std::at : Range-checked accesses to arbitrary containers

Document #: P3404R0
Date: 2024-09-21
Project: Programming Language C++
Audience: LEWG, SG23 (Safety and Security), SG9 (Ranges)

Reply-to: Andre Kostur
<andre@kostur.net>

Contents

1 Abstract

This proposal is to add a customization point object for `std::at()` which will forward to containers which have a member function `at()`. In addition this will also apply to arrays.

2 Motivation

Lately there has been a stronger push towards code safety, in particular performing range-checked accesses to containers. A number of existing containers have both an `operator[]` for unchecked accesses, as well as an `at()` member function for range-checked accesses which may throw an `std::out_of_range` exception. It would be useful to be able to have an `at()` for user-declared types which may not have provided their own as well as having range-checked accesses to arrays. This facilitates generic programming with containers in the same manner that `std::begin()` does.

3 Revision History

3.1 R0

Initial revision.

4 Design

Define a customization point object `std::at()` which may be used instead of calling a member-function `at()`.

Since C++23 `operator[]` can take any number of subscripts, it is reasonable to anticipate that user-defined classes will start to have a member function `at()` taking multiple subscripts. `std::mdspan` and `std::mdarray` do support multple subscripts, but do not yet have an `at()` member function. We should anticipate the addition of `std::mdspan::at()` and `std::mdarray::at()` that will have similar function signatures as `std::mdspan::operator[]` and `std::mdarray::operator[]`.

There are three cases that this proposal needs to consider:

1. Containers which have a member-function `at()`. `std::at()` will call the member function, forwarding all of the additional arguments.
2. Arrays cannot have member functions. `std::at()` will test the subscript to verify that it is in range, and will throw `std::out_of_range` when the range is violated.
3. User declared free-function `at()`, presumably in the same namespace as the container so as to take advantage of ADL to choose this function over the templated function that will be provided in the standard library.

As `at()` functions use exceptions as the error reporting mechanism, this is not applicable to freestanding and should be explicitly marked as freestanding-deleted, just as `std::span::at()` is.

5 Feature Test Macro

```
#define __cpp_lib_at xxxxxxL
```

6 Example implementation

These are only for demonstration purposes and are not intended to dictate implementation.

6.1 Case 1: Containers with `at()` member function(s)

```
template <typename _Tp, typename... _Idxs>
constexpr decltype(auto) at(_Tp& coll, _Idxs&&... idxs)
    requires requires(_Tp coll, _Idxs&&... idxs)
        { std::forward<_Tp>(coll).at(std::forward<_Idxs>(idxs)...); }
{
    return std::forward<_Tp>(coll).at(std::forward<_Idxs>(idxs)...);
}
```

6.2 Case 2: Arrays

```
template <typename _Tp, size_t N>
constexpr auto at(_Tp (&array)[N], size_t idx) -> _Tp&
{
    if (N <= idx)
    {
        throw std::out_of_range("generic at");
    }

    return array[idx];
}
```

6.3 Case 3: Custom overload

There is no example implementation as this is user-supplied.

7 Questions

1. Are we about to cause problems by introducing a `std::at` identifier like we did when we added `std::byte`? We should be able to introduce new identifiers in std:: namespace without worrying about this as it would greatly hobble the committee's ability to add new things. Additionaly the problem with byte was that for at least one platform, they had defined a macro "byte" which caused the issue, and they resolved that issue.
2. Do we want to have overloads to deal with containers that do not have an `at()` member function, but does have a single-subscript `operator[]` and something that `std::size()` would work on? This could allow `std::at()` to automatically range-check these containers as well. A concern I have is whether that container is guaranteed to start its subscript at 0, or whether it is even an integral subscript.
3. Do we want to acknowledge potentially multi-dimensional subscripting now, or should we defer that to a future paper when there is more demand?

8 Acknowledgements

Thank-you to Herb Sutter for the initial inspiration.