C++23 Standard Library Preview

Jeff Garland

Created: 2021-11-11 Thu 12:42

intro

status of c++23 -> ship on time

- it is an immutable property of committee process
  - 'train model' - what is ready ships
  - there is a plan: http://wg21.link/P0592
  - 'must focus' list has zero language features
- Library features in discussion
  - networking
  - modules support
  - concurrency support
  - bug fixes

on planning

“no plan survives contact with the enemy”

– various military leaders, summarized

c++23 - covid edition

- committee has **`always`** depended on face-to-face
  - 2020-02 meeting (Prague) was last
  - 2022-06 likely next meeting
  - 2022-02 –> c++23 feature freeze
- all committee work has gone virtual
  - subgroup meetings on going every week
  - 3 virtual plenary sessions / year
- c++23 changes will be less
  - more bug fix less feature?
  - some think thats good

priorities for c++23 - library

- bug fixes – always a high priority
  - C++20 ~~was perfect~~ mistakes were made
  - incompatible `format` and `ranges` changes made
- ~~networking is number 1~~
- support for modules
- concurrency support – co-routines
  `std::generator`?
- additions to c++20 features `ranges`, `format`
- mop up c++20 stuff that missed the train

outline of the talk

- string processing
- standard library modules support
- i/o additions
- ranges changes & additions
- constexpr all the things
- utilities – `stacktrace`, `expected`, `byte_swap`, monadic `optional`
- containers
- c interfaces

outline of the talk

- string processing **(partial)**
- standard library modules support
- i/o additions
- ranges changes & additions **(partial)**
- constexpr all the things **(one slide)**
- utilities – `stacktrace`, `expected`, `byte_swap`, monadic `optional` **(pick one)**
- ~~`containers`~~
- ~~`c interfaces`~~

things not covered

- in flight
  - `std::execution` https://wg21.link/P2300
  - `generator` https://wg21.link/p2168
  - `mdspan` https://wg21.link/p0009
  - freestanding
  - `flat_map` `flat_set`
  - extended floating point http://wg21.link/p1467
  - cbegin const itr http://wg21.link/p2278
- done
  - `invoke_r`
  - `move_only_function` https://wg21.link/p0288

string processing

- **`basic_string::contains`**
- **`string` and nullptr**
- `resize_and_overwrite`
- `string_view` range constructor

`string` and `string_view` `contains`

- `starts_with` and `ends_with` got added in c++20
- add to `std::string` and `std::string_view`
- in the working draft (gcc11, clang12, msvc19.30)
- https://wg21.link/P1679
- https://godbolt.org/z/eW7nvnT4v

```
std::string s( "the answer is 42" );

if ( !s.contains( "foo" ) && s.ends_with( "42" ) )
{
  std::print("so long, and thanks for all the fish\n");
}
```

string - string_view and nullptr

```cpp
std::string doit()
{
  // some long block of complex code
  return nullptr;
}

int main()
{
  std::string s = doit();
  std::print("{}\n", s);
}
```

`string` and `nullptr`

```
terminate called after throwing an instance of 'std::logic_error'
  what():  basic_string::_M_construct null not valid

to:

<source>: In function 'std::string doit()':
<source>:7:9: error: use of deleted function 'std::__cxx11::basic_strin
    7 |   return nullptr;
      |          ^~~~~~~
```

- in the working draft (gcc12, clang13, msvc 19.30)
- https://wg21.link/P2166
- https://godbolt.org/z/MxdM6a3fq

resize_and_overwrite

```
template<class Operation>
constexpr void resize_and_overwrite(size_type n, Operation op);
```

- facility to improve performance for strings
- resize and initialize with user callback
- skips steps otherwise required:
  - extra initialization
  - extra copies
  - extra bookkeeping
- use with caution
- pass an operation and a size
- https:wg21.link/P1072 `basic_string::resize_and_overwrite`

range constructor for `string_view`

```
template<class Range>
constexpr basic_string_view(Range&& r)
      requires ranges::contiguous_range and ranges::sized_range();
```

- https://wg21.link/p1394
- https://wg21.link/p1989
- in the working draft (gcc11, clang ?, msvc 19.30)

modules support

c++20 modules

- upgrade `#include` compilation model
- keyword `import` is used for module access
- modules support non-textual/binary model
- 'header modules' allow import of single header aka `<iostream>`

considerations: what granularity / size?

