

[Blog](#)[Home](#)[Archives](#)[Home](#)[Archives](#)

# Modern C++ Memory Management With `unique_ptr`

📅 2016-02-06 🏷️ #C++

This post is going to be a general background article on `unique_ptr` and how/why you should use it (if you are not already). In my current line of work I still deal with a large C++03 codebase, but with efforts ongoing to pull C++11 into the application I have spent a great deal of time thinking about how we can make the most out of C++11. One of the biggest wins is smart pointers and manual memory management, which are usable in C++03 with `boost`, but are so much more powerful in C++11 thanks to move semantics. I will focus on `unique_ptr` in this post as a start, but that doesn't mean you shouldn't also use the other smart pointer types included in C++11, `shared_ptr` and `weak_ptr`, when appropriate.

## Anatomy Of A `unique_ptr`

The `unique_ptr` introduced with C++11 (based on `boost::scoped_ptr`) is conceptually very simple. It wraps a raw pointer to a heap-allocated object with `RAII` semantics, destroying the associated object with the `unique_ptr` (i.e. at the end of whatever scope it is declared in). Because object destruction is triggered by `RAII`, there is no runtime or memory overhead for `unique_ptr` compared to a raw pointer. There are 3 ways to bind an object instance to a `unique_ptr`:

- The `constructor` that takes a raw `T*`.

```
1 std::unique_ptr<MyClass> instance(new MyClass());
```

- With `make_unique` (the return value of which can be move-assigned).

```
1 auto instance = std::make_unique<MyClass>();
```

- By using the `reset` member function.

```
1 std::unique_ptr<MyClass> instance;  
2 instance.reset(new MyClass());
```

~~There is only one way to destroy the object associated with the `unique_ptr` early, before the `unique_ptr` itself is destroyed.~~[1] Destroying the contained object early (before the `unique_ptr` itself is destroyed) can be triggered by calling the `reset` member function, which can also optionally take another raw pointer that the `unique_ptr` should subsequently take ownership of. However, before taking ownership of the new pointer it will always ensure the previous object is deleted first. If no new pointer is passed to `reset`, the `unique_ptr` will hold `nullptr` after the current object is deleted. The same logic applies to assignments; a `unique_ptr` can be move-assigned an object from another `unique_ptr`, or assigned `nullptr`. In both cases any object already held by the `unique_ptr` will be deleted before accepting the new value.

Using smart pointers is semantically the same as with raw pointers. The `*` and `->` operators have both been overloaded to provide familiar mechanics for accessing the underlying object:

```
1 std::unique_ptr<MyClass> instance(new MyClass());  
2 std::cout << instance->v << std::endl;  
3 int y = (*instance).calculate();
```

In fact modern smart pointers in C++ go one further by defining an implicit `bool` conversion; so the `== nullptr`, `== 0`, `== NULL`, or whatever null-pointer constant your organization has chosen, can be excluded.

```
1  std::unique_ptr<MyClass> instance(new MyClass());
2  std::unique_ptr<MyClass> noinstance;
3  assert(instance); // `instance` is initialized and valid
4  assert(!noinstance); // `noinstance` is uninitialised and invalid
```

## Block-Scoped Object Lifetimes

The time/space within the program between the creation and deletion of an object is the object's **lifetime**: the part of the program for which the object is valid. With smart pointers, this is dictated by the **owning** smart pointer(s) controlling the object's lifetime.

Consider the following C++03-ish example:

```
1  {
2      MyClass* instance(new MyClass());
3      // ... more code ...
4      delete instance;
5  }
```

With C++11 and `unique_ptr` this becomes:

```
1  {
2      std::unique_ptr<MyClass> instance(new MyClass());
3      // ... more code ...
4  } // `instance` deallocated here automatically
```

Or better yet, using `make_unique` from C++14:

```
1  {
2      auto instance = std::make_unique<MyClass>();
```

```
3      // ... more code ...  
4  } // `instance` deallocated here automatically
```

This concept can be extended with class scopes to yield a wider range of object lifetimes we can express using `unique_ptr`. For example, the traditional [PIMPL pattern](#) (simplified for this example):

```
1  class Outer {  
2  public:  
3      Outer() : impl(new Inner()) {};  
4      ~Outer() { delete impl; }  
5  
6  private:  
7      Inner* impl;  
8  }
```

With modern C++ this can be refactored to:

```
1  class Outer {  
2  public:  
3      Outer() : impl(std::make_unique<Inner>()) {};  
4  
5  private:  
6      std::unique_ptr<Inner> impl;  
7  }
```

Note we don't even need to define a destructor anymore, because our `impl` object is automatically destroyed when the `Outer` object is destroyed.

