JSON in C++: serialization

Pavel Novikov

𝕏 @cpp_ape

- importance of validation of UTF-8 during string escaping
- **serialization/stringification**

Plan for this talk

Constraints for the implementation:

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

Not in this talk:

- design of a C++ type for working with JSON values
- string escaping
- parsing

Constraints for this talk

```
text with newline↵
 and "quotation marks"
```

```
text with newline\n and \"quotation marks\"
```

JSON string escaping

RFC 8259 states:

8. String and Character Issues

8.1. Character Encoding

JSON text exchanged between systems that are not part of a closed ecosystem **MUST be encoded using UTF-8** [RFC3629].

JSON string escaping

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`? 🤷

Considerations for C++ string type

```
enum class Escape {
  Default,
  NonAscii
};

enum class Utf8Validation {
  IgnoreInvalidUtf8CodeUnits,
  FailOnInvalidUtf8CodeUnits
};
```

JSON string escaping

```
namespace impl {
  template<typename Sink>
  size_t escape(Sink&&,
                const std::string_view&,
                Escape,
                Utf8Validation);
}

template<typename String_t = std::string>
[[nodiscard]] String_t escape(const std::string_view&,
                              Escape = {},
                              Utf8Validation = {});
```

JSON string escaping

```cpp
template<typename String_t = std::string>
[[nodiscard]] String_t escape(const std::string_view &s,
                              Escape escapeMode = {},
                              Utf8Validation validation = {}) {
  if (!s.empty()) {
    String_t res;
    if (validation == Utf8Validation::IgnoreInvalidUtf8CodeUnits)
      res.reserve(s.size());
    using Sink = detail::StringSink<String_t>;
    if (impl::escape(Sink{ res }, s, escapeMode, validation) == s.size())
      return res;
  }
  return {};
}
```

```cpp
template<typename String_t = std::string>
[[nodiscard]] String_t escape(const std::string_view &s,
                              Escape escapeMode = {},
                              Utf8Validation validation = {}) {
  if (!s.empty()) {
    String_t res;
    if (validation == Utf8Validation::IgnoreInvalidUtf8CodeUnits)
      res.reserve(s.size());
    using Sink = detail::StringSink<String_t>;
    if (impl::escape(Sink{ res }, s, escapeMode, validation) == s.size())
      return res;
  }
  return {};
}
```

```
template<typename String_t = std::string>
[[nodiscard]] String_t escape(const std::string_view &s,
```

```
namespace detail {
  template<typename String>
  struct StringSink final {
    void operator()(const std::string_view &t) { s += t; }
    String &s;
  };
}
```

```
    using Sink = detail::StringSink<String_t>;
    if (impl::escape(Sink{ res }, s, escapeMode, validation) == s.size())
      return res;
  }
  return {};
}
```

```cpp
template<typename String_t = std::string>
[[nodiscard]] String_t escape(const std::string_view &s,
                              Escape escapeMode = {},
                              Utf8Validation validation = {}) {
  if (!s.empty()) {
    String_t res;
    if (validation == Utf8Validation::IgnoreInvalidUtf8CodeUnits)
      res.reserve(s.size());
    using Sink = detail::StringSink<String_t>;
    if (impl::escape(Sink{ res }, s, escapeMode, validation) == s.size())
      return res;
  }
  return {};
}
```

```cpp
template<typename String_t = std::string>
[[nodiscard]] String_t escape(const std::string_view &s,
                              Escape escapeMode = {},
                              Utf8Validation validation = {}) {
  if (!s.empty()) {
    String_t res;
    if (validation == Utf8Validation::IgnoreInvalidUtf8CodeUnits)
      res.reserve(s.size());
    using Sink = detail::StringSink<String_t>;
    if (impl::escape(Sink{ res }, s, escapeMode, validation) == s.size())
      return res;
  }
  return {};
}
```

```cpp
std::cout << escape(R"(text with newline
 and "quotation marks")");
```

```
SELECT *
 FROM employees
 WHERE last_name = 'Shmoe' AND age > 40 AND age < 50;

 age | first_name | last_name
-----+------------+-----------
  48 | Joe        | Shmoe
(1 row)
```

Importance of UTF-8 validation during escaping

```
hax\xc0'; \! ls #
```

Importance of UTF-8 validation during escaping

```
hax\xc0'; \! ls #
    \xc0\x27
    0b11100000 0b00100111
```

Importance of UTF-8 validation during escaping

```
hax\xc0'; \! ls #
    \xc0\x27
    0b11000000 0b00100111
```

one-byte code unit

```
proper two-byte UTF-8 code point:
    0b110xxxx 0b10xxxxxx
```

two-byte starting code unit

continuation code unit

Importance of UTF-8 validation during escaping

```
SELECT *
 FROM employees
 WHERE last_name = 'hax\xc0'; \! ls #' AND age > 40 AND age < 50;
```

Importance of UTF-8 validation during escaping

```
SELECT *
 FROM employees
 WHERE last_name = 'hax\xc0'; \! ls #' AND age > 40 AND age < 50;
```

Importance of UTF-8 validation during escaping

```
SELECT *
 FROM employees
 WHERE last_name = 'hax\xc0'; \! ls #' AND age > 40 AND age < 50;
```

```
SELECT * FROM employees WHERE last_name = 'haxÀ';
ERROR:  invalid byte sequence for encoding "UTF8": 0xc0 0x27
```

```
PG_VERSION              pg_multixact            pg_twophase
base                    pg_notify               pg_wal
global                  pg_replslot             pg_xact
logfile                 pg_serial               postgresql.auto.conf
pg_commit_ts            pg_snapshots            postgresql.conf
pg_dynshmem             pg_stat                 postmaster.opts
pg_hba.conf             pg_stat_tmp             postmaster.pid
pg_ident.conf           pg_subtrans
pg_logical              pg_tblspc
```

