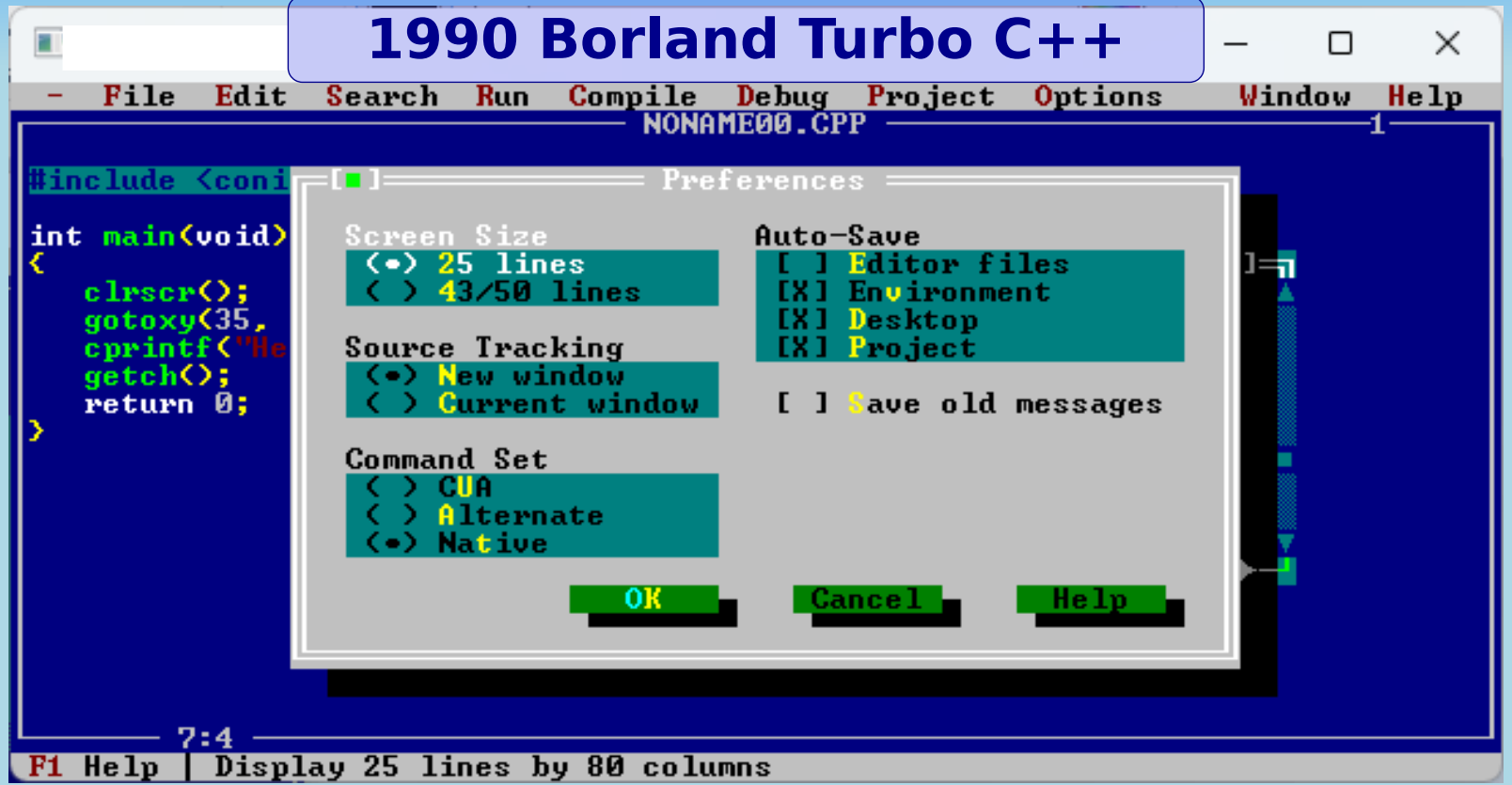
Rud Merriam, Retired
Embedded Systems & Robotics
The Woodlands, TX USA

Meandering Through C++
Creating *ranges::to*

Who **Doesn't** Recognize This??



