+ раскладываем данные из абстрактной структуры в параметры функции

Fun with type erasure

+ dispatching data from abstract structure to function parameters

Pavel Novikov

@cpp_ape

R&D Align Technology

What is type erasure?

```
std::any typeErasedObj;
```

holds an object of any type

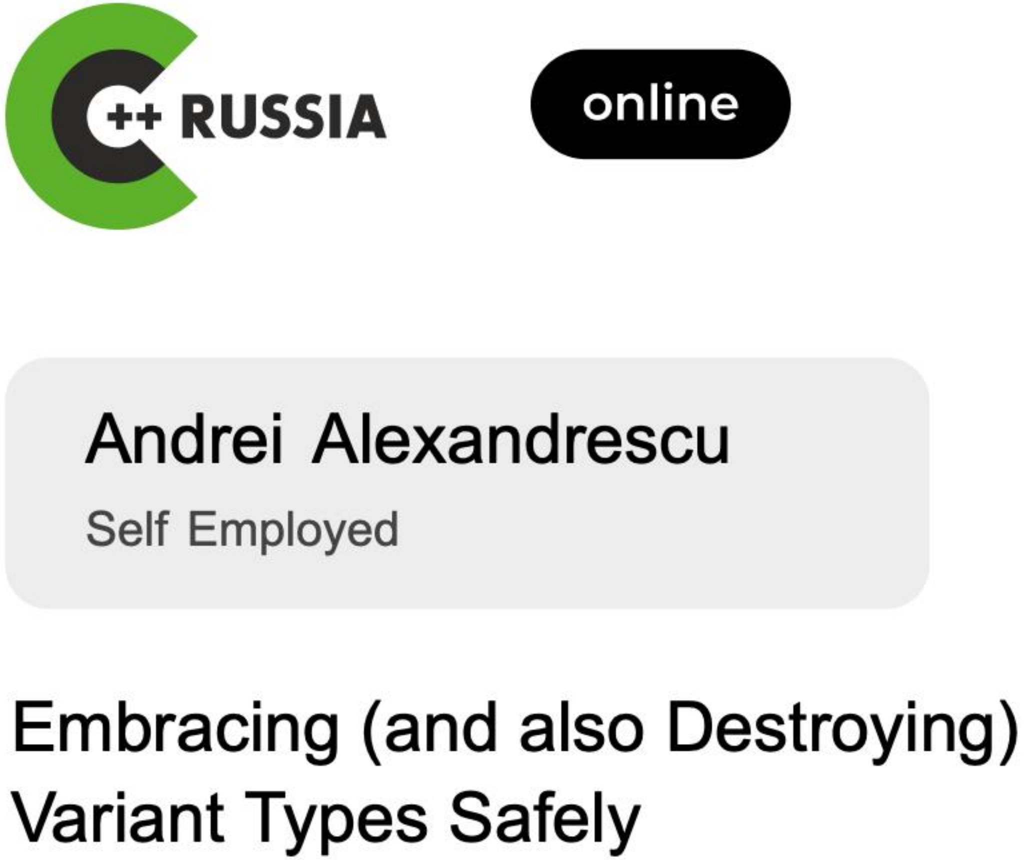
```
struct MyType {};
typeErasedObj = MyType{};
```

put an object into instance

```
MyType &object = std::any_cast<MyType&>(typeErasedObj);
```

retrieve the object
(if you "guess" the type right)

What is type erasure?



C++ RUSSIA

online

Andrei Alexandrescu

Self Employed

Embracing (and also Destroying) Variant Types Safely

What is type erasure?

```cpp
void foo() { /*...*/ }

std::function<void()> f = &foo;
```

wraps pointer to function

```cpp
const auto value = getValue();
f = [value]() {/*...*/};
```

or closure object

```cpp
struct Functor {
  void operator()() { /*...*/ }
};
f = Functor{};
```

or functor object

Basic idea

message:

```
{
  "request": "baz",
  "count": 1,
  "id": "two",
  "payload": { "three": 3 }
}
```

```
public class Program
{
  public static void foo()
  { }
  public static void bar([Name("count")] int i)
  { }
  public static void baz([Name("count")] int i,
                         [Name("id")] string s,
                         [Name("payload")] JsonElement json)
  { }
}
```

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What you’re going to see

- straightforward and annoying way to do it in C++
- how “normal” people do it: example of a way to do it in C#
- good: unpacking *array* into function parameters
- better: unpacking *dictionary* into function parameters
- automatically printing descriptions; supporting optional parameters, member functions; `constexpr`-ization

- a lot of code
- surprisingly, a lot of template programming, but fear not!
- lots and lots and LOTS of code

What we'll use

taoJSON

github.com/taocpp/json

```
namespace json = tao::json;
```

We will use C++20 (though it can be done in C++11..17)

- supported by latest MSVC (VS 2019 v. 16.11; VS 2022 v. 17)
- kinda supported by latest Clang (v. 12 and 13)
    - and GCC to a lesser degree (v. 11)

Disclaimer

```
void foo(const json::value &parameter) {
  std::cout << "parameter=\n"
    << json::to_string(parameter) << "\n";
}


foo(argument);
```

Annoying way to do it in C++

```cpp
void foo(const json::value &request) {
  std::cout << "got 'foo' request\n";
}

void bar(const json::value &request) {
  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{
              "parameter 'id' has unexpected type (expected: string)"
            };

  std::cout << "got 'bar' request with\n"
    " id='" << id->get_string() << "'\n";
}
```

Annoying way to do it in C++

```
void foo(const json::value &request) {
  std::cout << "got 'foo' request\n";
}

void bar(const json::value &request) {
  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{
              "parameter 'id' has unexpected type (expected: string)"
            };

  std::cout << "got 'bar' request with\n"
    " id='" << id->get_string() << "'\n";
}
```

Annoying way to do it in C++

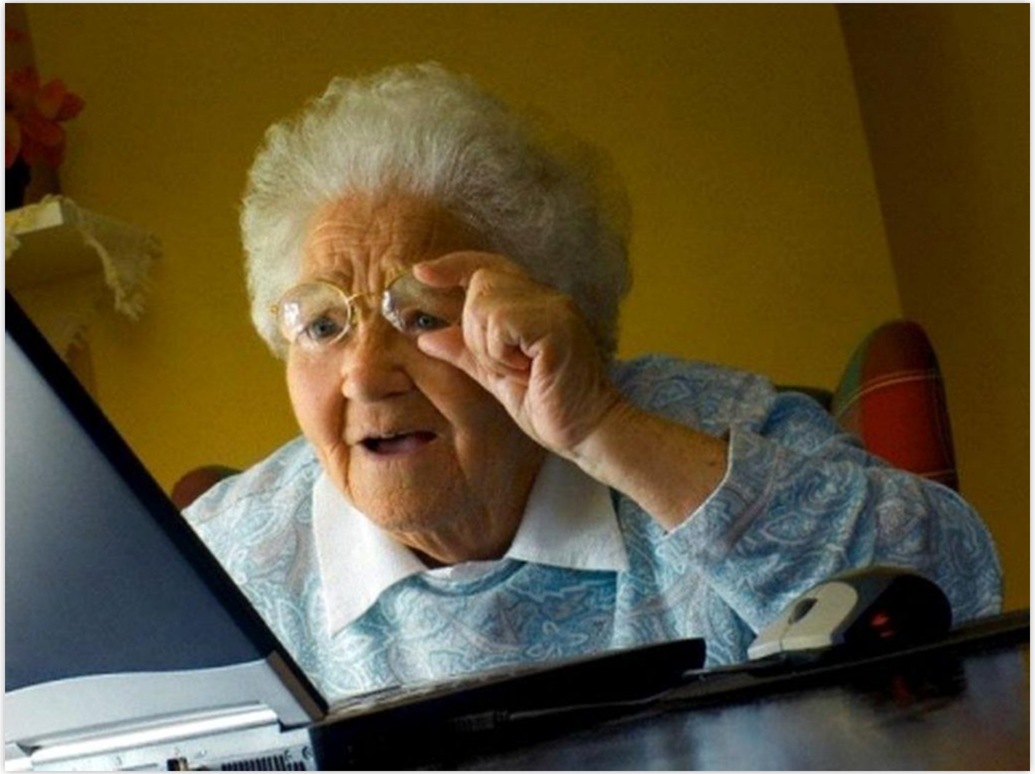
```
void foo(const json::value &request) {
  std::cout << "got 'foo' request\n";
}

void bar(const json::value &request) {
  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{
              "parameter 'id' has unexpected type (expected: string)"
            };

  std::cout << "got 'bar' request with\n"
    " id='" << id->get_string() << "'\n";
}
```

Annoying way to do it in C++

```
void foo(const json::value &request) {
  std::cout << "got 'foo' request\n";
}

void bar(const json::value &request) {
  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{
              "parameter 'id' has unexpected type (expected: string)"
            };

  std::cout << "got 'bar' request with\n"
    " id='" << id->get_string() << "'\n";
}
```

Annoying way to do it in C++

```
void baz(const json::value &request) {
  auto *count = request.find("count");
  if (!count)
    throw std::invalid_argument{ "missing parameter 'count'" };
  if (!count->is_integer())
    throw std::invalid_argument{ "parameter 'count' has unexpected type (expected: integer)" };

  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{ "parameter 'id' has unexpected type (expected: string)" };

  auto *payload = request.find("payload");
  if (!payload)
    throw std::invalid_argument{ "missing parameter 'payload'" };
  if (!payload->is_object())
    throw std::invalid_argument{ "parameter 'payload' has unexpected type (expected: object)" };

  std::cout << "got 'baz' request with\n"
    " count=" << count->as<int>() << ",\n"
    " id='" << id->get_string() << "',\n"
    " and\n"
    " payload=\n"
    << json::to_string(*payload) << "\n";
}
```