CVE-2024-12356
CVE-2025-1094

Importance of UTF-8 validation during escaping

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++

JSON value is a union type

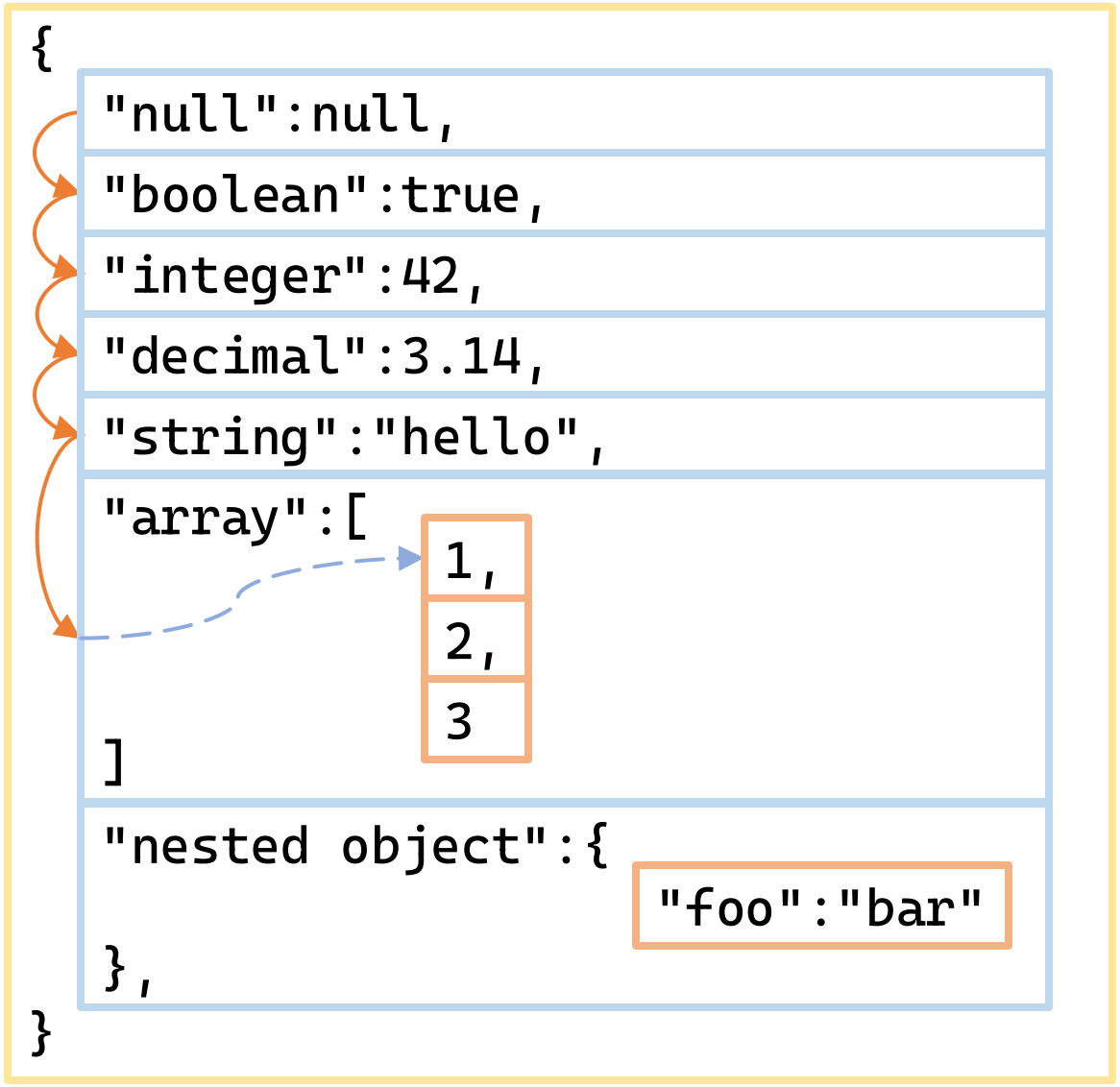
```
std::variant<std::monostate,
             bool,
             int64_t,
             double,
             std::string,
             std::vector<Value>,
             std::unordered_map<String, Value>>
```

null

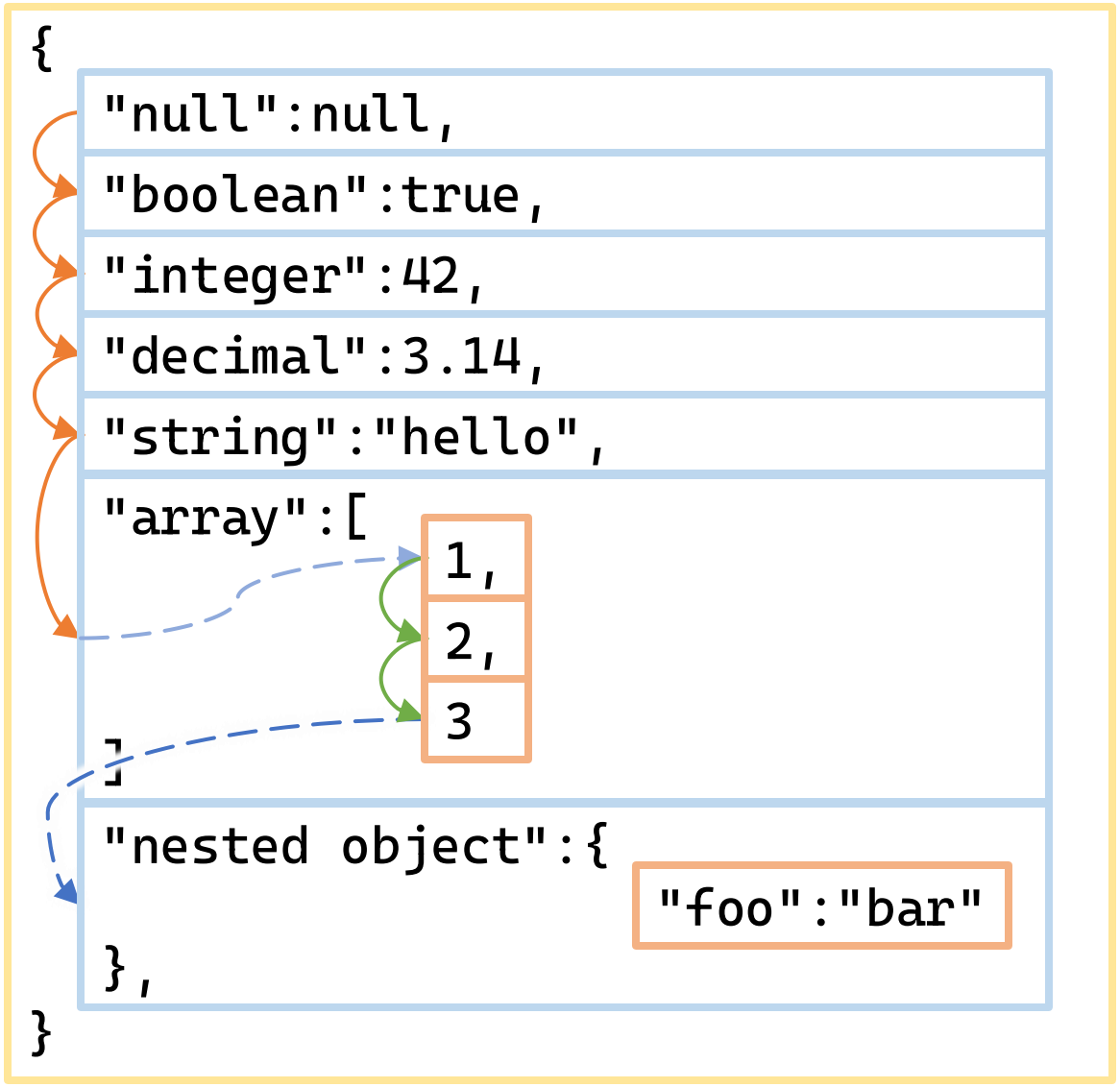
Mapping of JSON types into C++

```
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;
};
```

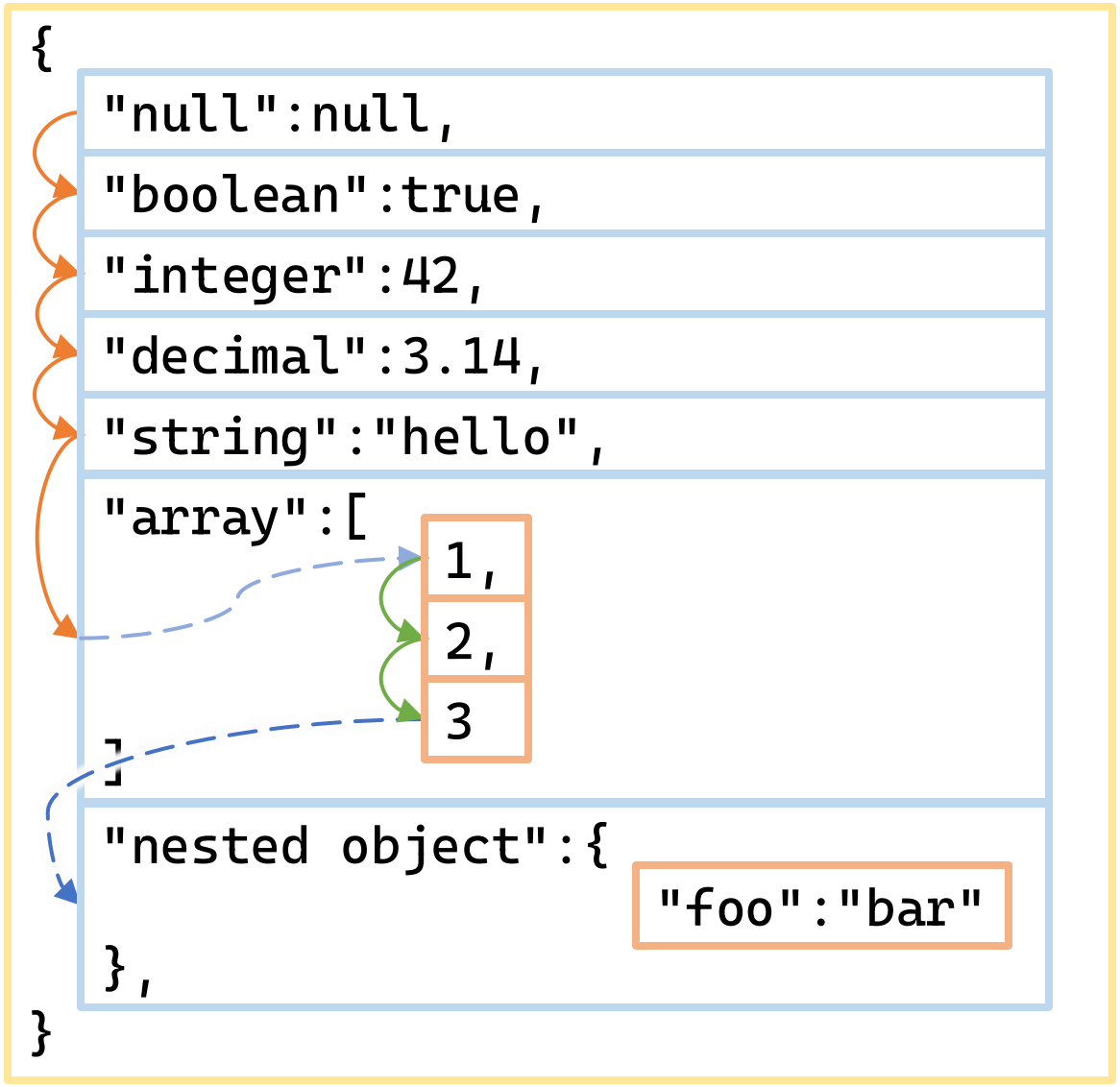
`Value` type



Serialization



Serialization



Serialization

We need a stack structure.

- recursion, i.e. call stack
  - can use stack space inefficiently
- structure on program stack
  - can waste program stack space
- structure on the heap
  - memory allocation expenses

```
constexpr std::string_view NullLiteral{ "null" };
constexpr std::string_view FalseLiteral{ "false" };
constexpr std::string_view TrueLiteral{ "true" };
```

Serialization

```
struct SerializationOptions {
  size_t indent = 0;
  Escape escape = {};
  Utf8Validation validation = {};
  std::string_view newlineSeparator = {};
  bool sortObjectKeys = false;
  //...
};
```

Serialization

```cpp
struct SerializationOptions {
  //...
  std::string_view nullLiteral = NullLiteral;
  std::string_view falseLiteral = FalseLiteral;
  std::string_view trueLiteral = TrueLiteral;
  std::string_view emptyObject = "{}";
  std::string_view objectOpeningBrace = "{";
  std::string_view objectClosingBrace = "}";
  std::string_view objectKeyValueSeparator = ":";
  std::string_view objectMemberSeparator = ",";
  std::string_view emptyArray = "[]";
  std::string_view arrayOpeningBracket = "[";
  std::string_view arrayClosingBracket = "]";
  std::string_view arrayMemberSeparator = ",";
  std::string_view openingStringQuotation = "\"";
  std::string_view closingStringQuotation = "\"";
};
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &options,
                 size_t initialIndentation = 0) {
  //...
}

void serializeToStream(std::ostream &s, const Value &v, const SerializationOptions &o = {}) {
  impl::serialize(detail::StdOStreamSink{ s }, v, o);
}

[[nodiscard]] Value::String serializeToString(const Value &v,
                                              const SerializationOptions &o = {}) {
  Value::String s;
  impl::serialize(detail::StringSink<Value::String>{ s }, v, o);
  return s;
}
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &options,
                 size_t initialIndentation = 0) {
```

```
void serializeToStream(std::ostream &s,
                       const Value &v,
                       const SerializationOptions &o = {}) {
  impl::serialize(detail::StdOStreamSink { s }, v, o);
}

[[nodiscard]]
Value::String serializeToString(const Value &v,
                                const SerializationOptions &o = {}) {
  Value::String s;
  impl::serialize(detail::StringSink<Value::String>{ s }, v, o);
  return s;
}
```

```
namespace im
  template<t
  void seria
```

```
namespace detail {
  struct StdOStreamSink final {
    void operator()(const std::string_view &v) const {
      s.write(v.data(), v.size());
    }
    std::ostream &s;
  };
}
```

```
void serialize(std::ostream &s,
               const Value &v,
               const SerializationOptions &o = {}) {
  impl::serialize(detail::StdOStreamSink { s }, v, o);
}

[[nodiscard]]
Value::String serializeToString(const Value &v,
                                const SerializationOptions &o = {}) {
  Value::String s;
  impl::serialize(detail::StringSink<Value::String>{ s }, v, o);
  return s;
}
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &options,
                 size_t initialIndentation = 0) {
```

```
void serializeToStream(std::ostream &s,
                       const Value &v,
                       const SerializationOptions &o = {}) {
  impl::serialize(detail::StdOStreamSink { s }, v, o);
}

[[nodiscard]]
Value::String serializeToString(const Value &v,
                                const SerializationOptions &o = {}) {
  Value::String s;
  impl::serialize(detail::StringSink<Value::String>{ s }, v, o);
  return s;
}
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &o,
                 size_t initialIndent = 0) {
    using Visitor = detail::SerializingVisitor<Sink>;
    if (o.indent || !o.newlineSeparator.empty()) {
      const std::string_view newlineSeparator =
        o.newlineSeparator.empty() ? "\n" : o.newlineSeparator;
      std::visit(
        Visitor{ std::forward<Sink>(sink), o, newlineSeparator, initialIndent },
        v.variant());
    }
    else {
      std::visit(
        Visitor{ std::forward<Sink>(sink), o, {}, 0 },
        v.variant());
    }
  }
}
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &o,
                 size_t initialIndent = 0) {
```

```
using Visitor = detail::SerializingVisitor<Sink>;
if (o.indent || !o.newlineSeparator.empty()) {
  const std::string_view newlineSeparator =
    o.newlineSeparator.empty() ? "\n" : o.newlineSeparator;
  std::visit(
    Visitor{ std::forward<Sink>(sink), o, newlineSeparator, initialIndent },
    v.variant());
}
else {
  std::visit(
    Visitor{ std::forward<Sink>(sink), o },
    v.variant());
}
```

```
  }
}
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &o,
                 size_t initialIndent = 0) {
```

```
using Visitor = detail::SerializingVisitor<Sink>;
if (o.indent || !o.newlineSeparator.empty()) {
  const std::string_view newlineSeparator =
    o.newlineSeparator.empty() ? "\n" : o.newlineSeparator;
  std::visit(
    Visitor{ std::forward<Sink>(sink), o, newlineSeparator, initialIndent },
    v.variant());
}
else {
  std::visit(
    Visitor{ std::forward<Sink>(sink), o },
    v.variant());
}
```

```
  }
}
```

```
namespace impl {
  template<typename Sink>
  void serialize(Sink &&sink,
                 const Value &v,
                 const SerializationOptions &o,
                 size_t initialIndent = 0) {
```

```
using Visitor = detail::SerializingVisitor<Sink>;
if (o.indent || !o.newlineSeparator.empty()) {
  const std::string_view newlineSeparator =
    o.newlineSeparator.empty() ? "\n" : o.newlineSeparator;
  std::visit(
    Visitor{ std::forward<Sink>(sink), o, newlineSeparator, initialIndent },
    v.variant());
}
else {
  std::visit(
    Visitor{ std::forward<Sink>(sink), o },
    v.variant());
}
```

```
  }
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    void operator()(const typename Value::Object &o) {/*...*/}
    void operator()(const typename Value::Array &a) {/*...*/}
    void operator()(const typename Value::String &t) {/*...*/}
    void operator()(double d) {/*...*/}
    void operator()(int64_t i) {/*...*/}
    void operator()(Boolean b) {/*...*/}
    void operator()(Null) {/*...*/}

    Sink &&sink;
    const SerializationOptions &options;
    const std::string_view newlineSeparator;
    size_t indentation;

  private:
    //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
    void operator()(double d) {
      char buf[24];
      const auto result = std::to_chars(buf, buf + sizeof(buf), d);
      const size_t size = static_cast<size_t>(result.ptr - buf);
      sink(std::string_view{ buf, size });
    }
    //...
  };
}
```

Serialization

```
namespace detail {
  template<typename Sink>
  stru
    //
    vo
      const auto result = std::to_chars(buf, buf + sizeof(buf), d);
      const size_t size = static_cast<size_t>(result.ptr - buf);
      sink(std::string_view{ buf, size });
    }
    //...
  };
}
```

`std::to_chars()` gives *shortest* representation without loss of precision, e.g

`1234.5678`

not

`1234.5678000000003383320290595293045043945312 5`

Serialization

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
    void operator()(double d) {
      char buf[24];
      const auto result = std::to_chars(buf, buf + sizeof(buf), d);
      const size_t size = static_cast<size_t>(result.ptr - buf);
      sink(std::string_view{ buf, size });
    }
    //...
  };
}
```

Serialization

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
    void operator()(const typename Value::String &t) { writeString(t); }
    //...
  private:
    void writeString(const std::string_view &s) const {
      //...
    }
    //...
  };
}
```

Serialization

```
void writeString(const std::string_view &s) const {
  sink(options.openingStringQuotation);
  const size_t escapedSize =
    impl::escape(sink, s, options.escape, options.validation);
  if (options.validation == Utf8Validation::FailOnInvalidUtf8CodeUnits &&
      escapedSize != s.size()) {
    const auto badCodeUnits =
      s.substr(escapedSize, getExpectedUtf8CodePointSize(s[escapedSize]));
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     badCodeUnits,
                                     escapedSize };
  }
  sink(options.closingStringQuotation);
}
```

"

```
void writeString(const std::string_view &s) const {
  sink(options.openingStringQuotation);
  const size_t escapedSize =
    impl::escape(sink, s, options.escape, options.validation);
  if (options.validation == Utf8Validation::FailOnInvalidUtf8CodeUnits &&
      escapedSize != s.size()) {
    const auto badCodeUnits =
      s.substr(escapedSize, getExpectedUtf8CodePointSize(s[escapedSize]));
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     badCodeUnits,
                                     escapedSize };
  }
  sink(options.closingStringQuotation);
}
```

"hello

```
void writeString(const std::string_view &s) const {
  sink(options.openingStringQuotation);
  const size_t escapedSize =
    impl::escape(sink, s, options.escape, options.validation);
  if (options.validation == Utf8Validation::FailOnInvalidUtf8CodeUnits &&
      escapedSize != s.size()) {
    const auto badCodeUnits =
      s.substr(escapedSize, getExpectedUtf8CodePointSize(s[escapedSize]));
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     badCodeUnits,
                                     escapedSize };
  }
  sink(options.closingStringQuotation);
}
```

