Shahar Mike's Web Spot

Yet another geek's blog

Exploring std::shared_ptr (1979 words)

Sun, Nov 13, 2016

Table of Contents

Today we'll talk about C++'s built-in smart pointer `std::shared_ptr`. If you have not yet read my previous post about `std::unique_ptr` I would highly recommend doing so before continuing.

`std::shared_ptr`

`shared_ptr` is another C++11 managed pointer. Like `unique_ptr`, it also saves you the need to call `new` and `delete` (and to generally worry about forgetting to release etc).

Unlike `unique_ptr`, `shared_ptr` can be shared. This means that multiple instances of `shared_ptr<T>` pointing to the same instance of `T` can co-exist. This is achieved via reference counting in a control block that's shared by all `shared_ptr`s pointing to the same object. When the last `shared_ptr` pointing to an instance of `T` is released, `T` is released as well.

Releasing a `shared_ptr` can be done in the following ways:

- Most commonly: via going out of scope, meaning calling the destructor automatically;
- Through assignment of another `shared_ptr`;
- Through calling `reset()` (more on this later).

Let's look at an example:

```
#include <iostream>
#include <memory>

struct Snitch {
public:
  Snitch() { std::cout << "c'tor" << std::endl; }
  ~Snitch() { std::cout << "d'tor" << std::endl; }
  Snitch(Snitch const&) { std::cout << "copy c'tor" << std::endl; }
  Snitch(Snitch&&) { std::cout << "move c'tor" << std::endl; }
};

int main() {
  auto snitch = std::make_shared<Snitch>();
  auto another_snitch = snitch;
  std::cout << "Equal?: " << (snitch == another_snitch) << std::endl

  {
    auto scoped_snitch = snitch;
    auto another_scoped_snitch = scoped_snitch;
  }  // destroy 'another_scoped_snitch' and 'scoped_snitch'
}  // destroy 'snother_snitch' and 'snitch'
```

Output:

```
c'tor
Equal?: 1
d'tor
```

Note that only 1 instance of `Snitch` is ever created, and that no copy/move constructors are used. All of `snitch`, `another_snitch`, `scoped_snitch` and `another_scoped_snitch` are equal. Also note that `snitch` has the type `shared_ptr<Snitch>`, as this is `make_shared()`'s return type:

```
std::is_same<decltype(snitch), std::shared_ptr<Snitch>>::value == tru
```

We'll go into `make_shared` soon.

Performance & thread safety

`unique_ptr` has performance similar to a raw pointer (with compiler optimizations), and also the size of a raw pointer. This is not the case for `shared_ptr`.

`shared_ptr` must have at least 2 pointers, so it's bigger than a raw pointer. It also guarantees thread-safety for all methods **as long as each thread has its own copy**. Thread safety includes assigment, reference increment / decrement and all other operations. However, it does **not** mean that locks are acquired prior to calling any of `T`'s methods - only `shared_ptr`'s own methods are guaranteed to be thread-safe. In other words, if you want `T` to be used from multiple thread concurrently you will have to implement thread-safety yourself.

As always, thread safety comes at a price – performance. For most cases it would probably be minimal, by utilizing atomic operations, but it's not guaranteed to be atomic and even atomics are not free.

If you'd like to read more on performance of C++ smart pointers vs raw pointers check out this cool post by Davide Coppola.

`std::make_shared()`

Much like `make_unique()`, `make_shared()` saves us from using `new` directly, arguably prodoces cleaner code, and is exception safe. In addition to all of these, and unlike `make_unique()`, it also brings a performance advantage.

Performance? Yes, *performance*. `shared_ptr<T>` manages a reference count to know when to release `T`. This is done via a shared *control block* to which all `shared_ptr`s point. Therefore, it must be dynamically allocated. And of course there's also the object itself, `T`, which needs to be dynamically allocated as well.

So creating a new `shared_ptr` would create 2 objects, thus call `new` twice. However, `make_shared()` can make a single allocation for both, and thus save some load. Cool, right?

It's good to know that:

1. This can't possibly be done through `shared_ptr`'s constructor, as the constructor is called on an already allocated memory block and there's no `realloc()` equivalent in C++.
2. You may use `std::allocate_shared()` if you need a custom allocator.

Construct from `unique_ptr`

`shared_ptr` has a special constructor that accepts a `unique_ptr&&`. This is useful when working with factories that return a `unique_ptr`, but you want to assign the value to a `shared_ptr`:

```
std::unique_ptr<MyObject> CreateMyObject() {
  return std::make_unique<MyObject>();
}

int main() {
  std::shared_ptr<MyObject> shared_object = CreateMyObject();
}
```

If you're implementing a factory and don't know if your callers will assign the value to a `unique_ptr` or a `shared_ptr` - always return `unique_ptr`.

No `release()` method, `reset()` doesn't necessarily release

Unlike `unique_ptr`, `shared_ptr` does not have a `release()` method. It wouldn't make sense to implement such a method since there's oftentimes no way to determine *at compile time* how many `shared_ptr`s point to the same instance.

