Help! My Codebase has 5 JSON Libraries

How Generic Programming Came to the Rescue

CHRISTOPHER MCARTHUR

Cppcon
The C++ Conference

2022
September 12th-16th

Focus of the talk - “implementing traits with functions”

- Explanation of template metaprogramming implementation to abstraction JSON libraries.
    - Detecting if a function or method are implement for a type
    - Checking if an ADL implementation exists
    - Compile time requirements and SFINAE

If you are looking for an introduction to the topic check out these other presentations:

- Back to Basics: Templates by Bob Steagall - CppCon 2021
- Type Traits by Joby Hagins - CppCon 2020

You do not need to understand JSON or JWT for this talk.

Were you really using 5 JSON libraries?!

This is the first question people ask when they learn about my talk. No, we only "used" 4 but the includes and linkers were getting all 5… someone *could* easily use the 5th. Not knowing is wo

So what? Why was this a problem?

The client side used different string types depending on the OS and the server side need schema validation and both directions need to parse with PicoJSON for OAuth2 which did not have the same limitations. **The code was convoluted and began to duplicated.**

During an industry workshop, a group of us C++ developers where gripping over lunch about the leap in complexity.

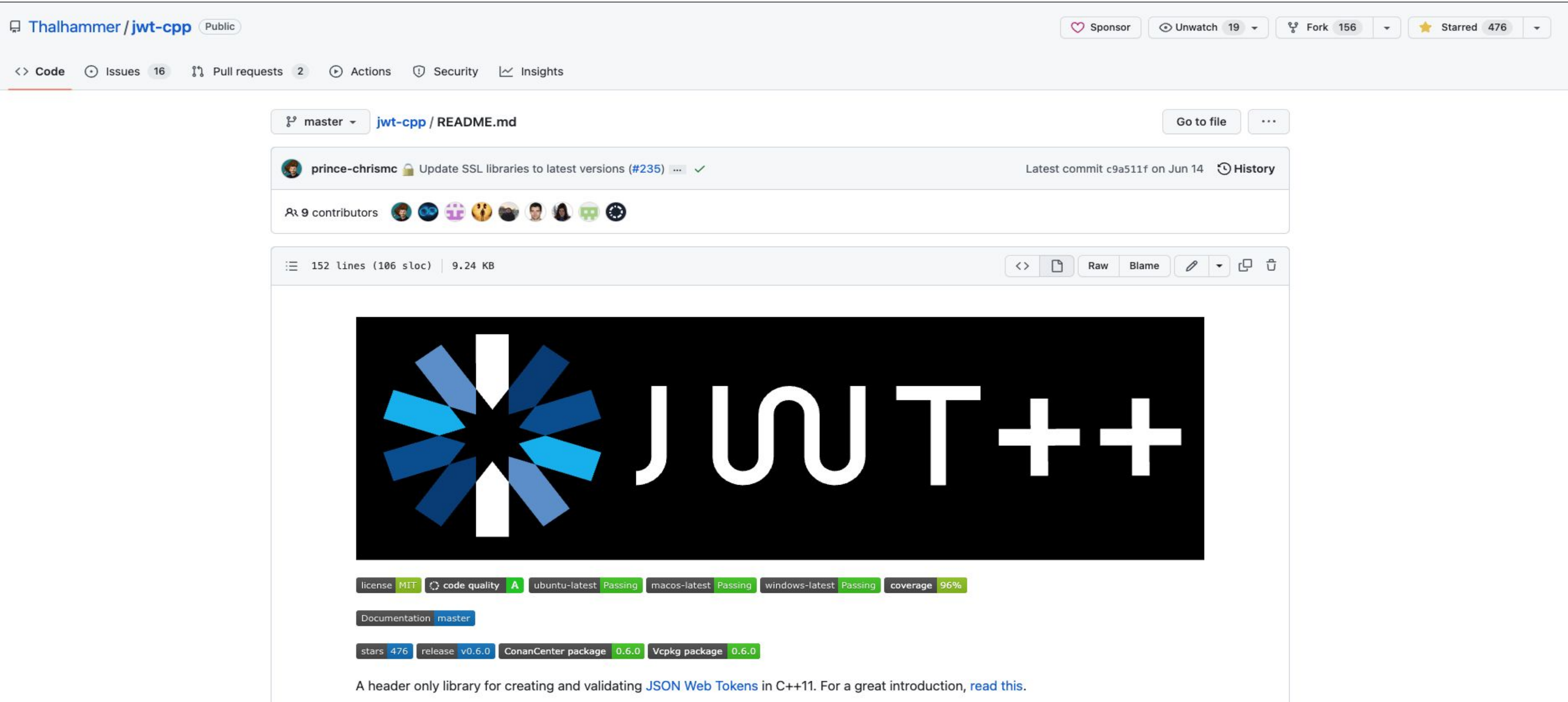
> “Why don’t you template out the logic and metaprogram a traits implementation?” - Geeze, Thanks Gareth

