【Effective CPP】Day3

Book	Effective C++
Author	
Summary	
Date	@2022/05/10

【Ch1】Accustoming to C++

Item 4: Make sure that objects are initialized before they are used

In some contexts, a variable is guaranteed to be initialized. For example,

```
int x;
```

`x` is guaranteed to be initialized to 0. But in others, it's not. If we write

```
class Point {
  int x, y;
};

Point p;
```

`p`'s data members are sometimes guaranteed to be initialized(to zero), but sometimes they're not.

The best way to deal with this indeterminate state of affairs is to always initialize objects before using them. For non-member objects of built-in types, we'll need to do this manually. For example,

```
int x = 0;
const char* text = "A C-style string";

double d;
std::cin >> d; // Initialization by reading from an input stream
```

For almost everything else, the responsibility for initialization falls on constructors.

However, there is a difference between assignment and initialization. Check the code below:

```
class PhoneNumber {...};

class ABEntry {
public:
  ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones);

private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int numTimesConsulted;
};
```

If we write the constructor as following:

```
ABEntryconst(std::string& name, const std::string& address, const std::list<PhoneNumber>& phones) {
  theName = name;
  theAddress = address;
  phones = thePhones;
  numTimesConsulted = 0;
}
```

This will yield `ABEntry` objects with the values we expect. However, it's still not the best approach. This way of constructing an object uses assignment rather than initialization.

The best approach would be to use member initialization list instead of assignments:

```
ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones)
  : theName(name), theAddress(address), phones(thePhones), numTimesConsulted(0) {}
```

We can also use the member initialization list even when we want to default construct a data member; just specify nothing as an initialization argument.

```
ABEntry::ABEntry() : theName(), theAddress(), thePhoens(), numTimesConsulted(0) {}
```

A `static` object is one that exists from the time it's constructed until the end of the program. Included are global objects, objects defined at namespace scope, objects declared static inside classes, objects declared static inside functions, and objects declared static at file scope.

Static objects inside functions are known as local static objects.(because they are local to a function)

The other kinds of static objects are known as non-local static objects. Static objects are destroyed when the program exits.

When we have one object whose initialization requires another object, the order of initialization becomes important. Unfortunately, we cannot determine explicitly the order of initialization of the two objects. See following:

```
class FileSystem {
public:
  std::size_t numDisks() const;
};

extern FileSystem tfs;
```

```
class Directory {
public:
  Directory(params);
};

Directory::Directory(params) {
  std::size_t disks = tfs.numDisks(); // Use the tfs object
}
```

Now the importance of initialization order becomes apparent: unless tfs is initialized before `tempDir`, `tempDir`'s constructor will attempt to use tfs before it's been initialized.

A small design change will solve this issue. We can replace the global static object with local static objects by replacing direct accesses to non-local static objects with calls to functions that return references to local objects.

We are guaranteed that the references we get back will refer to initialized objects.

```
class FileSystem{...};

FileSystem &tfs() {
  ststic FileSystem fs; // Define and initialize a local static object
  return fs;
}

class Directory {...};

Directory::Directory(params) {
  std::size_t disks = tfs().numDisks();
}
```

Things to Remember

1. Manually initialize objects of built-in type, because C++ only sometimes initializes them itself
2. In a constructor, prefer use of the member initialization list to assignment inside the body of the constructor. List data members in the initialization lsit in the same order they're declared in the class
3. Avoid initialization order problems across translation units by replacing non-local static objects with local static objects.