引用与复制

引用与复制

By 曹菡雯（计 03）、赵晨阳（软 01）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分主要是 L6-引用与复制的课堂笔记整理，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于每一页 PPT 都有完整的解读，同时联系了前后课程的内容与课程作业，对于课件中的操作也有一定的扩展。

本次笔记的 5、6、7、8 部分为第三次作业第二、三题的解析，建议结合阅读。

本次笔记用到了许多禁用返回值优化的编译条件，为了使输出结果统一，小组成员统一采用了在线编译器：

https://rextester.com/l/cpp_online_compiler_gcc

该编译器能够在线实现 g++编译指令，禁用返回值优化的编译指令如下：

g++-Wall -std=c++14 -O2 -fno-elide-constructors -o a.out source_file.cpp

最近一次更新：将原文档中所有代码重新进行了编写，有助于阅读。

如果阅读时间不够充足，建议阅读课堂的扩展部分。

2.4.2 默认生成移动构造函数

3.2.3 例二：完整定义所有函数但禁用返回值优化后的执行顺序

5. 结合第三次作业第二题讨论返回值优化

6. 结合第三次作业第三题讨论 delete 与检测

7. 结合第三次作业第三题讨论 move 与类型转换

8. 结合第三次作业第三题讨论置空操作

9.3 类型转换的实例详述

一、常量引用

1.1 常量引用的意义

按照最小特权原则：给函数足够的权限去完成相应的任务，但不要给予他任何多余的权限。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。

二、拷贝构造函数

2.1 定义与语法规则

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。

作用：用参数对象的内容初始化当前对象。

```
Vector::Vector(const Vector& other)
{
   capacity = other.capacity;
   len = other.len;
   array = new Node[other.capacity];
   for (int i = 0; i < len; i++)
   {
      array[i] =other.array[i];
   }
}
```

2.2 调用时机

拷贝构造函数被调用的三种常见情况。在这些情况下，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。

用一个类对象定义另一个新的类对象

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

函数调用时以类的对象为形参

```
Func(Test a)
```

函数返回类对象（无返回值优化的情况下）

```
Test Func(void)
```

2.3 隐式拷贝构造函数

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“隐式定义的拷贝构造函数”，其功能是调用所有数据成员的拷贝构造函数或拷贝赋值运算符。

对于基础类型来说，默认的拷贝方式为位拷贝(Bitwise Copy)，即直接对整块内存进行复制。

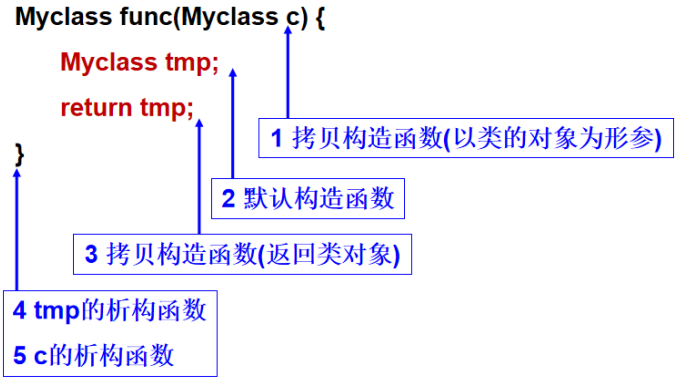
位拷贝原本是 C 中的概念。在 C++中，只有基础类型（int, double 等）才会进行位拷贝；对于自定义类，编译器会递归调用所有数据成员的拷贝构造函数或拷贝赋值运算符。但一些教材中仍然把这种行为称为“位拷贝”，以区别用户自定义的拷贝方法。

隐式定义拷贝构造函数的缺陷：隐式定义拷贝构造函数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。

2.4 执行顺序

2.4.1 基本的执行顺序

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。



值得一提的是，实际上在主函数体内，假设我们禁用返回值优化。Myclass a= func(d); 这个语句内的返回值类构造调用了拷贝构造，并且相关的析构非常值得研究，下文将在移动构造函数的例子中将进一步阐述。

2.4.2 例子

（编译指令 g++ test.cpp --std=c++11 -fno-elide-constructors –o test）

```
#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);
```

