The Surprising Complexity of Formatting Ranges

BARRY REVZIN

About Me

C++ Software Developer at Jump Trading since 2014

About Me

C++ Software Developer at Jump Trading since 2014

WG21 participant since 2016

- C++20: `<=>`, `[...args=args]{}`, `explicit(bool)`, conditionally trivial
- C++23: Deducing `this`, `if consteval`, bunch of `constexpr` and ranges papers

About Me

C++ Software Developer at Jump Trading since 2014

WG21 participant since 2016

- C++20: `<=>`, `[...args=args]{}`, `explicit(bool)`, conditionally trivial
- C++23: Deducing `this`, `if consteval`, bunch of `constexpr` and ranges papers

https://brevzin.github.io/

@BarryRevzin

Barry

SWIM SWAM

In the beginning, there was `printf`

```
std::printf("The price of %x is %d\n", 48879, 1234);
```

```
The price of beef is 1234
```

In the beginning, there was `printf`

```
std::printf("The price of %X is %d\n", 48879, 1234);
```

```
The price of BEEF is 1234
```

In the beginning, there was `printf`

```
std::printf("The price of %#X is %d\n", 48879, 1234);
```

```
The price of 0XBEEF is 1234
```

In the beginning, there was `printf`

Specification mini-language

- %[*flags*][*width*][.*precision*][*size*]*type*

Error prone

Non-extensible

```
struct Point {
  int x;
  int y;
};

void show(Point p) {
  std::printf("Point p is at %??\n", p);
}
```

Then C++ introduced iostreams

```
std::cout << "The price of "
          << 48879
          << " is "
          << 1234
          << '\n';
```

```
The price of 48879 is 1234
```

Then C++ introduced iostreams

```
std::cout << "The price of "
          << std::hex
          << 48879
          << " is "
          << 1234
          << '\n';
```

```
The price of beef is 4d2
```

Then C++ introduced iostreams

```
std::cout << "The price of "
          << std::hex << std::showbase << std::internal
          << std::uppercase << std::setfill('0')
          << std::setw(8)
          << 48879
          << " is "
          << 1234
          << '\n';
```

```
The price of 0X00BEEF is 0x4D2
             ^^^^^^^^    ^^^^^
                8           5
```

Then C++ introduced iostreams

Fixed set of manipulators (mostly sticky, error prone)

Extensible to user-defined types

Verbose

```
struct Point {
  int x;
  int y;

  friend auto operator<<(std::ostream& os, Point p)
    -> std::ostream&
  {
    return os << "(x=" << p.x << ", y=" << p.y << ')';
  }
};
```

Then C++ introduced iostreams

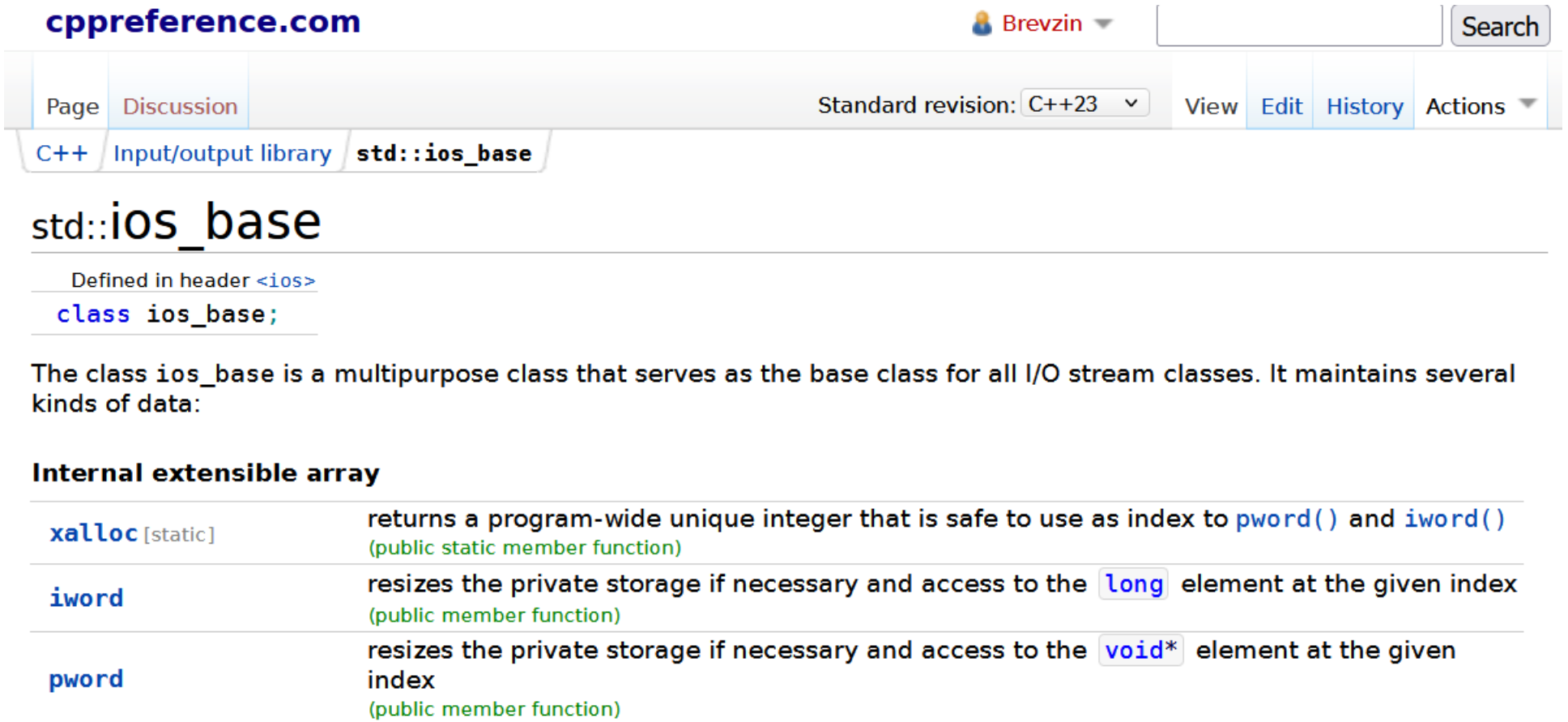
Fixed† set of manipulators (mostly sticky, error prone)

Extensible to user-defined types

Verbose

```
struct Point {
  int x;
  int y;

  friend auto operator<<(std::ostream& os, Point p)
    -> std::ostream&
  {
    return os << "(x=" << p.x << ", y=" << p.y << ')';
  }
};
```

Custom manipulators with iostreams



cppreference.com

Brevzin

Search

Page | Discussion | Standard revision: C++23 | View | Edit | History | Actions

C++ | Input/output library | std::ios_base

std::ios_base

Defined in header <ios>

```
class ios_base;
```

The class ios_base is a multipurpose class that serves as the base class for all I/O stream classes. It maintains several kinds of data:

Internal extensible array

xalloc [static]	returns a program-wide unique integer that is safe to use as index to pword() and iword() (public static member function)
iword	resizes the private storage if necessary and access to the long element at the given index (public member function)
pword	resizes the private storage if necessary and access to the void* element at the given index (public member function)

Then there was {fmt}

Intro to {fmt}

```
std::print("The price of {:x} is {}\n", 48879, 1234);
```

```
The price of beef is 1234
```

Intro to {fmt}

```
std::print("The price of {:#X} is {}\n", 48879, 1234);
```

```
The price of 0xBEEF is 1234
```

Intro to {fmt}

```
std::print("The price of {0:#X} is {1}\n", 48879, 1234);
```

```
The price of 0xBEEF is 1234
```

Intro to {fmt}

```
std::print("The price of {1:#X} is {0}\n", 1234, 48879);
```

```
The price of 0xBEEF is 1234
```

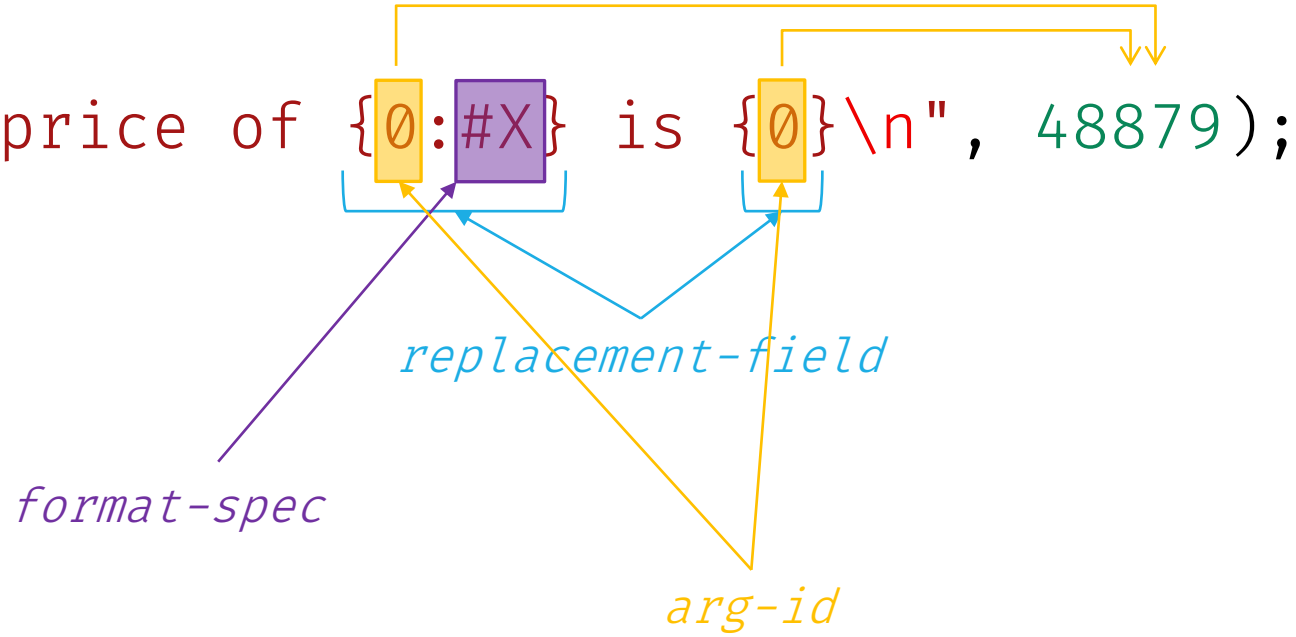
Intro to {fmt}

```
std::print("The price of {0:#X} is {0}\n", 48879);
```

```
The price of 0xBEEF is 48879
```

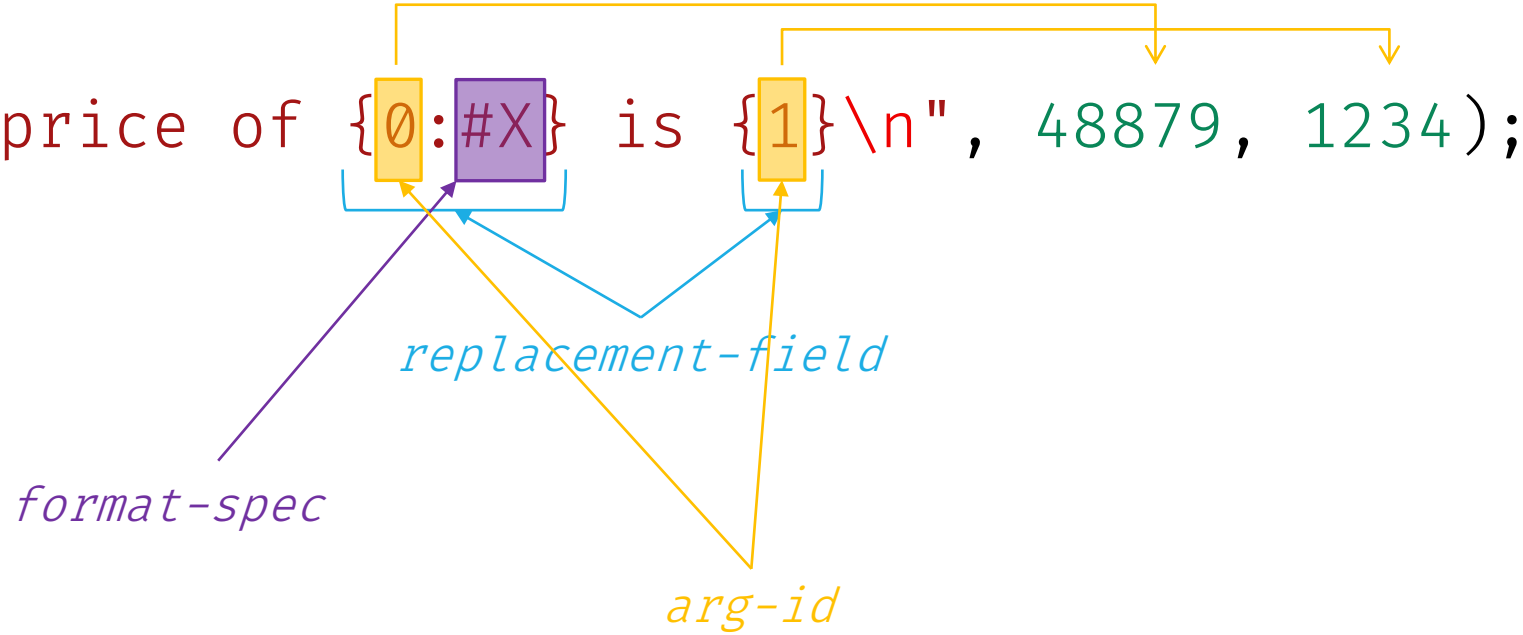
Intro to {fmt}

```
std::print("The price of {0:#X} is {0}\n", 48879);
```