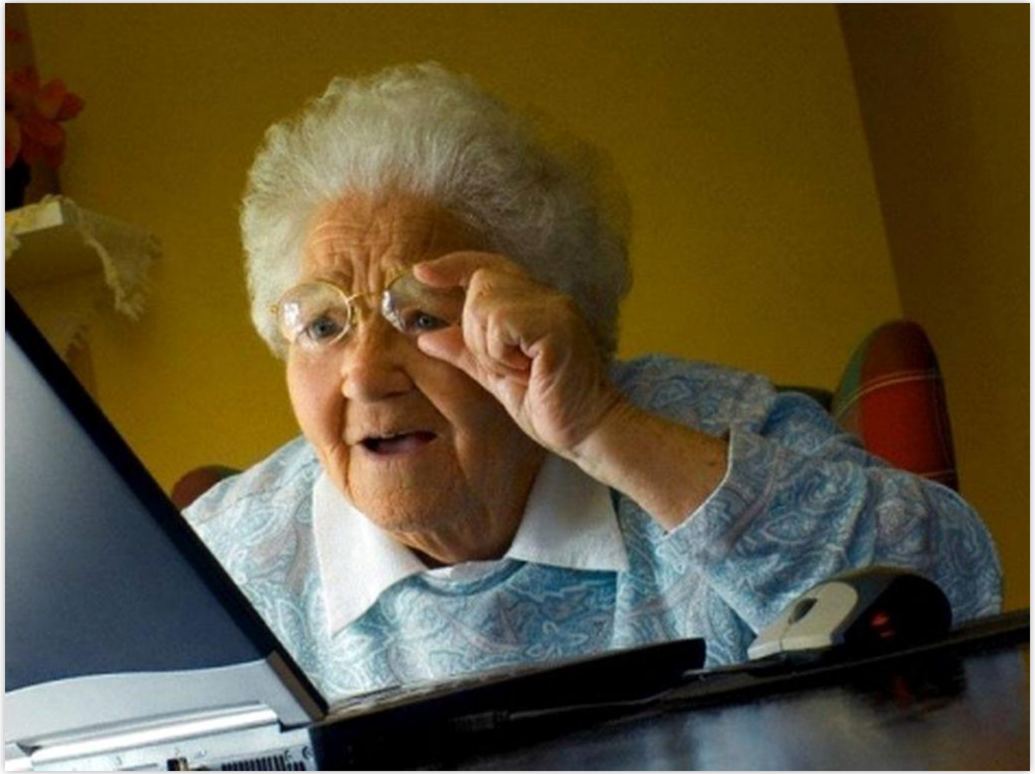


Annoying way to do it in C++

```
void baz(const json::value &request) {
  auto *count = request.find("count");
  if (!count)
    throw std::invalid_argument{ "missing parameter 'count'" };
  if (!count->is_integer())
    throw std::invalid_argument{ "parameter 'count' has unexpected type (expected: integer)" };

  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{ "parameter 'id' has unexpected type (expected: string)" };

  auto *payload = request.find("payload");
  if (!payload)
    throw std::invalid_argument{ "missing parameter 'payload'" };
  if (!payload->is_object())
    throw std::invalid_argument{ "parameter 'payload' has unexpected type (expected: object)" };

  std::cout << "got 'baz' request with\n"
    " count='" << count->as<int>() << "',\n"
    " id='" << id->get_string() << "',\n"
    " and\n"
    " payload=\n"
    << json::to_string(*payload) << "\n";
}
```

parameter validation

actual logic



Annoying way to do it in C++

```
using Name = std::string_view;
using Handler = void(*)(const json::value&);

constexpr std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz }
};

void processRequest(const std::string_view &message);
```

Annoying way to do it in C++

```
using Name = std::string_view;
using Handler = void(*)(const json::value&);

constexpr std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz }
};

void processRequest(const std::string_view &message);
```

Annoying way to do it in C++

```cpp
using Name = std::string_view;
using Handler = void(*)(const json::value&);

constexpr std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz }
};

void processRequest(const std::string_view &message);
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

```
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

```
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

```cpp
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

```
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

```
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

```
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

Annoying way to do it in C++

```
auto request = R"({
  "request":"baz",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

Annoying way to do it in C++

```
auto request = R"({
  "request":"baz",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

output:

```
* trying to process baz
got 'baz' request with
 count=1,
 id='two',
 and
 payload=
{"three":3}
```

How "normal" people do it

```
[System.AttributeUsage(System.AttributeTargets.Parameter)]
public class Name : System.Attribute {
  public string name;

  public Name(string name) {
    this.name = name;
  }
}
```

How "normal" people do it

```
public class Program {
  public static void foo() {
    Console.WriteLine("got 'foo' request");
  }

  public static void bar([Name("count")] int i) {
    Console.WriteLine("got 'bar' request with");
    Console.WriteLine($" count='{i}'");
  }

  public static void baz([Name("count")] int i, [Name("id")] string s, [Name("payload")] JsonElement p) {
    Console.WriteLine("got 'baz' request with");
    Console.WriteLine($" count='{i}'");
    Console.WriteLine($" id='{s}'");
    Console.WriteLine($" payload=\n{p.ToString()}");
  }
  //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty("request").GetString();

    var method = typeof(Program).GetMethod(name);
    var pars = new List<object>();
    foreach (var p in method.GetParameters()) {
      var paramName = p.GetCustomAttribute<Name>().name;
      var value = json.GetProperty(paramName);
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public class Program {
  //...
  public static void processRequest(string msg) {
    var json = JsonDocument.Parse(msg).RootElement;

    var name = json.GetProperty

    var method = typeof(Program
    var pars = new List<object>
    foreach (var p in method.Ge
      var paramName = p.GetCust
      var value = json.GetPrope
      pars.Add(getArg(p.ParameterType, value));
    }
    method.Invoke(null, pars.ToArray());
  }
 //...
}
```

```
public static object getArg(Type t, JsonElement v)
{
  if (t == typeof(int))
    return v.GetInt32();
  if (t == typeof(string))
    return v.GetString();
  if (t == typeof(JsonElement))
    return v;
  throw new Exception("unsupported type");
}
```

```
public class Program {
 //...
 public static void processRequest(string msg) {
  var json = JsonDocument.Parse(msg).RootElement;

  var name = json.GetProperty("request").GetString();

  var method = typeof(Program).GetMethod(name);
  var pars = new List<object>();
  foreach (var p in method.GetParameters()) {
   var paramName = p.GetCustomAttribute<Name>().name;
   var value = json.GetProperty(paramName);
   pars.Add(getArg(p.ParameterType, value));
  }
  method.Invoke(null, pars.ToArray());
 }
 //...
}
```

How "normal" people do it

```
public class Program {
  //...
  public static void Main()
  {
    const string msg = @"{
    ""request"":""baz"",
    ""count"":1,
    ""id"":""two"",
    ""payload"":{ ""three"":3 }
}";

    processRequest(msg);
  }
}
```

output:

```
got 'baz' request with
 count='1'
 id='two'
 payload=
{ "three":3 }
```

Back to C++

```cpp
void baz(const json::value &request) {
  auto *count = request.find("count");
  if (!count)
    throw std::invalid_argument{ "missing parameter 'count'" };
  if (!count->is_integer())
    throw std::invalid_argument{ "parameter 'count' has unexpected type (expected: integer)" };

  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{ "parameter 'id' has unexpected type (expected: string)" };

  auto *payload = request.find("payload");
  if (!payload)
    throw std::invalid_argument{ "missing parameter 'payload'" };
  if (!payload->is_object())
    throw std::invalid_argument{ "parameter 'payload' has unexpected type (expected: object)" };

  std::cout << "got 'baz' request with\n"
    " count=" << count->as<int>() << ",\n"
    " id='" << id->get_string() << "',\n"
    " and\n"
    " payload=\n"
    << json::to_string(*payload) << "\n";
}
```

What if...

```
void baz(const json::value &request) {
  auto *count = request.find("count");
  if (!count)
    throw std::invalid_argument{ "missing parameter 'count'" };
  if (!count->is_integer())
    throw std::invalid_argument{ "parameter 'count' has unexpected type (expected: integer)" };

  auto *id = request.find("id");
  if (!id)
    throw std::invalid_argument{ "missing parameter 'id'" };
  if (!id->is_string())
    throw std::invalid_argument{ "parameter 'id' has unexpected type (expected: string)" };
```

```
d: object)" };
```

message:

```
{
  "request": "baz",
  "count": 1,
  "id": "two",
  "payload": { "three": 3 }
}
```

message:

```
{
  "request": "baz",
  "args": [1, "two", { "three": 3 }]
}
```

```
void baz(int i,
         std::string_view s,
         const json::value &json) {
  std::cout
    << "got 'baz' request with\n"
       "  '" << i << "',\n"
       "  '" << s << "',\n"
       " and\n"
    << json::to_string(json) << '\n';
}
```

Unpacking array into function parameters

```
void foo() {
  std::cout << "got 'foo' request\n";
}

void bar(std::string_view s) {
  std::cout << "got 'bar' request with\n"
    " '" << s << "'\n";
}

void baz(int i, std::string_view s, const json::value &json) {
  std::cout << "got 'baz' request with\n"
    " '" << i << "',\n"
    " '" << s << "',\n"
    " and\n"
    << json::to_string(json) << '\n';
}
```

fits on a slide!!!

Unpacking array into function parameters

```
using Name = std::string_view;
// Handler = ???

const std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz }
};

void processRequest(const std::string_view &message);
void processRequest(const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };
  auto *args = json.find("args");

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler>(handler)(args);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

Unpacking array into function parameters

```
template<typename... Args>
void nope(void(*func)(Args...)) {

  void *erased = func; // nope

  reinterpret_cast<void*>(func); // UB

  sizeof(void*) ?= sizeof(void(*)()); // ???

  reinterpret_cast<void(*)()>(func); // OK

}
```

Unpacking array into function parameters

```
template<typename... Args>
void nope(void(*func)(Args...)) {

  void *erased = func; // nope

  reinterpret_cast<void*>(func); // UB

  sizeof(void*) ?= sizeof(void(*)()); // ???

  reinterpret_cast<void(*)()>(func); // OK

}
```

Unpacking array into function parameters

```
template<typename... Args>
void nope(void(*func)(Args...)) {

  void *erased = func; // nope

  reinterpret_cast<void*>(func); // UB

  sizeof(void*) ?= sizeof(void(*)()); // ???

  reinterpret_cast<void(*)()>(func); // OK

}
```

Unpacking array into function parameters

```
template<typename... Args>
void nope(void(*func)(Args...)) {

  void *erased = func; // nope

  reinterpret_cast<void*>(func); // UB

  sizeof(void*) ?= sizeof(void(*)()); // ???

  reinterpret_cast<void(*)()>(func); // OK

}
```

Unpacking array into function parameters

Different address widths for data and code are typical for (modified) Harvard architecture.

