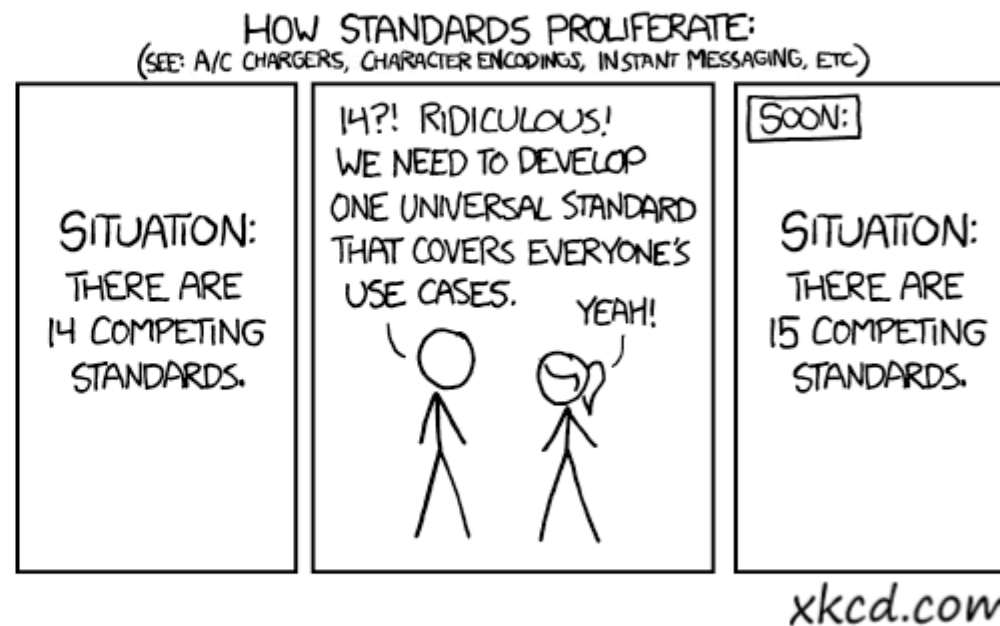


Range-Based Text Formatting

For a Future Range-Based Standard Library

- Text formatting everywhere
- Many different libraries/approaches
- Here is yet another one

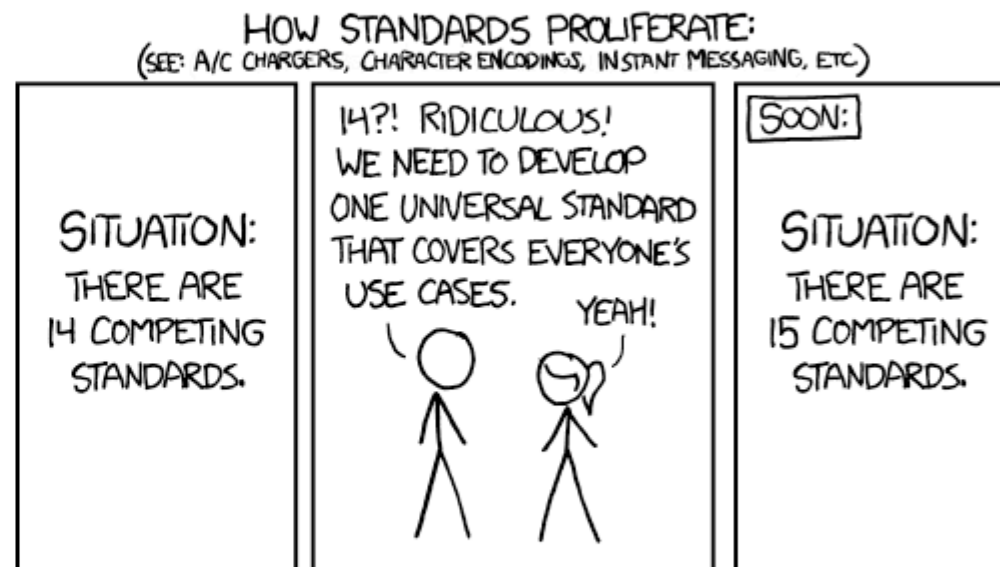
- Text formatting everywhere
- Many different libraries/approaches
- Here is yet another one



- Text formatting everywhere
- Many different libraries/approaches
- Here is yet another one

