# COMP6771
# Advanced C++ Programming

## Week 7.2
## Custom Iterators

# In this lecture

**Why?**

- When we define our own types, if we want them to be iterable we need to define that functionality ourselves.

**What?**

- Custom Iterators → Kinda like operator overloading.
- Iterator Invalidation
- Iterator Types

# Iterator revision

- Iterator is an abstract notion of a **pointer**
- Iterators are types that abstract container data as a sequence of objects
  - The glue between containers and algorithms
    - Designers of algorithms don't care about details about data structures
    - Designers of data structures don't have to provide extensive access operations

```
std::vector v{1, 2, 3, 4, 5};
++(*v.begin()); // vector<int>'s non-const iterator
*v.begin(); // vector<int>'s const iterator
v.cbegin(); // vector<int>'s const iterator
```

# Iterator invalidation

- Iterator is an abstract notion of a **pointer**
- What happens when we modify the container?
  - What happens to iterators?
  - What happens to references to elements?
- Using an invalid iterator is undefined behaviour

```
std::vector v{1, 2, 3, 4, 5};
// Copy all 2s
for (auto it = v.begin(); it != v.end(); ++it) {
  if (*it == 2) {
    v.push_back(2);
  }
}
// Erase all 2s
for (auto it = v.begin(); it != v.end(); ++it) {
  if (*it == 2) {
    v.erase(it);
  }
}
```

infinite loop

永远达不到

add break to fix

iter已经[空], 永远达不到

在 end 之后.

i

1 2 3 4 5 2 e

1 3 4 5 2 e

1 3 4 2 e

# Iterator invalidation - push_back

eg:

std::cout << &v[0];

v.push_back(6);

std::cout << &v[6];

- Think about the way a vector is stored
- "If the new size() is greater than capacity() then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated."

different size!!!

different address.

All iters invalidated.

```
std::vector v{1, 2, 3, 4, 5};
// Copy all 2s
for (auto it = v.begin(); it != v.end(); ++it) {
  if (*it == 2) {
    v.push_back(2);
  }
}
```

https://en.cppreference.com/w/cpp/container/vector/push_back

# Iterator invalidation - erase

- "Invalidates iterators and references at or after the point of the erase, including the end() iterator."
- For this reason, erase returns a new iterator

```
std::vector v{1, 2, 3, 4, 5};
// Erase all even numbers (C++11 and later)
for (auto it = v.begin(); it != v.end(); ) {
  if (*it % 2 == 0) {
    it = v.erase(it);
  } else {
    ++it;
  }
}
```

https://en.cppreference.com/w/cpp/container/vector/erase

# Iterator invalidation - general

- Containers generally don't invalidate when you modify values
- But they may invalidate when removing or adding elements
- std::vector invalidates everything when adding elements
- std::unordered_(map/set) invalidates everything when adding elements

# Iterator traits

ASS 3

- Each iterator has certain properties
  - Category (input, output, forward, bidirectional, random-access)
  - Value type (T)
  - Reference Type (T& or const T&)
  - Pointer Type (T* or T* const)
    - Not strictly required
  - Difference Type (type used to count how far it is between iterators)
- When writing your own iterator, you need to tell the compiler what each of these are

# Iterator requirements

A custom iterator class should look, at minimum, like this

```
1  #include <iterator>
2
3  template <typename T>
4  class Iterator {
5   public:
6    using iterator_category = std::forward_iterator_tag;
7    using value_type = T;
8    using reference = T&;
9    using pointer = T*; // Not strictly required, but nice to have.
10   using difference_type = int;
11
12   reference operator*() const;
13   Iterator& operator++();
14   Iterator operator++(int) {
15     auto copy{*this};
16     ++(*this);
17     return copy;
18   }
19   // This one isn't strictly required, but it's nice to have.
20   pointer operator->() const { return &(operator*()); }
21
22   friend bool operator==(const Iterator& lhs, const Iterator& rhs) { ... };
23   friend bool operator!=(const Iterator& lhs, const Iterator& rhs) { return !(lhs == rhs); }
24 };
```

