【Effective CPP】Day5

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

【Ch2】Constructors, Destructors, and Assignment Operators

Item 7: Declare destructors virtual in polymorphic base classes

Imagine we want to create a class to keep track of time:

```
class TimeKeeper {
public:
  TimeKeeper();
  ~TimeKeeper();
};

class AtomicClock: public TimeKeeper{...};
class WaterClock: public TimeKeeper{...};
```

Many clients want access to the time without worrying about the details of how it's calculated, so a factory function-a function that returns a base class pointer to a newly-created derived class object-can be used to return a pointer to a timekeeping object:

```
TimeKeeper* getTimeKeeper(); // Returns a pointer to a dynamically allocated object of a class derived from TimeKeeper
```

The returned object is on the heap, so it is important that it's properly deleted after use:

```
TimeKeeper *ptk = getTimeKeeper();
...
delete ptk;
```

The problem is that `getTimeKeeper` returns a pointer to a derived class object, that object is being deleted via base class pointer(i.e., a `TimeKeeper*` object), and the base class(`TimeKeeper`) has a non-virtual destructor.

This is a disaster, because C++ specifies that when a derived class object is deleted through a pointer to a base class with a non-virtual destructor, results are undefined.

What typically happens at runtime is that the derived part of the object is never destroyed.

Eliminating the problem is simple: give the base class a virtual destructor. Then deleting a derived class object will do exactly what we want. It will delete the entire object.

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
};
```

Base classes like `TimeKeeper` generally contain virtual functions other than the destructor, because the purpose of virtual functions is to allow customization of derived class implementations.

For example, `TimeKeeper` might have a virtual function, `getCurrentTime`, which would be implemented differently in the various derived classes.

If a class does not contain virtual functions, that often indicates it is not meant to be used as a base class. When a class is not intended to be a base class, making the destructor virtual is usually a bad idea.

```
class Point {
public:
  virtual ~Point();
private:
  int x, y;
};
```

A `Point` object can typically fit into a 64-bit register. However, if `Point`'s destructor is made virtual, the situation changes.

The implementation of virtual functions requires that objects carry information that can be used at runtime to determine which virtual functions should be invoked.

This information typically takes the form of a pointer called a vptr("virtual table pointer"), which will increase the size of our Point object.

Things to Remember

1. Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
2. Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.