拷贝控制操作

一个类通常都会有五种操作称之为**拷贝控制操作**：

- 拷贝构造函数
- 拷贝赋值函数
- 移动构造函数
- 移动赋值函数
- 析构函数

1. 拷贝和移动构造定义了用同类型的另一个对象初始化被对象时做什么。
2. 拷贝和移动复制定义了将同类型对象赋予另一个对象时做什么。
3. 析构函数定义该对象销毁时做什么。

> 如果我们没有显示给出定义，则编译器负责合成以上函数。

拷贝构造函数

```
class Foo{
public:
    Foo();       // 默认构造函数
    Foo(const Foo &);    // 拷贝构造函数
};
```

> 拷贝构造函数的参数必须是引用类型，因为如果不是引用类型，那么实参传递给形参时，需要拷贝操作，而由此陷入死循环。const则不是必须的，但是应当总是const，因为拷贝构造不需要修改原对象。

拷贝初始化

拷贝初始化和直接初始化：

```
    string dots(10,'.');// direct initialization
    string s(dots); // direct initialization
    string s2=dots;// copy initialization
    string str="this is a string";// copy ...
    string str_s=string(100,'.');// copy...
```

1. Direct initialization:
   the compiler will use the ordinary function matching to select the constructor the best matches the arguments we provide.
2. Copy Initialization
   the compiler will copy the right-hand operand into the object being created, converting that operand if necessary.

> **copy initialization** ordinarily uses the **copy constructor**. (move constructor in some cases)

拷贝初始化何时会发生?

1. 使用=时
2. 实参传递给非引用形参

```
// class A
void foo(const A a);//
// when we call the function foo()
// the argument will be copied to the parameter a.
// meaning the copy constructor is called.
```

3. return the object from a function that has a nonreference return type

```
A foo(...){
    A a;
    return a;
}
// a will be copied to a temporary object of class A
```

```
class Foo{
    private:
        int a;
        string str;
    public:
        Foo()=default;
        Foo(const Foo &f){// copy constructor
            this->a=f.a;// built-in assignment
            this->str=f.str;// uses string-assignment operator
        }
        Foo & operator=(const Foo& f){
            this->a=f.a;
            this->str=f.str;
            return *this;
        }
};
Foo a=b;//  directly initialize Foo a by calling the copy constructor
Foo a(b);// equivalent to the previous one

Foo a;// declare Foo a and uses the default constructor.
a=b;// call the assignment operator function;
```

> 某些类类型会对它们分配的对象使用拷贝初始化,如insert或push_back成员。当然也有直接初始化,如emplace。

拷贝初始化的限制

Note that if we use an initializer that requires conversion by an **explicit** constructor. Here we recall the notion of explicit.

```
string str="string";
// this is an example of implicitly conversion
//  the complier will call the constructor, string(const char *);
// In the right hand, we thereafter have a string type (by compiler using
string("string")).
// The copy constructor to initialize str will be called.

void foo(string &s);// foo is function with the paremeter of type string

foo("string");// ok, "string" can be converted to a type string.

void foo(vector<int> &a);
foo(12);// error, constructor that takes a size is explicit meaning 12 cannot be
converted to a vector type.
vector<int> a(12);// ok,direct initialization
foo(vector<int> (12));// ok, directly construct a temporary vector from an int.
```

拷贝复制运算符

> 成员函数的第一个参数（隐含的）是this指针形参。

```
class Foo{
    Foo &operator=(const Foo&);// 返回该类型的引用是引用为了和内置类型的赋值保持一致
                               // 从而可以实现类似于obj1=obj2=obj3,这种赋值操作
                               // 或者对于一个函数fun(Foo ),可以这样调用fun(obj1=obj2)
};
Foo a,b;
a=b;// uses the Foo copy- assignment operator
// 等价于
a.operator=(b);
```

> 拷贝构造，赋值构造函数，它们只会负责非static数据成员。static数据成员和非static数据成员在内存分布中也不是在一起的。

析构函数

析构函数负责释放所有非static数据成员的资源。

```
class Foo{
    Foo()// constructor
    ~Foo()// destructor
};
```

析构函数调用时：

- 类对象成员，将会调用其析构函数来销毁。
- 内置类型的数据成员，系统会自动回收，析构函数什么也不需要做。

> 析构函数不可重载，原因也是很显然的，没必要重载析构函数，一个即可，负责释放相应的资源。
> Smart pointers are class type and have destructors. As a result, members that are smart pointers are automatically destroyed during the destruction phase.

什么时候调用析构函数？

1. 局部类对象离开作用域，即生命周期终止了。
2. 类对象被销毁，其成员被销毁。（如类A中包含类类型B数据成员，A销毁时，将会调用B的析构函数）
3. 容器被销毁时（vector<Foo>）
4. delete动态分配的对象（delete运算符是和析构函数关联起来的）

> 析构函数体并不直接销毁数据成员，成员是在**析构函数体**执行完之后，**隐含的析构阶段**中被销毁的。

这里给出一个需要自己定义析构函数的（自己管理内存资源释放）的例子。

```
class HasPtr {
public:
    HasPtr(const std::string& s = std::string("hello")) :ps(new std::string(s)), i(0)
{}
    HasPtr(const HasPtr& h) {
        this->i = h.i;
        //this->ps = h.ps; // error, this->ps and h.ps will point to the same object,
and thus the pointer will be deleted twice.
        // What happens is undefined.
        this->ps = new std::string(*(h.ps)); // ok
    }
    HasPtr& operator=(const HasPtr& h) {
        this->i = h.i;
        delete ps;
        //this->ps = h.ps;// error
        this->ps = new std::string(*(h.ps));// ok
        return *this;
    }
    ~HasPtr() {
        delete(ps);
        ps = nullptr;
        std::cout << "~HasPtr has been destroyed!" << std::endl;
    }
    void foo() {
        std::cout << this->i << std::endl;
        if (ps)std::cout << *(this->ps) << std::endl;
    }
private:
    std::string* ps = nullptr;
    int i;
```

```
};
int main() {
    HasPtr a;
    a.foo();
    HasPtr b(a);
    b.foo();
    HasPtr c = b;
    c.foo();
}
```