Intro to {fmt}

```
std::print("The price of {0:#X} is {1}\n", 48879, 1234);
```



Intro to {fmt}

```
template <class T, class CharT=char>
struct formatter {
  template <class ParseContext>
  constexpr auto parse(ParseContext&)
    -> ParseContext::iterator;

  template <class FormatContext>
  auto format(T const&, FormatContext&) const
    -> FormatContext::iterator;
};
```

Intro to {fmt}

```
template <class T>
struct formatter {
  template <class ParseContext>
  constexpr auto parse(ParseContext&)
    -> ParseContext::iterator;

  template <class FormatContext>
  auto format(T const&, FormatContext&) const
    -> FormatContext::iterator;
};
```

Parse the *format-spec* (if any)

Emit representation

Parsing in {fmt}

DEALING WITH PARSE_CONTEXT

Parsing in {fmt}

```
template<class charT>
class basic_format_parse_context {
public:
  using char_type = charT;
  using const_iterator = basic_string_view<charT>::const_iterator;
  using iterator = const_iterator;

  constexpr const_iterator begin() const noexcept;
  constexpr const_iterator end() const noexcept;
  constexpr void advance_to(const_iterator it);

  constexpr size_t next_arg_id();
  constexpr void check_arg_id(size_t id);
};
```

Parsing in {fmt}

```
class format_parse_context {
public:
  using char_type = char;
  using const_iterator = string_view::const_iterator;
  using iterator = const_iterator;

  constexpr const_iterator begin() const noexcept;
  constexpr const_iterator end() const noexcept;
  constexpr void advance_to(const_iterator it);

  constexpr size_t next_arg_id();
  constexpr void check_arg_id(size_t id);
};
```

Basically a `string_view`

Automatic or Manual *arg-id* handling

Parsing in {fmt}

Format strings can be arbitrarily complicated

```
                    fill  align width
    std::print("{:*^{}}\n", "hi", 10);
```

```
****hi****
```

Parsing in {fmt}

Format strings can be arbitrarily complicated

fill align width

```
std::print("{0:*^{1}}\n", "hi", 10);
```

```
****hi****
```

Parsing in {fmt}

Format strings can be arbitrarily complicated

fill align width

```
std::print("{0:*^{1}}\n", "hi", 10);
```

```
****hi****
```

And can contain arbitrary characters

chrono-specs

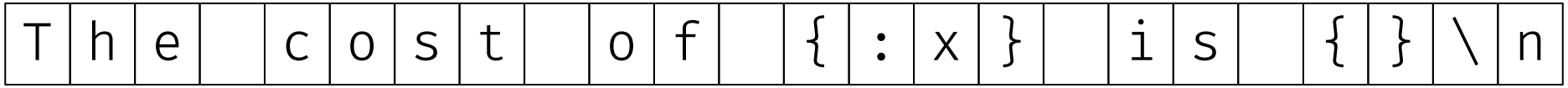
```
std::print("{:%Y-%m-%d %H:%M}\n", std::chrono::system_clock::now());
```

```
2022-08-07 16:49
```

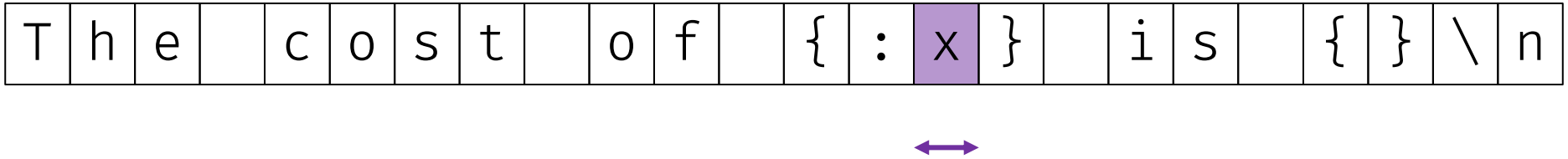
Parsing in {fmt}

```
std::print("The cost of {:x} is {}\n", 48879, 1234);
```

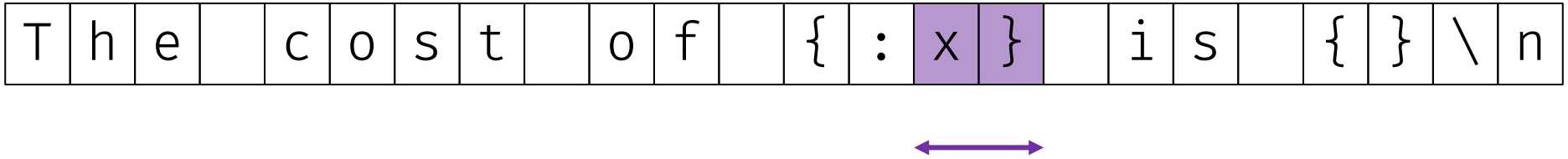
Parsing in {fmt}



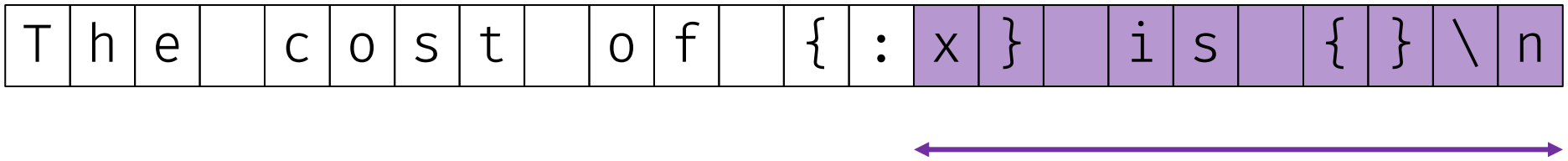
Parsing in {fmt}



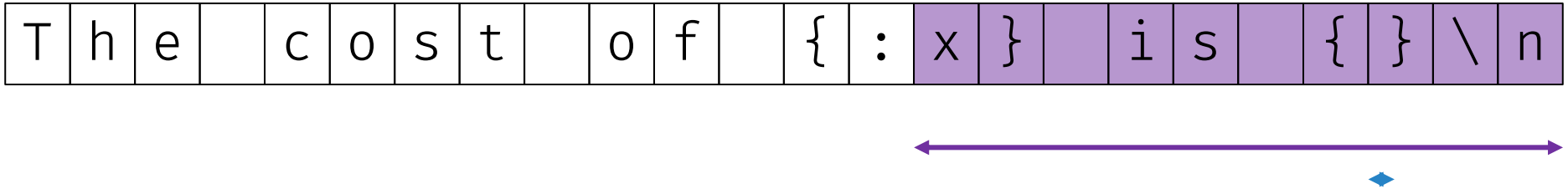
Parsing in {fmt}



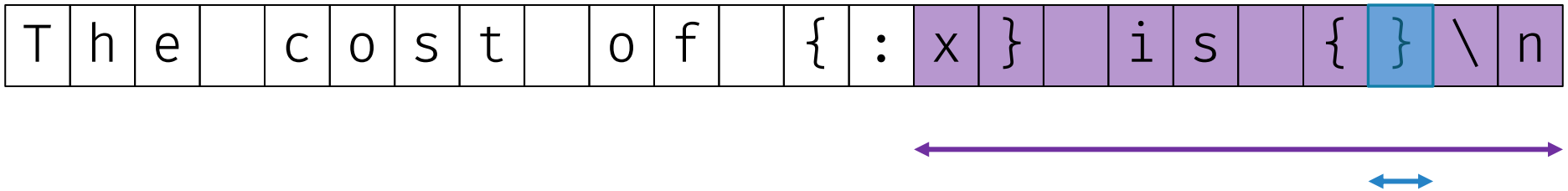
Parsing in {fmt}



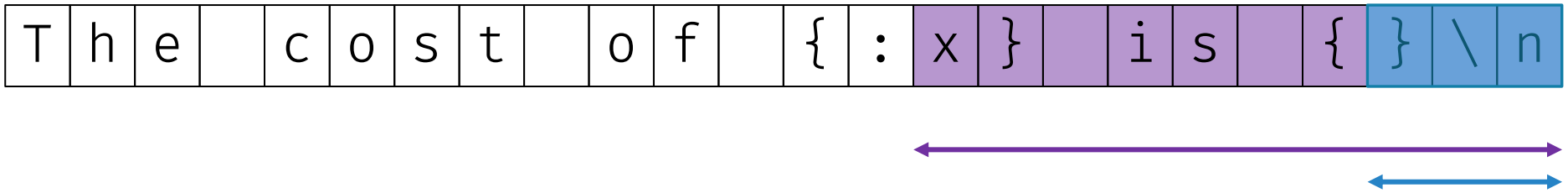
Parsing in {fmt}



Parsing in {fmt}



Parsing in {fmt}



Parsing in {fmt}

```
auto formatter<Point>::parse(auto& ctx) {
  // must have no format-spec
  return ctx.begin();
}
```

Parsing in {fmt}

```
template <> struct formatter<Point> {
  enum class coord {
    cartesian,
    polar
  };
  coord type = coord::cartesian;

  constexpr auto parse(auto& ctx);
};
```

Parsing in {fmt}

```
auto formatter<Point>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') {
    return it;
  }

  // coord type is just one character
  switch (*it++) {
    // ...
  }
  return it;
}
```

Parsing in {fmt}

```
auto formatter<Point>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // coord type is just one character
  this->type = [&]{
    switch (*it++) {
    case 'c':
    case 'r':
      return coord::cartesian;
    case 'p':
      return coord::polar;
    default:
      throw format_error("invalid type");
    }
  }();
  return it;
}
```

Formatting in {fmt}

USING WHAT WE PARSED

Formatting in {fmt}

```
template<class Out, class charT>
class basic_format_context {
public:
  using iterator = Out;
  using char_type = charT;
  template<class T> using formatter_type = formatter<T, charT>;

  basic_format_arg<basic_format_context> arg(size_t id) const noexcept;
  std::locale locale();

  iterator out();
  void advance_to(iterator it);
};
```

Formatting in {fmt}

```
template<class Out>
class basic_format_context {
public:
  using iterator = Out;
  using char_type = char;
  template<class T> using formatter_type = formatter<T, char>;

  basic_format_arg<basic_format_context> arg(size_t id) const noexcept;
  std::locale locale();

  iterator out();
  void advance_to(iterator it);
};
```

Formatting in {fmt}

```
template<class Out>
class basic_format_context {
public:
  using iterator = Out;
  using char_type = char;
  template<class T> using formatter_type = formatter<T, char>;

  basic_format_arg<basic_format_context> arg(size_t id) const noexcept;

  iterator out();
  void advance_to(iterator it);
};
```

Arbitrary `output_iterator<char const&>`

A large variant of several common types

Formatting in {fmt}

```
struct Char { char c; };

template <> struct formatter<Char> {
  constexpr auto parse(auto& ctx) {
    return ctx.begin();
  }

  auto format(Char c, auto& ctx) const {
    auto out = ctx.out();
    *out++ = c.c;
    return out;
  }
};
```

Formatting in {fmt}

```
template <>
struct formatter<Point> {
  enum class coord { cartesian, polar };
  coord type = coord::cartesian;

  constexpr auto parse(auto& ctx) { /* ... */ }

  auto format(Point p, auto& ctx) const {
    return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
  }
};
```

Formatting in {fmt}

```
template <>
struct formatter<Point> {
  enum class coord { cartesian, polar };
  coord type = coord::cartesian;

  constexpr auto parse(auto& ctx) { /* ... */ }

  auto format(Point p, auto& ctx) const {
    if (type == coord::cartesian) {
      return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
    } else {
      return format_to(ctx.out(), "(r={}, theta={})", p.r(), p.theta());
    }
  }
};
```

Formatting in {fmt}

```
std::print("Lagrange point is at {}", p);
```

```
Lagrange point is at (x=1, y=2)
```

Formatting in {fmt}

```
std::print("Lagrange point is at {:p}", p);
```

```
Lagrange point is at (r=2.23606797749979, theta=1.107148717794090 4)
```