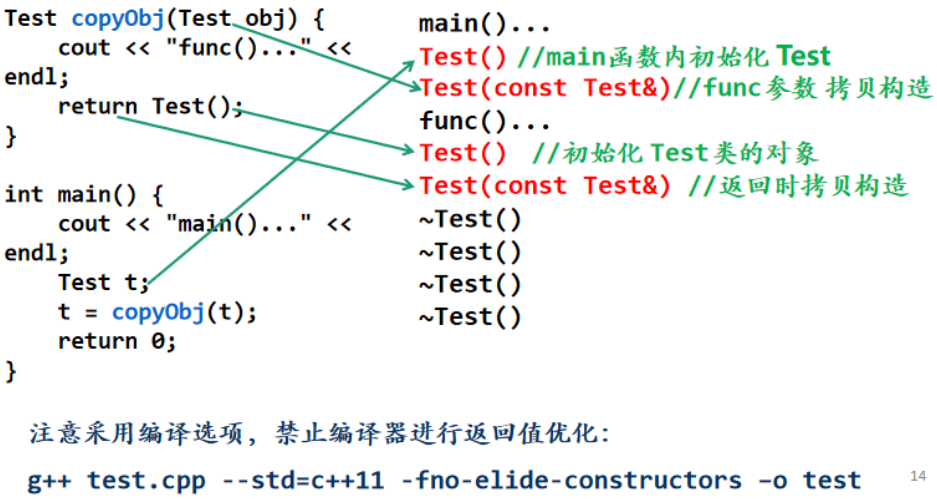
```
    return 0;
}
```

输出：

```
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()
```

对应关系并不复杂，如下图所示：



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
```

14

其实最有意思的是，这里对于返回类对象的构造。

我们在第三次作业第二题当中学习过：

“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。

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。

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。”

按此理解，此处构造返回值理应调用移动构造，而非拷贝构造，似乎产生了矛盾。然而，当我们显式定义了拷贝构造，编译器就不会默认生成移动构造了。

从而禁用了返回值优化后，在返回值的构造当中调用了拷贝构造而非移动构造，因为编译器不会默认生成移动构造，我也没有自己定义。

更进一步，如果我显式定义了拷贝赋值，就不会默认生成移动赋值了。如果两个都不定义，两个都会默认生成。

2.5 拷贝构造函数的缺陷

当类内含指针类型的成员时，拷贝构造函数会使得两个指正指向同一内存空间。在析构时，该内存空间可能被反复释放。为避免指针被重复删除，不应使用隐式定义的拷贝构造函数。

以及，当**对象较大**时，频繁的拷贝构造会造成程序**效率的显著下降**。

故而，正常情况下，应尽可能避免使用拷贝构造函数。

2.6 解决方法

使用引用/常量引用传参数或返回对象；

```
引用或常量引用传递参数

func(MyClass a)改为 func(const MyClass& a)

返回值为引用

MyClass func(…)改为 MyClass& func(…)
```

将拷贝构造函数声明为 private；

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

用 delete 关键字让编译器不生成拷贝构造函数的隐式定义版本。

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

三、移动构造函数

3.1 右值与右值引用

3.1.1 左右值

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。

虽然右值不能取地址也没有名字，但是右值可以运算。

```
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。

```
#include <iostream>
using namespace std;
int x();
 int main(){
    int &&a=x()+1;
    cout<<a<<endl;
    a++;
    cout<<a<<endl;
    return 0;
}
int x(){return 1;}
Output：
2
3
```

右值可以运算，且右值引用接收右值后也可以运算。

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;
```

上方操作允许，下方操作非法。

3.1.2 右值引用

虽然右值无法取地址，但可以被&&引用(右值引用)

右值引用可以绑定右值

```
int &&e = a + b;
```

右值引用无法绑定左值

```
int &&e = a;
```

该操作非法。

总之，左值引用能绑定左值，右值引用能绑定右值。特例: **常量左值引用能也绑定右值**,因为常量左值引用不会改变内存空间的数据,故而不会影响右值。也可以实现绑定。

```
const int &e = 3;
```

```
const int &e = a*b;
```

注意到，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。

```
#include <iostream>
using namespace std;
int x();
 int main(){
    int &&a=x()+1;
    cout<<a<<endl;
    a++;
    cout<<a<<endl;
    int &b=a;
    b++;
    cout<<a<<endl;
    return 0;
}
int x(){return 1;}
Output：
2
3
4
```

如此例，右值引用本身是左值，可以被取地址。

例子一

```
#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;
}
输出：

left 1
right 2
```

如果没有定义 ref(int &&x) 函数会发生什么？

ref (int&x)的函数参数类型是一个左值引用，而 2 是一个右值，不能调用 ref (int&x)。因此没有可供调用的函数。

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

例子二