In a C compiler for 8-bit AVR microcontrollers:

- **function** pointer is **16** bits
- **data** pointer can be **16** or **24** bit depending on the device memory capability

In a C compiler for 8-bit and 16-bit PIC microcontrollers:

- **function** pointer can be **8**, **16** or **24** bits
- **data** pointer can be **8**, **16**, **24** or **32** bits

(depending on the device, config, and optimizations)

Side note

```
// returns pointer to object or pointer to function
void *getEntity(ID id);
reinterpret_cast<int(*)()>(getEntity(42)); // UB
```

```
union TypeErasedEntity {
  void *object;
  void(*function)();
};
TypeErasedEntity getEntity(ID id);
reinterpret_cast<int(*)()>(getEntity(42).function); // OK
```

Side note

```
// returns pointer to object or pointer to function
void *getEntity(ID id);
reinterpret_cast<int(*)()>(getEntity(42)); // UB
```

```
union TypeErasedEntity {
  void *object;
  void(*function)();
};
TypeErasedEntity getEntity(ID id);
reinterpret_cast<int(*)()>(getEntity(42).function); // OK
```

Unpacking array into function parameters

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        //...
      } }
  {}
  void operator()(const json::value *args) const {
    handlerImpl(erasedHandler, args);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value*) = nullptr;
};
```

not constexpr

Unpacking array into function parameters

not constexpr

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        //...
      } }
  {}
  void operator()(const json::value *args) const {
    handlerImpl(erasedHandler, args);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value*) = nullptr;
};
```

Unpacking array into function parameters

not constexpr

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        //...
      } }
  {}
  void operator()(const json::value *args) const {
    handlerImpl(erasedHandler, args);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value*) = nullptr;
};
```

Unpacking array into function parameters

not constexpr

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        //...
      } }
  {}
  void operator()(const json::value *args) const {
    handlerImpl(erasedHandler, args);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value*) = nullptr;
};
```

Unpacking array into function parameters

not constexpr

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        //...
      } }
  {}
  void operator()(const json::value *args) const {
    handlerImpl(erasedHandler, args);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value*) = nullptr;
};
```

```
struct Handler {
 template<typename... Args>
 Handler(void(*handler)(Args...)) :
  erasedHandler{ reinterpret_cast<void(*)()>(handler) },
  handlerImpl{
   [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
     //...
    }
    else {
     //...
    }
   } }
 {}
 //...
};
```

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

        if constexpr (sizeof...(Args) == 0) {
          //...
        }
        else {
          //...
        }
      } }
  {}
  //...
};
```

```
struct Handler {
  template<typename... Args>
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value *args) {
        auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

        if constexpr (sizeof...(Args) == 0) {
          //...
        }
        else {
          //...
        }
      } }
  {}
  //...
};
```

```cpp
//...
handlerImpl{
  [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      if (args) {
        if (!args->is_array())
          throw std::invalid_argument{ "request 'args' is not an array" };
        if (!args->get_array().empty())
          throw std::invalid_argument{ "handler expects 0 arguments" };
      }
      handler();
    }
    else {
      //...
    }
  } }
//...
```

```
//...
handlerImpl{
  [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      if (args) {
        if (!args->is_array())
          throw std::invalid_argument{ "request 'args' is not an array" };
        if (!args->get_array().empty())
          throw std::invalid_argument{ "handler expects 0 arguments" };
      }
      handler();
    }
    else {
      //...
    }
  } }
//...
```

```
//...
handlerImpl{
  [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      if (args) {
        if (!args->is_array())
          throw std::invalid_argument{ "request 'args' is not an array" };
        if (!args->get_array().empty())
          throw std::invalid_argument{ "handler expects 0 arguments" };
      }
      handler();
    }
    else {
      //...
    }
  } }
//...
```

```
//...
handlerImpl{
 [](void(*erasedHandler)(), const json::value *args) {
  auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

  if constexpr (sizeof...(Args) == 0) {
   //...
  }
  else {
   if (!args)
    throw std::invalid_argument{ "request does not contain arguments" };
   if (!args->is_array())
    throw std::invalid_argument{ "request 'args' is not an array" };
   auto &argArray = args->get_array();
   if (!validateArgs<std::decay_t<Args>...>(argArray))
    throw std::invalid_argument{ "request arguments are invalid" };

   apply(handler, argArray);
  }
 } }
```

```
//...
handlerImpl{
  [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      //...
    }
    else {
      if (!args)
        throw std::invalid_argument{ "request does not contain arguments" };
      if (!args->is_array())
        throw std::invalid_argument{ "request 'args' is not an array" };
      auto &argArray = args->get_array();
      if (!validateArgs<std::decay_t<Args>...>(argArray))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, argArray);
    }
  } }
```

```
//...
handlerImpl{
 [](void(*erasedHandler)(), const json::value *args) {
  auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

  if constexpr (sizeof...(Args) == 0) {
   //...
  }
  else {
   if (!args)
    throw std::invalid_argument{ "request does not contain arguments" };
   if (!args->is_array())
    throw std::invalid_argument{ "request 'args' is not an array" };
   auto &argArray = args->get_array();
   if (!validateArgs<std::decay_t<Args>...>(argArray))
    throw std::invalid_argument{ "request arguments are invalid" };

   apply(handler, argArray);
  }
 } }
```

```
//...
handlerImpl{
  [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      //...
    }
    else {
      if (!args)
        throw std::invalid_argument{ "request does not contain arguments" };
      if (!args->is_array())
        throw std::invalid_argument{ "request 'args' is not an array" };
      auto &argArray = args->get_array();
      if (!validateArgs<std::decay_t<Args>...>(argArray))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, argArray);
    }
  } }
```

Unpacking array into function parameters

```
template<typename... Args>
bool validateArgs(const std::vector<json::value> &args) {
  constexpr size_t ArgCount = sizeof...(Args);
  const bool argCountIsValid = args.size() == ArgCount;
  if (!argCountIsValid)
    std::cout << "* invalid arg count: " << args.size()
      << " (expected: " << ArgCount << ")\n";

  bool isValid = argCountIsValid;
  size_t index = 0;
  ((
    isValid = index < args.size() && validateArg<Args>(index, args) && isValid,
    ++index
    ), ...);
  return isValid;
}
```

Unpacking array into function parameters

```
template<typename... Args>
bool validateArgs(const std::vector<json::value> &args) {
  constexpr size_t ArgCount = sizeof...(Args);
  const bool argCountIsValid = args.size() == ArgCount;
  if (!argCountIsValid)
    std::cout << "* invalid arg count: " << args.size()
      << " (expected: " << ArgCount << ")\n";

  bool isValid = argCountIsValid;
  size_t index = 0;
  ((
    isValid = index < args.size() && validateArg<Args>(index, args) && isValid,
    ++index
    ), ...);
  return isValid;
}
```

Unpacking array into function parameters

```
template<typename... Args>
bool validateArgs(const std::vector<json::value> &args) {
  constexpr size_t ArgCount = sizeof...(Args);
  const bool argCountIsValid = args.size() == ArgCount;
  if (!argCountIsValid)
    std::cout << "* invalid arg count: " << args.size()
      << " (expected: " << ArgCount << ")\n";

  bool isValid = argCountIsValid;
  size_t index = 0;
  ((
    isValid = index < args.size() && validateArg<Args>(index, args) && isValid,
    ++index
    ), ...);
  return isValid;
}
```

Unpacking array into function parameters

```
template<typename... Args>
bool validateArgs(const std::vector<json::value> &args) {
  constexpr size_t ArgCount = sizeof...(Args);
  const bool argCountIsValid = args.size() == ArgCount;
  if (!argCountIsValid)
    std::cout << "* invalid arg count: " << args.size()
      << " (expected: " << ArgCount << ")\n";

  bool isValid = argCountIsValid;
  size_t index = 0;
  ((
    isValid = index < args.size() && validateArg<Args>(index, args) && isValid,
    ++index
    ), ...);
  return isValid;
}
```

Unpacking array into function parameters

```
template<typename... Args>
bool validateArgs(const std::vector<json::value> &args) {
  constexpr size_t ArgCount = sizeof...(Args);
  const bool argCountIsValid = args.size() == ArgCount;
  if (!argCountIsValid)
    std::cout << "* invalid arg count: " << args.size()
      << " (expected: " << ArgCount << ")\n";

  bool isValid = argCountIsValid;
  size_t index = 0;
  ((
    isValid = index < args.size() && validateArg<Args>(index, args) && isValid,
    ++index
    ), ...);
  return isValid;
}
```

Unpacking array into function parameters

```
template<typename T>
bool validateArg(size_t i, const std::vector<json::value> &args) {
  if (isConvertibleTo<T>(args[i]))
    return true;
  reportInvalidArg<T>(i, args[i]);
  return false;
}
```

Unpacking array into function parameters

```
template<typename T>
bool validateArg(size_t i, const std::vector<json::value> &args) {
  if (isConvertibleTo<T>(args[i]))
    return true;
  reportInvalidArg<T>(i, args[i]);
  return false;
}
```

Unpacking array into function parameters

```
template<typename T>
bool isConvertibleTo(const json::value &value) {
  return std::visit(
    Overloaded{
      [](bool) { return std::is_same_v<T, bool>; },
      // worksn't with GCC as of 11.2 and Clang 12 (fixed in 13)
      [](std::floating_point auto) { return std::is_floating_point_v<T>; },
      [](std::integral auto) { return std::is_arithmetic_v<T>; },
      [](auto &v) { return std::is_convertible_v<decltype(v), T>; }
    },
    value.variant());
}
```

Unpacking array into function parameters

```
template<typename T>
bool isConvertibleTo(const json::value &value) {
  return std::visit(
    Overloaded{
      [](bool) { return std::is_same_v<T, bool>; },
      // worksn't with GCC as of 11.2 and Clang 12 (fixed in 13)
      [](std::floating_point auto) { return std::is_floating_point_v<T>; },
      [](std::integral auto) { return std::is_arithmetic_v<T>; },
      [](auto &v) { return std::is_convertible_v<decltype(v), T>; }
    },
    value.variant());
}
```

Unpacking array into function parameters

```
template<typename T>
bool isConvertibleTo(const json::value &value) {
  return std::visit(
    Overloaded{
      [](bool) { return std::is_same_v<T, bool>; },
      // worksn't at all with GCC as of 11.2
      []<std::floating_point U>(U) { return std::is_floating_point_v<T>; },
      []<std::integral U>(U) { return std::is_arithmetic_v<T>; },
      [](auto &v) { return std::is_convertible_v<decltype(v), T>; }
    },
    value.variant());
}
```

Unpacking array into function parameters

```
template<typename ExpectedType>
void reportInvalidArg(size_t i, const json::value &value) {
  std::cout << "* arg " << (i + 1)
    << " has unexpected type: " << getTypeName(value.variant())
    << " (expected: " << getTypeName<ExpectedType>() << ")\n";
}
```

```cpp
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

```
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

O pattern matching, where art thou?

```
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

O pattern matching, where art thou?

O pattern matching, where art thou?

```
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

O pattern matching, where art thou?

```cpp
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

O pattern matching, where art thou?

```cpp
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

```
template<typename T>
std::string_view getTypeName() {
 if constexpr (std::is_same_v<T, bool>)
  return "boolean";
 else if constexpr (std::is_integral_v<T>)
  return "integer";
 else if constexpr (std::is_floating_point_v<T>)
  return "floating point number";
 else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
  return "string";
 else if constexpr (std::is_same_v<T, json::value::array_t>)
  return "array";
 else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
  return "object";
 else
  return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
 return std::visit([](auto &&v) {
           return getTypeName<std::decay_t<decltype(v)>>();
          }, v);
}
```

O pattern matching, where art thou?

