An Exploration of the Deleter on the Memory Footprint of `unique_ptr`

…or "How I Learned to Love the Tuple."

Matt Bisson — cyberbisson.com

April 2017

Although the C++ standard does not explicitly spell it out, one may hope for a sophisticated implementation of the `std::unique_ptr` that consumes exactly as much memory as the pointer it tracks when it is configured to use a empty-class deleter (such as the default deleter type). The `unique_ptr` indeed allows the consumer to supply a custom deleter, which one intuitively expects to consume memory—if you store two pieces of data, you will consume two spots in memory somehow, right? Not quite. I will explore how this is done (specifically in GCC).

1 A Naïve Implementation

Once upon a time, I wrote a handle container type, similar to `unique_ptr`. In fact, I called this `unique_handle` (you can see it here). Because I wanted to optimize for my most common case—a simple `CloseHandle()`—I optimized the container as suggested above. If the caller used the default deleter, no `unique_handle` instance should consume more memory than the handle itself.

I achieved this behavior with template specialization. The generic `unique_handle` class template had storage for the deleter alongside the handle itself. I then provided a specific specialization for `unique_handle` that was given the default deleter (specified as a default template parameter). Unfortunately, specializing a class requires duplication of various bits of code, so the `unique_handle` type had a base class for common code, and two user-visible implementation types, adding to the overall complexity.

Another downside to this approach is the fact that *only* the recognized deleter type resulted in a smaller footprint. Even using a deleter derived from the default one resulted in increased size.

2 What GCC Produces

Our wishes become reality, because I can show that `unique_ptr` *does* consume the minimum possible size in GCC, as I suggested a "good" implementation would. Consider the following test code:

```
// Here is the memory we will track with std::unique_ptr.
struct blob {
    int something = 0;
};

// This is a deleter that takes no space.
struct delete_blob_empty {
    void operator()(blob *const in) const noexcept {
        delete in;
    }
};

// This deleter has state. Essentially, should we zero out the pointer
// after deletion.
struct delete_blob {
    bool null_it = false;
    void operator()(blob *in) const noexcept {
        delete in;
        if (null_it) in = nullptr;
    }
};

int main()
{
    delete_blob my_deleter;
    delete_blob_empty my_empty_deleter;

    std::unique_ptr<blob> some_blob(new blob);
    std::unique_ptr<blob, delete_blob> custom_blob(new blob, my_deleter);
    std::unique_ptr<blob, delete_blob_empty> empty_custom_blob(
        new blob, my_empty_deleter);

    std::cout << "Sizes: some_blob = "    << sizeof(some_blob)
              << "; custom_blob = "       << sizeof(custom_blob)
              << "; empty_custom_blob = " << sizeof(empty_custom_blob)
              << '\n';

    std::cout << "Done!\n";
    return 0;
}
```

…which prints:

```
Sizes: some_blob = 8 custom_blob = 16 empty_custom_blob = 8
Done!
```

Great! The only `unique_ptr` that consumes 16 bytes (this is a 64-bit system, so the size of a pointer will be 8 bytes) is the one that contains a custom deleter that has state (`delete_blob`). This means that not only have they addressed the default deleter case, as I had in `unique_handle`, but they even detect custom deleters that don't consume space, and react accordingly.

3 How Is It Implemented?

How is GCC doing this? It certainly isn't in the same manner as my earlier suggestion. A quick search of the `libstdc++` headers (specifically, `bits/unique_ptr.h`) shows that there is no specialization based on the deleter. Actually the only specialization of `unique_ptr` is to support different behavior when `_Tp` (the pointer type) is a simple pointer, versus an array. These `unique_ptr` implementations are quite succinct, and do not delegate any responsibility to implementation types. Here's what we have:

```
/// 20.7.1.2 unique_ptr for single objects.
template <typename _Tp, typename _Dp = default_delete<_Tp> >
  class unique_ptr
```

…and:

```
/// 20.7.1.3 unique_ptr for array objects with a runtime length
// [unique.ptr.runtime]
// _GLIBCXX_RESOLVE_LIB_DEFECTS
// DR 740: omit specialization for array objects with a compile time length
template<typename _Tp, typename _Dp>
  class unique_ptr<_Tp[], _Dp>
```

3.1 Enter the `std::tuple`

The secret to all of this is how `unique_ptr` stores its data. Back in our naïve implementation, we would have simply defined two fields, one for the pointer data, and a second for the deleter. This is not what GCC does. In the declaration for `unique_ptr`:

```
typedef std::tuple<typename _Pointer::type, _Dp>  __tuple_type;
__tuple_type                                      _M_t;
```

The compiler will reduce this down ultimately to:

```
std::tuple<blob*, std::default_delete<blob>> _M_t; // For some_blob
std::tuple<blob*, delete_blob> _M_t;       // For custom_blob
std::tuple<blob*, delete_blob_empty> _M_t; // For empty_custom_blob
```

So this means the magic is actually somewhere in `std::tuple`. This implication is great, and it means that not only are the entire class of `unique_ptr` instances memory optimized to store only what's required, but *anything that uses `std::tuple` automatically strips away types that require no in-memory footprint, while still giving you access to their facilities.* I will restate this another way: we have asked to store two pieces of information, and we can still use all the functions of both pieces of data, but only one actually takes memory resources. A simple `struct`, or even `std::pair` cannot make this promise.

