CS 3460

Introduction to Iterators
(more generic programming)

What is an iterator?

- Let's look at a couple of code examples…

```
std::array<int, 5> primes{ 2, 3, 5, 7, 11 };
for (std::size_t i{ 0 }; i < primes.size(); i++)
{
    std::cout << primes[i] << std::endl;
}
```

```
for (auto i{ primes.begin() }; i != primes.end(); ++i)
{
    std::cout << *i << std::endl;
}
```

- *Iteration* means to repeat a process
- In the first example, `i` is an iterator in the general sense, but not specifically what we want to discuss
- In the second example, `i` is also an iterator
  - The next slide breaks down the second example

C++ Iterator

```
std::array<int, 5> primes{ 2, 3, 5, 7, 11 };

for (auto i{ primes.begin() }; i != primes.end(); ++i)
{
    std::cout << *i << std::endl;
}
```

- `.begin()` returns an iterator; an object (instance of a class)
- The type of `i` is: `std::array<int, 5Ui64>::iterator`
  - `std` is the standard library namespace
  - `array<int, 5Ui64>` is the type of the array
    - `int` for the data type
    - 64-bit `unsigned int` for the size type
  - `iterator` is a nested class, inside of `std::array`
    - We'll take a detailed look at this soon
- `.end()` returns a value that is *after the last element*
- Notice the `++` increment operator to update the iterator
- Access to the value is by dereferencing the iterator (iterators are *pointer-like*)

C++ Iterator – What Else

- If an iterator satisfies the proper requirements, can use in a range-based for loop

```
std::array<int, 5> primes{ 2, 3, 5, 7, 11 };

for (auto i : primes)
{
    std::cout << i << std::endl;
}
```

```
std::array<int, 5> primes{ 2, 3, 5, 7, 11 };

for (auto&& i : primes)
{
    std::cout << i << std::endl;
}
```

Iterator Types

- (legacy)InputIterator
- (legacy)OutputIterator
- (legacy)ForwardIterator
- (legacy)BidirectionalIterator
- (legacy)RandomAccessIterator
- (legacy)ContinguousIterator

- Iterator Reference: https://en.cppreference.com/w/cpp/iterator

Again, What is an Iterator?

- A description of requirements
  - Defined by the methods it implements
    - Pre C++ 20: not a way to enforce/validate these
    - Post C++ 20: validated through *concepts*
  - Possible to write an *incomplete* iterator, that meets your specific needs
  - Should base an iterator on one of the six iterator types
- No inheritance hierarchy defining abstract types/methods

Iterator Requirements

- copy constructable
- copy assignable
- destructable
- l-values must be swappable
- dereferenceable
- incrementable

- Member typedefs/using
  - `iterator_category`
  - `value_type`
  - `difference_type`
  - `reference`
  - `pointer`

Implementing an Iterator – By Example

- Let’s make a custom container, an array

- No conflict with `std::array`
  - `usu::array`
- Templated on type
- Size specified at compile time
- Default constructor
- Initializer list constructor

- Array-like access `[ ]`
- Expose a `.size` method
- Exposes `.begin`/`.end` methods
- (legacy)RandomContiguousIterator

`usu::array` – Type Declaration

- The following is the array type, without the `iterator` class

```
namespace usu
{
    template <typename T, unsigned int N>
    class array
    {
      public:
        array() = default;
        array(const std::initializer_list<T>& list);

        reference operator[](unsigned int index);
        size_type size() { return N; }

        iterator begin() { return iterator(m_data); }
        iterator end() { return iterator(N, m_data); }

      private:
        T m_data[N];
    };
}
```

`usu::array` – Type Declaration

- The following is the array type, without the `iterator` class
  - Type is `T`
  - Size is a non-template parameter `N`

```
namespace usu
{
    template <typename T, unsigned int N>
    class array
    {
      public:
        array() = default;
        array(const std::initializer_list<T>& list);

        reference operator[](unsigned int index);
        size_type size() { return N; }

        iterator begin() { return iterator(m_data); }
        iterator end() { return iterator(N, m_data); }

      private:
        T m_data[N];
    };
}
```

`usu::array` – Type Declaration

- The following is the array type, without the `iterator` class
  - We accept the default constructor

```
namespace usu
{
    template <typename T, unsigned int N>
    class array
    {
      public:
        array() = default;
        array(const std::initializer_list<T>& list);

        reference operator[](unsigned int index);
        size_type size() { return N; }

        iterator begin() { return iterator(m_data); }
        iterator end() { return iterator(N, m_data); }

      private:
        T m_data[N];
    };
}
```

`usu::array` – Type Aliases

- The container class needs to define some types
  - Some developers do this, others don’t (I recommend it)
  - No universally accepted approach
  - We’ll follow the standard library convention

`usu::array` – Type Aliases

- The container class needs to define some types
  - Some developers do this, others don't (I recommend it)
  - No universally accepted approach
  - We'll follow the standard library convention

```
template <typename T, unsigned int N>
class array
{
  public:
    using value_type = T;
    using size_type = std::size_t;
    using pointer = T*;
    using reference = T&;
```

