【Effective CPP】Day5(2)

Book	Effective C++
Author	
Summary	
Date	@2022/05/12

【Ch2】Constructors, Destructors, and Assignment Operators

Item 9: Never call virtual functions during construction or destruction

We shouldn't call virtual functions during construction or destruction, because the calls won't do what we think.

Imagine we want a class called Transaction to records buys and sells:

```
class Transaction {
public:
  Transaction();
  virtual void logTransaction() const = 0; // Make type-dependent log entry
};

Transaction::Transaction() {
  logTransaction();
}

class BuyTransaction : public Transaction {
public:
  virtual void logTransaction() const;
};

class SellTransaction : public Transaction {
public:
  virtual void logTransaction() const;
};
```

When we execute the following code:

```
BuyTransaction b;
```

The version of `logTransaction` that is called is the one in `Transaction`, not the one in `BuyTransaction`. Because the base part is constructed first, by which time the `logTransaction` in `BuyTransaction` class is not seen by the compiler. The linker will thus complain.

We can easily avoid this by making `logTransaction` non-virtual, then require that derived class constructors pass the necessary log information to the `Transaction` constructor.

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);
  void logTransaction(const std::string& logInfo) const;
};

Transaction::Transaction(const std::string& logInfo) {
  logTransaction(logInfo);
}

class BuyTransaction : public Transaction {
public:
  BuyTransaction(params) : Transaction(createLogString(params)) {}

private:
  static std::string createLogString(params);
};
```

In this example, the use of the private static function `createLogString` helps create a value of pass to a base class constructor.

By making the function static, there's no danger of accidentally referring to the nascent `BuyTransaction` object's as-yet-uninitialized data members.

Things to Remember

Don't call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor.