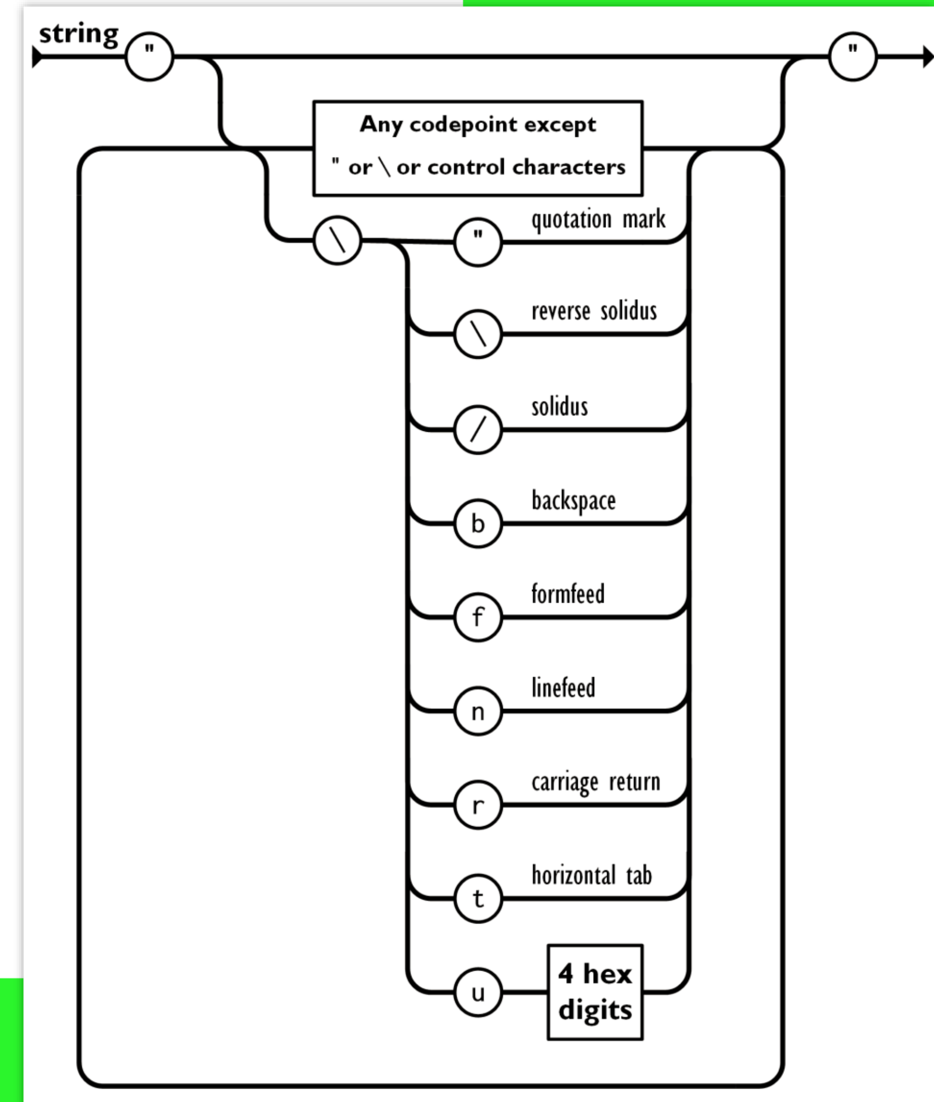
JSON IN C++: ESCAPING AND SERIALIZATION

Павел Новиков

C++-разработчик



Яндекс

JSON in C++:
escaping and serialization

Pavel Novikov

@cpp_ape

- quick overview of JSON
- JSON string escaping
  - deeper dive into the specification
- 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
- parsing

Constraints for this talk

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	structured types or collection types
`{ "key": "value" }`	object (dictionary)	

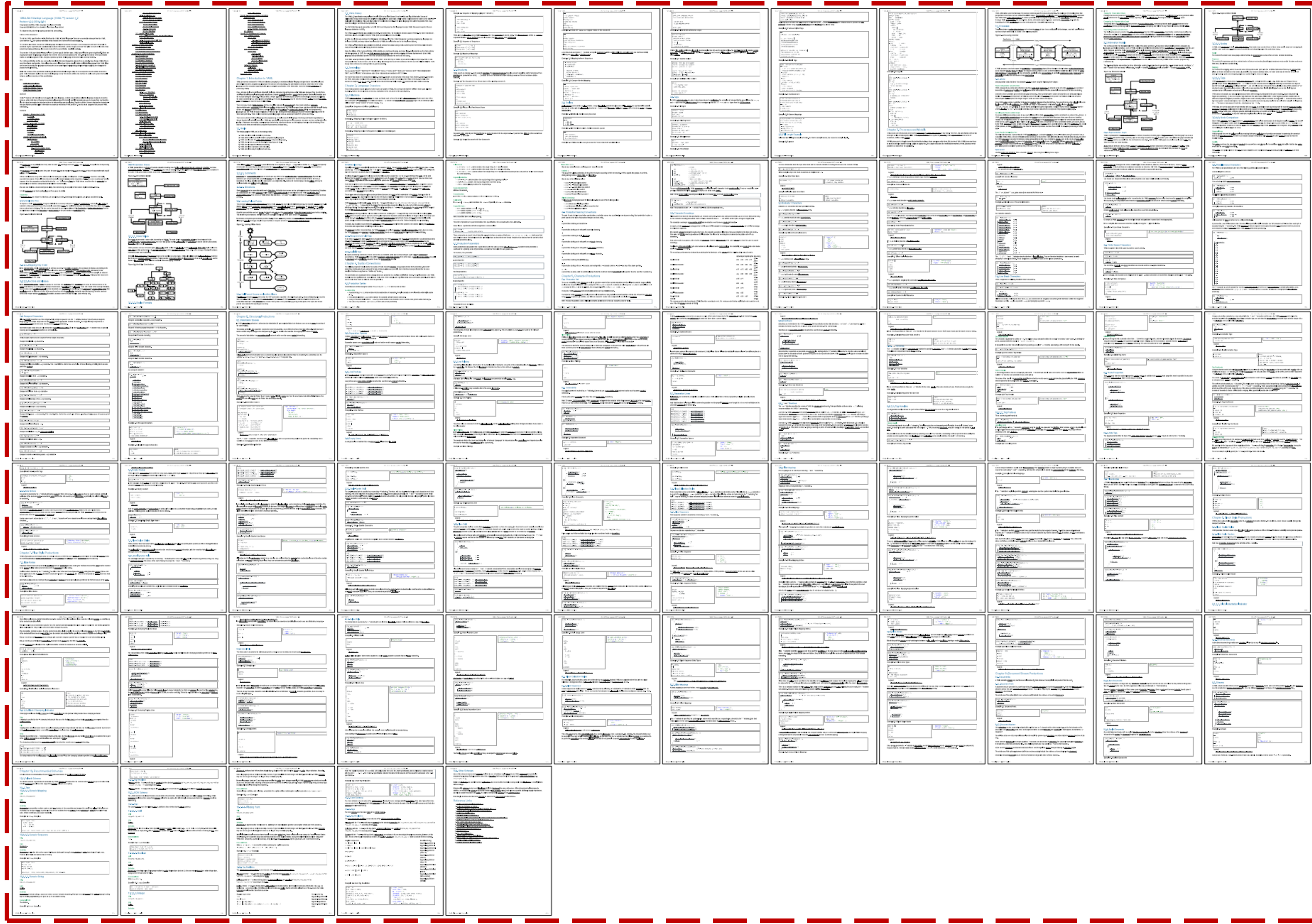
Overview of JSON

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
```

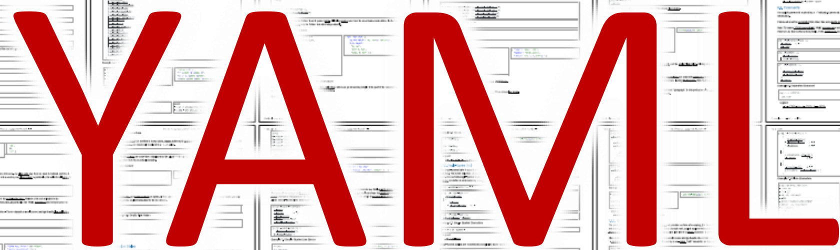
```
number = [ minus ] int [ frac ] [ exp ]
decimal-point = %x2E         ; .
digit1-9 = %x31-39           ; 1-9
e = %x65 / %x45              ; e E
exp = e [ minus / plus ] 1*DIGIT
frac = decimal-point 1*DIGIT
int = zero / ( digit1-9 *DIGIT )
minus = %x2D                 ; -
plus = %x2B                  ; +
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
```

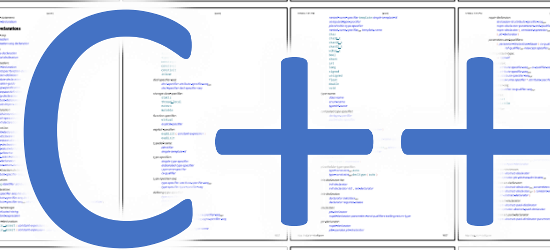
Overview of JSON



YAML

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





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

https://eel.is/c++draft/#gram

entire

RFC 8259

C++

https://eel.is/c++draft/#gram

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

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

JSON string escaping

Character	Escaped	Description
"	\"	quotation mark U+0022
\	\\	reverse solidus U+005C
/	\/	solidus U+002F
	\b	backspace U+0008
	\f	form feed U+000C
	\n	line feed U+000A
	\r	carriage return U+000D
	\t	tab U+0009
	\uXXXX	any Unicode character*

Characters that MUST be escaped:
"
\
U+0000..U+001F — control characters

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

Considerations for C++ string type

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 <u>code units</u>

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

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

```
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 {};
}
```

```
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,
                              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 {};
}
```

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

```
namespace impl {
  template<typename Sink>
  size_t escape(Sink &&sink,
                const std::string_view &s,
                Escape escapeMode,
                Utf8Validation validation) {
    detail::EscapedStringWriter<Sink> writer{ std::forward<Sink>(sink) };
    const char *escapedEnd =
      writer.write(s.data(), s.data() + s.size(), escapeMode, validation);
    return static_cast<size_t>(escapedEnd - s.data());
  }
}
```

JSON string escaping

```
namespace impl {
  template<typename Sink>
  size_t escape(Sink &&sink,
                const std::string_view &s,
                Escape escapeMode,
                Utf8Validation validation) {
    detail::EscapedStringWriter<Sink> writer{ std::forward<Sink>(sink) };
    const char *escapedEnd =
      writer.write(s.data(), s.data() + s.size(), escapeMode, validation);
    return static_cast<size_t>(escapedEnd - s.data());
  }
}
```

JSON string escaping

```
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    EscapedStringWriter(Sink &&sink) : sink{ std::forward<Sink>(sink) } {}

    const char *write(const char *begin, const char *end,
                      Escape escape, Utf8Validation validation);

  private:
    //...
    Sink &&sink;
    EscapedChar escaped;
    const char *pendingBegin;
  };
}
```

JSON string escaping

```
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    EscapedStringWriter(Sink &&sink) : sink{ std::forward<Sink>(sink) } {}

    const char *write(const char *begin, const char *end,
                      Escape escape, Utf8Validation validation);

  private:
    //...
    Sink &&sink;
    EscapedChar escaped;
    const char *pendingBegin;
  };
}
```

JSON string escaping

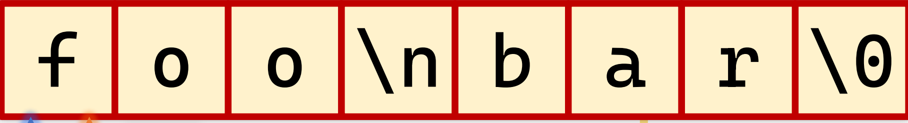
```cpp
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    EscapedStringWriter(Sink &&sink) : sink{ std::forward<Sink>(sink) } {}

    const char *write(const char *begin, const char *end,
                      Escape escape, Utf8Validation validation);

  private:
    //...
    Sink &&sink;
    EscapedChar escaped;
    const char *pendingBegin;
  };
}
```

JSON string escaping

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

characters with short escape sequences

control char

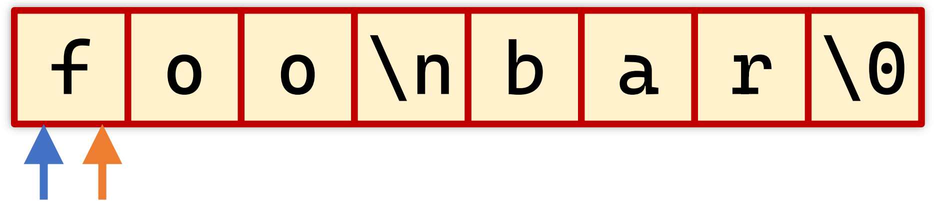


```
pendingBegin = begin;
while (pendingBegin != end) {
  const char *i = pendingBegin;
  for (;;) {
    //...
  } // for (;;)
} // while ()
return end;
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

characters with short escape sequences

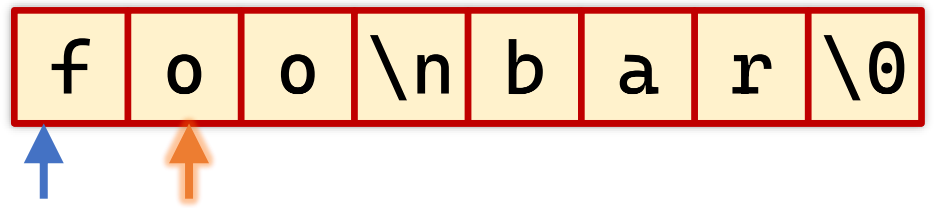
control characters



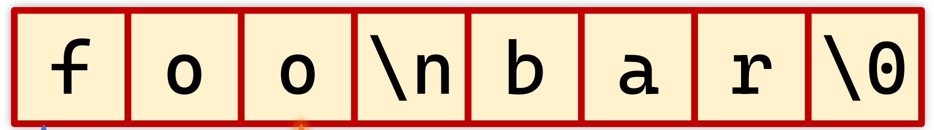
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

characters with short escape sequences

control characters

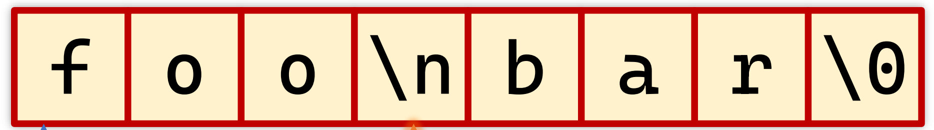


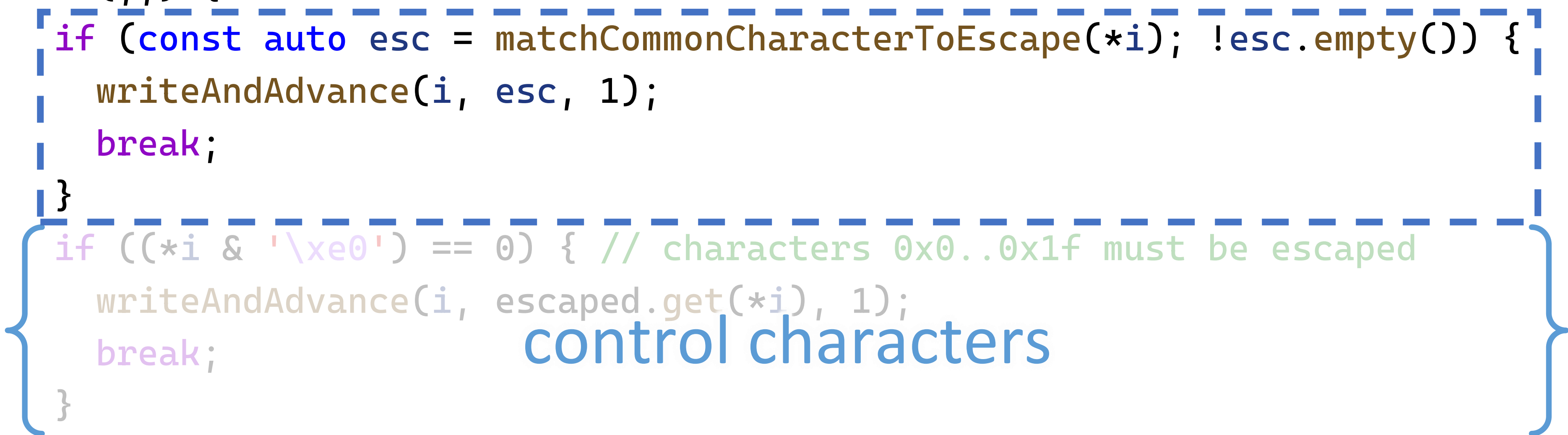
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```



characters with short escape sequences

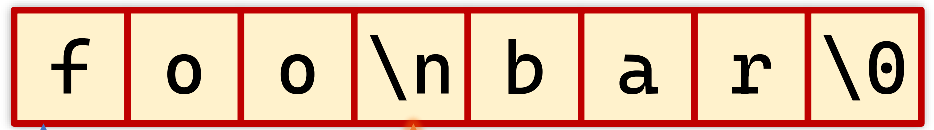
control characters






```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
      }
    } // for (;;)
  } // while ()
  return end;
}
```

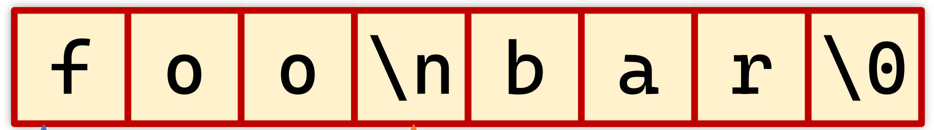



```
for (;;) {
  if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
    writeAndAdvance(i, esc, 1);
    break;
  }
  //...
} // for (;;)
```

```
namespace detail {
  std::string_view matchCommonCharacterToEscape(char c) {
    using namespace std::string_view_literals;
    switch (c) {
    case '\b': return "\\b"sv;
    case '\t': return "\\t"sv;
    case '\n': return "\\n"sv;
    case '\f': return "\\f"sv;
    case '\r': return "\\r"sv;
    case '\"': return "\\\""sv;
    case '\\': return "\\\\"sv;
    }
    return {};
  }
}
```