O pattern matching, where art thou?

```cpp
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
           return getTypeName<std::decay_t<decltype(v)>>();
          }, v);
}
```

O pattern matching, where art thou?

```cpp
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                      return getTypeName<std::decay_t<decltype(v)>>();
                    }, v);
}
```

```
template<typename T>
std::string_view getTypeName() {
  if constexpr (std::is_same_v<T, bool>)
    return "boolean";
  else if constexpr (std::is_integral_v<T>)
    return "integer";
  else if constexpr (std::is_floating_point_v<T>)
    return "floating point number";
  else if constexpr (std::is_same_v<T, std::string> || std::is_same_v<T, std::string_view>)
    return "string";
  else if constexpr (std::is_same_v<T, json::value::array_t>)
    return "array";
  else if constexpr (std::is_same_v<T, json::value> || std::is_same_v<T, json::value::object_t>)
    return "object";
  else
    return "unknown type";
}
template<typename Variant>
std::string_view getTypeName(const Variant &v) {
  return std::visit([](auto &&v) {
                        return getTypeName<std::decay_t<decltype(v)>>();
                      }, v);
}
```

O pattern matching, where art thou?

```
//...
handlerImpl{
  [](void(*erasedHandler)(), const json::value *args) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      //...
    }
    else {
      if (!args)
        throw std::invalid_argument{ "request does not contain arguments" };
      if (!args->is_array())
        throw std::invalid_argument{ "request 'args' is not an array" };
      auto &argArray = args->get_array();
      if (!validateArgs<std::decay_t<Args>...>(argArray))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, argArray);
    }
  } }
```

```
//...
handlerImpl{
 [](void(*erasedHandler)(), const json::value *args) {
  auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

  if constexpr (sizeof...(Args) == 0) {
   //...
  }
  else {
   if (!args)
    throw std::invalid_argument{ "request does not contain arguments" };
   if (!args->is_array())
    throw std::invalid_argument{ "request 'args' is not an array" };
   auto &argArray = args->get_array();
   if (!validateArgs<std::decay_t<Args>...>(argArray))
    throw std::invalid_argument{ "request arguments are invalid" };

   apply(handler, argArray);
  }
 } }
```

Unpacking array into function parameters

```
template<typename... Args, size_t... I>
void applyImpl(void (*handler)(Args...),
               const std::vector<json::value> &args,
               std::index_sequence<I...>) {
  handler(getAs<std::decay_t<Args>>(args[I])...);
}

template<typename... Args>
void apply(void (*handler)(Args...),
           const std::vector<json::value> &args) {
  applyImpl(handler,
            args,
            std::make_index_sequence<sizeof...(Args)>{});
}
```

Unpacking array into function parameters

```
template<typename... Args, size_t... I>
void applyImpl(void (*handler)(Args...),
               const std::vector<json::value> &args,
               std::index_sequence<I...>) {
  handler(getAs<std::decay_t<Args>>(args[I])...);
}

template<typename... Args>
void apply(void (*handler)(Ar
           const std::vector<
  applyImpl(handler,
            args,
            std::make_index_sequence<sizeof...(Args)>{});
}
```

```
template<typename T>
decltype(auto) getAs(const json::value &v) {
  if constexpr (std::is_same_v<T, json::value>)
    return v;
  else
    return v.as<T>();
}
```

Unpacking array into function parameters

```
template<typename... Args, size_t... I>
void applyImpl(void (*handler)(Args...),
               const std::vector<json::value> &args,
               std::index_sequence<I...>) {
  handler(getAs<std::decay_t<Args>>(args[I])...);
}

template<typename... Args>
void apply(void (*handler)(Args...),
           const std::vector<json::value> &args) {
  applyImpl(handler,
            args,
            std::make_index_sequence<sizeof...(Args)>{});
}
```

Unpacking array into function parameters

```
void foo() { /*...*/ }
void bar(std::string_view s) { /*...*/ }
void baz(int i, std::string_view s, const json::value &json){/**/}

const std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz }
};

void processRequest(const std::string_view &message);
```

Unpacking array into function parameters

```
auto request = R"({
  "request":"baz",
  "args":[1, 2]
})"sv;

try {
  processRequest(request);
}
catch (const std::exception &e) {
  std::cout << "error: " << e.what() << '\n';
}
```

Unpacking array into function parameters

```
auto request = R"({
  "request":"baz",
  "args":[1, 2]
})"sv;

try {
  processRequest(request);
}
catch (cons
  std::cout
}
```

output:

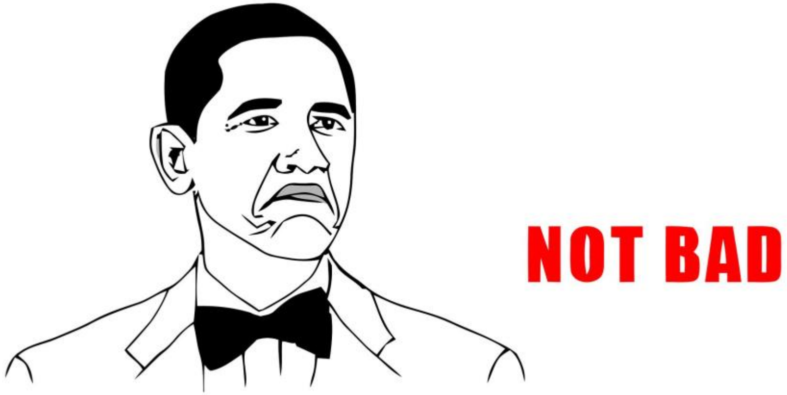
```
* trying to process baz
* invalid arg count: 2 (expected: 3)
* arg 2 has unexpected type: integer (expected: string)
error: request arguments are invalid
```

Unpacking array into function parameters

```
auto request = R"({
  "request":"baz",
  "args":[1, "two", { "three":3 }]
})"sv;

processRequest(request);
```

output:

```
* trying to process baz
got 'baz' request with
 '1',
 'two',
 and
{"three":3}
```



What if...

```
void foo();

void bar(std::string_view s);

void baz(int i,
         std::string_view s,
         const json::value &json);
```

message:

```
{
  "request": "baz",
  "args": [1, "two", { "three": 3 }]
}
```

message:

```
{
  "request": "baz",
  "count": 1,
  "id": "two",
  "payload": { "three": 3 }
}
```

```
void foo();

void bar(Param<"id", std::string_view>);

void baz(Param<"count", int>,
         Param<"id", std::string_view>,
         Param<"payload", json::value>);
```

Unpacking dictionary into named parameters

We will use C++20 because we can:

```
void bar(Param<"id", std::string_view> id);
```

But this can be done in C++11..17 (though in a bit janky way using macros):

```
void bar(PARAM("id", std::string_view) id);
```

Enter C++20!

literal type

```
template<size_t N>
struct StringLiteral {
  constexpr StringLiteral(const char(&str)[N]) {
    std::copy_n(str, N, value);
  }
  char value[N];
};
```

Enter C++20!

literal type

```
template<size_t N>
struct StringLiteral {
  constexpr StringLiteral(const char(&str)[N]) {
    std::copy_n(str, N, value);
  }
  char value[N];
};
```

Enter C++20!

literal type

```
template<size_t N>
struct StringLiteral {
  constexpr StringLiteral(const char(&str)[N]) {
    std::copy_n(str, N, value);
  }
  char value[N];
};
```

Enter C++20!

literal type

```
template<size_t N>
struct StringLiteral {
  constexpr StringLiteral(const char(&str)[N]) {
    std::copy_n(str, N, value);
  }
  char value[N];
};
```

usage:

```
template<StringLiteral Name, typename T>
struct Param {};

Param<"id", std::string_view> s;
```

Unpacking dictionary into named parameters

```
template<StringLiteral Name, typename T>
struct Param {
  using ValueType = T;
  const T &value;

  static std::string_view name() {
    return Name.value;
  }
};
```

Unpacking dictionary into named parameters

```
template<StringLiteral Name, typename T>
struct Param {
  using ValueType = T;
  const T &value;

  static std::string_view name() {
    return Name.value;
  }
};
```

Unpacking dictionary into named parameters

```
template<StringLiteral Name, typename T>
struct Param {
  using ValueType = T;
  const T &value;

  static std::string_view name() {
    return Name.value;
  }
};
```

```
void foo() {
  std::cout << "got 'foo' request\n";
}

void bar(Param<"id", std::string_view> s) {
  std::cout << "got 'bar' request with\n"
    " " << s.name() << "='" << s.value << "'\n";
}

void baz(Param<"count", int> i,
         Param<"id", std::string_view> s,
         Param<"payload", json::value> p) {
  std::cout << "got 'baz' request with\n"
    " " << i.name() << "='" << i.value << "',\n"
    " " << s.name() << "='" << s.value << "', and\n"
    " " << p.name() << "=\n"
    << json::to_string(p.value) << "\n";
}
```

```cpp
void foo() {
  std::cout << "got 'foo' request\n";
}

void bar(Param<"id", std::string_view> s) {
  std::cout << "got 'bar' request with\n"
    " " << s.name() << "='" << s.value << "'\n";
}

void baz(Param<"count", int> i,
         Param<"id", std::string_view> s,
         Param<"payload", json::value> p) {
  std::cout << "got 'baz' request with\n"
    " " << i.name() << "='" << i.value << "',\n"
    " " << s.name() << "='" << s.value << "', and\n"
    " " << p.name() << "=\n"
    << json::to_string(p.value) << "\n";
}
```

```cpp
void foo() {
  std::cout << "got 'foo' request\n";
}

void bar(Param<"id", std::string_view> s) {
  std::cout << "got 'bar' request with\n"
    "  " << s.name() << "='" << s.value << "'\n";
}

void baz(Param<"count", int> i,
         Param<"id", std::string_view> s,
         Param<"payload", json::value> p) {
  std::cout << "got 'baz' request with\n"
    "  " << i.name() << "='" << i.value << "',\n"
    "  " << s.name() << "='" << s.value << "', and\n"
    "  " << p.name() << "=\n"
    << json::to_string(p.value) << "\n";
}
```

Unpacking dictionary into named parameters

```
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value &request) { /*...*/ }
    }
  {}
  void operator()(const json::value &request) const {
    handlerImpl(erasedHandler, request);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value&) = nullptr;
};
```

Unpacking dictionary into named parameters

```
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value &request) { /*...*/ }
    }
  {}
  void operator()(const json::value &request) const {
    handlerImpl(erasedHandler, request);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value&) = nullptr;
};
```

Unpacking dictionary into named parameters

```
struct Handler {
 template<typename... Args>
  requires (IsParameter<Args>::value && ...)
 Handler(void(*handler)(Ar
  erasedHandler{ reinterpr
  handlerImpl{
   [](void(*erasedHandle
  }
 {}
 void operator()(const json::value &request) const {
  handlerImpl(erasedHandler, request);
 }
private:
 void(*erasedHandler)() = nullptr;
 void(*handlerImpl)(void(*erasedHandler)(), const json::value&) = nullptr;
};
```

```
template<typename T>
struct IsParameter : std::false_type {};

template<StringLiteral N, typename T>
struct IsParameter<Param<N, T>> : std::true_type {};
```