Dynamic Formatting in {fmt}

USING *ARG-ID*

Dynamic Formatting in {fmt}

```
std::print("Lagrange point is at {:p}", p);
```

```
Lagrange point is at (r=2.23606797749979, theta=1.1071487177940904)
```

Dynamic Formatting in {fmt}

```
std::print("Lagrange point is at {:{}}", p, 'p');
```

```
Lagrange point is at (r=2.23606797749979, theta=1.1071487177940904)
```

Dynamic Formatting in {fmt}

```
std::print("Lagrange point is at {0:{1}}", p, 'p');
```

```
Lagrange point is at (r=2.23606797749979, theta=1.107148717794090 4)
```

Dynamic Formatting in {fmt}

```
std::print("Lagrange point is at {0:{1}}", p, 'r');
```

```
Lagrange point is at (x=1, y=2)
```

Dynamic Parsing in {fmt}

```
template <> struct formatter<Point> {
  enum class coord {
    cartesian,
    polar
  };
  coord type = coord::cartesian;

  constexpr auto parse(auto& ctx);
};
```

Dynamic Parsing in {fmt}

```
template <> struct formatter<Point> {
  enum class coord {
    cartesian,
    polar,
    dynamic
  };
  coord type = coord::cartesian;
  size_t arg_id = -1;

  constexpr auto parse(auto& ctx);
};
```

Dynamic Parsing in {fmt}

```
auto formatter<Point>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // coord type is just one character
  this->type = [&]{
    switch (*it++) {
    case 'c':
    case 'r':
      return coord::cartesian;
    case 'p':
      return coord::polar;
    default:
      throw format_error("invalid type");
    }
  }();
  return it;
}
```

Dynamic Parsing in {fmt}

```
auto formatter<Point>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  switch (*it++) {
  case 'c':
  case 'r':
    type = coord::cartesian;
    break;
  case 'p':
    type = coord::polar;
    break;
  default:
    throw format_error("invalid type");
  }
  return it;
}
```

Dynamic Parsing in {fmt}

```
auto formatter<Point>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  switch (*it++) {
  case 'c':
  case 'r':
    type = coord::cartesian;
    break;
  case 'p':
    type = coord::polar;
    break;
  case '{': {
    // ...
    break;
  }
  default:
    throw format_error("invalid type");
  }
  return it;
}
```

Dynamic Parsing in {fmt}

```
case '{': {
  type = coord::dynamic;
```

```
  break;
}
```

Dynamic Parsing in {fmt}

```
case '{': {
  type = coord::dynamic;
  if (*it == '}') {
    arg_id = ctx.next_arg_id();
    ++it;
  } else {









  }
  break;
}
```

Dynamic Parsing in {fmt}

```
case '{': {
  type = coord::dynamic;
  if (*it == '}') {
    arg_id = ctx.next_arg_id();
    ++it;
  } else {
    auto [p, e] = std::from_chars(&*it, &*ctx.end(), arg_id);
    it = ctx.begin() + (p - &*ctx.begin());




  }
  break;
}
```

Dynamic Parsing in {fmt}

```
case '{': {
  type = coord::dynamic;
  if (*it == '}') {
    arg_id = ctx.next_arg_id();
    ++it;
  } else {
    auto [p, e] = std::from_chars(&*it, &*ctx.end(), arg_id);
    it = ctx.begin() + (p - &*ctx.begin());
    if (e == std::errc{} and it != ctx.end() and *it == '}') {
      ctx.check_arg_id(arg_id);
      ++it;
    } else {
      throw format_error("bad");
    }
  }
  break;
}
```

automatic indexing

manual indexing

Dynamic Formatting in {fmt}

```
template <>
struct formatter<Point> {
  enum class coord { cartesian, polar, dynamic };
  coord type = coord::cartesian;
  size_t arg_id = -1;

  constexpr auto parse(auto& ctx) { /* ... */ }

  auto format(Point p, auto& ctx) const {
    if (type == coord::cartesian) {
      return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
    } else {
      return format_to(ctx.out(), "(r={}, theta={})", p.r(), p.theta());
    }
  }
};
```

Dynamic Formatting in {fmt}

```
auto formatter<Point>::format(Point p, auto& ctx) const {
  if (type == coord::cartesian) {
    return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
  } else {
    return format_to(ctx.out(), "(r={}, theta={})", p.r(), p.theta());
  }
}
```

Dynamic Formatting in {fmt}

```
auto formatter<Point>::format(Point p, auto& ctx) const {
  coord const local_type = [&]{

  }();

  if (local_type == coord::cartesian) {
    return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
  } else {
    return format_to(ctx.out(), "(r={}, theta={})", p.r(), p.theta());
  }
}
```

Dynamic Formatting in {fmt}

```
auto formatter<Point>::format(Point p, auto& ctx) const {
  coord const local_type = [&]{
    if (type != coord::dynamic) {
      return type;
    } else {

    }
  }();

  if (local_type == coord::cartesian) {
    return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
  } else {
    return format_to(ctx.out(), "(r={}, theta={})", p.r(), p.theta());
  }
}
```

Dynamic Formatting in {fmt}

```
auto formatter<Point>::format(Point p, auto& ctx) const {
  coord const local_type = [&]{
    if (type != coord::dynamic) {
      return type;
    } else {
      return visit_format_arg([]<class C>(C const& c){
        if constexpr (same_as<C, char>) { return /* ... */; }
        else { throw format_error("dynamic type must be char"); }
      }, ctx.arg(arg_id));
    }
  }();

  if (local_type == coord::cartesian) {
    return format_to(ctx.out(), "(x={}, y={})", p.x, p.y);
  } else {
    return format_to(ctx.out(), "(r={}, theta={})", p.r(), p.theta());
  }
}
```

Generic Formatting in {fmt}

USING AN UNDERLYING FORMATTER<T>

A formatter for optional<T>

```
template <class T>
struct formatter<optional<T>> {
  constexpr auto parse(auto& ctx) {
    return ctx.begin();
  }

  auto format(optional<T> const& o, auto& ctx) const {
    if (o) {
      return format_to(ctx.out(), "Some({})", *o);
    } else {
      return format_to(ctx.out(), "None");
    }
  }
};
```

A formatter for optional<T>

```
template <class T>
struct formatter<optional<T>> {
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return ctx.begin();
  }

  auto format(optional<T> const& o, auto& ctx) const {
    if (o) {
      return format_to(ctx.out(), "Some({})", *o);
    } else {
      return format_to(ctx.out(), "None");
    }
  }
};
```

A formatter for optional<T>

```
template <class T>
struct formatter<optional<T>> {
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(optional<T> const& o, auto& ctx) const {
    if (o) {
      return format_to(ctx.out(), "Some({})", *o);
    } else {
      return format_to(ctx.out(), "None");
    }
  }
};
```

A formatter for optional<T>

```
template <class T>
struct formatter<optional<T>> {
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(optional<T> const& o, auto& ctx) const {
    if (o) {
      auto out = format_to(ctx.out(), "Some(");
      out = format_to(out, "{}", *o);
      return format_to(out, ")");
    } else {
      return format_to(ctx.out(), "None");
    }
  }
};
```

A formatter for optional<T>

```
template <class T>
struct formatter<optional<T>> {
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(optional<T> const& o, auto& ctx) const {
    if (o) {
      auto out = format_to(ctx.out(), "Some(");
      out = underlying.format(*o, ???);
      return format_to(out, ")");
    } else {
      return format_to(ctx.out(), "None");
    }
  }
};
```

A formatter for optional<T>

```
template <class T>
struct formatter<optional<T>> {
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(optional<T> const& o, auto& ctx) const {
    if (o) {
      auto out = format_to(ctx.out(), "Some(");
      ctx.advance_to(out);
      out = underlying.format(*o, ctx);
      return format_to(out, ")");
    } else {
      return format_to(ctx.out(), "None");
    }
  }
};
```

Formatting Ranges

Various Range Formats

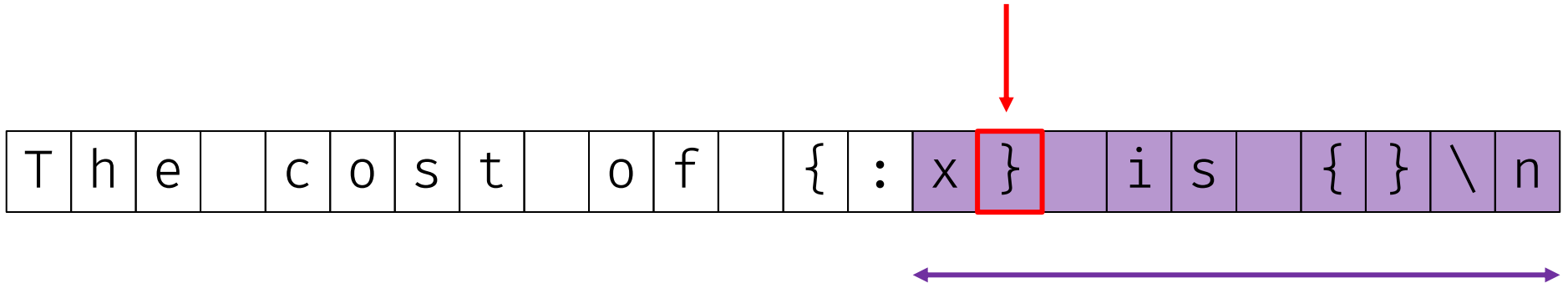
```
[1, 2, 3]
[[1, 2], [3]]
["hello", "world"]
['a', ',', ' ', '\n']
1, 2, 3
```

```
----------[1, 2, 3]
-----[1, 2, 3]-----
[1, 2, 3]----------
{1: 2, 3: 4}
{1, 2, 3}
```

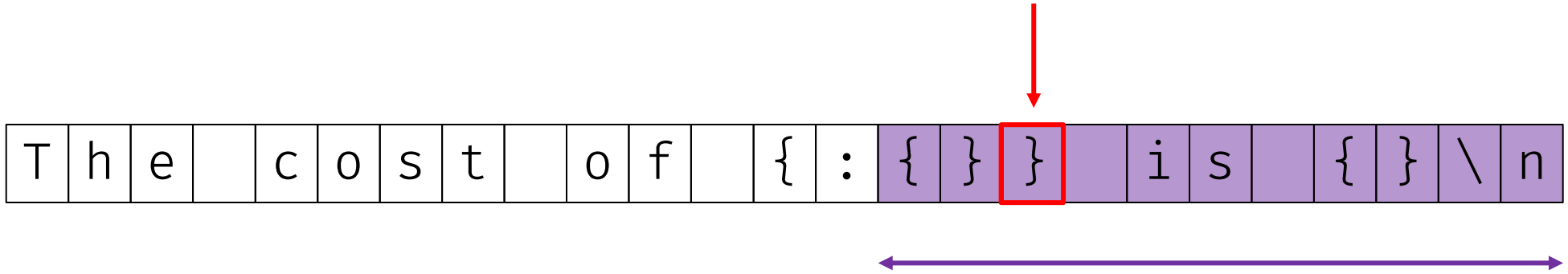
Various Range Formats for `vector<char>`

```
['H', 'e', 'l', 'l', 'o', '!']    48:65:6c:6c:6f:21

[H, e, l, l, o, !]                "Hello!"

[72, 101, 108, 108, 111, 33]      Hello!

[48, 65, 6c, 6c, 6f, 21]

[0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x21]
```

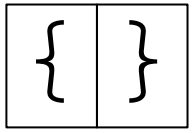
A *format-spec* for Ranges



A *format-spec* for Ranges

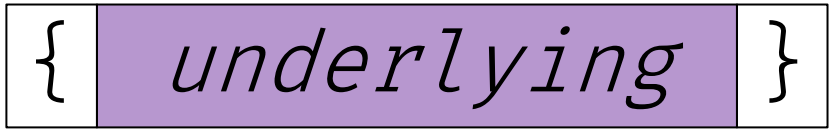


A *format-spec* for Ranges




```
['H', 'e', 'l', 'l', 'o', '!']
```

A *format-spec* for Ranges

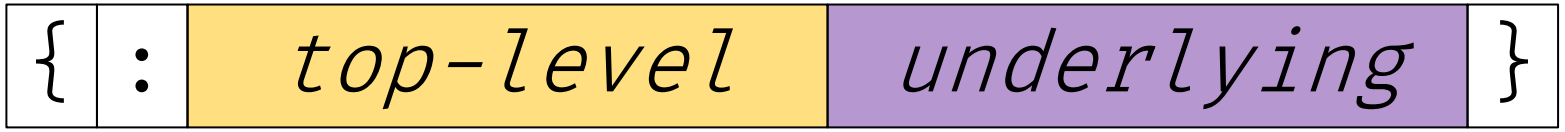



```
['H', 'e', 'l', 'l', 'o', '!']
```