Input

- components of text
- order of these components
- per component, conversion parameters
 - e.g., number of decimal places



xkcd.com

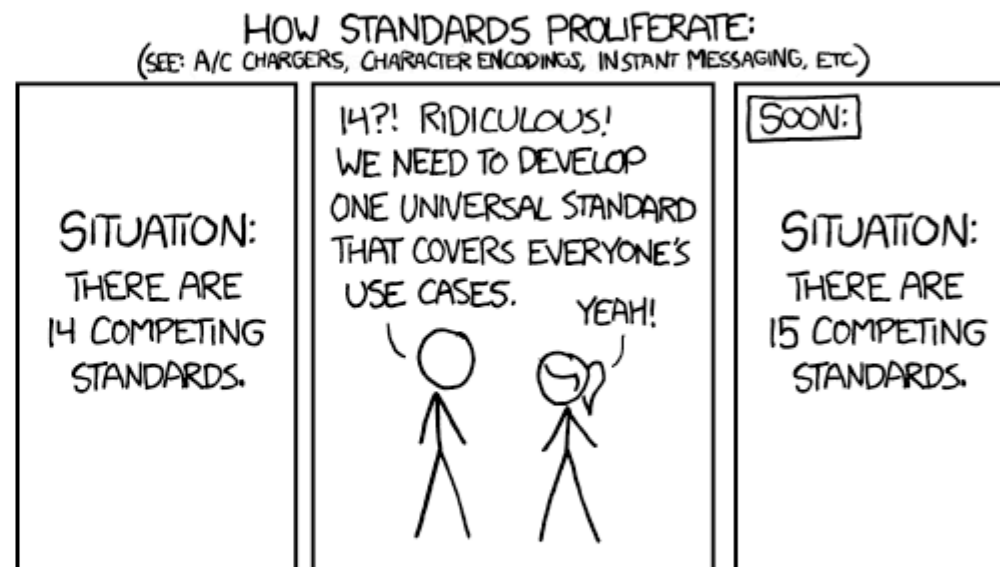
- Text formatting everywhere
- Many different libraries/approaches
- Here is yet another one

Input

- components of text
- order of these components
- per component, conversion parameters
 - e.g., number of decimal places

Output

- a string



xkcd.com

Syntax Options

Order of components and parameters described by

Order of components and parameters described by

- format string
 - `printf("Hello %f", 3.14);`
 - `absl::StrFormat` (printf syntax)
 - C++20 `std::format` (P0645) (Python syntax)

Order of components and parameters described by

- format string
 - `printf("Hello %f", 3.14);`
 - `absl::StrFormat` (printf syntax)
 - C++20 `std::format` (P0645) (Python syntax)
- 'Just C++': functions, parameters, operators
 - `std::stringstream() << std::setprecision(2) << 3.14;`
 - `"Hello " + std::to_string(3.14)`

- format string
 - `printf("Hello %f", 3.14);`
 - `absl::StrFormat` (printf syntax)
 - C++20 `std::format` (P0645) (Python syntax)

Pros

- syntax closer to resulting string
- can decide format string at runtime
 - forgoes compile-time format check

- format string
 - `printf("Hello %f", 3.14);`
 - `absl::StrFormat` (printf syntax)
 - C++20 `std::format` (P0645) (Python syntax)

Cons

- must escape format string (and remember it!)

- format string
 - `printf("Hello %f", 3.14);`
 - `absl::StrFormat` (printf syntax)
 - C++20 `std::format` (P0645) (Python syntax)

Cons

- must escape format string (and remember it!)
- extra language for parameters
 - why not use C++?
 - user-defined types need parser for parameters

- format string
 - `printf("Hello %f", 3.14);`
 - `absl::StrFormat` (printf syntax)
 - C++20 `std::format` (P0645) (Python syntax)

Cons

- must escape format string (and remember it!)
- extra language for parameters
 - why not use C++?
 - user-defined types need parser for parameters
- no gradual change in syntax from string concatenation to formatting
 - `str1 + str2` vs.
 - `std::format("{0}{1}", str1, str2)` vs.
 - `std::format("{0}{1}{2}", str1, n, str2)`

- translation outsourced to agencies
- set up to deal with text, not code
 - XLIFF (XML Localization Interchange File Format)

- translation outsourced to agencies
- set up to deal with text, not code
 - XLIFF (XML Localization Interchange File Format)
- text may contain placeholders

Dear {0}, thank you for your interest in our product.
- So must use format strings!

