目录

[一、 参数格式 2](#_Toc27254)

[二、 返回值优化 2](#_Toc24822)

[2.1优化条件 2](#_Toc15359)

[2.2优化意义 2](#_Toc22007)

[2.3优化实例 2](#_Toc4808)

[三、 f1的功能确定 3](#_Toc22616)

[3.1 第一次拷贝构造 3](#_Toc32525)

[3.2 中间的print函数 3](#_Toc8817)

[3.3 连续的构造 4](#_Toc14503)

[3.4 构造与析构顺序 4](#_Toc24594)

[3.5 两次析构 4](#_Toc10779)

[3.6 对A的移动构造 4](#_Toc20436)

[四、 f2的功能确定 5](#_Toc17695)

[4.1 f2的参数传入 5](#_Toc7183)

[4.2 f2的返回类型 5](#_Toc29752)

[4.3 f3的功能 5](#_Toc4220)

[五、 f3的功能确定 5](#_Toc23140)

[5.1 f3的参数传入 5](#_Toc14783)

[5.2 f3的返回类型 5](#_Toc22761)

[5.3 f3的功能 5](#_Toc9840)

[六、 给出三份答案如下 6](#_Toc21692)

[6.1 答案一 6](#_Toc32103)

[6.2 答案二 7](#_Toc16424)

[6.3 答案三 8](#_Toc31564)

[6.4 答案四 9](#_Toc21144)

oop第三次作业第二题第一部分

By 灯塔&zhaochen20

1. 参数格式

没有太多思考价值，故而建议直接阅读

1.1 首先是两种构造函数的参数格式是固定的，可以直接判断是哪种构造函数；

1.2 拷贝构造函数的参数类型必须是同类对象的常量引用const&。

1.3 移动构造函数的参数类型是同类对象的右值引用&&。

拷贝构造有点像用左值构造；移动构造有点像用右值构造。

1.4 建议详细可参考第三题的备注。

1. 返回值优化

本题重点

2.1优化条件

2.1.1 return的值类型与函数前面的返回值类型相同；

2.1.2 return的是一个局部对象。

2.2优化意义

2.2.1 在禁用返回值优化的条件下，如果一个函数的返回值是某个对象，那在函数返回前需要调用拷贝构造函数构建返回值。

2.2.2 即使我返回的东西就是我传进来的形参对象，返回前的这次构造仍然会发生，也就是这道题中f1的第二次构造函数调用。

2.2.3 但是对于返回值的构造调用的到底是移动构造函数，还是拷贝构造函数，对此C++11有规定：在无返回值优化的情况下，默认调用移动构造函数。

2.3优化实例

2.3.1 如果开启了返回值优化，那么如果我的函数里新定义了一个Test对象。（比如：Test tmp; return tmp;)那编译器会给我优化，先用构造函数构造tmp，然后直接返回tmp。

2.3.2 这里甚至不会调用移动构造，也就是说甚至不是用tmp移动构造了要返回的对象，就是直接返回。

2.3.3 但如果没有开启优化，会首先调用默认构造函数构造tmp，然后调用移动构造函数将用tmp移动构造返回值。

2.4 返回值构造

2.4.1如果返回值是普通的Test的话，那么函数返回时需要根据return后面的表达式来构造一个Test对象，所以才有移动构造或者拷贝构造的事情。

2.4.2如果函数返回类型是const &或者&，那return的时候就没必要再构造新的对象了，直接返回相应的引用。

(5)const Test& F(const Test& a){

Test b = a;

return Test(1);}

int main() {

Test a;

const Test &A = F(std::move(a));

return 0;}

比如此处，由于我的函数返回类型为引用，故而直接返回了返回值的引用。也就是Test(1)的常量左值引用。但是，在函数结束之后，这个常值引用所引用的Test(1)就被析构了，那这个常值引用就没用了。

总结而言，F的返回类型是一个常量左值引用，而返回值是Test(1)这一临时变量。临时变量会在函数体结束后被析构，从而引用失效。故而不能返回临时变量的引用，从而编译错误。

不过，Test(1)感觉只是个临时的右值，这能称为临时变量吗？

实际上是可以的，如果去看编译得到的汇编码，return test(1)和Test ret = test(1); return ret;生成的汇编码是一样的。

1. f1的功能确定

------before call f1------

f1():

Test(const Test&) called. this->buf @ 0x1e1590

a.buf @ 0x1e1590

Test(Test&&) called. this->buf @ 0x1e1590

Test(Test&&) called. this->buf @ 0x1e1590

~Test(): this->buf @ 0

~Test(): this->buf @ 0

------after f1 return------

3.1 第一次拷贝构造

观察f1，显然f1上来就调用了一次拷贝构造函数。回忆课堂上关于拷贝构造函数的调用时机。

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。

可以断定，这里必然是函数调用时以类的对象为形参。需要将形参利用实际调用的对象来拷贝构造。

3.2 中间的print函数

明显可见中间进行了一次print。

void print(const char \*name) {

        cout << name << ".buf @ " << hex << buf << endl;

