JSON в C++:

проектируем тип для работы с JSON значениями

Павел Новиков

@cpp_ape

Plan for this talk

- quick overview of JSON
- design for a C++ type for working with JSON values
  - standard containers, `std::variant`
- C++ allocator support

Constraints for this talk

Constraints for the implementation:

- use C++17
- write as little code as possible
  (while maintaining reasonable design and performance)

Not in this talk:

- JSON string escaping
- serialization/stringification
- parsing

Douglas Crockford specified JSON in the early 2000s.

RFC 8259

`null`	null	primitive types or scalar types
`true` or `false`	Boolean	
`3.14`	number	
`"hello"`	string	
`[ 1, 2, 3 ]`	array (sequence)	structured types or collection types
`{ "key": "value" }`	object (dictionary)	

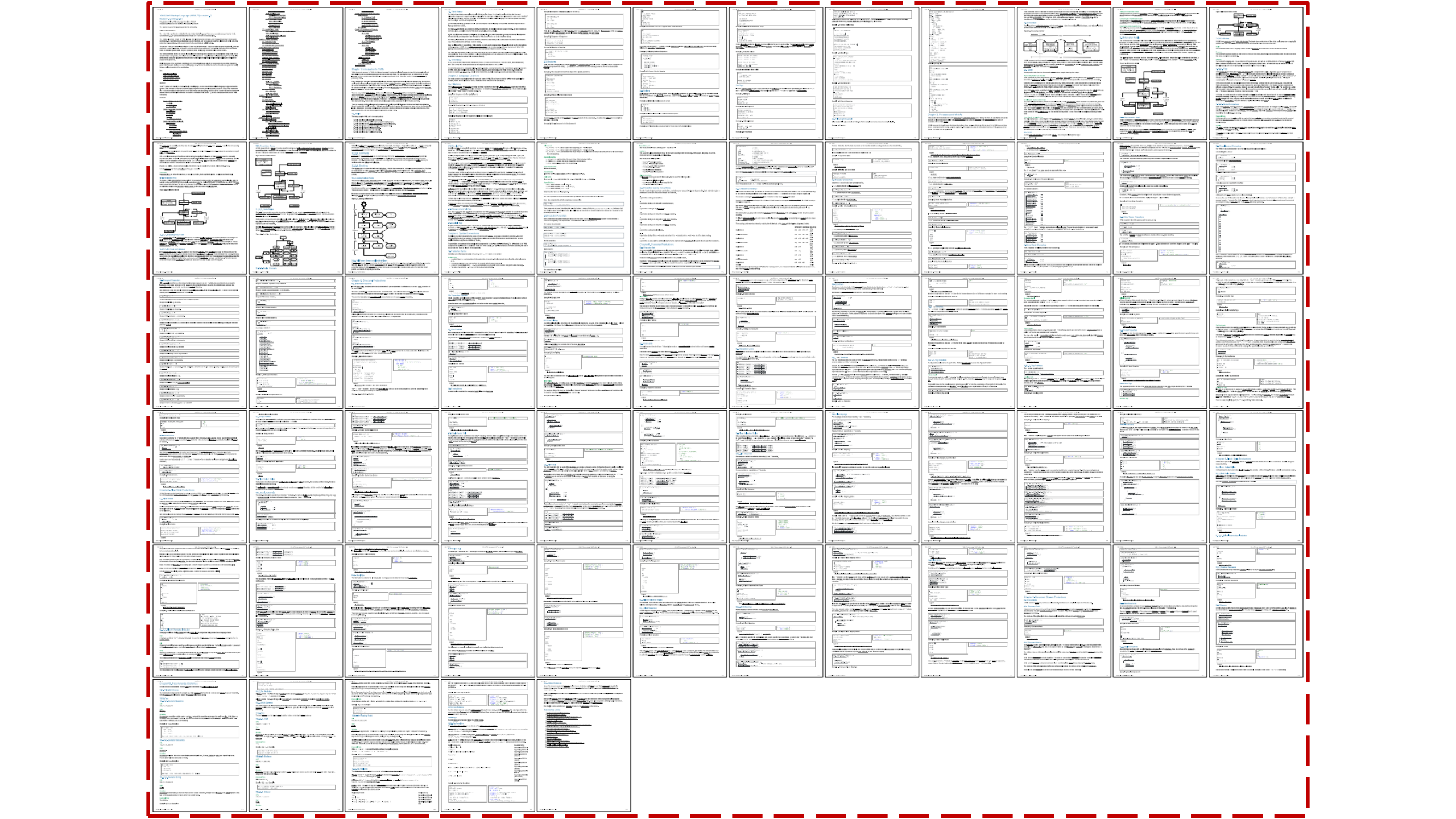
Overview of JSON

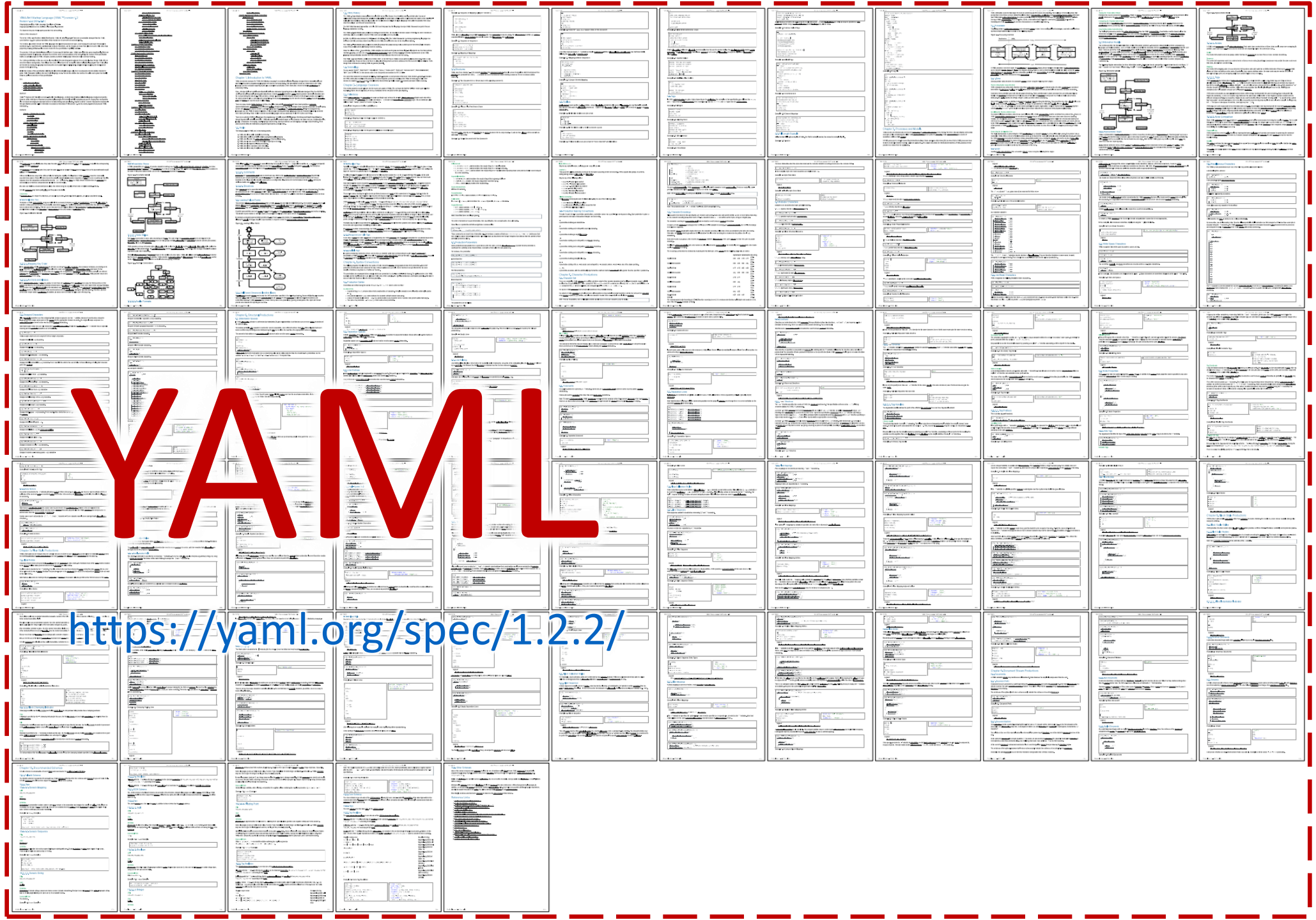
JSON grammar defined by RFC 8259

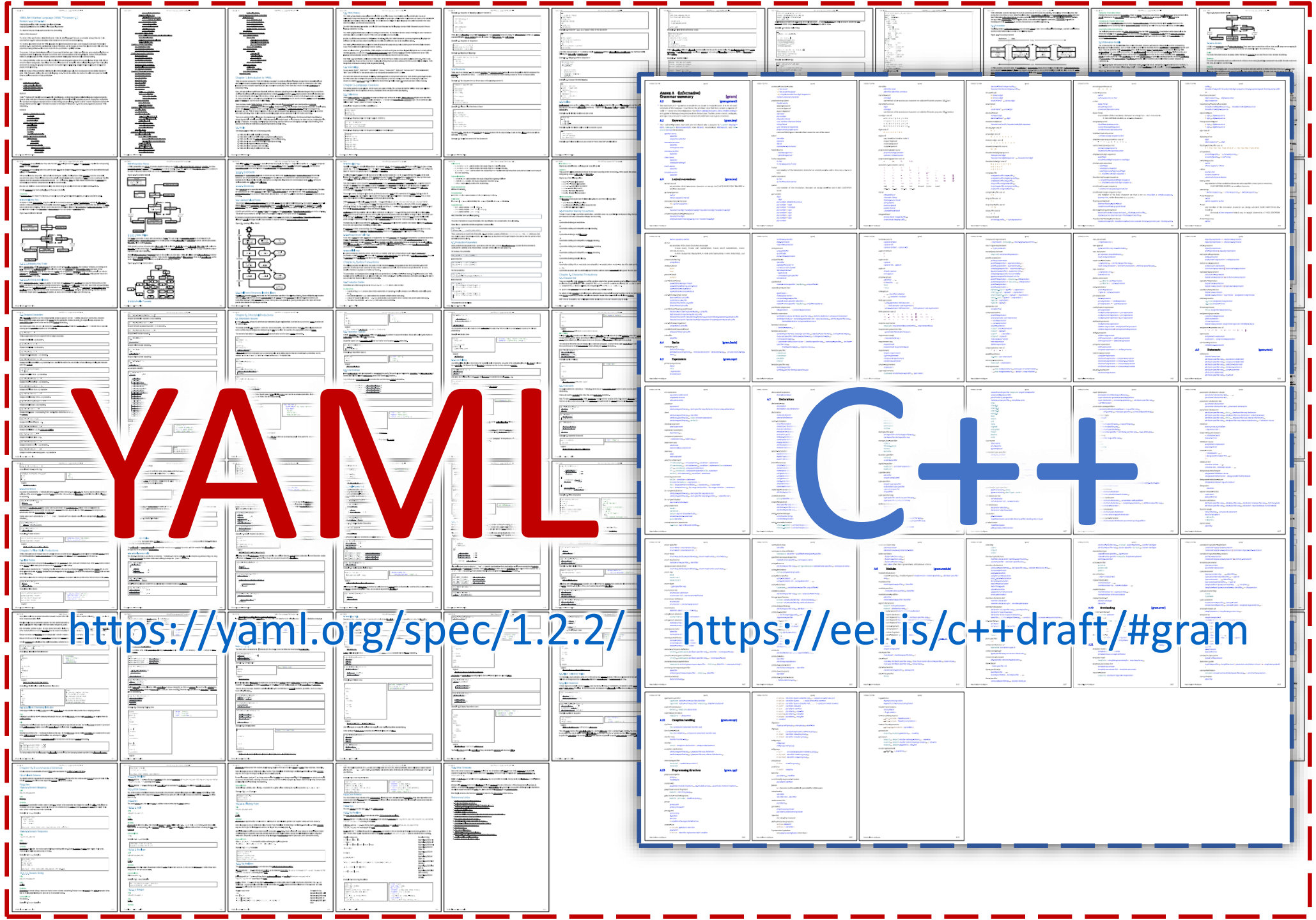
```
JSON-text = ws value ws
begin-array     = ws %x5B ws  ; [ left square bracket
begin-object    = ws %x7B ws  ; { left curly bracket
end-array       = ws %x5D ws  ; ] right square bracket
end-object      = ws %x7D ws  ; } right curly bracket
name-separator  = ws %x3A ws  ; : colon
value-separator = ws %x2C ws  ; , comma
ws = *(
        %x20 /              ; Space
        %x09 /              ; Horizontal tab
        %x0A /              ; Line feed or New line
        %x0D )              ; Carriage return
value = false / null / true / object / array / number / string
false = %x66.61.6c.73.65   ; false
null  = %x6e.75.6c.6c      ; null
true  = %x74.72.75.65      ; true
object = begin-object [ member *( value-separator member ) ]