- one for each header?
- import `std.core`, `std.container`?
- one for each type?
- import `std`?

considerations: ease of use

- `import std` is easy to teach
- most if not all macros are not included
  - what about `assert`?
- what about C functions in global namespace?
  - some might be there, but not all

modules example

```
//import -- no macros
import std           // all standard facilities except C global namespac
import std.compat    // above plus ::fopen and other global namespace th
#include <version>  // feature test macros not in modules
#include <cassert>  // if you need assert
```

compilation speed

	#include needed	import std	#include all	import all
hello world <iostream>	.87s	.08s	3.42s	.62s
mix (9 headers)	2.2s	.44s	3.53s	.99s

- Minimal module support https://wg21.link/p2412

significant performance advantage

For "the fair comparisons" (importing and #including all of the standard-library), import beats #include between 7 to 60 times. It would not be unreasonable to expect improvements in the 5 to 25 range for realistic examples.

module status

- Design and final wording approval is pending
- Standard Library Modules https://wg21.link/P2465
- Minimal module support https://wg21.link/p2412

i/o additions

- changes and additions to `format`
- `std::print`
- `format` specializations for std lib
- formatting ranges
- `spanstream`

Additions to `format`

- in c++20 we got `std::format`
- provides python like formatting to strings

format error example

```
int main()
{
  //oops, :s will not format 42
  string s = format( "{:s} answer is {:s}","The", 42 );
}
```

compile time vs run time

```
//runtime error:
Program returned: 139
terminate called after throwing an instance of 'fmt::v7::format_error'
  what():  invalid type specifier

to:
//compile time error:
error: call to non-'constexpr' function 'void fmt::v8::detail::error_ha
 2663 |     eh.on_error("invalid type specifier");
```

format fix detail

- mismatched parameters were a runtime error
- format string mismatch –> compiler error
- need to use `vformat` for runtime format strings
- fix back ported to c++20
- before: https://godbolt.org/z/16KzjrsPc
- after: https://godbolt.org/z/1aWx88W3Y
- also see: Charles Barto's talk from Thu cppcon 2021

`std::print` example

```
 #include <print> //new header
//c++20
string s = format( "{} answer is {}\n", "The", 42 );
cout << s;

//put string to cout
print( "{} answer is {}", "The", 42 );

//put string to a file handle
FILE* fp = fopen("junk.txt", "w+");
print(fp, "{} answer is {}", "The", 42 );

// or to a ostream
s = "charlie";
println(cerr, "sorry {}", s);
```

- https://godbolt.org/z/sGKPGe51K

`print` details

- pretty much eliminates the need for ostreams
- superior unicode support
- highly optimized code
- all based in `fmt` library
- specific support for unicode via `vprint_unicode`
- in final design review

`print` performance

- numbers are OS dependent
- `print` competative or outperforms `printf`
- 3x faster than cout

Benchmark	**Time**	**CPU**
`printf`	87.0 ns	86.9 ns
`ostream`	255 ns	255 ns
`print`	78.4 ns	78.3 ns
`print_cout`	89.4 ns	89.4 ns
`print_cout_sync`	91.5 ns	91.4 ns