"hello

```
void writeString(const std::string_view &s) const {
  sink(options.openingStringQuotation);
  const size_t escapedSize =
    impl::escape(sink, s, options.escape, options.validation);
  if (options.validation == Utf8Validation::FailOnInvalidUtf8CodeUnits &&
      escapedSize != s.size()) {
    const auto badCodeUnits =
      s.substr(escapedSize, getExpectedUtf8CodePointSize(s[escapedSize]));
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     badCodeUnits,
                                     escapedSize };
  }
  sink(options.closingStringQuotation);
}
```

"hello"

```
void writeString(const std::string_view &s) const {
  sink(options.openingStringQuotation);
  const size_t escapedSize =
    impl::escape(sink, s, options.escape, options.validation);
  if (options.validation == Utf8Validation::FailOnInvalidUtf8CodeUnits &&
      escapedSize != s.size()) {
    const auto badCodeUnits =
      s.substr(escapedSize, getExpectedUtf8CodePointSize(s[escapedSize]));
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     badCodeUnits,
                                     escapedSize };
  }
  sink(options.closingStringQuotation);
}
```

```
void writeString(const std::string_view &s) const {
  sink(options.openingStringQuotation);
  const size_t escapedSize =
    impl::escape(sink, s, options.escape, options.validation);
  if (options.validation == Utf8Validation::FailOnInvalidUtf8CodeUnits &&
      escapedSize != s.size()) {
    const auto badCodeUnits =
      s.substr(escapedSize, getExpectedUtf8CodePointSize(s[escapedSize]));
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     badCodeUnits,
                                     escapedSize };
  }
  sink(options.closingStringQuotation);
}
```

most reasonable way to report an error in this case

```
struct InvalidUtf8CodeUnitsError : std::runtime_error {
  InvalidUtf8CodeUnitsError(const char *msg,
                            const std::string_view &codeUnits,
                            size_t offset) :

    runtime_error{ msg },
    codeUnits{ codeUnits },
    offset{ offset }
  {}
  const std::string codeUnits;
  const size_t offset;
};
```

Serialization

```
struct InvalidUtf8CodeUnitsError : std::runtime_error {
  InvalidUtf8CodeUnitsError(const char *msg,
                            const std::string_view &codeUnits,
                            size_t offset) :
    runtime_error{ msg },
    codeUnits{ codeUnits },
    offset{ offset }
  {}
  const std::string codeUnits;
  const size_t offset;
};
```

Serialization

```
struct InvalidUtf8CodeUnitsError : std::runtime_error {
  InvalidUtf8CodeUnitsError(const char *msg,
                            const std::string_view &codeUnits,
                            size_t offset) :
    runtime_error{ msg },
    codeUnits{ codeUnits },
    offset{ offset }
  {}
  const std::string codeUnits;
  const size_t offset;
};
```

forbids `noexcept` copy ctor

Serialization

```
struct InvalidUtf8CodeUnitsError : std::runtime_error {
  InvalidUtf8CodeUnitsError(const char *msg,
                            const std::string_view &codeUnits,
                            size_t offset) :
    runtime_error{ msg },
    codeUnits{ codeUnits },
    offset{ offset }
  {}
  const std::string codeUnits;
  const size_t offset;
};
```

Serialization

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
    void operator()(const typename Value::Object &o) {/*...*/}
    //...
  private:
    //...
  };
}
```

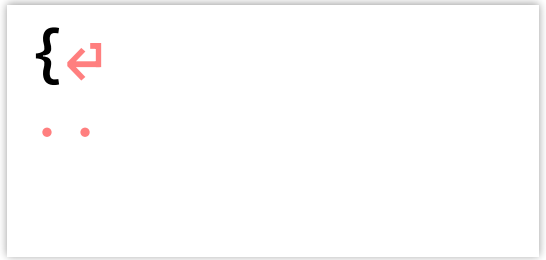
Serialization

```
void operator()(const typename Value::Object &o) {
  if (o.empty() && !options.emptyObject.empty()) {
    sink(options.emptyObject);
    return;
  }
  sink(options.objectOpeningBrace);
  indentation += options.indent;
  writeNewlineAndIndentation();
  //...
}
```

{}

Serialization

```
void operator()(const typename Value::Object &o) {
  if (o.empty() && !options.emptyObject.empty()) {
    sink(options.emptyObject);
    return;
  }
  sink(options.objectOpeningBrace);
  indentation += options.indent;
  writeNewlineAndIndentation();
  //...
}
```



Serialization

```
void operator()(const typename Value::Object &o) {
  //...
  if (o.size() == 1) {
    writeObjectMember(*o.begin());
  }
  else if (options.sortObjectKeys) {
    // ???
    std::sort(items.begin(), items.end(),
              /* ??? */);
    writeObjectMembers(items.begin(), items.end());
  }
  else {
    writeObjectMembers(o.begin(), o.end());
  }
  //...
}
```

```
{
  "foo":"bar"
```

```
void operator()(const typename Value::Object &o) {
  //...
  if (o.size() == 1) {
    writeObjectMember(*o.begin());
  }
  else if (options.sortObjectKeys) {
    // ???
    std::sort(items.begin(), items.end(),
              /* ??? */);
    writeObjectMembers(items.begin(), items.end());
  }
  else {
    writeObjectMembers(o.begin(), o.end());
  }
  //...
}
```

```
{
  "foo":"bar"
```

```
void operator()(const typename Value::Object &o) {
  //...
  indentation -= options.indent;
  writeNewlineAndIndentation();
  sink(options.objectClosingBrace);
}
```

```
{
  "foo":"bar"
}
```

Serialization

```
void operator()(const typename Value::Object &o) {
  //...
  if (o.size() == 1) {
    writeObjectMember(*o.begin());
  }
  else if (options.sortObjectKeys) {
    // ???
    std::sort(items.begin(), items.end(),
              /* ??? */);
    writeObjectMembers(items.begin(), items.end());
  }
  else {
    writeObjectMembers(o.begin(), o.end());
  }
  //...
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
  private:
    //...
    template<typename Iterator>
    void writeObjectMembers(Iterator begin, const Iterator &end) {
      writeObjectMember(*begin);
      for (++begin; begin != end; ++begin) {
        sink(options.objectMemberSeparator);
        writeNewlineAndIndentation();
        writeObjectMember(*begin);
      }
    }
  //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
  private:
    //...
    template<typename Iterator>
    void writeObjectMembers(Iterator begin, const Iterator &end) {
      writeObjectMember(*begin);
      for (++begin; begin != end; ++begin) {
        sink(options.objectMemberSeparator);
        writeNewlineAndIndentation();
        writeObjectMember(*begin);
      }
    }
  //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
  private:
    //...
    void writeObjectMember(const typename Value::Object::value_type &i) {
      const auto &[key, value] = i;
      writeString(key);
      sink(options.objectKeyValueSeparator);
      std::visit(*this, value.variant());
    }