         end-object
member = string name-separator value
array = begin-array [ value *( value-separator value ) ] end-array
```

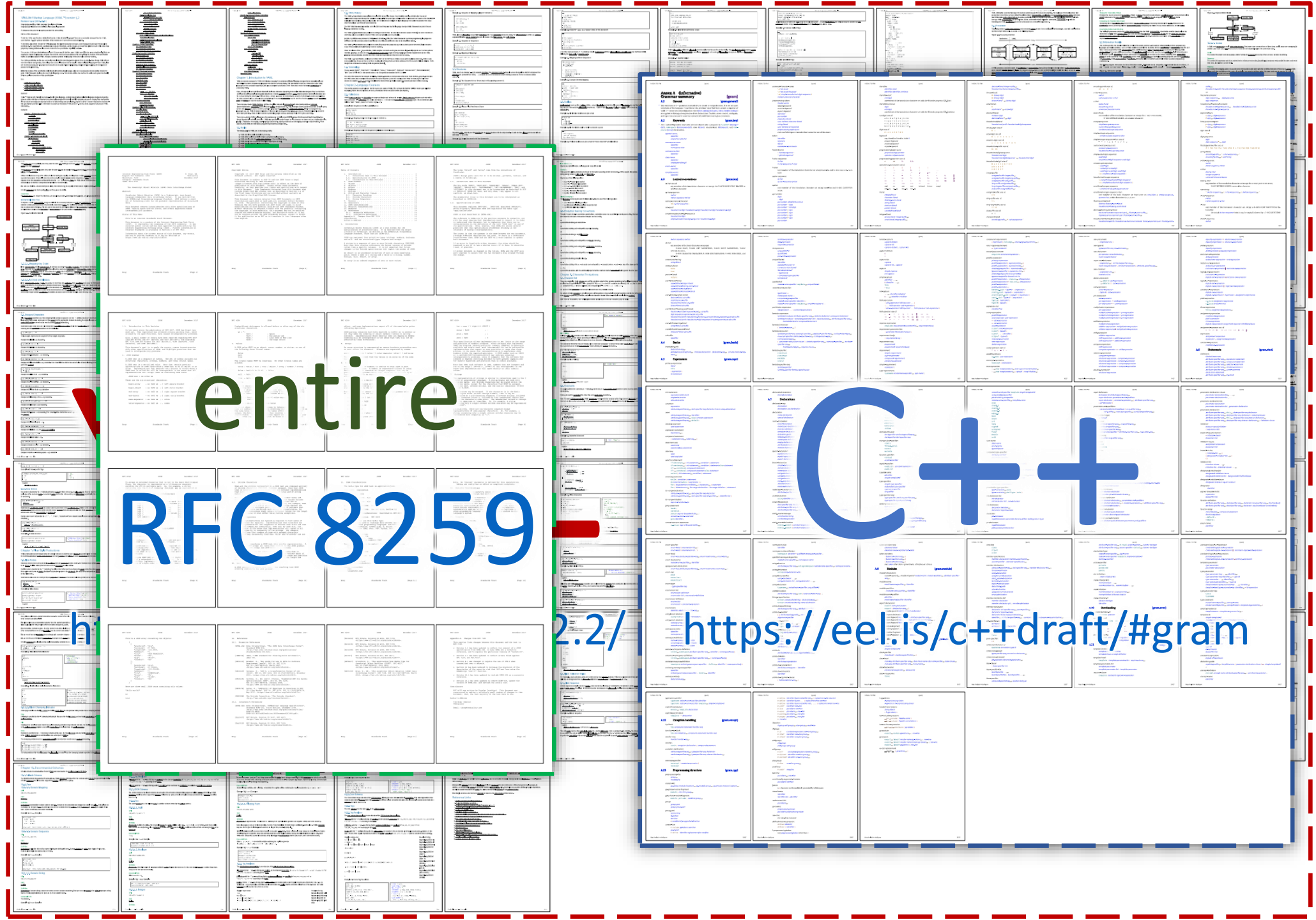
```
number = [ minus ] int [ frac ] [ exp ]
decimal-point = %x2E         ; .
digit1-9 = %x31-39           ; 1-9
e = %x65 / %x45              ; e E
exp = e [ minus / plus ] 1*DIGIT
frac = decimal-point 1*DIGIT
int = zero / ( digit1-9 *DIGIT )
minus = %x2D                 ; -
plus = %x2B                  ; +
zero = %x30                  ; 0
string = quotation-mark *char quotation-mark
char = unescaped /
    escape (
        %x22 /          ; "    quotation mark  U+0022
        %x5C /          ; \    reverse solidus U+005C
        %x2F /          ; /    solidus         U+002F
        %x62 /          ; b    backspace       U+0008
        %x66 /          ; f    form feed       U+000C
        %x6E /          ; n    line feed       U+000A
        %x72 /          ; r    carriage return U+000D
        %x74 /          ; t    tab             U+0009
        %x75 4HEXDIG )  ; uXXXX                U+XXXX
escape = %x5C                ; \
quotation-mark = %x22        ; "
unescaped = %x20-21 / %x23-5B / %x5D-10FFFF
```

Overview of JSON




YAML
https://yaml.org/spec/1.2.2/


YAML
C++
https://yaml.org/spec/1.2.2/
https://eel.is/c++draft/#gram


entire
RFC 8259
C++
https://eel.is/c++draft/#gram

JSON		C++
null	`null`	
Boolean	`true` or `false`	`bool`
number	`3.14`	`int64_t, double`
string	`"hello"`	`std::string`
array	`[ 1, 2, 3 ]`	`std::vector<Value>`
object	`{ "key": "value" }`	`std::unordered_map<std::string, Value>`

keys are unordered

Mapping of JSON types into C++

C++17:

`char` — ~~(usually)~~ 8 bit integer type

```
namespace std {
  using string = basic_string<char, char_traits<char>, allocator<char>>;
}
```

C++20:

`char8_t` — (at least) 8 bit integer type able to accommodate UTF-8 code units

```
namespace std {
  using u8string = basic_string<char8_t, char_traits<char8_t>, allocator<char8_t>>;
}
```

What about interoperability between `std::string` and `std::u8string`? 🤷

Mapping of JSON types into C++

JSON value is a union type

```
std::variant<std::monostate,
             bool,
             int64_t,
             double,
             std::string,
             std::vector<Value>,
             std::unordered_map<String, Value>>
```

null

```
struct Value;
```

Mapping of JSON types into C++

```
struct Value {
  using Null    = std::monostate;
  using Boolean = bool;
  using String  = std::string;
  using Array   = std::vector<Value>;
  using Object  = std::unordered_map<String, Value>;
  using Variant =
    std::variant<Null, Boolean, int64_t, double, String, Array, Object>;
  //...
private:
  //...
  Variant m_data;
};
```

Value type

```
Value v; // default ctor

Value boolean = true;
Value string = "hello";
Value number = 42;
```

`Value` type

```
struct Value {
  //...
  Value() = default;

  template<typename T,
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, int> = 0>
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>) :
    m_data{ std::forward<T>(v) }
  {}

  Value(std::string_view s) :
    m_data{ std::in_place_type<String>, s }
  {}

  //...
};
```

```
struct Value {
  //...
  Value() = default;

  template<typename T,
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, int> = 0>
  Val                                pt(std::is_nothrow_constructible_v<Variant, T&&>) :
    m                                ard<T>(v) }
  {}

  Value(std::string_view s) :
    m_data{ std::in_place_type<String>, s }
  {}

  //...
};
```

safe to put into template paremeters

```
struct Value {
  namespace detail {
    template<typename T>
    using RemoveCVRef = std::remove_cv_t<std::remove_reference_t<T>>;
  }

  template<typename T,
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, int> = 0>
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>) :
    m_data{ std::forward<T>(v) }
  {}

  Value(std::string_view s) :
    m_data{ std::in_place_type<String>, s }
  {}

  //...
};
```

```
struct Value {
  //...
  Value() = default;

  template<typename T>
    requires          (!std::is_same_v<  std::remove_cvref<T>, Value> &&
                        std::is_convertible_v<T&&, Variant>)
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>) :
    m_data{ std::forward<T>(v) }
  {}

  Value(std::string_view s) :
    m_data{ std::in_place_type<String>, s }
  {}

  //...
};
```

```
struct Value {
  //...
  Value() = default;

  template<typename T>
    requires          (!std::is_same_v<  std::remove_cvref<T>, Value> &&
                        std::is_convertible_v<T&&, Variant>)
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>) :
    m_data{ std::forward<T>(v) }
  {}

  Value(std::string_view s) :
    m_data{ std::in_place_type<String>, s }
  {}

  //...
};
```

```
struct Value {
  //...
  Value() = default;

  template<typename T>
    requires          (!std::is_same_v<  std::remove_cvref<T>, Value> &&
                        std::is_convertible_v<T&&, Variant>)
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>) :
    m_data{ std::forward<T>(v) }
  {}

  Value(std::string_view s) :
    m_data{ std::in_place_type<String>, s }
  {}

  //...
};
```

```
struct Value {
  //...
  template<typename T>
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, Value>
  Value &operator=(T &&v)
    noexcept(std::is_nothrow_assignable_v<Variant, T&&>) {
    variant() = std::forward<T>(v);
    return *this;
  }

  Value &operator=(std::string_view s) {
    variant().template emplace<String>(s);
    return *this;
  }
  //...
};
```

need to be
the return type

```
struct Value {
  //...
  template<typename T,
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, int> = 0>
  Value &operator=(T &&v)
    noexcept(std::is_nothrow_assignable_v<Variant, T&&>);
  //...
};

Value v;
v.operator=<bool, 0>(true);
v.operator=<bool, 1>(true);
v.operator=<bool, 2>(true);
v.operator=<bool, 3>(true);
v.operator=<bool, 4>(true);
```

```
struct Value {
  //...
  template<typename T>
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, Value>
        &operator=(T &&v)
    noexcept(std::is_nothrow_assignable_v<Variant, T&&>) {
    variant() = std::forward<T>(v);
    return *this;
  }

  Value &operator=(std::string_view s) {
    variant().template emplace<String>(s);
    return *this;
  }
  //...
};
```

```
struct Value {
  //...
  template<typename T>
    requires          (!std::is_same_v<  std::remove_cvref<T>, Value> &&
                        std::is_convertible_v<T&&, Variant>)
  Value &operator=(T &&v)
    noexcept(std::is_nothrow_assignable_v<Variant, T&&>) {
    variant() = std::forward<T>(v);
    return *this;
  }

  Value &operator=(std::string_view s) {
    variant().template emplace<String>(s);
    return *this;
  }
  //...
};
```

```
struct Value {
  //...
  template<typename T>
    requires          (!std::is_same_v<  std::remove_cvref<T>, Value> &&
                        std::is_convertible_v<T&&, Variant>)
  Value &operator=(T &&v)
    noexcept(std::is_nothrow_assignable_v<Variant, T&&>) {
    variant() = std::forward<T>(v);
    return *this;
  }

  Value &operator=(std::string_view s) {
    variant().template emplace<String>(s);
    return *this;
  }
  //...
};
```

```
struct Value {
  //...
  template<typename T>
    requires          (!std::is_same_v<  std::remove_cvref<T>, Value> &&
                        std::is_convertible_v<T&&, Variant>)
  Value &operator=(T &&v)
    noexcept(std::is_nothrow_assignable_v<Variant, T&&>) {
    variant() = std::forward<T>(v);
    return *this;
  }

  Value &operator=(std::string_view s) {
    variant().template emplace<String>(s);
    return *this;
  }
  //...
};
```

```
struct Value {
  //...
  Value() = default;
  template<typename T,
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>>
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>);
  Value(std::string_view s);
  template<typename T>
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, Value>
  Value &operator=(T &&v);
  Value &operator=(std::string_view s);



  //...
};
```

Which rule?

rule of three

rule of five

rule of zero

```
struct Value {
  //...
  Value() = default;
  template<typename T,
    std::enable_if_t<!std::is_same_v<
                     std::is_converti
  Value(T &&v) noexcept(std::is_nothr
  Value(std::string_view s);
  template<typename T>
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, Value>
  Value &operator=(T &&v);
  Value &operator=(std::string_view s);


  //...
};
```

Which rule?

- destructor
- copy constructor
- copy assignment

→ rule of three

- move constructor
- move assignment

→ rule of five

- none of the above

→ rule of zero

```
struct Value {
  //...
  Value() = default;
  template<typename T,
    std::enable_if_t<!std::is_same_v<
                     std::is_converti
  Value(T &&v) noexcept(std::is_nothr
  Value(std::string_view s);
  template<typename T>
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, Value>
  Value &operator=(T &&v);
  Value &operator=(std::string_view s);


  //...
};
```

Which rule?

- destructor
- copy constructor
- copy assignment

→ rule of three

- move constructor
- move assignment

→ rule of five

- none of the above

→ rule of zero

```
struct Value {
  //...
  friend bool operator==(const Value &a, const Value &b) {
    return a.variant() == b.variant();
  }
  friend bool operator!=(const Value &a, const Value &b) {
    return a.variant() != b.variant();
  }

  //...
};
```

```
struct Value {
  //...
  friend bool operator==(const Value&, const Value&) = default;

  //...
};
```

`operator!=` is synthesized from `operator==`

```
struct Value {
  //...
  [[nodiscard]] const Variant &variant() const& noexcept { return m_data; }
  [[nodiscard]] Variant &variant() & noexcept { return m_data; }
  [[nodiscard]] Variant &&variant() && noexcept {
    return std::move(m_data);
  }
  [[nodiscard]] const Variant &&variant() const&& noexcept {
    return std::move(m_data);
  }
  operator const Variant&() const& noexcept { return m_data; }
  operator Variant&() & noexcept { return m_data; }
  operator Variant&&() && noexcept { return std::move(m_data); }
  operator const Variant&&() const&& noexcept { return std::move(m_data); }

  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] const Variant &variant() const& noexcept { return m_data; }
  [[nodiscard]] Variant &variant() & noexcept { return m_data; }
  [[nodiscard]] Variant &&variant() && noexcept {
    return std::move(m_data);
  }
  [[nodiscard]] const Variant &&variant() const&& noexcept {
    return std::move(m_data);
  }
  operator const Variant&() const& noexcept { return m_data; }
  operator Variant&() & noexcept { return m_data; }
  operator Variant&&() && noexcept { return std::move(m_data); }
  operator const Variant&&() const&& noexcept { return std::move(m_data); }

  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] const Variant &variant() const& noexcept { return m_data; }
  [[nodiscard]] Variant &variant() & noexcept { return m_data; }
  [[nodiscard]] Variant &&variant() && noexcept {
    return std::move(m_data);
  }
  [[nodiscard]] const Variant &&variant() const&& noexcept {
    return std::move(m_data);
  }
  operator const Variant&() const& noexcept { return m_data; }
  operator Variant&() & noexcept { return m_data; }
  operator Variant&&() && noexcept { return std::move(m_data); }
  operator const Variant&&() const&& noexcept { return std::move(m_data); }

  //...
};
```

```
using Object  = Value::Object;
using Array   = Value::Array;
using String  = Value::String;
using Boolean = bool;
using Null    = std::monostate;
using Variant = Value::Variant;
```