Background

- Experience with many langagues
  - FORTRAN IV (1968!), PL/M, assembly, C, Turbo Pascal, Forth
- Taught *Introduction to C++* - U of Houston / Clear Lake
- C++ in NASA / NIST Robotic Competitions after retired
- Writing
  - Magazines in 90s - Embedded Systems, Software Development, Programmers Journal, PC Magazine
  - Hackaday.com: C++ for embedded systems (Arduino, Raspberry Pi)
  - Medium.com (https://medium.com/@rudmerriam)

What is ranges::to?

From P1206R7, *Conversions from ranges to containers*

ranges::to

a function that can materialize any range as a container, including nonstandard containers, and recursive containers

"copy"

Container containing containers

Container not in *std*

Why ranges::to?

Simplify Code
Invoke Pipelines

- **This...**

```
std::list lst = /*...*/;
std::vector vec {std::begin(lst), std::end(lst)};
```

- **...becomes...**

```
std::vector vec = lst | ranges::to();
```

- **...or**

```
std::vector vec = ranges::to(lst);
```

- **This...**

```
auto view = ranges::iota(42);
vector < iter_value_t< iterator_t > > vec;
if constexpr(SizedRanged) {
   vec.reserve(ranges::size(view)));
   }
ranges::copy(view, std::back_inserter(vec));
```

- **...becomes...**

```
auto vec = ranges::iota(0, 42) | ranges::to();
```

- **This...**

```
std::map map = get_widgets_map();
std::vector<typename decltype(map)::value_type> vec;
vec.reserve(map.size());
ranges::move(map, std::back_inserter(vec));
```

- **...becomes...**

```
auto vec = get_widgets_map() | ranges::to();
```

What Capabilities?

```cpp
constexpr std::string_view data {"ZYXWVU"sv};

// create a container with the elements of data
auto a = ranges::to<std::vector<char>>(data);

// explicit conversion char -> long
auto b = ranges::to<std::vector<long>>(data);


// deduce the element value type
auto c = ranges::to<std::vector>(data);
```

```cpp
// Convert sequence containers to associative ones
auto d = ranges::to<std::set>(c);

// Convert associative to sequence containers
auto e = ranges::to<std::list>(d);
```

```cpp
// Convert nested ranges
std::list<std::forward_list<int>> lst =
    {{0, 1, 2, 3}, {4, 5, 6, 7}};


auto vec1 =
    ranges::to<std::vector<std::vector<int>>>(lst);


auto vec2 =
    ranges::to<std::vector<std::deque<double>>>(lst);
```

```
// Work in pipelines with deduced elements

auto f = data |
         ranges::view::take(3) |
         ranges::to<std::vector>();

// Also work with specified elements to support conversion

auto g = data |
         ranges::view::take(3) |
         ranges::to<std::vector<long>>();
```

Substandard C++ Warning

Conceptware Ahead!

Works with GCC 13.2 / 14.0
and Clang 18.1 C++23

Code Conventions

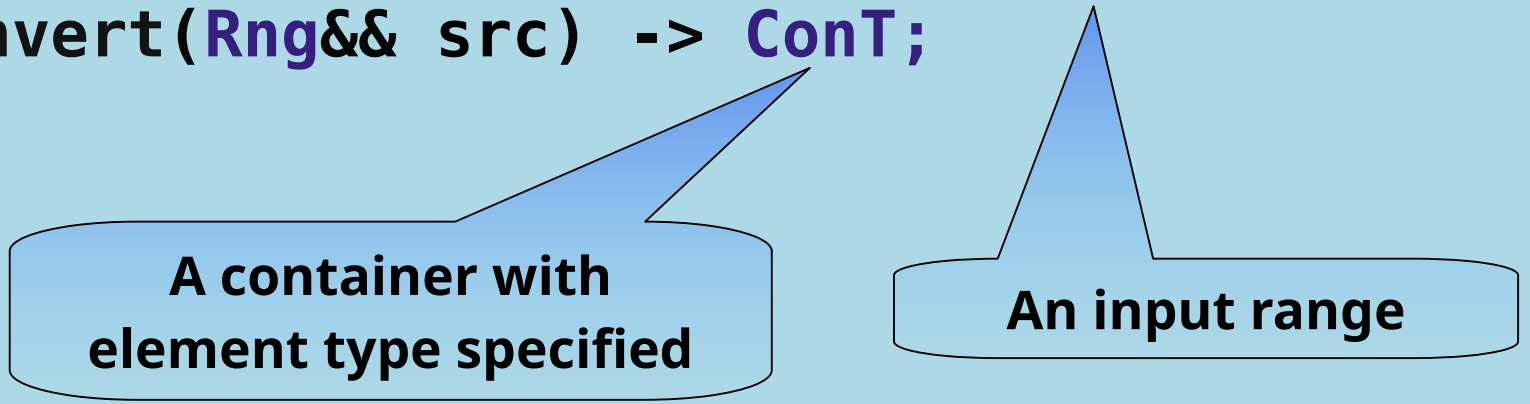
Template parameter naming convention

```
T     => element data type: int
Con   => container without data type: vector
ConT  => container with data type: vector<int>
Rng   => range: rng::input_range
```

```
namespace rng = std::ranges;
namespace vws = std::views;

constexpr std::string_view data {"ZYXWVU"sv};

template<typename ConT>
void con_test(std::string_view const text,
              auto&& src) {
   fmt::println("{}: {}", text, convert<ConT>(src));
}
```

