Doc No.: P2568R0
Audience: LEWG
Project: Programming Language - C++ (WG21)
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2022-03-18

Proposal of std::map::at_ptr

Summary

We propose the addition of a new member access function, called `.at_ptr`, to the standard containers that currently provide `.at`.

`.at_ptr` is the same as `.at`, but for the following differences:

- `.at_ptr` returns a pointer, whereas `.at` returns a reference.
- when the entry is not found, `.at_ptr` returns `nullptr`, whereas `.at` throws

The relationship between `.at` and `.at_ptr` is similar to that between `dynamic_cast<T&>` and `dynamic_cast<T*>`.

Example

```
std::map<int, int> m = /*...*/;

if (int* v = m.at_ptr(42))
    /* do something with *v */
else
    /* do something else */
```

Motivation

It is a common operation in a wide range of programs and algorithms to lookup a key on a container and do something with the corresponding value if present, or do something else if it is not present.

Consider some kind of search function that has a precalculated cache of common queries. At the start of the search function you want to look up the cache and return the precalculated value if present, otherwise you conduct the full search.

There is extensive existing practice of user functions in various core libraries with the proposed semantics. We propose standardizing that existing practice.

Design

Should this be a free function or member function?

We think `.at_ptr` belongs right next to `.at` as a "member access" member function, given the similarities between them.

What should this be called?

Names that were considered included:

```
.get_if
.at_ptr
.ptr_at
.at_if
.if_at
.pat
.try_at
.at_or_null
.present
.lookup
```

We also considered making it a pointer overload of `.at` but of course this doesn't work with rvalues. (ie `m.at(&42)` won't compile.)

We feel that `m.at_ptr(k)` is the next best thing to `m.at(&k)` - essentially just moving the `&` to a `_ptr` suffix on the `at` identifier.

Which containers should provide this?

We think all the containers that currently provide .at is the right subset. In particular that means std::string, std::string_view, std::array, std::vector, std::map, std::unordered_map and std::deque.

What about returning std::optional<T&> instead of pointer?

WG21 has decided that std::optional shall not support reference types, so this is not feasible.

Implementation

The implementations of `.at_ptr` are essentially identical to that of `.at`, but rather than throwing it `return nullptr`, and of course it has `return &expr` rather than `return expr` - and if possible it's marked `noexcept`

Here is the implementation in libstdc++ for std::map:

```
mapped_type*
at_ptr(const key_type& __k)
{
  iterator __i = lower_bound(__k);
  if (__i == end() || key_comp()(__k, (*__i).first))
    return nullptr;
  return &((*__i).second);
}

const mapped_type*
at_ptr(const key_type& __k) const
{
  const_iterator __i = lower_bound(__k);
  if (__i == end() || key_comp()(__k, (*__i).first))
    return nullptr;
  return &((*__i).second);
}
```

And here is the implementation in libstdc++ for std::vector:

```
_GLIBCXX20_CONSTEXPR
pointer
at_ptr(size_type __n) _GLIBCXX_NOEXCEPT
{
  if (__n >= this->size())
    return nullptr;

  return &((*this)[__n]);
}

_GLIBCXX20_CONSTEXPR
const_pointer
at_ptr(size_type __n) const _GLIBCXX_NOEXCEPT
{
  if (__n >= this->size())
```

```
          return nullptr;

        return &((*this)[__n]);
      }
```

Specification

[string.view.template.general]

```
    // [string.view.access], element access
    constexpr const_reference operator[](size_type pos) const;
    constexpr const_reference at(size_type pos) const;
    constexpr const_pointer at_ptr(size_type pos) const noexcept;
    constexpr const_reference front() const;
    constexpr const_reference back() const;
    constexpr const_pointer data() const noexcept;
```

[string.view.access]

```
constexpr const_pointer at_ptr(size_type pos) const noexcept;
```

Returns: `data_ + pos` if `pos < size()`, otherwise returns `nullptr`

[basic.string.general]

```
    // [string.access], element access
    constexpr const_reference operator[](size_type pos) const;
    constexpr reference       operator[](size_type pos);
    constexpr const_reference at(size_type n) const;
    constexpr reference       at(size_type n);
    constexpr const_pointer   at_ptr(size_type n) const noexcept;
    constexpr pointer         at_ptr(size_type n) noexcept;
    constexpr const_reference front() const;
    constexpr const_reference back() const;
    constexpr const_pointer data() const noexcept;
```

[string.access]

```
constexpr const_pointer   at_ptr(size_type n) const noexcept;
constexpr pointer         at_ptr(size_type n) noexcept;
```

Returns: `&(*(begin()+pos))` if pos < size(), otherwise returns `nullptr`.