JSON string escaping

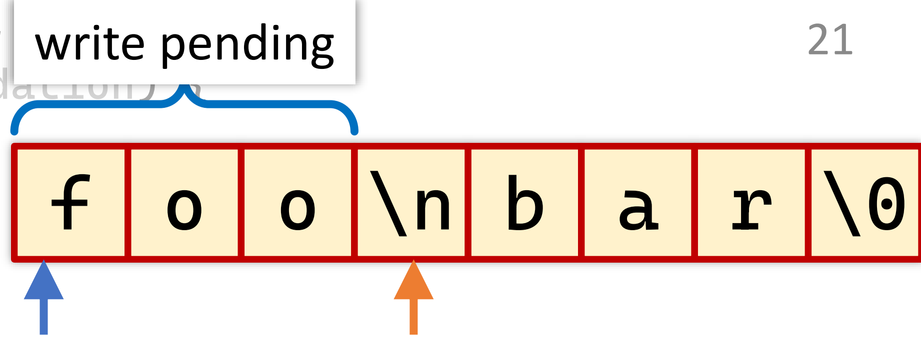
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
```




```
for (;;) {
  if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
    writeAndAdvance(i, esc, 1);
    break;
  }
  //...
} // for (;;)
```

```
      }
    } // for (;;)
  } // while ()
  return end;
}
```

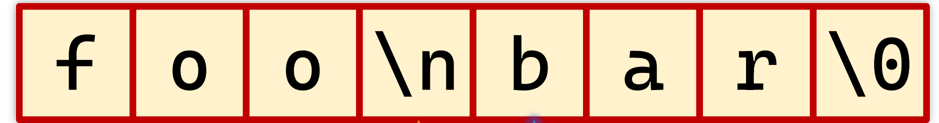
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
```




```
void writeAndAdvance(const char *i,
                     const std::string_view &escapeSequence,
                     size_t inc) {
  if (pendingBegin != i)
    writePending(i);
  pendingBegin = i + inc;
  sink(escapeSequence);
}
void writePending(const char *i) {
  sink(std::string_view{ pendingBegin, static_cast<size_t>(i - pendingBegin) });
}
```

```
  } // while ()
  return end;
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
```




```
void writeAndAdvance(const char *i,
                     const std::string_view &escapeSequence,
                     size_t inc) {
  if (pendingBegin != i)
    writePending(i);
  pendingBegin = i + inc;
  sink(escapeSequence);
}
void writePending(const char *i) {
  sink(std::string_view{ pendingBegin, static_cast<size_t>(i - pendingBegin) });
}
```

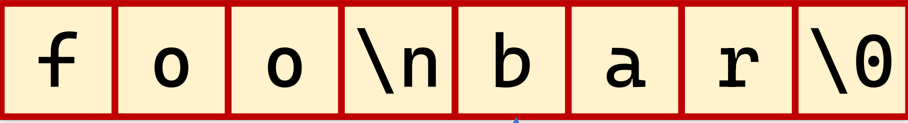
```
  } // while ()
  return end;
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
```

```
void writeAndAdvance(const char *i,
                     const std::string_view &escapeSequence,
                     size_t inc) {
  if (pendingBegin != i)
    writePending(i);
  pendingBegin = i + inc;
  sink(escapeSequence);
}
void writePending(const char *i) {
  sink(std::string_view{ pendingBegin, static_cast<size_t>(i - pendingBegin) });
}
```

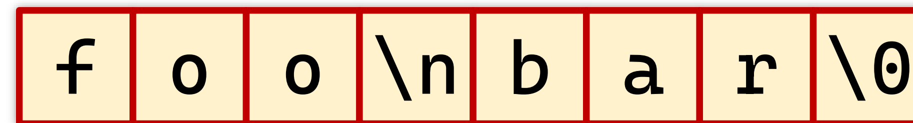
```
  } // while ()
  return end;
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
      }
    } // for (;;)
  } // while ()
  return end;
}
```




```
for (;;) {
  if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
    writeAndAdvance(i, esc, 1);
    break;
  }
  //...
} // for (;;)
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

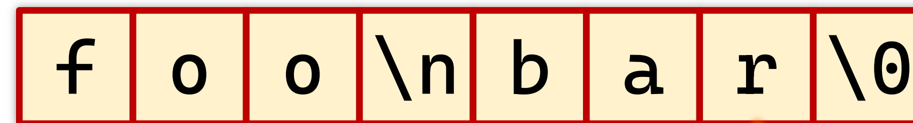


control characters

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

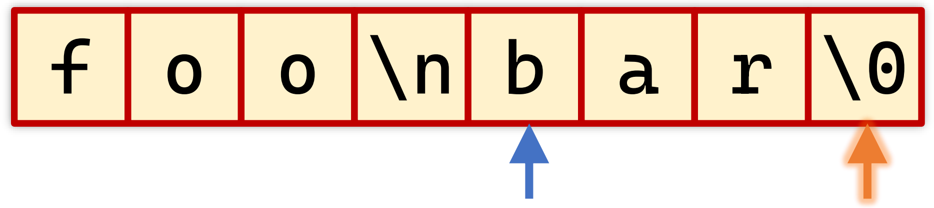
control characters

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

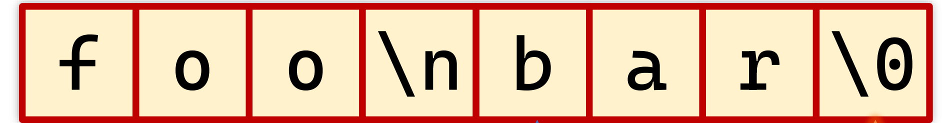


control characters

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

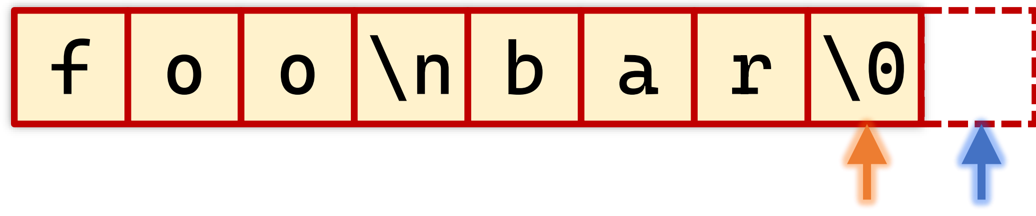


```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
```




```
for (;;) {
  //...
  if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
    writeAndAdvance(i, escaped.get(*i), 1);
    break;
  }
  //...
} // for (;;)
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
```

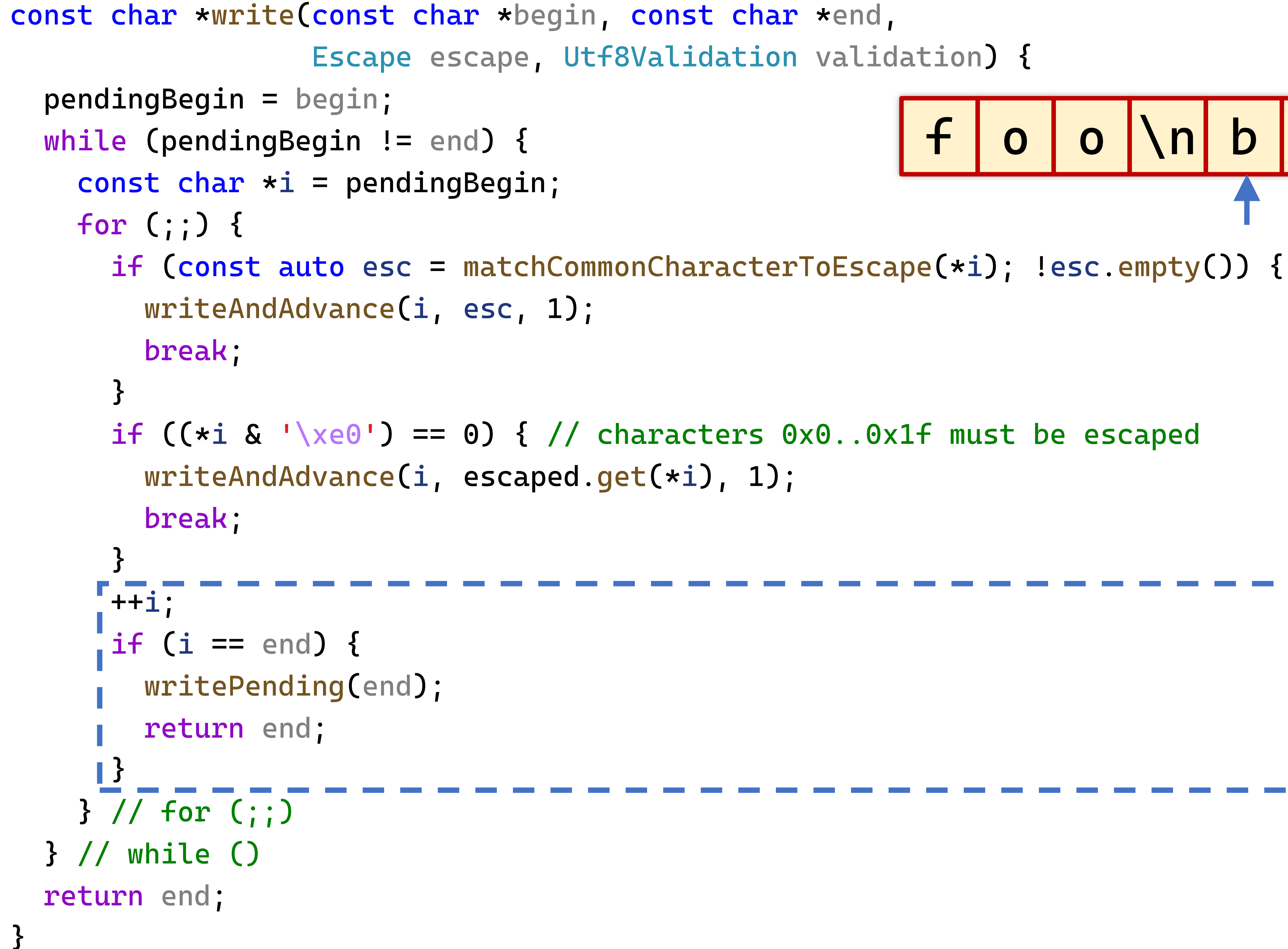