Unpacking dictionary into named parameters

```
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value &request) { /*...*/ }
    }
  {}
  void operator()(const json::value &request) const {
    handlerImpl(erasedHandler, request);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value&) = nullptr;
};
```

Unpacking dictionary into named parameters

```
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value &request) { /*...*/ }
    }
  {}
  void operator()(const json::value &request) const {
    handlerImpl(erasedHandler, request);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value&) = nullptr;
};
```

Unpacking dictionary into named parameters

```
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(*handler)(Args...)) :
    erasedHandler{ reinterpret_cast<void(*)()>(handler) },
    handlerImpl{
      [](void(*erasedHandler)(), const json::value &request) { /*...*/ }
    }
  {}
  void operator()(const json::value &request) const {
    handlerImpl(erasedHandler, request);
  }
private:
  void(*erasedHandler)() = nullptr;
  void(*handlerImpl)(void(*erasedHandler)(), const json::value&) = nullptr;
};
```

```
//...
erasedHandler{ reinterpret_cast<void(*)()>(handler) },
handlerImpl{
  [](void(*erasedHandler)(), const json::value &request) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      handler();
    }
    else {
      if (!validateArgs<Args...>(request))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, request);
    }
  } }
//...
```

```cpp
//...
erasedHandler{ reinterpret_cast<void(*)()>(handler) },
handlerImpl{
 [](void(*erasedHandler)(), const json::value &request) {
  auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

  if constexpr (sizeof...(Args) == 0) {
   handler();
  }
  else {
   if (!validateArgs<Args...>(request))
    throw std::invalid_argument{ "request arguments are invalid" };

   apply(handler, request);
  }
 } }
//...
```

```
//...
erasedHandler{ reinterpret_cast<void(*)()>(handler) },
handlerImpl{
  [](void(*erasedHandler)(), const json::value &request) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      handler();
    }
    else {
      if (!validateArgs<Args...>(request))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, request);
    }
  } }
//...
```

```
//...
erasedHandler{ reinterpret_cast<void(*)()>(handler) },
handlerImpl{
  [](void(*erasedHandler)(), const json::value &request) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      handler();
    }
    else {
      if (!validateArgs<Args...>(request))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, request);
    }
  } }
//...
```

```
//...
erasedHandler{ reinterpret_cast<void(*)()>(handler) },
handlerImpl{
  [](void(*erasedHandler)(), const json::value &request) {
    auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

    if constexpr (sizeof...(Args) == 0) {
      handler();
    }
    else {
      if (!validateArgs<Args...>(request))
        throw std::invalid_argument{ "request arguments are invalid" };

      apply(handler, request);
    }
  } }
//...
```

Unpacking dictionary into named parameters

```
template<typename... Args>
bool validateArgs(const json::value &request) {
  bool isValid = true;
  ((
    isValid = validateArg<Args>(request) && isValid
    ), ...);
  return isValid;
}
```

Unpacking dictionary into named parameters

```
template<typename... Args>
bool validateArgs(const json::value &request) {
  bool isValid = true;
  ((
    isValid = validateArg<Args>(request) && isValid
    ), ...);
  return isValid;
}
```

Unpacking dictionary into named parameters

```
template<typename Param>
bool validateArg(const json::value &request) {
  auto *value = request.find(Param::name());
  if (!value || value->is_null()) {
    std::cout << "* missing parameter '" << Param::name() << "'\n";
    return false;
  }
  using T = typename Param::ValueType;
  if (!isConvertibleTo<T>(*value)) {
    reportInvalidArg<T>(Param::name(), *value);
    return false;
  }
  return true;
}
```

Unpacking dictionary into named parameters

```
template<typename Param>
bool validateArg(const json::value &request) {
  auto *value = request.find(Param::name());
  if (!value || value->is_null()) {
    std::cout << "* missing parameter '" << Param::name() << "'\n";
    return false;
  }
  using T = typename Param::ValueType;
  if (!isConvertibleTo<T>(*value)) {
    reportInvalidArg<T>(Param::name(), *value);
    return false;
  }
  return true;
}
```

Unpacking dictionary into named parameters

```
template<typename Param>
bool validateArg(const json::value &request) {
  auto *value = request.find(Param::name());
  if (!value || value->is_null()) {
    std::cout << "* missing parameter '" << Param::name() << "'\n";
    return false;
  }
  using T = typename Param::ValueType;
  if (!isConvertibleTo<T>(*value)) {
    reportInvalidArg<T>(Param::name(), *value);
    return false;
  }
  return true;
}
```

Unpacking dictionary into named parameters

```
template<typename ExpectedType>
void reportInvalidArg(std::string_view name, const json::value &value) {
  std::cout << "* parameter '" << name
    << "' has unexpected type: " << getTypeName(value.variant())
    << " (expected: " << getTypeName<ExpectedType>() << ")\n";
}
```

```
//...
handlerImpl{
 [](void(*erasedHandler)(), const json::value &request) {
  auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

  if constexpr (sizeof...(Args) == 0) {
   handler();
  }
  else {
   if (!validateArgs<Args...>(request))
    throw std::invalid_argument{ "request arguments are invalid" };

   apply(handler, request);
  }
 } }
//...
```

```
//...
handlerImpl{
 [](void(*erasedHandler)(), const json::value &request) {
  auto handler = reinterpret_cast<void(*)(Args...)>(erasedHandler);

  if constexpr (sizeof...(Args) == 0) {
   handler();
  }
  else {
   if (!validateArgs<Args...>(request))
    throw std::invalid_argument{ "request arguments are invalid" };

   apply(handler, request);
  }
 } }
//...
```

Unpacking dictionary into named parameters

```
template<typename... Args>
void apply(void (*handler)(Args...), const json::value &request) {
  handler(
    { getAs<typename Args::ValueType>(request.find(Args::name())) }...
  );
}
```

Unpacking dictionary into named parameters

```
template<typename... Args>
void apply(void (*handler)(Args...), const json::value &request) {
  handler(
    { getAs<typename Args::ValueType>(request.find(Args::name())) }...
  );
}
```

```
template<typename T>
decltype(auto) getAs(const json::value *v) {
  if constexpr (std::is_same_v<T, json::value>)
    return *v;
  else
    return v->as<T>();
}
```

Unpacking dictionary into named parameters

```
void foo();
void bar(Param<"id", std::string_view> s);
void baz(Param<"count", int> i,
         Param<"id", std::string_view> s,
         Param<"payload", json::value> p);

const std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz }
};

void processRequest(const std::string_view &message);
```

identical to the initial one

Unpacking dictionary into named parameters

```
auto request = R"({
  "request":"baz",
  "count":"one",
  "payload":{ "three":3 }
})"sv;

try {
  processRequest(request);
}
catch (const std::exception &e) {
  std::cout << "error: " << e.what() << '\n';
}
```

Unpacking dictionary into named parameters

```
auto request = R"({
   "request":"baz",
   "count":"one",
   "payload":{ "three":3 }
})"sv;

try {
```

output:

```
* trying to process baz
* parameter 'count' has unexpected type: string (expected: integer)
* missing parameter 'id'
error: request arguments are invalid
```

```
}
```

Unpacking dictionary into named parameters

```
auto request = R"({
  "request":"baz",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

output:

```
* trying to process baz
got 'baz' request with
 count='1',
 id='two', and
 payload=
{"three":3}
```

Optional parameters

```
void qux(Param<"count", int> i,
         Param<"id", std::optional<std::string_view>> s,
         Param<"payload", json::value> p) {
  std::cout << "got 'qux' request with\n"
    " " << i.name() << "='" << i.value << "',\n";
  if (s.value) {
    std::cout << ' ' << s.name() << "='" << *s.value << "',\n";
  }
  else {
    std::cout << ' ' << s.name() << " parameter is absent,\n";
  }
  std::cout << " and\n " << p.name() << "=\n"
    << json::to_string(p.value) << "\n";
}
```

Optional parameters

```
void qux(Param<"count", int> i,
         Param<"id", std::optional<std::string_view>> s,
         Param<"payload", json::value> p) {
  std::cout << "got 'qux' request with\n"
    " " << i.name() << "='" << i.value << "',\n";
  if (s.value) {
    std::cout << ' ' << s.name() << "='" << *s.value << "',\n";
  }
  else {
    std::cout << ' ' << s.name() << " parameter is absent,\n";
  }
  std::cout << " and\n " << p.name() << "=\n"
    << json::to_string(p.value) << "\n";
}
```

Optional parameters

primary template

```
template<StringLiteral Name, typename T>
struct Param {
  using ValueType = T;
  const T &value;

  static std::string_view name() { return Name.value; }
};
```

template specialization

```
template<StringLiteral Name, typename T>
struct Param<Name, std::optional<T>> {
  using ValueType = std::optional<T>;
  std::optional<T> value;

  static std::string_view name() { return Name.value; }
};
```

Optional parameters

primary template

```
template<StringLiteral Name, typename T>
struct Param {
  using ValueType = T;
  const T &value;

  static std::string_view name() { return Name.value; }
};
```

template specialization

```
template<StringLiteral Name, typename T>
struct Param<Name, std::optional<T>> {
  using ValueType = std::optional<T>;
  std::optional<T> value;

  static std::string_view name() { return Name.value; }
};
```

Optional parameters

primary template

```
template<StringLiteral Name, typename T>
struct Param {
  using ValueType = T;
  const T &value;

  static std::string_view name() { return Name.value; }
};
```

template specialization

```
template<StringLiteral Name, typename T>
struct Param<Name, std::optional<T>> {
  using ValueType = std::optional<T>;
  std::optional<T> value;

  static std::string_view name() { return Name.value; }
};
```

Optional parameters

```
template<typename Param>
bool validateArg(const json::value &request) {
  auto *value = request.find(Param::name());
  if (!value || value->is_null()) {
    if constexpr (isOptional<typename Param::ValueType>)
      return true;
    std::cout << "* missing parameter '" << Param::name() << "'\n";
    return false;
  }
  //...
}
```

Optional parameters

```
template<typename Param>
bool validateArg(const json::value &request) {
  auto *value = request.find(Param::name());
  if (!value || value->is_null()) {
    if constexpr (isOptional<typename Param::ValueType>)
      return true;
    std::cout << "*
    return false;
  }
  //...
}
```

```
template<typename T>
struct IsOptional : std::false_type {};
template<typename T>
struct IsOptional<std::optional<T>> : std::true_type {};

template<typename T>
inline constexpr bool isOptional = IsOptional<T>::value;
```

```
template<typename Param>
bool validateArg(const json::value &request) {
 auto *value = request.find(Param::name());
 //...
 if constexpr (isOptional<typename Param::ValueType>) {
  using T = typename Param::ValueType::value_type;
  if (!isConvertibleTo<T>(*value)) {
   reportInvalidArg<T>(Param::name(), *value);
   return false;
  }
 }
 else {
  using T = typename Param::ValueType;
  if (!isConvertibleTo<T>(*value)) {
   reportInvalidArg<T>(Param::name(), *value);
   return false;
  }
 }
 return true;
}
```

Optional parameters