`std::print` resources

- paper https://wg21.link/P2093
- fmt lib https://fmt.dev/latest/index.html

Format specializations for std library types

- in c++20 a fair number of types didn't get format specializations
- `error_code`, `bitset`, `regex::sub_match`, `thread::id`
- won't be fixed: `shared_ptr`, `unique_ptr`, `complex`, `filesystem::path`
- https://wg21.link/P1636

formatting ranges

```
std::vector v{10,20,30,40,50,60};
std::string hex = format("{:02X}", fmt::join(v, ":"));
//hex ==  0A:14:1E:28:32:3C
print("hex == {}\n", hex);

std::vector<std::string> vs = {"foo", "bar", "baz"};
//["foo", "bar", "baz"]
print("[{}]\n");
```

ranges formatting

- https://godbolt.org/z/17sjEdaqK
- formatting ranges wg21.link/p2286

spanstream

span based input/output

- user provided buffer for std streams
- access to buffer via span
- types `basic_spanbuf`, `basic_spanstream`
- feature from original 1998 `strstream`

`span` refresher

- a non-owning 'view' over contiguous sequence of object
- cheap to copy - implementation is a pointer and size
- constant time complexity for all member functions
- defined in header `<span>`
- unlike most 'view types' can mutate
- constexpr ready

span construction

```
vector<int> vi = { 1, 2, 3, 4, 5 };
span<int>   si ( vi );

array<int, 5> ai = { 1, 2, 3, 4, 5 };
span<int>     si2 ( ai );

int cai[] = { 1, 2, 3, 4, 5 };
span<int>     si3( cai );

array<string, 2> sa = { "world" , "hello "};
span<string> ss ( sa );
```

span as a function parameter

```
void print_reverse(span<int> si) { //by value
  for ( auto i  : std::ranges::reverse_view{si} ) {
    cout << i << " " ;
  }
  cout << "\n";
}

int main () {

  vector<int> vi = { 1, 2, 3, 4, 5 };
  print_reverse( vi ); //5 4 3 2 1

  int cai[] = { 1, 2, 3, 4, 5 };
  print_reverse ( cai ); //5 4 3 2 1

  span<int> si ( vi );
  print_reverse( si.first(2) ); //2 1
  print_reverse( si.last(2) );  //5 4
```

`spanstream` example

```
//warning: uncompiled example

#include <spanstream> //new header

char deep_thoughts[] = "foo of the bar";
ispanstream is{span<char>{deep_thoughts}};
ospanstream os{};
os.span( deep_thoughts ); //set the span
os << "hello world";
span s = os.span(); //get the span
```

`spanstream` status and references

- in working draft (gcc ?, clang ?, msvc 19.30)
- http://wg21.link/p0448
- https://github.com/PeterSommerlad/SC22WG21_Pap

ranges additions

- fix c++20 issues
- new range algorithms
- new range adaptors and views
- convert range to container `ranges::to`
- pipe support for user views
- support interoperation with non-range algorithms
- http://wg21.link/P2214 plan for ranges 23
- Tina Ulbrich How to rangify your code - meeting c++ 2021

c++20 ranges 1 minute review

range	something that can be iterated over
range algo	algorithm that takes range
view	lazy range thats cheap (to copy)
range adaptor	make a range into a view

the ranges way: `sort`

```
std::array<int, 6> a { 6, 2, 3, 4, 5, 1 };

std::sort ( a.begin(), a.end() );

std::ranges::sort ( a ); //clear, obvious meaning, -13 characters
```

New algorithms overview

- `shift_left`, `shift_right`
- `fold_left`, `fold_right`
- `starts_with`, `ends_with`
- `iota`
- `find_last`

`fold`

- `fold` is a generalization of accumulate
- takes an algorithm and optional initial value
- directional: from 'left' or 'right'
- use start of range for initial value
  `fold_left_first`, `fold_right_last`
- supports projections (aka filters)

`fold` example

```
template<...>
constexpr auto ranges::fold_left(R&&, F f, Proj proj = {});


std::vector<double> v = {0.25, 0.75};
auto result = fold_left( v, 0, std::plus() ) //1.0
```