Signature of *convert* with Element Type

```
// convert<vector<int>>(…)

template<typename ConT, rng::input_range Rng> //
auto convert(Rng&& src) -> ConT;
```



```cpp
template<typename ConT, rng::input_range Rng> //
auto convert(Rng&& src) -> ConT {
    ConT dst(src.size());
    rng::copy(src, dst.begin());
    return dst;
}

auto vec {convert<std::vector<char>>(data)};
auto f_list {convert<std::forward_list<char>>(data)};

// error: class std::set has no size constructor
auto set {convert<std::set<char>>(data)};

// error: class std::stack has no 'begin' member
conf_test<std::stack<char>>("stack", data);
```

- Containers without size constructors or ‘resize’ members
  - set, multiset, unordered_set, unordered_multiset
- Containers without ‘begin’ members
  - stack, queue, priority_queue

```cpp
template<typename ConT, rng::input_range Rng>
auto convert(Rng&& src) -> ConT {
    ConT dst;
    for (auto&& s: src) {
        dst.emplace(s);
    }
    return dst;
}


    con_test<std::set<char>>("set", data);
    con_test<std::stack<char>>("stack", data);
    con_test<std::queue<char>>("queue", data);

    // error: class std::vector has no 'emplace(item)' member
      con_test<std::vector<char>>("vector", data);
```

- Containers without ‘**emplace(item)**’ members
  - deque, forward_list, list, vector
  - They have **emplace(pos, item)**
- **Must have *copy* and *range-for* versions**

Meander: Requires Clause

- Yields a boolean constant expression
- Specifies constraints on template arguments...

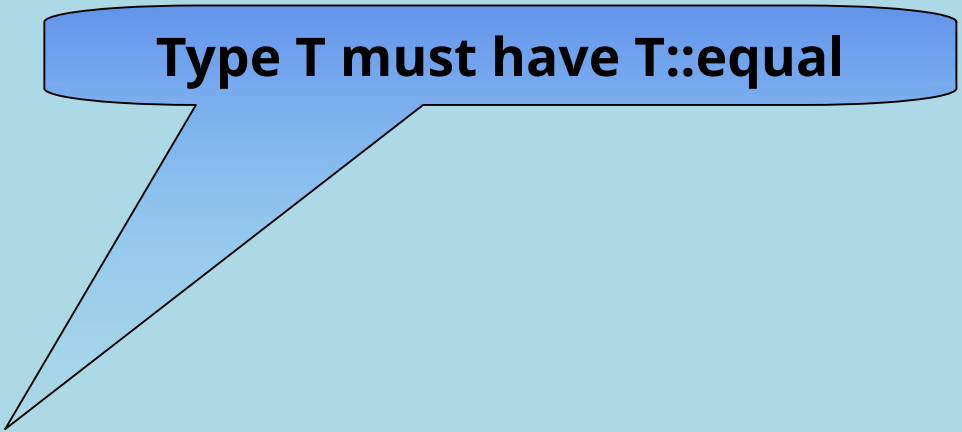
```
template<typename T> requires Eq<T>
void f(T&&){...}
```

- ...or on a function declaration

```
template<typename T>
void f(T&&) requires Eq<T> {...}
```