    }

注意到，print是Test类的一个成员函数。他传入的参数是char类型的指针，也就是字符数组，也即字符串。

如何调用print呢？比如我构造一个Test tmp；那么调用方法是：

  tmp.print("a");

单引号是字符，双引号是字符串，const char \*name是个字符串，但是const char name是是个字符

在C语言中，字符串通过char\*方式表示；C++沿用了这一表示方法，同时提供了更高级的封装类string来进行一些操作。

3.3 连续的构造

f1连续两次调用移动构造函数，难道是中间构造了一个临时变量Test tmp？

Test f1(Test c)

{

    c.print("a");

    Test tmp=c;

    return tmp;

}

如上图，我们先假设构造了临时变量Test tmp，这样看上去f1函数体内的确会有两次构造。第一次用c构造了tmp。而第二次，因为我们禁用了返回值优化，故而返回值会被移动构造一次，这样看上去就有了两次构造，似乎也会有两次析构。

但是，在函数体外Test A = f1(a);这里貌似还有一次构造，那么应该有三次构造函数对应的输出才对。以及，假设我构造了tmp，可是Test tmp=c。为什么会调用移动构造呢？

毕竟，c本身是个左值，会调用拷贝构造函数。如果你想调用移动构造那可能要写成Test tmp=std::move(c)。即便如此，仍然没法解释缺失的A的构造语句。

综上所述，中间这个临时变量是不合理的。但是如何理解这里的两次构造输出和析构输出呢？

3.4 构造与析构顺序

构造A和析构返回值和形参的先后顺序：先执行完f1的函数体，暂不析构。然后执行完整个语句，具体到这个语句，就是外部对A的移动构造。（为什么是移动构造接下来解释）

Test A = f1(a);(等号不一定是赋值，等的时机决定了等号是赋值还是构造)

完成了对A的移动构造之后，就已经完成了两次对应移动构造的输出。分别是因为返回值优化被禁用而对返回值进行的规定的移动构造，以及返回的对象对于A的移动构造。接下来，这一语句执行完毕后才进行函数体的析构，也就是连续的两次析构。第一次析构掉形参，第二次析构掉返回值。（先构造则先析构）

3.5 两次析构

你以为在析构之前，需要想办法把形参和返回值的地址都置为0吗？显然不用，因为形参和返回值都被移动过，而被移动者是会被完全置空的。（这里应参见第三次作业第三题2.1.4）

3.6 对A的移动构造

 Test A = f1(a);//f1(a)是个右值，因为你没法对f1(a)++; 因为f1(a)是个右值，故而默认调用了移动构造函数

这里需要意识到，f1(a)本身是个右值，但是A就是个左值了。

1. f2的功能确定

------before call f2------

f2():

b.buf @ 0x1e1570

------after f2 return------

------before call f3------

4.1 f2的参数传入

f2的输出其实相当之简单，如果我们f2传入的如果是形参。那么必然有拷贝构造，但是这里没有相应拷贝构造的输出，从而我们怀疑是传的引用。

另一方面，f2已经输出了传入的参数。输出的参数的地址已经出现过，也就是传入的是同一块地址的不同名称，那必然是传的引用。

4.2 f2的返回类型

注意到语句的写法，Test& B = f2(b);f2(b)如果返回的是个Test类的对象，那他就是个Test类型的右值。那么你只能够写Test B = f2(b)；也就是把一个右值移动构造给左值，可是你不能让一个左值引用去引用右值。也就是f2(b)返回的应该也是个Test类的引用。也就是传入一个引用，返回一个引用。

引用的标准写法，比如int a=3; int &b=a;这就是说b是a的引用，但是你不能写int &b=3,只有常量左值引用可以引用右值。

4.3 f3的功能

f3实际上传入b的引用，返回了b的引用。（当然直接return b就可以了）

让B成为了b的引用。

1. f3的功能确定

f3():

Test(Test&&) called. this->buf @ 0x1e1550

~Test(): this->buf @ 0

------after f3 return------

5.1 f3的参数传入

和f2相似，如果f3传入的不是引用，那么还会有两次拷贝构造。然而并没有这样的输出，所以f3传入的仍然是引用。

5.2 f3的返回类型

    f3(a, b);

f3并没有成为右值，怀疑其返回值为void。

5.3 f3的功能

仅仅观察f3当然看不出来功能，但是注意到下方的析构输出。先构造则后析构。在上述过程中依次序构造了三个对象，a、b、A，注意到B不是对象而仅仅是b的引用。因而需要析构的对象按次序为A、b、a。

------exiting main------

~Test(): this->buf @ 0x1e1590

~Test(): this->buf @ 0x1e1550

~Test(): this->buf @ 0x1e1570

这样一来，明显感觉到a和b的地址发生了交换。于是f3的功能可想而知，必然是一个swap，并且这个swap是基于移动的swap，不然会有相应基于构造的swap的输出。并且，f3结束时，销毁了中间用于移动的临时变量，符合我们的假设。

