Shahar Mike's Web Spot

Yet another geek's blog

Exploring std::unique_ptr (1104 words)

Sat, Nov 12, 2016

Table of Contents

Today we'll talk about C++'s built-in smart pointer `std::unique_ptr`, which is an extremely powerful, simple & common tool.

`std::unique_ptr`

C++98 had `std::auto_ptr`. It's problematic in areas I do not wish to discuss here, and so it was deprecated and replaced by the awesome `unique_ptr`.

`unique_ptr` is a simple 'smart' pointer: it holds an instance of an object and deletes it when it goes out of scope. In terms of lifetime behvior it's much like any regular C++ object with a constructor and destructor, only with dynamic memory allocation. No reference counting, no fancy tricks.

And that's the beauty. Simple, elegant & efficient, yet extremely powerful. With `unique_ptr` you no longer have to worry about `new` and `delete`. Simply call `make_unique()` (instead of `new`), and the destructor will call `delete` automatically. It's as simple as that, and covers 90% of use-cases that require dynamic memory.

`std::make_unique()`

Above I mentioned `make_unique()`. This is a new standard function that came in C++14. You may think that it's supposed to save us the need to typing the type we're interested in, similar to `make_pair`. Well, not quite. Let's look at `make_unique()`'s signature:

```
// inside namespace std
template <typename T, typename ... Args>
std::unique_ptr<T> make_unique(Args&&... args);
// '&&' here means forwarding references, not necessarily rvalues. I
// explain what these are in a future post.
```

Note that `T` is not an argument to the function, thus it can't possible be deduced by the compiler. So to use `make_unique()` one must *always* provide T explicitly, like so:

```
auto u_int = std::make_unique<int>(123);
cout << (*u_int == 123) << endl;
auto u_string = std::make_unique<std::string>(3, '#');
cout << (*u_string == "###") << endl;
```

Output:

```
1
1
```

So essentially all `make_unique()` does is call `new` and pass `args` as arguments to `T`'s constructor. As a matter of fact, here is a feature-complete implementation:

```
template <typename T, typename ... Args>
std::unique_ptr<T> make_unique(Args&& ... args) {
  return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
  // If you don't know what std::forward it simply read it as:
```

```
  // return std::unique_ptr<T>(new T(args...));
}
```

That's it. If so, why do we even need it? What's the difference between the following?:

```
auto a = std::make_unique<MyClass>();
auto b = std::unique_ptr<MyClass>(new MyClass());
std::unique_ptr<MyClass> c(new MyClass());
```

First difference is exception safety. In C++ < 17 calling the following *can* lead to a memory leak:

```
MyFunction(std::unique_ptr<MyClass>(new MyClass()),
           std::unique_ptr<MyClass>(new MyClass()));
```

How? Well, C++ doesn't define the order of evaluation even between sub-expressions, so in theory it could evaluate the first `new MyClass()`, then the second `new MyClass()` and only then `unique_ptr`'s constructors. Now, if the first call to `new` succeeds and the second call to `new` throws an exception (like from `MyClass`'s constructor) it would leak memory as no class owns the newly created object yet.

But it's not just exception safety. Look at the lines above, comparing the initialization of `a`, `b` and `c`. I think that `a` is the cleanest and least verbose. Furthermore, it's the only one who doesn't repeat `MyClass` twice.

And one last thing - I also prefer using `make_unique()` for assignment, not only for construction:

```
auto a = std::make_unique<int>(123);
a = std::make_unique<int>(456);  // Cool.
a.reset(new int(789));  // Works, but not as nice.
```

How to use `unique_ptr`

Use `unique_ptr` to represent any owned object that's not shared. Here are some common examples:

Return a dynamically allocated object from a function

```
std::unique_ptr<MyObject> CreateMyObject();
```

This way whoever *calls* your function won't leak. Even if they don't assign the returned value to a variable:

```
SomeFunction();
CreateMyObject();  // no assignment, yet no leak
SomeOtherFunction();
```

Take ownership of a dynamically-allocated object

```
void TakeOwnership(std::unique_ptr<AnObject> obj);
```

With such signature callers can’t ignore the fact that you’re taking ownership of obj:

```
TakeOwnership(3);  // ERROR: no conversion from 'int' to 'std::unique

int* p = new int(3);
TakeOwnership(p);  // ERROR: no conversion from 'int*' to 'std::uniqu
                   // This is because unique_ptr's constructor is exp

auto u = std::make_unique<int>(3);
TakeOwnership(u);  // ERROR: no copy constructor

TakeOwnership(std::move(u));  // OK
TakeOwnership(std::make_unique<int>(3));  // OK
TakeOwnership(nullptr);  // OK -- due to constructor accepting nullp
```

As you may have noticed, `unique_ptr` supports C++11’s move semantics, but does not allow copying. This makes sense – as the name suggests, each instance is supposed to only track a unique instance.

