Inside std::function, part 2: Storage optimization

devblogs.microsoft.com/oldnewthing/20200514-00

May 14, 2020

Raymond Chen

Last time, we looked at the basic idea behind `std::function`. For each callable type, it creates a custom `callable` wrapper class that knows how to invoke the instance of that type. Each `callable` derives from a common base interface, and the main `std::function` holds a pointer to that interface.

As I noted, the standard requires that if the callable is a plain function pointer or a `reference_wrapper`, then no additional allocations are permitted.

The way the optimization works is that the `function` also carries with it a small buffer of raw bytes. If the `callable` is small (as it will be for a function pointer or or a `reference_wrapper`), then the `callable` is stored directly in the buffer, avoiding an extra allocation. If the `callable` is large, then a separate allocation is done as before.[1]

The concept behind the optimization is simple, but the execution is rather complicated. Since the memory is now being partly managed by the class itself, we can't use a `unique_ptr` any more. Instead, we keep a raw pointer and do manual memory management.

Let's start with what we need in our `function`:

```
inline constexpr size_t storage_size =
  std::max(sizeof(callable<void(*)()>),
           sizeof(reference_wrapper<char>));

using storage_t = typename
           aligned_storage<storage_size>::type;

struct function
{
  callable_base* m_callable = nullptr;

  storage_t m_storage;

  void* storage() { return addressof(m_storage); }

  ...
};
```

In addition to the `callable_base` pointer, we also have some storage that we can use for small callables. In order to satisfy the standard, our storage needs to be large enough to hold a callable function pointer or a callable reference wrapper.

There's a bit of a trick here: Calculating the size of an arbitrary function pointer and an arbitrary reference wrapper. Fortunately, the standard requires that all function pointers have the same size,[2] due to the rule that says that you can `reinterpret_cast` among function pointers, and if you `reinterpret_cast` back to where you started, the result is the same. A `reference_wrapper` is a wrapper around a pointer, and the largest pointer is the `void*`, which the standard requires to be the same size as a `char*`.[3]

The idea is that the `m_callable` points either to a heap-allocated `callable<T>` (if it is large) or to a `callable<T>` constructed via placement new into the `m_storage` (if it is small).

It's a simple idea, but the implementation gets kind of messy, because the `callable<T>`'s `clone` method has to do different things depending on whether it is a large or small object.

```
struct function
{
  ...

  template<typename T>
  function(T&& t)
  {
    if constexpr (sizeof(*new callable(t)) <= storage_size) {
      m_callable = new(storage()) callable(t);
    } else {
      m_callable = new callable(t);
    }
  }

  ...
};
```

To create a `function`, we take the functor object and construct it either in-place (if it is small) or on the heap (if it is large).

```
struct function
{
  ...

  function(const function& other)
  {
    if (other.m_callable) {
      m_callable = other.m_callable.clone(m_storage);
    }
  }

  ...
};
```

To copy a `function`, we take the source's `m_callable` and ask it to clone itself, using our storage if it can.

```
struct function
{
  ...

  function(function&& other)
  {
    if (other.m_callable == other.storage()) {
      m_callable = other.m_callable.move_to(m_storage);
      exchange(other.m_callable, nullptr)->~callable();
    } else if (other.m_callable) {
      m_callable = exchange(other.m_callable, nullptr);
    }
  }

  ...
};
```

To move a `function`, there are three cases.

- If the callable is small (the `m_callable` points to its storage), then move it into the destination's storage and destruct the original, leaving the source empty.
- If the callable is large (the `m_callable` is non-null but does not point to its storage), then transfer the pointer into the new object and leave the source empty.
- If there is no callable (`m_callable` is `nullptr`), then leave both source and destination empty.

The order of operations in the `move_to` case is tricky. The `move_to` comes first, because we don't want to make any changes to the source if the operation fails. Once the move succeeds, we need to null out the `other.m_callable` before destructing it, so that an exception during destruction will not result in double-destruction later.

```
struct function
{
  ...

  ~function()
  {
    if (m_callable == storage()) {
      m_callable->~callable();
    } else {
      delete m_callable;
    }
  }

  ...
};
```

To destruct a `function`, there are again three cases, but two of them collapse.