So what does the solution look like?

```
template<typename json_traits>
class basic_claim {
    static_assert(details::is_valid_traits<json_traits>::value,
        "traits must satisfy requirements");
    static_assert(
        details::is_valid_json_types<typename json_traits::value_type,
            typename json_traits::string_type,
            typename json_traits::integer_type,
            typename json_traits::object_type,
            typename json_traits::array_type>::value,
        "must satisfy json container requirements");
}
```

https://github.com/Thalhammer/jwt-cpp/blob/c9a511f436eaa13857336ebeb44dbc5b7860fe01/include/jwt-cpp/jwt.h#L2123-L2141

Open Source Project Highlight



So what does the solution look like?

```cpp
template<typename json_traits>
class basic_claim {
    static_assert(details::is_valid_traits<json_traits>::value,
        "traits must satisfy requirements");
    static_assert(
        details::is_valid_json_types<
            typename json_traits::value_type,
            typename json_traits::string_type,
            typename json_traits::integer_type,
            typename json_traits::object_type,
            typename json_traits::array_type>::value,
        "must satisfy json container requirements");
}
```

?????

https://github.com/Thalhammer/jwt-cpp/blob/c9a511f43
0fe01/include/jwt-cpp/jwt.h#L2123-L2141

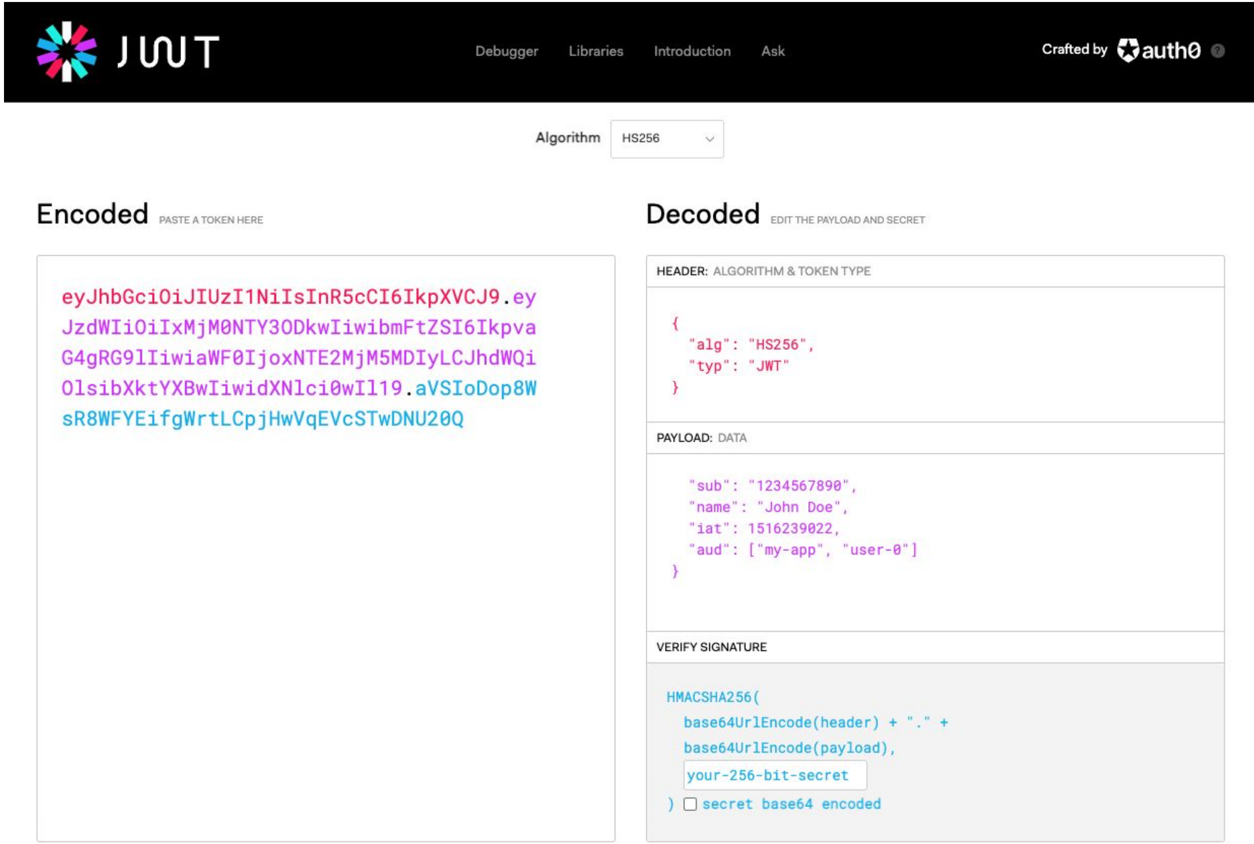
So what is a JWT?