- translation outsourced to agencies
- set up to deal with text, not code
 - XLIFF (XML Localization Interchange File Format)
- text may contain placeholders

Dear {0}, thank you for your interest in our product.
- So must use format strings!
- BUT: give agency only as much control as necessary
 - insertion position
- formatting parameters provided by OS setting/culture database
 - decimal separator (comma vs. period)
 - number of decimal places
 - date format

```
std::stringstream() << std::setprecision(2) << 3.14;
```

- abuse of operator overloading
 - only because no variadic templates?
- stateful ("manipulators")
 - `std::setprecision` applies to all following items,
 - `std::width` only to next item, ARGH!
- slow due to
 - virtual calls
 - extra copy `std::stringstream` → `std::string`

```
(std::stringstream() << std::setprecision(2) << 3.14).str()
```

```
"Hello " + std::to_string(3.14)
```

- different abuse of operator overloading
- no formatting options
- slow
 - many temporaries

```
"Hello " + std::to_string(3.14)
```

- different abuse of operator overloading
- no formatting options
- slow
 - many temporaries
- BUT: conceptually the essence of formatting
 - turn data into string snippets
 - concatenate the snippets into whole text
 - naturally extends to user-defined types/parameters: call a function
- Overcome weaknesses with Ranges!

Introduction to Ranges

- Who knows the range-based `for`-loop?

Introduction to Ranges

- Who knows the range-based `for`-loop?
- Who knows C++20 Ranges?

Introduction to Ranges

- Who knows the range-based `for`-loop?
- Who knows C++20 Ranges?
- Who knows Eric Niebler's Range-v3 library?

- Who knows the range-based `for`-loop?
- Who knows C++20 Ranges?
- Who knows Eric Niebler's Range-v3 library?
- Who uses ranges every day?
 - Range-v3?

- Who knows the range-based `for`-loop?
- Who knows C++20 Ranges?
- Who knows Eric Niebler's Range-v3 library?
- Who uses ranges every day?
 - Range-v3?
 - Boost.Range?

- Who knows the range-based `for`-loop?
- Who knows C++20 Ranges?
- Who knows Eric Niebler's Range-v3 library?
- Who uses ranges every day?
 - Range-v3?
 - Boost.Range?
 - think-cell library?

- Who knows the range-based `for`-loop?
- Who knows C++20 Ranges?
- Who knows Eric Niebler's Range-v3 library?
- Who uses ranges every day?
 - Range-v3?
 - Boost.Range?
 - think-cell library?
 - home-grown?

```
std::find(itBegin, itEnd, x)
```

→

```
std::ranges::find(rng, x) // C++20
```

- `rng` anything with `begin`, `end`

```
std::find(itBegin, itEnd, x)
```

→

```
std::ranges::find(rng, x) // C++20
```

- `rng` anything with `begin`, `end`
 - containers that own elements (`vector`, `basic_string`, etc.)

```
std::find(itBegin, itEnd, x)
```

→

```
std::ranges::find(rng, x) // C++20
```

- `rng` anything with `begin`, `end`
 - containers that own elements (`vector`, `basic_string`, etc.)
 - views that reference elements (= iterator pairs wrapped into single object)

```
std::find(itBegin, itEnd, x)
```

→

```
std::ranges::find(rng, x) // C++20
```

- `rng` anything with `begin`, `end`
 - containers that own elements (`vector`, `basic_string`, etc.)
 - views that reference elements (= iterator pairs wrapped into single object)
- ranges may do lazy calculations
 - `rng | std::view::filter(pred)` only captures `rng` and `pred`, performs no work
 - skips elements while iterating

Why do I think I know something about ranges?

- think-cell has range library
 - evolved from Boost.Range
 - more powerful than std::ranges
- 1 million lines of production code use it
- chicken-and-egg problem of library design
 - can only learn good design by lots of use
 - with lots of use, cannot change design
- avoid by
 - all code in-house
 - extra resources dedicated to refactoring

Ranges for Text 101: Replace `basic_string` Member Functions by Range Algorithms

```
index = str.find(...);
```

→

```
iterator = std::ranges::find(str,...); // C++20 or tc::find_*
```

- same generic algorithms for character and other sequences
- flexible with string types
 - wrap OS-/library-specific string types in range interface
 - treat uniformly in syntax/algorithms

Ranges for Text 101: Replace `basic_string` Member Functions by Range Algorithms

```
index = str.find(...);
```

→

```
iterator = std::ranges::find(str,...); // C++20 or tc::find_*
```

- same generic algorithms for character and other sequences
- flexible with string types
 - wrap OS-/library-specific string types in range interface
 - treat uniformly in syntax/algorithms
- C++17 `basic_string_view` perpetuates `basic_string` member interface:-(
 - wrap OS-/library-specific string types in range interface
 - treat uniformly in syntax/algorithms

