Inside std::function, part 1: The basic idea

devblogs.microsoft.com/oldnewthing/20200513-00

May 13, 2020

Raymond Chen

The C++ language standard library comes with a `std::function` template type which represents a "thing you can invoke". It can hold any callable, such as

- Function pointer.
- Lambda.
- Other object with `operator()`.

The way this is done is with the assistance of a polymorphic helper object that understands the specific callable it is wrapping.

Here's a sketch. For concreteness, let's say we're implementing `std::function<bool(int, char*)>`. For readability, I've de-uglified[1] the identifiers.

```
struct callable_base
{
  callable_base() = default;
  virtual ~callable_base() { }
  virtual bool invoke(int, char*) = 0;
  virtual unique_ptr<callable_base> clone() = 0;
};

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);
  }

  unique_ptr<callable_base> clone() override
  {
    return make_unique<callable>(m_t);
  }
};

struct function
{
  std::unique_ptr<callable_base> m_callable;

  template<typename T>
  function(T&& t) :
    m_callable(new callable<decay_t<T>>
                (forward<T>(t)))
  {
  }

  function(const function& other) :
    m_callable(other.m_callable ?
               other.m_callable->clone() : nullptr)
  {
  }

  function(function&& other) = default;

  bool operator()(int a, char* b)
  {
    // TODO: bad_function_call exception
    return m_callable->invoke(a, b);
  }
};
```

The idea is that each `function` has a `callable_base`, which is an interface that allows us to perform basic operations on callable objects: Create a copy, invoke it, and destroy it. Invoking the `function` forwards the invoke to the `callable_base`. Copying the `function` requires a special `clone` method on the `callable_base`, because `unique_ptr` is not copyable.

Constructing the `function` is a matter of creating a custom `callable` for the specific functor. It's conceptually simple, but the C++ language makes us write out a bunch of stuff to get it to work. We just want a callable that wraps the thing that was passed to the constructor.

The `std::function` in the standard library is basically like this, but with additional optimizations to avoid an allocation in the case of a small `callable`. Said optimizations are in fact mandatory by the standard if the callable is a plain function pointer or a `reference_wrapper`.

We'll look at that optimization next time, because it gives us some insight into how we can do similar things with our own types.

[1] Uglification is the process of taking readable names and transforming them into names that are reserved for the implemenmtation. Different libraries have different uglification conventions. For the Microsoft Visual C++ implementation of the standard library, the uglifications tend to be

- `_My` prefix for member variables.
- `_Ty` prefix for type names.
- `_Fn` prefix for functors.
- `_P` prefix for pointers.
- `_` (and capital first letter) for most other things.

Raymond Chen

Follow