```

```
Test(): this->buf @ 0x7f8951c04b90
GetTemp(): tmp.buf @ 0x7f8951c04b90
Test(Test&&) called. this->buf
    @ 0x7f8951c04b90
~Test(): this->buf @ 0x0 (tmp)
Test(Test&&) called. this->buf
    @ 0x7f8951c04b90 ~a=GetTemp()
~Test(): this->buf @ 0x0 ~GetTemp()
main() : a.buf @ 0x7f8951c04b90
Test(const Test&) called. this->buf
    @ 0x7f8951c04ba0
fun(Test t): t.buf @ 0x7f8951c04ba0
~Test(): this->buf @ 0x7f8951c04ba0
~Test(): this->buf @ 0x7f8951c04b90
```

增加编译选项，禁止编译器进行返回值优化

```
g++ test.cpp --std=c++11 -fno-elide-constructors -o test
```

移动构造函数：实例

```
Test GetTemp() {
    Test tmp;
    cout << "GetTemp(): tmp.buf @ "
    << hex << tmp.buf << endl;
    return tmp;
}
void fun(Test t) {
    cout << "fun(Test t): t.buf @ "
    << hex << t.buf << endl;
}
int main()
{
    Test a = GetTemp();
    cout << “main() : a.buf @ ” <<
         hex << a.buf << endl;
    fun(a);
    return 0;
}
```

```
Test(): this->buf @ 0x7fabf8c04b50
GetTemp(): tmp.buf @ 0x7fabf8c04b50
Test(const Test&) called. this->buf
    @ 0x7fabf8c04b60
~Test(): this->buf @ 0x7fabf8c04b50
Test(const Test&) called. this->buf
    @ 0x7fabf8c04b50
~Test(): this->buf @ 0x7fabf8c04b60
main() : a.buf @ 0x7fabf8c04b50
Test(const Test&) called. this->buf
    @ 0x7fabf8c04b60
fun(Test t): t.buf @ 0x7fabf8c04b60
~Test(): this->buf @ 0x7fabf8c04b60
~Test(): this->buf @ 0x7fabf8c04b50
```

删除移动构造函数、并且禁止编译器优化的输出结果

```
g++ test.cpp --std=c++11 -fno-elide-constructors -o test
```

右值引用：移动语义

■ 如何加快左值初始化的构造速度

- 移动构造函数加快了右值初始化的构造速度。
- 如何对左值调用移动构造函数以加快左值初始化的构造速度?

■ std::move函数

- 输入：左值（包括变量等，该左值一般不再使用）
- 返回值：该左值对应的右值的引用

```
Test a;
Test b = std::move(a) //对于上个实例中定义的Test类，该处
                      调用移动构造函数对b进行初始化
```

- 注意：move函数本身不对对象做任何操作，仅做类型转换，即转换为右值引用。移动的具体操作在移动构造函数内实现。

详细阅读：https://blog.csdn.net/swartz_lubel/article/details/59620868

右值引用：移动语义

- 右值引用结合std::move可以显著提高swap函数的性能。
  - std::move引起移动构造函数或移动赋值运算的调用

```
template <class T>
swap(T& a, T& b) {
      T tmp(a); //copy a to tmp
      a = b; //copy b to a
      b = tmp; //copy tmp to b
}
```

```
template <class T>
swap(T& a, T& b) {
      T tmp(std::move(a));
      a = std::move(b);
      b = std::move(tmp);
}
```

避免3次不必要的拷贝操作

构造函数综合实例

写出以下代码的运行结果：

编译指令加 --std=c++11
-fno-elide-constructors

```
#include <iostream>

class Test {
public:
    Test() {
        printf("Test()\n");
    } //默认构造函数

    ~Test() {
        printf("~Test()\n");
    } //析构函数

    Test(const Test &con) {
        printf("Test(const Test &con)\n");
    } //拷贝构造函数

    Test(Test &&con) {
        printf("Test(Test &&con)\n");
    } //移动构造函数

};

Test func(Test a) {
    return Test();
}

int main() {
    Test a;
    Test b = func(a);
    return 0;
}
```

答案

```
Test func(Test a)
{
  return Test();
}

int main() {
  Test a;
  Test b = func(a);
  return 0;
}
```

我们用(1+)和(1-)这样的形式来对应类的构造和析构。

```
Test()                    //(1+) 执行Test a ;
Test(const Test &con)     //(2+)Test b = func(a);
                          //func(a)传参调用拷贝构造函数
Test()                    //(3+)return Test();
                          //Test()对应的构造函数
Test(Test &&con)          //(4+)return Test();
                          //为了传值调用的移动构造函数
~Test()                   //(3-)return Test();
                          //Test()对应的析构函数
Test(Test &&con)          //(5+)Test b = func(a);
                          //中给b传值时调用的移动构造函数
~Test()                   //(4-)Test b = func(a);
                          //完成赋值后func(a)返回值
                              对应的析构函数
~Test()                   //(2-)Test b = func(a);
                          //参数释放对应的析构函数
~Test()                   //(5-) 析构b
~Test()                   //(1-) 析构a
```

赋值运算符

- 已定义的对象之间相互赋值，在C++中是通过调用对象的“赋值运算符函数”来实现的

```
ClassName& operator= (const ClassName& right) {
    if (this != &right) {// 避免自己赋值给自己
            // 将right对象中的内容复制到当前对象中…
     }
    return *this;
}
```

- 注意区分下面两种代码：