- All Range libraries already have concatenation

```
tc::concat("Hello ", strName) // similar syntax in Range-v3, not yet in C++20
```

- All Range libraries already have concatenation

```
tc::concat("Hello ", strName) // similar syntax in Range-v3, not yet in C++20
```

- To format data, add formatting functions like `tc::as_dec`

```
double f=3.14;  
tc::concat("You won ", tc::as_dec(f,2), " dollars.")
```

- All Range libraries already have concatenation

```
tc::concat("Hello ", strName) // similar syntax in Range-v3, not yet in C++20
```

- To format data, add formatting functions like `tc::as_dec`

```
double f=3.14;  
tc::concat("You won ", tc::as_dec(f,2), " dollars.")
```

- not like `<iostream>`: `double` itself is not a character range:

```
tc::concat("You won ", f, " dollars.") // DOES NOT COMPILE
```

- All Range libraries already have concatenation

```
tc::concat("Hello ", strName) // similar syntax in Range-v3, not yet in C++20
```

- To format data, add formatting functions like `tc::as_dec`

```
double f=3.14;  
tc::concat("You won ", tc::as_dec(f,2), " dollars.")
```

- not like `<iostream>`: `double` itself is not a character range:

```
tc::concat("You won ", f, " dollars.") // DOES NOT COMPILE
```

- No need for special `format` function

- Extensible by functions returning ranges

```
auto dollars(double f) {  
    return tc::concat(tc::as_dec(f,2), " dollars");  
}  
  
double f=3.14;  
tc::concat("You won ", dollars(f), ".");
```

- Extensible by functions returning ranges

```
auto dollars(double f) {  
    return tc::concat(tc::as_dec(f,2), " dollars");  
}  
  
double f=3.14;  
tc::concat("You won ", dollars(f), ".");
```

- Range algorithms work

```
tc::for_each(  
    tc::as_dec(f,2),  
    [](char c){...}  
);  
  
if( tc::all_of/tc::any_of(  
    tc::concat("You won ", tc::as_dec(f,2), " dollars."),  
    [](char c){ return c!='1'; }  
)) {...}
```

`std::string` gives us

- Empty Construction

```
std::string s; // compiles
```

- Construction from literal, another string

```
std::string s1("Hello"); // compiles  
std::string s2(s1); // compiles
```

`std::string` gives us

- Empty Construction

```
std::string s; // compiles
```

- Construction from literal, another string

```
std::string s1("Hello"); // compiles  
std::string s2(s1); // compiles
```

- Add construction from 1 Range

```
std::string s3(tc::as_dec(3.14,2)); // suggested  
std::string s4(tc::concat("You won ", tc::as_dec(3.14,2), " dollars.)); // suggested
```

`std::string` gives us

- Empty Construction

```
std::string s; // compiles
```

- Construction from literal, another string

```
std::string s1("Hello"); // compiles  
std::string s2(s1); // compiles
```

- Add construction from 1 Range

```
std::string s3(tc::as_dec(3.14,2)); // suggested  
std::string s4(tc::concat("You won ", tc::as_dec(3.14,2), " dollars.)); // suggested
```

- Add construction from N Ranges

```
std::string s5("Hello", " World"); // suggested  
std::string s6("You won ", tc::as_dec(3.14,2), " dollars.)); // suggested
```

- What about existing constructors?

```
std::string s1("A", 3 );  
std::string s2('A', 3 );  
std::string s3( 3 , 'A');
```

- What about existing constructors?

```
std::string s1("A", 3 ); // UB, buffer "A" overrun
std::string s2('A', 3 );
std::string s3( 3 , 'A');
```

- What about existing constructors?

```
std::string s1("A", 3 ); // UB, buffer "A" overrun  
std::string s2('A', 3 ); // Adds 65x Ctrl-C  
std::string s3( 3 , 'A');
```

- What about existing constructors?

```
std::string s1("A", 3 ); // UB, buffer "A" overrun
std::string s2('A', 3 ); // Adds 65x Ctrl-C
std::string s3( 3 , 'A' ); // Adds 3x 'A'
```

- What about existing constructors?

```
std::string s1("A", 3 ); // UB, buffer "A" overrun
std::string s2('A', 3 ); // Adds 65x Ctrl-C
std::string s3( 3 , 'A'); // Adds 3x 'A'
```