`Value` type

```
void print(const Value &value) {
  std::visit(Overloaded{
    [](Null) { std::cout << "null"; },
    [](bool b) { std::cout << "boolean: " << (b ? "true" : "false"); },
    [](int64_t i) { std::cout << "integer: " << i; },
    [](double d) { std::cout << "decimal: " << d; },
    [](const std::string &s) { std::cout << "string: " << s; },
    [](const Array&) { std::cout << "array"; },
    [](const Object&) { std::cout << "object"; }
  }, value.variant());
  std::cout << '\n';
}
```

`Value` type

```
void print(const Value &value) {
  std::visit(Overloaded{
    [](Null) { std::cout << "null"; },
    [](bool b) { std::cout << "boolean: " << (b ? "true" : "false"); },
    [](
    [](
    [](
    [](const Array&) { std::cout << "array"; },
    [](const Object&) { std::cout << "object"; }
  }, value.variant());
  std::cout << '\n';
}
```

```
template<typename... F>
struct Overloaded : F... { using F::operator()...; };
template<typename... F>
Overloaded(F...) -> Overloaded<F...>; // needed until C++20
```

`Value` type

```
void print(const Value &value) {
  std::visit(Overloaded{
    [](Null) { std::cout << "null"; },
    [](bool b) { std::cout << "boolean: " << (b ? "true" : "false"); },
    [](int64_t i) { std::cout << "integer: " << i; },
    [](double d) { std::cout << "decimal: " << d; },
    [](const std::string &s) { std::cout << "string: " << s; },
    [](const Array&) { std::cout << "array"; },
    [](const Object&) { std::cout << "object"; }
  }, value.variant());
  std::cout << '\n';
}
```

`Value` type

```
Value boolean = true;
print(boolean);
```

output:

Clang 18 -std=c++17

```
boolean: true
```

MSVC (VS 2022) -std:c++17

```
boolean: true
```

`Value` type

```
Value string = "hello";
print(string);
```

output:

Clang 18 -std=c++17

```
string: hello
```

MSVC (VS 2022) -std:c++17

```
boolean: true
```

`Value` type

```
Value number = 42;
print(number);
```

output:

Clang 18 -std=c++17

```
integer: 42
```

MSVC (VS 2022) -std:c++17

error : no viable conversion from 'int' to 'Value'

`Value` type

In C++17 variant alternative is chosen as if by overload resolution:

```
void test(std::monostate); // Null
void test(bool); // Boolean
void test(int64_t); // integer
void test(double); // decimal
void test(std::string); // String
void test(std::vector<Value>); // Array
void test(std::unordered_map<String, Value>) // Object

test(true);    // calls test(bool)
test("hello"); // calls test(bool)
test(42);      // error : call to 'test' is ambiguous
```

This nonsense is fixed in P0608.
MS left the original behavior in -std=c++17 mode
for backwards compatibility.

```
#if defined(_MSC_VER) && \
    (defined(_MSVC_LANG) && _MSVC_LANG > __cplusplus && \
     _MSVC_LANG == 201703L || \
     __cplusplus == 201703L)
#define NEED_WORKAROUND_FOR_UNIMPLEMENTED_P0608
#endif
```

`Value` type

```
struct Value {
  //...
  Value() = default;
  template<typename T,
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>, int> = 0>
  Value(T &&v) noexcept(std::is_nothrow_constructible_v<Variant, T&&>);
  Value(std::string_view s);
#if defined(NEED_WORKAROUND_FOR_UNIMPLEMENTED_P0608)
  Value(const char *s) : m_data{ std::in_place_type<String>, s } {}
  Value(int i) noexcept : m_data{ std::in_place_type<int64_t>, i } {}
#endif



  //...
};
```

```
  //...
  template<typename T>
    std::enable_if_t<!std::is_same_v<detail::RemoveCVRef<T>, Value> &&
                     std::is_convertible_v<T&&, Variant>,
  Value> &operator=(T &&v);
  Value &operator=(std::string_view s);
#if defined(NEED_WORKAROUND_FOR_UNIMPLEMENTED_P0608)
  Value &operator=(const char *s) {
    variant().template emplace<String>(s);
    return *this;
  }
  Value &operator=(int i) noexcept {
    variant().template emplace<int64_t>(i);
    return *this;
  }
#endif
  //...
```

output:

```
Value boolean = true;
print(boolean);
```

```
boolean: true
```

```
Value string = "hello";
print(string);
```

```
string: hello
```

```
Value number = 42;
print(number);
```

```
integer: 42
```

`Value` type

```
using json = nlohmann::json;

json j2 = {
  {"pi", 3.141},
  {"happy", true},
  {"name", "Niels"},
  {"nothing", nullptr},
  {"answer", {
    {"everything", 42}
  }},
  {"list", {1, 0, 2}},
  {"object", {
    {"currency", "USD"},
    {"value", 42.99}
  }}
};
```

```
json x = {
  {}
};

std::cout << x.dump(2) << '\n';
```

outputs:

```
[
  null
]
```

```
json x = {
  { "value" }
};

std::cout << x.dump(2) << '\n';
```

outputs:

```
[
  [
    "value"
  ]
]
```

```
json x = {
  { "key?", "value?" }
};

std::cout << x.dump(2) << '\n';
```

outputs:

```
{
  "key?": "value?"
}
```

```
json x = {
  { "key?", "value?", "oops" }
};

std::cout << x.dump(2) << '\n';
```

outputs:

```
[
  [
    "key?",
    "value?",
    "oops"
  ]
]
```

```
Value json = Object{
  { "null", Null{} },
  { "boolean", true },
  { "integer", 42 },
  { "decimal", 3.14 },
  { "string", "hello" },
  { "array", Array{
    1, 2, 3
  } },
  { "nested object", Object{
    { "foo", "bar" }
  } }
};
```

`std::unordered_map<String, Value>`

`std::vector<Value>`

`Value` type

```
struct Value {
  //...
  [[nodiscard]] bool isObject() const noexcept {
    return std::holds_alternative<Object>(variant());
  }
  [[nodiscard]] const Object &asObject() const& {
    return std::get<Object>(variant());
  }
  [[nodiscard]] Object &asObject() & {
    return std::get<Object>(variant());
  }
  [[nodiscard]] Object &&asObject() && {
    return std::move(std::get<Object>(variant()));
  }


  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] bool isArray() const noexcept {
    return std::holds_alternative<Array>(variant());
  }
  [[nodiscard]] const Array &asArray() const& {
    return std::get<Array>(variant());
  }
  [[nodiscard]] Array &asArray() & {
    return std::get<Array>(variant());
  }
  [[nodiscard]] Array &&asArray() && {
    return std::move(std::get<Array>(variant()));
  }


  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] bool isString() const noexcept {
    return std::holds_alternative<String>(variant());
  }
  [[nodiscard]] const String &asString() const& {
    return std::get<String>(variant());
  }
  [[nodiscard]] String &asString() & {
    return std::get<String>(variant());
  }
  [[nodiscard]] String &&asString() && {
    return std::move(std::get<String>(variant()));
  }


  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] bool isDouble() const noexcept {
    return std::holds_alternative<double>(variant());
  }
  [[nodiscard]] double asDouble() const {
    return std::get<double>(variant());
  }
  [[nodiscard]] double &asDouble() {
    return std::get<double>(variant());
  }
  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] bool isInt() const noexcept {
    return std::holds_alternative<int64_t>(variant());
  }
  [[nodiscard]] int64_t asInt() const {
    return std::get<int64_t>(variant());
  }
  [[nodiscard]] int64_t &asInt() {
    return std::get<int64_t>(variant());
  }









  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] bool isBool() const noexcept {
    return std::holds_alternative<Boolean>(variant());
  }
  [[nodiscard]] Boolean asBool() const {
    return std::get<Boolean>(variant());
  }
  [[nodiscard]] Boolean &asBool() {
    return std::get<Boolean>(variant());
  }









  //...
};
```

```
struct Value {
  //...
  [[nodiscard]] bool isNull() const noexcept {
    return std::holds_alternative<Null>(variant());
  }

  //...
};
```

```
Value json = Object{
  //...
};

assert(json.isObject());
auto &object = json.asObject();

assert(object["array"].isArray());
auto &array = object["array"].asArray();
array[0]; // can do whatever std::vector allows

assert(object["nested object"].isObject());
auto &nested = object["nested object"].asObject();
nested["foo"]; // can do whatever std::unordered_map allows
```