```
for (;;) {
  //...
  if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
    writeAndAdvance(i, escaped.get(*i), 1);
    break;
  }
  //...
} // for (;;)
```

```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }

  private:
    static void write(char *p, uint16_t c) {
      p[0] = '\\'; p[1] = 'u';
      constexpr char digits[] =
        { '0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f' };
      p[2] = digits[c >> 12];             p[3] = digits[(c >> 8) & 0xfu];
      p[4] = digits[(c >> 4) & 0xfu]; p[5] = digits[c & 0xfu];
    }
    char buf[12];
  };
}
```

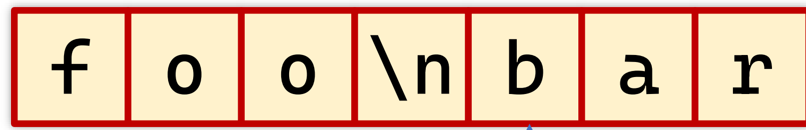
```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }

  private:
    static void write(char *p, uint16_t c) {
      p[0] = '\\'; p[1] = 'u';
      constexpr char digits[] =
        { '0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f' };
      p[2] = digits[c >> 12];          p[3] = digits[(c >> 8) & 0xfu];
      p[4] = digits[(c >> 4) & 0xfu]; p[5] = digits[c & 0xfu];
    }
    char buf[12];
  };
}
```

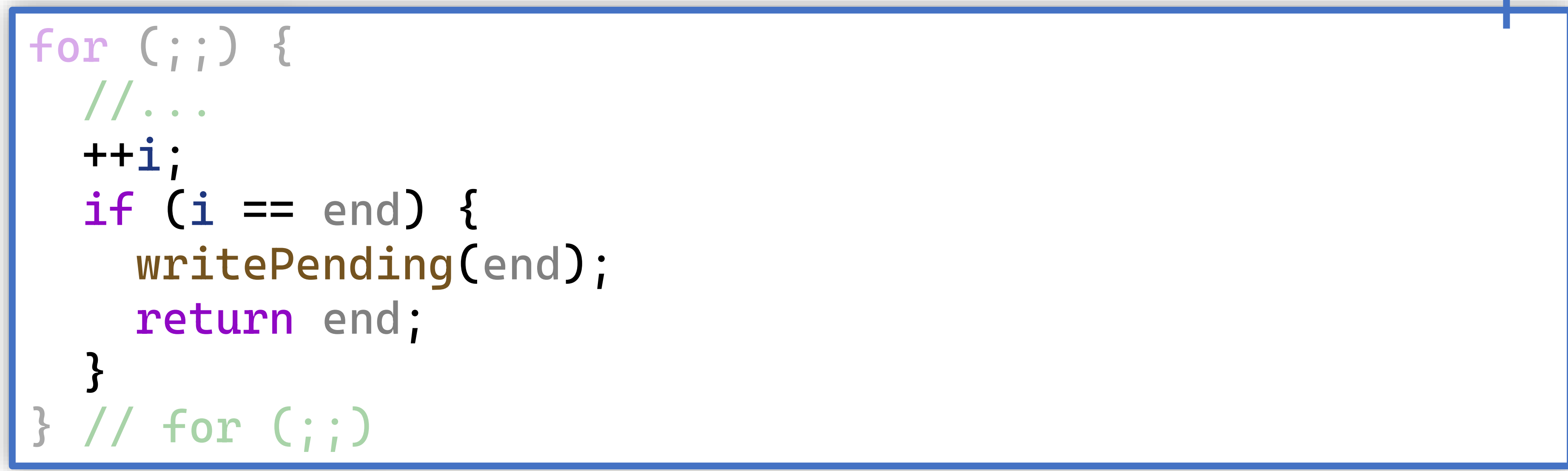
\0 U+0 → \u0000




```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

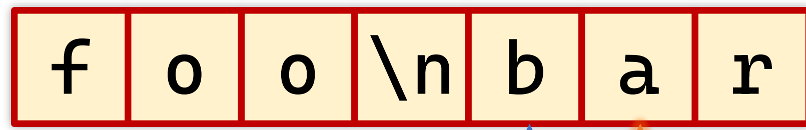
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
```

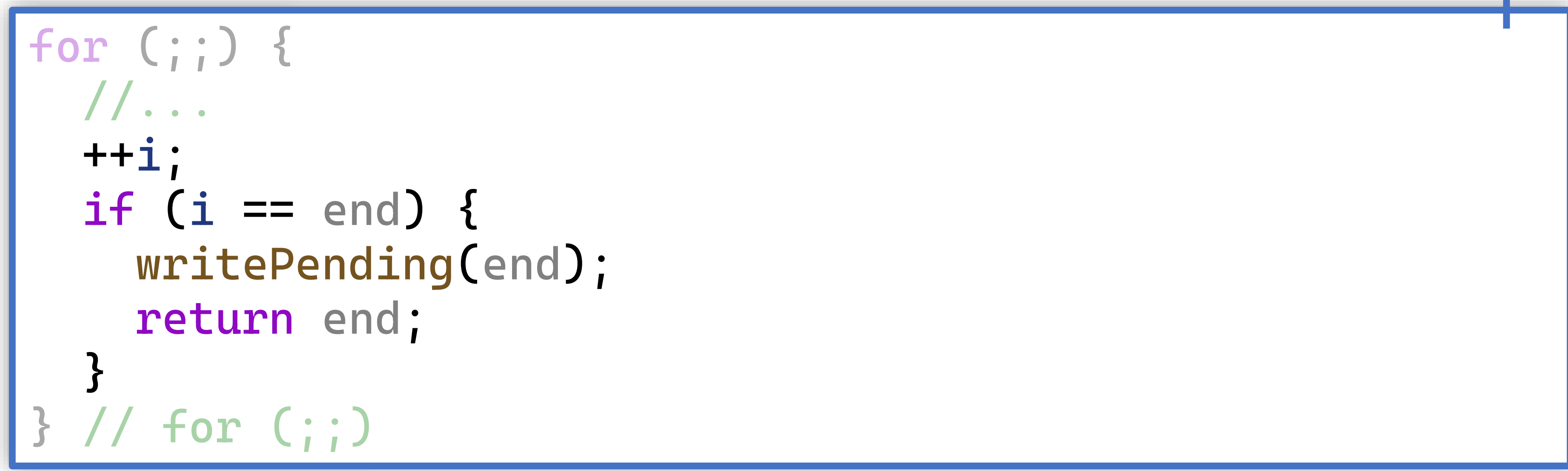




```
    } // for (;;)
  } // while ()
  return end;
}
```

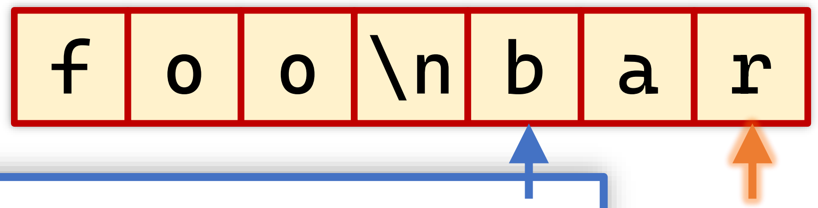
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
```

