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

Raymond Chen

May 27th, 2019 13 0

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

Tagged Code

Read next

Why does my C++/WinRT project get errors of the form "unresolved external symbol … consume_Something"?

Declared but not defined, but what exactly wasn't defined?

Raymond Chen
May 29, 2019

5 comments

Why does my C++/WinRT project get errors of the form "consume_Something: function that returns 'auto' cannot be used before it is defined"?

Narrowing down the source of the missing header file.

Raymond Chen
May 30, 2019

5 comments

13 comments

Comments are closed.

Paul Topping May 27, 2019 8:23 am

I love this example! Raymond Chen, you're a smart fellow!
Here's a thought that I often have when reading this kind of C++ code. Is there some language that makes this kind of code easier to read and write? Asking this another way, is there a language where this kind of thing is so easy that a post like this is unnecessary? And finally, what would modern C++ look like if backward compatibility was ignored during its development?

Comments are closed.

Malle, Joseph May 27, 2019 9:52 am

Does this also work:

template <typename... Args>void AddHandler(uint32_t command, void(*func)(Args...)){   std::apply(func, std::tuple<Args...>({Read<Args>()...}));}

Comments are closed.

Роман Донченко May 27, 2019 11:18 am

No, because it runs into the evaluation order problem. (And if it didn't, the tuple wouldn't even be necessary.)

Comments are closed.

Malle, Joseph May 27, 2019 1:49 pm

Note the curly braces. And from the standard: Within the initializer-list of a braced-init-list, the initializer-clauses, including any that result from pack expansions (14.5.3), are evaluated in the order in which they appear. That is, every value computation and side effect associated with a given initializer-clause is sequenced before every value computation and side effect associated with any initializer-clause that follows it in the comma-separated list of the initializer-list. [Note: This evaluation ordering holds regardless of the semantics of the initialization; for example, it applies when the elements of the initializer-list are interpreted as arguments of a constructor call, even though ordinarily there are no sequencing constraints on the arguments of a call. —end note]

Comments are closed.

Daniel Grunwald May 27, 2019 1:55 pm

In C++17, you can use class template argument deduction to avoid `std::make_tuple` and instead use braced initialization for `std::tuple`:

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

Unlike function call syntax, the direct-list-initialization syntax guarantees left-to-right evaluation; so this way you can side-step the whole need for recursive templates. At least according to the C++ standard.
Unfortunately in the real world, some compilers haven't gotten the memo that it is the call syntax, not the call semantics (constructor call or aggregate initialization) that should matter for the evaluation order guarantee.
This stackoverflow question indicates there were compiler bugs in both GCC and MSVC as of 2016. I don't know if these problems have been fixed since.

Comments are closed.

Noam Raz May 27, 2019 2:18 pm

I'm pretty sure that at least the MSVC bug has been fixed, since MSVC's standard library actually uses braced-init-lists to guarantee order of evaluation in its internal _For_each_tuple_element.

Comments are closed.

Henry Skoglund May 27, 2019 6:05 pm

Hi, one way to get around that pesky evaluation order/recursion workarund, is to let those **Read<...>** helper template functions not return the read data from the packet, but instead return a lambda, which reads the data when called. And then construct the tuple from the lambdas (not from any data). The lambdas can then be invoked (to get the data from the packet) either in the AddHandler's lambda or even in the handler functions themselves.

Comments are closed.

George Gonzalez May 28, 2019 6:44 am

That's swell,, but it reminds me of my early programming days, when I would try to write a text editor in VMS command language. Or a screen-oriented editor in TECO. Or egads, autoconf in bash. They're all DOABLE, but in no sense understandable or maintainable. Please think of the poor intern that will be asked to add a new command handler. Hooray for being able to craft such a thing, but booo if you intend it to be used in production code and maintained by anyone with an IQ under 145. Like, most of us.

Comments are closed.

Pierre Baillargeon May 28, 2019 7:26 am

I t seems to me that adding a level of indirection by providing a templated argument reader lambda for each command that takes as the first parameter the lambda to call to handle the command result in multiple benefits:
1. The param reader enforce the correct type to be read. Important because there are multipl eimplicit conversion, so a slightly mistyped command handler lambda might successfully be passed and yet read slightly incorrect data from the stream.
2. The code is much clearer and simpler for mere mortal.
3. A clear separation of param reading and command handling.
The AddHandler() function becomes templated to take these intermediary param reading lambdas.

Comments are closed.

Henry Skoglund May 28, 2019 1:39 pm

"All problems in computer science can be solved by another level of indirection" quote from https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

Comments are closed.

James Touton May 28, 2019 2:39 pm

You absolutely can do this recursive template stuff with function templates, it's just harder because you can't partially specialize function templates. However:
return std::tuple_cat(first, read_tuple<Rest>(packet));This part fails mainly because Rest is not expanded; it's missing a trailing ellipsis. Similarly, in AddHandler, the call to read_tuple is missing a template argument list. It should be:
auto args = read_tuple<Args...>(packet);You have another problem in that the zero-argument overload of read_tuple is not a specialization of a template, which means it can't be invoked with angle brackets. This can be dealt with, but partial specialization via a class template is more straightforward.
All that said, I think there's still a piece missing to solve the problem. Presumably, each command has a particular binary format; that format is tied to the command id, not the handler. That is, I would like to get a compile-time failure if I try to pass a handler that is not appropriate to the command. You can set this up in a generic fashion by changing the command id from a run-time function argument to a compile-time template argument. From there, you can set up a trait that maps the command id to a handler type:
template <uint32_t command> struct handler_type;
template <uint32_t command> using handler_type_t = typename handler_type::type;
template <> struct handler_type<0> { using type = void (int32_t, int32_t); };
// others

AddHandler becomes:
template <uint32_t command> void AddHandler(handler_type_t<command>* func);
or
template <uint32_t command> void AddHandler(std::function<handler_type_t<command>> func);

Users can supply their own mappings by specializing handler_type for their needs; if you want to allow that, then both the dispatcher class and the trait should be specialized on a tag type that is unique to the consumer. From there, the rest of this blog post kicks in to handle the actual dispatch.

Comments are closed.

Raymond Chen May 28, 2019 5:01 pm

Sorry I didn't make it clear. The idea is that this class is a generic "packet deserializer" class that doesn't know the command IDs. (After all, if it did know, then we wouldn't have needed to do all of this in the first place. We could just have hard-coded all the legal command signatures.) There's presumably a corresponding "packet serializer" class, and clients who want to communicate with each other would set up a "packet serializer" on one side and a "packet deserializer" on the other side, using a set of commands specific to their usage case.

Comments are closed.

James Touton May 29, 2019 1:29 pm

Ugh; my reply above looks like a hideous wall of text. Thanks for taking the time to read through that! Any chance of (perhaps limited) markdown support coming to blog comments? It would be nice if replies to a coding blog could actually have readable code…
Re: generic deserializing, that's why I brought up the idea of allowing the client code to specialize the trait to provide (and enforce) their own mappings.

Comments are closed.

Archive

August 2022
July 2022
June 2022
May 2022
April 2022
March 2022
February 2022
January 2022
December 2021
November 2021
October 2021

Relevant Links

I wrote a book
Ground rules
Disclaimers and such
My necktie's Twitter

Categories

Code

History

Tips/Support

Other

Non-Computer