- Deprecate them!

```
std::string s(tc::repeat_n('A', 3)); // suggested, repeat_n as in Range-v3
```

- think-cell library uses `tc::explicit_cast` to simulate adding/removing explicit constructors:

```
auto s4=tc::explicit_cast<std::string>("Hello", " World");  
auto s5=tc::explicit_cast<std::string>("You won ", tc::as_dec(f,2), " dollars.");
```

- think-cell library uses `tc::explicit_cast` to simulate adding/removing explicit constructors:

```
auto s4=tc::explicit_cast<std::string>("Hello", " World");  
auto s5=tc::explicit_cast<std::string>("You won ", tc::as_dec(f,2), " dollars.");
```

- `tc::cont_emplace_back` wraps `.emplace_back/.push_back`, uses `tc::explicit_cast` as needed:

```
std::vector<std::string> vec;  
tc::cont_emplace_back( vec, tc::as_dec(3.14,2) );
```

- think-cell library uses `tc::explicit_cast` to simulate adding/removing explicit constructors:

```
auto s4=tc::explicit_cast<std::string>("Hello", " World");  
auto s5=tc::explicit_cast<std::string>("You won ", tc::as_dec(f,2), " dollars.");
```

- `tc::cont_emplace_back` wraps `.emplace_back/.push_back`, uses `tc::explicit_cast` as needed:

```
std::vector<std::string> vec;  
tc::cont_emplace_back( vec, tc::as_dec(3.14,2) );
```

- Can `tc::append`:

```
std::string s;  
tc::append( s, tc::concat("You won ", tc::as_dec(f,2), " dollars.") );  
tc::append( s, "You won ", tc::as_dec(f,2), " dollars." );
```

```
tc::concat(  
    "<body>", html_escape(  
        tc::placeholders( "You won {0} dollars.", tc::as_dec(f,2) )  
    ), "</body>"  
)
```

```
tc::concat(  
    "<body>", html_escape(  
        tc::placeholders( "You won {0} dollars.", tc::as_dec(f,2) )  
    ), "</body>"  
)
```

- support for names

```
tc::concat(  
    "<body>", html_escape(  
        tc::placeholders( "You won {amount} dollars on {date}."  
            , tc::named_arg("amount", tc::as_dec(f,2))  
            , tc::named_arg("date", tc::as_ISO8601(  
                std::chrono::system_clock::now()  
            ))  
        )  
    ), "</body>"  
)
```

Naive Implementation

- each formatter returns `std::string`
- `tc::concat` returns `std::string`
- `tc::append` appends `std::strings`

- each formatter returns `std::string`
- `tc::concat` returns `std::string`
- `tc::append` appends `std::strings`

Pro

- simple

- each formatter returns `std::string`
- `tc::concat` returns `std::string`
- `tc::append` appends `std::strings`

Pro

- simple

Con

- need to allocate and copy many strings
- talk would be over

Avoid Heap Allocation For Components

- Make formatter ranges lazy
 - generate character sequence during iteration
 - size of `as_dec`-like formatter objects known at compile-time, no heap allocation

Avoid Heap Allocation For Components

- Make formatter ranges lazy
 - generate character sequence during iteration
 - size of `as_dec`-like formatter objects known at compile-time, no heap allocation

```
auto dollars(double f) {  
    return tc::concat(tc::as_dec(f,2), " dollars");  
}  
  
double f=3.14;  
auto s=tc::explicit_cast<std::string>(tc::concat("You won ", dollars(f), "."));
```

- `tc::as_dec(f,2)` stores `{f,2}`,
- `tc::concat` stores components
 - lvalues stored by reference
 - rvalues stored by copy/move
- like expression templates

Fast Formatting Into Containers

- determine string length
- allocate memory for whole string at once
- fill in characters

- determine string length
- allocate memory for whole string at once
- fill in characters

```
template<typename Cont, typename Rng>  
auto explicit_cast(Rng const& rng) {  
    return Cont(std::begin(rng), std::end(rng));  
}
```

```
// note: there are more explicit_cast implementations for types other than containers
```

- determine string length
- allocate memory for whole string at once
- fill in characters

```
template<typename Cont, typename Rng>  
auto explicit_cast(Rng const& rng) {  
    return Cont(std::begin(rng), std::end(rng));  
}
```

```
// note: there are more explicit_cast implementations for types other than containers
```

- formatters are not random-access
- `string` ctor runs twice over `rng` :-(
 - first determine size
 - then copy characters