https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM

JSON (JavaScript Object Notation) Web Token (RFC 7519 and more)

- Predefined keys with special meanings.
- Limited number of types

What should be inside a token is documented.



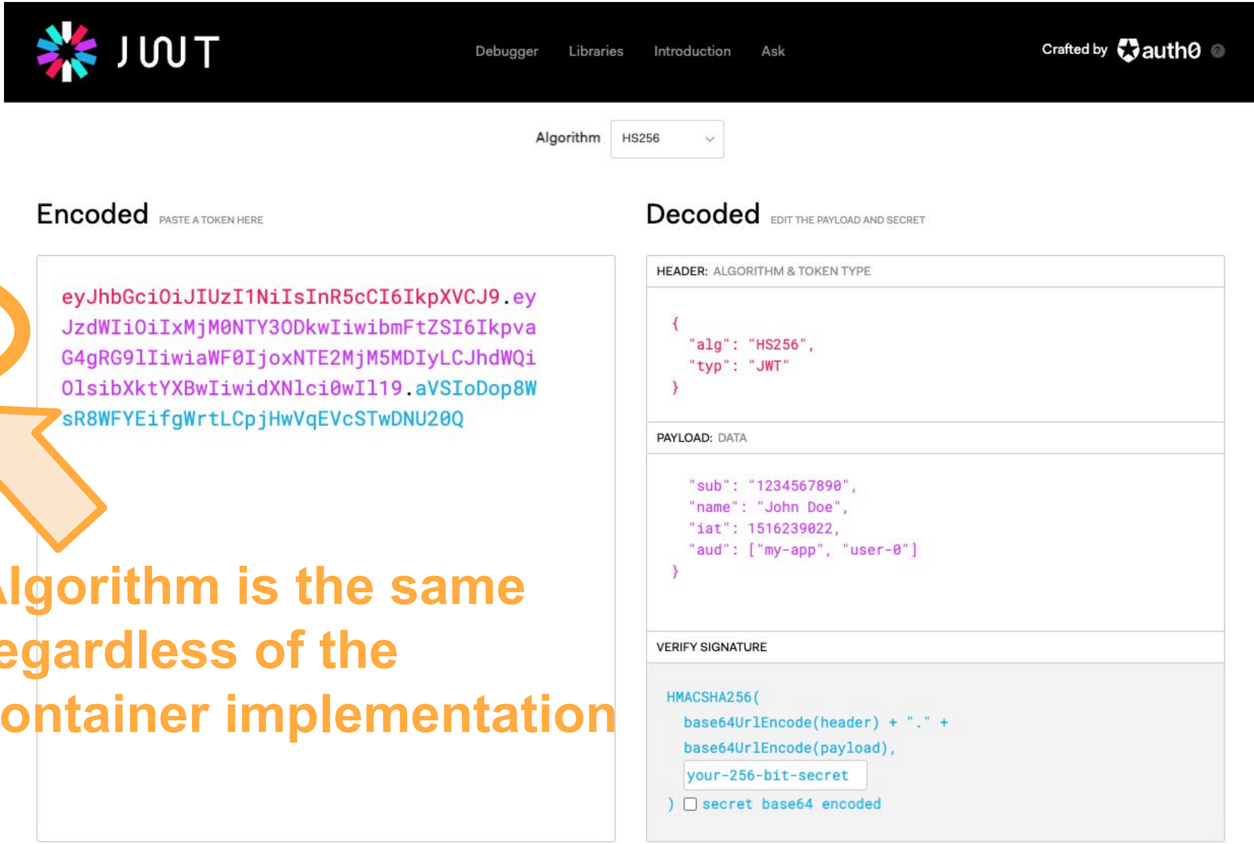
So what is a JWT?

https://jwt.io/#debugger-io?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM

JSON (JavaScript Object Notation) Web Token (RFC 7519 and more)

- Predefined keys with special meanings.
- Limited number of types

What should be inside a token is documented.