- If the callable is small (the `m_callable` points to our storage), then destruct it in place.
- Otherwise the callable is large or null. We take advantage of the fact that `delete`-ing a null pointer is harmless.

The `operator()(int, char*)` is the same as before.

We need to make some adjustments to `callable_base`:

```
struct callable_base
{
  callable_base() = default;
  virtual ~callable_base() { }
  virtual bool invoke(int, char*) = 0;
  virtual callable_base* clone(storage_t& storage) = 0;
  virtual callable_base* move_to(storage_t& storage) = 0;
};
```

The `callable_base` has new `clone` and `move_to` methods.

```
template<typename T>
struct callable : callable_base
{
  T m_t;

  callable(T const& t) : m_t(t) {}
  callable(T&& t) : m_t(move(t)) {}


  bool invoke(int a, char* b) override
  {
    return m_t(a, b);
  }

  ...
};
```

The constructors and `invoke` method haven't changed.

```
template<typename T>
struct callable : callable_base
{
  ...

  callable_base* clone(storage_t& storage) override
  {
    if constexpr (sizeof(*this) <= sizeof(storage)) {
      return new(addressof(storage)) callable(m_t);
    } else {
      return new callable(m_t);
    }
  }

  callable_base* move_to(storage_t& storage) override
  {
    if constexpr (sizeof(*this) <= sizeof(storage)) {
      return new(addressof(storage)) callable(move(m_t));
    } else {
      return nullptr; // should not be called
    }
  }
};
```

The `clone` and `move_to` methods use the provided storage if it is large enough. Otherwise, the `clone` allocates the cloned object on the heap. (On the other hand `move_to` should never be called for heap-allocated objects, for in those cases, we move the pointer to the callable instead of the callable itself.)

I think that's the necessary changes to keep track of the memory bookkeeping for our miniature `function`. I left out a bunch of methods that are required by the standard, particularly the assignment operators. I'm not even going to leave them as an exercise,

because you should just be using the `std::function` implementation that came with your compiler.

The purpose of this entry was to give you an idea of what's happening under the hood. This will help you when you need to debug one of these things, and the techniques may come in handy later. Like next time, for instance. Stay tuned.

[1] Implementations typically do some tuning to decide how big this small buffer should be. You want the buffer to be small, so that the function object consumes less memory. But you want the buffer to be large enough to hold the most common callables. (Plus, of course, the two callables required by the standard.)

[2] Note that the same requirement does not apply to pointers to member functions.

[3] Technically, I guess, you could have an implementation where function pointers were different sizes, but where `reinterpret_cast` applies appropriate conversions to "squeeze out the air" from large function pointers so they can be recovered from small function pointers;[4] or an implementation where `reference_ wrapper` did something weird based on the type. But since `std::function` is part of the implementation, it can make these sorts of implementation-dependent assumptions.

[4] For example, you might have "fast function pointers" which are fat (say, because they consist of a code pointer plus a table of contents) and "slow function pointers" which are smaller but slower (consisting of just the code pointer). The idea is that the bytes immediately before the code pointer hold the table of contents. Calling through a fast function pointer consists of loading the table of contents into the global pointer register, and calling the code pointer.

```
; assume fast function pointer is in r34/r35
mov   gp, r34                // set up global pointer
mov   b6, r35                // set up branch target
br.call.dptk.many rp = b6 ;;  // call it
```

Calling through a slow function pointer requires the call site to recover the table of contents from memory before calling the target.

```
; assume slow function pointer is in r34
add   r30 = r34, -8          // get address of TOC
mov   b6, r34 ;;             // set up branch target
mov   gp, [r30]              // set up global pointer
br.call.dptk.many rp = b6 ;;  // call the target
```

The slow version costs a memory access and takes an extra cycle. It may also burn an extra cache line, because we have to access the data that comes before the start of the code, and the code may have to satisfy certain alignment requirements which cause it tend to appear at the start of a cache line.

To convert a fast function pointer to a slow one, discard the table of contents pointer. To convert a slow function pointer to a fast one, fetch the table of contents pointer from the memory immediately before the code address.

Raymond Chen

Follow