Value type

```cpp
using json = nlohmann::json;

json j2 = {
  //...
  {"list", {1, 0, 2}},
  //...
};

j2["list"][0];
```

```
using json = nlohmann::json;

json x; // null
x[0]; // now it's an array: [ null ]

json y; // null
y["foo"]; // now it's an object: { "foo": null }

json z = 42;
z["foo"]; // compiles, throws an exception
```

```cpp
using json = nlohmann::json;

json x; // null
x[0]; // now it's an array: [ null ]

json y; // null
y["foo"]; // now it's an object: { "foo": null }

json z = 42;
z["foo"]; // compiles, throws an exception
```

```
using json = nlohmann::json;

json x; // null
x[0]; // now it's an array: [ null ]

json y; // null
y["foo"]; // now it's an object: { "foo": null }

json z = 42;
z["foo"]; // compiles, throws an exception
```

```
using json = nlohmann::json;

json x; // null
x[0]; // now it's an array: [ null

json y; // null
y["foo"]; // now it's an object: {

json z = 42;
z["foo"]; // compiles, throws an ex
```

```
Value x = Object{
   { "items", Array{ 1, 2, 3 } }
};

Value *item = x.resolve("items", 0);
if (item)
   print(*item); // use *item
```

implies that `x` is an object

implies that `x.asObject()["items"]` is an array

`Value` type

```
Value y = Array{
  Object{ { "foo", 1 } }, "two", 3
};

Value *foo = y.resolve(0, "foo");
if (foo)
  print(*foo); // use *foo
```

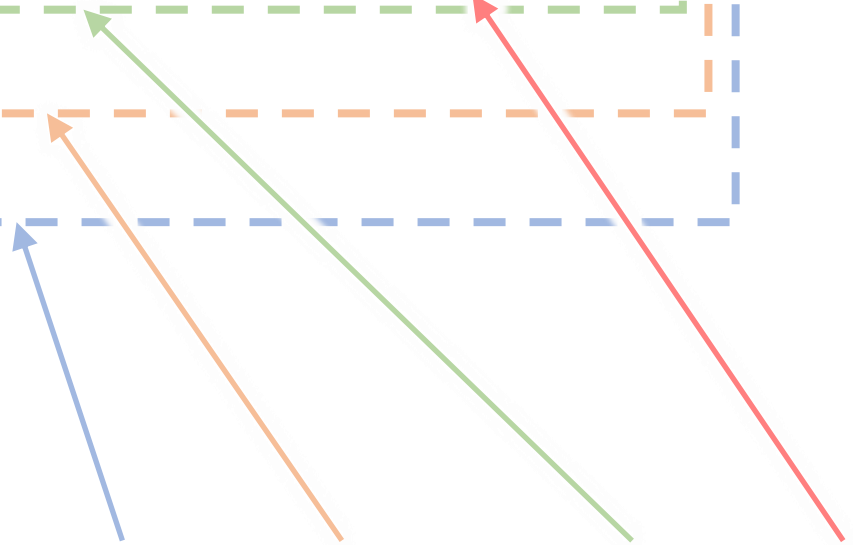
implies that `y` is an array

implies that `y.asArray()[0]` is an object

`Value` type

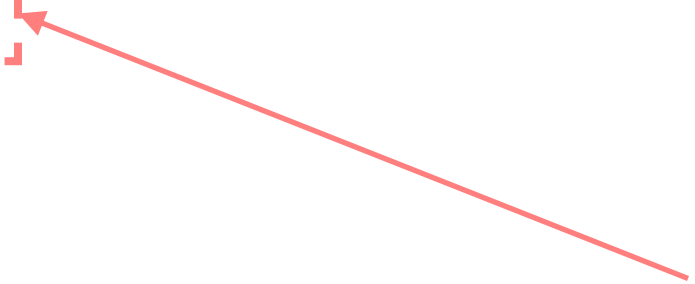
```
Value z = Array{
  Object{
    { "foo", Object{
      { "bar", Array{ 1, 2, 3 } }
    } }
  },
  42
};

auto *v = z.resolve(0, "foo", "bar", 2);
if (v)
  print(*v); // use *v
```

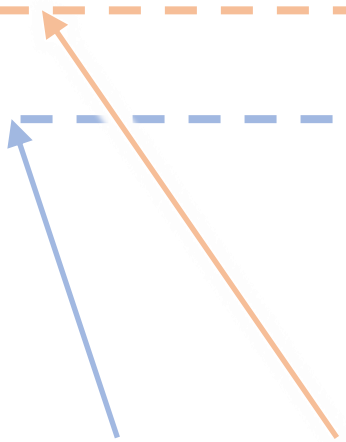


```
Value z = Array{
  Object{
    { "foo", Object{
      { "bar", Array{ 1, 2, 3 } }
    } }
  },
  42
};

auto *v = z.resolve(1);
if (v)
  print(*v); // use *v
```



```
Value z = Array{
  Object{
    { "foo", Object{
      { "bar", Array{ 1, 2, 3 } }
    } }
  },
  42
};
```



???

```
auto *v = z.resolve(0, "foo", 2);
if (v)
  print(*v); // use *v
```

```
struct Value {
  //...
private:
  const Value *resolveImpl(size_t index) const noexcept {
    auto *array = std::get_if<Array>(&variant());
    if (array && index < array->size())
      return &(*array)[index];
    return nullptr;
  }
  const Value *resolveImpl(const String &key) const noexcept {
    if (auto *object = std::get_if<Object>(&variant())) {
      if (auto i = object->find(key); i != object->end())
        return &i->second;
    }
    return nullptr;
  }
  //...
```

```
struct Value {
  //...
private:
  const Value *resolveImpl(size_t index) const noexcept {
    auto *array = std::get_if<Array>(&variant());
    if (array && index < array->size())
      return &(*array)[index];
    return nullptr;
  }
  const Value *resolveImpl(const String &key) const noexcept {
    if (auto *object = std::get_if<Object>(&variant())) {
      if (auto i = object->find(key); i != object->end())
        return &i->second;
    }
    return nullptr;
  }
  //...
```

```
struct Value {
  //...
  template<typename... T>
  [[nodiscard]] std::enable_if_t<sizeof...(T) >= 1 &&
                  detail::AreTypesConvertibleToStringXorSizeT<T&&...>,
                const Value*> resolve(T&&... refTokens) const noexcept {
    const Value *value = this;
    (value = value->resolveImpl(std::forward<T>(refTokens)) && ...);
    return value;
  }


  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (                          sizeof...(T) >= 1 &&
                    detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] const Value*  resolve(T&&... refTokens) const noexcept {
    const Value *value = this;
    ((value = value->resolveImpl(std::forward<T>(refTokens))) && ...);
    return value;
  }


  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (                    sizeof...(T) >= 1 &&
                  detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [                                                          cept {

  //...
};
```

```
namespace detail {
  template<typename... T>
  inline constexpr bool AreTypesConvertibleToStringXorSizeT =
    ((std::is_convertible_v<T, std::string> != // exclusive OR
      std::is_convertible_v<T, size_t>) && ...);
}}
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (                    sizeof...(T) >= 1 &&
                  detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] const Value*  resolve(T&&... refTokens) const noexcept {
    const Value *value = this;
    ((value = value->resolveImpl(std::forward<T>(refTokens))) && ...);
    return value;
  }


  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (                    sizeof...(T) >= 1 &&
                  detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] const Value*  resolve(T&&... refTokens) const noexcept {
    const Value *value = this;
    ((value = value->resolveImpl(std::forward<T>(refTokens))) && ...);
    return value;
  }


  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (                    sizeof...(T) >= 1 &&
                  detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] const Value*  resolve(T&&... refTokens) const noexcept {
    const Value *value = this;
    (static_cast<bool>(
       value = value->resolveImpl(std::forward<T>(refTokens))
     ) && ...);
    return value;
  }
  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (                    sizeof...(T) >= 1 &&
                  detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] const Value*  resolve(T&&... refTokens) const noexcept {
    const Value *value = this;
    (void)(static_cast<bool>(
       value = value->resolveImpl(std::forward<T>(refTokens))
     ) && ...);
    return value;
  }
  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (sizeof...(T) >= 1 &&
              detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] Value* resolve(T&&... refTokens) noexcept {
    return const_cast<Value*>(
      std::as_const(*this).resolve(std::forward<T>(refTokens)...)
    );
  }


  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (sizeof...(T) >= 1 &&
              detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] Value* resolve(T&&... refTokens) noexcept {
    return const_cast<Value*>(
      std::as_const(*this).resolve(std::forward<T>(refTokens)...)
    );
  }


  //...
};
```

`Value` type

```
struct Value {
  //...
  template<typename... T>
    requires (sizeof...(T) >= 1 &&
              detail::AreTypesConvertibleToStringXorSizeT<T&&...>)
  [[nodiscard]] Value* resolve(T&&... refTokens) noexcept {
    return const_cast<Value*>(
      std::as_const(*this).resolve(std::forward<T>(refTokens)...)
    );
  }

  //...
};
```

```
Value x = Object{
  { "items", Array{ 1, 2, 3 } }
};

Value *item = x.resolve("items", 0);
if (item)
  print(*item); // use *item
```

Value type

```
struct Value {
  //...
  using Array = std::vector<Value>;
  using Object = std::unordered_map<String, Value>;
  //...
private:
  //...
  Variant m_data;
};
```

`Value` is incomplete at this point

`Value` type

`std::vector` has to support incomplete types according to the standard

`std::unordered_map` does not *have to* support incomplete types, but de facto supports them in all major implementations of the C++ standard library

except for libstdc++ versions <12

`Value` type

```
struct Value {
  //...
private:
  //...
#if !defined(_GLIBCXX_RELEASE) || _GLIBCXX_RELEASE >= 12
  Variant m_data;
#else
  struct Workaround final {
    //...
  } m_data;
#endif
};
```