A *format-spec* for Ranges

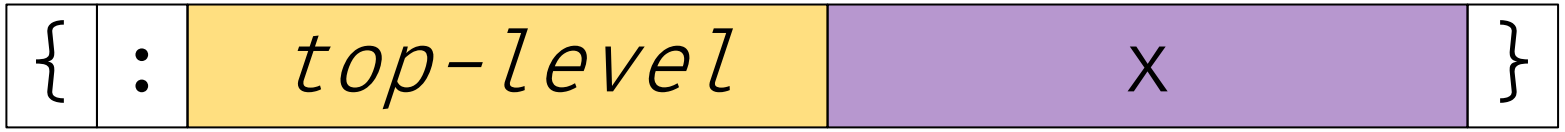

```
['H', 'e', 'l', 'l', 'o', '!']
```

A *format-spec* for Ranges



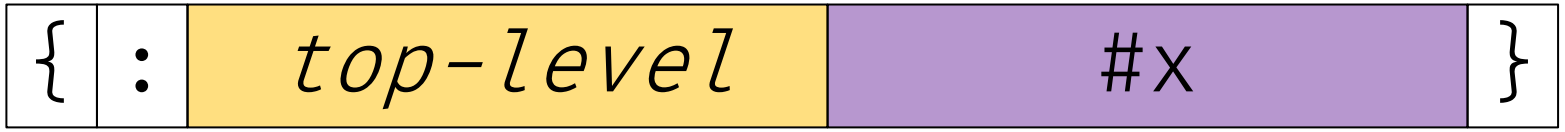

```
['H', 'e', 'l', 'l', 'o', '!']
```

A *format-spec* for Ranges



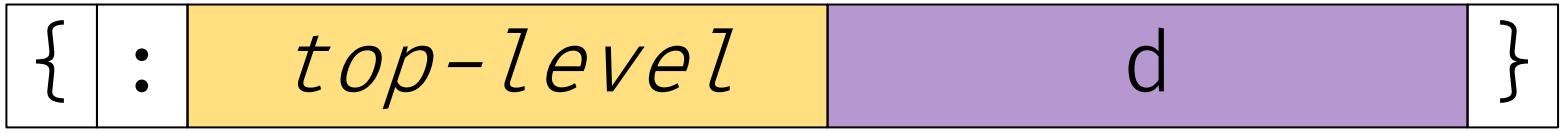

```
[48, 65, 6c, 6c, 6f, 21]
```

A *format-spec* for Ranges



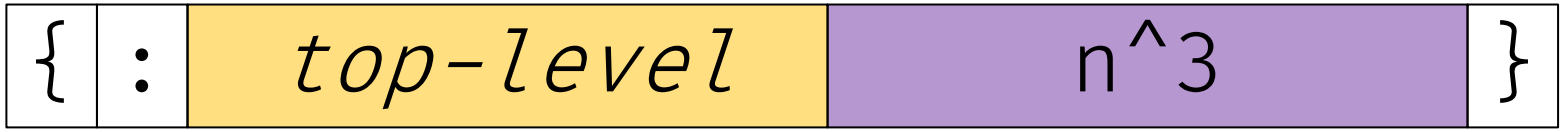

```
[0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x21]
```

A *format-spec* for Ranges




```
[72, 101, 108, 108, 111, 33]
```

A *format-spec* for Ranges

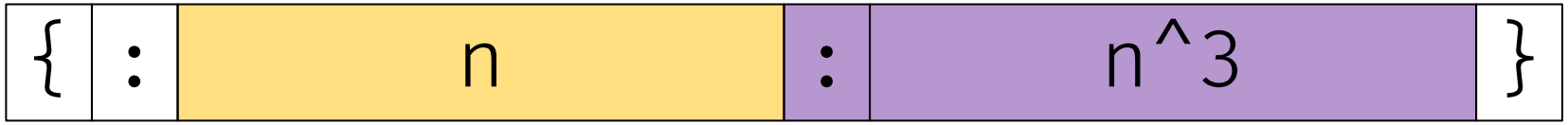



```
[nHn, nen, nln, nln, non, n!n]
```

A *format-spec* for Ranges

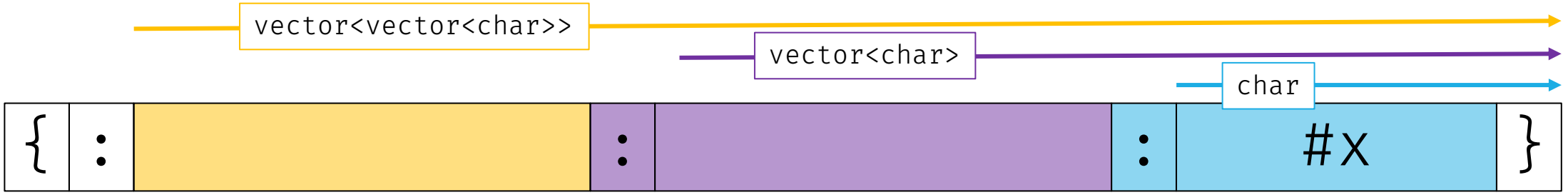

```
nHn, nen, nln, nln, non, n!n
```

A *format-spec* for Ranges



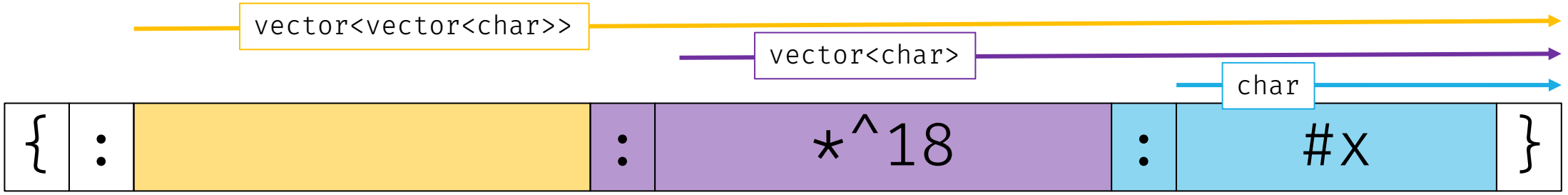

```
nHn, nen, nln, nln, non, n!n
```

A *format-spec* for Ranges



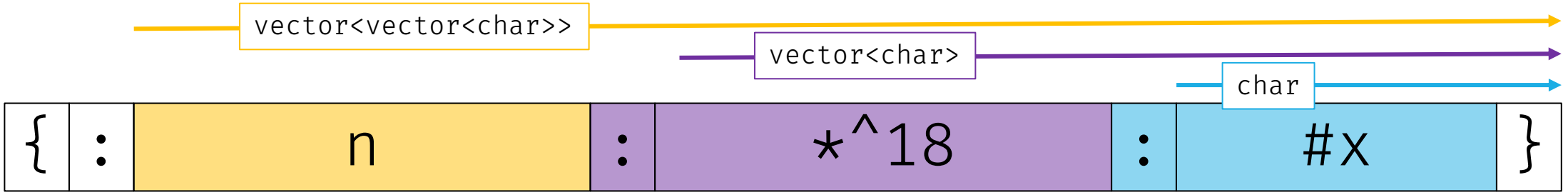

```
[[0x48], [0x65, 0x6c], [0x6c, 0x6f, 0x21]]
```

A *format-spec* for Ranges




```
[******[0x48]******, ***[0x65, 0x6c]***, [0x6c, 0x6f, 0x21]]
```

A *format-spec* for Ranges




```
******[0x48]******, ***[0x65, 0x6c]***, [0x6c, 0x6f, 0x21]
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {

};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }
};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {

  }
};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");

    return format_to(out, "]");
  }
};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    for (auto&& elem : r) {

    }
    return format_to(out, "]");
  }
};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    for (auto&& elem : r) {
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Implementing `formatter` for Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Implementing `formatter` for Ranges

```
int main() {
  vector<int> v = {10, 20, 30};
  print("{}\n", v);
}
```

```
[10, 20, 30]
```

Implementing `formatter` for Ranges

```
int main() {
  vector<int> v = {10, 20, 30};
  print("{}\n", v);
  print("{}\n", vector{v, v});
}
```

```
[10, 20, 30]
[[10, 20, 30], [10, 20, 30]]
```

Implementing `formatter` for Ranges

```
int main() {
  vector<int> v = {10, 20, 30};
  print("{}\n", v);
  print("{}\n", vector{v, v});
  print("{:x} {:#x}\n", v, vector{v, v});
}
```

```
[10, 20, 30]
[[10, 20, 30], [10, 20, 30]]
[a, 14, 1e], [[0xa, 0x14, 0x1e], [0xa, 0x14, 0x1e]]
```

Implementing `formatter` for Ranges

```
int main() {
  vector<int> v = {10, 20, 30};
  print("{}\n", v);
  print("{}\n", vector{v, v});
  print("{:x} {:#x}\n", v, vector{v, v});
  print("{}\n", v | views::transform(_1 * 2));
}
```

```
[10, 20, 30]
[[10, 20, 30], [10, 20, 30]]
[a, 14, 1e], [[0xa, 0x14, 0x1e], [0xa, 0x14, 0x1e]]
[20, 40, 60]
```

Implementing `formatter` for Ranges

```
int main() {
  vector<int> v = {10, 20, 30};
  print("{}\n", v);
  print("{}\n", vector{v, v});
  print("{:x} {:#x}\n", v, vector{v, v});
  print("{}\n", v | views::transform(_1 * 2));
  print("{}\n", v | views::filter(_1 > 15));
}
```

```
[10, 20, 30]
[[10, 20, 30], [10, 20, 30]]
[a, 14, 1e], [[0xa, 0x14, 0x1e], [0xa, 0x14, 0x1e]]
[20, 40, 60]
```

Supporting non-const-iterable Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Supporting non-const-iterable Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : views::all(r)) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Supporting non-const-iterable Ranges

```
namespace std {
  // [format.functions], formatting functions
  template<class... Args>
    string format(string_view fmt, const Args&... args);
  template<class... Args>
    wstring format(wstring_view fmt, const Args&... args);
  template<class... Args>
    string format(const locale& loc, string_view fmt, const Args&... args);
  template<class... Args>
    wstring format(const locale& loc, wstring_view fmt, const Args&... args);

  template<class Out, class... Args>
    Out format_to(Out out, string_view fmt, const Args&... args);
  template<class Out, class... Args>
    Out format_to(Out out, wstring_view fmt, const Args&... args);
  template<class Out, class... Args>
    Out format_to(Out out, const locale& loc, string_view fmt, const Args&... args);
  template<class Out, class... Args>
    Out format_to(Out out, const locale& loc, wstring_view fmt, const Args&... args);
}
```

P2418R2
Add support for `std::generator`-like types to `std::format`
Published Proposal, 2021-09-24

Author:
Victor Zverovich

Supporting non-const-iterable Ranges

```
namespace std {
  // [format.functions], formatting functions
  template<class... Args>
    string format(string_view fmt, Args&&... args);
  template<class... Args>
    wstring format(wstring_view fmt, Args&&... args);
  template<class... Args>
    string format(const locale& loc, string_view fmt, Args&&... args);
  template<class... Args>
    wstring format(const locale& loc, wstring_view fmt, Args&&... args);

  template<class Out, class... Args>
    Out format_to(Out out, string_view fmt, Args&&... args);
  template<class Out, class... Args>
    Out format_to(Out out, wstring_view fmt, Args&&... args);
  template<class Out, class... Args>
    Out format_to(Out out, const locale& loc, string_view fmt, Args&&... args);
  template<class Out, class... Args>
    Out format_to(Out out, const locale& loc, wstring_view fmt, Args&&... args);
}
```

P2418R2
Add support for `std::generator`-like types to `std::format`
Published Proposal, 2021-09-24

Author:
Victor Zverovich

Supporting non-const-iterable Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Supporting non-const-iterable Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Supporting non-const-iterable Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(R const& r, auto& ctx) const { return format_impl(r, ctx); }
  auto format(R      & r, auto& ctx) const { return format_impl(r, ctx); }

  auto format_impl(auto& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Supporting non-const-iterable Ranges

```
template <ranges::input_range R>
struct formatter<R> {
  using T = remove_cvref_t<ranges::range_reference_t<R>>;
  formatter<T> underlying;

  constexpr auto parse(auto& ctx) {
    return underlying.parse(ctx);
  }

  auto format(fmt-maybe-const<R>& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

```
template <class R>
using fmt-maybe-const = conditional_t<
    const-formattable-range<R>, R const, R>;
```

Supporting non-const-iterable Ranges

```
int main() {
  vector<int> v = {10, 20, 30};
  print("{}\n", v);
  print("{}\n", vector{v, v});
  print("{:x} {:#x}\n", v, vector{v, v});
  print("{}\n", v | views::transform(_1 * 2));
  print("{}\n", v | views::filter(_1 > 15)); ☑
}
```

```
[10, 20, 30]
[[10, 20, 30], [10, 20, 30]]
[a, 14, 1e], [[0xa, 0x14, 0x1e], [0xa, 0x14, 0x1e]]
[20, 40, 60]
[20, 30]
```

Adding top-level specifiers

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Adding top-level specifiers: n