```
template<typename T>
decltype(auto) getAs(const json::value *v) {
  if constexpr (isOptional<T>) {
    return v && !v->is_null() ? T{ getAs<typename T::value_type>(v) }
                              : T{};
  }
  else
  {
    if constexpr (std::is_same_v<T, json::value>)
      return *v;
    else
      return v->as<T>();
  }
}
```

Optional parameters

```
void foo();
void bar(Param<"id", std::string_view> s);
void baz(Param<"count", int> i,
         Param<"id", std::string_view> s,
         Param<"payload", json::value> p);
void qux(Param<"count", int> i,
         Param<"id", std::optional<std::string_view>> s,
         Param<"payload", json::value> p);

const std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz },
  { "qux", &qux }
};
```

Optional parameters

```
void foo();
void bar(Param<"id", std::string_view> s);
void baz(Param<"count", int> i,
         Param<"id", std::string_view> s,
         Param<"payload", json::value> p);
void qux(Param<"count", int> i,
         Param<"id", std::optional<std::string_view>> s,
         Param<"payload", json::value> p);

const std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz },
  { "qux", &qux }
};
```

Optional parameters

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":null,
  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

output:

```
* trying to process qux
got 'qux' request with
 count='1’,
 id parameter is absent,
 and
 payload=
{"three":3}
```

Optional parameters

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":null,
  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

output:

```
* trying to process qux
got 'qux' request with
 count='1’,
 id parameter is absent,
 and
 payload=
{"three":3}
```

Optional parameters

```
auto request = R"({
  "request":"qux",
  "count":1,

  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

output:

```
* trying to process qux
got 'qux' request with
 count='1’,
 id parameter is absent,
 and
 payload=
{"three":3}
```

Optional parameters

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

processRequest(request);
```

output:

```
* trying to process qux
got 'qux' request with
 count='1',
 id='two',
 and
 payload=
{"three":3}
```

Automatic parameter description generation

```
void qux(Param<"count", int> i,
         Param<"id", std::optional<std::string_view>> s,
         Param<"payload", json::value> json);
```

```
'count': integer
'id': string (optional)
'payload': object
```

Automatic parameter description generation

```
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(*handler)(Args...)) //...
    //...
    printParamDescriptionsImpl{ [] {
      //...
    } }
  {}
  //...
  void printParamDescriptions() const {
    printParamDescriptionsImpl();
  }
private:
  //...
  void(*printParamDescriptionsImpl)() = nullptr;
};
```

Automatic parameter description generation

```
//...
printParamDescriptionsImpl{ [] {
  if constexpr (sizeof...(Args) == 0)
    std::cout << " (no arguments)\n";
  else
    (printParamDescription<Args>(), ...);
} }
//...
```

Automatic parameter description generation

```
template<typename P>
void printParamDescription() {
  using T = typename P::ValueType;
  std::cout << " '" << P::name() << "': ";
  if constexpr (isOptional<T>) {
    std::cout << getTypeName<typename T::value_type>()
      << " (optional)\n";
  }
  else {
    std::cout << getTypeName<T>() << '\n';
  }
}
```

Automatic parameter description generation

```
template<typename P>
void printParamDescription() {
  using T = typename P::ValueType;
  std::cout << " '" << P::name() << "': ";
  if constexpr (isOptional<T>) {
    std::cout << getTypeName<typename T::value_type>()
      << " (optional)\n";
  }
  else {
    std::cout << getTypeName<T>() << '\n';
  }
}
```

Automatic parameter description generation

```
template<typename P>
void printParamDescription() {
  using T = typename P::ValueType;
  std::cout << " '" << P::name() << "': ";
  if constexpr (isOptional<T>) {
    std::cout << getTypeName<typename T::value_type>()
      << " (optional)\n";
  }
  else {
    std::cout << getTypeName<T>() << '\n';
  }
}
```

Automatic parameter description generation

```
template<typename P>
void printParamDescription() {
  using T = typename P::ValueType;
  std::cout << " '" << P::name() << "': ";
  if constexpr (isOptional<T>) {
    std::cout << getTypeName<typename T::value_type>()
      << " (optional)\n";
  }
  else {
    std::cout << getTypeName<T>() << '\n';
  }
}
```

Automatic parameter description generation

```
const std::tuple<Name, Handler> handlers[] = {
  { "foo", &foo },
  { "bar", &bar },
  { "baz", &baz },
  { "qux", &qux }
};

void printDescriptions() {
  for (auto handler : handlers) {
    std::cout << '\n' << std::get<Name>(handler) << '\n';
    std::get<Handler>(handler).printParamDescriptions();
  }
}
```

Automatic parameter description generation

```
std::cout << "API description:\n";
printDescriptions();
```

Automatic parameter description generation

```
std::cout << "API description:\n";
printDescriptions();
```

```
API description:

foo
 (no arguments)

bar
 'id': string

baz
 'count': integer
 'id': string
 'payload': object

qux
 'count': integer
 'id': string (optional)
 'payload': object
```

Automatic parameter description generation

```cpp
std::cout << "API description:\n";
printDescriptions();
```

```
API description:

foo
 (no arguments)

bar
 'id': string

baz
 'count': integer
 'id': string
 'payload': object

qux
 'count': integer
 'id': string (optional)
 'payload': object
```

Unpacking dictionary into named parameters

Profit:

- virtually **no boring manual code** to parse arguments within handlers
- automatic **consistent detailed error reporting** regarding arguments
- very **easy to add new handlers** with various parameters
- automatic **generation of description**
- actual implementation of **request data structure can be abstracted**

Unpacking dictionary into named parameters

Profit:

- virtually **no boring manual code** to parse arguments within handlers
- automatic **consistent detailed error reporting** regarding arguments
- very **easy to add new handlers** with various parameters
- automatic **generation of description**
- actual implementation of **request data structure can be abstracted**

Unpacking dictionary into named parameters

Profit:

- virtually **no boring manual code** to parse arguments within handlers
- automatic **consistent detailed error reporting** regarding arguments
- very **easy to add new handlers** with various parameters
- automatic **generation of description**
- actual implementation of **request data structure can be abstracted**

Unpacking dictionary into named parameters

Profit:

- virtually **no boring manual code** to parse arguments within handlers
- automatic **consistent detailed error reporting** regarding arguments
- very **easy to add new handlers** with various parameters
- automatic **generation of description**
- actual implementation of **request data structure can be abstracted**

Unpacking dictionary into named parameters

Profit:

- virtually **no boring manual code** to parse arguments within handlers
- automatic **consistent detailed error reporting** regarding arguments
- very **easy to add new handlers** with various parameters
- automatic **generation of description**
- actual implementation of **request data structure can be abstracted**

Type erasing member functions

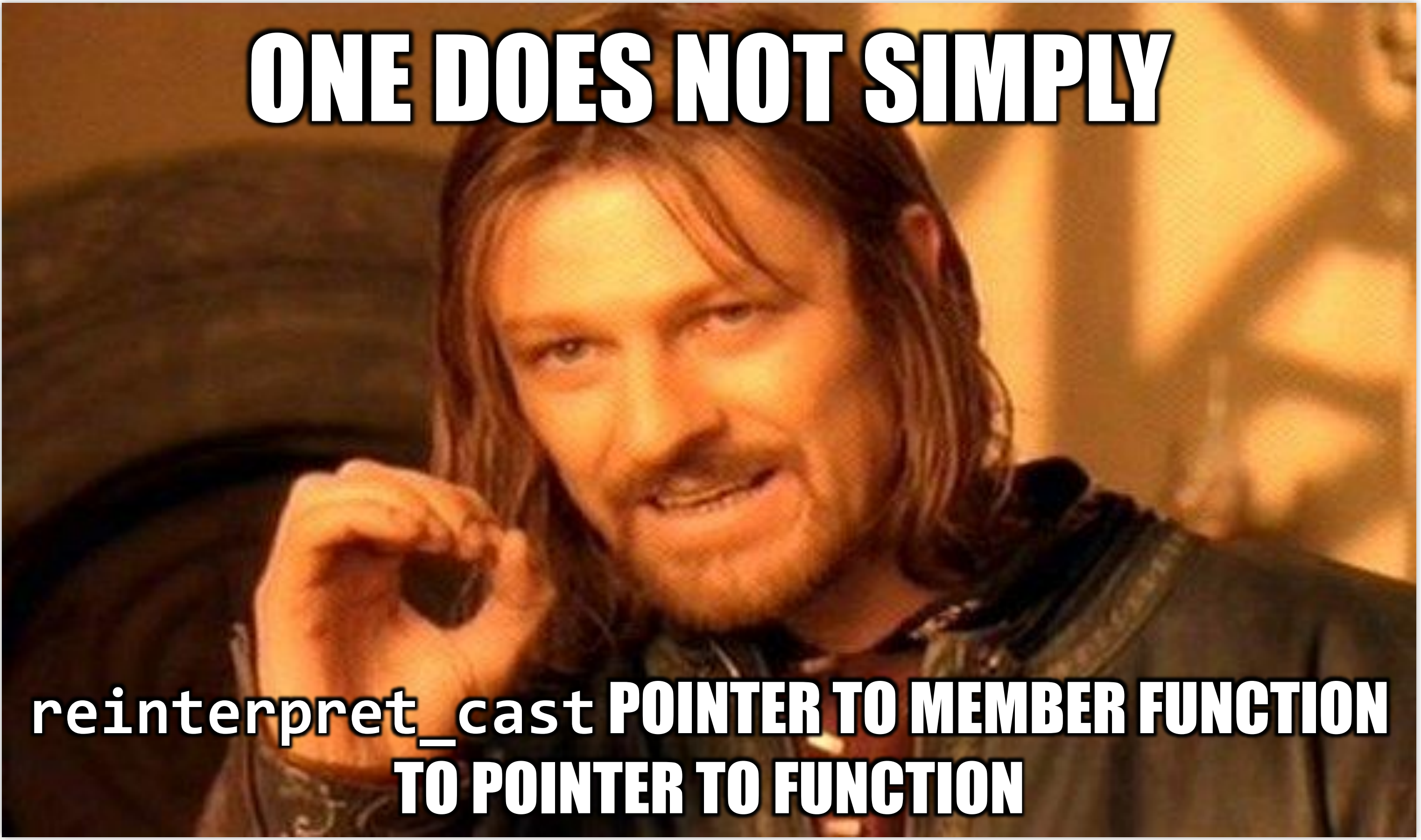
Slight nuance:

```
template<typename T, typename... Args>
void nope(void(T::*member)(Args...)) {

  reinterpret_cast<void(*)()>(member); // nope

  sizeof(member) ?= sizeof(void(*)()); // ???