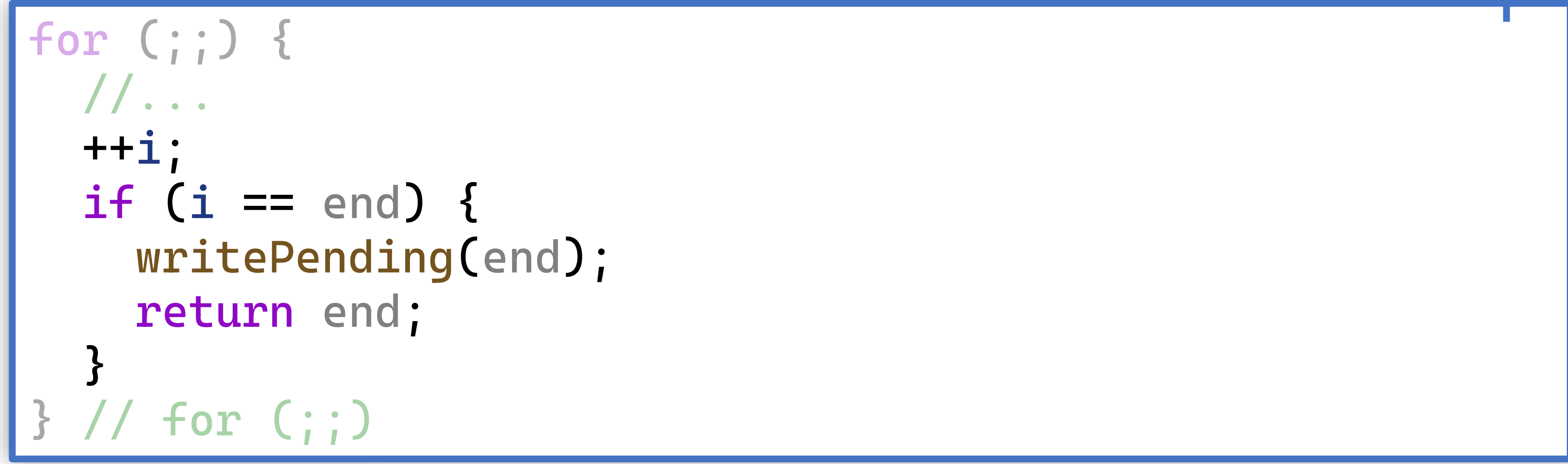




```
    } // for (;;)
  } // while ()
  return end;
}
```

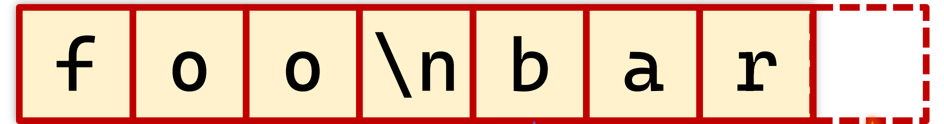
```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
```





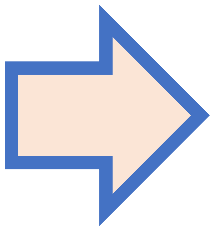
```
    } // for (;;)
  } // while ()
  return end;
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      //...
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
    ++i;
    if (i == end) {
      writePending(end);
      return end;
    }
  } // for (;;)
  } // while ()
  return end;
}
```



```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  pendingBegin = begin;
  while (pendingBegin != end) {
    const char *i = pendingBegin;
    for (;;) {
      if (const auto esc = matchCommonCharacterToEscape(*i); !esc.empty()) {
        writeAndAdvance(i, esc, 1);
        break;
      }
      if ((*i & '\xe0') == 0) { // characters 0x0..0x1f must be escaped
        writeAndAdvance(i, escaped.get(*i), 1);
        break;
      }
      ++i;
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  } // while ()
  return end;
}
```

validation
goes here



UTF-8 code unit		encoded code points range
0zzzzzzz	1 byte code point	U+0000..U+007F
10zzzzzz	continuation code unit	
110zzzzz	starts 2 byte code point	U+0080..U+07FF
1110zzzz	starts 3 byte code point	U+0800..U+FFFF
11110zzz	starts 4 byte code point	U+10000..U+10FFFF

UTF-8 crash course

UTF-8 code unit		encoded code points range
0zzzzzzz	1 byte code point	U+0000..U+007F
10zzzzzz	continuation code unit	
110zzzzz	starts 2 byte code point	U+0080..U+07FF
1110zzzz	starts 3 byte code point	U+0800..U+FFFF
11110zzz	starts 4 byte code point	U+10000..U+10FFFF

	Unicode code point in binary	UTF-8 code units
$ U+0024	00100100	00100100
£ U+00A3	00000000 10100011	11000010 10100011
€ U+20AC	00100000 10101100	11100010 10000010 10101100
🤓 U+1F913	01 11111001 00010011	11110000 10011111 10100100 10010011

UTF-8 crash course

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      //... common and control character escaping
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          if (codePointSize == 0)
            return i;
          i += codePointSize;
          goto CheckEnd;
        }
      }
      ++i;
    CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          if (codePointSize == 0)
            return i;
          i += codePointSize;
          goto CheckEnd;
        }
      }
      ++i;
    CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          if (codePointSize == 0)
            return i;
          i += codePointSize;
          goto CheckEnd;
        }
      }
      ++i;
    CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
  if (*i & '\x80') { // handling UTF-8 multibyte code point
    const size_t codePointSize = detectUtf8CodePointSize(i, end);
    if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
      if (codePointSize == 0)
        return i;
      i += codePointSize;
      goto CheckEnd;
    }
  }
  ++i;
CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          if (codePointSize == 0)
            return i;
          i += codePointSize;
          goto CheckEnd;
        }
      }
      ++i;
    CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          if (codePointSize == 0)
            return i;
          i += codePointSize;
          goto CheckEnd;
        }
      }
      ++i;
    CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
    if (*i & '\x80') { // handling UTF-8 multibyte code point
      const size_t codePointSize = detectUtf8CodePointSize(i, end);
      if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
        if (codePointSize == 0)
          return i;
        i += codePointSize;
        goto CheckEnd;
      }
    }
    ++i;
CheckEnd:
      if (i == end) {
        writePending(end);
        return end;
      }
    } // for (;;)
  //...
}
```

```
size_t meh_detectUtf8CodepointLength(char c) {
 if ((c & 0x80) == 0)
    return 1;
  if ((c & 0xE0) == 0xC0)
    return 2;
  if ((c & 0xF0) == 0xE0)
    return 3;
  if ((c & 0xF8) == 0xF0)
    return 4;
  return 0;
}
```

Typical implementation on the internets

```
size_t meh_detectUtf8CodepointLength(char c) {
 if ((c & 0x80) == 0)
    return 1;
  if ((c & 0xE0)? == 0xC0)?
    return 2;
  if ((c & 0xF0)? == 0xE0)?
    return 3;
  if ((c & 0xF8)? == 0xF0)?
    return 4;
  return 0;
}
```

Typical implementation on the internets

```
size_t meh_detectUtf8CodepointLength(char c) {
 if ((c & 0x80) == 0)
    return 1;
  if ((c & 0xE0)? == 0xC0)?
    return 2;
  if ((c & 0xF0)? == 0xE0)?
    return 3;
  if ((c & 0xF8)? == 0xF0)?
    return 4;
  return 0;
}
```

Typical implementation on the internets

```
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;
  }

  size_t detectUtf8CodePointSize(const char *begin, const char *end) {
    const size_t expectedSize = getExpectedUtf8CodePointSize(*begin);
    if (expectedSize != 1 &&
        static_cast<size_t>(end - begin) >= expectedSize &&
        std::all_of(begin + 1, begin + expectedSize, &isUtf8CodeUnit<1>))
      return expectedSize;
    return 0;
  }
}
```

```
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;
  }

  size_t detectUtf8CodePointSize(const char *begin, const char *end) {
    const size_t expectedSize = getExpectedUtf8CodePointSize(*begin);
    if (expectedSize != 1 &&
        static_cast<size_t>(end - begin) >= expectedSize &&
        std::all_of(begin + 1, begin + expectedSize, &isUtf8CodeUnit<1>))
      return expectedSize;
    return 0;
  }
}
```

```
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;
  }

  size_t detectUtf8CodePointSize(const char *begin, const char *end) {
    const size_t expectedSize = getExpectedUtf8CodePointSize(*begin);
    if (expectedSize != 1 &&
        static_cast<size_t>(end - begin) >= expectedSize &&
        std::all_of(begin + 1, begin + expectedSize, &isUtf8CodeUnit<1>))
      return expectedSize;
    return 0;
  }
}
```

```
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;
  }

  size_t detectUtf8CodePointSize(const char *begin, const char *end) {
    const size_t expectedSize = getExpectedUtf8CodePointSize(*begin);
    if (expectedSize != 1 &&
        static_cast<size_t>(end - begin) >= expectedSize &&
        std::all_of(begin + 1, begin + expectedSize, &isUtf8CodeUnit<1>))
      return expectedSize;
    return 0;
  }
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      //... common and control character escaping
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (escape == Escape::NonAscii) {
          //... escape non-ASCII and validate
        }
        else if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          //...
        }
      }
      //...
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
    for (;;) {
      //... common and control character escaping
      if (*i & '\x80') { // handling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (escape == Escape::NonAscii) {
          //... escape non-ASCII and validate
        }
        else if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          //...
        }
      }
      //...
    } // for (;;)
  //...
}
```

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
if (escape == Escape::NonAscii) {
  if (codePointSize != 0) {
    const auto escapeSequence = std::invoke([this, codePointSize, i] {
      switch (codePointSize) {
      default: // avoids compilation error
      case 2: return escaped.get(gatherBits<uint16_t, 5, 6>(i));
      case 3: return escaped.get(gatherBits<uint16_t, 4, 6, 6>(i));
      case 4:
        return escaped.getSurrogates(gatherBits<uint32_t, 3, 6, 6, 6>(i));
      }
    });
    writeAndAdvance(i, escapeSequence, codePointSize);
    break; // for (;;)
  }
  else if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
    return i;
  }
}
```

```
uint32_t meh_decode4ByteUtf8Codepoint(const char *bytes) {
  return
    static_cast<char32_t>(bytes[0] & 0b0000'0111) << 18 |
    static_cast<char32_t>(bytes[1] & 0b0011'1111) << 12 |
    static_cast<char32_t>(bytes[2] & 0b0011'1111) << 06 |
    static_cast<char32_t>(bytes[3] & 0b0011'1111);
}
```