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = format_to(ctx.out(), "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    return format_to(out, "]");
  }
};
```

Adding top-level specifiers: n

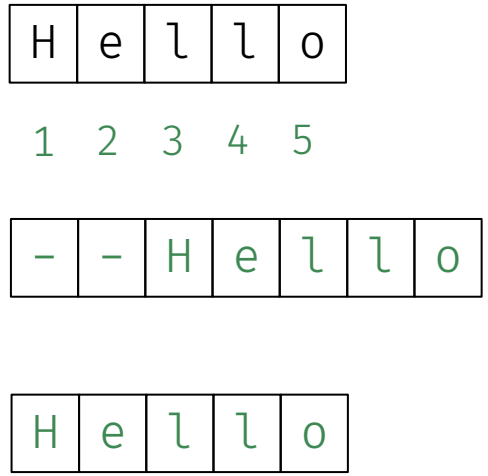
```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```

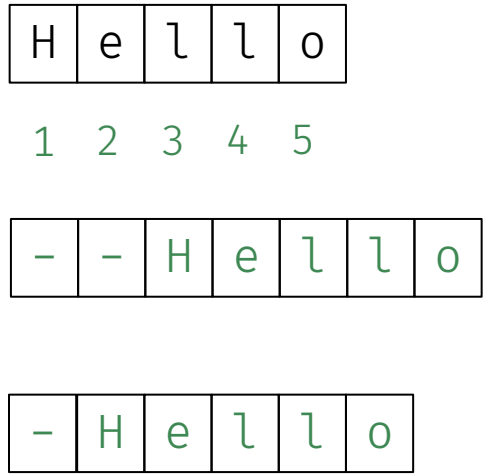
Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier
  format_specs specs = {};  // fill, align, width

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```



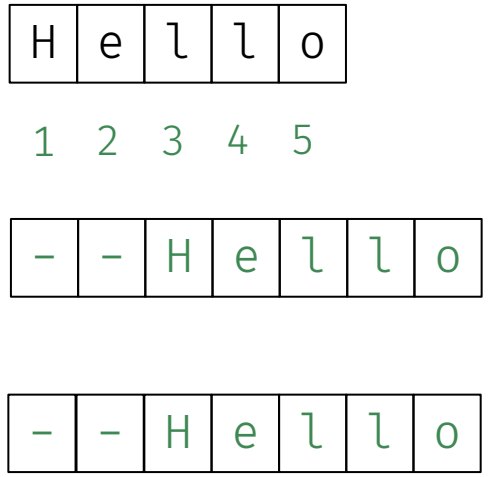
Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier
  format_specs specs = {};  // fill, align, width

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```



Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier
  format_specs specs = {};  // fill, align, width

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```



Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier
  format_specs specs = {};  // fill, align, width

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```

`ctx.out()` may be write-once
(e.g. `back_inserter<string>`)

No idea how many characters to write

Can't iterate the range twice

But we must format into `ctx`

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
struct formatter<R> {
  formatter</* ... */> underlying;
  bool no_brackets = false; // the 'n' specifier
  format_specs specs = {};  // fill, align, width

  constexpr auto parse(auto& ctx);

  auto format(fmt_maybe_const<R>& r, auto& ctx) const {
    auto out = ctx.out();
    if (not no_brackets) out = format_to(out, "[");
    bool first = true;
    for (auto&& elem : r) {
      if (not first) out = format_to(out, ", ");
      first = false;
      ctx.advance_to(out);
      out = underlying.format(elem, ctx);
    }
    if (not no_brackets) out = format_to(out, "]");
    return out;
  }
};
```

`ctx.out()` may be write-once
(e.g. `back_inserter<string>`)

No idea how many characters to write

Can't iterate the range twice

But we must format into *some* `context`

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  auto out = ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    ctx.advance_to(out);
    out = underlying.format(elem, ctx);
  }
  if (not no_brackets) out = format_to(out, "]");
  return out;
}
```

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf)};

  auto out = ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    ctx.advance_to(out);
    out = underlying.format(elem, ctx);
  }
  if (not no_brackets) out = format_to(out, "]");
  return out;
}
```

new, local format context

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf)};

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");
  return out;
}
```

new, local format context

write into local context

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf)};

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

new, local format context

write into local context

transfer to main context

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf)};

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

```
namespace std {
  template<class Out>
  class format_context {
    basic_format_args<format_context> args_;      // exposition only
    Out out_;                                     // exposition only

  public:
    using iterator = Out;
    using char_type = char;
    template<class T> using formatter_type = formatter<T>;

    basic_format_arg<format_context> arg(size_t id) const noexcept;

    iterator out();
    void advance_to(iterator it);
  };
}
```

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf)};

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

```
namespace std {
  template<class Out>
  class format_context {
    basic_format_args<format_context> args_;      // exposition only
    Out out_;                                     // exposition only

  public:
    using iterator = Out;
    using char_type = char;
    explicit format_context(Out);
    template<class T> using formatter_type = formatter<T>;

    basic_format_arg<format_context> arg(size_t id) const noexcept;

    iterator out();
    void advance_to(iterator it);
  };
}
```

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf), ctx.args()};

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

```
namespace std {
  template<class Out>
  class format_context {
    basic_format_args<format_context> args_;      // exposition only
    Out out_;                                     // exposition only

  public:
    using iterator = Out;
    using char_type = char;
    explicit format_context(Out, basic_format_args<format_context>);
    template<class T> using formatter_type = formatter<T>;

    basic_format_arg<format_context> arg(size_t id) const noexcept;

    iterator out();
    void advance_to(iterator it);
  };
}
```

Exploring `basic_format_args<Context>`

```
template <class Context>
using basic_format_args ≈ span<basic_format_arg<Context> const>;
```

Exploring basic_format_arg<Context>

```
template <class Context>
using basic_format_args ≈ span<basic_format_arg<Context> const>;

template <class Context>
class basic_format_arg {
public:
  class handle;

private:
  variant<monostate, bool, char,
          int, unsigned int, long long int, unsigned long long int,
          float, double, long double,
          const char*, string_view,
          const void*, handle> value;   // exposition only
};
```

Exploring basic_format_arg<Context>

```
template <class Context>
using basic_format_args ≈ span<basic_format_arg<Context> const>;

template <class Context>
class handle;

template <class Context>
using basic_format_arg ≈
  variant<monostate, bool, char,
          int, unsigned int, long long int, unsigned long long int,
          float, double, long double,
          const char*, string_view,
          const void*, handle<Context>>;
```

Exploring basic_format_arg<Context>

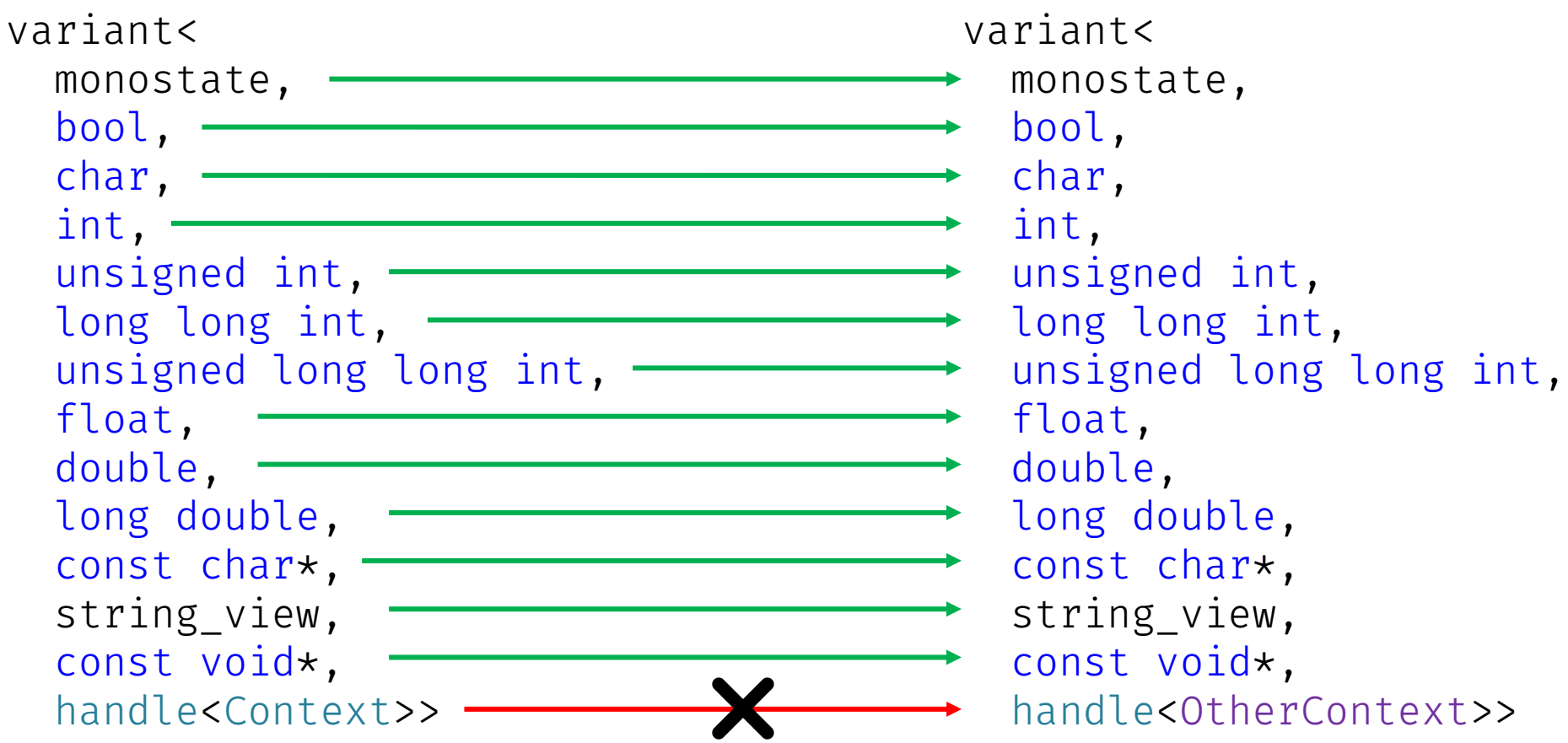
```
template <class Context>
using basic_format_args ≈ span<basic_format_arg<Context> const>;

template <class Context>
class handle {
  void const* ptr_;
  void (*format_)(format_parse_context&, Context&, void const*);
};

template <class Context>
using basic_format_arg ≈
  variant<monostate, bool, char,
          int, unsigned int, long long int, unsigned long long int,
          float, double, long double,
          const char*, string_view,
          const void*, handle<Context>>;
```

Exploring `basic_format_arg<Context>`

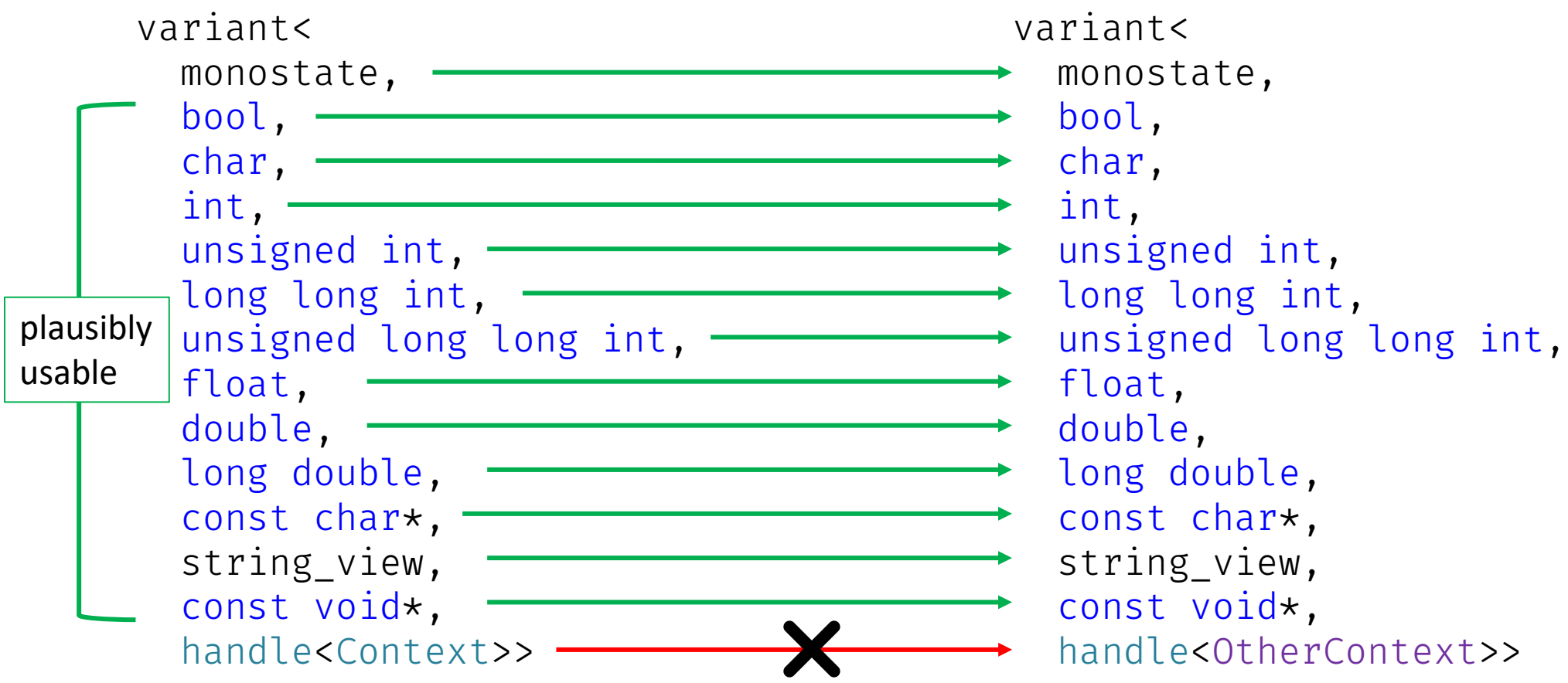
```
variant<
  monostate,
  bool,
  char,
  int,
  unsigned int,
  long long int,
  unsigned long long int,
  float,
  double,
  long double,
  const char*,
  string_view,
  const void*,
  handle<Context>>
```

```
variant<
  monostate,
  bool,
  char,
  int,
  unsigned int,
  long long int,
  unsigned long long int,
  float,
  double,
  long double,
  const char*,
  string_view,
  const void*,
  handle<OtherContext>>
```