Algorithm is the same regardless of the container implementation

What properties or “policies” does JWT-CPP’s algorithm need?

There are two “user stories” one to create and issue new tokens and the other to verify tokens that have been received.

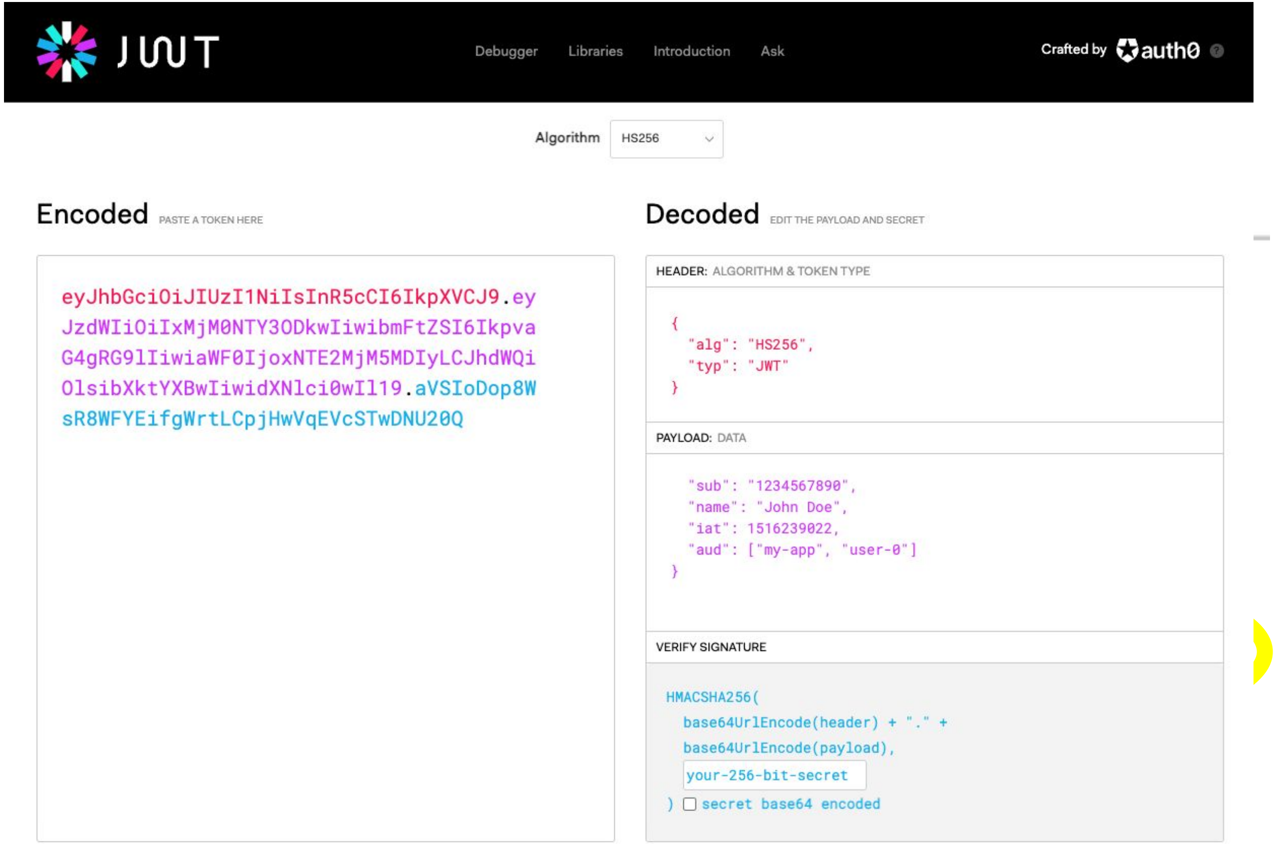
- Create: build JSON objects with strings, integers, and arrays. Convert from JSON to string to be encoded.
- Verify: parse string to JSON objects. Iterate over keys and convert generic values to specific types to validate claims.

Our traits needs a set of types and functions to implement the algorithm.

So how does JWT-CPP verify claims?

Let's take a look at the `aud` claim.

