15.x — Chapter 15 comprehensive review

BY ALEX ON MAY 3RD, 2017 | LAST MODIFIED BY ALEX ON AUGUST 16TH, 2017

A smart pointer class is a composition class that is designed to manage dynamically allocated memory, and ensure that memory gets deleted when the smart pointer object goes out of scope.

Copy semantics allow our classes to be copied. This is done primarily via the copy constructor and copy assignment operator.

Move semantics mean a class will transfer ownership of the object rather than making a copy. This is done primarily via the move constructor and move assignment operator.

std::auto_ptr is deprecated and should be avoided.

An r-value reference is a reference that is designed to be initialized with an r-value. An r-value reference is created using a double ampersand. It's fine to write functions that take r-value reference parameters, but you should almost never return an r-value reference.

If we construct an object or do an assignment where the argument is an l-value, the only thing we can reasonably do is copy the l-value. We can't assume it's safe to alter the l-value, because it may be used again later in the program. If we have an expression "a = b", we wouldn't reasonably expect b to be changed in any way.

However, if we construct an object or do an assignment where the argument is an r-value, then we know that r-value is just a temporary object of some kind. Instead of copying it (which can be expensive), we can simply transfer its resources (which is cheap) to the object we're constructing or assigning. This is safe to do because the temporary will be destroyed at the end of the expression anyway, so we know it will never be used again!

You can use the delete keyword to disable copy semantics for classes you create by deleting the copy constructor and copy assignment operator.

std::move allows you to treat an l-value as r-value. This is useful when we want to invoke move semantics instead of copy semantics on an l-value.

std::unique_ptr is the smart pointer class that you should probably be using. It manages a single non-shareable resource. std::make_unique() (in C++14) should be preferred to create new std::unique_ptr. std::unique_ptr disables copy semantics.

std::shared_ptr is the smart pointer class used when you need multiple objects accessing the same resource. The resource will not be destroyed until the last std::shared_ptr managing it is destroyed. std::make_shared() should be preferred to create new std::shared_ptr. With std::shared_ptr, copy semantics should be used to create additional std::shared_ptr pointing to the same object.

std::weak_ptr is the smart pointer class used when you need one or more objects with ability to view and access a resource managed by a std::shared_ptr, but unlike std::shared_ptr, std::weak_ptr is not considered when determining whether the resource should be destroyed.

Quiz time

1) Explain when you should use the following types of pointers.

1a) std::unique_ptr

Hide Solution

std::unique_ptr should be used when you want a smart pointer to manage a dynamic object that is not going to be shared.

1b) std::shared_ptr

Hide Solution

std::shared_ptr should be used when you want a smart pointer to manage a dynamic object that may be shared. The object won't be deallocated until all std::shared_ptr holding the object are destroyed.

1c) std::weak_ptr

Hide Solution

std::weak_ptr should be used when you want access to an object that is being managed by a std::shared_ptr, but don't want the lifetime of the std::shared_ptr to be tied to the lifetime of the std::weak_ptr.

1d) std::auto_ptr

Hide Solution

std::auto_ptr is deprecated and slated for removal in C++17. It should not be used.

2) Explain how r-values references enable move semantics.

Hide Solution

Because r-values are temporary, we know they are going to get destroyed after they are used. When passing or return an r-value by value, it's wasteful to make a copy and then destroy the original. Instead, we can simply move (steal) the r-value's resources, which is generally more efficient.

3) What's wrong with the following code? Update the programs to be best practices compliant.

3a)

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
    Resource *res = new Resource;
    std::shared_ptr<Resource> ptr1(res);
    std::shared_ptr<Resource> ptr2(res);

    return 0;
}
```

Hide Solution

ptr2 was created from res instead of from ptr1. This means that you now have two std::shared_ptr each independently trying to manage the Resource (they are not aware of each other). When one goes out of scope, the other will be left with a dangling pointer.

ptr2 should be copied from ptr1 instead of from res, and std::make_shared() should be used

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
    auto ptr1 = std::make_shared<Resource>();
    auto ptr2(ptr1);

    return 0;
}
```

3b)

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Something; // assume Something is a class that can throw an exception

int main()
{
    doSomething(std::shared_ptr<Something>(new Something), std::shared_ptr<Something>(new Something));

    return 0;
}
```

Hide Solution

If the constructor for Something throws an exception, one of the Somethings may not be deallocated properly.

The solution is to use make_shared:

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Something; // assume Something is a class that can throw an exception

int main()
{
    doSomething(std::make_shared<Something>(), std::make_shared<Something>());

    return 0;
}
```



16.1 -- The Standard Template Library (STL)


Index


15.7 -- Circular dependency issues with std::shared_ptr, and std::weak_ptr

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

16 comments to 15.x — Chapter 15 comprehensive review

Matthias
November 1, 2017 at 9:49 am · Reply

In 3b you wrote:

doSomething(std::shared_ptr<Something>(new Something), std::shared_ptr<Something>(new Something));
[...]
If the constructor for Something throws an exception, one of the Somethings may not be deallocated properly.

But: At the time "doSomething()" runs, both shared_ptrs are constructed and will deallocate properly at some point.
The problem is that the compiler might choose to create both "new Something"s before storing them inside their shared_ptrs,