}
```

Type erasing member functions

Slight nuance:

```
template
void nop

  reinte                                              e

  sizeof

}
```



ONE DOES NOT SIMPLY
reinterpret_cast POINTER TO MEMBER FUNCTION TO POINTER TO FUNCTION

Type erasing member functions

One more nuance:

```
using Storage = std::array<std::byte, sizeof(member)>;

auto erased = *reinterpret_cast<Storage*>(&member);  //UB

Storage erased;
std::memcpy(erased.data(), &member, sizeof(member)); //OK
```

Type erasing member functions

One more nuance:

```
using Storage = std::array<std::byte, sizeof(member)>;

auto erased = *reinterpret_cast<Storage*>(&member);  //UB

Storage erased;
std::memcpy(erased.data(), &member, sizeof(member)); //OK
```

Type erasing member functions

One more nuance:

```
Storage erased;
std::memcpy(erased.data(), &member, sizeof(member));
```

```
Storage erased = std::bit_cast<Storage>(member); // C++20
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
private:
  static constexpr size_t PtrToMemberFuncSize =
    sizeof(void(Processor::*)(const json::value &));
  using ErasedHandler = std::array<std::byte, PtrToMemberFuncSize>;
  ErasedHandler erasedHandler = {};
  void(*handlerImpl)(Processor&, ErasedHandler, const json::value&) =
    nullptr;
};
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
private:
  static constexpr size_t PtrToMemberFuncSize =
    sizeof(void(Processor::*)(const json::value &));
  using ErasedHandler = std::array<std::byte, PtrToMemberFuncSize>;
  ErasedHandler erasedHandler = {};
  void(*handlerImpl)(Processor&, ErasedHandler, const json::value&) =
    nullptr;
};
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
private:
  static constexpr size_t PtrToMemberFuncSize =
    sizeof(void(Processor::*)(const json::value &));
  using ErasedHandler = std::array<std::byte, PtrToMemberFuncSize>;
  ErasedHandler erasedHandler = {};
  void(*handlerImpl)(Processor&, ErasedHandler, const json::value&) =
    nullptr;
};
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
private:
  static constexpr size_t PtrToMemberFuncSize =
    sizeof(void(Processor::*)(const json::value &));
  using ErasedHandler = std::array<std::byte, PtrToMemberFuncSize>;
  ErasedHandler erasedHandler = {};
  void(*handlerImpl)(Processor&, ErasedHandler, const json::value&) =
    nullptr;
};
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
private:
  static constexpr size_t PtrToMemberFuncSize =
    sizeof(void(Processor::*)(const json::value &));
  using ErasedHandler = std::array<std::byte, PtrToMemberFuncSize>;
  ErasedHandler erasedHandler = {};
  void(*handlerImpl)(Processor&, ErasedHandler, const json::value&) =
    nullptr;
};
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value && ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
private:
  static constexpr size_t PtrToMemberFuncSize =
    sizeof(void(Processor::*)(const json::value &));
  using ErasedHandler = std::array<std::byte, PtrToMemberFuncSize>;
  ErasedHandler erasedHandler = {};
  void(*handlerImpl)(Processor&, ErasedHandler, const json::value&) =
    nullptr;
};
```

Type erasing member functions

```
//...
template<typename... Args>
  requires (IsParameter<Args>::value && ...)
Handler(void(Processor::*handler)(Args...)) :
  erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
  handlerImpl{
    [](Processor &processor,
       ErasedHandler erasedHandler,
       const json::value &request) {
       const auto handler =
         std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);
      //...
    } }
{}
//...
```

Type erasing member functions

```
//...
template<typename... Args>
  requires (IsParameter<Args>::value && ...)
Handler(void(Processor::*handler)(Args...)) :
  erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
  handlerImpl{
    [](Processor &processor,
       ErasedHandler erasedHandler,
       const json::value &request) {
       const auto handler =
         std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);
      //...
    } }
{}
//...
```

Type erasing member functions

```
//...
template<typename... Args>
  requires (IsParameter<Args>::value && ...)
Handler(void(Processor::*handler)(Args...)) :
  erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
  handlerImpl{
    [](Processor &processor,
       ErasedHandler erasedHandler,
       const json::value &request) {
       const auto handler =
         std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);
      //...
    } }
{}
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

```
//...
erasedHandler{ std::bit_cast<ErasedHandler>(handler) },
handlerImpl{
  [](Processor &processor,
     ErasedHandler erasedHandler,
     const json::value &request) {
     const auto handler =
       std::bit_cast<void(Processor::*)(Args...)>(erasedHandler);

     if constexpr (sizeof...(Args) == 0) {
       (processor.*handler)();
     }
     else {
       if (!validateArgs<Args...>(request))
         throw std::invalid_argument{ "request arguments are invalid" };

       apply(processor, handler, request);
     }
  } }
//...
```

Type erasing member functions

```
template<typename Processor, typename... Args>
void apply(Processor &processor,
           void (Processor::*handler)(Args...),
           const json::value &request) {
  (processor.*handler)(
    { getAs<typename Args::ValueType>(request.find(Args::name())) }...
  );
}
```

Type erasing member functions

```
template<typename Processor, typename... Args>
void apply(Processor &processor,
           void (Processor::*handler)(Args...),
           const json::value &request) {
  (processor.*handler)(
    { getAs<typename Args::ValueType>(request.find(Args::name())) }...
  );
}
```

Type erasing member functions

```
template<typename Processor>
struct Handler {
  //...
  void operator()(Processor &processor,
                  const json::value &request) const {
    handlerImpl(processor, erasedHandler, request);
  }
  //...
};
```

```cpp
struct Processor {
  void foo();
  void bar(Param<"id", std::string_view> s);
  void baz(Param<"count", int> i,
           Param<"id", std::string_view> s,
           Param<"payload", json::value> p);
  void qux(Param<"count", int> i,
           Param<"id", std::optional<std::string_view>> s,
           Param<"payload", json::value> p);
};

const std::tuple<Name, Handler<Processor>> handlers[] = {
  { "foo", &Processor::foo },
  { "bar", &Processor::bar },
  { "baz", &Processor::baz },
  { "qux", &Processor::qux }
};

void processRequest(Processor &processor, const std::string_view &message);
```

```
struct Processor {
  void foo();
  void bar(Param<"id", std::string_view> s);
  void baz(Param<"count", int> i,
           Param<"id", std::string_view> s,
           Param<"payload", json::value> p);
  void qux(Param<"count", int> i,
           Param<"id", std::optional<std::string_view>> s,
           Param<"payload", json::value> p);
};

const std::tuple<Name, Handler<Processor>> handlers[] = {
  { "foo", &Processor::foo },
  { "bar", &Processor::bar },
  { "baz", &Processor::baz },
  { "qux", &Processor::qux }
};

void processRequest(Processor &processor, const std::string_view &message);
```

```
struct Processor {
  void foo();
  void bar(Param<"id", std::string_view> s);
  void baz(Param<"count", int> i,
           Param<"id", std::string_view> s,
           Param<"payload", json::value> p);
  void qux(Param<"count", int> i,
           Param<"id", std::optional<std::string_view>> s,
           Param<"payload", json::value> p);
};

const std::tuple<Name, Handler<Processor>> handlers[] = {
  { "foo", &Processor::foo },
  { "bar", &Processor::bar },
  { "baz", &Processor::baz },
  { "qux", &Processor::qux }
};

void processRequest(Processor &processor, const std::string_view &message);
```

```
struct Processor {
  void foo();
  void bar(Param<"id", std::string_view> s);
  void baz(Param<"count", int> i,
           Param<"id", std::string_view> s,
           Param<"payload", json::value> p);
  void qux(Param<"count", int> i,
           Param<"id", std::optional<std::string_view>> s,
           Param<"payload", json::value> p);
};

const std::tuple<Name, Handler<Processor>> handlers[] = {
  { "foo", &Processor::foo },
  { "bar", &Processor::bar },
  { "baz", &Processor::baz },
  { "qux", &Processor::qux }
};

void processRequest(Processor &processor, const std::string_view &message);
void processRequest(Processor &processor,
                    const std::string_view &message) {
  const auto json = json::from_string(message);
  if (!json.is_object())
    throw std::invalid_argument{ "request is not a valid JSON" };
  auto *request = json.find("request");
  if (!request)
    throw std::invalid_argument{ "request does not contain name" };

  auto &name = request->get_string();
  std::cout << "* trying to process " << name << '\n';
  for (auto handler : handlers) {
    if (std::get<Name>(handler) == name) {
      std::get<Handler<Processor>>(handler)(processor, json);
      return;
    }
  }
  std::cout << "* could not handle request " << name << '\n';
}
```

Type erasing member functions

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

Processor processor{};

processRequest(processor, request);
```

Type erasing member functions

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

Processor processor{};

processRequest(processor, request);
```

Type erasing member functions

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

Processor processor{};

processRequest(processor, request);
```

output:

```
* trying to process qux
got 'qux' request with
 count='1',
 id='two',
 and
 payload=
{"three":3}
```

Using std::function

```
template<typename Processor>
struct Handler {
  template<typename... Args>
    requires (IsParameter<Args>::value&& ...)
  Handler(void(Processor::*handler)(Args...)) //...
  //...
  void operator()(Processor &processor,
                  const json::value &request) const {
    erasedHandler(processor, request);
  }
private:
  std::function<void(Processor&, const json::value&)> erasedHandler;
};
```

Using std::function

```
//...
Handler(void(Processor::*handler)(Args...)) :
  erasedHandler{
    [handler](Processor &processor, const json::value &request) {
      if constexpr (sizeof...(Args) == 0) {
        (processor.*handler)();
      }
      else {
        if (!validateArgs<Args...>(request))
          throw std::invalid_argument{ "request arguments are invalid" };

        apply(processor, handler, request);
      }
    } }
{}
//...
```

constexpr-ization

😐

```
const std::tuple<Name, Handler<Processor>> handlers[] = {
  { "foo", &Processor::foo },
  { "bar", &Processor::bar },
  { "baz", &Processor::baz },
  { "qux", &Processor::qux }
};
```

constexpr-ization

```
template<auto value>
void idea() {
  auto f = [] {
    value;
  };
}
```

constexpr-ization

```
template<typename Processor>
struct Handler {
  template<auto handler>
  // TODO: requires (...)
  static constexpr Handler makeHandler() {
    return makeHandlerImpl<handler>(handler);
  }

  constexpr Handler() = default;
  void operator()(Processor &processor, const json::value &request) const {
    handlerImpl(processor, request);
  }
private:
  //...
  void(*handlerImpl)(Processor&, const json::value&) = nullptr;
};
```

constexpr-ization

```
template<typename Processor>
struct Handler {
  template<auto handler>
  // TODO: requires (...)
  static constexpr Handler makeHandler() {
    return makeHandlerImpl<handler>(handler);
  }

  constexpr Handler() = default;
  void operator()(Processor &processor, const json::value &request) const {
    handlerImpl(processor, request);
  }
private:
  //...
  void(*handlerImpl)(Processor&, const json::value&) = nullptr;
};
```

constexpr-ization