- This can be either a string **or** an array of string.



https://github.com/Thalhammer/jwt-cpp/blob/c9a511f43b7860fe01/include/jwt-cpp/jwt.h#L3016-L3017

Let's verify the `aud` claim?

```
verifier& with_audience(const typename json_traits::array_type& aud) {
    claims["aud"] = verify_ops::is_subset_claim<json_traits>{aud};
    return *this;
}

// Snippet of `is_subset_claim::operator()`
if (claim.get_type() == json::type::string) {
    if (expected.size() != 1 || *expected.begin() != claim.as_string())
        // ...
} else if (claim.get_type() == json::type::array) {
    auto jc = claim.as_array();
    for (auto& e : expected) {
        // ...
    }
}
```

Generic "value_type" claim that we need to convert to an "array_type"

PAYLOAD:


```
"sub": "1234567890",
"name": "John Doe",
"iat": 1516239022,
"aud": ["my-app", "user-0"]
}
```

What might this look like?

```
// pseudo example of a partial traits implementation
struct value{};
using array = std::vector<value>;
struct example_traits{
  static array as_array(const value&);
};

int main()
{
    static_assert(/* INSERT_MAGIC_HERE */,
                  "missing `array as_array(const value&)`");
}
```

Generic “value_type” claim that we need to convert to an “array_type”

How can we check a type implements a function?

Is Detected

This a an STL function, Template Fundamentals v2 which is experimental but available.

- Uses SFINAE to detect if a “named entity” is present
  - Compiles to true or false type
  - When true it capture the type otherwise it defaults to “nonesuch”
- Takes an “operator” to extract an entity and a “type” check.

https://en.cppreference.com/w/cpp/experimental/is_detected

Is Detected

```
// `is_detected` requires an "operator" to extract the type
template<typename traits_type>
using as_array_t = decltype(traits_type::as_array);

struct value{};
using array = std::vector<value>;
struct example_traits{static array as_array(const va

int main()
{
    static_assert(std::experimental::is_detected<
                        as_array_t, example_traits>::value,
                  "missing `array as_array(const value&)`");
}
```

https://godbolt.org/z/hT54nz3rd

Is Detected

```
// `is_de
template<
using as_

struct ba
    int as
};

int main(
{
    stati

            missing  array as_array(const value&)  );
}
```

mpiled
essfully!

https://godbolt.org/z/6xYdqxfbd https://en.cppreference.com/w/cpp/utility/functional/function

Is Function

```
// `is_detected` requires an "operator" to extract the type
template<typename traits_type>
using as_array_t = decltype(traits_type::as_array);

struct bad_traits{
   int as_array;
};

int main()
{
    static_assert(std::experimental::is_detected<
                           as_array_t, bad_traits>::value
                  && std::is_function<as_array_t<bad_traits>>::value,
                  "missing `array as_array(const value&)`");
}
```

Compilation fails because of static_assert

https://godbolt.org/z/56WPcKbGx

Is Function

```
// `is_detected` rec                    act the type
template<typename tr
using as_array_t = d                    ay);

struct bad_traits{
   static int as_ar
};

int main()
{
    static_assert(st                    d<
                                        :value
             &&                         <bad_traits>>::value,
              "n                        value&)`");
}
```

Compiled Successfully!

https://godbolt.org/z/19o7d4oro

Is Signature

There are no build in methods for this one. So we will need to write our own.

```
template<typename op, typename signature>
using is_signature = typename std::is_same<
    op,          // Extract type of “as array”
    signature>; // Compare function signature
```

Here we are using an “operator” to get the type of a entity and comparing it to a “signature”

Is Signature

```
template<typename op, typename signature>
using is_signature = typename std::is_same<
    op,          // Extract type of “as array”
    signature>; // Compare function signature
```

We can re-use the “extract operator” we made for `is_detected`

```
// `is_detected` requires an "operator" to extract the type
template<typename traits_type>
using as_array_t = decltype(traits_type::as_array);
```

Is Signature

```
template<typename op, typename signature>
using is_signature = typename std::is_same<
    op,         // Extract type of “as array”
    signature>; // Compare function signature
```

Function types are slightly different from function pointers or references as they have no id or name. Other the syntax should feel.

```
struct example_traits{
   static array as_array(const value&);
};
```

```
array(const value&);
```

Is Signature

```
template<typename op, typename signature>
using is_signature = typename std::is_same<op, signature>;

struct value{};
using array = std::vector<value>;
struct example_traits{static array as_array(const val

int main()
{
    static_assert(
        std::experimental::is_detected<as_array_t, exampl                :value
        && std::is_function<as_array_t<example_traits>>::value
        && is_signature<as_array_t<example_traits>,
                        array(const value&)>::value,
        "missing `array as_array(const value&)`");
}
```

Compiled Successfully!

https://godbolt.org/z/5xr1Txs9s

Is function signature detected

```cpp
template<typename traits_type, template<typename...> class Op, typename
Signature>
struct is_function_signature_detected{
    using type = Op<traits_type>;
    static constexpr auto value =
        std::experimental::is_detected<Op, traits_type>::value &&
        std::is_function<type>::value && is_signature<type>, Signature>::value;
};

struct value{};
using array = std::vector<value>;
struct example_traits{static array as_array(const value&);};

int main()
{
    static_assert(is_function_signature_detected<example_traits, as_array_t,
                  array(const value&)>::value,"must be present");
}
```

https://godbolt.org/z/1WbY7rKxf

So we have verified a claim… What about creating tokens?

