Copy control

> When we define a class, we specify-explicitly or implicitly-what happens when objects of that class type are copied, moved, assigned, and destroyed.

copy control:

1. The copy and move constructors define what happens when an object is initialized from another object of the same type.
2. The copy- and move- assignment operators define what happens when we assign an oject of a class type to another object of that same class type.
3. The destructor defines what happens when an object of the type ceases to exits.

> If we don't define the copy control members of class, the compiler defines automatically the missing operations.

Copy constructor

The only one that is merit know is the difference between assignment and copy.

Copy Initialization

```
    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)

What's useful to know is when copy initialization happens and that copy initialization requires either the copy constructor or the move constructor.

Copy initialization happens when we

- [Using an operator]using an oprerator =
  (An interesting example is given at the end of this documnet)

  ```
  // It is useful to realize that the the copy constructor or the assignment
  operator function is called.
  ```

```
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;
```

- pass an object as an argument to a parameter of nonreference type

```
// 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.
```

- 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
```

- brace initialization the elements in an array or the members of an **aggregate class**

Note: Some class type also use copy initialization for the objects the allocate. For example, the libary containers copy initialize their elements when we initialize the container, or when we call an insert or push member. By contrast, elements created by an emplace member are direct initialized.

> Thus, **the copy constructor's own paremeter must be a reference.** If not, the call would never succeed-to call the copy constructor, we'd need to use the copy constructor to copy the argument, but to copy the argument, we'd need to call the copy constructor, and so on indefinitely.

constraints on Copy Initialization

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.
```

The Copy-Assignment Operator

A class controls how objects of its class are assigned, just as a class controls how objects of that class are initialized.

```
class Foo{
    Foo()=default;
    //...
};
Foo a,b;
a=b;// uses the Foo copy- assignment operator
```

If the class does not define its own copy-assignment operator, the compiler synthesizes one as with the copy constructor. It assigns each `nonstatic` member of the right-hand object to the corresponding member of the left-hand object using the copy-assignment operator for the type of that member (See the equivalent copy-assignment function `Foo & operator=(const Foo &f)`).

A simple example is given here.

```
class Foo{
    private:
        int val;
        string str;
    public:
        Foo()=default;
        Foo(const int &v,const string&s):val(v),str(s){};


        Foo(const Foo &)=default;// copy construtor by compiler synthesizing
        // equivalent to the above statement
        Foo(const Foo& f){
            this->val=f.val;// built-in assignment
            this->str=f.str;// call the string-assignment operation for copy
initialization
        }

        Foo & operator=(const Foo &)=default;
        // equivalent to the above statement
        Foo & operator=(const Foo &a){
            this->val=a.val;// uses the built-in int assignment
            this->str=a.str;// calls the string::operator=(const string &)
            return *this;// return a reference to this object
        }
        ~Foo()=default;
};
```

Overloaded operators are functions that have the name operator followed by the symbol for the operator being defined. Hence, the assignment operator is a function named **operator=**. Like any other function, an operator function has a return type and a paremeter list.

```
using T=Class_Type;
T & operator=(const T &); // assignment operator
```

The parameter`const T &`represent the operands of the operator. Some operators must be defined as member functions. When an operator is a member function, the left-hand operand is bound to the implicit ***this*** parameter.

To be consistent with assignment for the built-in types, assignment operators usually return a reference to their left-hand operand.

```
int a=3;
int b;
void foo(int val){// output the value val}
foo(b=a);// the expression `b=a` returns a reference to b. Thus 3 will be printed.
```

The Destructor

The destructor operates inversely to the constructors:

- Constructors initialize the `nonstatic` data members of an object and may do ohter work;

- Destructor do whatever work is needed to free the resources used by an object and destroy the `nonstatic` data members of the object.

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

Note that *it takes no paremeter, and thereby cannot be overloaded.* There is always only one destructor for a given class.

What a Destructor Does
Let's recall the constructor. In a constructor, members are initialized before the function body is executed.

In a destructor, the function body is executed first and then the members are destoryed in reverse order from the order in which they were initialized.

> The destruction part is implicit.

What happens when a member is destroyed depends on the type of the member:

- members of class type are destoryed by running the member's own destructor.
- the built-in types do not have destructors, nothing is done to destroy members of built-in type.

> Smart pointers are class type and have destructors. As a result, members that are smart pointers are automatically destroyed during the destruction phase.

When a Destructor Is Called

The destructor is used automatically whenever an object of its type is destroyed:

- variables are destroyed when they go out of scope.
- members of an object are destroyed when the object of which they are a part is destroyed.
- elements in a container are destroyed when the container is destroyed.
- dynamiclly allocated objects are destroyed when the `delete` operator is applied to a pointer to the object.
- temporary objects are destroyed at the end of the full expression in which the temporary was created.

```
// class Foo...
{// a scope
    Foo * p=new Foo;
    auto p2=make_shared<Foo>();// p2 is a shared_ptr
    //auto p2=(shared_ptr<Foo>) new Foo;// equivalent to the previous one

    vecotr<Foo> vec;// local object
    vec.push_back(*p2);// copies the object to which p2 points
    delete(p);// destroy expilicitly the object that p points to
}
// go out of the scope;
// destructor called on p2 and vec
// destroying p2 and decrements its use count. If the use count goes to 0, the
object that p2 points to is freed
// destroying vec destroys the elements in vec
```

> It is important to realize that the destructor body does not directly destroy the members themselves. Members are destroyed as part of the implicit destruction phase that follows the destructor function body.

Some class needs to define a destructor to free the memory allocated by its constructor. In this case, the copy constructor and copy-assignment operator need to be defined by users rather than synthesized by the compiler.

Here an example is shown.

```
#include<iostream>
#include<string>
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();
}
```

Preventing Copies

> For some classes there really is no sensible meaning for copy constructor and copy-assignment operator. The such classes must be defined so as to prevent copies or assignments from being made.(e.g. `iostream` classes)

Defining a Function as Deleted

using `=delete` can reach this purpose.

> A deleted functions is one that is declared but may not be used in any other way. The `=delete` signals to the compiler that we are intentionally not defining these members.

```
class Foo{
    Foo()=default;// use the systhesized default constructor
    Foo(const Foo &)=delete;// no copy
    Foo &operator=(const Foo &)=delete;// no assignment
    ~Foo()=default;
};
```

Note that **the destructor should not be a `deleted` member**.

The copy-Control Members May Be Synthesized as Deleted

For some cases, the compiler defines these synthesized members as delted functions:

- the synthesized destructor is defined as deleted if the class has a member whose own destructor is deleted or is inaccessible(e.g.,`private`).

  ```
  Easy to understand.
  ```

- 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**.

```
// 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;
    - ... 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;
            ~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
```

In essence, these rules mean that if a class type has a data member that cannot be default constructed, copied, assigned, or destroyed, then the corresponding member will be a deleted function.

Private Copy Control

Prior to the new standard, classes pervented copies by declaring their copy constructor and copy-assignment operator as `private`.

```
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();
};
```

Since the destructor is `public`, users are able to define `PrivateFoo` objects. However, the copy constructor and copy-assignment operator are `private`(inaccessible), user code will not be able to copy such objects.

Note: Friends and members of the class can still make copies. To prevent copies by friends and members, we declare these members as `private` but do not define them.

> 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;
}
```