Meander: Requires Expression

- Boolean expression specifying requirement on…
- *if constexpr* or...
- ... on *requires clause*




```
template<typename T>
requires
requires(T t) { t.equal(1); }
void f(T&&){...}
```

```cpp
template<typename ConT, rng::input_range Rng> //
auto convert(Rng&& src) -> ConT {
    ConT dst;
    if constexpr (requires { dst.resize(1); }) {

        dst.resize(src.size());
        rng::copy(src, dst.begin());

    } else {

        for (auto&& s: src) {
            dst.emplace(s);
        }
    }
    return dst;
}
```

```cpp
template<typename ConT, rng::input_range Rng> //
requires( requires(ConT c) { c.resize(1);} )
auto convert(Rng&& src) -> ConT {
    ConT dst(src.size());
    rng::copy(src, dst.begin());
    return dst;
}

template<typename ConT, rng::input_range Rng> //
auto convert(Rng&& src) -> ConT {
    ConT dst;
    for (auto&& s: src) {
        dst.emplace(s);
    }
    return dst;
}
```

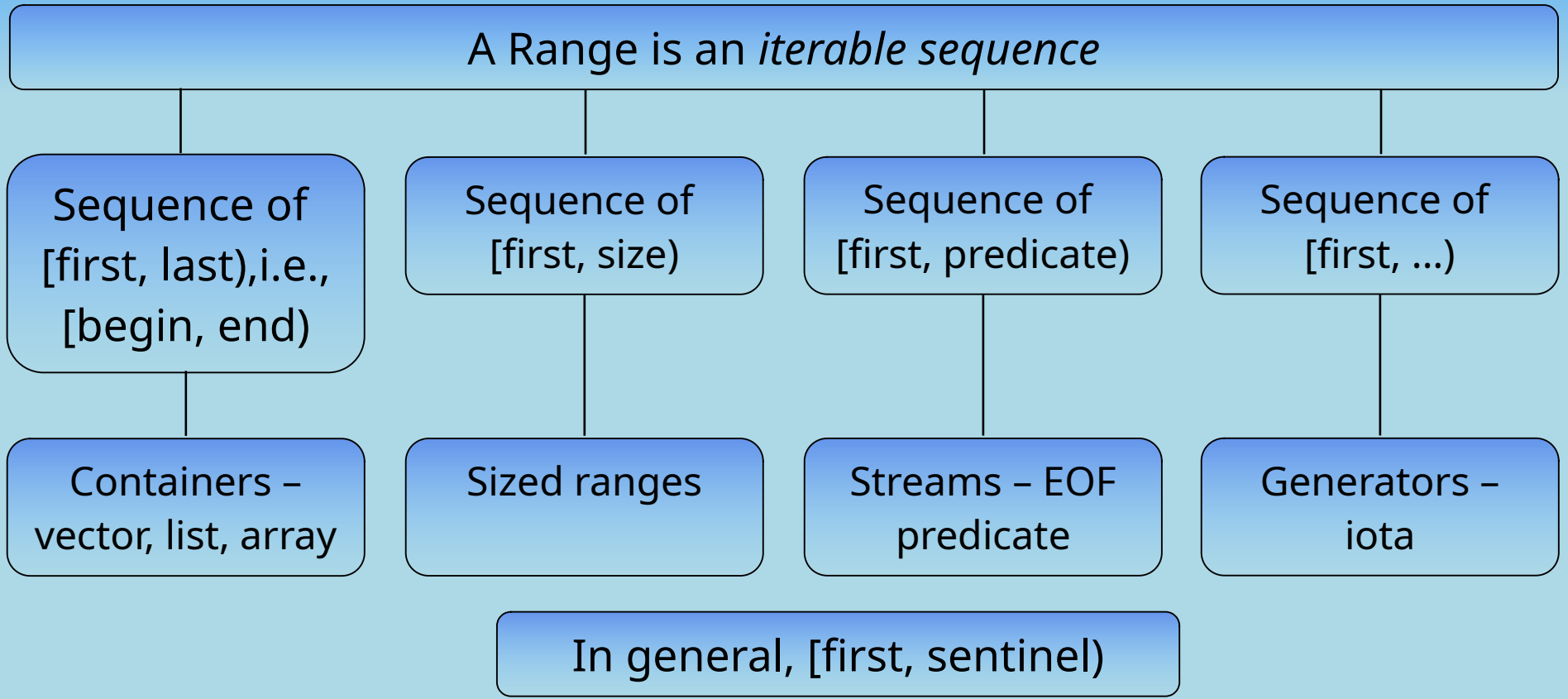
```
template<typename ConT, rng::input_range Rng> //
requires( requires(ConT c) { c.resize(1);} )
auto convert(Rng&& src) -> ConT {…}


template <typename E>
concept Emplaceable = requires(E e) { e.emplace(1); };

template<Emplaceable ConT, rng::range Rng>
auto convert(Rng&& src) -> ConT {
    ConT dst;
    for (auto&& s: src) {
        dst.emplace(s);
    }
    return dst;
}
```

```
template<typename ConT, rng::input_range Rng> //
auto convert(Rng&& src) -> ConT {
    return ConT {rng::cbegin(src), rng::cend(src)};
}
```

