C++17/20/23/26 for old C++ Developer

C++14 implementation of latest STL, compatible with the old compilers and existing STL

Introduction

Since the introduction of modern C++ in 2011, the language has seen numerous enhancements, particularly in compiler support and the STL.
But are you using the latest C++ version for your project? Then you must be lucky! According to *2024 Annual C++ Developer Survey "Lite"* by ISOCPP, only 31% replied "Are you using C++20?" with yes. The response includes personal project and students, so it is likely that real-world project has much more strict version policies.

In such cases, *STL-Preview* can help you use the latest STL regardless of your C++ version and compiler.

```
namespace ranges = preview::ranges;
namespace views = preview::views;

// m = {{0, 'A'}, {1, 'B'}, {2, 'C'}, {3, 'D'}}
auto m = views::iota('A', 'E') | views::enumerate |
         ranges::to<std::map>;

// This includes views::iota introduced in C++20,
// ranges::to introduced in C++23,
// CTAD introduced in C++17 and
// pair-like to std::pair conversion introduced in C++23.
// All available in C++ 14!
```

Why STL-Preview ?

STL-Preview is not meant to be a competitor to the standard library. Its primary role is to serve as a bridge to the latest standard, bridging the gap for industries.
Users can simply change `preview::` to `std::` whenever they decide to upgrade their C++ version or compiler and use the standard library.
Because of the reasons stated above, *STL-Preview* not only focuses on implementing the latest standard but also makes it compatible with existing STL.

Other alternative standard libraries like Boost, ranges-v3 and abseil-cpp may provide more functionalities or something that *STL-Preview* doesn't provide. However, these libraries do not strictly conform to the standard and are not always compatible with existing STL.
All implementation of *STL-Preview* strictly conforms to the standard - no less, no more - which makes it a better choice when compatibility and simplicity are crucial.
Plus, there is no learning curve if you know STL already.

Features

- Provides the latest C++ standard (23 or 26) if possible
- Compatible with existing STL
- Cross-platform, standalone
- Strictly conforms the standard
- List of libraries (still work in progress)
  - concepts (30/30)
  - expected (4 / 4)
  - numbers (13 / 13)
  - optional (7 / 7)
  - span (4 / 4)
  - string_view (4 / 4)
  - variant (9 / 9)
  - iterator (57 / 59)
  - ranges (76 / 82)
  - algorithm (53/115)
  - functional (10/16)
  - memory (9 / 43)
  - type_traits (17 / 26)
  - utility (7 / 8)
  - ...

Concept

Without C++20, implementing a *real* concept is impossible. All concepts introduced in C++20 and C++23 are implemented in a *type_traits-like* struct and used in the many other libraries of *STL-Preview*.

However, **providing meaningful error messages** - one of the core functionality of the concept - is still possible before C++20. Below is an example using concept-v2. Concept-v2 is in the optimization process and will replace the existing concept in the project once completed. They defaults to *real* concept if using C++20 or custom macro is defined.

```
#define require(...) \
    typename decltype(preview::resolve_require(__VA_ARGS__))::valid = true

template<typename T, typename U, require(
    (preview::integral<T> || preview::floating_point<T>) &&
     preview::signed_integral<U>
)>
void foo(T, U) {}

foo("hello, C++", 20); // compile error!
```

```
/preview/test/concepts_v2.cc:8:1: error: no matching function for call to 'foo'
foo("hello, C++", 20);
^~~
/preview/test/concepts_v2.cc:6:6: note: candidate template ignored: substitution
failure
[with T = const char *, U = int]: no type named 'valid' in
'constraints_not_satisfied<integral<const char *>, at<0, 3>,
because<constraints_not_satisfied<integral<const char *>, at<0, 1>,
because<std::is_integral<const char *>, is_false>>>,
and_,
floating_point<const char *>, at<1, 3>,
because<constraints_not_satisfied<floating_point<const char *>, at<0, 1>,
because<std::is_floating_point<const char *>, is_false>>>
>' void foo(T, U) {}
```

* Linebreak depends on your environment

* Lexical type name will be optimized(i.e., remove `at<0, 1>` and nested `constraints_not_staisfied`)

