【C++】Day25(3)

Class	C++
Date	@December 16, 2021
Material	
Series Number	
Summary	

【Ch7】Classes

7.5 Constructors Revisited

7.5.1 Constructor Initializer List

If we do not explicitly initialize a member in the constructor initializer list, that member is default initialized before the constructor body starts executing.

For example:

```
//legal but sloppier way to write the Saeles_data constructor:no constructor initializers
Sales_data::Sales_data(const string &s, unsigned cnt, double price) {
  bookNo = s;
  unit_sold = cnt;
  revenue = cnt * price;
}
```

This version assigns values to the data members.

Constructor Initializers are Sometimes Required

We can often, but not always, ignore the distinction between whether a member is initialized or assigned. Members that are `const` or references must be initialized. Similarly, members that are of a class type that does not define a default constructor also must be initialized. For example:

```
class ConstRef {
public:
  ConstRef(int ii);
private:
  int i;
  const int ci;
  int &ri;
}
```

Like any other const object or reference, the members `ci` and `ri` must be initialized. As a result, omitting a constructor initializer for these members is an error:

```
ConstRef::ConstRef(int ii) : i(ii), ci(ii), ri(ii) {}
```

By the time the body of the constructor begins executing, initialization is complete. Our only chance to initialize const or reference data members is in the constructor initializer.

Note: We must use the constructor initializer list to provide values for members that are const, reference, or of a class type that does not have a default constructor.

Advice: Use Constructor Initializers

In many classes, the distinction between initialization and assignment is a matter of low-level efficiency: a data member is initialized and then assigned when it could be have been initialized directly.

Order of Member Initialization

Each member may be named only once in the constructor initializer.

What might be surprising is that the constructor initializer list specifies only the values used to initialize the members, not the order in which those initialization are performed.

Members are initialized in the order in which they appear in the class definition. The order in which initializers appear in the constructor initializer list does not change the order of initialization.

As an example, look at the following class:

```
class X {
  int i, j;
public:
  //undefined: i is initialized before j
  X(int val) : j(val), i(j) {}
}
```

In this case, i is initialized first. The effect of this initializer is to initialize i with the undefined value of j.

Best Practices: It is a good idea to write constructor initializers in the same order as the members are declared. Moreover, when possible, avoid using members to initialize other members.