Typical implementation on the internets

```
uint32_t meh_decode4ByteUtf8Codepoint(const char *bytes) {
  return
    static_cast<char32_t>(bytes[0] & 0b0000'0111) << 18 ?
    static_cast<char32_t>(bytes[1] & 0b0011'1111) << 12 ?
    static_cast<char32_t>(bytes[2] & 0b0011'1111) << 06 ?
    static_cast<char32_t>(bytes[3] & 0b0011'1111);
}
```

Typical implementation on the internets

```
namespace detail {
  template<size_t Size>
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail { 3
  template<size_t Size>
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

3

0b00000111 (0x7)

```
namespace detail {
  template<size_t Size>
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail { 3
  template<size_t Size>                    0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail {
  template<size_t Size>
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

3

0b00000111 (0x7)

3

```
namespace detail {  3
  template<size_t Size>                          0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                          3            6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail { 3
  template<size_t Size>                                  0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                              3          6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {             *p == 0b11110000 (0xF0)
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

3

0b00000111 (0x7)

3

6, 6, 6

*p == 0b11110000 (0xF0)

```
namespace detail {
  template<size_t Size>
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail { 3
  template<size_t Size>                              0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                        3           6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {            *p == 0b11110000 (0xF0)
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail { 3
  template<size_t Size>                          0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                             3            6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {          *p == 0b11110000 (0xF0)
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {      result = *p & 0x7
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail { 3
  template<size_t Size>                              0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                           3           6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {             *p == 0b11110000 (0xF0)
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {       result = *p & 0x7
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail {
  template<size_t Size>
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;

  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {
      return result;
    }
    else {
      return
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

3

0b00000111 (0x7)

3

6, 6, 6

*p == 0b11110000 (0xF0)

result = *p & 0x7

```
namespace detail {  3
  template<size_t Size>                              0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                             3           6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {          *p == 0b11110000 (0xF0)
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {     result = *p & 0x7
      return result;
    }
    else {
      return          6 + 6 + 6 = 18
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

```
namespace detail {  3
  template<size_t Size>                         0b00000111 (0x7)
  inline constexpr uint32_t LowerBitsMask = (1u << Size) - 1u;
                                        3           6, 6, 6
  template<typename Int_t, size_t Size0, size_t... Size1>
  Int_t gatherBits(const char *p) {          *p == 0b11110000 (0xF0)
    static_assert((Size0 + ... + Size1) <= sizeof(Int_t) * 8);
    const Int_t result = static_cast<Int_t>(*p) & LowerBitsMask<Size0>;
    if constexpr (sizeof...(Size1) == 0) {     result = *p & 0x7
      return result;
    }
    else {
      return          6 + 6 + 6 = 18                    6, 6, 6
        (result << (Size1 + ...)) | gatherBits<Int_t, Size1...>(p + 1);
    }
  }
}
```

UTF-8 code unit		encoded code points range
…		17 bits · 21 bits
`11110zzz`	starts 4 byte code point	U+10000..U+10FFFF

`\u`XXXX on its own can only encode code points up to U+FFFF · 16 bits

UTF-16 surrogates crash course

UTF-8 code unit		encoded code points range
…		17 bits 21 bits
11110zzz	starts 4 byte code point	U+10000..U+10FFFF

\uXXXX on its own can only encode code points up to U+FFFF 16 bits

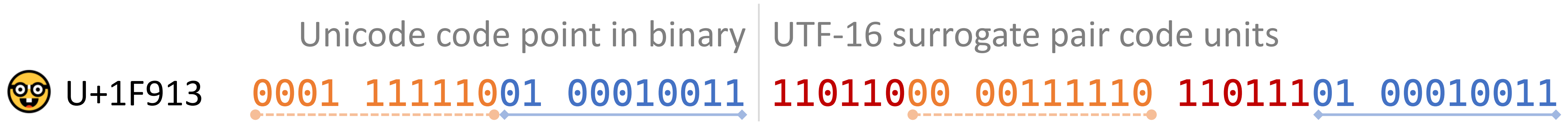
UTF-16 code unit		range of values
110110zz zzzzzzzz	high surrogate	0xD800..0xDBFF
110111zz zzzzzzzz	low surrogate	0xDC00..0xDFFF

UTF-16 surrogates crash course

UTF-8 code unit		encoded code points range
…		17 bits / 21 bits
11110zzz	starts 4 byte code point	U+10000..U+10FFFF

\uXXXX on its own can only encode code points up to U+FFFF — 16 bits

UTF-16 code unit		range of values
110110zz zzzzzzzz	high surrogate	0xD800..0xDBFF
110111zz zzzzzzzz	low surrogate	0xDC00..0xDFFF



UTF-16 surrogates crash course

```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }
    std::string_view getSurrogates(uint32_t c) {
      const uint32_t surrogate = c - 0x10000u;
      write(buf, static_cast<uint16_t>((surrogate >> 10) | 0xd800u));
      write(buf + 6,
        static_cast<uint16_t>(surrogate & LowerBitsMask<10> | 0xdc00u));
      return { buf, 12 };
    }
  private:
    //...
    char buf[12];
  };
}
```

```
namespace detail {
  struct Esca
    std::stri
      write(b
      return
    }
    std::string_view getSurrogates(uint32_t c) {
      const uint32_t surrogate = c - 0x10000u;
      write(buf, static_cast<uint16_t>((surrogate >> 10) | 0xd800u));
      write(buf + 6,
        static_cast<uint16_t>(surrogate & LowerBitsMask<10> | 0xdc00u));
      return { buf, 12 };
    }
  private:
    //...
    char buf[12];
  };
}
```

```
           17 bits    21 bits
c ∈ [ 0x10000..0x10FFFF ]

                      1 bit 20 bits
(c - 0x10000) ∈ [ 0x0..0xFFFFF ]
```

```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }
    std::string_view getSurrogates(uint32_t c) {
      const uint32_t surrogate = c - 0x10000u;
      write(buf, static_cast<uint16_t>((surrogate >> 10) | 0xd800u));
      write(buf + 6,
        static_cast<uint16_t>(surrogate & LowerBitsMask<10> | 0xdc00u));
      return { buf, 12 };
    }
  private:
    //...
    char buf[12];
  };
}
```

```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }
    std::string_view getSurrogates(uint32_t c) {
      const uint32_t surrogate = c - 0x10000u;
      write(buf, static_cast<uint16_t>((surrogate >> 10) | 0xd800u));
      write(buf + 6,
        static_cast<uint16_t>(surrogate & LowerBitsMask<10> | 0xdc00u));
      return { buf, 12 };
    }
  private:
    //...
    char buf[12];
  };
}
```

```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }
    std::string_view getSurrogates(uint32_t c) {
      const uint32_t surrogate = c - 0x10000u;
      write(buf, static_cast<uint16_t>((surrogate >> 10) | 0xd800u));
      write(buf + 6,
        static_cast<uint16_t>(surrogate & LowerBitsMask<10> | 0xdc00u));
      return { buf, 12 };
    }
  private:
    //...
    char buf[12];
  };
}
```

```
namespace detail {
  struct EscapedChar final {
    std::string_view get(uint16_t c) {
      write(buf, c);
      return { buf, 6 };
    }
    std::string_view getSurrogates(uint32_t c) {
      const uint32_t surrogate = c - 0x10000u;
      write(buf, static_cast<uint16_t>((surrogate >> 10) | 0xd800u));
      write(buf + 6,
        static_cast<uint16_t>(surrogate & LowerBitsMask<10> | 0xdc00u));
      return { buf, 12 };
    }
  private:
    //...
    char buf[12];
  };
}
```

U+1F913 → \ud83e\udd13

```
const char *write(const char *begin, const char *end,
                  Escape escape, Utf8Validation validation) {
  //...
      //... common and control character escaping
      if (*i & '\x80') { andling UTF-8 multibyte code point
        const size_t codePointSize = detectUtf8CodePointSize(i, end);
        if (escape == Escape::NonAscii) {
          if (codePointSize != 0) {
            //...
          }
          else if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
            return i;
          }
        }
        else if (validation == Utf8Validation::FailOnInvalidUtf8CodeUnits) {
          //...
        }
      }
  //...
}
```

```
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    //...
  private:
    enum Mode {
      Default = 0,
      EscapeNonAscii = 1,
      ValidateUtf8 = 1 << 1,
      EscapeNonAsciiAndValidateUtf8 = EscapeNonAscii | ValidateUtf8
    };
    //...
  };
}
```

JSON string escaping

```
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    //...
  private:
    //...
    template<Mode mode>
    const char *writeImpl(const char *begin, const char *end) {
      //...
    }

    //...
  };
}
```

JSON string escaping

```
template<Mode mode>
const char *writeImpl(const char *begin, const char *end) {
  //...
      //... common and control character escaping
      if constexpr (mode != Default) {
        if (*i & '\x80') { // handling UTF-8 multibyte code point
          const size_t codePointSize = detectUtf8CodePointSize(i, end);
          if constexpr (mode & EscapeNonAscii) {
            if (codePointSize != 0) {
              //...
            }
            else if constexpr (mode & ValidateUtf8) {
              //...
            }
          }
          else /* mode == ValidateUtf8 */ {
            //...
          }
        }
      }
  //...
}
```