- avoid traversing `rng` twice
 - `rng` implements `size()` member
 - explicit loop to take advantage of `std::size`

```
template<typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
> >
auto explicit_cast(Rng const& rng) {
    Cont cont;
    cont.reserve( std::size(rng) );
    for(auto it=std::begin(rng); it!=std::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
    return cont;
}
```

- also have `tc::append`

```
template<typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
> >
void append(Cont& cont, Rng const& rng) {
    cont.reserve( cont.size() + std::size(rng) );
    for(auto it=std::begin(rng); it!=std::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

- also have `tc::append`

```
template<typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
> >
void append(Cont& cont, Rng const& rng) {
    cont.reserve( cont.size() + std::size(rng) );
    for(auto it=std::begin(rng); it!std::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

- all good?

- also have `tc::append`

```
template<typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
> >
void append(Cont& cont, Rng const& rng) {
    cont.reserve( cont.size() + std::size(rng) );
    for(auto it=std::begin(rng); it!=std::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

- `.reserve` is evil!!!

- when adding N elements, guarantee $O(N)$ moves and $O(\log(N))$ memory allocations!

```
template< typename Cont >
void cont_reserve( Cont& cont, typename Cont::size_type n ) {
    if( cont.capacity()<n ) {
        cont.reserve(max(n,cont.capacity()*8/5));
    }
}
```

```
template<typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
> >
void append(Cont& cont, Rng const& rng) {
    tc::cont_reserve( cont.size() + std::size(rng) );
    for(auto it=std::begin(rng); it!=std::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

```
template<typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
> >
void append(Cont& cont, Rng const& rng) {
    tc::cont_reserve( cont.size() + std::size(rng) );
    for(auto it=std::begin(rng); it!=std::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

Next bottleneck: iterators!

- `concat`

- iterator is `std::variant` of component iterators
- each `operator*` and `operator++` branches on the variant

```
iterator::operator++() {
    std::visit( make_overload(
        [&](Iterator1& it1){
            ++it1;
            if (it1==std::end(m_rng1)) {
                m_variant_of_its=std::begin(m_rng2);
            }
        },
        [&](Iterator2& it2){
            ++it2;
        }
    ), m_variant_of_its );
}
```

- `concat`

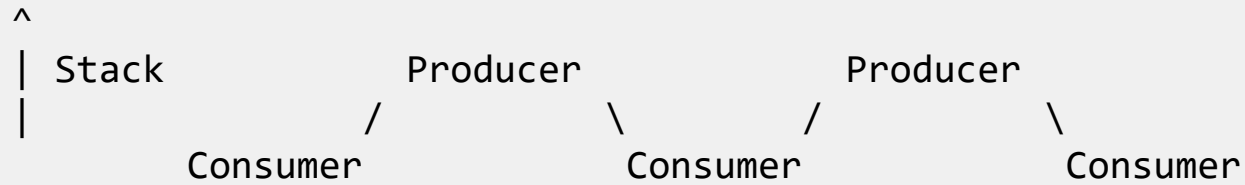
- iterator is `std::variant` of component iterators
- each `operator*` and `operator++` branches on the variant

