【Effective CPP】Day7

Book	Effective C++
Author	
Summary	
Date	@2022/05/15

【Ch3】Resource Management

Common resources include memories, file descriptors, mutex locks, fonts and brushes in GUIs, database connections, and network sockets. Regardless of the resource, it's important that it be released when we are finished it.

Item 13: Use objects to manage resources

Suppose we are working with a library for modelling investments(e.g., stocks, bonds, etc.), where the various investment types inherit from a root class Investment:

```
class Investment{...}; // Root class of hierachy of investment types
```

Further suppose that the way the library provides us with specific Investment objects is through a factory function:

```
Investment* createInvestment(); // Return ptr to dynamically allocated object in the Investment hierarchy;
// The caller must delete it
```

Consider, a function f written to delete the dynamically allocated object after use:

```
void f() {
  Investment *pInv = createInvestment();
  ...
  delete pInv;
}
```

This looks okay, but there are several ways f could fail to delete the investment object.

There might be a premature return statement somewhere inside the "..." part of the function. If such a return were executed, control would never reach the `delete` statement.

Some statement in “...” may throw an exception. If so, control would again not get to the `delete`.

To make sure that the resource returned by `createInvestment` is always released, we need to put that resource inside an object whose destructor will automatically release the resource when control leaves f.

The standard library’s `auto_ptr` is tailor-made for this kind of situation. `auto_ptr` is a pointer-like object(a smart pointer) whose destructor automatically calls delete on what it points to. Here’s how to use auto_ptr to prevent f’s potential resource leak:

```
void f() {
  std::auto_ptr<Investment> pInv(createInvestment());
}
```

This simple example demonstrates the two critical aspects of using objects to manage resources:

- **Resources are acquired and immediately turned over to resource-managing objects.**

  The idea of using objects to mange resources is often called Resource Acquisition Is Initialization.

- **Resource-managing objects use their destructors to ensure that resources are released.**

Because an `auto_ptr` automatically deletes what it points to when the `auto_ptr` is destroyed, it’s important that there never be more than one `auto_ptr` pointing to an object.

If there were, the object would be deleted more than once, and that would put our program on undefined behaviour.

To prevent such problems, `auto_ptrs` have an unusual characteristic: copying them sets them to null, and the copying pointer assumes sole ownership of the resource:

```
std::auto_ptr<Investment> pInv1(createInvestment());

std::auto_ptr<Investment> pInv2(pInv1); // pInv1 is now null

pInv1 = pInv2; // pInv2 is now null
```

An alternative to `auto_ptr` is a reference-counting smart pointer.(RCSP) An RCSP is a smart pointer that keeps track of how many objects point to a particular resource and automatically deletes the resource when nobody is point to it any longer.

```
void f() {
  std::shared_ptr<Investment> spInv(createInvestment());
}
```

Note: Unfortunately, neither `auto_ptr` nor `shared_ptr` can destroy dynamically allocated array. Doing so would result in undefined behavior.(It will compile though)

Things to Remember

- To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.
- Two commonly useful RAII classes are `tr1::shared_ptr` and `auto_ptr`. `shared_ptr` is usually the better choice.