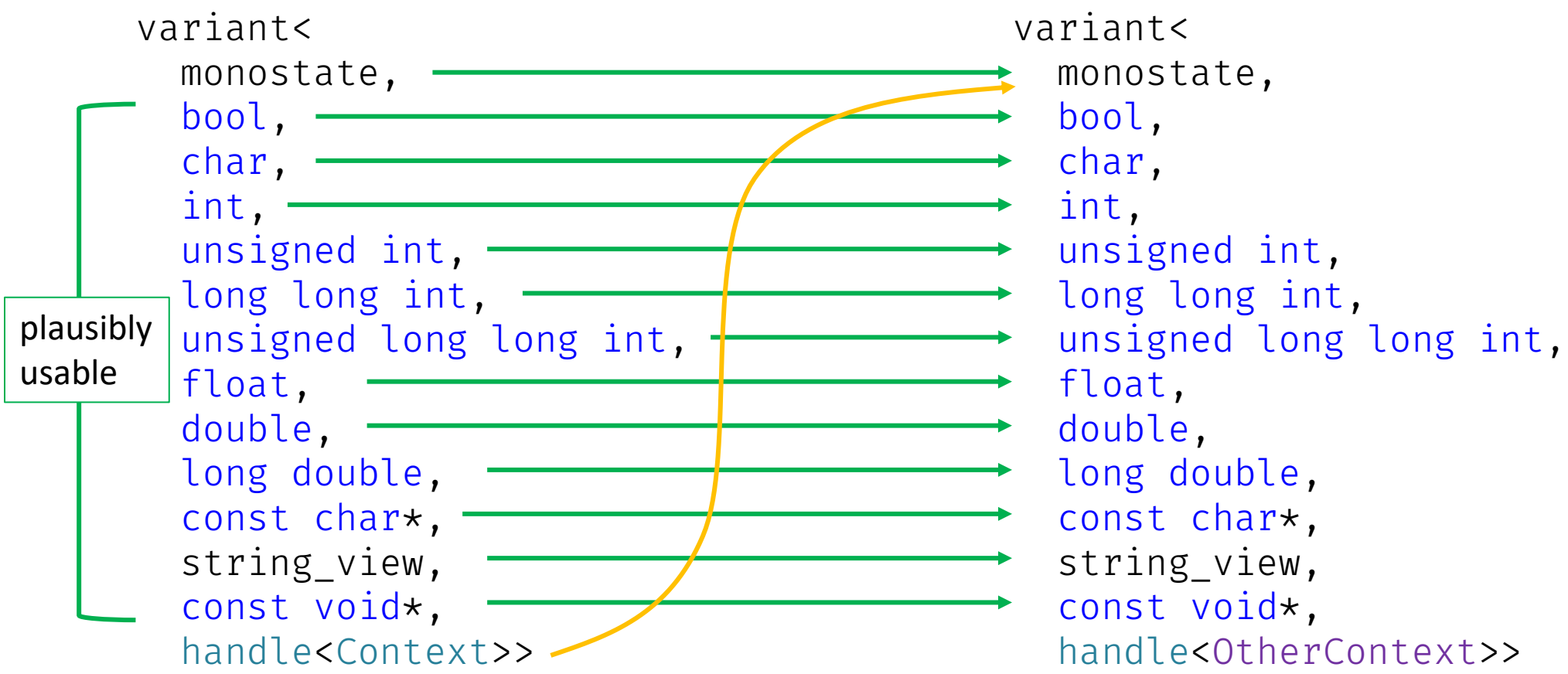
Exploring `basic_format_arg<Context>`



Exploring `basic_format_arg<Context>`



Exploring `basic_format_arg<Context>`



Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  format_context new_ctx{back_inserter(buf), ctx.args()};

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  retargeted_format_context rctx{ctx, back_inserter(buf)};
  auto& new_ctx = rctx.context();

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

does the `basic_format_arg` conversions

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  vector<char> buf;
  retargeted_format_context rctx{ctx, back_inserter(buf)};
  auto& new_ctx = rctx.context();

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

does the `basic_format_arg` conversions (only if necessary)

Adding top-level specifiers: fill/align/width

```
template <ranges::input_range R>
auto formatter<R>::format(fmt_maybe_const<R>& r, auto& ctx) const {
  memory_buffer buf;
  retargeted_format_context rctx{ctx, appender(buf)};
  auto& new_ctx = rctx.context();

  auto out = new_ctx.out();
  if (not no_brackets) out = format_to(out, "[");
  bool first = true;
  for (auto&& elem : r) {
    if (not first) out = format_to(out, ", ");
    first = false;
    new_ctx.advance_to(out);
    out = underlying.format(elem, new_ctx);
  }
  if (not no_brackets) out = format_to(out, "]");


  return write_padded_aligned(ctx.out(), specs, buf);
}
```

does the `basic_format_arg` conversions (may not be necessary here)

https://godbolt.org/z/cs1d9YEv8

Adding top-level specifiers: delimiter

```
int main() {
  vector<uint8_t> mac = {0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
  print("{}\n", mac);
}
```

```
[170, 187, 204, 221, 238, 255]
```

Adding top-level specifiers: delimiter

```
int main() {
  vector<uint8_t> mac = {0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
  print("{}\n", mac);
  print("{::02x}\n", mac);
}
```

```
[170, 187, 204, 221, 238, 255]
[aa, bb, cc, dd, ee, ff]
```

Adding top-level specifiers: delimiter

```
int main() {
  vector<uint8_t> mac = {0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
  print("{}\n", mac);
  print("{::02x}\n", mac);
  print("{:n:02x}\n", mac);
}
```

```
[170, 187, 204, 221, 238, 255]
[aa, bb, cc, dd, ee, ff]
aa, bb, cc, dd, ee, ff
```

Adding top-level specifiers: delimiter

```
int main() {
  vector<uint8_t> mac = {0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff};
  print("{}\n", mac);
  print("{::02x}\n", mac);
  print("{:n:02x}\n", mac);
  print("{:02x}\n", join(mac, ":"));
}
```

```
[170, 187, 204, 221, 238, 255]
[aa, bb, cc, dd, ee, ff]
aa, bb, cc, dd, ee, ff
aa:bb:cc:dd:ee:ff
```

Adding top-level specifiers: delimiter

```
print("{:02x}\n", join(mac, ":"));
```

```
aa:bb:cc:dd:ee:ff
```

Adding top-level specifiers: delimiter

```
print("{:02x}\n", join(mac, ":"));
print("{::02x}\n",
    some_macs | views::transform([](auto&& m){
        return join(m, ":");
    }));
```

```
aa:bb:cc:dd:ee:ff
[aa:bb:cc:dd:ee:ff, 00:00:5e:00:53:af, 00:00:0a:bb:28:fc]
```

Adding top-level specifiers: delimiter

```
print("{:02x}\n", join(mac, ":"));
print("{::02x}\n",
    some_macs | views::transform([](auto&& m){
        return join(m, ":");
    }));
print("{:-^23}\n", format("{:02x}", join(mac, ":")));
```

```
aa:bb:cc:dd:ee:ff
[aa:bb:cc:dd:ee:ff, 00:00:5e:00:53:af, 00:00:0a:bb:28:fc]
---aa:bb:cc:dd:ee:ff---
```

Adding top-level specifiers: delimiter

```
print("{:nd{}:02x}\n", mac, ":");
print("{::nd{}:02x}\n", some_macs, ":");
print("{:-^23nd{}:02x}\n", mac, ":");
```

```
aa:bb:cc:dd:ee:ff
[aa:bb:cc:dd:ee:ff, 00:00:5e:00:53:af, 00:00:0a:bb:28:fc]
---aa:bb:cc:dd:ee:ff---
```

Adding top-level specifiers: delimiter

```
print("{:nd[:]:02x}\n", mac);
print("{::nd[:]:02x}\n", some_macs);
print("{:-^23nd[:]:02x}\n", mac);
```

```
print("{:02x}\n", join(mac, ":"));
print("{::02x}\n",
    some_macs | views::transform([](auto&& m){
        return join(m, ":");
    }));
print("{:-^23}\n", format("{:02x}", join(mac, ":")));
```

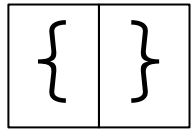
```
aa:bb:cc:dd:ee:ff
[aa:bb:cc:dd:ee:ff, 00:00:5e:00:53:af, 00:00:0a:bb:28:fc]
---aa:bb:cc:dd:ee:ff---
```

https://godbolt.org/z/cs1d9YEv8

Formatting Tuples

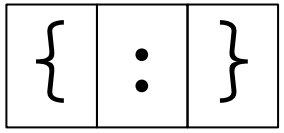
THE FINAL BOSS

A *format-spec* for pair<int, int>




```
(10, 1729)
```

A *format-spec* for pair<int, int>

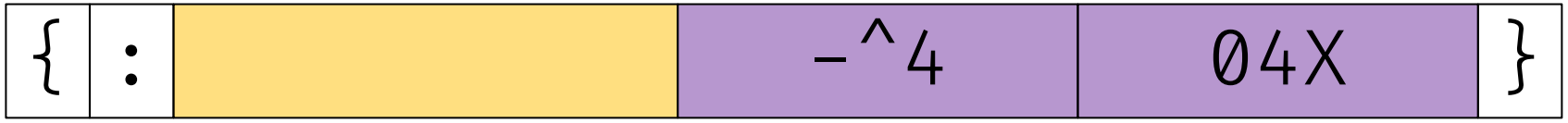



```
(10, 1729)
```

A *format-spec* for pair<int, int>


```
(10, 1729)
```

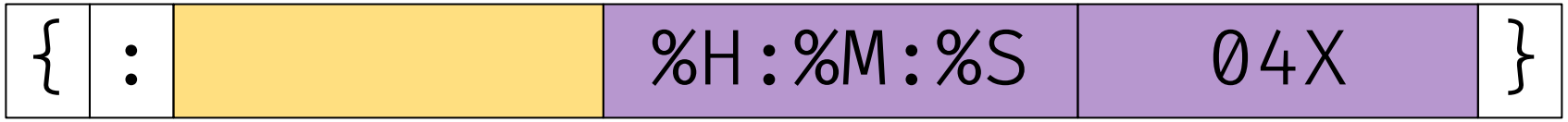
A *format-spec* for pair<int, int>



(-10-, 06C1)

How do we find this boundary?

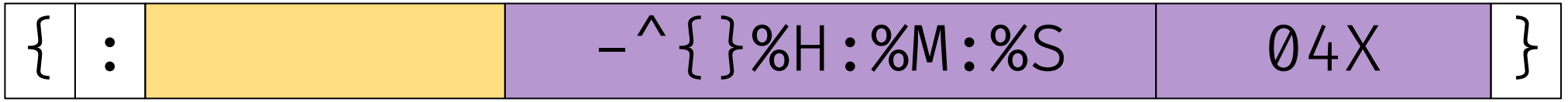
A *format-spec* for pair<system_clock::time_point, int>



(20:33:37, 06C1)

How do we find this boundary?