Dynamically-allocated class members

Use `unique_ptr` to automatically release class members when a class is released:

```
class SomeObject {
  // No destructor needed

private:
  std::unique_ptr<SomeElse> m_SomethingElse;
};
```

Casting

`unique_ptr` supports construction of `unique_ptr<T>` from `unique_ptr<U>` if `T*` is convertible to `U*` (which usually means up-casting). Example:

```
struct Base { virtual ~Base() = default; };
struct Derived : Base {};

// ...
std::unique_ptr<Derived> derived = std::make_unique<Derived>();
std::unique_ptr<Base> base(std::move(derived));
```

Note that we must call `std::move()` on `derived` – that's because there can't be 2 instances of `unique_ptr` pointing to the same object, even if they are of different types.

Custom deleter

`unique_ptr` allows to specify a custom object that will be used for releasing the object. To use this, however, one must provide a second template argument. For example:

```
FILE* file = fopen("...", "r");
auto FILE_releaser = [](FILE* f) { fclose(f); };
std::unique_ptr<FILE, decltype(FILE_releaser)> file_ptr(file, FILE_re
```

As demonstrated above, this can be very useful when working with C APIs, or APIs which have custom release logic.

Notes:

- `file_ptr` above is not compatible with `std::unique_ptr<FILE>` as their 2nd template argument is different. Move-assignment, for example, will fail.
- `file_ptr` still have the size of a single pointer. However, if we created `FILE_releaser` such that it captured variables – then `file_ptr`'s size would have increased as well.

Misusing `unique_ptr`

If you try hard, you *could* do some nasty things with `unique_ptr`. But you have to put some effort to do so. Here are a few examples:

Assigning the same pointer to multiple `unique_ptr`s

```
int* p = new int(123);
std::unique_ptr<int> a(p);
std::unique_ptr<int> b(p);  // Oops - b's destructor will double dele
```

The above example can be avoided by always using `make_unique()` instead of calling `new` directly.

Here's a similar example, but done without calling `new` directly:

```
std::unique_ptr<int> a = std::make_unique<int>(123);
std::unique_ptr<int> b(a.get());
```

Using `unique_ptr`'s constructor directly is not recommended, and passing the pointer returned by `.get()` to a function that is taking ownership is an error.

Deleting memory managed by `unique_ptr`s

```
auto u = std::make_unique<int>(123);
delete u.get();
```

This is an example of why you:

- Don't want to call `delete` directly;
- Are not supposed to mess with `unique_ptr`'s memory

A word about arrays

`unique_ptr` also have partial specializaion to handle arrays. Specifically it calls `delete[]` rather than `delete`. However, using C-style arrays is something that should generally be avoided. Prefer `std::array` or `std::vector` where possible.

That's it for today

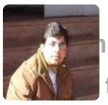
In the next post we'll talk about `std::shared_ptr` – `unique_ptr`'s brother which is very interesting, however less frequently used.

<< Exploring std::string Exploring std::shared_ptr >>

Comments

2 Comments **ShaharMike.com** **Upadhyay Vikas**

Recommend 11 Tweet Share **Sort by Best**



harMike.com requires you to verify your email address before posting. Send verification to vicky.with.luck@gmail.com ×

Join the discussion…

Alan King • a year ago

thanks - this was very helpful. I was lost inside boost smart pointer jungle, and this post helped me sort out a good pattern.

1 ^ | v • Reply • Share ›

Namgoo Lee • a year ago • edited

Thanks for this great post. It really helped me to sort out good practices for unique_ptr.

^ | v • Reply • Share ›

ALSO ON **SHAHARMIKE.COM**

Return Value Optimization | Shahar Mike's Web Spot

5 comments • 2 years ago

Bartlomiej Filipek — Nice!In C++17 we have guaranteed copy elision: but only for temp objects, so not for NRVOmore in:

Shahar Mike's Web Spot

3 comments • 4 years ago

shaharmike — By doing:(gdb) p d1 $3 = {<parent> = {_vptr$Parent = 0x400b50 <vtable for="" derived+16="">}, <no data=""

Template SFINAE & type-traits | Shahar Mike's Web Spot

1 comment • 4 years ago

Sozin — I am commenting because no one

UserData class | Shahar Mike's Web Spot

2 comments • 4 years ago

shaharmike — Wrapper is templated, and thus we can't have a generic non-templated