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.

   ```
   class A{
       private:
           int val;
       public:
           A()=default;// default constructor;

           A(const A &)=default;// copy constructor systhesized by compiler
           A(const A & a){this->a=a.a;}// copy constructor defined explicitly
   };
   ```

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.

   ```
   class A{
       private:
           int a;
       public:
           // ...
           A & operator=(const A &a){
               this->a=a->a;
               return *this;
           }
   }
   ```

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 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 oprerator =
- 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<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.

```
~Foo()
```