A CODERS JOURNEY
LIFE LESSONS FROM A CODER

CODE CRAFT

p 10 dumb mistakes to avoid th C++ 11 smart pointers

May 15, 2016 Deb Haldar 49 Comments C++, C++ 11, C++ 14

I love the new C++ 11 smart pointers. In many ways, they were a godsent for many folks who hate managing their own memory. In my opinion, it made teaching C++ to newcomers much easier.

However, in the two plus years that I've been using them extensively, I've come across multiple cases where improper use of the C++ 11 smart pointers made the program inefficient or simply crash and burn. I've catalogued them below for easy reference.

Before we begin, let's take a look at a simple Aircraft class we'll use to illustrate the mistakes.

Search

RECENT POSTS

System Design Interview Concepts – Consistent Hashing

Top 20 C++ multithreading mistakes and how to avoid them

Coding Interview Question – Optimizing toy purchases

C++11 Multithreading Tutorial via Q&A – Thread Management Basics

6 Tips to supercharge C++11 vector performance

Event based synchronization of threads with main game loop

Selecting an API ? Watch out for these 9 Red Flags !

Archives

October 2017 (1)

August 2017 (2)

```
class Aircraft
{
private:
        string m_model;

public:

        int m_flyCount;

        weak_ptr myWingMan;

        void Fly()
        {
                cout << "Aircraft type" << m_model <<
.s flying !" << endl;
        }

        Aircraft(string model)
        {
                m_model = model;
                cout << "Aircraft type " << model <<
 is created" << endl;
        }

        Aircraft()
        {
                m_model = "Generic Model";
                cout << "Generic Model Aircraft
:eated." << endl;
        }

        ~Aircraft()
        {
                cout << "Aircraft type  " << m_model
<< " is destroyed" << endl;
        }

};
```


Mistake # 1 : Using a shared pointer where an unique pointer suffices !!!

I've recently been working in an inherited codebase which uses a shared_ptr for creating and managing every object. When I analyzed the code, I found that in 90% of the cases, the resource wrapped by the shared_ptr is not shared.

This is problematic because of two reasons:

1. If you have a resource that's really meant to be owned exclusively, using a shared_ptr instead of a unique_ptr makes the code susceptible to unwanted resource leaks and bugs.

- **Subtle Bugs:** Just imagine if you never imagined a scenario where the resource is shared out by some other programmer by assigning it to another shared pointer which inadvertently modifies the resource !
- **Unnecessary Resource Utilization:** Even if the other pointer does not modify the shared resource, it might hang on to it far longer than necessary thereby hogging your RAM unnecessarily even after the original shared_ptr goes out of scope.

eating a shared_ptr is more resource intensive than creating a ue_ptr.

A shared_ptr needs to maintain the threadsafe refcount of objects it points to and a control block under the covers which makes it more heavyweight than an unique_ptr.

ommendation – By default, you should use a unique_ptr. If a rement comes up later to share the resource ownership, you can ys change it to a shared_ptr.

Mistake # 2 : Not making resources/objects shared by shared_ptr threadsafe !

Shared_ptr allows you to share the resource thorough multiple pointers which can essentially be used from multiple threads. It's a common mistake to assume that wrapping an object up in a shared_ptr makes it inherently thread safe. It's still your responsibility to put synchronization primitives around the shared resource managed by a shared_ptr.

Recommendation – If you do not plan on sharing the resource between multiple threads, use a unique_ptr.

Mistake # 3 : Using auto_ptr !

The auto_ptr feature was outright dangerous and has now been deprecated. The transfer of ownership executed by the copy constructor when the pointer is passed by value can cause fatal crashes in the system when the original auto pointer gets 'erenced again. Consider an example:


```
ıt main()

        auto_ptr myAutoPtr(new Aircraft("F-15"));
        SetFlightCountWithAutoPtr(myAutoPtr); //
ıvokes the copy constructor for the auto_ptr
        myAutoPtr->m_flyCount = 10; // CRASH !!!
```

ɔmmendation – unique_ptr does what auto_ptr was intended to 'ou should do a search and find on your codebase and replace ɩto_ptr with unique_ptr. This is pretty safe but don't forget to t your code ! 🙂

ıtake # 4 : Not using make_shared to ialize a shared_ptr !