[sequence.reqmts]

`a.at_ptr(n)`

 Result: `pointer`; `const_pointer` for constant `a`

 Returns: `&(*(a.begin() + n))` if `n < a.size()`, otherwise `nullptr`

 Remarks: Required for `basic_string`, `array`, `deque`, and `vector`.

[array.overview]

```
    // element access
    constexpr reference       operator[](size_type n);
    constexpr const_reference operator[](size_type n) const;
    constexpr reference       at(size_type n);
    constexpr const_reference at(size_type n) const;
    constexpr pointer         at_ptr(size_type n);
    constexpr const_pointer   at_ptr(size_type n) const;
    constexpr reference       front();
    constexpr const_reference front() const;
    constexpr reference       back();
    constexpr const_reference back() const;
```

[vector.overview]

```
    // element access
    constexpr reference       operator[](size_type n);
    constexpr const_reference operator[](size_type n) const;
    constexpr const_reference at(size_type n) const;
    constexpr reference       at(size_type n);
    constexpr const_pointer   at_ptr(size_type n) const noexcept;
    constexpr pointer         at_ptr(size_type n) noexcept;
    constexpr reference       front();
    constexpr const_reference front() const;
    constexpr reference       back();
    constexpr const_reference back() const;
```

[vector.bool]

```
    // element access
    constexpr reference       operator[](size_type n);
    constexpr const_reference operator[](size_type n) const;
    constexpr const_reference at(size_type n) const;
    constexpr reference       at(size_type n);
    constexpr const_pointer   at_ptr(size_type n) const noexcept;
    constexpr pointer         at_ptr(size_type n) noexcept;
    constexpr reference       front();
    constexpr const_reference front() const;
    constexpr reference       back();
    constexpr const_reference back() const;
```

[deque.overview]

```
    // element access
    reference       operator[](size_type n);
    const_reference operator[](size_type n) const;
    reference       at(size_type n);
    const_reference at(size_type n) const;
    pointer         at_ptr(size_type n) noexcept;
    const_pointer   at_ptr(size_type n) const noexcept;
    reference       front();
    const_reference front() const;
    reference       back();
    const_reference back() const;
```

[map.overview]

```
    // [map.access], element access
    mapped_type& operator[](const key_type& x);
    mapped_type& operator[](key_type&& x);
    mapped_type&       at(const key_type& x);
    const mapped_type& at(const key_type& x) const;
    mapped_type*       at_ptr(const key_type& x);
    const mapped_type* at_ptr(const key_type& x) const;
```

[map.access]

```
mapped_type*       at_ptr(const key_type& x);
const mapped_type* at_ptr(const key_type& x) const;
```

Returns: A pointer to the `mapped_type` corresponding to `x` in `*this`, or `nullptr` if no such element is present.

Complexity: Logarithmic.

[unord.map.overview]

```
    // [unord.map.elem], element access
    mapped_type& operator[](const key_type& k);
    mapped_type& operator[](key_type&& k);
    mapped_type& at(const key_type& k);
    const mapped_type& at(const key_type& k) const;
    mapped_type* at_ptr(const key_type& k);
    const mapped_type* at_ptr(const key_type& k) const;
```

[unord.map.elem]

```
mapped_type* at_ptr(const key_type& k);
const mapped_type* at_ptr(const key_type& k) const;
```

Returns: A pointer to `x.second`, where `x` is the (unique) element whose key is equivalent to k, or `nullptr` if no such element is present.

References

Initial lib-ext discussion thread: https://lists.isocpp.org/lib-ext/2022/02/22652.php