The `std::tuple` class template (found in the header file, `tuple`) consists of a recursively-defined variadic template that forwards to a type called `_Tuple_impl`. Most of `_Tuple_impl`'s purpose is to keep track of the position in the `tuple` of the current element, and to break off a type from the template parameter pack, handing it to `_Head_base`.

Finally, with `_Head_base` we see some template specialization. Here is the (empty) class template declaration.

```
template<std::size_t _Idx, typename _Head, bool _IsEmptyNotFinal>
  struct _Head_base;
```

Pay close attention to this third template parameter, `_IsEmptyNotFinal`. `_Head_base` has these two specializations:

```
template<std::size_t _Idx, typename _Head>
  struct _Head_base<_Idx, _Head, true>
  : public _Head
  {
    constexpr _Head_base()
    : _Head() { }

    constexpr _Head_base(const _Head& __h)
    : _Head(__h) { }

    // ... [MB] Interface omitted ...
  };

template<std::size_t _Idx, typename _Head>
  struct _Head_base<_Idx, _Head, false>
  {
    constexpr _Head_base()
    : _M_head_impl() { }

    constexpr _Head_base(const _Head& __h)
    : _M_head_impl(__h) { }

    // ... [MB] Interface omitted ...

    _Head _M_head_impl;
  };
```

Simply put, if we get an `_IsEmptyNotFinal` that is `true`, the class that's defined doesn't even have a data member. For completeness, the `std::tuple` implementation sets this parameter by passing the pointer type to the metafunction, `__empty_not_final` (which is a topic for another day) but it essentially boils down to `std::is_empty`:

```
template<typename _Tp>
  struct __is_empty_non_tuple : is_empty<_Tp> { };

// Using EBO for elements that are tuples causes ambiguous base errors.
template<typename _El0, typename... _El>
  struct __is_empty_non_tuple<tuple<_El0, _El...>> : false_type { };

// Use the Empty Base-class Optimization for empty, non-final types.
template<typename _Tp>
  using __empty_not_final
  = typename conditional<__is_final(_Tp), false_type,
                         __is_empty_non_tuple<_Tp>>::type;
```

Note: If I make `delete_blob_empty` a `final` class, the `tuple` again takes a full 16 bytes. This restriction exists because `_Head_base` derives from the passed-in template type, so accepting the `final` type would result in compilation errors.

3.2 In The Debugger

To get a concrete feel for how this looks in memory, let's examine it in the debugger. If I examine the `unique_ptr` instance (from the example code) named `some_blob`, I see this (I have manually pretty-printed GDB output a little more from the basic "`p /r`" output):

```
<std::_Tuple_impl<0ul, blob*, std::default_delete<blob> >> = {

    <std::_Tuple_impl<1ul, std::default_delete<blob> >> = {
        <std::_Head_base<1ul, std::default_delete<blob>, true>> = {
            <std::default_delete<blob>> = { },
        },
    },

    <std::_Head_base<0ul, blob*, false>> = { _M_head_impl = 0x614c20 },
};
```

Examining `custom_blob` looks like this:

```
<std::_Tuple_impl<0ul, blob*, delete_blob>> = {

    <std::_Tuple_impl<1ul, delete_blob>> = {
        <std::_Head_base<1ul, delete_blob, false>> = {
            _M_head_impl = { null_it = false }
        },
    },

    <std::_Head_base<0ul, blob*, false>> = { _M_head_impl = 0x614c40 },
}
```

Pay close attention to the `_Head_base`'s third parameter, the boolean value we know as `_IsEmptyNotFinal`. The only field that `std::tuple` places into memory is `_M_head_impl`, and we can see that a value of `true` elides the creation of this data in memory.

It is also worth noticing that `tuple` reverses the in-memory ordering of its data. This is because the implementation class defines the "tail" of the data-structure as its base class, stashing

the "head" into the derived class. As you likely already know, the layout of a class is such that the base classes exist first, and derived classes append any new fields past the end of the base class. For our `unique_ptr` use-case, this means that we need to be a bit careful:

- Sometimes memory ordering is important for performance reasons, so an empty deleter means that we must offset the memory from the beginning of our `unique_ptr` by that much more every time we access the contained pointer.
- If we do something really nasty that assumes that the pointer exists at offset `0` of the `unique_ptr`, this assumption will be woefully incorrect in the custom deleter case.

4 Conclusions

This examination has lead me to a number of interesting conclusions here.

It's obvious that we want the `std::unique_ptr` class to mimic the resource consumption and emitted code of a bald pointer as closely as possible, while adding useful functionality like automatic freeing of resources and exception safety. Now we know that GCC actually accomplishes this for us under most circumstances.

The vast majority of C++ applications will use the default deleter, but for those that require custom deleters, keep the following in mind:

- If at all possible, your custom deleter should be an "empty" class. In other words, provide a simple functor with no member data. This keeps `std::unique_ptr` memory-optimized.
- Do not mark your deleter as a `final` class. At least on GCC, this disables the optimization because of the design of `std::tuple`.

With respect to design principles, `std::unique_ptr` offers an interesting case-study in code that pushes certain design decisions into lower layers. Since the container type, `std::tuple` knows how to take any empty class, and optimize it to consume zero bytes in memory, this optimization can be carried to more than just `unique_ptr`. This results in less complicated code than what I had written using very targeted template specialization in `unique_handle`.

Lastly, we learned that `std::tuple` occupies memory in the reverse order of its contained types, and by extension, a `std::unique_ptr` with a stateful deleter cannot be type-punned to its pointer type, as this would corrupt the deleter's internal state.