ɜ_shared has two distinct advantages over using a raw pointer:

1. Performance : When you create an object with new , and then create a shared_ptr , there are two dynamic memory allocations that happen : one for the object itself from the new, and then a second for the manager object created by the shared_ptr constructor.

```
shared_ptr pAircraft(new Aircraft("F-16")); // Two
Dynamic Memory allocations - SLOW !!!
```

On the contrary, when you use make_shared, C++ compiler does a single memory allocation big enough to hold both the manager object and the new object.

```
shared_ptr pAircraft = make_shared("F-16"); // Single
allocation - FAST !
```

~~**2. Safety:** Consider the situation where the Aircraft object is created and then for some reason the shared pointer fails to be created. In case, the Aircraft object will not be deleted and will cause ory leak !~~ After looking at the implementation in MS compiler ory header I found that if the allocation fails, the resource/object leted. So Safety is no longer a concern for this type of usage.

ommendation: Use make_shared to instantiate shared pointers ad of using the raw pointer.

take # 5 : Not assigning an object(raw nter) to a shared_ptr as soon as it is ated !

bject should be assigned to a shared_ptr as soon as it is ed. The raw pointer should never be used again.

ider the following example:

```
ıt main()
{
        Aircraft* myAircraft = new Aircraft("F-16");

        shared_ptr pAircraft(myAircraft);
        cout << pAircraft.use_count() << endl; //
ref-count is 1

        shared_ptr pAircraft2(myAircraft);
        cout << pAircraft2.use_count() << endl; //
ref-count is 1

        return 0;
}
```

It'll cause an ACCESS VIOLATION and crash the program !!!

The problem is that when the first shared_ptr goes out of scope, the myAircraft object is destroyed. When the second shared_ptr goes out of scope , it tries to destroy the previously destroyed object again !

ommendation: If you're not using make_shared to create the ed_ptr , at least create the object managed by the smart pointer e same line of code – like :

```
iared_ptr pAircraft(new Aircraft("F-16"));
```

take # 6 : Deleting the raw pointer used by shared_ptr !

can get a handle to the raw pointer from a shared_ptr using the ***ed_ptr.get()*** api. However, this is risky and should be avoided. sider the following piece of code:

```
oid StartJob()

        shared_ptr pAircraft(new Aircraft("F-16"));
        Aircraft* myAircraft = pAircraft.get(); //
eturns the raw pointer
        delete myAircraft;  // myAircraft is gone
}
```

Once we get the raw pointer (myAircraft) from the shared pointer, we delete it. However, once the function ends, the shared_ptr pAircraft goes out of scope and tries to delete the myAircraft object which has already been deleted. The result is an all too familiar ACCESS VIOLATION !

Recommendation: Think really hard before you pull out the raw pointer from the shared pointer and hang on to it. You never know when someone is going to call delete on the raw pointer and cause your shared_ptr to Access Violate.

Mistake # 7 : Not using a custom deleter when using an array of pointers with a shared_ptr !

Consider the following piece of code:


```
id StartJob()

        shared_ptr ppAircraft(new Aircraft[3]);
```

shared pointer will just point to Aircraft[0] — Aircraft[1] and aft[2] have memory leaks will not be cleaned up when the smart er goes out of scope. If you're using Visual Studio 2015, you'll heap corruption error.

ommendation: Always pass a custom delete with array objects aged by shared_ptr. The following code fixes the issue:

```
id StartJob()

        shared_ptr ppAircraft(new Aircraft[3],
(Aircraft* p) {delete[] p; });
```

Mistake # 8 : Not avoiding cyclic references when using shared pointers !

In many situations , when a class contains a shared_ptr reference , you can get into cyclical references. Consider the following scenario – we want to create two Aircraft objects – one flown my Maverick and one flown by Iceman (I could not help myself from using the TopGun reference !!!). Both maverick and Iceman needs to hold a reference to each Other Wingman.

So our initial design introduced a self referencial shared_ptr inside the Aircraft class:

class Aircraft
{
private:
string m_model;
public:
int m_flyCount;
shared_ptr<Aircraft> myWingMan;
….

in our main() , we create Aircraft objects, Maverick and Goose , nake them each other's wingman:

```
ıt main()

        shared_ptr pMaverick = make_shared("Maverick:
-14");
        shared_ptr pIceman = make_shared("Iceman:
-14");

        pMaverick->myWingMan = pIceman; // So far so
ood - no cycles yet
        pIceman->myWingMan = pMaverick; // now we got
 cycle - neither maverick nor goose will ever be
estroyed

        return 0;
```

When main() returns, we expect the two shared pointers to be destroyed – but neither is because they contain cyclical references to one another. Even though the smart pointers themselves gets cleaned from the stack, the objects holding each other references keeps both the objects alive.

Here's the output of running the program:

Aircraft type Maverick: F-14 is created
Aircraft type Iceman: F-14 is created

So what's the fix ? we can change the shared_ptr inside the Aircraft class to a weak_ptr ! Here's the output after re-executing the main().

Aircraft type Maverick: F-14 is created
Aircraft type Iceman: F-14 is created
Aircraft type Iceman: F-14 is destroyed
Aircraft type Maverick: F-14 is destroyed

:e how both the Aircraft objects were destroyed.

ommendation: Consider using weak_ptr in your class design ı ownership of the resource is not needed and you don't want to te the lifetime of the object.

