Storing a non-capturing lambda in a generic object

devblogs.microsoft.com/oldnewthing/20200515-00

May 15, 2020

Raymond Chen

Suppose you want an object that can store a non-capturing lambda. You don't want to use `std::function` because it's too heavy, for some sense of "heavy". Perhaps you don't like the fact that its copy constructor can throw. Or that it sometimes requires two allocations. Or that there's a virtual function call.

Fortunately, non-capturing lambdas are convertible to function pointers, so really you are just storing function pointers. Let's say that the lambda is going to be called with a single integer parameter.

```
struct event_source
{
  using handler_t = bool (*)(int);

  handler_t m_handler = nullptr;

  void set_handler(handler_t handler)
  {
    m_handler = handler;
  }

  void raise(int value)
  {
    if (m_handler) m_handler(value);
  }
};
```

Registering a handler with a captureless lambda is straightforward thanks to the implicit conversion to a function pointer:

```
event_source e;

void f()
{
  e.set_handler([](int value) { log(value); return true; });
}

void g()
{
  e.raise(42);
}
```

A plain function pointer isn't that great because it doesn't have any context. Let's provide a context parameter so the function can remember its environment.

```
struct event_source
{
  using handler_t = bool (*)(int, void*);

  const void* m_context;
  handler_t m_handler = nullptr;

  void set_handler(handler_t handler, const void* context)
  {
    m_handler = handler;
    m_context = context;
  }

  void raise(int value)
  {
    if (m_handler) {
      m_handler(value, const_cast<void*>(m_context));
    }
  }
};
```

So far so good, but one frustration is that the handler has to cast the `context` parameter back to whatever it originally was:

```
void f(widget* widget)
{
  e.set_handler([](int value, void* context)
    {
      auto widget = reinterpret_cast<widget*>(context);
      ... use the widget and value ...
      return true;
    }, widget);
}
```

Wouldn't it be nice if we could automatically pass the context back through in the same form it was received?

```
struct event_source
{
  template<typename T = void>
  using handler_t = bool (*)(int, T*);

  const void* m_context;
  handler_t<> m_handler = nullptr;
  bool (*m_adapter)(handler_t<>, int, const void*);

  template<typename T>
  void set_handler(handler_t<T> handler, T* context)
  {
    m_handler = reinterpret_cast<handler_t&lt>>(handler);
    m_context = context;
    m_adapter = [](handler_t&lt> raw_handler,
                   int value, const void* raw_context)
    {
      auto handler = reinterpret_cast<handler_t<T>>(raw_handler);
      auto context = reinterpret_cast<T*>(raw_context);
      return handler(value, context);
    };
  }

  void raise(int value)
  {
    if (m_handler) {
      m_adapter(m_handler, value, m_context);
    }
  }
};
```

We are basically mimicking what `std::function` does, but taking advantage of optimizations that are available because of our special restrictions.

First of all, we know that the handler is just a function pointer, so we don't need variable-sized storage. As we noted earlier, C++ requires that a function pointer can be cast to any other kind of function pointer, and then recovered by casting back. So we'll just use `handler_t<>` as our generic storage.

Second, function pointers are scalar, hence trivial, so they can be copied and moved by `memmove` and require no special construction or destruction. This means that the only remaining virtual method of our `callable_base` is `invoke`.

Since there is only one virtual method, we can dispense with the vtable and just record the function pointer directly. This avoids a level of indirection.

The function pointer we record is the "adapter" which takes the raw storage and raw context, casts them back to the original function pointer type and context type, and then calls the original function pointer with the original context.

On modern architectures, the “adapter” function is effectively a nop because all pointers are ABI-compatible, but the C++ language is not as permissive as the ABI, so we need the adapter to keep everything honest. In practice, every type will have the same adapter, and the adapter itself will be trivial.

If you know that the calling convention for your ABI is register-based, you can do some micro-optimizing by moving the `handler` parameter to the end of the parameter list. That makes the adapter a single *jump to register* instruction.

Let’s take this out for a spin:

```
void f()
{
  e.set_handler([](int value, const char* context)
    { log(context, value); return true; }, "hello");
}
```

This fails to compile:

```
error: no matching member function for call to 'set_handler'
note: candidate template ignored: could not match 'handler_t<T>' against '(lambda)'
```

Although there is a conversion from the captureless lambda to `bool (*)(int, const char*)`, the conversion is not considered by the template matching machinery.

We’ll have to help it along.

```
  template<typename TLambda, typename T>
  void set_handler(TLambda&& handler, T* context)
  {
    m_handler = reinterpret_cast<handler<>>(
                  static_cast<handler_t<T>>(handler));
    m_context = context;
    m_adapter = [](handler_t&lt> raw_handler,
                   int value, const void* raw_context)
    {
      auto handler = reinterpret_cast<handler_t<T>>(raw_handler);
      auto context = reinterpret_cast<T*>(raw_context);
      return handler(value, context);
    };
  }
```

Our `set_handler` accepts anything as its first parameter, but then immediately `static_cast`s it to a function accepting a matching context.

This design makes the final parameter (the `context`) the final arbiter of the signature of the lambda. This is troublesome if the context is `nullptr`, because the type of `nullptr` is `nullptr_t`, which is not a pointer to anything. You’ll have to cast the `nullptr_t` explicitly to the desired context type.

```
void f()
{
  e.set_handler([](int value, const char* context)
    { log(context, value); return true; },
    (const char*)nullptr);
}
```

So there you have it. A “lighter” version of `std::function` that accepts only function pointers and things convertible to function pointers (which includes captureless lambdas), plus a context parameter, which is basically a special case of `std::bind`.

Raymond Chen

Follow