Why is there a make_unique? Why not just overload the unique_ptr constructor?

devblogs.microsoft.com/oldnewthing/20221019-00

October 19, 2022

Raymond Chen

At first, there was no `make_unique`. Only `unique_ptr`. And for expository simplicity, let's focus just on the non-array version of `unique_ptr`.

There's the proposal for `make_unique`, written by our pal Stephan T. Lavavej. It cites a few motivating issues for the `make_unique` function:

1. Parallel construction with `make_shared`.
2. Avoiding the need to use the `new` operator explicitly, thereby permitting the simple rule: "Don't write `new`." Prior to `make_unique`, the rule was "Don't write `new`, except to construct a `unique_ptr`."
3. Avoiding having to say the type name twice: `std::unique_ptr<T>(new T(args))`.
4. Avoid a memory leak due to unspecified order of evaluation if a `std::unique_ptr` is constructed from a newly `new`'d pointer as part of a larger expression which could throw. More details here.

But couldn't we have solved this problem by adding a new constructor to `unique_ptr`?

```
template<typename T>
struct unique_ptr
{
    ...

    template<typename... Args>
    unique_ptr(Args&&... args) :
        unique_ptr(new T(std::forward<Args>(args)...)) {}
};
```

With this new overload, you can write

```
// was p = std::make_unique<Thing>(arg1, arg2, arg3);
auto p = std::unique_ptr<Thing>(arg1, arg2, arg3);
```

This seems convenient (avoids introducing a new name), but it still has problems. For example, consider this:

```
struct Node
{
    Node(Node* parent = nullptr);
};

auto create_child(Node* parent)
{
    // was return std::make_unique<Node>(parent);
    return std::unique_ptr<Node>(parent);
}
```

This version looks like it's create a new child node with the specified parent, but since the constructor parameter is a pointer to the same type, what this really does is create a `unique_ptr` that manages the parent pointer. Everything will compile, and it may even run for a while, inadvertently updating the wrong node, and eventually leading to a double-free bug.

And then there's the converse problem:

```
struct NodeSource
{
    operator Node*();
};

auto wrap_proxy(NodeSource const& source)
{
    // was return std::make_unique<Node>(source);
    return std::unique_ptr<Node>(source);
}
```

This time, we want to create a `unique_ptr` that manages the object produced by the `NodeSource`'s conversion operator. A common case where you encounter this is if the `NodeSource` is some sort of proxy object. But since the parameter is not literally a `Node*`, this gets picked up by the new overload and is interpreted as

```
    return std::unique_ptr<Node>(new Node(source));
```

For backward compatibility, both of these cases must resolve to the constructor that takes a raw pointer to a `Node`. That can probably be accomplished via a special overload that takes exactly one universal reference, and a little SFINAE, but it's starting to get complicated.

The default constructor has entered the chat:

```
auto make_something()
{
    // was return std::make_unique<Node>();
    return std::unique_ptr<Node>();
}
```

Does this create an empty `unique_ptr`? Or does it create a new default-constructed `Node` and then create a `unique_ptr` that manages it?

For backward compatibility, this must create an empty `unique_ptr`, so now you have a third special case where passing `Node` constructor parameters to `unique_ptr` doesn't actually construct a `Node`.

The move and copy constructors have entered the chat:

```
struct ListNode
{
    ListNode(std::unique_ptr<ListNode> rest);
};

auto prepend_node(std::unique_ptr<ListNode> rest)
{
    // was return std::unique_ptr<ListNode>(
    //     new ListNode(std::move(rest));
    return std::unique_ptr<ListNode>(std::move(rest));
}
```

Does this create a new `ListNode` object, using `rest` as the constructor parameter? Or does this move-construct an existing `std::unique_ptr`? Again, for backward compatibility, this must move-construct the `std::unique_ptr`.

Okay, so if you do some SFINAE magic and carve out the special cases for backward compatibility, you've resolved the *technical* ambiguity. But you've done nothing to address the *semantic* ambiguity.

```
contoso::table<Node*> nodes;
...
auto p = std::unique_ptr<Node>(nodes.get(i));
```

Does this get a `Node*` from the table and transfer ownership of it to a `unique_ptr`? Or does this get a `Node*` from the table and create a new `Node` from it?

As we noted earlier, compatibility requires that we interpret this as an ownership transfer, and if you want to create a new node, you have to do so explicitly:

```
auto p = std::unique_ptr<Node>(new Node(nodes.get(i));
```

What makes this even more confusing is that similar expressions represent the creation of a new `Node` without having to write out the `new`:

```
// new Node(Node*, bool)
auto p = std::unique_ptr<Node>(nodes.get(i), true);

// new Node(42)
auto p = std::unique_ptr<Node>(42);

// does not create a new Node (!)
auto p = std::unique_ptr<Node>(nodes.get(i));
```

In addition to the confusion over whether this is an ownership transfer or a creation, it is unforgiving of typos like

```
Node* n;

// This takes ownership of n
auto p = std::unique_ptr<Node>(n);

// This creates a new Node that is a copy of *n
auto p = std::unique_ptr<Node>(*n);
```

To avoid this pit of failure, we probably should use a tag type to indicate whether we are taking ownership or making a new object.

```
template<typename T>
struct unique_ptr
{
    ...

    template<typename... Args>
    unique_ptr(in_place_t, Args&&... args) :
        unique_ptr(new T(std::forward<Args>(args)...)) {}
};

Node* n;

// Take ownership of n
auto p = std::unique_ptr<Node>(n);

// Create a new Node with n as its parent
auto p = std::unique_ptr<Node>(std::in_place, n);

// Create an empty unique_ptr
auto p = std::unique_ptr<Node>();

// Create a new default Node and wrap it in a unique_ptr
auto p = std::unique_ptr<Node>(std::in_place);

// Move-construct a new unique_ptr from an existing one
std::unique_ptr<ListNode> rest = /* ... */;
auto q = std::unique_ptr<ListNode>(std::move(rest));

// Move-construct a new unique_ptr from an existing one
auto q = std::unique_ptr<ListNode>(std::in_place, std::move(rest));
```

At this point, the new overload seems much more hassle than it's worth. You may as well just factor the "make a new Node" feature into a separate function `make_unique`. This is more explicit that it makes a new Node, and it's less typing anyway.

```
// Take ownership of n
std::unique_ptr<Node> p(n);

// Create a new Node with n as its parent
auto p = std::make_unique<Node>(n);

// Create an empty unique_ptr
auto p = std::unique_ptr<Node>();

// Create a new default Node and wrap it in a unique_ptr
auto p = std::make_unique<Node>();

// Move-construct a new unique_ptr from an existing one
std::unique_ptr<ListNode> rest = /* ... */;
auto q = std::unique_ptr<ListNode>(std::move(rest));

// Move-construct a new unique_ptr from an existing one
auto q = std::make_unique<ListNode>(std::move(rest));
```

If you want to make a new object, use the `make_unique` function.

Raymond Chen

Follow