```
ClassName a;
ClassName b;
a = b;
```

```
ClassName a = b;
```

赋值运算符：实例

```
Test& operator= (const Test& right) {
   if (this == &right)  cout << "same obj!\n";
   else {
      for(int i=0; i<10; i++)
         buf[i] = right.buf[i];  //拷贝数据
      cout << “operator=(const Test&) called.\n”;
   }
   return *this;
}
```

赋值重载函数必须要是类的非静态成员函数(non-static member function)，不能是友元函数。

移动赋值运算

- 和移动构造函数原理类似

```
Test& operator= (Test&& right) {
    if (this == &right)  cout << "same obj!\n";
    else {
        this->buf = right.buf;  //直接赋值地址
        right.buf = nullptr;
        cout << "operator=(Test&&) called.\n“;
    }
    return *this;
}
```

- 示例：

```
swap(Test& a, Test& b) {
        Test tmp(std::move(a)); // 第一行调用移动构造函数
        a = std::move(b);    // std::move的结果为右值引用，
        b = std::move(tmp); // 后两行均调用移动赋值运算
}
```

编译器自动合成的函数/运算符

- 类中特殊的成员函数/运算符，即便用户不显式定义，编译器也会根据自身需要自动合成
  - 默认构造函数
  - 拷贝构造函数
  - 移动构造函数（C++11起）
  - 拷贝赋值运算符
  - 移动赋值运算符（C++11起）
  - 析构函数

进一步阅读：https://zh.cppreference.com/w/cpp/language/classes

类型转换

- 当编译器发现表达式和函数调用所需的数据类型和实际类型不同时，便会进行自动类型转换。
- 自动类型转换可通过定义特定的转换运算符和构造函数来完成。
- 除自动类型转换外，在有必要的时候还可以进行强制类型转换。

自动类型转换：方法一

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


class Dst { //目标类Destination
public:
  Dst() { cout << "Dst::Dst()" << endl; }
};


class Src { //源类Source
public:
  Src() { cout << "Src::Src()" << endl; }
  operator Dst() const {
    cout << "Src::operator Dst() called" << endl;
    return Dst();
  }
};
```

1. 在源类中定义“目标类型转换运算符”

自动类型转换：方法二

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

class Src; // 前置类型声明，因为在Dst中要用到Src类
class Dst {
public:
  Dst() { cout << "Dst::Dst()" << endl; }
  Dst(const Src& s) {
    cout << "Dst::Dst(const Src&)" << endl;
  }
};

class Src {
public:
  Src() { cout << "Src::Src()" << endl; }
};
```

2.在目标类中定义“源类对象作参数的构造函数”

自动类型转换

```
void Transform(Dst d) { }

int main()
{
  Src s;
  Dst d1(s);

  Dst d2 = s;
  Transform(s);
  return 0;
}
```

两种方法任选一种，以上代码均可运行。

注意：两种自动类型转换的方法不能同时使用，使用时请任选其中一种。

自动类型转换：实例1

下面类型转换运算符代码哪些语句有错，原因是？

```
class SmallInt;
operator int(SmallInt&);
class SmallInt{
public:
      int operator int() const;
      operator int(int = 0) const;
      operator int*() const {return 42;}
};
```

自动类型转换：实例1

下面类型转换运算符代码哪些语句有错，原因是？

```
class SmallInt;
operator int(SmallInt&); //错误：不是成员函数
class SmallInt{
public:
        int operator int() const; //错误：不能返回类型
        operator int(int = 0) const; //错误：参数列表不为空
        operator int*() const {return 42;} //错误：42不是
一个指针,返回值是与转换的类型应相同
};
```

自动类型转换：实例2

给定类如下，请写出代码的准确输出：

```
class SmallInt{
public:
    SmallInt (int i=0): val(i){
        cout<<"SmallInt_Init"<<endl;
    }
    operator int() const { //转换运算符
        cout<<"Int_Transform"<<endl;
        return val;
    }
    void print() {
        cout << val << endl;
    }
private:
    size_t val;
};
```

```
int main()
{
        SmallInt si;
        si = 4.10;
        si = si + 3;
        si.print();
        return 0;
}
```

自动类型转换：实例2

```
int main()
{
        SmallInt si;
        si = 4.10;
        si = si + 3;
        si.print();
        return 0;
}
```

最终输出：

```
SmallInt_Init
//SmallInt si，调用构造函数
SmallInt_Init
//si = 4.10，首先内置类型转换将double转换为int，然后调用构造函数隐式地将4转换成SmallInt
Int_Transform
//si + 3，调用类型转换运算符将si隐式地转换成int
SmallInt_Init
//si = si + 3，调用构造函数隐式地将si + 3的结果转换成SmallInt
7
```

禁止自动类型转换

- 如果用explicit修饰类型转换运算符或类型转换构造函数，则相应的类型转换必须显式地进行

```
explicit operator Dst() const;
```

或使用

```
explicit Dst(const Src& s);
```

强制类型转换

- const_cast，去除类型的const或volatile属性。
- static_cast，类似于C风格的强制转换。无条件转换，静态类型转换。
- dynamic_cast，动态类型转换，如派生类和基类之间的多态类型转换。
- reinterpret_cast，仅仅重新解释类型，但没有进行二进制的转换。

强制类型转换

- 之前的示例可修改为

```
int main()
{
  Src s;
  Dst d1(s);

  Dst d2 = static_cast<Dst>(s);
  Transform(static_cast<Dst>(s));
  return 0;
}
```

课后阅读及思考

- 《C++编程思想》
  - 自动类型转换，p306-p312
  - 引用和拷贝构造函数，p254-p271
- 课件中的补充材料
  - 引用的绑定，课件p21-p23
  - 返回值优化，课件p34-p37
  - std::move函数，课件p38-p39
  - 编译器自动合成的函数/运算符，课件p46

结束