C++26 for C++14 Developers: STL-Preview

Author

Tony Lee

Reply-To

cosgenio@gmail.com

1. Introduction

Since the introduction of modern C++ in 2011, the language has seen numerous enhancements, particularly in compiler support and the STL. By 2024, C++23 has introduced many useful standard libraries like ranges, format, expected, and span. Despite these advancements, many industries still rely on C++14 or C++17, with minimal adoption of C++20 or later[1]. This paper introduces *stl-preview*[2], a library that backports most of the latest STL features (including those from C++26) to C++14 (but not limited to), ensuring compatibility with existing STL implementations and bridging the gap for industries yet to upgrade their compilers.

2. Relevance

stl-preview currently implements 281 out of 690 STL functions and classes introduced between C++17 and C++26 in C++14. This includes `ranges`, `concepts`, `span`, and `expected`. It has been tested on various compilers including GCC, Clang, MSVC, Emscripten, and NDK with C++14 or later.

Besides the extensive implementations, *stl-preview* is compatible with the existing STL provided by the user's compiler. Alternatives like Boost or original libraries such as ranges-v3 or format do not strictly conform to the standard and are not fully compatible with existing STLs. Other STL libraries, such as Microsoft-STL or libc++, are heavy, bound to their compiler, and not compatible with existing STLs, making them challenging to use for maintaining cross-platform C++ projects that use two or more compilers.

By backporting the latest (C++26) STL features to older standards (C++14), it allows developers to leverage modern C++ functionalities without modifying their compilers or breaking existing codebases. *stl-preview* works as a bridge, so the user can simply change the namespace from `preview` to `std` whenever they decide to use the STL provided by the compiler.

3. Discussion

Although *stl-preview* can replace existing STL, its primary role is to serve as a bridge to the latest standard. Some core functionalities like coroutines or reflections cannot be implemented without compiler support, so *stl-preview* cannot 100% replace STL. Managing compatibility with existing STL is crucial. Major challenges

include handling the latest C++ semantics and conversions, compatibility with pre-C++20 iterators, and addressing defects of old compilers.

3-1. Latest C++ Semantics and Conversions

`requires` clauses are implemented using classic template metaprogramming, and constraints on class template arguments are replaced with `static_assert` if possible. Comparison operators are synthesized under the namespace `preview::rel_ops` to provide a C++20 experience even if user-defined class provides single comparison operator in C++14.

One of the trickiest parts was providing the latest conversion between STLs such as *pair-like* to `std::pair`, which was introduced in C++23 (although Clang already provides them from the past) because *stl-preview* cannot modify the existing STL. So implementations that perform unspecified conversions (i.e., `preview::ranges::to`) are equipped with C++23 conversions, so the following code compiles in C++14:

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

views::iota('A', 'E') | views::enumerate | ranges::to<std::map>();
// While evaluating CTAD, std::tuple to std::pair is found to be valid thus
// deduced to std::map<int, char> with additional views::transform layer if
// the conversion is not provided by the existing STL
```

3-2. Compatibility with pre-C++20 iterator

`std::iterator_traits` is not SFINAE-friendly until C++17 and requires all five typedefs to be defined until C++20[3]. To achieve C++20 behavior without modifying `std::iterator_traits`, iterators must define all five (or six if `iterator_concept` is included) typedefs or use a special placeholder tag for not-defined typedefs. This is necessary for post-C++20 iterators like `preview::ranges::iota_view::iterator`[4] when used with pre-C++20 STL, which relies on `std::iterator_traits`.

The opposite situation can be handled by specializing `preview::incrementable_traits` with pre-C++20 iterators[5] such as `std::back_insert_iterator`, which are used to determine `preview::iterator_traits::difference_type`. Since *stl-preview* follows C++26 standard, it is self-consistent.

Before C++17, where `std::contiguous_iterator_tag` did not exist, it was impossible to detect if an iterator modeled `contiguous_iterator`. As a workaround, `LegacyContiguousIterator`[6] and detecting iterators of STL containers like `std::vector<bool>::iterator` and `std::deque::iterator` are manually performed if `std::contiguous_iterator_tag` is not defined in STL. `contiguous_range` can be implemented without `std::contiguous_iterator_tag`[7].

Detecting if `std::iterator_traits` is a primary template is required for iterators like `preview::counted_iterator`, but there is no standard way to detect if a type is primary or not. Thus, detecting every specialization of `std::iterator_traits` for STL iterators is performed as a workaround.

3-3. Handle Defects of Old Compilers

Every implementation undergoes automated unit tests using GitHub Actions, tested with every combination of compilers and C++ standards possible. Some confronted defects include allowing ambiguous base casting, infinite TMP evaluation loops when evaluating some ranges functions, and operator synthesis. *stl-preview* has been tested on various compilers including:

- MSVC: 19.29.30154.0 (Visual Studio 16 2019) to 19.40.33811.0 (Visual Studio 17 2022)
- GCC: 9.5.0 to 13.1.0
- Clang: 11.1.0 to 15.0.7
- Apple Clang: 14.0.0.14000029 to 15.0.0.15000040
- Android NDK: r18 (Clang 7.0) to r26 (Clang 17.0.2)
- Emscripten: 3.1.20 (Clang 16.0.0) to latest (3.1.61) (Clang 19.0.0)

4. Completion Status

Currently, around 300 out of approximately 700 functions and classes introduced between C++17 and C++26 are implemented. Key headers like `concepts`, `expected`, `iterator`, `optional`, `ranges`, `span`, `string_view`, `utility`, and `variant` are nearly fully implemented. For C++20, 206 out of 428 (48%) features have been implemented so far. By the time of the poster presentation at CppCon 2024, it is expected that at least 90% of the C++20 STL will be implemented. Additional testing for Intel C++ and MinGW is also planned. Detailed implementation status can be found on GitHub.

5. Supporting Material

stl-preview can be found in GitHub under BSD-3 license: https://github.com/lackhole/stl-preview

References:

---

[1] https://isocpp.org/files/papers/CppDevSurvey-2024-summary.pdf

[2] https://github.com/lackhole/stl-preview

[3] https://en.cppreference.com/w/cpp/iterator/iterator_traits#Member_types

[4] https://github.com/lackhole/stl-preview/blob/main/include/preview/__ranges/views/iota_view.h

[5] https://github.com/lackhole/stl-preview/blob/main/include/preview/__iterator/incrementable_traits.h

[6] https://en.cppreference.com/w/cpp/named_req/ContiguousIterator

[7] https://github.com/lackhole/stl-preview/blob/main/include/preview/__ranges/contiguous_range.h