  //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
  private:
    //...
    void writeObjectMember(const typename Value::Object::value_type &i) {
      const auto &[key, value] = i;
      writeString(key);
      sink(options.objectKeyValueSeparator);
      std::visit(*this, value.variant());
    }



  //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
  private:
    //...
    void writeObjectMember(const typename Value::Object::value_type &i) {
      const auto &[key, value] = i;
      writeString(key);
      sink(options.objectKeyValueSeparator);
      std::visit(*this, value.variant());
    }



  //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    //...
  private:
    //...
    void writeObjectMember(const typename Value::Object::value_type &i) {
      const auto &[key, value] = i;
      writeString(key);
      sink(options.objectKeyValueSeparator);
      std::visit(*this, value.variant());
    }



  //...
  };
}
```

possible recursion

```
void operator()(const typename Value::Object &o) {
  //...
  if (o.size() == 1) {
    writeObjectMember(*o.begin());
  }
  else if (options.sortObjectKeys) {
    // ???
    std::sort(items.begin(), items.end(),
              /* ??? */);
    writeObjectMembers(items.begin(), items.end());
  }
  else {
    writeObjectMembers(o.begin(), o.end());
  }
  //...
}
```

```
void operator()(const typename Value::Object &o) {
  //...
  else if (options.sortObjectKeys) {
    using ReferenceWrapper =
      std::reference_wrapper<const typename Value::Object::value_type>;

    std::sort(items.begin(), items.end(),
              [](auto a, auto b) {
                // ???
              });
    writeObjectMembers(items.begin(), items.end());
  }
  //...
}
```

```
void operator()(const typename Value::Object &o) {
  //...
  else if (options.sortObjectKeys) {
    using ReferenceWrapper =
      std::reference_wrapper<const typename Value::Object::value_type>;
    std::vector<ReferenceWrapper> items{ o.begin(), o.end() };
    std::sort(items.begin(), items.end(),
              [](auto a, auto b) {
                // ???
              });
    writeObjectMembers(items.begin(), items.end());
  }
  //...
}
```

```
void operator()(const typename Value::Object &o) {
  //...
  else if (options.sortObjectKeys) {
    using ReferenceWrapper =
      std::reference_wrapper<const typename Value::Object::value_type>;
    std::vector<ReferenceWrapper> items{ o.begin(), o.end() };
    std::sort(items.begin(), items.end(),
              [](auto a, auto b) {
                return std::get<0>(a.get()) < std::get<0>(b.get());
              });
    writeObjectMembers(items.begin(), items.end());
  }
  //...
}
```

```
void operator()(const typename Value::Object &o) {
  //...
  else if (options.sortObjectKeys) {
    using ReferenceWrapper =
      std::reference_wrapper<const typename Value::Object::value_type>;
    std::vector<ReferenceWrapper> items{ o.begin(), o.end() };
    std::sort(items.begin(), items.end(),
              [](auto a, auto b) {
                return std::get<0>(a.get()) < std::get<0>(b.get());
              });
    writeObjectMembers(items.begin(), items.end());
  }
  //...
}
```

```
namespace detail {
  template<typename Sink>
  struct SerializingVisitor final {
    void operator()(const typename Value::Object &o) {/*...*/}
    void operator()(const typename Value::Array &a) {/*...*/}
    void operator()(const typename Value::String &t) {/*...*/}
    void operator()(double d) {/*...*/}
    void operator()(int64_t i) {/*...*/}
    void operator()(Boolean b) {/*...*/}
    void operator()(Null) {/*...*/}

    Sink &&sink;
    const SerializationOptions &options;
    const std::string_view newlineSeparator;
    size_t indentation;

  private:
    //...
  };
}
```

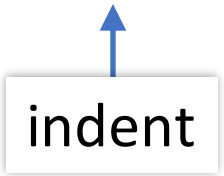
```
std::cout << serializeToString(json);
```

output:

```
{"decimal":3.14,"null":null,"boolean":true,"nested object":{"fo
```

Serialization

```
std::cout << serializeToString(json, { 2 });
```



output:

```
{
  "decimal":3.14,
  "null":null,
  "boolean":true,
  "nested object":{
    "foo":"bar"
  },
  "integer":42,
  "string":"hello",
  "array":[
    1,
    2,
    1234.5678
  ]
}
```

Serialization

```
SerializationOptions options;
options.indent = 2;
options.sortObjectKeys = true;
serializeToStream(std::cout, json, options);
```

output:

```
{
  "array":[
    1,
    2,
    1234.5678
  ],
  "boolean":true,
  "decimal":3.14,
  "integer":42,
  "nested object":{
    "foo":"bar"
  },
  "null":null,
  "string":"hello"
}
```

Serialization

```
SerializationOptions options;
options.indent = 2;

serializeToStream(std::cout, json, options);
```

output:

```
{
  "decimal":3.14,
  "null":null,
  "boolean":true,
  "nested object":{
    "foo":"bar"
  },
  "integer":42,
  "string":"hello",
  "array":[
    1,
    2,
    1234.5678
  ]
}
```

Serialization

```
SerializationOptions options;
options.indent = 2;
options.objectKeyValueSeparator = ": ";
serializeToStream(std::cout, json, options);
```

output:

```
{
  "decimal": 3.14,
  "null": null,
  "boolean": true,
  "nested object": {
    "foo": "bar"
  },
  "integer": 42,
  "string": "hello",
  "array": [
    1,
    2,
    1234.5678
  ]
}
```

Serialization

```
SerializationOptions options;
options.indent = 2;
options.nullLiteral = "\x1b[90mnull\x1b[0m";
options.falseLiteral = "\x1b[96mfalse\x1b[0m";
options.trueLiteral = "\x1b[96mtrue\x1b[0m";
options.emptyObject = "\x1b[91m{}\x1b[0m";
options.objectOpeningBrace = "\x1b[91m{\x1b[0m";
options.objectClosingBrace = "\x1b[91m}\x1b[0m";
options.objectKeyValueSeparator = "\x1b[31m:\x1b[0m ";
options.objectMemberSeparator = "\x1b[31m,\x1b[0m";
options.emptyArray = "\x1b[95m[]\x1b[0m";
options.arrayOpeningBracket = "\x1b[95m[\x1b[0m";
options.arrayClosingBracket = "\x1b[95m]\x1b[0m";
options.arrayMemberSeparator = "\x1b[35m,\x1b[0m";
options.openingStringQuotation = "\x1b[90m\"\x1b[97m";
options.closingStringQuotation = "\x1b[90m\"\x1b[0m";
serializeToStream(std::cout, json, options);
```