```
iterator::operator++() {
    std::visit( make_overload(
        [&](Iterator1& it1){
            ++it1;
            if (it1==std::end(m_rng1)) {
                m_variant_of_its=std::begin(m_rng2);
            }
        },
        [&](Iterator2& it2){
            ++it2;
        }
    ), m_variant_of_its );
}
```

- `tc::as_dec`

- iterator bookkeeping costs performance

- C++ iterators do external iteration
- Consumer calls producer to get new element



- Consumer is at bottom of stack
- Producer is at top of stack

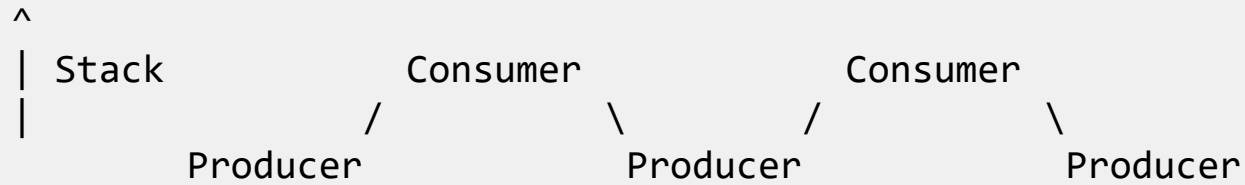
Consumer is at bottom of stack

- contiguous code path for whole range
- easier to write
- better performance
 - state encoded in instruction pointer
 - no limit for stack memory

Producer is at top of stack

- contiguous code path for each item
- harder to write
- worse performance
 - single entry point, must restore state
 - fixed amount of memory or go to heap

- Formatting text is more efficient with internal iteration
- Producer calls consumer to offer new element



Producer is at bottom of stack

- ... all the advantages of being bottom of stack ...

Consumer is at top of stack

- ... all the disadvantages of being top of stack ...

Many Range Algorithms OK with Internal Iteration

| Algorithm | Internal Iteration? |
|---------------|---|
| binary_search | no (random access iterators) |
| find | no (single pass iterators); yes if only value |

Many Range Algorithms OK with Internal Iteration

| Algorithm | Internal Iteration? |
|---------------|---|
| binary_search | no (random access iterators) |
| find | no (single pass iterators); yes if only value |
| for_each | yes |
| accumulate | yes |
| all_of | yes |
| any_of | yes |
| none_of | yes |
| ... | |

Many Range Algorithms OK with Internal Iteration

| Algorithm | Internal Iteration? |
|---------------|---|
| binary_search | no (random access iterators) |
| find | no (single pass iterators); yes if only value |
| for_each | yes |
| accumulate | yes |
| all_of | yes |
| any_of | yes |
| none_of | yes |
| ... | |

| Adaptor | Internal Iteration? |
|-----------|---------------------|
| filter | yes |
| transform | yes |

Many Range Algorithms OK with Internal Iteration

| Algorithm | Internal Iteration? |
|---------------|---|
| binary_search | no (random access iterators) |
| find | no (single pass iterators); yes if only value |
| for_each | yes |
| accumulate | yes |
| all_of | yes |
| any_of | yes |
| none_of | yes |
| ... | |

| Adaptor | Internal Iteration? |
|-----------|---------------------|
| filter | yes |
| transform | yes |

- Extend Range concept to internal iteration!

- Range implements `operator()` that takes sink functor
 - Con: C++20 `std::span::operator()` already used, must SFINAE it out
 - Pro: can be written as lambda

```
tc::for_each(  
    // the range  
    [](auto sink) {  
        sink(1);  
        sink(2);  
    },  
    // the sink functor  
    [](int n) {  
        consume(n);  
    }  
);
```

- `tc::for_each` uses internal iteration if available (never slower than iterators)
- otherwise uses iterators

```
template<typename... Rngs>
struct concat {
    std::tuple<Rngs...> m_rng;

    template<typename Sink>
    void operator()(Sink sink) const {
        // tc::for_each also works on tuples
        tc::for_each(m_rng, [&](auto const& rng) {
            tc::for_each(rng, sink);
        });
    }
};
```

- no overhead

- introduce `appender` sink for `explicit_cast` and `append` to use

```
template<typename Cont, typename Rng>
void append(Cont& cont, Rng&& rng) {
    tc::for_each(std::forward<Rng>(rng), tc::appender(cont));
}
```

- introduce `appender` sink for `explicit_cast` and `append` to use

```
template<typename Cont, typename Rng>
void append(Cont& cont, Rng&& rng) {
    tc::for_each(std::forward<Rng>(rng), tc::appender(cont));
}
```

- `appender` customization point
 - returned by `container::appender()` member function
 - default for `std::` containers

```
template<typename Cont>
struct appender {
    Cont& m_cont;
    template<typename T> void operator()(T&& t) {
        tc::cont_emplace_back(m_cont, std::forward<T>(t));
    }
};
```

- What about `reserve`?
 - Sink needs whole range to call `std::size` before iteration

- What about `reserve`?
 - Sink needs whole range to call `std::size` before iteration
- new Sink customization point `chunk`
 - if available, `tc::for_each` calls it with whole range

```
template<typename Cont, enable_if<Cont has reserve()> >
struct reserving_appender : appender<Cont> {
    template<typename Rng, enable_if<Rng has size()> >
    void chunk(Rng&& rng) const {
        tc::cont_reserve( m_cont, m_cont.size()+std::size(rng) );
        tc::for_each( std::forward<Rng>(rng),
            static_cast<appender<Cont> const&>(*this)
        );
    }
};
```

- file sink advertises interest in contiguous memory chunks