```
#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;
}
```

输出：

```
right 1
left 1
```

如前文所述，所有引用本身都是左值，故而 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。

3.2 移动构造函数

3.2.1 定义与意义

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。

```
ClassName(ClassName&& VariableName);
```

3.2.2 两种构造本质区别

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。

3.2.3 例子

例一：完整定义所有函数以及启用返回值优化

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

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

```
}
output：
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
```

没有调用移动构造函数，也少调用了几次拷贝构造函数。

关键点在于，返回值优化之后不会通过移动构造来构造返回值类，也不会通过移动构造来构造 a，实现了直接 return。

例二：完整定义所有函数但禁用返回值优化

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

这里课件上的注释有些误导，我在 rextester 输出该段代码，采用的指令为 g++-Wall -std=c++14 -O2 -fno-elide-constructors -o a.out source_file.cpp

```
output：
Test(): this->buf @ 0x563a2616ae70
GetTemp(): tmp.buf @ 0x563a2616ae70
Test(Test&&) called. this->buf @ 0x563a2616ae70
~Test(): this->buf @ 0
Test(Test&&) called. this->buf @ 0x563a2616ae70
~Test(): this->buf @ 0
main() : a.buf @ 0x563a2616ae70
Test(const Test&) called. this->buf @ 0x563a2616beb0
fun(Test t): t.buf @ 0x563a2616beb0
~Test(): this->buf @ 0x563a2616beb0
~Test(): this->buf @ 0x563a2616ae70
```

禁用返回值优化但是同时定义了移动构造函数，故而需要通过移动构造来构造返回值类。之后立刻将 tmp 析构(由于移动构造已经将 tmp 的指针置空了，故而 buf 地址为 0)，但是没有析构返回值类 GetTemp。返回值类 GetTemp 对 a 移动构造，这使得 GetTemp 的 buf 地址也为 0。这一步移动构造后，马上析构了 GetTemp。

综上所述，Test a = GetTemp();这一语句的执行顺序是先构造 tmp，移动构造返回值类。立刻析构 tmp，返回值类移动构造 a，接着析构返回值类。

结合例子 2.4.1，我们来探究下 4,5 和返回值类在禁用返回值优化条件下的析构顺序。测试代码如下，编译指令为 g++-Wall -std=c++14 -O2 -fno-elide-constructors -o a.out source_file.cpp

```
#include<iostream>
using namespace std;
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
    }
};
Test GetTemp(Test x) {
    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 d;
    Test a = GetTemp(d);
    cout << "main() : a.buf @ " << hex << a.buf << endl;
    fun(a);
    return 0;
}
Output:
```

```
Test(): this->buf @ 0x5572ec8c2e70
Test(const Test&) called. this->buf @ 0x5572ec8c3eb0
Test(): this->buf @ 0x5572ec8c3ee0
GetTemp(): tmp.buf @ 0x5572ec8c3ee0
Test(Test&&) called. this->buf @ 0x5572ec8c3ee0
~Test(): this->buf @ 0
Test(Test&&) called. this->buf @ 0x5572ec8c3ee0
~Test(): this->buf @ 0
~Test(): this->buf @ 0x5572ec8c3eb0
main() : a.buf @ 0x5572ec8c3ee0
Test(const Test&) called. this->buf @ 0x5572ec8c3eb0
fun(Test t): t.buf @ 0x5572ec8c3eb0
~Test(): this->buf @ 0x5572ec8c3eb0
~Test(): this->buf @ 0x5572ec8c3ee0
~Test(): this->buf @ 0x5572ec8c2e70
```

可以发现，先拷贝构造了 x，接着构造 tmp，移动构造 GetTemp，马上析构 tmp，移动构造 a，析构 GetTemp，析构 x。故 4 和 5 之间可能还会进行非常多的操作。

例三：删除移动构造函数且禁用返回值优化

编译指令 g++-Wall -std=c++14 -O2 -fno-elide-constructors -o a.out source_file.cpp