Value type

```
struct Workaround final {
  //...
private:
  using DummyUnorderedMap = std::unordered_map<String, Array>;
  using DummyVariant = std::variant<Null, Boolean, int64_t,
                                    double, String, Array,
                                    DummyUnorderedMap>;

  alignas(DummyVariant) uint8_t storage[sizeof(DummyVariant)];
} m_data;
```

`Value` type

```
struct Workaround final {
  //...
private:
  using DummyUnorderedMap = std::unordered_map<String, Array>;
  using DummyVariant = std::variant<Null, Boolean, int64_t,
                                    double, String, Array,
                                    DummyUnorderedMap>;

  alignas(DummyVariant) uint8_t storage[sizeof(DummyVariant)];
} m_data;
```

Value type

```
struct Workaround final {
  //...
private:
  using DummyUnorderedMap = std::unordered_map<String, Array>;
  using DummyVariant = std::variant<Null, Boolean, int64_t,
                                    double, String, Array,
                                    DummyUnorderedMap>;
  alignas(DummyVariant) uint8_t storage[sizeof(DummyVariant)];
} m_data;
```

`std::aligned_storage` is deprecated in C++23

`Value` type

```
struct Workaround final {
  Workaround() noexcept { ::new(&storage) Variant{}; }
  Workaround(const Workaround &other) {
    ::new(&storage) Variant{ other.operator const Variant&() };
  }
  Workaround(Workaround &&other)
    noexcept(std::is_nothrow_move_constructible_v<Variant>) {
    ::new(&storage) Variant{ std::move(other.operator Variant&()) };
  }
  template<typename... T>
  Workaround(T&&... v)
    noexcept(std::is_nothrow_constructible_v<Variant, T&&...>) {
    ::new(&storage) Variant{ std::forward<T>(v)... };
  }

  //...
} m_data;
```

```
struct Workaround final {
  Workaround() noexcept { ::new(&storage) Variant{}; }
  Workaround(const Workaround &other) {
    ::new(&storage) Variant{ other.operator const Variant&() };
  }
  Workaround(Workaround &&other)
    noexcept(std::is_nothrow_move_constructible_v<Variant>) {
    ::new(&storage) Variant{ std::move(other.operator Variant&()) };
  }
  template<typename... T>
  Workaround(T&&... v)
    noexcept(std::is_nothrow_constructible_v<Variant, T&&...>) {
    ::new(&storage) Variant{ std::forward<T>(v)... };
  }

  //...
} m_data;
```

```
struct Workaround final {
  Workaround() noexcept { ::new(&storage) Variant{}; }
  Workaround(const Workaround &other) {
    ::new(&storage) Variant{ other.operator const Variant&() };
  }
  Workaround(Workaround &&other)
    noexcept(std::is_nothrow_move_constructible_v<Variant>) {
    ::new(&storage) Variant{ std::move(other.operator Variant&()) };
  }
  template<typename... T>
  Workaround(T&&... v)
    noexcept(std::is_nothrow_constructible_v<Variant, T&&...>) {
    ::new(&storage) Variant{ std::forward<T>(v)... };
  }

  //...
} m_data;
```

```
struct Workaround final {
  Workaround() noexcept { ::new(&storage) Variant{}; }
  Workaround(const Workaround &other) {
    ::new(&storage) Variant{ other.operator const Variant&() };
  }
  Workaround(Workaround &&other)
    noexcept(std::is_nothrow_move_constructible_v<Variant>) {
    ::new(&storage) Variant{ std::move(other.operator Variant&()) };
  }
  template<typename... T>
  Workaround(T&&... v)
    noexcept(std::is_nothrow_constructible_v<Variant, T&&...>) {
    ::new(&storage) Variant{ std::forward<T>(v)... };
  }

  //...
} m_data;
```

```
struct Workaround final {
  //...
  ~Workaround() {
    static_assert(sizeof(DummyVariant) == sizeof(Variant));
    static_assert(alignof(DummyVariant) == alignof(Variant));
    operator Variant&().~Variant();
  }
  auto &operator=(const Workaround &other) {
    return operator Variant&() = other.operator const Variant&();
  }
  auto &operator=(Workaround &&other)
    noexcept(std::is_nothrow_move_assignable_v<Variant>) {
    return operator Variant&() = std::move(other.operator Variant&());
  }

  //...
} m_data;
```

```
struct Workaround final {
  //...
  ~Workaround() {
    static_assert(sizeof(DummyVariant) == sizeof(Variant));
    static_assert(alignof(DummyVariant) == alignof(Variant));
    operator Variant&().~Variant();
  }
  auto &operator=(const Workaround &other) {
    return operator Variant&() = other.operator const Variant&();
  }
  auto &operator=(Workaround &&other)
    noexcept(std::is_nothrow_move_assignable_v<Variant>) {
    return operator Variant&() = std::move(other.operator Variant&());
  }

  //...
} m_data;
```

```
struct Workaround final {
  //...
  ~Workaround() {
    static_assert(sizeof(DummyVariant) == sizeof(Variant));
    static_assert(alignof(DummyVariant) == alignof(Variant));
    operator Variant&().~Variant();
  }
  auto &operator=(const Workaround &other) {
    return operator Variant&() = other.operator const Variant&();
  }
  auto &operator=(Workaround &&other)
    noexcept(std::is_nothrow_move_assignable_v<Variant>) {
    return operator Variant&() = std::move(other.operator Variant&());
  }

  //...
} m_data;
```

Which rule?

```
struct Workaround final {
  Workaround() noexcept;
  Workaround(const Workaround &other);
  Workaround(Workaround &&other)
    noexcept(std::is_nothrow_move_constructible_v<Variant>);
  template<typename... T>
  Workaround(T&&... v)
    noexcept(std::is_nothrow_constructible_v<Variant, T&&...>);
  ~Workaround();
  auto &operator=(const Workaround &other);
  auto &operator=(Workaround &&other)
    noexcept(std::is_nothrow_move_assignable_v<Variant>);



  //...
} m_data;
```

```
struct Workaround final {
  Workaround() noexcept;
  Workaround(const Workaround &other);
  Workaround(Workaround &&other)
    noexcept(std::is_nothrow_move_constructible_v<Variant>);
  template<typename... T>
  Workaround(T&&... v)
    noexcept(std::is_nothrow_constructible_v<Variant, T&&...>);
  ~Workaround();
  auto &operator=(const Workaround &other);
  auto &operator=(Workaround &&other)
    noexcept(std::is_nothrow_move_assignable_v<Variant>);



  //...
} m_data;
```

Which rule?

Rule of five

```
struct Workaround final {
  //...
  operator const Variant&() const& noexcept {
    return *std::launder(reinterpret_cast<const Variant*>(&storage));
  }
  operator Variant&() & noexcept {
    return *std::launder(reinterpret_cast<Variant*>(&storage));
  }
  operator Variant&&() && noexcept {
    return std::move(operator Variant&());
  }
  operator const Variant&&() const&& noexcept {
    return std::move(operator const Variant&());
  }

  //...
} m_data;
```

```
struct Value {
  //...
private:
  //...
#if !defined(_GLIBCXX_RELEASE) || _GLIBCXX_RELEASE >= 12
  Variant m_data;
#else
  struct Workaround final {
    //...
  } m_data;
#endif
};
```

`Value` type

```
Value v;
auto visitor = [](auto&&) {/*...*/};

std::visit(visitor, v); // does not work

std::visit(visitor, v.variant());

visit(visitor, v);
```

visit()

```
Value v;
auto visitor = [](auto&&) {/*...*/};

std::visit(visitor, v); // does not work

std::visit(visitor, v.variant());

visit(visitor, v);
```

visit()

```
Value v;
auto visitor = [](auto&&) {/*...*/};

std::visit(visitor, v); // does not work

std::visit(visitor, v.variant());

visit(visitor, v);
```

visit()

std::visit

Defined in header <variant>

```
template< class Visitor, class... Variants >
constexpr /* see below */ visit( Visitor&& vis, Variants&&... vars );          (1) (since C++17)

template< class R, class Visitor, class... Variants >
constexpr R visit( Visitor&& vis, Variants&&... vars );                        (2) (since C++20)

template< class... Ts >
auto&& as-variant( std::variant<Ts...>& var );                                 (3) (exposition only*)

template< class... Ts >
auto&& as-variant( const std::variant<Ts...>& var );                           (4) (exposition only*)

template< class... Ts >
auto&& as-variant( std::variant<Ts...>&& var );                                (5) (exposition only*)

template< class... Ts >
auto&& as-variant( const std::variant<Ts...>&& var );                          (6) (exposition only*)
```

Applies the visitor `vis` (a *Callable* that can be called with any combination of types from variants) to the variants `vars`

Given VariantBases as `decltype(as-variant(std::forward<Variants>(vars)))...` (a pack of `sizeof...(Variants)` types):

1) Invokes `vis` as if by

```
INVOKE(std::forward<Visitor>(vis),
       std::get<indices>(std::forward<VariantBases>(vars))...)
```

where `indices` is `as-variant(vars).index()...`.

2) Invokes `vis` as if by

```
INVOKE<R>(std::forward<Visitor>(vis),
          std::get<indices>(std::forward<VariantBases>(vars))...
```

where `indices` is `as-variant(vars).index()...`.