`shift_left` and `shift_right` algorithms

- move elements a fixed amount
- return an iterator to one past shift point
- `size` of sequence unchanged

`shift_left` example

```
vector<int>    vi = { 1, 2, 3, 4, 5 };

auto itr = ranges::shift_left(vi, 2);
//vi = 3 4 5 4 5
//           ^
//         itr
vi.erase( itr, vi.end());

for ( auto i : vi ) {
  cout << i << " ";
}
cout << "\n";  //3 4 5
```

- https://godbolt.org/z/dr3PW3jso

`shift` synopsis

```
// 25.6.14, shift
template<class ForwardIterator>
constexpr ForwardIterator
shift_left(ForwardIterator first, ForwardIterator last,
           typename iterator_traits<ForwardIterator>::difference_type n

template<class ForwardIterator>
constexpr ForwardIterator
shift_right(ForwardIterator first, ForwardIterator last,
            typename iterator_traits<ForwardIterator>::difference_type

 //also overloads for parallel execution
```

`shift` reference

- merged in working draft
- c++20 proposal http://wg21.link/p0769
- c++20 impl https://github.com/danra/shift_proposal
- c++23 https://wg21.link/p2440

`starts_with` and `ends_with`

- check if a range begins or ends with another range

`starts_with` example

```
auto some_ints = view::iota(0, 10);
auto some_more_ints = view::iota(0, 5);
if (ranges::starts_with(some_ints, some_more_ints)) {
   // do something
}

std::vector<char> fname{"foo.exe"};
if ( ranges::ends_with(fname, "exe") )
{
   print("is executable");
}
```

views and adaptors

- `adjacent`, `adjacent_transform`
- `cartesian_product`
- `chunk`, `chunk_by`
- `join_with`
- `slide`
- `zip`, `zip_transform`

review: `views::filter` example

```
std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };

auto is_even = [](int i) { return 0 == i % 2; };

// view on stack with adaptor
auto evens = vi | std::views::filter( is_even );

for (int i : evens)
{
  std::cout << i << " ";
}
cout << "\n";

// view on in range-for loop
for (int i : std::ranges::filter_view( vi, is_even ))
{
  std::cout << i << " ";
}
cout << "\n";
```

c++20 issue fixes

- `istream_view` was broken in c++20
- `join_view` was broken in c++20
- `split_view` was less than optimal
  - rename to `lazy_split_view`
  - make `split_view` do what you'd expect

ranges and non-range algorithms

```
std::vector<int> data = {1, 5, 3 };
auto v = data | std::views::transform([](int x){ return x * x; });

// parallel algorithms are NOT rangified, this call fails in c++20
int sum_of_squares = std::reduce(std::execution::par, begin(v), end(v))
```

- http://wg21.link/P2408

pipe support for user views

- c++20 - user defined views & adaptors - ok
- c++20 - pipe operators for user adaptors - no

```
C(R)
R | C
```

pipe support resources

- see Jason Rice's presentation from Thu cppcon 2021
- http://wg21.link/p2387

`chunk`, `slide`, `chunk_by`

`chunk` and `slide` example

```
std::vector v = {1, 2, 3, 4, 5};
// [[1, 2], [3, 4], [5]]
fmt::print("{}\n", v | std::views::chunk(2));

// [[1, 2], [2, 3], [3, 4], [4, 5]]
fmt::print("{}\n", v | std::views::slide(2));
```

`chunk_by` example

```
std::vector v = {1, 2, 2, 3, 0, 4, 5, 2};
 // [[1, 2, 2, 3], [0, 4, 5], [2]]
fmt::print("{}\n", v | std::views::chunk_by(ranges::less_equal{}));
```

`zip` and `zip_transform`

- bind multiple ranges with pairwise matching
- especially important for dealing `tuple` and `pair`