```
#include<iostream>
using namespace std;
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 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;
}
output :
Test(): this->buf @ 0x5638c49e8e70
GetTemp(): tmp.buf @ 0x5638c49e8e70
Test(const Test&) called. this->buf @ 0x5638c49e9eb0
~Test(): this->buf @ 0x5638c49e8e70
Test(const Test&) called. this->buf @ 0x5638c49e8e70
~Test(): this->buf @ 0x5638c49e9eb0
main() : a.buf @ 0x5638c49e8e70
Test(const Test&) called. this->buf @ 0x5638c49e9eb0
fun(Test t): t.buf @ 0x5638c49e9eb0
~Test(): this->buf @ 0x5638c49e9eb0
~Test(): this->buf @ 0x5638c49e8e70
```

这里只有拷贝构造，编译器没有默认生成移动构造。拷贝构造了返回值类后，立刻析构了 tmp，释放了尾号为 70 的空间。注意到，空间释放之后马上就可以再次被使用，所以返回值类去拷贝构造 a 时，用的便是上次被析构释放出来的尾号为 70 的空间。接下来析构掉返回值类，释放了尾号为 b0 的空间，下一次 fun(a)当中对形参的拷贝构造马上就利用了 b0 空间。

3.3 移动语句

3.3.1 语法与意义

移动构造函数加快了右值初始化的构造速度，故而希望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。移动后就不能再次使用）

返回值：该左值对应的右值

```
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
```

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。移动的具体操作在移动构造函数内实现。

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。

例如：

```
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
}
```

改写为：

```
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。

3.3.2 例子
禁用返回值优化

```
#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 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。

output:

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
```

这当中的析构顺序和 3.2.3 例二完全相同。

3.4 两类构造函数的调用时机

3.4.1 判断依据

引用的绑定规则

拷贝构造函数的形参类型为常量左值引用，可以绑定常量左值、左值和右值。移动构造函数的形参类型为右值引用，可以绑定右值。引用的绑定存在优先

级，例如常量左值引用和右值引用均能绑定右值，当传入实参类型为右值时优先匹配形参类型为右值引用的函数。

3.4.2 拷贝构造函数的常见调用时机

用一个类对象/引用/常量引用初始化另一个新的类对象

以类的对象为函数形参，传入实参为类的对象/引用/常量引用

函数返回类对象（类中未显式定义移动构造函数，不进行返回值优化）

3.4.3 移动构造函数的常见调用时机

用一个类对象的右值初始化另一个新的类对象（常配合 std::move 函数一起使用）: Test b = func(a); Test b = std::move(a); 与 Test b = a; 不同

以类的对象为函数形参，传入实参为类对象的右值（常配合 std::move 函数一起使用）: func(Test());func(std::move(a));与 func(a)不同

函数返回类对象（类中显式定义移动构造函数，不进行返回值优化）: {return Test(); or return tmp;}均调用移动构造

例子

Test类中显式声明了三类构造函数：

普通构造函数： Test(int val);
移动构造函数： Test(Test&& t);
拷贝构造函数： Test(const Test& t);

现给出一段测试代码，下列描述错误的是

```
Test F(Test&& a){
    Test b = a;    // (2)
    const Test& c = b; // (3)
    return c; // (4)
}
int main(){
    Test A = F(1); // (1)
    return 0;
}
```

A (1)处将1传入F时会调用普通构造函数Test(int val)以构建临时对象。

B (2)处调用移动构造函数。

C (3)处调用拷贝构造函数。

D (4)处返回局部变量的引用，可能会为程序带来潜在的风险。

注意到 D 是错的，返回值虽然是非静态临时变量的常量左值引用，但是返回类型根本不是引用，并不是返回临时变量的引用。

四、拷贝与移动赋值运算符

4.1 拷贝赋值运算符定义与意义

已定义的对象之间相互赋值，可通过调用对象的“拷贝赋值运算符函数”来实现的。

```
ClassName& operator= (const ClassName& right) {
  if (this != &right) {// 避免自己赋值给自己

     // 将 right 对象中的内容拷贝到当前对象中...

  }
```

```
  return *this;
}
```

注意做出区分：

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

此处为拷贝复制运算。

```
ClassName a = b;
```

而此处为定义新对象。

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。因为复制运算是对于这个类而言的，显然是类函数。同时，赋值运算依赖于具体的对象，不能是静态的。（详见 L4·创建与销毁 2 笔记）

```
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;
}
```