```
template<Mode mode>
const char *writeImpl(const char *begin, const char *end) {
  //...
      //... common and control character escaping
if constexpr (mode != Default) {
  if (*i & '\x80') { // handling UTF-8 multibyte code point
    const size_t codePointSize = detectUtf8CodePointSize(i, end);
    if constexpr (mode & EscapeNonAscii) {
      if (codePointSize != 0) {
        //...
      }
      else if constexpr (mode & ValidateUtf8) {
        //...
      }
    }
    else /* mode == ValidateUtf8 */ {
      //...
    }
  }
}
  //...
}
```

```
template<Mode mode>
const char *writeImpl(const char *begin, const char *end) {
  //...
      //... common and control character escaping
if constexpr (mode != Default) {
  if (*i & '\x80') { // handling UTF-8 multibyte code point
    const size_t codePointSize = detectUtf8CodePointSize(i, end);
    if constexpr (mode & EscapeNonAscii) {
      if (codePointSize != 0) {
        //...
      }
      else if constexpr (mode & ValidateUtf8) {
        //...
      }
    }
    else /* mode == ValidateUtf8 */ {
      //...
    }
  }
}
  //...
}
```

```
template<Mode mode>
const char *writeImpl(const char *begin, const char *end) {
  //...
      //... common and control character escaping
if constexpr (mode != Default) {
  if (*i & '\x80') { // handling UTF-8 multibyte code point
    const size_t codePointSize = detectUtf8CodePointSize(i, end);
    if constexpr (mode & EscapeNonAscii) {
      if (codePointSize != 0) {
        //...
      }
      else if constexpr (mode & ValidateUtf8) {
        //...
      }
    }
    else /* mode == ValidateUtf8 */ {
      //...
    }
  }
}
  //...
}
```

```
template<Mode mode>
const char *writeImpl(const char *begin, const char *end) {
  //...
      //... common and control character escaping
if constexpr (mode != Default) {
  if (*i & '\x80') { // handling UTF-8 multibyte code point
    const size_t codePointSize = detectUtf8CodePointSize(i, end);
    if constexpr (mode & EscapeNonAscii) {
      if (codePointSize != 0) {
        //...
      }
      else if constexpr (mode & ValidateUtf8) {
        //...
      }
    }
    else /* mode == ValidateUtf8 */ {
      //...
    }
  }
}
  //...
}
```

```
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    //...
    const char *write(const char *begin, const char *end,
                      Escape escape, Utf8Validation validation) {
      if (escape == Escape::Default) {
        return validation == Utf8Validation::FailOnInvalidUtf8CodeUnits ?
          writeImpl<ValidateUtf8>(begin, end) :
          writeImpl<Default>(begin, end);
      }
      return validation == Utf8Validation::FailOnInvalidUtf8CodeUnits ?
        writeImpl<EscapeNonAsciiAndValidateUtf8>(begin, end) :
        writeImpl<EscapeNonAscii>(begin, end);
    }
    //...
  };
}
```

```
namespace detail {
  template<typename Sink>
  struct EscapedStringWriter final {
    //...
    const char *write(const char *begin, const char *end,
                      Escape escape, Utf8Validation validation) {
      if (escape == Escape::Default) {
        return validation == Utf8Validation::FailOnInvalidUtf8CodeUnits ?
          writeImpl<ValidateUtf8>(begin, end) :
          writeImpl<Default>(begin, end);
      }
      return validation == Utf8Validation::FailOnInvalidUtf8CodeUnits ?
        writeImpl<EscapeNonAsciiAndValidateUtf8>(begin, end) :
        writeImpl<EscapeNonAscii>(begin, end);
    }
    //...
  };
}
```

```
//                 "$ \xc2\xa3 \xe2\x82\xac \xf0\x9f\xa4\x93"sv;
const auto text = "$ £ € 🤓"sv;

std::cout << escape(text) << '\n';

std::cout << escape(text, Escape::EscapeNonAscii) << '\n';
```

output:

```
$ £ € 🤓
$ \u00a3 \u20ac \ud83e\udd13
```

JSON string escaping

Thanks for listening

minjsoncpp

Minimalistic JSON C++ library

https://github.com/toughengineer/minjsoncpp

JSON in C++: escaping and serialization

Pavel Novikov

Slides: [bit.ly/3Wv28mV](bit.ly/3Wv28mV)

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,   ← null
             bool,
             int64_t,
             double,
             std::string,
             std::vector<Value>,
             std::unordered_map<String, Value>>
```

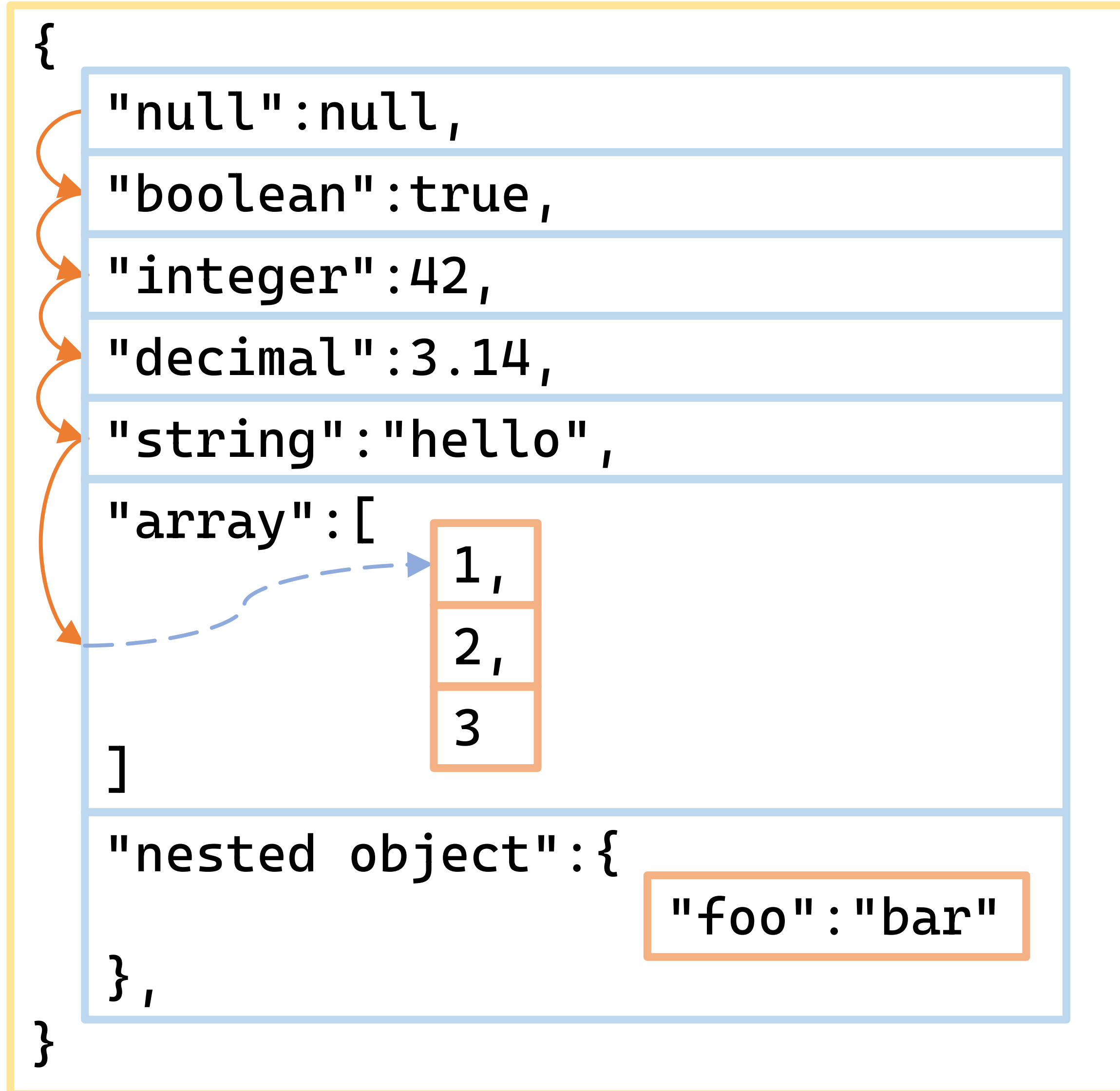
Mapping of JSON types into C++

```
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;
};
```