How concept-v2 works

```
// Simple concept of the mechanism. Actual implementation is more complex.

template<typename Constraints, typename... Information>
struct constraints_not_satisfied : std::false_type {};

// Good-old CRTP
template<typename Derived, typename Base>
struct concept_base : Base {
  /* operator&&, operator|| and operator! are defined, which returns
   * True<N> or constraints_not_satisfied<Ci, at<i, N>, because<...>>
   */
};

// Define a basic concept
template<typename T>
struct integral_c : concept_base<integral_t<T>, std::is_integral<T>> {};
template<typename T>
inline constexpr integral_c<T> integral;

// Same if-else-endif macro
// Define a nested concept
template<typename T>
struct signed_integral_c : concept_base<signed_integral_c<T>, decltype(
    integral<T> && std::is_signed<T>{}
)> {};
template<typename T>
inline constexpr signed_integral_c<T> signed_integral;

// Equal to True<2> which inherits std::true_type
static_assert(integral<int> || integral<int>);

// Equal to
// constraints_not_satisfied<signed_integral<float>, at<1, 2>, because<...>>
// which inherits std::false_type
static_assert(integral<int> && signed_integral<float>);
```

Contribution / Future

This is an open-source project and is still in development. All types of contributions are welcomed! Feel free to criticize, report a bug, open a PR or just give a like :)
The goal of this year is to release C++20 except for concurrency libraries.

rs: STL-Preview

Tony Lee | cosgenio@gmail.com

github.com/lackhole/stl-preview

Cppcon The C++ Conference | 2024 September 15-20 Aurora, Colorado, USA

linkedin.com/in/yonggyu-lee/

Iterator

Iterator is one of the core libraries of STL, categorizing iterators using `std::iterator_traits`. However, `std::iterator_traits` is not SFINAE-friendly until C++17 and conditionally SFINAE-friendly until C++20.

For *STL-Preview* to be compatible with pre-C++20 STLs, all post-C++20 iterators must define all five typedefs. This also includes iterators of views(e.g., `preview::views::iota_view::iterator`).

*Note: typedefs not defined in the standard are removed if using C++20 or later

The opposite case - using pre-C++20 STLs with *STL-Preview* - can be handled easily. Thankfully, the typedefs of C++20 iterator_traits doesn't directly rely on iterator's typedefs, so specializing preview::incrementable_traits is enough for old iterators(e.g., `std::back_insert_iterator`, `std::ostream_iterator`).

STL-Preview defines its iterator_traits, so it is self-consistent(of course).

Limitations

`contiguous_iterator`

Checking if given `random_access_iterator` also models `contiguous_iterator` is not always 100% accurate without `std::contiguous_iterator_tag` which was introduced in C++20(i.e, `std::vector<bool>::iterator`, `std::deque::iterator`). All iterators defined in `std` are manually checked, but user-provided iterators may produce false negative results. Thus, ranges library that provides a specialized algorithm(e.g., `preview::ranges::copy_n`) doesn't trust `preview::contiguous_iterator` if the tag is not defined in `std` unless it is a raw pointer.

`preview::contiguous_range` is implemented without a false behavior.

Detecting if `iterator_traits<I>` is primary template

The Core functionality of the iterator library also depends on whether `iterator_traits<I>` is a primary template(e.g., `iter_difference_t`, `iter_value_t`, *`ITER_CONCEPT`*). All `iterator_traits<I>` defined in `std` are manually checked, but user-specializations cannot be checked(although some STL provides a non-portable way). Hope `std::is_primary_template` will be proposed in the future...

But hey, do you specialize `std::iterator_traits` for your iterator?

Constraints on specialization of `iterator_traits`

A `requires` clause can also set constraints on template specializations. Specializations are selected only if the constraints are satisfied; otherwise, the primary template is used as a fallback.

