【C++】Day57

Class	C++
Date	@February 16, 2022
Material	
Series Number	
Summary	Classes define destructors must define copy constructor and copy-assignment operator, synthesized =default, and = delete

【Ch13】Copy Control

13.1.4 The Rule of Three/Five

Classes That Need Destructors Need Copy and Assignment

When we decide whether a class needs to define its own versions of the copy-control members is to decide first whether the class needs a destructor.

It a class needs a destructor, it almost surely needs a copy constructor and copy-assignment operator as well.

The `HasPtr` class that we have used is a good example. That class allocates dynamic memory in its constructor. The synthesized destructor will not delete a data member that is a pointer. Therefore, this class needs to define a destructor to free the memory allocated by its constructor.

What may e less clear-but what our rule of thumb tells us-is that `HasPtr` also needs a copy constructor and copy-assignment operator.

Consider what would happen if we gave `HasPtr` a destructor but used the synthesized version of the copy constructor and copy-assignment operator:

```
class HasPtr {
public:
  HasPtr(const std::string &s = string()) : ps(new string(s)), i(0) {}
  ~HasPtr() {
    delete ps;
  }
  // WRONG: HasPtr needs a copy constructor and copy-assignment operator
  // other members as before
};
```

In this version of the class, the memory allocated in the constructor will be freed when a `HasPtr` object is destroyed. This version of the class uses the synthesized versions of copy and assignment. Those functions copy the pointer member, meaning that multiple `HasPtr` objects may be pointing to the same memory.

```
HasPtr f(HasPtr hp) {
  HasPtr ret = hp;
  return ret;
}
```

When `f` returns, both `hp` and `ret` are destroyed and the `HasPtr` destructor is run on each of these objects. That destructor will delete the point member in `ret` and in `hp`. This code will delete that pointer twice, which is an error.

Tip: If a class needs a destructor, it almost surely also needs the copy-assignment operator and a copy constructor.

Exercise

Exercise 13.14: Assume that `numbered` is a class with a default constructor that generates a unique serial number for each object, which is stored in a data member named `mysn`. Assuming `numbered` uses the synthesized copy-control members and given the following function:

Click here to view code image

```
void f (numbered s) { cout << s.mysn << endl; }
```

what output does the following code produce?

Click here to view code image

Fifth Edition

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

See 13_14.cpp for code

13.1.5 Using = default

We can explicitly ask the compiler to generate the synthesized versions of the copy-control members by defining them as `= default`:

```
class Sales_data {
public:
  Sales_data() = default;
  Sales_data(const Sales_data&) = default;
  Sales_data& operator=(const Sales_data&);
  ~Sales_data() = default;
};
```

Note: We can use `=default` only on member functions that have a synthesized version.

13.1.6 Preventing Copies

Best Practices: Most classes should define-either implicitly or explicitly-the default and copy constructors and the copy-assignment operator.

Defining a Function as Deleted

Under the new standard, we can prevent copies by defining the copy constructor and copy-assignment operator as deleted functions. A deleted function is one that is declared but may not be used in any other way.

We indicate that we want to define a function as deleted by following its parameter list with `= delete`:

```
struct NoCopy {
  NoCopy() = default; // use the synthesized default constructor
  NoCopy(const NoCopy&) = delete; //no copy
  NoCopy& operator=(cosnt NoCopy&) = delete; //no assignment
  ~NoCopy() = default; //uses the synthesized destructor
};
```

The `=vdelete` signals to the compiler(and to readers of our code) that we are intentionally not defining these members.

Unlike `= default`, `= delete` must appear on the first declaration of a deleted function.

Also unlike `= default`, we can specify `= delete` on any function.(We can only use `= default` on the default constructor or a copy-control member that the compiler can synthesize).

The Destructor Should Not be a Deleted Member

It is worth noting that we did not delete the destructor. If the destructor is delted, then there is no way to destroy objects of that type. The compiler will not let us define variables or create temporaries of a type that has a deleted destructor.

Warning: It is not possible to define an object or delete a pointer to a dynamically allocated object of a type with a deleted destructor.