These overloads participate in overload resolution only if every type in VariantBases is a valid type. If the expression denoted by *INVOKE* or *INVOKE<R>* (since C++20) is invalid, or the results of *INVOKE* or *INVOKE<R>* (since C++20) have different types or value categories for different `indices`, the program is ill-formed.

3-6) The exposition-only *as-variant* function templates accept a value whose type can be deduced for `std::variant<Ts...>` (i.e. either `std::variant<Ts...>` or a type derived from `std::variant<Ts...>`), and return the `std::variant` value with the same const-qualification and value category.

3,4) Returns `var`.

5,6) Returns `std::move(var)`.

Argument to `std::visit()` can be either

- `std::variant<T...>`

or

- a type derived from `std::variant<T...>`

```
template<typename F, typename T>
  requires (std::is_base_of_v<detail::RawBaseValueType<T>,
                              std::remove_cvref_t<T>>)
decltype(auto) visit(F &&f, T &&value) {
  using BaseValueType = detail::RawBaseValueType<T>;
  return std::visit(std::forward<F>(f),
                    std::forward<T>(value).BaseValueType::variant());
}
```

visit()

```
template<typename F, typename T>
  requires (std::is_base_of_v<detail::RawBaseValueType<T>,
                              std::remove_cvref_t<T>>)
decltype(auto) visit(F &&f, T &&value) {
  using BaseValueType = detail::RawBaseValueType<T>;
  return std::visit(std::forward<F>(f),
                    std::forward<T>(value).BaseValueType::variant());
}
```

visit()

```
template<typename F, typename T>
  requires (std::is_base_of_v<detail::RawBaseValueType<T>,
                              std::remove_cvref_t<T>>)
decltype(auto) visit(F &&f, T &&value) {
  using BaseValueType = detail::RawBaseValueType<T>;
  return std::visit(std::forward<F>(f),
                    std::forward<T>(value).BaseValueType::variant());
}
```

visit()

```
template<typename F, typename T>
  requires (std::is_base_of_v<detail::RawBaseValueType<T>,
                              std::remove_cvref_t<T>>)
decltype(auto) visit(F &&f, T &&value) {
  using BaseValueType = detail::RawBaseValueType<T>;
  return std::visit(std::forward<F>(f),
                    std::forward<T>(value).BaseValueType::variant());
}
```

visit()

```
namespace detail {
  void  GetValue(...);           // intentionally not implemented
  Value GetValue(const Value&); // intentionally not implemented

  template<typename T>
  using RawBaseValueType =
    decltype(GetValue(std::declval<std::remove_cvref_t<T>>()));
}
```

visit()

```
namespace detail {
  void  GetValue(...);           // intentionally not implemented
  Value GetValue(const Value&); // intentionally not implemented

  template<typename T>
  using RawBaseValueType =
    decltype(GetValue(std::declval<std::remove_cvref_t<T>>()));
}
```

visit()

```
auto foo() {
  return expr;
}

const auto &bar() {
  return expr;
}

decltype(expr) baz() {
  return expr;
}
```

decltype(auto)

```
auto foo() {
  return expr;
}

const auto &bar() {
  return expr;
}

decltype(expr) baz() {
  return expr;
}
```

returns `T` as deduced in a call `test(expr)` to template function

```
template<typename T>
void test(T);
```

decltype(auto)

```
T   foo() {
  return expr;
}

const auto &bar() {
  return expr;
}

decltype(expr) baz() {
  return expr;
}
```

returns T as deduced in a call
`test(expr)`
to template function

```
template<typename T>
void test(T);
```

decltype(auto)

```
auto foo() {
  return expr;
}

const auto &bar() {
  return expr;
}

decltype(expr) baz() {
  return expr;
}
```

returns T as deduced in a call
`test(expr)`
to template function

```
template<typename T>
void test(const T&);
```

decltype(auto)

```
auto foo() {
  return expr;
}

const  T  &bar() {
  return expr;
}

decltype(expr) baz() {
  return expr;
}
```

returns T as deduced in a call
`test(expr)`
to template function

```
template<typename T>
void test(const T&);
```

decltype(auto)

```
auto foo() {
  return expr;
}

const auto &bar() {
  return expr;
}

decltype(expr) baz() {
  return expr;
}
```

decltype(auto)

```
auto foo() {
  return expr;
}

const auto &bar() {
  return expr;
}

decltype(auto) baz() {
  return expr;
}
```

decltype(auto)

```
template<typename F, typename T>
  requires (std::is_base_of_v<detail::RawBaseValueType<T>,
                              std::remove_cvref_t<T>>)
decltype(auto) visit(F &&f, T &&value) {
  using BaseValueType = detail::RawBaseValueType<T>;
  return std::visit(std::forward<F>(f),
                    std::forward<T>(value).BaseValueType::variant());
}
```

visit()

```
struct Value {
  using Null = std::monostate;
  using Boolean = bool;
  using String = std::string;
  using Array = std::vector<Value>;
  using Object = std::unordered_map<String, Value>;
  using Variant =
    std::variant<Null, Boolean, int64_t, double, String, Array, Object>;
  //...
};
```

Allocator support

```cpp
template<typename Allocator>
struct BasicValue {
  using Null = std::monostate;
  using Boolean = bool;
  using String = std::basic_string<char, std::char_traits<char>,
    detail::ReboundAllocator<Allocator, char>>;
  using Array = std::vector<BasicValue,
    detail::ReboundAllocator<Allocator, BasicValue>>;
  using Object = std::unordered_map<String, BasicValue,
    std::hash<String>, std::equal_to<String>,
    detail::ReboundAllocator<Allocator,
                             std::pair<const String, BasicValue>>>;
  using Variant =
    std::variant<Null, Boolean, int64_t, double, String, Array, Object>;
  //...
};
```

```
namespace detail {
  template<typename Allocator, typename T>
  using ReboundAllocator =
    typename std::allocator_traits<Allocator>::template rebind_alloc<T>;
}
  using String = std::basic_string<char, std::char_traits<char>,
    detail::ReboundAllocator<Allocator, char>>;
  using Array = std::vector<BasicValue,
    detail::ReboundAllocator<Allocator, BasicValue>>;
  using Object = std::unordered_map<String, BasicValue,
    std::hash<String>, std::equal_to<String>,
    detail::ReboundAllocator<Allocator,
                             std::pair<const String, BasicValue>>>;
  using Variant =
    std::variant<Null, Boolean, int64_t, double, String, Array, Object>;
  //...
};
```

```
template<typename Allocator>
struct BasicValue {
  using Null = std::monostate;
  using Boolean = bool;
  using String = std::basic_string<char, std::char_traits<char>,
    detail::ReboundAllocator<Allocator, char>>;
  using Array = std::vector<BasicValue,
    detail::ReboundAllocator<Allocator, BasicValue>>;
  using Object = std::unordered_map<String, BasicValue,
    std::hash<String>, std::equal_to<String>,
    detail::ReboundAllocator<Allocator,
                             std::pair<const String, BasicValue>>>;
  using Variant =
    std::variant<Null, Boolean, int64_t, double, String, Array, Object>;
  //...
};
```

```
template<typename Allocator>
struct BasicValue {
  using Null = std::monostate;
  using Boolean = bool;
  using String = std::basic_string<char, std::char_traits<char>,
    detail::ReboundAllocator<Allocator, char>>;
  using Array = std::vector<BasicValue,
    detail::ReboundAllocator<Allocator, BasicValue>>;
  using Object = std::unordered_map<String, BasicValue,
    std::hash<String>, std::equal_to<String>,
    detail::ReboundAllocator<Allocator,
                             std::pair<const String, BasicValue>>>;
  using Variant =
    std::variant<Null, Boolean, int64_t, double, String, Array, Object>;
  //...
};
```

```
template<typename Allocator>
struct BasicValue {
  //...
  BasicValue(const Allocator&) noexcept {}
  BasicValue(const BasicValue &other, const Allocator &a) :
    m_data{ construct(other.variant(), a) } {}
  BasicValue(BasicValue &&other, const Allocator &a) :
    m_data{ construct(std::move(other.variant()), a) } {}
  //...
};
```

