三/五法则

三个基本操作控制类的拷贝操作：

- 拷贝构造函数
- 拷贝赋值函数
- 析构函数
  当然，C++11引入另外两个操作，移动构造和移动赋值操作。

由于拷贝控制操作由三个基本操作函数完成，因此成为“C++三法则”，后来又支持移动语义，故变成了“五法则”。“三/五”法则总结为：

- 需要析构函数的类也需要拷贝构造函数和拷贝赋值函数
- 需要拷贝操作的类也需要赋值操作，反之依然
- 析构函数不可是deleted的，即不能删除
- 如果一个类的析构函数是不可访问的（nonpublic）或不可删除的，那么其默认和拷贝构造函数会被定义为删除的
- 如果一个类又const或引用成员，则不能使用合成的拷贝赋值操作

需要析构函数的类也需要拷贝和赋值操作

首先，构造函数时动态分配了内存，显然需要自己定义析构函数，来释放构造函数动态分配的内存。

为什么说需要析构函数的类通常需要拷贝和赋值操作（自定义的，而不是合成的），因为合成的拷贝和赋值操作会引起错误。如

```
class Hasptr{
private:
    string *ptr;
public:
    Hasptr(string &s=string()):ptr(new string(s)){}
    ~Hasptr(){delete ptr;ptr=nullptr;}
};
// 合成的拷贝构造函数将会类似于下面
Hasptr::Hasptr(const Hasptr &rhs){
    this->ptr=rhs->ptr;// 两者指向了同一内存
}
Hasptr & Hasptr::operator=(const Hasptr &rhs){
    this->ptr=rhs->ptr;// 两者指向了同一内存
}
```

可见合成的拷贝构造函数和赋值操作都是不安全的。例如当

```
Hasptr foo(Hasptr para){
    Hasptr ret=para;    // ret和para指向同一内存
    ///...
```

```
    return ret;         // ret和para直接传入的形参都是指向的同一内存,但是调用析构函数
                        // delete了多次同一内存
}
```

需要拷贝的操作的类也需要赋值操作,反之亦然

大部分时候,不需要我们自定义析构函数,如没有构造函数没有动态分配内存,就不需要我们定义析构函数释放内存,靠编译器合成即可。

> 拷贝和赋值通常都是成对出现的。一个类需要自定义拷贝操作(必须要自定定义的,而不是那些自己定义了和合成无差别的),则几乎需要自定义赋值操作,反之亦然。

default

显示请求编译器合成拷贝控制操作。

阻止拷贝

> 大多数类而言,拷贝构造,默认构造,拷贝赋值等都是需要的,或者显示(自定义),或者隐式(编译器合成)给出。

某些类而言,则不需要拷贝构造和拷贝赋值,如`iostream`类,阻止拷贝或赋值操作,以**避免多个对象写入或读取相同的IO缓冲**。如果我们不定义拷贝和赋值,编译器则会为我们合成,因此我们应当显示请求编译器不要生成这些操作。

定义deleted函数

如同`=default`,我们使用`=deleted`,请求编译器不要生成对应的操作代码。

```
struct NoCopy{
    NoCopy()=default;
    ~NoCopy()=default;
    NoCopy(const NoCopy &)=delete;  // 阻止拷贝
    NoCopy &operator=(const NoCopy&)=delete;    // 阻止赋值
};
```

> `delete`也是提醒编译器以及**他人**定义这些成员。因此`delete`可以对任何函数使用,比如我们不希望一个函数定义加法操作,则将其定义为删除的。

> **定义为删除的函数,不会进行函数匹配**。即使我们定义,也不会匹配该函数。

析构函数不能是删除的

原因很显然,析构函数是删除的,那无法销毁该类型对象。如果该类将析构函数定义为`=delete`,则无法创建该类型的对象(包括临时对象)。同样,如果类中含有某个成员,该成员的析构函数是删除的,我们依然不能定义该类类型的任何对象。

> 析构函数如果定义为删除的，说明该类类型不可以在栈上创建对象，但可以在heap上创建，new，但是不可以delete，因为delete会调用析构函数。这种应用场景暂时不是很清楚。

合成的拷贝控制成员可能是删除的

- 如果类的某个成员的析构函数是删除或不可访问的（private），则类的合成析构函数被定义为删除的（很容易理解）
- the synthesized copy constructor is defined as delted:
  - if the class has a member whose own copy constructor is deleted or inaccessible.

```
Easy to understand.
```