`zip` example

```
std::vector v1 = {1, 2};
std::vector v2 = {'a', 'b', 'c'};
std::vector v3 = {3, 4, 5};

// {(1, 'a'), (2, 'b')}
fmt::print("{}\n", std::views::zip(v1, v2));
// {3, 8}
fmt::print("{}\n", std::views::zip_transform(std::multiplies(), v1, v3)
// {('a', 'b'), ('b', 'c')}
fmt::print("{}\n", v2 | std::views::pairwise);
// {7, 9}
fmt::print("{}\n", v3 | std::views::pairwise_transform(std::plus()));
```

enumerate

```
std::vector<string> stuff { "foo", "bar", "baz" };

for ( auto elem : stuff)
{
   print("{}", elem);
}
```

- range for loops, what if we need index?

`enumerate` range for with indexes

```
std::vector<string> stuff { "foo", "bar", "baz" };

for ( auto elem : std::views::enumerate(stuff))
{
   // 0:foo 1:bar 2:baz
   print("{}:{} ", elem.index, elem.value);
}
```

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2164r5.p

`ranges::to`

- convert a range into a collection
- supports a variety of easy collection conversions
- adds constructors to containers for ranges

`ranges::to` example

```
/* output https://godbolt.org/z/MhThnY8rb
[0, 2, 4, 6, 8]
*/
namespace rv = ranges::views;

int main()
{
    std::vector v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    vector<int> data = v | rv::stride(2) | ranges::to<vector>;
    fmt::print("{}\n", data);
}
```

`ranges::to` string example

```
/* output https://godbolt.org/z/bd558h47j
foo -- bar -- baz --
foo-bar-baz
*/
int main()
{
    namespace rv = ranges::views;

    vector<string> vs = {"foo", "bar", "baz", ""};
    string s = vs | rv::join(string_view(" -- ")) | ranges::to<string>(
    cout << s << "\n";

    auto is_not_empty = [](const string& s) { return !s.empty(); };
    s = vs | rv::filter(is_not_empty) | rv::join('-') | ranges::to<stri
    cout << s << "\n";
}
```

ranges resources

- cbegin cleanup http://wg21.link/p2278
- superior string splitting http://wg21.link/p2210
- fix `istream_view` http://wg21.link/P2432
- fix `join_view` http://wg21.link/p2328

constexpr all the things, `if consteval`

- `unique_ptr`
- `std::variant` and `std::optional`
- `type_info`
- `cmath` functions
- `to_chars` and `from_chars`

consteval

- constexpr allows runtime or compile time
- consteval is compile error if cannot resolve at compile time
- how can we do this now?

is_constant_evaluated example

```
constexpr size_t strlen(char const* s) {
    if constexpr (std::is_constant_evaluated()) {
        for (const char *p = s; ; ++p) {
            if (*p == '\0') {
                return static_cast<std::size_t>(p - s);
            }
        }
    } else {
        __asm__("SSE 4.2 insanity");
    }
}
```

constexpr resources

- `to_chars` https://wg21.link/P2291
- `unique_ptr` https://wg21.link/P2273

containers

`flat_map` and `flat_set`

- container adapters that are alternative to `set`, `map`, `multimap`
- most interfaces are the same
- so why? It's all about different performance optimizations
- storage for keys and values is separate and does not use red-black tree
  - awesome for 'read mostly' associative containers

performance comparison to std containers

- Contiguous memory allows better cache perfomance on modern processors
- Much faster iteration
- Less memory allocations typically
- Reduced memory consumption
    - Faster lookup (possibly)
    - Slower insert and erase

hello `flat_map`

```
//...
#include <range/v3/all.hpp>
#include <boost/container/flat_map.hpp>
namespace rng = ranges::v3;
using std::cout, std::string, boost::container::flat_map;

int main()
{
  flat_map<string, int> fm;
  fm["world"] = 2;
  fm["hello"] = 1;

  //world:2 hello:1
  for ( auto [k, v]  : rng::reverse_view{fm} ) {
    cout << k << ":" << v << "\n" ;
  }
  cout << "\n";
}
```