```
SerializationOptions options;
options.indent = 2;
options.nullLiteral = "\x1b[90mnull\x1b[0m";
options.falseLiteral = "\x1b[96mfalse\x1b[0m";
options.trueLiteral = "\x1b[96mtrue\x1b[0m";
options.emptyObject = "\x1b[91m{}\x1b[0m";
options.objectOpeningBrace = "\x1b[91m{\x1b[0m";
options.objectClosingBrace = "\x1b[91m}\x1b[0m";
options.objectKeyValueSeparator = "\x1b[31m:\x1b[
options.objectMemberSeparator = "\x1b[31m,\x1b[0m
options.emptyArray = "\x1b[95m[]\x1b[0m";
options.arrayOpeningBracket = "\x1b[95m[\x1b[0m";
options.arrayClosingBracket = "\x1b[95m]\x1b[0m";
options.arrayMemberSeparator = "\x1b[35m,\x1b[0m"
options.openingStringQuotation = "\x1b[90m\"\x1b[
options.closingStringQuotation = "\x1b[90m\"\x1b[
serializeToStream(std::cout, json, options);
```

```
{
  "decimal": 3.14,
  "null": null,
  "boolean": true,
  "nested object": {
    "foo": "bar"
  },
  "integer": 42,
  "string": "hello",
  "array": [
    1,
    2,
    1234.5678
  ]
}
```

```
SerializationOptions options;
options.indent = 2;
options.nullLiteral = R"(<span style="color: gray;">null</span>)"sv;
options.falseLiteral = R"(<span style="color: blue;">false</span>)"sv;
options.trueLiteral = R"(<span style="color: blue;">true</span>)"sv;
options.emptyObject = R"(<span style="color: red;">{}</span>)"sv;
options.objectOpeningBrace = R"(<span style="color: red;">{</span>)"sv;
options.objectClosingBrace = R"(<span style="color: red;">}</span>)"sv;
options.objectKeyValueSeparator = R"(<span style="color: darkred;">: </span>
options.objectMemberSeparator = R"(<span style="color: darkred;">,</span>)"s
options.emptyArray = R"(<span style="color: magenta;">[]</span>)"sv;
options.arrayOpeningBracket = R"(<span style="color: magenta;">[</span>)"sv
options.arrayClosingBracket = R"(<span style="color: magenta;">]</span>)"sv
options.arrayMemberSeparator = R"(<span style="color: darkmagenta;">,</span>
options.openingStringQuotation = R"(<span style="color: lightgray;">"</span>
options.closingStringQuotation = R"(</span><span style="color: lightgray;">"
serializeToStream(std::cout, json, options);
```

```
SerializationOptions options;
```

output:

```
<span style="color: red;">{</span>
  <span style="color: lightgray;">"</span><span style="color: teal;"
  <span style="color: lightgray;">"</span><span style="color: teal;"
  <span style="color: lightgray;">"</span><span style="color: teal;"
  <span style="color: lightgray;">"</span><span style="color: teal;"
    <span style="color: lightgray;">"</span><span style="color: teal
  <span style="color: red;">}</span><span style="color: darkred;">,<
  <span style="color: lightgray;">"</span><span style="color: teal;"
  <span style="color: lightgray;">"</span><span style="color: teal;"
  <span style="color: lightgray;">"</span><span style="color: teal;"
  <span style="color: lightgray;">"</span><span style="color: teal;"
    1,
    2,
    1234.5678
  <span style="color: magenta;">]</span>
<span style="color: red;">}</span>
```

rendered HTML

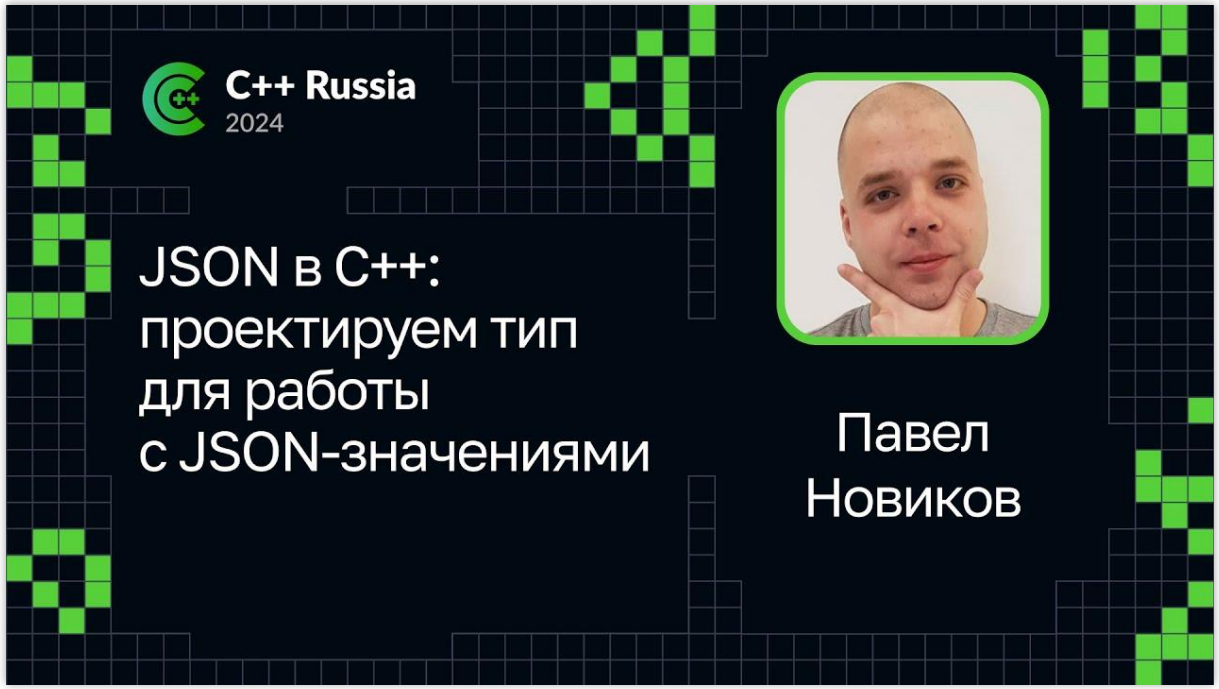
```
{
  "decimal": 3.14,
  "null": null,
  "boolean": true,
  "nested object": {
    "foo": "bar"
  },
  "integer": 42,
  "string": "hello",
  "array": [
    1,
    2,
    1234.5678
  ]
}
```

Serialization

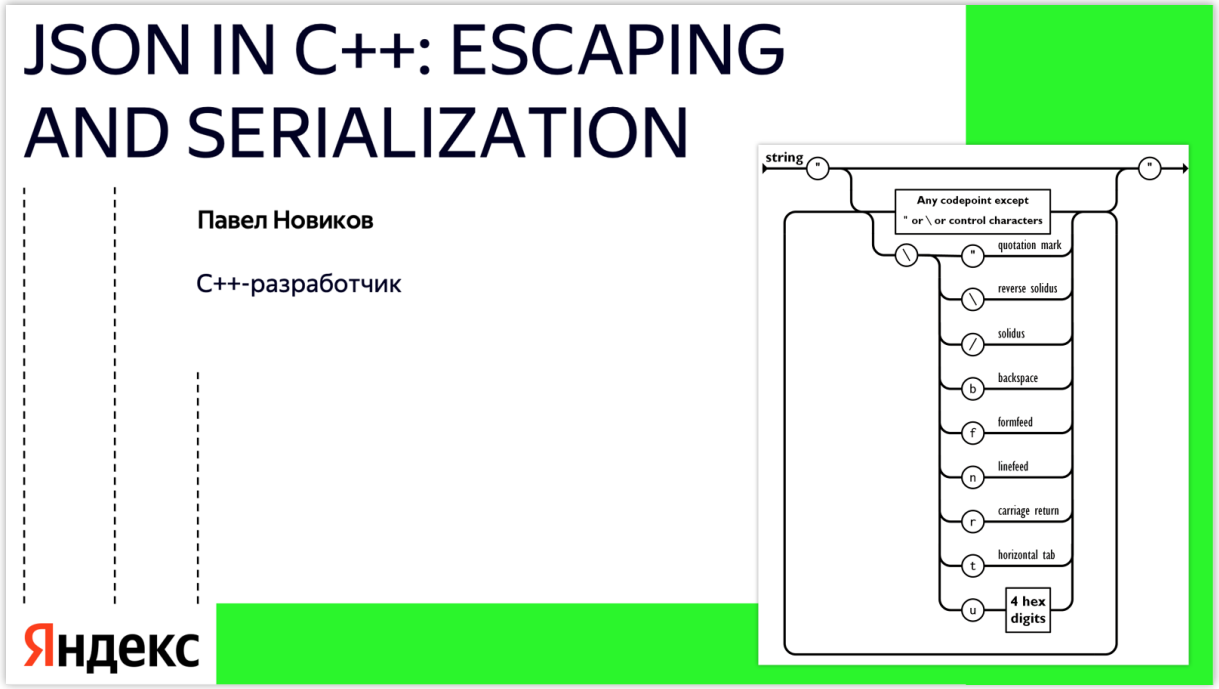
- UTF-8 validation may be important during escaping
- use `std::to_chars()` for floating point numbers
- exception types *should* have `noexcept` copy constructor
- know the standard library
  (e.g. using `std::vector` of `std::reference_wrapper`s to iterate in sorted order)

Conclusion

youtu.be/-JjE5AhfhcM



youtu.be/1THwOpon4vg



minjsoncpp

Minimalistic JSON C++ library

https://github.com/toughengineer/minjsoncpp

JSON in C++: serialization

Pavel Novikov

𝕏 @cpp_ape

Slides: bit.ly/3J2nL9G

References

- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format https://datatracker.ietf.org/doc/html/rfc8259
- CVE-2024-12356 analysis https://attackerkb.com/topics/G5s8ZWAbYH/cve-2024-12356/rapid7-analysis
- ANSI escape code: Colors https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
- JSON в C++: проектируем тип для работы с JSON-значениями https://www.youtube.com/watch?v=-JjE5AhfhcM
- JSON in C++: escaping and serialization https://www.youtube.com/watch?v=1THwOpon4vg
- minjsoncpp — Minimalistic JSON C++ library https://github.com/toughengineer/minjsoncpp

Bonus slides