```
template<typename Processor>
struct Handler {
  template<auto handler>
  // TODO: requires (...)
  static constexpr Handler makeHandler() {
    return makeHandlerImpl<handler>(handler);
  }

  constexpr Handler() = default;
  void operator()(Processor &processor, const json::value &request) const {
    handlerImpl(processor, request);
  }
private:
  //...
  void(*handlerImpl)(Processor&, const json::value&) = nullptr;
};
```

constexpr-ization

```
template<typename Processor>
struct Handler {
  template<auto handler>
  // TODO: requires (...)
  static constexpr Handler makeHandler() {
    return makeHandlerImpl<handler>(handler);
  }

  constexpr Handler() = default;
  void operator()(Processor &processor, const json::value &request) const {
    handlerImpl(processor, request);
  }
private:
  //...
  void(*handlerImpl)(Processor&, const json::value&) = nullptr;
};
```

```cpp
template<typename Processor>
struct Handler {
  //...
private:
  constexpr Handler(void(*handlerImpl)(Processor&, const json::value&)) :
    handlerImpl{ handlerImpl }
  {}

  template<auto handler, typename... Args>
  static constexpr auto makeHandlerImpl(void (Processor::*)(Args...)) {
    return Handler{
      [](Processor &processor, const json::value &request) {
        //...
      }
    };
  }

  void(*handlerImpl)(Processor&, const json::value&) = nullptr;
};
```

```
template<typename Processor>
struct Handler {
  //...
private:
  constexpr Handler(void(*handlerImpl)(Processor&, const json::value&)) :
    handlerImpl{ handlerImpl }
  {}

  template<auto handler, typename... Args>
  static constexpr auto makeHandlerImpl(void (Processor::*)(Args...)) {
    return Handler{
      [](Processor &processor, const json::value &request) {
        //...
      }
    };
  }

  void(*handlerImpl)(Processor&, const json::value&) = nullptr;
};
```

```
//...
template<auto handler, typename... Args>
static constexpr auto makeHandlerImpl(void (Processor::*)(Args...)) {
  return Handler{
    [](Processor &processor, const json::value &request) {
      if constexpr (sizeof...(Args) == 0) {
        (processor.*handler)();
      }
      else {
        if (!validateArgs<Args...>(request))
          throw std::invalid_argument{ "request arguments are invalid" };

        apply(processor, handler, request);
      }
    }
  };
}
//...
```

```
//...
template<auto handler, typename... Args>
static constexpr auto makeHandlerImpl(void (Processor::*)(Args...)) {
  return Handler{
    [](Processor &processor, const json::value &request) {
      if constexpr (sizeof...(Args) == 0) {
        (processor.*handler)();
      }
      else {
        if (!validateArgs<Args...>(request))
          throw std::invalid_argument{ "request arguments are invalid" };

        apply(processor, handler, request);
      }
    }
  };
}
//...
```

constexpr-ization

```
template<auto handler>
constexpr auto h() {
  return Handler<Processor>::makeHandler<handler>();
}
```

```
constexpr std::tuple<Name, Handler<Processor>> handlers[] = {
  { "foo", h<&Processor::foo>() },
  { "bar", h<&Processor::bar>() },
  { "baz", h<&Processor::baz>() },
  { "qux", h<&Processor::qux>() }
};
```

constexpr-ization

```cpp
template<auto handler>
constexpr auto h(std::string_view name) {
  return std::tuple{
    name, Handler<Processor>::makeHandler<handler>()
  };
}


constexpr std::tuple<Name, Handler<Processor>> handlers[] = {
  h<&Processor::foo>("foo"),
  h<&Processor::bar>("bar"),
  h<&Processor::baz>("baz"),
  h<&Processor::qux>("qux")
};
```

constexpr-ization

```
template<StringLiteral name, auto handler>
constexpr auto h() {
  return std::tuple{
    Name{ name.value },
    Handler<Processor>::makeHandler<handler>()
  };
}
```

😀

```
constexpr std::tuple<Name, Handler<Processor>> handlers[] = {
  h<"foo", &Processor::foo>(),
  h<"bar", &Processor::bar>(),
  h<"baz", &Processor::baz>(),
  h<"qux", &Processor::qux>()
};
```

constexpr-ization

```
auto request = R"({
  "request":"qux",
  "count":1,
  "id":"two",
  "payload":{ "three":3 }
})"sv;

Processor processor{};

processRequest(processor, request);
```

output:

```
* trying to process qux
got 'qux' request with
 count='1',
 id='two',
 and
 payload=
{"three":3}
```

Common parameter aliases

```
struct Processor {
  void foo();
  void bar(Param<"id", std::string_view> s);
  void baz(Param<"count", int> i,
           Param<"id", std::string_view> s,
           Param<"payload", json::value> p);
  void qux(Param<"count", int> i,
           Param<"id", std::optional<std::string_view>> s,
           Param<"payload", json::value> p);
};
```

Common parameter aliases

```
using IDParam = Param<"id", std::string_view>;
using CountParam = Param<"count", int>;
using PayloadParam = Param<"payload", json::value>;

struct Processor {
  void foo();
  void bar(IDParam s);
  void baz(CountParam i,
           IDParam s,
           PayloadParam p);
  void qux(CountParam i,
           Param<"id", std::optional<std::string_view>> s,
           PayloadParam p);
};
```

Future?

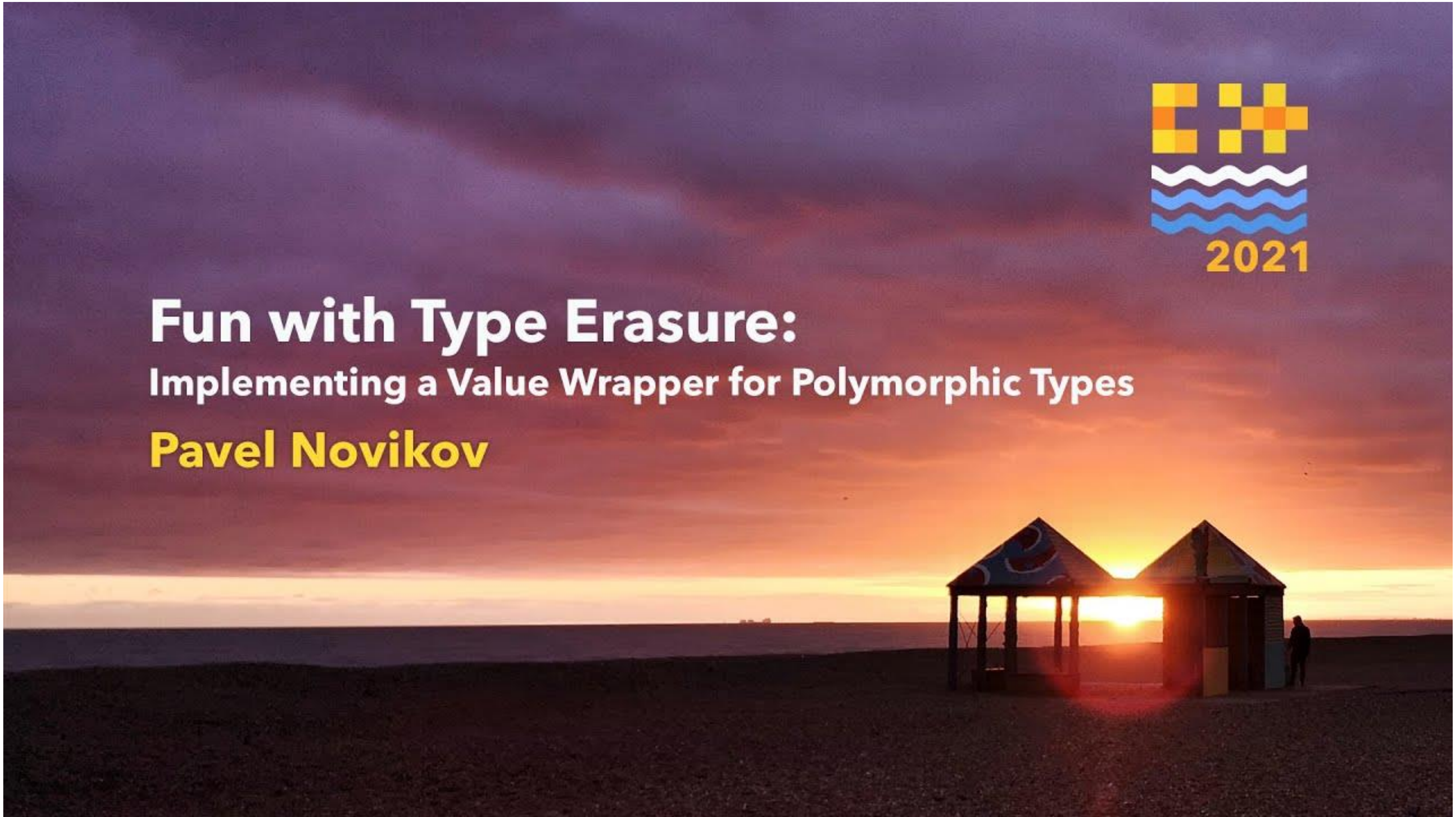
- using attributes and metaprogramming?

```
[HandlerName("qux")] void qux(
  [Name("count")] int i,
  [Name("id")] std::optional<std::string_view> s,
  [Name("payload")] const json::value &p);
```

- parsing/generating schemas from a set of handlers at build time?

Thanks for listening!

https://youtu.be/bkWpL6wEX6M



Fun with Type Erasure:

Implementing a Value Wrapper for Polymorphic Types

Pavel Novikov

Fun with type erasure

+ dispatching data from abstract structure to function parameters

Pavel Novikov

@cpp_ape

R&D Align Technology

align

Thanks to Timur Doumler for feedback!

Slides: https://git.io/JKWQx

References

- taoJSON https://github.com/taocpp/json
- Harvard architecture https://en.wikipedia.org/wiki/Harvard_architecture
- MPLAB® XC Compilers documentation https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-xc-compilers#Documentation
- fun with strings as template arguments in C++17, github gist: https://git.io/JGLV2
- CppCon 2019: Timur Doumler “Type punning in modern C++” https://youtu.be/_qzMpk-22cc

Demos: https://git.io/J1Y5H

	taocpp/json	nlohmann/json
• unpacking array into function parameters:	godbolt.org/z/z1zGdfoG1	godbolt.org/z/zroor1E8f
• unpacking dictionary into named parameters:	godbolt.org/z/b7sqdKv5e	godbolt.org/z/Pe9446aEP
• member functions support:	godbolt.org/z/x57GbM4Kr	godbolt.org/z/GT31x5T9a

Bonus slides

Overload

```
template<typename... F>
struct Overloaded : F... {
  using F::operator()...;
};
// deduction guide is needed until
// all compilers fully implement C++20
template<typename... F>
Overloaded(F...)->Overloaded<F...>;
```