# Container requirements

- All a container needs to do is to allow std::[cr]begin / std::[cr]end
  - This allows use in range-for loops, and std algorithms
- Easiest way is to define begin/end/cbegin/cend methods
- By convention, we also define a type Container::[const_]iterator

eg:

intstack

IntStack::iter

```
class Container {
  // Make the iterator using one of these by convention.
  class iterator {...};
  using iterator = ...;

  // Need to define these.
  iterator begin();
  iterator end();

  // If you want const iterators (hint: you do), define these.
  const_iterator begin() const { return cbegin(); }
  const_iterator cbegin() const;
  const_iterator end() const { return cend(); }
  const_iterator cend() const;
};
```

# Custom bidirectional iterators

- Need to define operator--() on your iterator
  - Need to move from c.end() to the last element
    - c.end() can't just be nullptr
- Need to define the following on your container:

```
class Container {
  // Make the iterator
  class reverse_iterator {...};
  // or
  using reverse_iterator = ...;

  // Need to define these.
  reverse_iterator rbegin();
  reverse_iterator rend();

  // If you want const reverse iterators (hint: you do), define these.
  const_reverse_iterator rbegin() const { return crbegin(); }
  const_reverse_iterator crbegin();
  const_reverse_iterator rend() const { return crend(); }
  const_reverse_iterator crend() const;
};
```

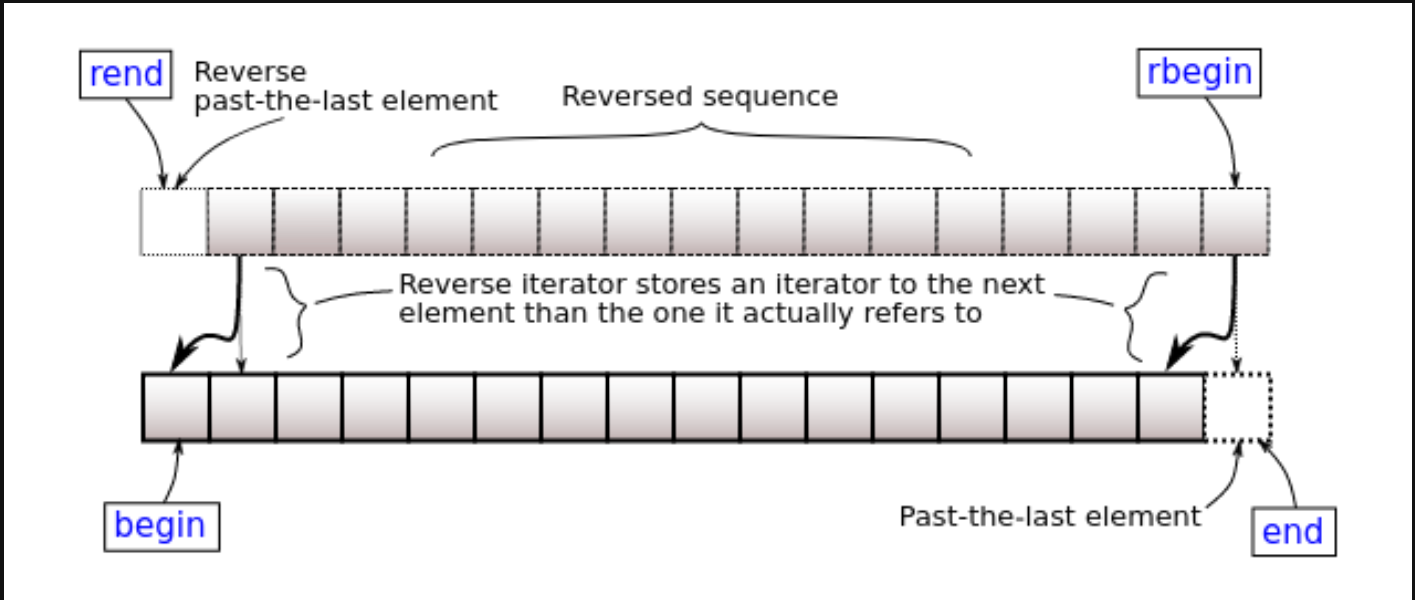
# Automatic reverse iterators

- Reverse iterators can be created by std::reverse_iterator
  - Requires a **bidirectional iterator**
- You should be able to just copy-and-paste the following code

```
class Container {
  // Make the iterator using these.
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;

  // Need to define these.
  reverse_iterator rbegin() { return reverse_iterator{end()}; }
  reverse_iterator rend() { return reverse_iterator{begin()}; }

  // If you want const reverse iterators (hint: you do), define these.
  const_reverse_iterator rbegin() const { return crbegin(); }
  const_reverse_iterator rend() const { return crend(); }
  const_reverse_iterator crbegin() const { return const_reverse_iterator{cend()}; }
  const_reverse_iterator crend() const { return const_reverse_iterator{cbegin()}; }
};
```

