The following is an excerpt from Scott Meyers' new book, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*.

Item 9: Never call virtual functions during construction or destruction.

I'll begin with the recap: you shouldn't call virtual functions during construction or destruction, because the calls won't do what you think, and if they did, you'd still be unhappy. If you're a recovering Java or C# programmer, pay close attention to this Item, because this is a place where those languages zig, while C++ zags.

Suppose you've got a class hierarchy for modeling stock transactions, e.g., buy orders, sell orders, etc. It's important that such transactions be auditable, so each time a transaction object is created, an appropriate entry needs to be created in an audit log. This seems like a reasonable way to approach the problem:

```
class Transaction {                              // base class for all
public:                                          // transactions
  Transaction();
  virtual void logTransaction() const = 0;       // make type-dependent
                                                 // log entry
  ...
};
Transaction::Transaction()                       // implementation of
{                                                // base class ctor
  ...
  logTransaction();                              // as final action, log this
}                                                // transaction
class BuyTransaction: public Transaction {       // derived class
public:
  virtual void logTransaction() const;           // how to log trans-
                                                 // actions of this type
  ...
};
class SellTransaction: public Transaction {      // derived class
public:
  virtual void logTransaction() const;           // how to log trans-
                                                 // actions of this type
  ...
};
```

Consider what happens when this code is executed:

```
BuyTransaction b;
```

Clearly a BuyTransaction constructor will be called, but first, a Transaction constructor must be called; base class parts of derived class objects are constructed before derived class parts are. The last line of the Transaction constructor calls the virtual function logTransaction, but this is where the surprise comes in. The version of logTransaction that's called is the one in Transaction, *not* the one in BuyTransaction — even though the type of object being created is BuyTransaction. During base class construction, virtual functions never go down into derived classes. Instead, the object behaves as if it were of the base type. Informally speaking, during base class construction, virtual functions aren't.

There's a good reason for this seemingly counterintuitive behavior. Because base class constructors execute before derived class constructors, derived class data members have not been initialized when base class constructors run. If virtual functions called during base class construction went down to derived classes, the derived class functions would almost certainly refer to local data members, but those data members would not yet have been initialized. That would be a non-stop ticket to undefined behavior and late-night debugging sessions. Calling down to parts of an object that have not yet been initialized is inherently dangerous, so C++ gives you no way to do it.

It's actually more fundamental than that. During base class construction of a derived class object, the type of the object *is* that of the base class. Not only do virtual functions resolve to the base class, but the parts of the language using runtime type information (e.g., dynamic_cast (see Item 27) and typeid) treat the object as a base class type. In our example, while the Transaction constructor is running to initialize the base class part of a BuyTransaction object, the object is of type Transaction. That's how every part of C++ will treat it, and the treatment makes sense: the BuyTransaction-specific parts of the object haven't been initialized yet, so it's safest to treat them as if they didn't exist. An object doesn't become a derived class object until execution of a derived class constructor begins.

The same reasoning applies during destruction. Once a derived class destructor has run, the object's derived class data members assume undefined values, so C++ treats them as if they no longer exist. Upon entry to the base class destructor, the object becomes a base class object, and all parts of C++ — virtual functions, dynamic_casts, etc., — treat it that way.

In the example code above, the Transaction constructor made a direct call to a virtual function, a clear and easy-to-see violation of this Item's guidance. The violation is so easy to see, some compilers issue a warning about it. (Others don't. See Item 53 for a discussion of warnings.) Even without such a warning, the problem would almost certainly become apparent before runtime, because the logTransaction function is pure virtual in Transaction. Unless it had been defined (unlikely, but possible — see Item 34), the program wouldn't link: the linker would be unable to find the necessary implementation of Transaction::logTransaction.

It's not always so easy to detect calls to virtual functions during construction or destruction. If Transaction had multiple constructors, each of which had to perform some of the same work, it would be good software engineering to avoid code replication by putting the common initialization code, including the call to logTransaction, into a private non-virtual initialization function, say, init:

```
class Transaction {
public:
  Transaction()
  { init(); }                                  // call to non-virtual...
  virtual void logTransaction() const = 0;
  ...
private:
  void init()
  {
    ...
    logTransaction();                          // ...that calls a virtual!
  }
};
```

This code is conceptually the same as the earlier version, but it's more insidious, because it will typically compile and link without complaint. In this case, because logTransaction is pure virtual in Transaction, most runtime systems will abort the program when the pure virtual is called (typically issuing a message to that effect). However, if logTransaction were a "normal" virtual function (i.e., not pure virtual) with an implementation in Transaction, that version would be called, and the program would merrily trot along, leaving you to figure out why the wrong version of logTransaction was called when a derived class object was created. The only way to avoid this problem is to make sure that none of your constructors or destructors call virtual functions on the object being created or destroyed and that all the functions they call obey the same constraint.

But how *do* you ensure that the proper version of logTransaction is called each time an object in the Transaction hierarchy is created? Clearly, calling a virtual function on the object from the Transaction constructor(s) is the wrong way to do it.

There are different ways to approach this problem. One is to turn logTransaction into a non-virtual function in Transaction, then require that derived class constructors pass the necessary log information to the Transaction constructor. That function can then safely call the non-virtual logTransaction. Like this:

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);
  void logTransaction(const std::string& logInfo) const;// now a non-
                                                        // virtual func
  ...
};

Transaction::Transaction(const std::string& logInfo)
{
  ...
  logTransaction(logInfo);                              // now a non-
}                                                       // virtual call

class BuyTransaction: public Transaction {
public:
  BuyTransaction( parameters )
  : Transaction(createLogString( parameters ))         // pass log info
  { ... }                                               // to base class
  ...                                                   // constructor

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

In other words, since you can't use virtual functions to call down from base classes during construction, you can compensate by having derived classes pass necessary construction information up to base class constructors instead.

In this example, note the use of the (private) static function createLogString in BuyTransaction. Using a helper function to create a value to pass to a base class constructor is often more convenient (and more readable) that going through contortions in the member initialization list to give the base class what it needs. By making the function static, there's no danger of accidentally referring to the nascent BuyTransaction object's as-yet-uninitialized data members. That's important, because the fact that those data members will be in an undefined

state is why calling virtual functions during base class construction and destruction doesn't go down into derived classes in the first place.

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.