:take # 9 : Not deleting a raw pointer ırned by unique_ptr.release() !

Release() method does not destroy the object managed by the ıe_ptr, but the unique_ptr object is released from the ınsibility of deleting the object. Someone else (YOU!) must e this object manually.

ollowing code below causes a memory leak because the Aircraft :t is still alive at large once the main() exits.

```
ıt main()
{
        unique_ptr myAircraft = make_unique("F-22");
        Aircraft* rawPtr = myAircraft.release();
        return 0;
}
```

Recommendation: Anytime you call Release() on an unique_ptr, remember to delete the raw pointer. If your intent is to delete the object managed by the unique_ptr, consider using unique_ptr.reset().

Mistake # 10 : Not using a expiry check when

calling weak_ptr.lock() !

Before you can use a weak_ptr, you need to acquire the weak_ptr by calling a lock() method on the weak_ptr. The lock() method essentially upgrades the weak_ptr to a shared_ptr such that you can 't. However, if the shared_ptr object that the weak_ptr points to is nger valid, the weak_ptr is emptied. Calling any method on an ed weak_ptr will cause an ACESS VIOLATION.

example, in the code snippet below, the shared_ptr that /ingMan" weak_ptr is pointing to has been destroyed via nan.reset(). If we execute any action now via myWingman _ptr, it'll cause an access violation.

```
ıt main()

        shared_ptr pMaverick = make_shared("F-22");
        shared_ptr pIceman = make_shared("F-14");

        pMaverick->myWingMan = pIceman;
        pIceman->m_flyCount = 17;

        pIceman.reset(); // destroy the object
ınaged by pIceman

        cout <<
laverick->myWingMan.lock()->m_flyCount << endl; //
:CESS VIOLATION

        return 0;
}
```

It can be fixed easily by incorporating the following if check before using the myWingMan weak_ptr.

```
        if (!pMaverick->myWingMan.expired())
        {
                cout <<
pMaverick->myWingMan.lock()->m_flyCount << endl;
        }
```

EDIT: As many of my readers pointed out, the above code should not be used in a multithreaded environment – which equates to 99% of the software written nowadays. The weak_ptr might expire between the time it is checked for expiration and when the lock is acquired on it. A HUGE THANKS to my readers who called it out ! I'll adopt **Manuel Freiholz's** solution here : Check if the shared_ptr is not empty after ...g lock() and before using it.


```
...red_ptr<aircraft> wingMan = pMaverick->myWingMan.lock();
...vingMan)

    cout << wingMan->m_flyCount << endl;
```

...ommendation: Always check if a weak_ptr is valid – actually if a ...empty shared pointer is returned via lock() function before using ...your code.

What's Next ?

...u want to learn more about the nuances of C++ 11 smart pointers ...++ 11 in general, I recommend the following books.

...++ Primer (5th Edition) by Stanley Lippman

...fective Modern C++: 42 Specific Ways to Improve Your Use of ...11 and C++14 by Scott Meyers

...e best in your journey of exploring C++ 11 further. Please share if you liked the article. 🙂

← Are you safe from a market collapse if you're a long term investor ?

Top 10 C++ header file mistakes and how to fix them →

49 Comments **A Coders Journey** 1 Login

Recommend 1 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

1.9k Shares

985

106

Marcelo Cantos • a year ago

The given solution to #10 is flat-out wrong (not thread-safe, as another poster points out). The best solution is to lock and test in one go:

```
if (auto wingMan = pMaverick->myWingMan.lock()) {
  cout << wingMan->m_flyCount << endl;
}
```

This has the advantage of making it impossible to accidentally refer to wingMan when it's null.

7 • Reply • Share ›

Deb Haldar Mod Marcelo Cantos • a year ago

Yup - should be fixed now.

• Reply • Share ›

Marcelo Cantos Deb Haldar • a year ago

Sure, but the advice to not use the original version in multithreaded code is still wrong. The original version should never be used at all. It has no advantages, and the disadvantage — even in the single-threaded case — that it has to check twice for pointer validity. It's unlikely the compiler can optimise away the double-check, since the compiler cannot assume a single-threaded environment.

• Reply • Share ›

Deb Haldar Mod Marcelo Cantos • a year ago

Marcelo, so I looked at the implementation of each in VC++ compiler. If the object has indeed expired, in a single threaded case this is cheaper because you don't invoke the shared pointer constructor and hinge on a simple count kept internally. That's the theoretical side. Practically, when I profiled both solutions in a loop 1M times, I found no difference in performance. If you find evidence to the contrary, I'd be curious to know and happy to edit the article.