4.2 移动赋值运算符作用与意义

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;
}
```

例如

```
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
```

```
}
```

4.3 调用时机

和拷贝/移动构造函数的调用时机类似，主要判断依据是引用的绑定规则

拷贝赋值运算符函数的形参类型为常量左值引用，可以绑定常量左值、左值和右值

移动赋值运算符函数的形参类型为右值引用，可以绑定右值(常量、表达式、函数返回)

引用的绑定存在优先级，例如常量左值引用和右值引用均能绑定右值，当赋值运算符右侧为右值时优先匹配形参类型为右值引用的赋值运算符函数

根据赋值运算符右侧变量的类型决定调用拷贝或移动赋值运算符函数

4.4 自动合成的函数与运算符

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

五、返回值优化

该部分主要是第三次作业第二题的解析，建议结合阅读。

5.1 优化条件

5.1.1 return 的值类型与函数前面的返回值类型相同。
5.1.2 return 的是一个局部对象的左值。

5.2 优化意义

5.2.1 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构造函数构建返回值。

5.2.2 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。

5.2.3 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。

5.2.4 综上所述，返回一个局部对象的左值，通过移动构造构造返回值，返回值再移动赋值或移动构造给主函数语句。

5.3 优化实例

5.3.1 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。

5.3.2 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。

5.3.3 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。

5.4 返回值构造

5.4.1 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。

5.4.2 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。

5.5 例子

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

class Test{
public:
   int data = 0;
```

```
    Test(){}
    Test(const Test& t){}
    Test(Test&& t){}
};
Test fn1(){
    Test tmp; return tmp; //(1)
}
Test&& fn2(){
    Test tmp; return move(tmp); //(2)
}
Test fn3(){
    Test tmp; return move(tmp); //(3)
}
int main(){
    const Test& a = fn1(); //(4)
    Test&& b = fn1(); //(5)
    Test c = fn1(); //(6)
    Test&& d = fn2(); //(7)
    return 0;
}
```

建议做法包括(1)(4)(5)(6),避免多余拷贝，优化资源利用

Test fn1(); 满足返回值优化条件

可利用常量左值引用(2)，右值引用(5)，构造新对象(7)的方式接收返回值

不建议做法包括(2)(3)(7)

(2)(7) d 会指向被析构的 tmp，出现运行错误

std::move()将左值转变为右值，不进行返回值优化，(3)会移动构造临时变量

关于(2)(7)，首先记住，如果返回引用，只有返回常量局部变量的左值引用是合法的，其他的都不合法。即使在某些编译器上通过了，也很危险。在较为严格的编译器上不一定能通过。

```
int &func(){static int x=0;return x;}
```

用此处(2)(7)举例，(2)(7)的意思是返回了一个右值的右值引用，把这个右值引用的返回值赋值给了一个右值引用。

我们写出如下一个类似的例子：

```
#include <iostream>
using namespace std;
int&&func(){
    int tmp=1;
    return std::move(tmp);
}
int main(){
    int&&x=func()+1;
```

```
    cout<<x;
    return 0;
}
```

采用 glot 平台运行，output：2

采用 retexter 平台运行，编译指令采用 clang++-Wall -std=c++14 -stdlib=libc++ -O2 -o a.out source_file.cpp，输出结果为完全不确定的整数。虽然直觉上可以完成赋值，但是由于 tmp 不是静态局部变量，逐语句结束后会被析构，导致 x 在访问非法内存。

采用 retexter 平台运行，编译指令采用 g++--Wall -std=c++14 -O2 -o a.out source_file.cpp，输出结果为 1，warning 如下：

```
1944460822/source.cpp: In function ‘int&& func()’:
1944460822/source.cpp:5:25: warning: function returns address of local variable [-Wreturn-local-addr]
    return std::move(tmp);
                     ^
1944460822/source.cpp:4:9: note: declared here
    int tmp=1;
        ^~~
1944460822/source.cpp: In function ‘int main()’:
1944460822/source.cpp:8:19: warning: ‘tmp’ is used uninitialized in this function [-Wuninitialized]
    int&&x=func()+1;
                  ^
```

总而言之，返回非静态临时变量的引用风险极大。

备注，这个例子本身并不太好。如果实在想检测这样的函数，不应该用 int 类型。因为 int 是编译器自带的基本类型，采用过多的优化，故而 glot 输出 2 也是合理的。这样的检测最好使用自定义类型并完整地定义所有的构造，析构，拷贝构造，移动构造，移动赋值与拷贝赋值，确保编译器优化不多。

六、delete 与检测

该部分为第三次作业第三题的解析，建议结合阅读

6.1 Delete 的意义

```
delete[]array;//delete：释放指针指向的内存区
```

故而 delete 之后，内存区的数据就清除了。

delete 释放的是指针指向的内存空间，指针变量本身仍然存在可以使用（用于赋值 etc）。但是调用 array[i]会产生运行时错误 segmentation fault，因为它没有指向一块内存空间用于存储数据。

清除不等同于设为 0。而且实际上，在这道题里面，如果 delete 之后还把 array[i]赋值为 0，不仅没有意义（因为内存已经被清除了），而且会因为调用了多次 Node 的拷贝构造函数而浪费了效率。比如下面这个写法纯属浪费效率，还