# Automatic reverse iterators

- Reverse iterators can be created by std::reverse_iterator
  - rbegin() stores end(), so *rbegin is actually *(--end())



# Random access iterators

```
1  class Iterator {
2    ...
3    using reference = T&;
4    using difference_type = int;
5
6    Iterator& operator+=(difference_type rhs) { ... }
7    Iterator& operator-=(difference_type rhs) { return *this += (-rhs); }
8    reference operator[](difference_type index) { return *(*this + index); }
9
10   friend Iterator operator+(const Iterator& lhs, difference_type rhs) {
11     Iterator copy{*this};
12     return copy += rhs;
13   }
14   friend Iterator operator+(difference_type lhs, const Iterator& rhs) { return rhs + lhs; }
15   friend Iterator operator-(const Iterator& lhs, difference_type rhs) { return lhs + (-rhs); }
16   friend difference_type operator-(const Iterator& lhs, const Iterator& rhs) { ... }
17
18   friend bool operator<(Iterator lhs, Iterator rhs) { return rhs - lhs > 0; }
19   friend bool operator>(Iterator lhs, Iterator rhs) { return rhs - lhs < 0; }
20   friend bool operator<=(Iterator lhs, Iterator rhs) { !(lhs > rhs); }
21   friend bool operator>=(Iterator lhs, Iterator rhs) { !(lhs < rhs); }
22 }
```

See legacy requirements for random access iterators

# Custom `stack<T>` Demo

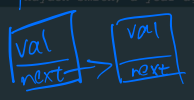
- Live demo

# Feedback

```
#include <algorithm>
#include <cassert>
#include <iterator>
#include <memory>
#include <utility>

using std::begin;
using std::cbegin;
using std::end;


class IntStack {
private:

    struct Node {
        Node(int value, std::unique_ptr<Node>&& next)
        : value{value}
        , next{std::move(t: next)} {}
        int value;
        std::unique_ptr<Node> next;
    };

    // Exercise to the reader once we've covered templates:
    // Try making the const iterator and the non-const iterator with one class template.

    class Iterator {
    public:
        using iterator_category = std::forward_iterator_tag;
        using value_type = int;
        using reference = int&;
        using pointer = int*;
        using difference_type = int;

        reference operator*() const {
            return node_->value;
        }
        pointer operator->() const {
            return &(operator*());
        }
        Iterator operator++() {
            node_ = node_->next.get();
            return *this;
        }
        Iterator operator++(int) {
            auto copy: Iterator{*this};
            ++(*this);
            return copy;
        }

        friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
            return lhs.node_ == rhs.node_;
        }

        friend bool operator!=(const Iterator& lhs, const Iterator& rhs) {
            return !(lhs == rhs);
        }

    private:
        explicit Iterator(Node* node)
        : node_{node} {}
        Node* node_;

        friend class IntStack;
    };

public:
    // TODO(lecture): show how make const and non-const iterators during lecture.
    using iterator = Iterator;
    using const_iterator = Iterator;

    iterator begin() {
        return iterator{node: head_.get()};
    }
    const_iterator begin() const {
        return cbegin();
    }
    const_iterator cbegin() const {
        return const_iterator{node: head_.get()};
    }
    iterator end() {
        return iterator{node: nullptr};
    }
    const_iterator end() const {
        return cend();
    }
    const_iterator cend() const {
        return const_iterator{node: nullptr};
    }

    void push(int value) {
        head_ = std::make_unique<Node>(value, std::move(t: head_));
    }

    // TODO(students): Why doesn't std::stack::pop return the value you pop?
    void pop() {
        head_ = std::move(t: head_->next);
    }

    const int& top() const {
        return *cbegin();
    }
    int& top() {
private:
    std::unique_ptr<Node> head_;
};
```



Node Node




Iterator constructor called

Before After