Creating a JWT and Concatenating strings

When creating a JWT one of the key steps is concatenating the 3 parts together.

“string_type” from our traits

```
const auto header = encode(serialize(header_claims));
const auto payload = encode(erialize(payload_claims));
const auto token = header + "." + payload;

auto signature = algo.sign(token, ec);
if (ec) return {};

return token + "." + encode(signature);
```

std::operator+(string_type, string_type)

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyLCJhdWQi
OlsibXktYXBwIiwidXNlci0wIl19.aVSIoDop8W
sR8WFYEifgWrtLCpjHwVqEVcSTwDNU20Q

https://github.com/Thalhammer/jwt-cpp/blob/c9a511f436eaa138573eb44dbc5b7860fe01/include/jwt-cpp/jwt.h#L2869-L2876

Is std::operator+ implemented?

We already have `is_signature` that should be easy enough?

```
template<typename string_type>
using is_std_operate_plus_signature =
    is_signature<decltype(std::operator+),
                 string_type(string_type, string_type)>;

int main()
{
   static_assert(
      is_std_operate_plus_signature<std::string>::value,
      missing `std::operator+(std::string, std::string`");
}
```

https://godbolt.org/z/Woeqbjz8z

Is std::o

We alread

```
template<type
using is_std_
    is_signat

int main()
{
   static_ass

}
```

error: 'decltype
x86-64 gcc 12

)

on

Is std::operator+ implemented?

How do you resolve an overloaded function at compile time? https://youtu.be/dLZcocFOb5Q?t=2390
Same way as always we give it parameters and see if it compiles!

```
template<typename string_type>
using is_std_operate_plus_signature = typename std::is_same<
    decltype(
        // Let's call `std:operator+`
        std::operator+(
            // passing a reference to "string_type" at compile time
            std::declval<string_type>(), std::declval<string_type>())
    // ^^^ `decltype` will capture the return type
    ),
    string_type>; // And we expect it to return "string_type"
```

Is std::operator+ implemented?

```
template<typename string_type>
using is_std_operate_plus_signature = typename std::is_same<
    decltype(
        std::operator+(
            std::declval<string_type>(), std::declval                 )),
    string_type>;

int main()
{
   static_assert(
      is_std_operate_plus_signature<std::string>::value,
      missing `std::operator+(std::string, std::string`");
}
```

https://godbolt.org/z/6rhqEabb5

Splitting a JWT into each section

Parsing a JWT requires splicting each of the 3 parts to decode individually.

```
const auto hdr_end = token.find('.');
const eauto payload_end = token.find('.', hdr_end + 1);
header = decode(token.substr(0, hdr_end));
payload = decode(token.substr(hdr_end + 1, payload_end -
hdr_end - 1));
signature = decode(token.substr(payload_end + 1));
```

string_type string_type::substr(
interger_type, integer_type)

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyLCJhdWQi
OlsibXktYXBwIiwidXNlci0wIl19.aVSIoDop8W
sR8WFYEifgWrtLCpjHwVqEVcSTwDNU20Q

https://github.com/Thalhammer/jwt-cpp/blob/c9a511f436eaa138573eb44dbc5b7860fe01/include/jwt-cpp/jwt.h#L2582-L2588

Signature of member functions

Will `is_signature` work this time?

```
template<typename string_type>
using string_substr = decltype(string_type::substr);

int main()
{
   static_assert(is_function_signature_detecte
        std::string, string_substr,
        std::string(std::string::size_type,std::        type)
           >::value,
                 "invlaid `string_type::substr()` method");
}
```

Compilation Failure!

```
error: invalid use of non-static member function
   6 | using string_substr = decltype(string_type::substr);
```