Initialization instead of copy

Use *range::begin / end*, not member functions to handle *[first, sentinel)*

Meander: What is a Range?



```cpp
template<typename ConT, rng::input_range Rng>
auto convert(Rng&& src) -> ConT {
    return ConT {rng::cbegin(src), rng::cend(src)};
}

// conversion from char to int works in all versions
con_test<std::vector<int>>("vec int", data);
con_test<std::set<int>>("set int", data);

// pipeline works with all versions
con_test<std::vector<char>>("take",
                            data | vws::take(3));

//  error: can't do nested – nested type not char
con_test<std::vector<char>>("chunk",
                            data | vws::chunk(2));
```

Convert that Deduces Type

***Con* has element type and other default parameters**

Obtain type of element in range, Rng

```
template< template<typename...> typename Con,
          rng::input_range Rng,
          typename T = rng::range_value_t<Rng>>
auto convert(Rng&& src) -> Con<T> {
    return convert< Con<T> >(src);
}
```

Call existing function with typed container

What is Working?

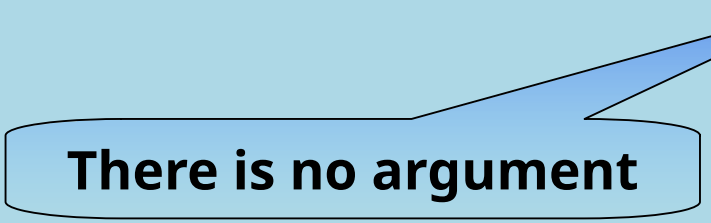
```
// convert with container element type specified
auto a = convert<std::vector<char>>(data |vws::take(3));

// convert to new container element type
auto b = convert<std::vector<long>>(data |vws::take(3));

// convert deducing element type from range type
auto c = convert<std::vector>(data | vws::take(3));

// convert nested ranges from pipeline
auto d = convert<std::vector>(data | vws::chunk(2));

==> chunk: [['Z', 'Y'], ['X', 'W'], ['V', 'U']]
```

Warning!