On the other hand, `reset()` exists, but it does not necessarily delete the underlying object. Here's an example:

```
#include <iostream>
#include <type_traits>
#include <memory>

struct Snitch {  // Same as above, no changes
public:
  Snitch() { std::cout << "c'tor" << std::endl; }
  ~Snitch() { std::cout << "d'tor" << std::endl; }
  Snitch(Snitch const&) { std::cout << "copy c'tor" << std::endl; }
  Snitch(Snitch&&) { std::cout << "move c'tor" << std::endl; }
};
```

```
int main() {
  std::cout << "Creating 1st Snitch" << std::endl;
  auto snitch1 = std::make_shared<Snitch>();
  auto snitch2 = snitch1;

  std::cout << "Calling reset" << std::endl;
  snitch1.reset();  // object will *not* be released

  std::cout << "Moving out of scope" << std::endl;
}
```

Output:

```
Creating 1st Snitch
c'tor
Calling reset
Moving out of scope
d'tor
```

Cyclic references & `std::weak_ptr`

`shared_ptr`s are almost perfect. Their one imperfection is that they don't support cycles. Example:

```
#include <iostream>
#include <type_traits>
#include <memory>

struct Node {  // Binary tree
  Node() { std::cout << "c'tor" << std::endl; }
  ~Node() { std::cout << "d'tor" << std::endl; }

  std::shared_ptr<Node> parent;
  std::shared_ptr<Node> left;
  std::shared_ptr<Node> right;
};

int main() {
  auto root = std::make_shared<Node>();
  root->left = std::make_shared<Node>();
  root->left->parent = root;
}
```

Output:

```
c'tor
c'tor
```

As you can see, no destructor has been called due to the vicious cycle I introduced, thus a memory leak occurred.

`weak_ptr` was created to allow us to have cycles that won't leak. A `weak_ptr` holds a non-owning pointer. Essentially it means that `weak_ptr` won't prevent its pointee from being released.

In the above example, simply modifying the parent declaration from

```
  std::shared_ptr<Node> parent;
```

To:

```
  std::weak_ptr<Node> parent;
```

Without changing anything else in the code, and we get the following output:

```
c'tor
c'tor
d'tor
d'tor
```

Magic, right? That's cool, however there are a few things we should know. The most important is that the object pointed by the `weak_ptr` can be released while the `weak_ptr` is still alive. That's pretty much the definition of a weak pointer.

Another thing is that `weak_ptr` has a very basic API, which **does not even include a `get()` method**. WAT? Yes. But, of course, it's not useless. In order to use the object pointed to by a `weak_ptr` one must *upgrade* it to a `shared_ptr` by calling `lock()` and checking if the returned `shared_ptr` is empty:

```
  std::weak_ptr<std::string> weak = // ...
  std::shared_ptr<std::string> shared = weak.lock();
  if (shared) {
    // object exists
  } else {
    // object has been released
  }
```

Once we have a `shared_ptr` in our hands the object no longer can be released, so that's a pretty smart and cool design decision.

One caveat of using `weak_ptr` is that while the *object* is released after the last `shared_ptr` is released, the *control block* remains alive until the last `weak_ptr` is released. If the control block and object are allocated together (see `make_shared` above) – this would mean that `weak_ptr` will cause the memory to remain alive (even though the object will be destroyed).

Control block

As previously mentioned, `shared_ptr`s are managed via a *control block*. These control blocks are up to the implementations to define, however they generally contain the following:

- The managed object (either a pointer or the object itself if created via `make_unique()`);
- Reference count (for both other `shared_ptr`s and `weak_ptr`s);
- Deletion function.

The control block is always accessed in a thread-safe way, either via atomics or a mutex.

Earlier I wrote that a `shared_ptr` has the size of 2 pointers, while here I descrive the control block as pointing to (or containing) the object. So what does `shared_ptr` need to point to, other than the control block? Read the next section to find out :)

Point to `A`, manage `B`

This may sound somewhat bizarre at first, so bear with me. Say you have an internal object, `B`. This `B` has a few fields, but one of them, `A`, is exposed externally. One possible such scenario is where `B` is a channel to a database including the internal socket etc, and `A` is the API object on which a user acts. Usually you would have these as private members, or hide them behind an interface. But, again, bear with me – suppose you have a good reason.

Now, you want to manage `B` in a shared way, but you only want to give users `A` what do you do? It turns out `shared_ptr` supports this via a feature called *aliasing*.

With aliasing one can create a `shared_ptr` from another `shared_ptr`, so that their control blocks are the same, but have the `get()` method return any arbitrary pointer, even one that has nothing to do with them.

Here's an example:

```
struct DatabaseConnection {}; // exposed to the user

struct InternalDatabaseConnection {
  // socket
  // authentication information
  DatabaseConnection connection;
};

std::shared_ptr<DatabaseConnection> CreateDatabaseConnection() {
  auto tmp = std::make_shared<InternalDatabaseConnection>();
  return std::shared_ptr<DatabaseConnection>(tmp, &tmp->connection);
}
```

Note that `delete` will never be called on `&tmp->connection` which is a `DatabaseConnection`, but rather only on `InternalDatabaseConnection` allocated by `make_shared`.