实际上，我们在做swap的时候，主要考虑的是swap内存上的数据，不太care内存本身，但是可以从内存本身看出是哪种swap。

 比如我们用拷贝类型的swap，Test tmp=b；b=a；a=tmp；你会发现a和b的地址没变，但是二者内存空间的数据变了。

而移动的实现必须依赖于指针，a和b的指针指向的地址交换，但是这俩地址对应的内存空间的数据没有改变。

oop第三次作业第二题第二部分

By 灯塔&zhaochen20

1. 给出三份答案如下

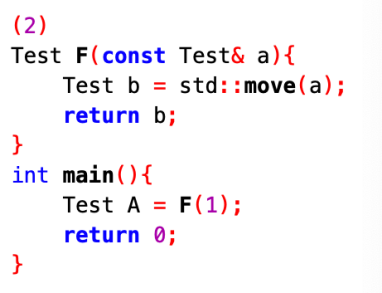
6.1 答案一



6.2 答案二

第二部分的坑：

这里传进去的参数是由参数1构造的右值；传进F之后的a的类型是const&,再move之后就变成了const&&。而拷贝构造的参数类型是&&.所以这里发生了类型的自动转换，调用了拷贝构造函数（而不是移动构造函数）





6.3 答案三

构造函数、复制构造函数、移动构造函数、赋值运算、移动赋值运算被调用的位置及次数

（1）

没有错误。

默认构造函数：

    1、执行Test a;时

重载构造函数（Test （int））：

    1、执行a = 1；时

复制构造函数：

    1、调用F函数时，首先调用拷贝构造函数拷贝a为形参；

移动构造函数：

    1、执行Test b = std::move(a)时调用移动构造函数；

    2、F函数返回时调用移动构造函数

    3、构造Test A时调用移动构造函数；

移动赋值运算符：

    1、执行a = 1给a赋值时

（2）

没有错误。

重载构造函数 （Test (int)）：

    1、进入主函数执行F（1）时调用

复制构造函数：

    1、进入F函数，构造 Test b = std::move(a);时调用

移动构造函数：

    1、F函数return时调用

    2、用F(1)的返回值构造A时调用

（3）

没有错误。

重载构造函数（Test (int)）：

    1、进入主函数执行F（1）时调用

移动构造函数：

    1、进入F函数执行Test b = std::move(a);构造b时调用

    2、F函数return时调用

    3、main函数中用F(1)返回值构造A时调用

(4)

有编译错误。

        错误1：出现在Test A = F(Test(1));

        Test(1)返回的是临时对象，是右值，而Test&& F(Test& a)要求接收左值引用，需要绑定一个左值，因此会报错。

（5）

有错误：

        错误1：出现在函数F返回时；

        函数F返回临时对象b，而返回值类型为常量引用，临时对象无法作为常量引用。

6.4 答案四

(1)

无错误与风险

①构造函数：

1.执行Test a;时，调用构造函数构造a。

2.执行a = 1;时，隐式调用了Test(int val)这一传入参数为int的构造函数 构造了Test(1)（并在之后移动赋值给a）

②复制构造函数：

1.调用F函数时，首先将调用拷贝构造函数将a拷贝给形参。

③移动构造函数：

1.在F函数体内，执行执行Test b = std::move(a)时调用移动构造函数构造了b。

2.在F函数体内，返回时调用移动构造函数移动构造返回值。

3.执行 Test A = F(a);语句时，调用移动构造函数移动构造A。

④赋值运算：

无。

⑤移动赋值运算：

1.执行a=1;时，通过构造函数构造Test(1)后将其移动赋值给a。

(2)

无错误与风险

①构造函数：

1.执行F(1)时，隐式调用了Test(int val)这一传入参数为int的构造函数构造了Test(1)。

②复制构造函数：

1.F函数体内，执行 Test b = std::move(a)前，a的类型是常量左值引用。但是接下来move使得a发生了类型转换，转换为const&&。这既是一个const同时也是个右值，故而只能够被const&绑定，从而调用了拷贝构造函数。

③移动构造函数：

1.在F函数体内，返回时调用移动构造函数移动构造返回值。

2.执行Test A = F(1);语句时，调用移动构造函数移动构造A。

④赋值运算：

无。

⑤移动赋值运算：

无。

(3)

无错误与风险

①构造函数：

1.执行F(1)时，隐式调用了Test(int val)这一传入参数为int的构造函数 构造了Test(1)。

②复制构造函数：

无。

③移动构造函数：

1.在F函数体内，执行执行Test b = std::move(a)时调用移动构造函数构

造了b。

2.在F函数体内，用b移动构造了返回值。

3.执行Test A = F(1);语句时，调用移动构造函数移动构造A。

④赋值运算：

无。

⑤移动赋值运算：

无。

（4）

存在编译错误

错误出现在Test A = F(Test(1));传入参数这一步。

注意到Test(1)返回的是一个右值。而Test&& F(Test& a)要求传入参数为左值引用，需要绑定左值，因此出现了编译错误。

（5）

存在潜在风险，但可通过编译

风险出现在F的函数体内。

注意到F的返回类型是一个常量左值引用，而返回值是Test(1)这一临时变量。临时变量会在函数体结束后被析构，从而导致引用失效。故而不能返回临时变量的引用，这一过程存在风险(warning)。但是程序没有error，故而可以通过编译。