Programming puzzle: Creating a map of command handlers given only the function pointer

devblogs.microsoft.com/oldnewthing/20190527-00

May 27, 2019

Raymond Chen

Suppose you have some sort of communication protocol that sends packets of binary-encoded information. There is some information at the start of the packet that describe what command is being sent, and the rest of the bytes in the packet describes the parameters to the command.

The puzzle is to come up with a generic dispatcher that accepts command / handler pairs and does the work of extracting the command parameters from the packet and calling the handler.

You are given this class:

```
class Packet
{
public:
 int32_t ReadInt32();
 uint32_t ReadUInt32();
 int8_t ReadInt8();
 std::string ReadString();
 ... and so on ...
};
```

The `Read` methods parse the next bytes in the packet and produces a corresponding object. Sometimes the object is simple, like an integer. Sometimes it's complicated, like a string. Don't worry about the details of the parsing; the `Packet` object will do it.

The puzzle is to implement the `Dispatcher` class:

```
class Dispatcher
{
public:
  void AddHandler(uint32_t command, ??? SOMETHING ???);
  void DispatchCommand(uint32_t command, Packet& packet);
};
```

The intended usage is like this:

```
// Some handler functions
void HandleFoo(int32_t, int32_t);
void HandleBar(int32_t);
void HandleBaz(int32_t, std::string);

// Command 0 is the "Foo" command that takes
// two 32-bit integers.
dispatcher.AddHandler(0, HandleFoo);

// Command 1 is the "Bar" command that takes
// one 32-bit integer.
dispatcher.AddHandler(1, HandleBar);

// Command 4 is the "Baz" command that takes
// a 32-bit integer and a string.
dispatcher.AddHandler(4, HandleBaz);

// We received a packet. Dispatch it to a handler.
dispatcher.DispatchCommand(command, packet);
```

The `DispatchCommand` method looks up the `commandId` and executes the corresponding handler. In this case, the effect would be as if the `DispatchCommand` were written like this:

```
void DispatchCommand(uint32_t command, Packet& packet)
{
 switch (command) {
 case 0:
  {
   auto param1 = packet.ReadInt32();
   auto param2 = packet.ReadInt32();
   HandleFoo(param1, param2);
   break;
  }
 case 1:
  {
   auto param1 = packet.ReadInt32();
   HandleBar(param1);
   break;
  }
 case 4:
  {
   auto param1 = packet.ReadInt32();
   auto param2 = packet.ReadString();
   HandleFoo(param1, param2);
   break;
  }

 default: std::terminate();
 }
}
```

For the purpose of the puzzle, we won't worry too much about the case where an invalid command is received. The puzzle is really about the dispatching of valid commands.

Okay, let's roll up our sleeves. One way to attack this problem is to do it in a way similar to how we implemented message crackers for Windows messages: Write a custom dispatcher for each function signature.

```
class Dispatcher
{
 std::map<uint32_t, std::function<void(Packet&)>> commandMap;

public:
 void AddHandler(uint32_t command, void (*func)(int32_t, int32_t))
 {
  commandMap.emplace(command, [func](Packet& packet) {
   auto param1 = packet.ReadInt32();
   auto param2 = packet.ReadInt32();
   func(param1, param2);
  });
 }

 void AddHandler(uint32_t command, void (*func)(int32_t))
 {
  commandMap.emplace(command, [func](Packet& packet) {
   auto param1 = packet.ReadInt32();
   func(param1);
  });
 }

 void AddHandler(uint32_t command, void (*func)(int32_t, std::string))
 {
  commandMap.emplace(command, [func](Packet& packet) {
   auto param1 = packet.ReadInt32();
   auto param2 = packet.ReadString();
   func(param1, param2);
  });
 }

 ... and so on ...

 void DispatchCommand(uint32_t command, Packet& packet)
 {
  auto it = commandMap.find(command);
  if (it == commandMap.end()) std::terminate();
  it->second(packet);
 }
};
```

We write a version of `AddHandler` for each function signature we care about, and adding a handler consists of creating a lambda which which extracts the relevant parameters from the packet and then calls the handler. These lambdas are captured into a `std::function` and

saved in the map for future lookup.

The problem with this technique is that it's tedious writing all the lambdas, and the `Dispatcher` class needs to know up front all of the possible function signatures, so it can had an appropriate `AddHandler` overload. What would be better is if the compiler could write the lambdas automatically based on the parameters to the function. This avoids having to write out all the lambdas, and it means that the `Dispatcher` can handle arbitrary function signatures, not just the ones that were hard-coded into it.

First, we write some helper functions so we can invoke the `Read` methods more template-y-like.

```
template<typename T> T Read(Packet& packet) = delete;

template<> int32_t Read<int32_t>(Packet& packet)
    { return packet.ReadInt32(); }

template<> uint32_t Read<uint32_t>(Packet& packet)
    { return packet.ReadUInt32(); }

template<> int8_t Read<int8_t>(Packet& packet)
    { return packet.ReadInt8(); }

template<> std::string Read<std::string>(Packet& packet)
    { return packet.ReadString(); }

... and so on ...
```

If somebody needs to read a different kind of thing from a packet, they can add their own specialization of the `Read` function template. They don't need to come back to you to ask you to change your `Dispatcher` class.

Now the hard part: Autogenerating the lambdas.