## Why Smart Pointers Instead Of Manual Memory Management?

I have had people ask this question when I have suggested using smart pointers instead of traditional manual memory management (using `delete`), because their existing code works perfectly fine and causes no leaks. The most obvious benefit of using smart pointers is avoiding memory leaks (`shared_ptr` `reference cycles` being an exception). But how does using smart pointers avoid leaks?

When manual memory management is done correctly it does indeed work, but it becomes more brittle over time as code is refactored and extended. As a simplified example, say we had some code like this hiding somewhere in our application:

```
1  {
2      MyClass* instance = new MyClass();
3      // ... more code ...
4      delete instance;
5  }
```

Then, while adding a new feature, someone modifies the code so there are multiple places where a pointer is initialized with an instance of different types (using `runtime polymorphism`). In reality a mistake like this is much less obvious than in this example. This example intentionally shows very poor design to make the flaw more obvious.

```
1  {
2      MyClass* instance = 0;
3      if (someCondition) {
4          instance = new SubclassA();
5      }
6      // ... more code ...
7      if (someOtherCondition) {
8          instance = new SubclassB();
9      }
10     // ... more code ...
11     delete instance;
12 }
```

Whoops, we may have just caused a memory leak: if both conditions are `true` the first object assigned to `instance` is never deleted. We could fix this up by calling `delete` to dispose of the first object before we create the second if the `instance` pointer is not zero.

```
1  {
2      MyClass* instance = 0;
3      if (someCondition)
4          instance = new SubclassA();
5      // ... more code ...
6      if (someOtherCondition) {
7          if (instance != nullptr)
8              delete instance;
9          instance = new SubclassB();
10     }
11     // ... more code ...
12     delete instance;
13 }
```

This sort of breakage can be very common when modifying code that uses traditional manual memory management techniques comprised of `delete` calls scattered throughout code. This error could have just as well been a `double free` due to over-application of `delete`. It could have also been a `use after free` error where the pointer is not reset to `nullptr` after deletion, and our program continues to use it unaware until suddenly `Cthulhu` starts wreaking havoc on our application. The more resilient modern technique to eliminate all of these simple errors is to apply smart pointers in these scenarios instead wherever possible. For example, if we rewrite the above original example using `unique_ptr`:

```
1  {
2      auto instance = std::make_unique<MyClass>();
3      // ... more code ...
4  } // `instance` automatically deleted here
```

With the same naive refactoring applied this would become:

```
1  {
2      std::unique_ptr<MyClass> instance;
3      if (someCondition) {
4          instance.reset(new SubclassA()); // no delete here, `ins
5      }
6      // ... more code ...
7      if (someOtherCondition) {
8          instance.reset(new SubclassB()); // any previous object
9      }
10     // ... more code ...
11 } // `instance` automatically deleted here
```

And it Just Works™. No pitfalls to be seen here; the API of smart pointers does *not* allow an already held pointer to be overwritten without first triggering a release of the object associated with that pointer. It's extremely difficult to screw up the usage of `unique_ptr` in these types of scenarios.

## Raw Pointers/References And `unique_ptr`

So with the advent of smart pointers in modern C++ are we supposed to completely throw away raw pointers and references? Of course not!

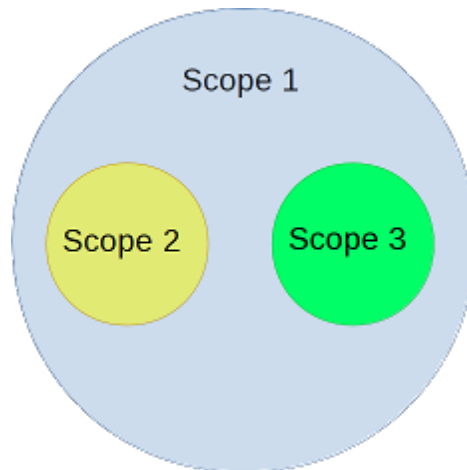
Raw pointers and references still serve a purpose for referencing/accessing objects without affecting their lifetime. The usual rules apply: for raw pointers and references to be valid, the lifetime of the raw pointers/references must be a **subset** of the lifetime of the object being referred to. It is easier to align the lifetimes of both smart and raw pointers if RAII and scope are used to manage both.

