【C++】Day52

Class	C++
Date	@February 6, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

12.1.4 Smart Pointers and Exceptions

Programs that use exception handling to continue processing after an exception occurs need to ensure that resources are properly freed if an exception occurs. One easy way to make sure resources are freed is to use smart pointers.

When we use a smart pointer, the smart pointer class ensures that memory is freed when it is no longer needed even if the block is exited prematurely:

```
void f() {
  shared_ptr<int> p(new int(1024));
  //code that throws an exception that is not caught inside f
} //shared_ptr freed automatically when the function ends
```

When a function is exited, whether through normal processing or due to an exception, all the local objects are destroyed. In this case, `p` is a `shared_pointer`, so destroying `p` checks its reference count. Here, `p` is the only pointer to the memory it manages; that memory will be freed as part of destroying `p`.

In contrast, memory that we manage directly is not automatically freed when an exception occurs. If we use built-in pointers to manage memory and an exception occurs after a `new` but before the corresponding `delete`, then that memory won't be freed:

```
void f() {
  int *ip = new int(1024); //dynamically allocate a new object
  //code taht throws an exception that is not caught inside f
  delete ip; //free the memory before exiting
}
```

If an exception happens between the `new` and the `delete`, and is not caught inside f, then this memory can never be freed. There is no pointer to this memory outside the function `f`. Thus, there is no way to free this memory.

Smart Pointers and Dumb Classes

Many C++ classes, including all the library classes, define destructors that take care of cleaning up the resources used by that object. However, not all classes are so well behaved. In particular, classes that are designed to be used by both C and C++ generally require the user to specifically free any resources that are used.

Using Our Own Deletion Code

By default, `shared_ptr`s assume that they point to dynamic memory. Hence, by default, when a `shared_ptr` is destroyed, it executes delete on the pointer it holds.

To use a `shared_ptr` to manage a connection, we must first define a function to use in place of `delete`. It must be possible to call this deleter function with the pointer stored inside the `shared_ptr`. In this case, our deleter must take a single argument of type `connection*`.

```
void end_conneciton(connection *p) { disconnect(*p); }
```

When we create a `shared_ptr`, we can pass an optional argument that points to a deleter function:

```
void f(destination &d) {
  connection c = connect(&d);
```

```
    shared_ptr<connection> p(&c, end_connection);
}
```

Caution: Smart Pointer Pitfalls

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they re used properly. To use smart pointers correctly, we must adhere to a set of conventions:

1. Don't use the same built-in pointer value to initialize(or reset) more than one smart pointer
2. Don't delete the pointer returned from `get()`
3. Don't use `get()` to initialize or reset another smart pointer
4. If we use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.
5. If we use a smart pointer to manage a resource other than memory allocated by new, remember to pass a deleter.

Exercise

Exercise 12.14: Write your own version of a function that uses a `shared_ptr` to manage a `connection`.

See 12_14.cpp