Interface comparison summary

- iterators are random access instead of bi-directional
- iterators are invalidated on insert and erase
- value types require copyable or moveable
- direct access to internal collections
  - `extract` and `replace`
  - `.keys()` and `.values()` interface
    - incomplete types with non std containers

`flat_map` example

```
 template <class Key, class T, class Compare = less<Key>,
           class KeyContainer = vector<Key>,
           class MappedContainer = vector<T>>
  class flat_map;


vector<string> keys = { "hello", "world" };
vector<int>    vals = { 1, 2 };

//not now...
flat_map<string, int, less<string>,
         vector<string>, vector<int>> fm( keys, vals );
```

`flat_map` 'already sorted' constructor

```
//If your data is already sorted then constant time construction

flat_map(sorted_unique_t,
         key_container_type key_cont,
         mapped_container_type mapped_cont);

Effects: Initializes c.keys with std::move(key_cont)
         and c.values with std::move(mapped_cont);
         value-initializes compare.

Complexity: Constant.
```

`flat_map` Resources and Status

- in LWG review
- P0429 map/multimap - Post Kona R6 latest http://wg2
- 1222 (set/multiset) http://wg21.link/P1222
- Arthur O'Dwyer `flat_set` implementation https://gi
- `boost.flat_map` https://www.boost.org/doc/libs/1_70_0/doc/html/cont
- Matt Austern's: why not to use `std::set`, time complexity, etc http://lafstern.org/matt/col1.pdf

Heterogeneous associative container

- extension of changes to emplace in c++20
- tldr: dont convert key if it's comparible
- erase is already voted in
- rest of the interface - on the bubble

erase example

- allows for key comparison with compatible types
- example char* and `string_view` for `string` key

```
#include <string>
#include <string_view>
#include <map>

std::map<std::string, int> string_to_int { { "foo", 1}, { "bar", 2} };

const char* const k = "bar";

//c++20 construct string, 23 just compare
string_to_int.erase( k );
```

remaining functions

- `try_emplace`
- `insert` and `insert_or_assign`
- `at` and `operator[]`

heterogeneous map status and reference

- erase is done and in working draft
- no implementations so far
- erase https://wg21.link/P2077
- remaining overloads https://wg21.link/P2363

utilities

- monadic `optional`
- `expected`
- `stacktrace`
- `is_scoped_enum`
- `byte_swap`
- `to_underlying`

monadic `optional`

- adds 3 template functions to optional
  - `and_then` used to compose functions
  - `or_else` returns the optional if it has a value, otherwise it calls a given function
  - `transform` apply a function to change the value (and possibly the type)
- enables functional style of programming

usage example

```
 1: // chain a series of functions until there's an error
 2: std::optional<string> opt_string("10");
 3: std::optional<int> i = opt_string
 4:                          .and_then(std::stoi)
 5:                          .transform([](auto i) { return i * 2; });
 6:
 7: //fails, transform not called, j == nullopt
 8: std::optional<string> opt_string_bad("abcd");
 9: std::optional<int> j = opt_string_bad
10:                          .and_then(std::stoi)
11:                          .transform([](auto i) { return i * 2; });
12:
```

optional references and status

- in the working draft
- http://wg21.link/p0798
- github https://github.com/TartanLlama/optional

`expected<Return, Error>`

- an error handling type like `optional`
- adds an custom error type payload
- provides support for void returns

`expected` example

```
#include <tl/expected.hpp>
#include <boost/date_time.hpp>
#include <iostream>
#include <string>
using std::cout, std::endl;
using namespace boost::posix_time;


using time_expected = tl::expected<boost::posix_time::ptime, std::strin

time_expected from_iso_str( std::string time )
{
  try {
    ptime t  = from_iso_string( time );
    return t;
  }
  catch( std::exception& e )
  {
     return tl::make_unexpected( e.what() );
  }
}
```