Casting

Like `unique_ptr`, `shared_ptr` also supports automatic cast from `shared_ptr<T>` to `shared_ptr<U>` if `T*` is convertible to `U*`.

Unlike `unique_ptr`, `shared_ptr` will always call the destructor it was constructed with, even when casting to a parent with no virtual destructor. Example:

```
#include <iostream>
#include <memory>

struct Base { ~Base() { std::cout << "non-virtual ~Base()" << std::er
struct Derived : Base { ~Derived() { std::cout << "~Derived()" << std

int main() {
        std::shared_ptr<Base> base = std::make_shared<Derived>();
}
```

Output:

```
~Derived()
non-virtual ~Base()
```

If we were to replace `shared_ptr` with `unique_ptr` (and `make_shared` with `make_unique`) the program would not call `~Derived`.

In addition to that, there are 4 utility functions to allow creating a `shared_ptr` when implicit conversion doesn't happen:

- `std::static_pointer_cast`
- `std::dynamic_pointer_cast`
- `std::const_pointer_cast`
- `std::reinterpret_pointer_cast`

Let's look at an example:

```
auto derived = std::make_shared<Derived>();
std::shared_ptr<Base> base = derived;  // OK.
//std::shared_ptr<Derived> derived2 = base;  // ERROR: no implicit do
std::shared_ptr<Derived> derived2 = std::static_pointer_cast<Derived>
std::shared_ptr<Derived> derived3 = std::dynamic_pointer_cast<Derived
```

Note that `T*` needs to be convertible to `U*`, which is different from `T` being convertible to `U`:

```
auto shared_short = std::make_shared<short>(123);
//std::shared_ptr<int> shared_int = shared_short;  // ERROR: no cast
```

`std::enable_shared_from_this`

This is somewhat odd and very specific, so one last time I need you to bear with me;

Suppose you're implementing a class `WeirdClass`. And suppose you know that this class will be managed by `shared_ptr`. And suppose that for some reason you would like to return a `shared_ptr` **to yourself** (yourself being an instance of `WeirdClass`). How would you do that? Let's consider the following:

```
#include <iostream>
#include <memory>

struct WeirdClass {
  std::shared_ptr<WeirdClass> CreateSharedPtrToThis() {
    return std::shared_ptr<WeirdClass>(this);  // DON'T DO THIS.
  }
};
```

```cpp
int main() {
  auto weird_class = std::make_shared<WeirdClass>();
  auto tmp = weird_class->CreateSharedPtrToThis();
}  // ERROR: double delete
```

This kind of error can generally be avoided by not calling `shared_ptr`'s constructor directly, but `std::make_shared` instead.

However, in this specific case the object is already allocated - we merely want to copy a `shared_ptr` that already exists, but is unknown in the context of `WeirdClass`. What do we do? This is exactly why `std::enable_shared_from_this` was invented:

```cpp
#include <iostream>
#include <memory>

struct WeirdClass : std::enable_shared_from_this<WeirdClass> {
  std::shared_ptr<WeirdClass> CreateSharedPtrToThis() {
    return shared_from_this();
  }
};

int main() {
  auto weird_class = std::make_shared<WeirdClass>();
  auto tmp = weird_class->CreateSharedPtrToThis();
}  // no problem!
```

But please, don't take this as a reason to use `enable_shared_from_this`. Some features are best not used :)

That's it for today

I hope you found this post useful. Please let me know if I missed anything, have an error somewhere, or if you have any question!

<< Exploring std::unique_ptr Using libclang to Parse C++ (aka libclang 101) >>

Comments

4 Comments ShaharMike.com Upadhyay Vikas

Recommend 4 Tweet Share Sort by Best

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

Join the discussion…

Marek Knápek • 3 years ago
Great article, you should also mention one disadvantage of `make_shared` that is: when you release all `shared_prt`s, but have at least one `weak_ptr` alive, the managed object `T` will be destroyed (by its destructor), but the memory it occupied cannot be deallocated, because we need to manage the weak count and the weak count was allocated together with `T`.

So, when using `weak_ptr` with large objects, consider avoid using `make_shared`.
2 ^ | ˅ • Reply • Share ›

shaharmike Mod → Marek Knápek • 3 years ago
That's a great point, thank you! (added.)
1 ^ | ˅ • Reply • Share ›

ayy lmao • 3 years ago
Really great blog you have here. Love these articles immensely
1 ^ | ˅ • Reply • Share ›

shaharmike Mod → ayy lmao • 3 years ago
Thanks! I'm very glad you find them interesting. BTW if you have a suggestion for a topic - I'm always looking :)
^ | ˅ • Reply • Share ›

ALSO ON **SHAHARMIKE.COM**

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=""

Exploring std::unique_ptr | Shahar Mike's Web Spot
2 comments • 3 years ago
Alan King — thanks - this was very helpful. I was lost inside boost smart pointer jungle, and this post helped me sort out a good

Exploring std::string | Shahar Mike's Web Spot

Compiling Clang from Scratch | Shahar Mike's Web Spot