We want a local variable for each parameter. The template parameter pack syntax doesn't let us create a variable number of variables, but we can fake it by putting all the variables into a tuple.

```
template <typename... Args>
void AddHandler(uint32_t command, void(*func)(Args...))
{
  commandMap.emplace(command, [func](Packet& packet) {
    auto args = std::make_tuple(Read<Args>(packet)...);
    std::apply(func, args);
  };
}
```

The idea here is that we create a tuple, each of whose components is the next parameter read from the packet. The templatized `Read` method extracts the parameter from the packet. We take all those parameters, bundle them up into a tuple, and then `std::apply` the function to the tuple, which calls the function with the tuple as arguments.

Unfortunately, this doesn't work because it relies on left-to-right order of evaluation of parameters, which C++ does not guarantee. (And in practice, it often isn't.)

We need to build up the tuple one component at a time.

```
template<typename First, typename... Rest>
std::tuple<First, Rest...>
read_tuple(Packet& packet)
{
  auto first = std::make_tuple(Read<First>(packet));
  return std::tuple_cat(first, read_tuple<Rest>(packet));
}

std::tuple<> read_tuple(Packet& packet)
{
  return std::tuple<>();
}

template <typename... Args>
void AddHandler(uint32_t command, void(*func)(Args...))
{
  commandMap.emplace(command, [func](Packet& packet) {
    auto args = read_tuple(packet);
    std::apply(func, args);
  };
}
```

We use the standard template metaprogramming technique of employing recursion to process each template parameter one at a time. You must resist the temptation to simplify

```
  auto first = std::make_tuple(Read<First>(packet));
  return std::tuple_cat(first, read_tuple<Rest>(packet));
```

to

```
  return std::tuple_cat(std::make_tuple(Read<First>(packet)),
                        read_tuple<Rest>(packet));
```

because that reintroduces the order-of-evaluation problem the `read_tuple` function was intended to solve!

The attempted solution doesn't compile because you can't do this sort of recursive template stuff with functions. (I'm not sure why.) So we'll have to wrap it inside a templatized helper class.

```
template<typename... Args>
struct tuple_reader;

template<>
struct tuple_reader<>
{
  static std::tuple<> read(Packet&) { return {}; }
};

template<typename First, typename... Rest>
struct tuple_reader<First, Rest...>
{
  static std::tuple<First, Rest...> read(Packet& packet)
  {
    auto first = std::make_tuple(Read<First>(packet));
    return std::tuple_cat(first,
                tuple_reader<Rest...>::read(packet));
  }
};

template <typename... Args>
void AddHandler(uint32_t command, void(*func)(Args...))
{
  commandMap.emplace(command, [func](Packet& packet) {
    auto args = tuple_reader<Args...>::read(packet);
    std::apply(func, args);
  };
}
```

We start by defining our `tuple_reader` helper template class as one with a variable number of template parameters.

Next comes the base case: There are no parameters at all. In that case, we return an empty tuple.

Otherwise, we have the recursive case: We peel off the first template parameter and use it to `Read` the corresponding actual parameter from the packet. Then we recursively call ourselves to read the remaining parameters from the packet. And finally, we combine our actual parameter with the tuple produced by the remaining parameters, resulting in the complete tuple.

The `std::tuple_cat` function requires tuples, so we take our first parameter and put it in a one-element tuple, so that we can concatenate the second tuple to it.

Now I'm going to pull a sneaky trick and combine the forward declaration with the recursion base case:

```
// Delete
//
// template<typename... Args>
// struct tuple_reader;
//
// template<>
// struct tuple_reader<>
// {
//   static std::tuple<> read(Packet&) { return {}; }
// };

template<typename... Args>
struct tuple_reader
{
  static std::tuple<> read(Packet&) { return {}; }
};
```

This trick works because the only thing that will match the template instantiation is the zero-parameter case. If there is one or more parameter, then the `First, Rest...` version will be the one chosen by the compiler.

We're almost there. If one of the parameters is non-copyable, the above solution won't work because the `first` is passed by copy to `std::tuple_ cat`, and the `args` is passed by copy to `std::apply`.

Even if the parameters are all copyable, the `std::move` is helpful because it avoids unnecessary copies. For example, if a very large string was passed in the packet, we don't want to make a copy of the large string just so we can pass it to the handler function. We just want to let the handler function use the string we already read.

To fix that, we do some judicious `std::move`ing.

```
template<typename... Args>
struct tuple_reader
{
  static std::tuple<> read(Packet&) { return {}; }
};

template<typename First, typename... Rest>
struct tuple_reader<First, Rest...>
{
  static std::tuple<First, Rest...> read(Packet& packet)
  {
    auto first = std::make_tuple(Read<First>(packet));
    return std::tuple_cat(std::move(first), // moved
                tuple_reader<Rest...>::read(packet));
  }
};

template <typename... Args>
void AddHandler(uint32_t command, void(*func)(Args...))
{
  commandMap.emplace(command, [func](Packet& packet) {
    auto args = tuple_reader<Args...>::read(packet);
    std::apply(func, std::move(args)); // moved
  };
}
```

The `AddHandler` method could be condensed slightly, which also saves us the trouble of having to `std::move` the tuple explicitly.

```
    std::apply(func, tuple_reader<Args...>::read(packet));
```

Exercise 1: Change the `tuple_ reader` so it evaluates the template parameters from right to left.

Exercise 2: Suppose the `Packet` has methods for sending a response to the caller. In that case, the handler should receive a `Packet&` as its first parameter, before the other optional parameters. Extend the above solution to support that.

Exercise 3: (Harder.) Extend the above solution to support passing an arbitrary function object as a handler, such as a lambda or `std::function`.

Raymond Chen

Follow