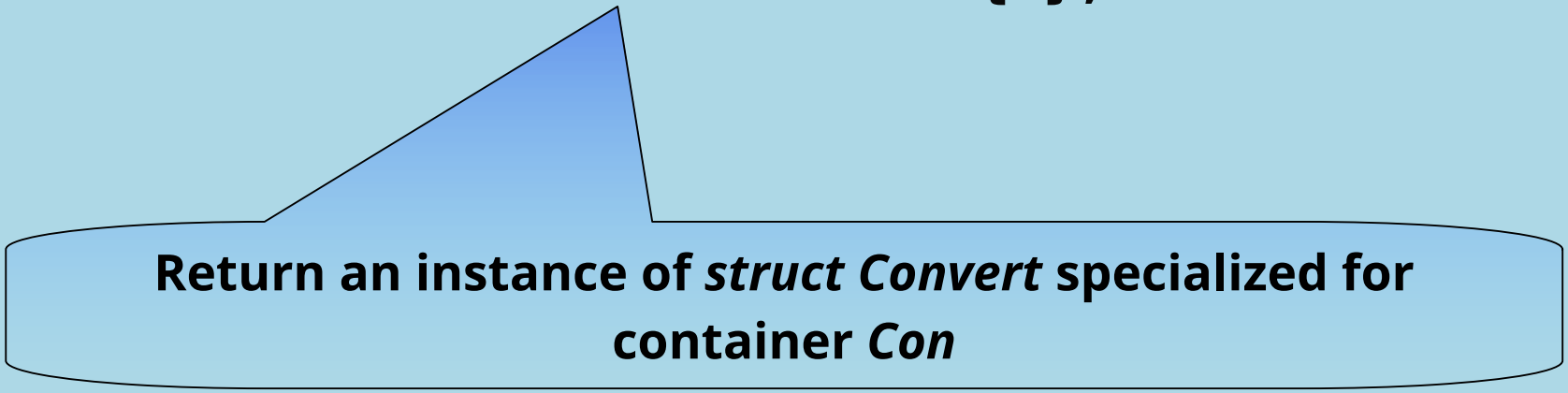
Here There be Dragons

What Remains?

```
// Creating a range adaptor to handle pipe syntax

// deducing data type
auto f = data | vws::take(3) | convert<vector>();

// explicit data type
auto f = data | vws::take(3) | convert<vector<long>();
```



```
// handle: data | convert<container>()

template<template<typename...> typename Con>
constexpr auto convert() -> detail::Convert<Con> {
    return detail::Convert<Con> { };
}
```



```
namespace detail {  // deducing version

   template<template<typename...> typename Con>
   struct Convert :
          rng::range_adaptor_closure<Convert<Con>> {

      template<typename Rng>
      constexpr auto operator()(Rng&& src) {
         return convert<Con>(src);
      }
   };
}
```

The call operator invokes the deducing type *convert* function

Convert inherits from base class providing itself as a parameter. This is the *Curiously Recurring Template Pattern*

Meander: The Curiously Recurring Template Pattern (CRTP)

- Named by James Coplien in the 90s
  - It is an *idiiom* not a design pattern
  - Called *recurring* because it was seen repeatedly
  - The derived class is passed as a template argument to the base class
- A form of *static* or *compile time* polymorphism

```
//  Effectively...

template<typename Derived>
   struct range_adaptor_closure {...};


template<template<typename...> typename Con>
struct Convert :
      range_adaptor_closure<Convert<Con>> {…}


//  define a pipe operator
auto operator|(range_adaptor_closure&& lhs,
               range_adaptor_closure&& rhs) {...}
```

```cpp
template<typename ConT> // explicit type version
struct Convert_Typed :
 std::ranges::range_adaptor_closure<Convert_Typed<ConT>>
{
   template<typename Rng>
   constexpr auto operator()(Rng&& src) {
      return convert<ConT>(src);
      }
   };

// handle: data | convert<container<type>>()
template<typename ConT>
constexpr auto convert() {
   return detail::Convert_Typed<ConT> { };
}
```

Testing Pipelines and Views

- Pipelines are lazy evaluation
- Cannot be stored in a container since each is a different type
- A *std::tuple* manages different types

```
// create a tuple containing pipelines

std::tuple pipes_to_test {
    vws::drop(3),
    vws::take(3),

    vws::chunk(3) | vws::reverse,
    vws::chunk(3) | vws::enumerate,

    vws::split('X') | vws::join | vws::common,
    vws::split('X') | vws::enumerate | vws::common,
};

test_pipes(pipes_to_test, data);
```

vws::join & enumerate don't always model a *common_range*, i.e. provides *begin/end*

```cpp
auto test_a_pipe = [ ](auto pipe, auto const& data) {
    auto vec = data | pipe | convert<std::vector>();

    if constexpr (
      rng::range<typename decltype(vec)::value_type>){

      // handle nested ranges
       fmt::println("R {}", fmt::join(vec, " "));
    }
    else {
       fmt::println("T {}", vec);
    }
};
```

```
void test_pipes(auto& pipes, auto const& test_data) {
    auto convert_pipes = [&test_data]<typename... Ts>
         (Ts const& ... a_pipe) {
             ((test_a_pipe(a_pipe, test_data)), ...);
         };

    std::apply(convert_pipes, pipes);
}
```

std::apply **creates a call to** ***test_a_pipe*** **for each pipeline in** ***pipes***

Rud Merriam, Retired
Embedded Systems & Robotics
The Woodlands, TX USA

I hope you enjoy Toronto and CppNorth 2024

Meandering Through C++
Creating *ranges::to*