uses-allocator machinery support

Allocator support

```
template<typename V>
Variant construct(V &&other, const Allocator &a) {
  if (other.valueless_by_exception())
    return std::forward<V>(other);
  return std::visit([&a](auto &&val) {
    using U = decltype(val);
    using T = detail::RemoveCVRef<U>;
    if constexpr (std::is_same_v<T, Object> ||
                  std::is_same_v<T, Array> ||
                  std::is_same_v<T, String>) {
      return Variant{ std::in_place_type<T>, std::forward<U>(val), a };
    }
    else {
      return Variant{ std::in_place_type<T>, val };
    }
  }, std::forward<V>(other));
}
```

```
template<typename V>
Variant construct(V &&other, const Allocator &a) {
  if (other.valueless_by_exception())
    return std::forward<V>(other);
  return std::visit([&a](auto &&val) {
    using U = decltype(val);
    using T = detail::RemoveCVRef<U>;
    if constexpr (std::is_same_v<T, Object> ||
                  std::is_same_v<T, Array> ||
                  std::is_same_v<T, String>) {
      return Variant{ std::in_place_type<T>, std::forward<U>(val), a };
    }
    else {
      return Variant{ std::in_place_type<T>, val };
    }
  }, std::forward<V>(other));
}
```

```
template<typename V>
Variant construct(V &&other, const Allocator &a) {
  if (other.valueless_by_exception())
    return std::forward<V>(other);
  return std::visit([&a](auto &&val) {
    using U = decltype(val);
    using T = detail::RemoveCVRef<U>;
    if constexpr (std::is_same_v<T, Object> ||
                  std::is_same_v<T, Array> ||
                  std::is_same_v<T, String>) {
      return Variant{ std::in_place_type<T>, std::forward<U>(val), a };
    }
    else {
      return Variant{ std::in_place_type<T>, val };
    }
  }, std::forward<V>(other));
}
```

```
template<typename V>
Variant construct(V &&other, const Allocator &a) {
  if (other.valueless_by_exception())
    return std::forward<V>(other);
  return std::visit([&a](auto &&val) {
    using U = decltype(val);
    using T = detail::RemoveCVRef<U>;
    if constexpr (std::is_same_v<T, Object> ||
                  std::is_same_v<T, Array> ||
                  std::is_same_v<T, String>) {
      return Variant{ std::in_place_type<T>, std::forward<U>(val), a };
    }
    else {
      return Variant{ std::in_place_type<T>, val };
    }
  }, std::forward<V>(other));
}
```

```
template<typename V>
Variant construct(V &&other, const Allocator &a) {
  if (other.valueless_by_exception())
    return std::forward<V>(other);
  return std::visit([&a](auto &&val) {
    using U = decltype(val);
    using T = detail::RemoveCVRef<U>;
    if constexpr (std::is_same_v<T, Object> ||
                  std::is_same_v<T, Array> ||
                  std::is_same_v<T, String>) {
      return Variant{ std::in_place_type<T>, std::forward<U>(val), a };
    }
    else {
      return Variant{ std::in_place_type<T>, val };
    }
  }, std::forward<V>(other));
}
```

```
template<typename Allocator>
struct BasicValue {
  //...
};

using Value = BasicValue<std::allocator<char>>;

namespace std {
  template<typename A1, typename A2> // uses-allocator machinery support
  struct uses_allocator<BasicValue<A1>, A2> : true_type {};
}
```

Allocator support

```
struct Workaround final {
  //...
private:
  using DummyUnorderedMap = std::unordered_map<String, Array,
        std::hash<String>, std::equal_to<String>,
        detail::ReboundAllocator<Allocator,
                                 std::pair<const String, Array>>>;
  using DummyVariant = std::variant<Null, Boolean, int64_t,
                                    double, String, Array,
                                    DummyUnorderedMap>;

  alignas(DummyVariant) uint8_t storage[sizeof(DummyVariant)];
} m_data;
```

Allocator support

```
namespace detail {
  void GetValue(...);                // intentionally not implemented
  template<typename Allocator>
  BasicValue<Allocator> GetValue(const BasicValue<Allocator>&);

  template<typename T>
  using RawBaseValueType =
    decltype(GetValue(std::declval<std::remove_cvref_t<T>>()));
}
```

Allocator support

Most Malleable Memory Management Method in C++ by Björn Fahller

https://youtu.be/ptMFLSAkRj0

```
struct Foo {
  Value v;
  std::vector<int> bar;
};

const Array a;
Value c;
c = a;
```

Clang (any version) + libstdc++ version <12:

```
error: call to implicitly-deleted copy constructor of
'std::pair<const std::basic_string<char>, BasicValue<std::allocator<char>>>'
```

Allocator support

```
struct Foo {
  Value v;
  std::vector<int> bar;
};

const Array a;
Value c;
c = a;
```

https://github.com/llvm/llvm-project/issues/84487
[clang] fails to compile valid code involving `variant` and `unordered_map` with libstdc++ version <12

Clang (any version) + libstdc++ version <12:

```
error: call to implicitly-deleted copy constructor of
'std::pair<const std::basic_string<char>, BasicValue<std::allocator<char>>>'
```

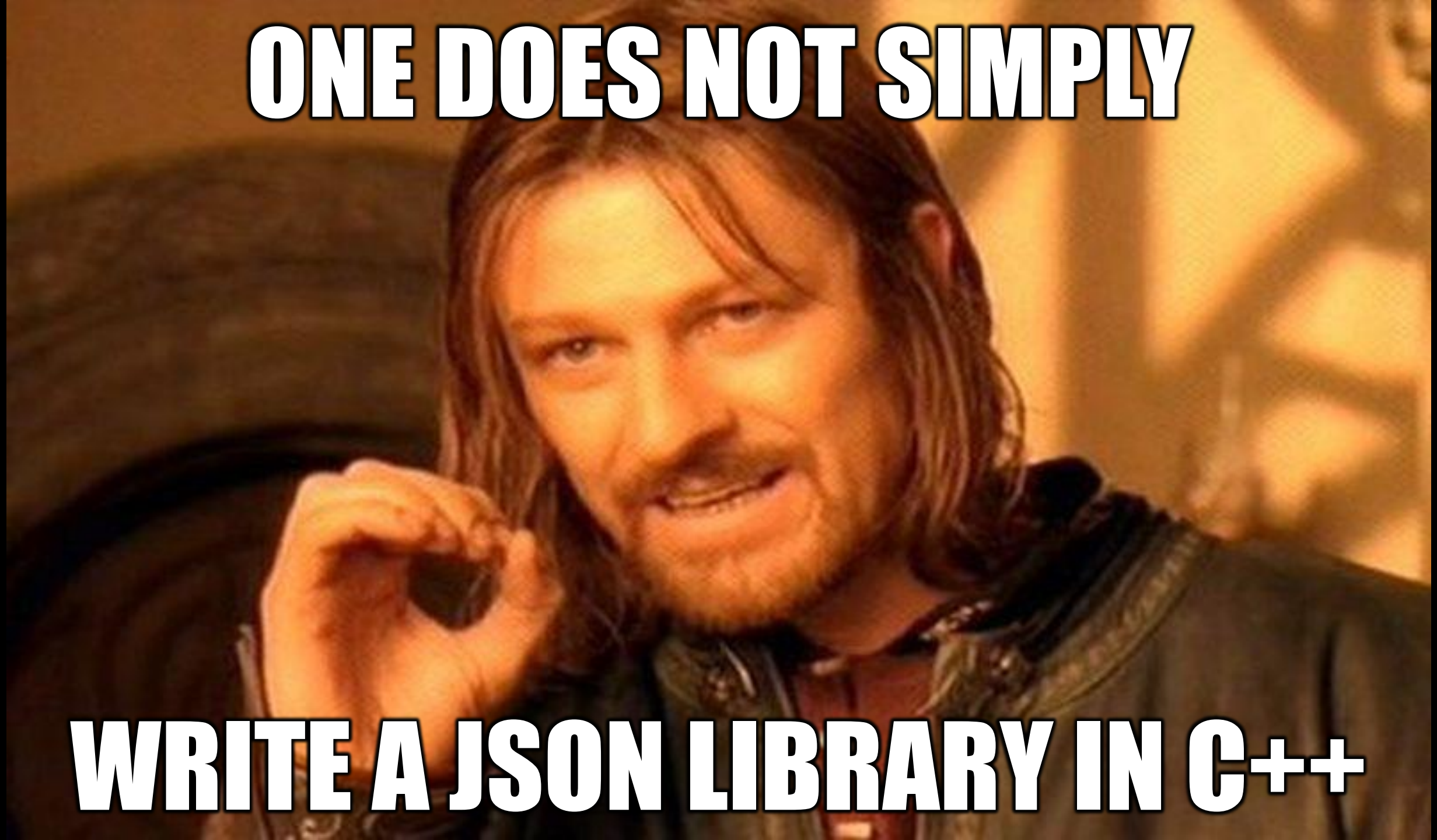
Allocator support

```
#if defined(_GLIBCXX_RELEASE) && _GLIBCXX_RELEASE < 12 && \
    defined(__clang__)
// clang needs this for some reason
static_assert(std::is_copy_constructible_v<Value>);
#endif
```

Allocator support

- implemented a variant-like value type to work with JSON values
  - rule of zero
  - worked around bogus variant construction/assignment in MS implementation (fixed within P0608)
  - implemented rich interface that's easy to use correctly and hard to use incorrectly
  - worked around inability to use incomplete types (`std::unordered_map` in libstdc++ versions <12)
    - rule of five
- allocator support, uses-allocator functionality support
  - worked around Clang compilation bug with libstdc++ versions <12

What we've covered


ONE DOES NOT SIMPLY
WRITE A JSON LIBRARY IN C++

minjsoncpp

Minimalistic JSON C++ library

https://github.com/toughengineer/minjsoncpp

JSON в C++:
проектируем тип для работы с JSON значениями

Павел Новиков

@cpp_ape

Slides: bit.ly/3WYVRkd

References

- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format https://datatracker.ietf.org/doc/html/rfc8259
- “A sane variant converting constructor” by Zhihao Yuan https://wg21.link/p0608
- Deprecate std::aligned_storage and std::aligned_union by CJ Johnson https://wg21.link/p1413
- Most Malleable Memory Management Method in C++ by Björn Fahller https://www.youtube.com/watch?v=ptMFLSAkRj0
- [clang] fails to compile valid code involving `variant` and `unordered_map` with libstdc++ version <12 https://github.com/llvm/llvm-project/issues/84487
- minjsoncpp — Minimalistic JSON C++ library https://github.com/toughengineer/minjsoncpp