```
struct file_appender {  
    void chunk(std::span<unsigned char const> rng) const {  
        std::fwrite(rng.begin(), 1, rng.size(), m_file);  
    }  
    void operator()(unsigned char ch) const {  
        chunk(tc::single(ch));  
    }  
};
```

Performance: Appender vs Hand-Written

- How much loss compared to hand-written code?
 - trivial formatting task 10x 'A' + 10x 'B' + 10x 'C' best to expose overhead

```
struct Buffer {
    char achBuffer[1024];
    char* pchEnd=&achBuffer[0];
} buffer;

void repeat_handwritten(char chA, int cchA,
                       char chB, int cchB,
                       char chC, int cchC
) {
    for (auto i = cchA; 0 < i; --i) {
        *buffer.pchEnd=chA;
        ++buffer.pchEnd;
    }
    ... cchB ... chB ...
    ... cchC ... chC ...
}
```

```
struct Buffer {
    ...
    auto appender() & {
        struct appender_t {
            Buffer* m_buffer;
            void operator()(char ch) noexcept {
                *m_buffer->pchEnd=ch;
                ++m_buffer->pchEnd;
            }
        };
        return appender_t{this};
    }
} buffer;
void repeat_with_ranges(char chA, int cchA,
                       char chB, int cchB,
                       char chC, int cchC ) {
    tc::append(buffer, tc::repeat_n(chA,cchA), tc::repeat_n(chB,cchB),
               tc::repeat_n(chC,cchC));
}
```

Performance: Appender vs Hand-Written

- `repeat_n` iterator-based
 - ~50% more time than hand-written (Visual C++ 15.8)
- `repeat_n` supports internal iteration
 - ~15% more time than hand-written (Visual C++ 15.8)
- Test is worst case: actual work is trivial
 - smaller difference for, e.g., converting numbers to strings

- toy `basic_string` implementation
 - only heap: pointers `begin`, `end`, `end_of_memory`
- Again trivial formatting task: 10x 'A' + 10x 'B' + 10x 'C'

```
void repeat_with_ranges(  
    char chA, int cchA,  
    char chB, int cchB,  
    char chC, int cchC  
) {  
    tc::append(mystring,  
        tc::repeat_n(chA,cchA), tc::repeat_n(chB,cchB),  
        tc::repeat_n(chC,cchC));  
}
```

- Standard Appender

```
template<typename Cont>
struct appender {
    Cont& m_cont;
    template<typename T>
    void operator()(T&& t) {
        m_cont.emplace_back(std::forward<T>(t));
    }
};

template<typename Cont, enable_if<Cont has reserve()> >
struct reserving_appender : appender<Cont> {
    template<typename Rng, enable_if<Rng has size()> >
    void chunk(Rng&& rng) const {
        tc::cont_reserve( m_cont, m_cont.size()+std::size(rng) );
        tc::for_each( std::forward<Rng>(rng),
            static_cast<appender<Cont> const&>(*this)
        );
    }
};
```

- Custom Appender

```
template<typename Cont>
struct mystring_appender : appender<Cont> {
    Cont& m_cont;
    template<typename T>
    void operator()(T&& t) {
        m_cont.emplace_back(std::forward<T>(t));
    }
    template<typename Rng, enable_if<Rng has size()> >
    void chunk(Rng&& rng) const {
        tc::cont_reserve( m_cont, m_cont.size()+std::size(rng) );
        tc::for_each( std::forward<Rng>(rng),
            [&](auto&& t) {
                *m_cont.m_ptEnd=std::forward<decltype(t)>(t);
                ++m_cont.m_ptEnd;
            }
        );
    }
};
```

Performance: Custom vs. Standard Appender

- String was only 30 characters
- Heap allocation
- Custom Appender ~20% less time (Visual C++ 15.8)
- Requires own `basic_string` implementation
 - uninitialized buffer not exposed by `std::basic_string/std::vector`

- if not all snippets implement `size()`: new customization point `min_size()`?
 - `concat::min_size()` is sum of `min_size()` of components
 - `min_size()` never wrong to return `0`
- custom file appender that fills fixed I/O buffer
 - replace `std::FILE` buffer with own buffer
 - offer unchecked write as long as snippet `size()` still fits
 - new customization point `max_size`?

Conclusion

- Use Range syntax and algorithms for text formatting
- For performance, need new customization points, `Range::operator()`, `appender`, `chunk`
- Then performance competitive with hand-written code

think-cell library is at <https://github.com/think-cell/think-cell-library> [NEWS: now under Boost license]



Or if you want to help: www.think-cell.com/developers