有错误。在 delete[]array;语句后，如果想进行赋值为 0 的操作，array 要重新申请指向一块内存。

```
Vector&Vector::operator=(const Vector& other)
{    if (this != &other) {
      if (this->array != nullptr)
      {delete[]array;}
      for (int i = 0; i < capacity; i++)
      {array[i] = 0;}.....}
```

6.2 Delete 的检测

建议在 delete 前检测指针是否为空指针，因为空指针 delete 会报错。

```
vector::~Vector()
{if (array != nullptr) { delete[]array; }}
```

6.3 赋值的检测

和 delete 相似，建议赋值也要先进行检测，避免浪费效率

```
Vector&Vector::operator=(const Vector& other)
{      if (this != &other)
      //建议赋值的时候检测是否相同，检测的原理本质上就是检测地址是否相同
      if (this->array != nullptr)
      { delete[]array; }.....
}
```

七、 move 与类型转换

该部分为第三次作业第二题的解析，建议结合阅读

7.1 move 的意义

move 的作用就是仅仅把左右值这个属性改为右值，其余的属性不变。

7.2 例子

```
Test F(const Test& a){
   Test b = std::move(a);
return b;}
int main(){
   Test A = F(1);
return 0;}
```

这个地方极其有趣的是 F 函数体内对 b 的构造是采用了拷贝构造而非移动

构造。

这是由于 move 和 const 的双重作用。如同前文所述，move 一定可以把任意对象调整为右值，不管是不是 const。

深层次地说，变量储存在内存里的时候存的都是它的数据，而不会专门开辟一块空间来说明它是不是 const，是不是&等，const 和&这些“类型”是由编译器来处理的。

我们说类型转换，指的就是改变类型，而不变动它在内存里的储存形式。比如一个指针，它在内存里存的就是一串数字表示地址，我们看它的内存情况时，完全可以把这块内存当做存的是一个整型变量。我们强行把指针当做整型变量来看的时候就是进行了一个类型转换。

所以 std::move(a)这个表达式的类型就是 const&&，因为它发生了类型转换。而转换之后，a 转为 const&&，这是一个常量右值引用，C++固定其只能够被 const&绑定。故而在构造函数重载的情况下调用了相应 const&的拷贝构造函数。

7.3 进一步讨论

Parameter → / Argument ↓	&	const&	&&	const&&
lvalue	✔ 1st	✔ 2nd		
const lvalue		✔ 1st		
rvalue		✔ 3rd	✔ 1st	✔ 2nd
const rvalue		✔ 2nd		✔ 1st

这张表给出了关于绑定和左右值的关系。左边的一列是参数的属性，Lvalue 是指左值，rvalue 是指右值，const 是指常量。上方一行给出了能够绑定的类型。而表中的次序代表多种绑定时的优先级。从而可见，常量右值优先在我们的两个构造函数当中只能绑定 const&。（因为没有 const&&类型的重载，故而绑定的是 2^{nd}）从而调用了拷贝构造函数。

八、置空性讨论

8.0 析构置空

析构函数需要将成员数据中的指针指向的内存空间置空（delete）

```
Vector::~Vector()
{if (array != nullptr) { delete[]array; }}
```

下文以 A=B 为例，讨论赋值过程中的置空性

8.1 移动置空

8.1.0 每次移动时，我们将 B 完全置空但是不 delete

8.1.1 移动和拷贝的语义区别

移动和拷贝的语义区别在于，被拷贝的对象之后还可能会被使用的，而被移动的对象必须置空（但不是 delete，delete 出现在赋值置空和析构置空当中）。因为移动的本质原理是把对同一块内存空间更换为新指针，并且将原指针置空。

8.1.2 移动和拷贝的原理区别

8.1.2.1 拷贝的原理

A 与 B 各自有一个指针，如果 A！=B 时，两个指针指向两块独立的内存空间。（A=B 时无需拷贝）现将 A 内存空间里所有数据清除（为什么要清除在 2.2.1 进行了解释），接着把 B 的内存空间内所有数据传递给 A 的内存空间。这一传递过程不会破坏 B 的数据，但是效率较低。

8.1.2.2 移动的原理

A 与 B 各自有一个指针。如果 A！=B 时，两个指针指向两块独立的内存空间。（A=B 时无需移动）如果此时 A 的指针指向的内存空间不为空，也就是 A 指针不为空指针，那么将 A 的指针指向的内存空间通过 delete 清除，A 指针即转为空指针。将 B 的指针赋值给 A 的指针，并将 B 指针置为空指针。这一过程实际上是改变了内存空间的指针但是没有改变内存空间。