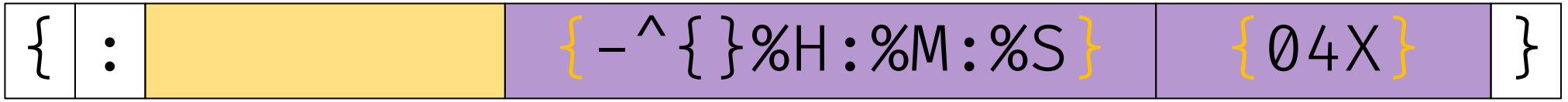
A *format-spec* for pair<system_clock::time_point, int>



(--20:33:37--, 06C1)

How do we find this boundary?

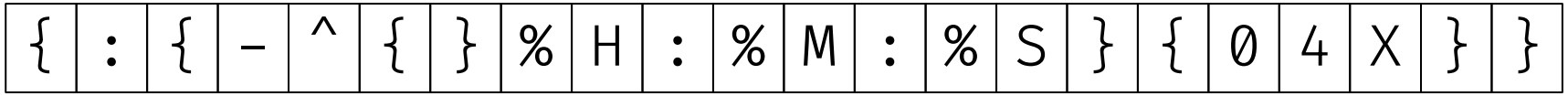
A *format-spec* for pair<system_clock::time_point, int>



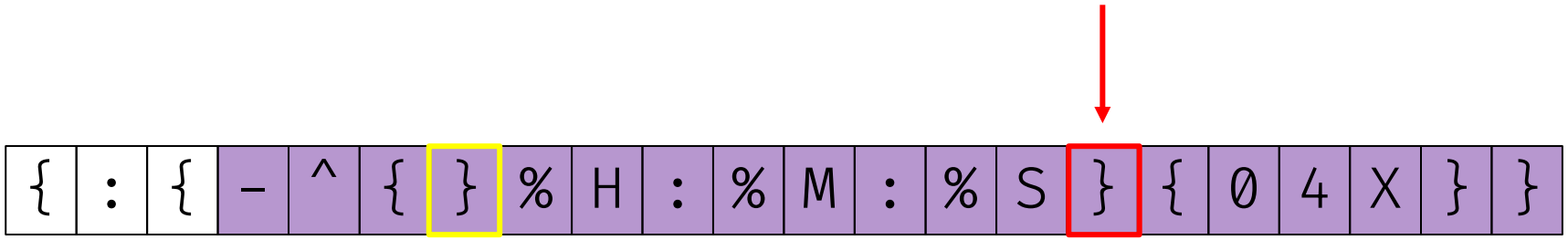
(--20:33:37--, 06C1)

How do we find this boundary?

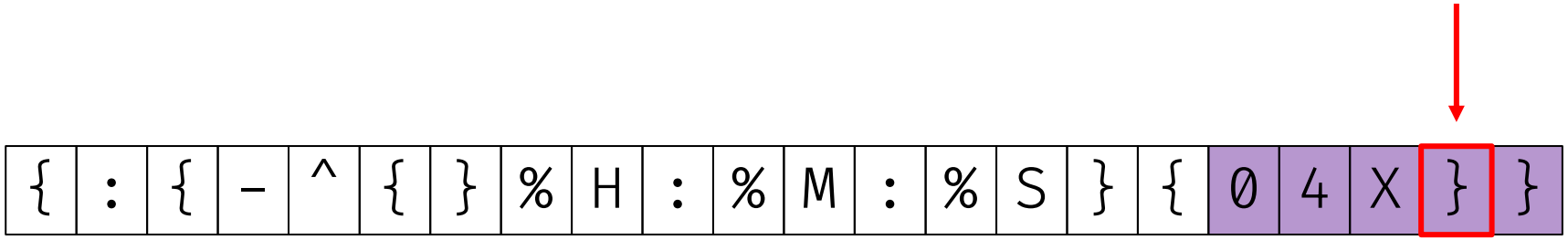
A *format-spec* for pair<system_clock::time_point, int>



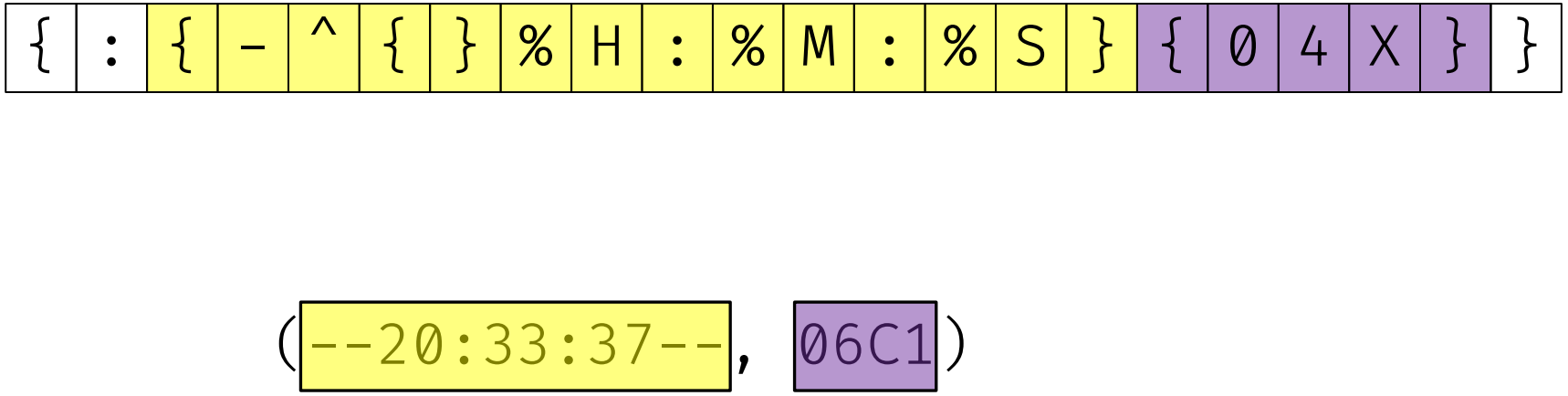
A *format-spec* for pair<system_clock::time_point, int>



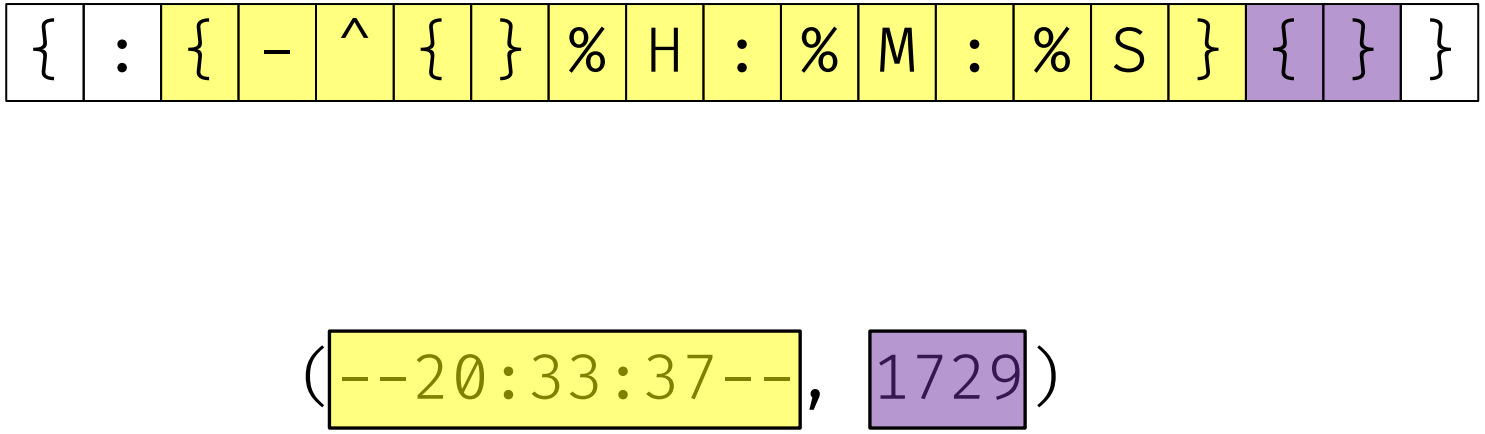
A *format-spec* for pair<system_clock::time_point, int>



A *format-spec* for pair<system_clock::time_point, int>



A *format-spec* for pair<system_clock::time_point, int>



Parsing *format-spec* for Tuples

```
template <formattable... Ts>
struct formatter<tuple<Ts...>> {
  std::tuple<formatter<remove_cvref_t<Ts>>...> underlying;

  constexpr auto parse(auto& ctx) {

  }

  auto format(tuple<Ts...> const&, auto& ctx) const;
};
```

Parsing *format-spec* for Tuples

```
template <formattable... Ts>
struct formatter<tuple<Ts...>> {
  std::tuple<formatter<remove_cvref_t<Ts>>...> underlying;

  constexpr auto parse(auto& ctx) {
    auto it = ctx.begin();
    if (it == ctx.end() or *it == '}') {
      return it;
    }

  }

  auto format(tuple<Ts...> const&, auto& ctx) const;
};
```

Parsing *format-spec* for Tuples

```
template <formattable... Ts>
struct formatter<tuple<Ts...>> {
  std::tuple<formatter<remove_cvref_t<Ts>>...> underlying;

  constexpr auto parse(auto& ctx) {
    auto it = ctx.begin();
    if (it == ctx.end() or *it == '}') {
      return it;
    }

    tuple_for_each(underlying, [&](auto& f){
      // ...
    });
    return it;
  }

  auto format(tuple<Ts...> const&, auto& ctx) const;
};
```

Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  tuple_for_each(underlying, [&](auto& f){
    // ...
  });
  return it;
}
```

Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  tuple_for_each(underlying, [&](auto& f){
    // opening brace
    if (it == ctx.end() or *it != '{') throw format_error("bad");
  });
  return it;
}
```

Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  tuple_for_each(underlying, [&](auto& f){
    // opening brace
    if (it == ctx.end() or *it != '{') throw format_error("bad");
    // format-spec
    ctx.advance_to(it + 1);
    it = f.parse(ctx);
  });
  return it;
}
```

Parsing *format-spec* for Tuples

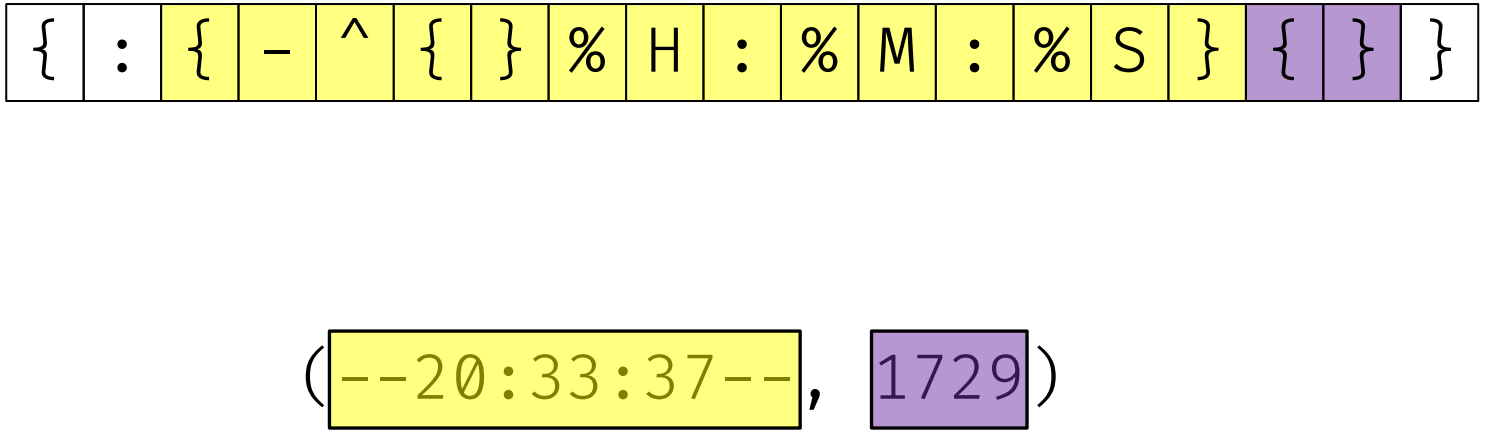
```
template <formattable ... T>
constexpr auto formatter<tuple<T ... >>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  tuple_for_each(underlying, [&](auto& f){
    // opening brace
    if (it == ctx.end() or *it ≠ '{') throw format_error("bad");
    // format-spec
    ctx.advance_to(it + 1);
    it = f.parse(ctx);
    // closing brace
    if (it == ctx.end() or *it ≠ '}') throw format_error("bad");
    ++it;
  });
  return it;
}
```

Formatting *format-spec* for Tuples

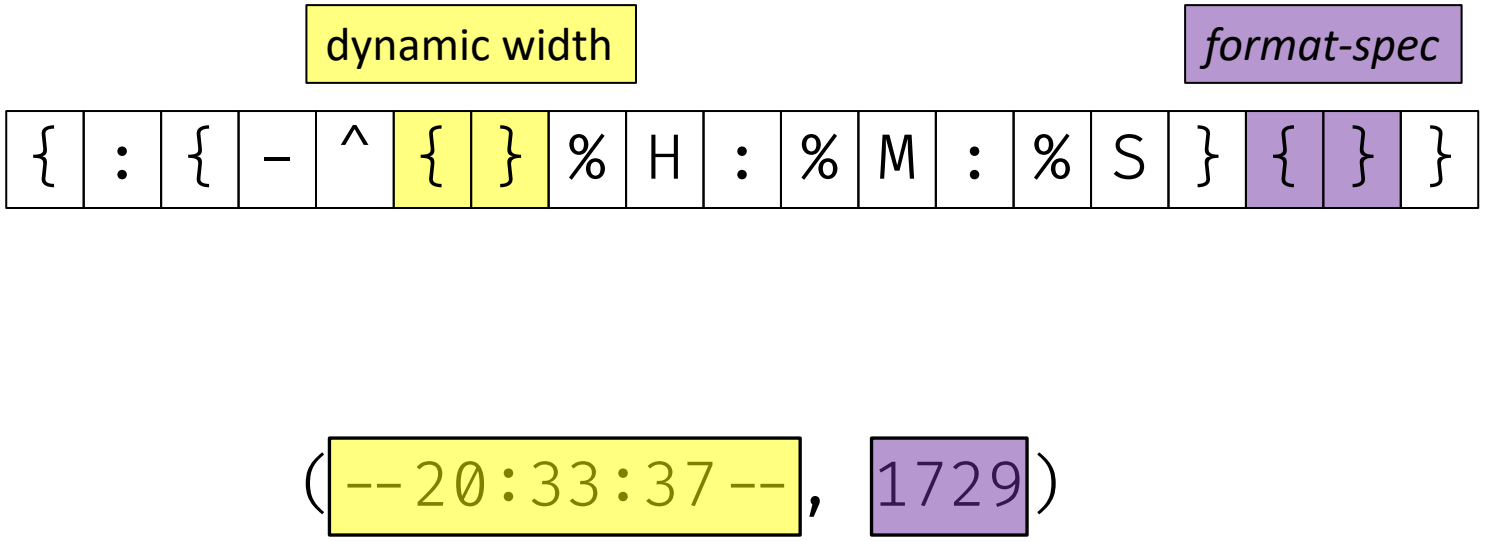
```
template <formattable... T>
auto formatter<tuple<T...>>::format(std::tuple<Ts...> const& t, auto& ctx) const {
  auto out = fmt::format_to(ctx.out(), "(");
  tuple_enumerate(underlying, [&](auto I, auto& f){
    if (I > 0) {
      out = fmt::format_to(out, ", ");
    }
    ctx.advance_to(out);
    out = f.format(std::get<I>(t), ctx);
  });
  return fmt::format_to(out, ")");
}
```

https://godbolt.org/z/vPfE7er3M

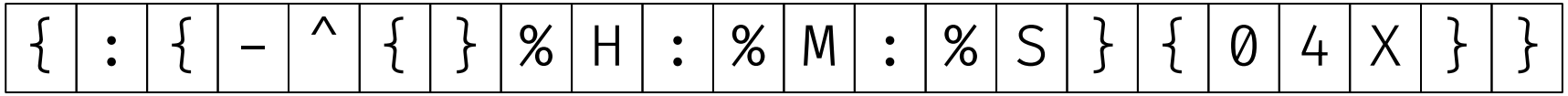
A *format-spec* for pair<system_clock::time_point, int>



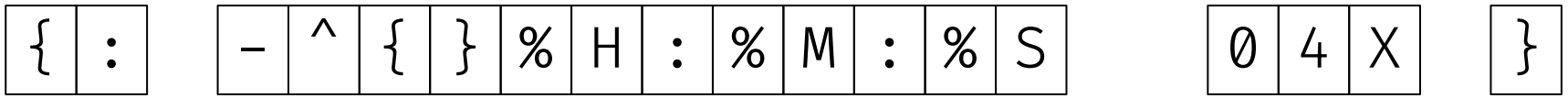
A *format-spec* for pair<system_clock::time_point, int>



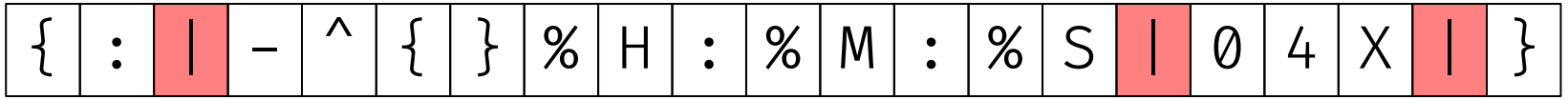
A *format-spec* for pair<system_clock::time_point, int>



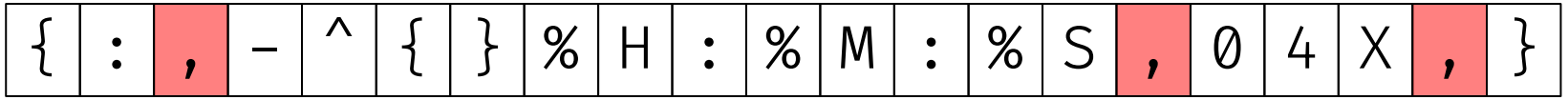
A *format-spec* for pair<system_clock::time_point, int>



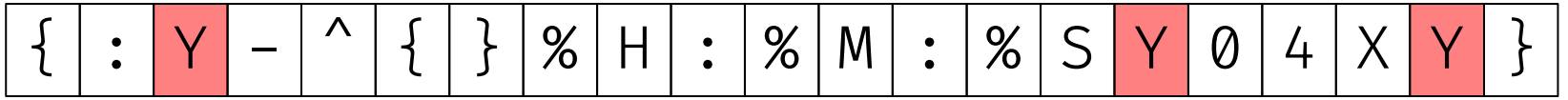
A *format-spec* for pair<system_clock::time_point, int>



A *format-spec* for pair<system_clock::time_point, int>



A *format-spec* for pair<system_clock::time_point, int>



formatter<T>::parse(ctx) is looking for either:

- '}', or
- ctx.end()

Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  tuple_for_each(underlying, [&](auto& f){
    // opening brace
    if (it == ctx.end() or *it != '{') throw format_error("bad");
    // format-spec
    ctx.advance_to(it + 1);
    it = f.parse(ctx);
    // closing brace
    if (it == ctx.end() or *it != '}') throw format_error("bad");
    ++it;
  });
  return it;
}
```

Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // determine delimiter
  char const delim = *it++;
  ctx.advance_to(it);
  tuple_for_each(underlying, [&](auto& f){
    // ...
  });
  return ctx.begin();
}
```

Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // determine delimiter
  char const delim = *it++;
  ctx.advance_to(it);
  tuple_for_each(underlying, [&](auto& f){
    // find the next delim
    auto next_delim = ranges::find(ctx, delim);
    if (next_delim == ctx.end()) throw format_error("bad");

    // ...
  });
  return ctx.begin();
}
```

Parsing *format-spec* for Tuples

```
template <formattable ... T>
constexpr auto formatter<tuple<T ... >>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // determine delimiter
  char const delim = *it++;
  ctx.advance_to(it);
  tuple_for_each(underlying, [&](auto& f){
    // find the next delim
    auto next_delim = ranges::find(ctx, delim);
    if (next_delim == ctx.end()) throw format_error("bad");

    // parse up to the next delim
    auto const real_end = ctx.end();
    ctx.set_end(next_delim);
    if (f.parse(ctx) != next_delim) throw format_error("bad");

    // ...
  });
  return ctx.begin();
}
```

Parsing *format-spec* for Tuples

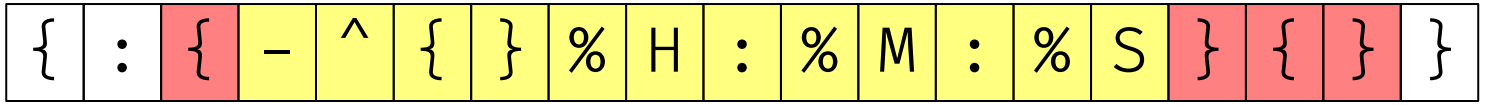
```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // determine delimiter
  char const delim = *it++;
  ctx.advance_to(it);
  tuple_for_each(underlying, [&](auto& f){
    // find the next delim
    auto next_delim = ranges::find(ctx, delim);
    if (next_delim == ctx.end()) throw format_error("bad");

    // parse up to the next delim
    auto const real_end = ctx.end();
    ctx.set_end(next_delim);
    if (f.parse(ctx) != next_delim) throw format_error("bad");

    // onto the next one
    ctx.advance_to(next_delim + 1);
    ctx.set_end(real_end);
  });
  return ctx.begin();
}
```

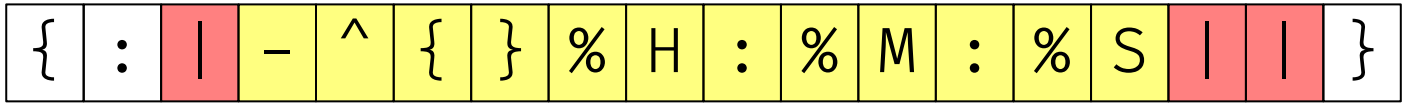
Parsing *format-spec* for Tuples

```
template <formattable... T>
constexpr auto formatter<tuple<T...>>::parse(auto& ctx) {
  auto it = ctx.begin();
  if (it == ctx.end() or *it == '}') { return it; }

  // determine delimiter
  char const delim = *it++;
  ctx.advance_to(it);
  tuple_for_each(underlying, [&](auto& f){
    // find the next delim
    auto next_delim = ranges::find(ctx, delim);
    if (next_delim == ctx.end()) throw format_error("bad");

    // parse up to the next delim
    end_sentry _(ctx, next_delim);
    if (f.parse(ctx) != next_delim) throw format_error("bad");

    // onto the next one
    ctx.advance_to(next_delim + 1);
  });
  return ctx.begin();
}
```

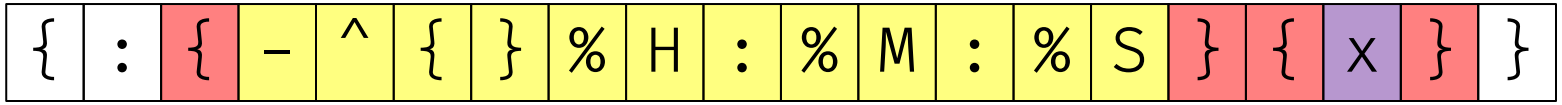
https://godbolt.org/z/PadrMch4x

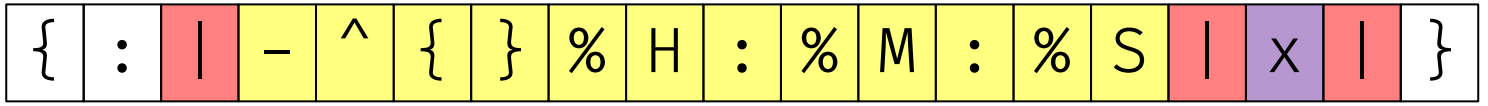
How to do *format-spec* for Tuples?





How to do *format-spec* for Tuples?





Looking to C++23

WHAT'S IN STORE

P2286: Formatting Ranges

R0: January, 2021 (8 pages)

§ 3 Proposal

The standard library should add specializations of `formatter` for:

— any type that satisifies `range` whose `value_type` and `reference` are formattable,

— `pair<T, U>` if `T` and `U` are formattable,

— `tuple<Ts...>` if all of `Ts...` are formattable,

— `vector<bool>::reference` (which does as `bool` does).

The choice of formatting is implementation defined though implementors are encouraged to format ranges and tuples differently).

The standard library should also add a utility `std::format_join` (or any other suitable name, knowing that `std::views::join` already exists), following in the footsteps of `fmt::join`, which allows the user to provide more customization in how ranges and tuples get formatted.

For types like `std::generator<T>` (which are move-only, non-const-iterable ranges), users will have to use `std::format_join` facility.

P2286: Formatting Ranges

R0: January, 2021 (8 pages)

R8: May, 2022 (42 pages)

- Adopted for C++23 
- Formatting for ranges and tuples
- Utility for more convenient range formatting (`range_formatter`)
- Range specifiers for fill/align/width, no brackets, string, map, underlying
- Tuple specifiers for fill/align/width, no brackets, map
- String/char escaping

P2286: Formatting Ranges

R0: January, 2021 (8 pages)

R8: May, 2022 (42 pages)

Future work

- Utility for fill/align/width for user types (`retargeted_format_context?`)
- Delimiter specifier for ranges
- Element-wise specifiers for tuples (`end_sentry?`)

P2286: Formatting Ranges

R0: January, 2021 (8 pages)

R8: May, 2022 (42 pages)

Future work

This paper (and work) would not exist without:

- Victor Zverovich
- Tim Song
- Peter Dimov