  - if the class has a member with a deleted or inaccessible destructor.

```
If a member has a deleted or inaccessible destructor,
we could create objects that we could not destroy.
```

- the synthesized copy-assignment operator is defined as deleted if a member has a deleted or inaccessible copy-assignment operator, **or if the class has a const or reference member**.
  - const成员是不可改变的，因此合成的赋值操作符试图改变该成员。因此是错误的。
  - 含有引用类型的成员时，合成的赋值操作符，可以进行赋值操作，但是其行为不是我们希望的，例子如下

```
// It is impossible to assign a new value to a const object.
// But we can reassign a new value to a reference member.
class Foo{
    private:
        std::string s;// “工具人”
        std::string &str=s;// 默认使用s绑定左值引用str
    public:
        Foo()=default;// 不适用初始化列表绑定str的时候，str默认绑定到s上

        Foo(const string &s1,string &s2):s(s1),str(s2){}// 直接使用s绑定str
        // member functions
};
Foo &operator=(const Foo &f){
    // 虽然合法，但是我们改变的不是this对象中的引用本身，而且引用指向对象的值，因此该行为不是我们所希望的。
    this->s=f.s;
    this->str=f.str;
    return *this;
}
// ok, but this behaviour is unlikely to be desired.
// the left-hand operand would continue to refer to the same
// object as it did before the assignment. It would not refer the same
// object as the right-hand operand.
// Thus, the synthesized copy-assignment operator is defined as deleted, if the
```

```
class has a reference member.
// But, if we define the copy-assignment operator explicitly.
// The program works, although that is not our desired behavior.
```

- the synthesized default constructor is defined as deleted:
  - if the class has a member with a deleted or inaccessible destructor;
  - ... has a reference member that does not have an in-class initializer;
    - 此时，默认构造函数无法初始化此引用类型成员，因此会出错。
  - ... has a `const` member whose type does not explicitly define a default constructor and that member does not have an in-class initializer.

```
// wrong version
class Foo{
    private:
        const int val;
        const string & str;// string &str;
    public:
        Foo()=default;  // 默认构造函数无法初始化const成员和引用成员
        ~Foo()=default;
};
Foo f;// error, the default constructor is deleted.

// correct version
class Foo{
    private:
        const int val;
        const string &str="123";
    public:
        Foo(const int &v=2):val(v){};// expilicitly
        ~Foo()=default;
};
Foo f;// ok

class Foo{
    //... same as before
    const int val=3;
    const string &str="123";
    public:
        Foo()=default;
        //...
}
Foo f;// ok
```

private拷贝控制

C++11标准之前，通过将拷贝构造函数和拷贝赋值运算符声明为private的来阻止拷贝。

```
class PrivateFoo{
    // no access specifier; following members are private by default;
    // copy control is private and so is inaccessible to ordinary user code
    PrivateFoo(const PrivateFoo &);
    PrivateFoo &operator=(const PrivateFoo &);
    // ...
    public:
        PrivateFoo()=default;
        ~PrivateFoo();
};
```

这种情况下，析构函数时public的，因此可以创建该类类型的对象。但是不可拷贝以及赋值这个类型的对象，但是这个声明为deleted的区别即，该情况下，友元和类内成员函数是可以使用拷贝和赋值操作的。为了阻止友元和成员函数进行拷贝，我们将拷贝控制成员声明为private，**同时不定义它们**。

> We should use the new standard method to prevent copyinig,i.e., `=delete` rather than making those members `private`.

An interesting example

```
#include <iostream>
using namespace::std;
class numbered {
public:
    numbered() { mysn = ++num; }        //default constructor
    numbered(const numbered& n) {mysn = ++num; }// copy constructor
    numbered & operator=(numbered& n) {// overload assignment operator
        this->mysn = n.mysn;
        cout << "here" << std::endl;
        return  *this;
    }
    int mysn = 0;
    ~numbered() = default;
private:
    static int num;
};

int numbered::num = 0;
void f(numbered s){
    cout << s.mysn << endl;
}
int main(){
    numbered a;// call the default constructor，num equals 1
    f(a);// the parameter is nonreference, so the copy constructor will be called
    // thus, output 2
    numbered b;// num=3
    b = a;// call the assignment operator functions,out "here"
    f(b);// num=4,output 4
```

```
    numbered c = b;
    f(c);// 6
    return 0;
}
```