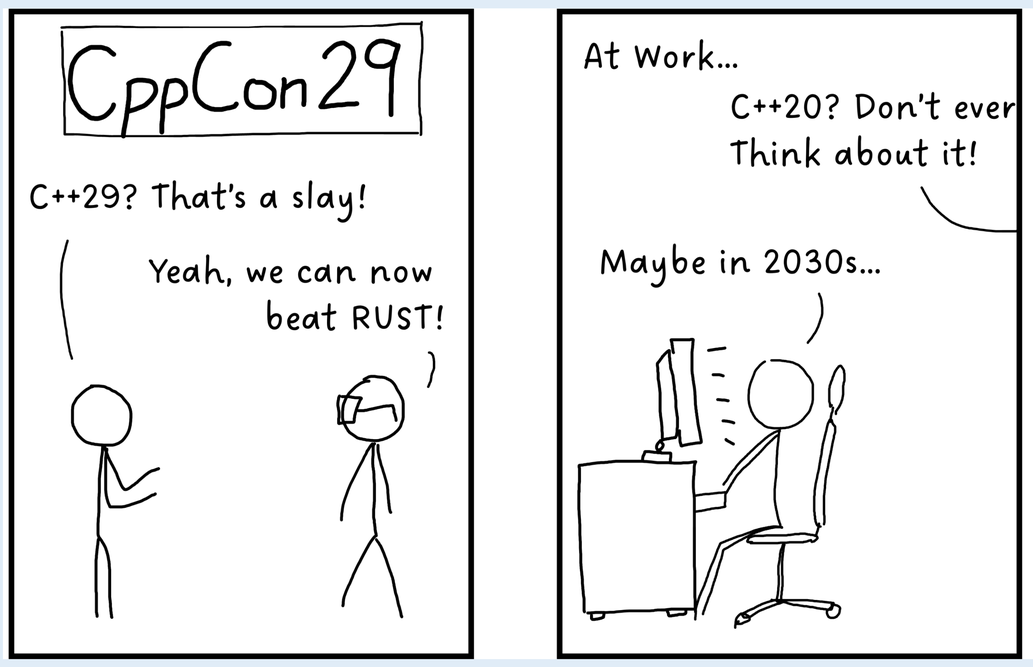
Branching typedefs are insufficient in this case, as the reasons stated above section. This complex situation cannot be handled correctly before C++20.

This limitation only applies when mixed usage of STL-preview and existing STL before C++20.

Supported(Tested) Compilers

	Minimum version tested	Maximum version tested
MSVC	19.29.30154.0 (Visual Studio 2019)	19.40.33811.0 (Visual Studio 2022)
GCC	9.5.0	13.1.0
Clang	11.1.0	15.0.7
Apple Clang	14.0.0.14000029	15.0.0.15000040
Android NDK	r18 (Clang 7.0)	r26 (Clang 17.0.2)
Emscripten	3.1.20 (Clang 16.0)	3.1.61 (Clang 19.0.0)
MinGW	13.1.0	14.2.0
Intel C++	Not tested yet	2024.2.1

Table: tested compilers (versions not listed here may work)



Ranges

Although ranges library is one of the main implementation targets, only 93% of <ranges> and 46% of <algorithm> have been implemented so far due to its extensive amount(~140 / ~200).
Ranges are determined by ADL-or-member functions(e.g., `begin`, `end`), it is compatible with STL seamlessly (if STL is implemented correctly!).

```cpp
// Mixing C++20 STL and C++23 STL-Preview
auto m = std::views::iota('A', 'E')
       | preview::views::enumerate
       | preview::ranges::to<std::map>;
```

`preview::ranges::to` does unspecified conversion operations, so it is equipped with C++23 conversions if possible(i.e., *pair-like* to `std::pair`).
While evaluating CTAD(limited CTAD in C++14), `std::tuple` to `std::pair` is found to be valid thus deduced to `std::map<int, char>`, with additional `preview::views::transform` layer if the conversion is not provided by STL(Clang provides this conversion since C++11)

Utility Libraries

Following utility libraries are implemented so far. Although C++26 standard is not complete, these libraries are being updated if announced.

- bind_front, bind_back
- optional
  - Monadic operations
- span
  - C++23/26 operations
- string_view
  - Member search
- variant
  - Member visit
- rel_ops
  - C++20 operator synthesis
  - All comparisons in *STL-Preview* relies on this
- expected

Concurrency Libraries

All concurrency libraries introduced after C++20 rely on atomic wait/notify operations, but implementing these operations compatible with std::atomic is impossible(i.e., gcc defines atomic implementation as private) without performance loss. Implementing the entire atomic is required, which is far from the purpose of *STL-Preview*. Maybe a wrapper of Boost.Atomic can be a solution, but haven't decided yet.