Consider the following example:

```
1  {
2      // Scope 1
3      auto instance = std::make_unique<MyClass>();
4      {
```

```
5      // Scope 2
6      MyClass* ptrA = instance.get();
7      // ... some usage of ptrA ...
8  }
9  // ... more code ...
10 {
11     // Scope 3
12     MyClass* ptrB = instance.get();
13     /// ... some usage of ptrB
14 }
15 }
```

The lifetimes in this example could be represented by the following Venn diagram:



The scopes of the raw pointers both fall completely within `_Scope 1_`

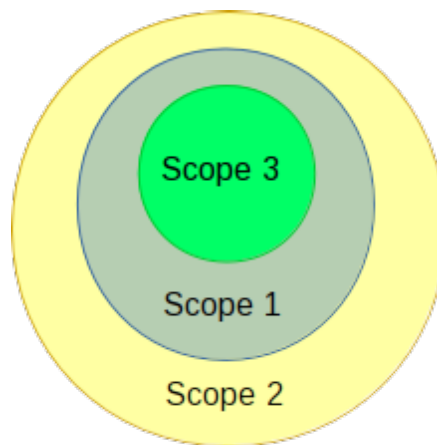
Whereas if we got the lifetimes wrong and did something like this:

```
1  {
2      // Scope 2
3      MyClass* ptrA;
4      {
5          // Scope 1
6          auto instance = std::make_unique<MyClass>();
7          ptrA = instance.get();
```



```
8      {  
9          // Scope 3  
10         MyClass* ptrB = instance.get();  
11         /// ... some usage of ptrB  
12     }  
13 }  
14 // ... some usage of ptrA ...  
15 // oops! ptrA is invalid because `instance` and the associat  
16 }
```

The associated Venn diagram of the lifetimes would look more like this:



Whoops, part of `ptrA`'s Scope 2 is outside of `instance`'s Scope 1!

This would result in a use-after-free error when we try to access `ptrA` after the end of Scope 1, and possibly an application crash.

## Smart Pointers For A Better Tomorrow

There are a multitude of safety and maintainability advantages when using smart pointers instead of traditional manual memory management with direct calls to `delete`. Using smart pointers won't provide any immediate gratification over working `delete` code, but in the long term they make code much more resilient in the face of maintenance, refactoring and extension.

With the addition of `shared_ptr`, `weak_ptr` and `move semantics` from C++11 there should be *few* real scenarios which still *require* manual `delete` calls. Hopefully as C++ continues

to develop and advance this will be ever more true. `delete` is still useful to have in your C++ development toolbox, but it should be a tool of last resort.


Stay tuned for a follow-up post on `shared_ptr` and `weak_ptr`.

Update:

[1] Thanks to /u/immutablestate on reddit for pointing out that [assignment operations can also trigger an early release of an object held by a `unique\_ptr`](#).

Thanks to /u/corysama and /u/malaprop0s for other corrections.

---

 Share

## NEWER

Simple Artificial Neural Networks with FANN and C++

## OLDER

Travis CI and Modern C++

## RECENTS

THE MAKING OF A MASTERS

2017-02-28

TYPE-SAFE UNIONS IN C++ AND RUST

2016-10-07

PASSING REFERENCES TO DEFERRED FUNCTION CALLS WITH `STD::REF`

2016-05-29

SIMPLE ARTIFICIAL NEURAL NETWORKS WITH FANN AND C++  
2016-03-19

MODERN C++ MEMORY MANAGEMENT WITH UNIQUE\_PTR  
2016-02-06

## ARCHIVES

- February 2017 (1)
- October 2016 (1)
- May 2016 (1)
- March 2016 (1)
- February 2016 (1)
- January 2016 (1)

## TAGS

- C++ (5)
- Machine Learning (1)
- Masters (1)
- Rust (1)

## TAG CLOUD

C++ Machine Learning Masters Rust



---

© 2017 Nick Sarten  
Powered by [Hexo](#). Theme by [PPOffice](#)