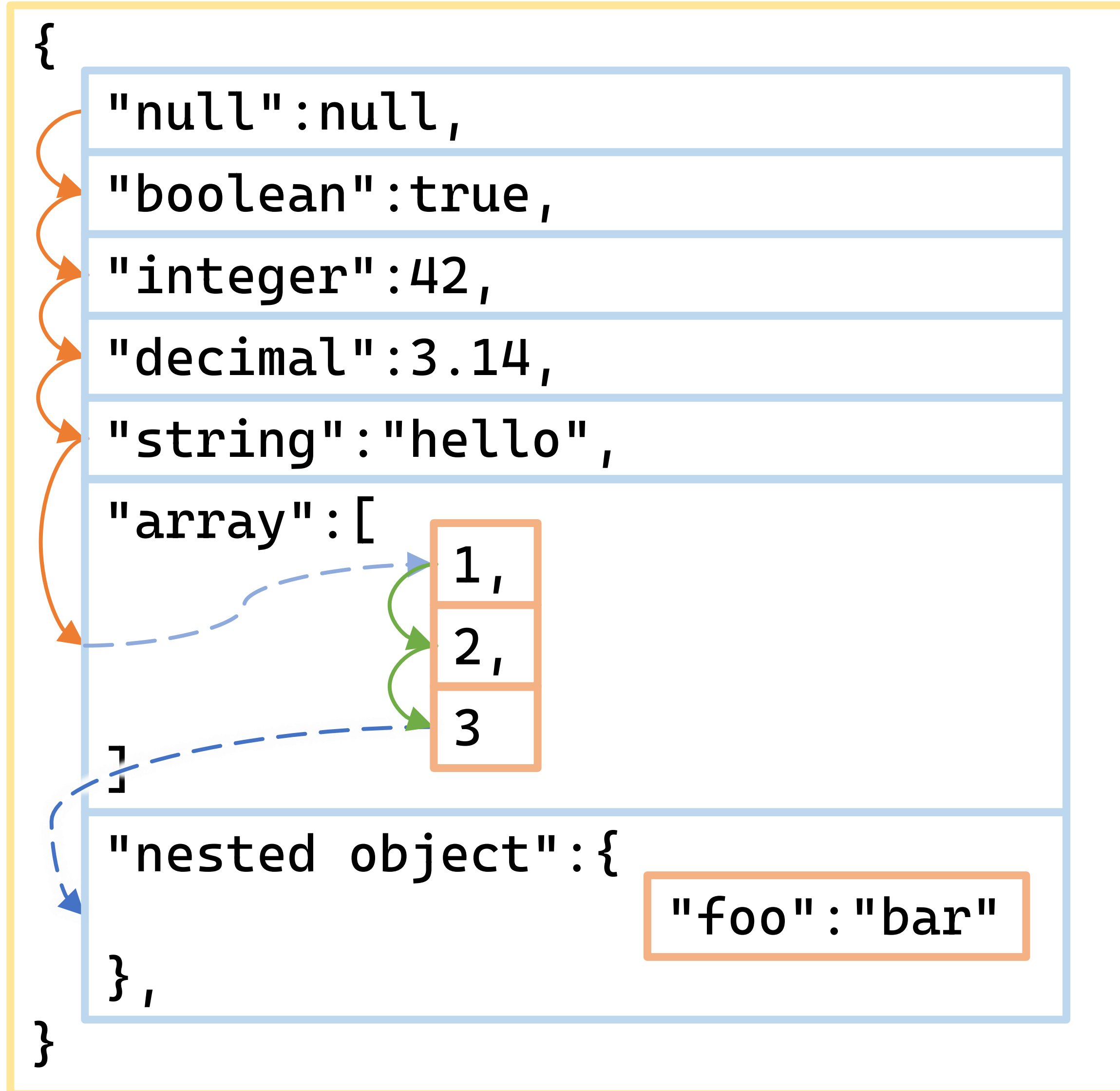
Value type

```
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);
  }

  //...
};
```

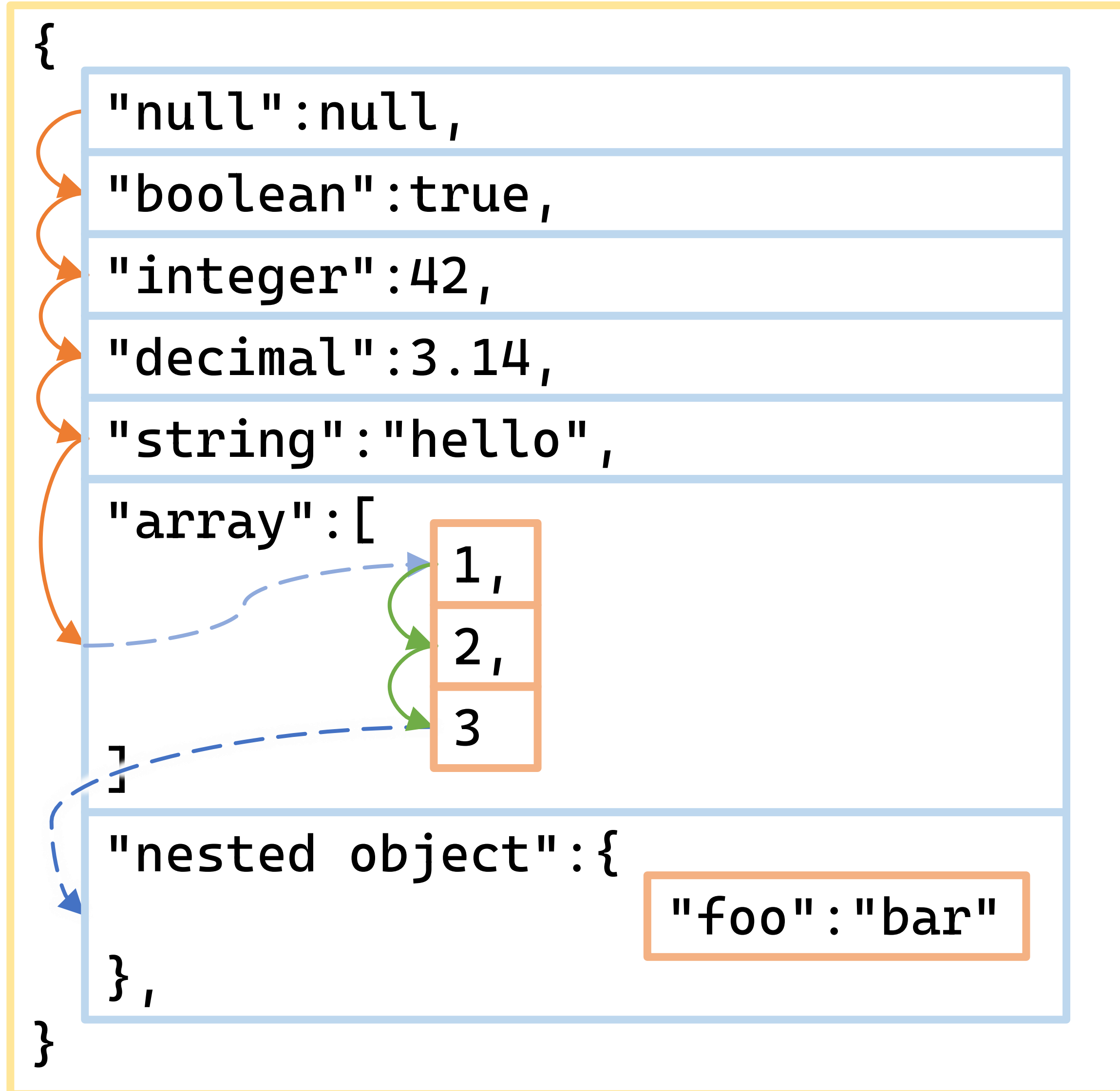
`Value` type



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

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

Serialization

```
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 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;
}
```

```cpp
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
```

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

1234.5678

not

1234.567800000000033833202905952930450439453125

```
      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()(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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     s.substr(escapedSize, reportedSize),
                                     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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     s.substr(escapedSize, reportedSize),
                                     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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     s.substr(escapedSize, reportedSize),
                                     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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     s.substr(escapedSize, reportedSize),
                                     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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     s.substr(escapedSize, reportedSize),
                                     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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                     s.substr(escapedSize, reportedSize),
                                     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 size_t expectedCodePointSize =
      getExpectedUtf8CodePointSize(s[escapedSize]);
    const size_t reportedSize =
      std::min(expectedCodePointSize, s.size() - escapedSize);
    throw InvalidUtf8CodeUnitsError{ "invalid UTF-8 code units",
                                      s.substr(escapedSize, reportedSize),
                                      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

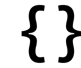
```
struct InvalidUtf8CodeUnitsError : std::runtime_error {
  InvalidUtf8CodeUnitsError(const char *msg,
                            const std::string_view &codeUnits,
                            size_t offset) :
    runtime_error{ msg },
    codeUnits{ codeUnits },
    offset{ offset }
  {}
  const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

Serialization

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

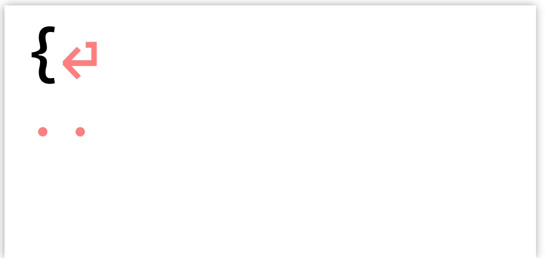
Serialization

```
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"
```

```cpp
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 });
```

indent

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);
```

Serialization

output:

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

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

output:

```json
{
  "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;
    options.indent = 2;
```

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;'
    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

Спасибо за внимание!

Павел Новиков

C++-разработчик

Яндекс

References

- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format https://datatracker.ietf.org/doc/html/rfc8259
- ANSI escape code: Colors https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
- minjsoncpp — Minimalistic JSON C++ library https://github.com/toughengineer/minjsoncpp