```
namespace detail {
  template<size_t N>
  bool isUtf8CodeUnit(char c) {
    constexpr uint8_t mask = static_cast<uint8_t>(0xffu << (7 - N));
    constexpr uint8_t pattern = static_cast<uint8_t>(0xffu << (8 - N));
    return (mask & static_cast<uint8_t>(c)) == pattern;
  }
}
```

example:

```
N = 3
mask    = 0xffu << (7 - N) = 0b11110000
pattern = 0xffu << (8 - N) = 0b11100000
```

JSON string escaping

```
namespace detail {
  size_t getExpectedUtf8CodePointSize(char c) {
    if (isUtf8CodeUnit<2>(c)) return 2;
    if (isUtf8CodeUnit<3>(c)) return 3;
    if (isUtf8CodeUnit<4>(c)) return 4;
    return 1;
  }
}
```

JSON string escaping

```
postgres=# SELECT version();
SELECT version();
                                                              version
---------------------------------------------------------------------------------------------------------------------------------
-
 PostgreSQL 14.5 on i686-buildroot-linux-musl, compiled by i686-buildroot-linux-musl-gcc.br_real (Buildroot 2022.02.8) 11.3.0, 32-bit
(1 row)

postgres=# INSERT INTO employees VALUES(25, 'Joe', 'Shmoe');
INSERT INTO employees VALUES(25, 'Joe', 'Shmoe');
INSERT 0 1
postgres=# INSERT INTO employees VALUES(48, 'Joe', 'Shmoe');
INSERT INTO employees VALUES(48, 'Joe', 'Shmoe');
INSERT 0 1
```

PostgreSQL demo

```
postgres=# select * from employees;
select * from employees;
 age | first_name | last_name
-----+------------+-----------
  25 | Joe        | Shmoe
  48 | Joe        | Shmoe
(2 rows)

postgres=# SELECT * FROM employees WHERE last_name = 'Shmoe' AND age > 40 AND age < 50;
SELECT * FROM employees WHERE last_name = 'Shmoe' AND age > 40 AND age < 50;
 age | first_name | last_name
-----+------------+-----------
  48 | Joe        | Shmoe
(1 row)
```

PostgreSQL demo

```
postgres=# \set q1 `echo -e "SELECT * FROM employees WHERE last_name = 'hax\xc0'; \! ls #' AND age > 40 AND age < 50;"`
postgres=# :q1
SELECT * FROM employees WHERE last_name = 'haxÀ';
ERROR:  invalid byte sequence for encoding "UTF8": 0xc0 0x27
PG_VERSION            pg_multixact          pg_twophase
base                  pg_notify             pg_wal
global                pg_replslot           pg_xact
logfile               pg_serial             postgresql.auto.conf
pg_commit_ts          pg_snapshots          postgresql.conf
pg_dynshmem           pg_stat               postmaster.opts
pg_hba.conf           pg_stat_tmp           postmaster.pid
pg_ident.conf         pg_subtrans
pg_logical            pg_tblspc
```

PostgreSQL demo