`expected` example II

```
  int main()
  {
    std::string ts ( "20210726T000000" );
    auto res = from_iso_str ( ts ) ;
    if ( res ) {
       cout << to_iso_extended_string( res.value() ) << endl;
    }
    std::string bad_ts ( "F0210726T000000" );
    auto bad_res = from_iso_str ( bad_ts ) ;
    if ( !bad_res ) {
       cout << bad_res.error() << endl;
    }
  }

: 2021-07-26T00:00:00
: bad lexical cast: source type value could not be interpreted as targe
```

`expected` references and status

- last phase of committee review
- github https://github.com/TartanLlama/optional
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0323r10.

`stacktrace`

- facility for capturing stacetrace
- good for asserts, exceptions
- unfinished business from c++20
- added Nov 2020 plenary

stacktrace general

- based on `boost.stacktrace`
- header `<stacktrace>`
- stacktrace reference
  - paper http://wg21.link/p0881
  - boost http://boost.org/libs/stacktrace
- details are 'complex'
  - compiler modes still matter and not standard
  - release and optimization settings matter
  - tldr: results not portable

`stacktrace` example

```
#include <stacktrace>

// ... foo(int) called recursively
std::cout << std::stacktrace::stacktrace();

0# foo(int) at /path/to/source/file.cpp:70
1# foo(int) at /path/to/source/file.cpp:70
2# foo(int) at /path/to/source/file.cpp:70
3# foo(int) at /path/to/source/file.cpp:70
4# main at /path/to/main.cpp:93
5# __libc_start_main in /lib/x86_64-linux-gnu/libc.so.6
6# _start
```

`stacktrace` synopsis

```
class stacktrace_entry {
 public:
   string description() const;
   string source_file() const;
   uint_least32_t source_line() const;
 };

class stacktrace {  //container of entries
 public:
   begin(); //iterate over stacktrace entries
   end();
   empty();
   //...
};

string to_string(const stacktrace& st);

template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
   operator<<(basic_ostream<charT, traits>& os,
              const basic_stacktrace<Allocator>& st);
```

`is_scoped_enum`

- `is_scoped_enum` type trait
  - similar to `is_enum`
  - http://wg21.link/P1048

byte swap

- swap the byte order of an integer type

```
constexpr auto byteswap (integral auto value) noexcept;
```

https://wiki.edg.com/pub/Wg21telecons2021/LibraryW

`to_underlying`

- eliminates `static_cast` for enum to type conversions
- provides more type saftey

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1682r3.htm

interfacing with C

Support C atomics in C++

- C atomics integration
  - see `atomics.compat`
  - new header stdatomic.h
  - http://wg21.link/P0943

`out_ptr` and `inout_ptr`

- smart ptr that can be passed to C api
- `out_ptr` and `inout_ptr`
- wraps `unique_ptr` or `shared_ptr`
- expected performance on par or > hand coded C
- http://wg21.link/p1132

`out_ptr` std example - file wrapper

```cpp
//example from standard
#include <memory>
#include <cstdio>

int fopen_s(std::FILE** f, const char* name, const char* mode);

struct fclose_deleter
{
    void operator()(std::FILE* f) const noexcept
    {
      std::fclose(f);
    }
};

std::unique_ptr<std::FILE, fclose_deleter> file_ptr;
```

`out_ptr` example - file wrapper

```
int err = fopen_s(std::out_ptr<std::FILE*>(file_ptr), "some_file", "r+b

if (err != 0)
{
  throw ... //file handle is invalid
}

// *file_ptr is valid here
```

`atexit()`

"Form and Function are one" – Frank Loyld Wright

- http://crystalclearsoftware.com/2021cppcon/cpp23lib
- https://en.cppreference.com/w/cpp/compiler_support
- https://en.cppreference.com/w/cpp/utility/program/at