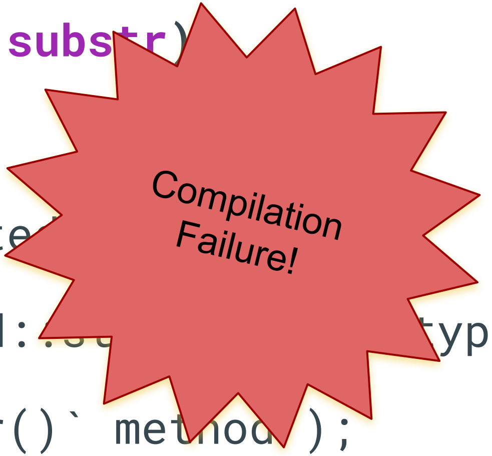
https://godbolt.org/z/T9P67e1ra

Signature of member functions

Easy, we already made `is_function_signature_detected`.

```
template<typename string_type>
using string_substr = decltype(&string_type::subs    r)

int main()
{
   static_assert(is_function_signature_detecte
       std::string, string_substr,
       std::string(std::string::size_type,std::            type)
          >::value,
             "invalid `string_type::substr()` met  od");
}
```



error: static assertion failed

https://godbolt.org/z/q8Wa5rhaW

Signature of member functions

Breaking down the assertions.

```
template<typename string_type>
using string_substr = decltype(&string_type::substr);

template<typename op, typename signature>
using is_signature = typename std::is_same<op, signature>;

int main()
{
   static_assert(
      std::experimental::is_detected<string_substr, std::string>::value,
       "detected");
   static_assert(
       std::is_function<string_substr<std::string>>::value, "function");
   static_assert(
       is_signature<string_substr<std::string>,
std::string(std::string::size_type,std::string::size_type)>::value,
      "signature");
}
```

error: static assertion failed

https://godbolt.org/z/xbrz49box

Signature of member functions

What type are we getting?

```
// Extract the type of the member function pointer
template<typename string_type>
using string_substr = decltype(&string_type::substr);


// But this is not a function?!
static_assert(
       std::is_function<string_substr<std::string>>::value,
"function");
```

Signature of member functions

If we take peek into the deduced types with *C++ Insights* help

```
template<typename string_type>
using string_substr = decltype(&string_type::substr);

int main()
{
  using MemberVarPtr_9 =
     std::basic_string<char> (std::basic_string<char>::*)
     (std::basic_string<char>::size_type, std::basic_string<char>::size_type)
     const;
  MemberVarPtr_9 not_what_we_expected_ty
  return 0;
}
```

Pointer of an instance!
Function pointer is <u>not</u> a function's signature type

https://cppinsights.io/lnk?code=I2I

Signature of member functions

Let’s try `std::declval`. Maybe matching string_type’s substr return type.

```
template<typename string_type, typename integer_type>
using is_substr_start_end_index_signature =
    typename std::is_same<
        // With declval we can get a compile time instance to call from
        decltype(std::declval<string_type>().substr(
        std::declval<integer_type>(), std::declval<integer_type>())),
    string_type>;
```

https://github.com/Thalhammer/jwt-cpp/blob/c9
01/include/jwt-cpp/jwt.h#L2053-L2057

Signature of member functions

```
template<typename string_type, typename integer_type>
using is_substr_start_end_index_signature =
    typename std::is_same<
        decltype(std::declval<string_type>().substr(
        std::declval<integer_type>(), std::declval<integ
    string_type>;

int main()
{
   static_assert(
       is_substr_start_end_index_signature<std::string,
                                    std::string::size_t pe  :value,
       "missing `string_type:: substr()`");
}
```

Compiled Successfully!

https://godbolt.org/z/8GvTje5rG

What if the member function is not implemented?
What if there are two different methods that need to be supported?

SFINAE Member Function Signatures

```
namespace best_api_json_library {
    class object_map{
        // ...
        value operator[](string
key);
    }
}
```

```
namespace performance_json_libary {
    class ordered_object{
        // ...
        value at(sting key);
    }
}
```

We have two different methods from different JSON libraries for accessing keys.

Signature of member functions (recall)

```
template<typename string_type, typename integer_type>
using is_substr_start_end_index_signature =
    typename std::is_same<
        decltype(std::declval<string_type>().substr(
        std::declval<integer_type>(), std::declval<integer_t
    string_type>;

int main()
{
   static_assert(
       is_substr_start_end_index_signature<std::string,
                                           std::string::size         ue,
       "missing `string_type:: substr()`");
}
```

Compiled Successfully!

https://godbolt.org/z/8GvTje5rG

Signature of member functions

```
struct basic_string{};
using integer = unsigned long;

int main()
{
   static_assert(
       is_substr_start_end_index_signature<basic_string,
       "missing `string_type:: substr()`");
}
```