8.1.3 移动过程中 B 指针置空

在析构函数里，我们会调用 delete。

```
Vector::~Vector()
{if (array != nullptr) { delete[]array; }}
```

倘若不把 B 指针置空，那么我们会析构 delete 一次 B 指针指向的内存空间，然而 A 指针不为空。在析构时，又将对同一个内存空间 delete 一次。Delete 一空的空间是不允许的。故而移动构造必然要将 B 指针置空以避免多次 delete 同一空间。这样也导致了 B 指针不能再次使用（整个 B 无法再去赋值），这就是为什么移动构造必然要破坏用来构造的对象：因为不能出现两个指针指向同一块空间的情况。

同样的，为了模拟这种对 B 指针的破坏性，在第三部分技术细节里提及了 Node 的移动构造机制。这一机制也将“B 指针”（实际上是原来的 int）置空，故而被移动了的 Node 也是没法再次利用的。

8.1.4 移动过程中 B 完全置空

我们在上文已经叙述了为什么需要将 B 指针置空。实际上，由于 B 对象被移动之后已经不再次使用，需要将其完全置空（所有的成员数据都要置空，而不只是 B 里面的指针对象），从而避免内存泄漏。（这里点到为止，具体的细节参见第三部分）

8.2 赋值置空

8.2.0 Def 每次赋值时，我们需要通过 delete 将 A 置空

8.2.1 拷贝赋值

拷贝赋值时，检测完 A==B 后，我们需要使用 delete 将 A 置空。

这里为什么要先清除 A 的原内存空间？

首先，我们的移动赋值需要二者赋值完后完全相同，如果 A 原内存空间超出 B 的那部分容积有多于内容，不清除将 A 则无法使得 A 与 B 完全相同。其次，不清除更大的问题是内存泄露严重。

注意到拷贝赋值现将 A 的内存空间 delete 之后，A 的指针需要指向一块内存空间才能使 A 接受 B 的拷贝赋值。故需要申请新的内存空间。（与 B 的内存空间大小完全相同）

8.2.2 移动赋值

综合前文所述，移动需要将 B 完全置空但是不 delete，而赋值需要将 A 通过 delete 置空之后再次申请新的空间。所以用 B 移动赋值 A 的时候需要两次置空，一次 delete（对 A 的）而另一次不 delete（对 B 的）。

九、类型转换

9.1 意义

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。

```
void print(int d) { }

int main()
{
    print(3.5);
    print('c');
    return 0;
}
```

9.2 语法

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();
  }
};
```

在源类中定义“源类中重载目标类型转换运算符”

注意：不需要指定返回类型，因为 operator 后 Dst()已经指明，返回值是 Dst（），返回函数名故而不需要返回类型。这是类型转换的固定语法，如同构造函数不需要就好。

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; }
};
```

在目标类中定义“目标类中定义源类对象作参数的构造函数”。

两种方法任选一种即可运行。

但是一定要注意区分两种转换方式，如果没有做区分，常常会导致转换方向错误。此外，两种自动类型转换的方法不能同时使用，使用时必须任选其中一种。

9.3 例子

例一

```
#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();
  }
};
void Transform(Dst d) { }
int main()
{
  Src s;
  Dst d1(s);
  Dst d2 = s;
  Transform(s);
  return 0;
}
output：
Src::Src()
Src::operator Dst() called
Dst::Dst()
Src::operator Dst() called
Dst::Dst()
Src::operator Dst() called
Dst::Dst()
```

注意到强制类型转换并不会发生类似切片的效果，只是用某一个类型生成了另一类型而已。这里主函数内 Dst d2 = s;等价于 Dst d2(s)；而 Void Transform(Dst d)；调用的参数本该是 Dst 类型，由于定义了从 s 到 Dst 的类型转换，故而也可以执行，并进行了转换。

例二

寻找错误

```
class SmallInt;
operator int(SmallInt&); //错误：不是成员函数；无论哪种转换方式，都是定义在
类内的成员函数
class SmallInt{
public:
   int operator int() const;
```

```
//错误：不能返回类型

operator int(int = 0) const; //错误：在源类中定义目标类的重载运算符需要参数列

表为空

operator int*() const {return 42;}

//错误：42 不是一个合法指针,本意：将 SmallInt 对象转换为 int* 指针

};
```