`usu::array` – Type Declaration

- The following is the array type, without the `iterator` class

```
namespace usu
{
    template <typename T, unsigned int N>
    class array
    {
      public:
        array() = default;
        array(const std::initializer_list<T>& list);

        reference operator[](unsigned int index);
        size_type size() { return N; }

        iterator begin() { return iterator(m_data); }
        iterator end() { return iterator(N, m_data); }

      private:
        T m_data[N];
    };
}
```

`usu::array` – Initializer List Constructor

- Copy values from `std::initializer_list` into internal array storage

```
template <typename T, unsigned int N>
array<T, N>::array(const std::initializer_list<T>& list)
{
    if (list.size() > N)
    {
        throw new std::exception("Initializer list contains too many elements");
    }

    for (size_type pos{ 0 }; auto && value : list)
    {
        m_data[pos++] = value;
    }
}
```

usu::array – [ ] Operator

- Return a `reference` to the value

```
template <typename T, unsigned int N>
typename array<T, N>::reference array<T, N>::operator[](unsigned int index)
{
    if (index < 0 || index >= N)
    {
        throw new std::exception("Index out of bounds");
    }
    return m_data[index];
}
```

Iterator Type – ForwardIterator Requirements

- Default, Overloaded, Copy, and Move constructors
- Copy and Move assignment operators
- Prefix/Postfix increment operators
- Dereference operator
- Equality and inequality relational operators

- Things we need, in addition to the requirements
  - Current position of the iterator in the container
  - A pointer to the array data from the parent class

Iterator Type – Iterator Category

- Need to declare the iterator category

```
template <typename T, unsigned int N>
class array
{
  public:
    . . .

    class iterator
    {
      public:
         using iterator_category = std::forward_iterator_tag;
```

Iterator Type – Member Data

- Add in the position and pointer to the data

```
template <typename T, unsigned int N>
class array
{
  public:
    . . .

    class iterator
    {
      public:
         using iterator_category = std::forward_iterator_tag;

      private:
         size_type m_pos;
         pointer m_data;
```

Iterator Type – Constructors

- Default and Overloaded Constructors

```
iterator() :
    iterator(nullptr)  // DefaultConstructable
{
}
iterator(pointer ptr) :
    m_data(ptr),
    m_pos(0)
{
}
iterator(size_type pos, pointer ptr) :
    m_pos(pos),
    m_data(ptr)
{
}
```

Iterator Type – Copy/Move Constructors

- Declarations

```
template <typename T, unsigned int N>
class array
{
  public:
    . . .

    class iterator
    {
      public:
        . . .
        iterator(const iterator& obj);      // CopyConstructable
        iterator(iterator&& obj) noexcept; // MoveConstructable
```

Iterator Type – Copy/Move Constructors

- Definitions

```
template <typename T, unsigned int N>
array<T, N>::iterator::iterator(const iterator& obj)
{
     this->m_pos = obj.m_pos;
     this->m_data = obj.m_data;
}

template <typename T, unsigned int N>
array<T, N>::iterator::iterator(iterator&& obj) noexcept
{
     this->m_pos = obj.m_pos;
     this->m_data = obj.m_data;

     obj.m_pos = 0;
     obj.m_data = nullptr;
}
```

Iterator Type – Prefix/Postfix Increment

- Declarations

```
template <typename T, unsigned int N>
class array
{
  public:
    . . .

    class iterator
    {
      public:
        . . .
        iterator operator++();    // incrementable e.g., ++r
        iterator operator++(int); // incrementable e.g., r++
```

Iterator Type – Prefix/Postfix Increment

- Definitions

```
template <typename T, unsigned int N>
typename array<T, N>::iterator array<T, N>::iterator::operator++()
{
     m_pos++;
     return *this;
}

template <typename T, unsigned int N>
typename array<T, N>::iterator array<T, N>::iterator::operator++(int)
{
     iterator i = *this;
     m_pos++;
     return i;
}
```

notice...returning a copy

Iterator Type – Copy/Move Assignment

- Declarations

```
template <typename T, unsigned int N>
class array
{
  public:
    . . .

    class iterator
    {
      public:
        . . .
        iterator& operator=(const iterator& rhs); // CopyAssignable
        iterator& operator=(iterator&& rhs);      // MoveAssignable
```

Iterator Type – Copy/Move Assignment

- Definitions

```
template <typename T, unsigned int N>
typename array<T, N>::iterator& array<T, N>::iterator::operator=(const iterator& rhs)
{
    this->m_pos = rhs.m_pos;
    this->m_data = rhs.m_data;

    return *this;
}

template <typename T, unsigned int N>
typename array<T, N>::iterator& array<T, N>::iterator::operator=(iterator&& rhs)
{
    if (this != &rhs)
    {
        std::swap(this->m_pos, rhs.m_pos);
        std::swap(this->m_data, rhs.m_data);
    }

    return *this;
}
```

Iterator Type – Dereference Operator

```
template <typename T, unsigned int N>
class array
{
  public:
    . . .

    class iterator
    {
      public:
        . . .
        reference operator*()      // Derefrenceable
        {
            return m_data[m_pos];
        }
```

Iterator Type – Relational Operators

- We compare based on position

```
bool operator==(const iterator& rhs) { return m_pos == rhs.m_pos; }
bool operator!=(const iterator& rhs) { return m_pos != rhs.m_pos; }
```

Whew!
Let’s do a code demonstration