Compilation Failure!

```
error: 'struct basic_string' has no member named 'substr'
x86-64 gcc 12.2 #1
```

https://godbolt.org/z/8s1MGPaa1

Signature of member functions

```
template<typename string_type, typename integer_type>
using is_substr_start_end_index_signature =
    typename std::is_same<
        decltype(std::declval<string_type>().substr(
        std::declval<integer_type>(), std::declval<integer_type>())),
    string_type>;
```

Generates an instance during compile time

Signature of member functions

```
template<typename string_type, typename integer_type>
using is_substr_start_end_index_signature =
    typename std::is_same<
        decltype(std::declval<string_type>().substr(
        std::declval<integer_type>(), std::declval<integer_type>())),
    string_type>
```

Invokes that instance's method call to get the return type

Since we are working with a specific instance and evaluating it.

There’s no substitution

SFINAE for Member Function Signature

```
template<typename object_type, typename string_type>
struct has_subcription_operator {
    template<class>
    struct sfinae_true : std::true_type {};

    template<class T, class A0>
    static auto test_operator_plus(int) ->
      sfinae_true<decltype(
         std::declval<T>().operator[](std::declval<A0>()))>;
    template<class, class A0>
    static auto test_operator_plus(long) -> std::false_type;

    static constexpr auto value =
       decltype(test_operator_plus<object_type, string_type>(0)){};
};
```

https://stackoverflow.com/a/9154394/8480874

https://github.com/Thalhammer/jwt-cpp/blob/c9a511f4
1/include/jwt-cpp/jwt.h#L2002-L2014

SFINAE for Member Function Signature

Main test for the function as before

```
template<typename object_type, typename string_type>
struct has_subcription_operator {
    template<class>
    struct sfinae_true : std::true_type {};

    template<class T, class A0>
    static auto test_operator_plus(int) ->
      sfinae_true<decltype(
         std::declval<T>().operator[](std::declval<A0>()))>;
    template<class, class A0>
    static auto test_operator_plus(long) -> std::false_type;

    static constexpr auto value =
       decltype(test_operator_plus<object_type, string_type>(0)){};
};
```

https://stackoverflow.com/a/9154394/8480874

SFINAE for Member Function Signature

```
template<typename object_type, typename string_type>
struct has_subcription_operator {
    template<class>
    struct sfinae_true : std::true_type {};

    template<class T, class A0>
    static auto test_operator_plus(int) ->
      sfinae_true<decltype(
         std::declval<T>().operator[](std::declval<A0>()))>;
    template<class, class A0>
    static auto test_operator_plus(long) -> std::false_type;

    static constexpr auto value =
       decltype(test_operator_plus<object_type, string_type>(0)){};
};
```

Help to allow shorthand

Instead of decltype(void(...), std::true_type)

https://stackoverflow.com/a/9154394/8480874

SFINAE for Member Function Signature

```
template<typename object_type, typename string_type>
struct has_subcription_operator {
    template<class>
    struct sfinae_true : std::true_type {};

    template<class T, class A0>
    static auto test_operator_plus(int) ->
      sfinae_true<decltype(
         std::declval<T>().operator[](std::declval<A0>()))>;
    template<class, class A0>
    static auto test_operator_plus(long) -> std::false_type;

    static constexpr auto value =
       decltype(test_operator_plus<object_type, string_type>(0)){};
};
```

`0` is a better match to `int` so we can marking this one as prefered.

https://stackoverflow.com/a/9154394/8480874

SFINAE for Member Function Signature

```
template<typename object_type, typename string_type>
struct has_subcription_operator {
    template<class>
    struct sfinae_true : std::true_type {};

    template<class T, class A0>
    static auto test_operator_plus(int) ->
      sfinae_true<decltype(
         std::declval<T>().operator[](std::declval<A0>()))>;
    template<class, class A0>
    static auto test_operator_plus(long) -> std::false_type;

    static constexpr auto value =
       decltype(test_operator_plus<object_type, string_type>(0)){};
};
```

Will resolve to either `true_type` of `false_type`
if the member function is present

https://stackoverflow.com/a/9154394/8480874

Human readable errors

Using static asserts

Limitations and Drawbacks

- Does not support “any” JSON library
    - For example parsing optimized read only libraries don't support “create” side
    - Requires inline value construction – very difficult to type trait a library with a separate builder class (i.e RapidJSON’s Document class)

Review

- Check static function signatures with `is_detected`, `is_function`, and `is_same`
- We can resolve overloaded functions with the help of `declval`
- To overcome `declval`'s lack of substitution we can add template helpers to return `true_type` of `false_type` is it does not resolve.
    - More indirection is usually the answer with SFINAE