例三

```
#include<iostream>
using namespace std;
class SmallInt{
public:
    SmallInt (int i=0): val(i){
        cout<<"SmallInt_Init"<<endl;
}
//构造函数:以 int 为参数的 SmallInt 构造函数，从而将 int 转换为 SmallInt

    operator int() const {
        cout<<"Int_Transform"<<endl;
        return val;
}
//转换运算符:从 SmallInt 转换为 int；在源类中定义“目标类型转换运算符”

//注意到这个例子既有 int 到 smallint 的转换，也有 smallint 到 int 的转换。

//smallint 既做了源类，又做了目标类

    void print() {
        cout << val << endl;
    }
private:
    size_t val;
};
int main()
{
    SmallInt si;
    si.print();
    si = 4.10;
    si.print();
    si = si + 3;
```

```
    si.print();
    return 0;
}
output:
SmallInt_Init
0
SmallInt_Init
4
Int_Transform
SmallInt_Init
7
```

首先，这里定义的 val 是 size_t 类型。它是一种“整型”类型，里面保存的是一个整数，就像 int, long 一样。这种整数用来记录一个大小(size)。

size_t 的全称应该是 size type，就是说“一种用来记录大小的数据类型”。通常我们用 sizeof(XXX)操作，这个操作所得到的结果就是 size_t 类型。因为 size_t 类型的数据其实是保存了一个整数，所以它也可以做加减乘除，也可以转化为 int 并赋值给 int 类型的变量。

注意第二个 SmallInt_Init;出现在 si = 4.10，首先内置类型转换将 double 转换为 int，然后调用构造函数构造了以 int 4 为参数的 SmallInt（4），隐式地将 4 转换成 SmallInt 类型。之后把这个 Smallint（4）赋值给 Si，系统自动生成了 Smallint 的赋值运算符重载。

si = si + 3；首先执行 si + 3，之后执行赋值语句。

这可能有两种情况，情况一，把 si 转为 int，然后执行 int 的加法，得到 7；得到 7 之后的步骤即为赋值语句 si=7，和上方的 si=4 类似。调用构造函数构造了以 int 7 为参数的 SmallInt（7），隐式地将 7 转换成 SmallInt 类型，之后再调用编译器自动生成的赋值运算符重载。

情况二，把 3 转为 smallint 再和 si 相加。这就有个问题，我们没有显式地重载 smallint 的+运算符，故而无法这个转换即是完成了，也无法继续。

另一方面，如果我们在此基础上重载了 smallint 的+运算符，我们不仅有 smallint 转为 int 然后两个 int 相加再构造 smallint(7)再赋值这条路，同时还有 3 利用构造函数隐式转为 smallint 之后两个 smallint 相加，再赋值给 si 这条路。会出现路径歧义而编译错误。如何只进行第二条路，我们通过例四来实现。

例四

```
#include <iostream>
using namespace std;
class SmallInt{
public:
    SmallInt(int i=0): val(i){
```

```
        cout<<"SmallInt_Init" << endl;
    }
    SmallInt& operator=(const SmallInt &src){
        if( this == &src ) return *this;
        cout <<"operator="<<endl;
        this->val = src.val;
        return *this;
    }
SmallInt operator+(const SmallInt& b) {
        cout<<"operator+"<< endl;
        return SmallInt(this->val + b.val);
    }
private:
    size_t val;
};
int main(){
    SmallInt si;
    si = 4.10;
    si = si + 3;
    return 0;
}
output :
SmallInt_Init
SmallInt_Init
operator=
SmallInt_Init
operator+
SmallInt_Init
operator=
```

这一例子验证了之前的理解。我们有了 si=smallint（4）对应的=拷贝赋值运算符重载的输出。在 si=si+3 这一步，首先把 3 转为了 smallint（3），接着利用了 operator+。注意到 operator+的返回值再次调用了构造函数，接着把返回值拷贝赋值给 si。

但是例四是如何避免了路径歧义呢？注意到我们删除了从 smallint 转为 int 的类型转换运算符，故而没有路径一，只有路径二。

9.4 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。例如：

```
explicit operator Dst() const;
或使用
explicit Dst(const Src& s);
```

在例一中如果使用 explicit，那么：

```
int main()
{
  Src s;
  Dst d1(s);       //可以执行，被认为是显式初始化

  //Dst d2 = s;    //错误，隐式转换

  //Transform(s); //错误，隐式转换

  return 0;
}
```

9.5 四类强制类型转换

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。
之前的例子重写为：

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

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