

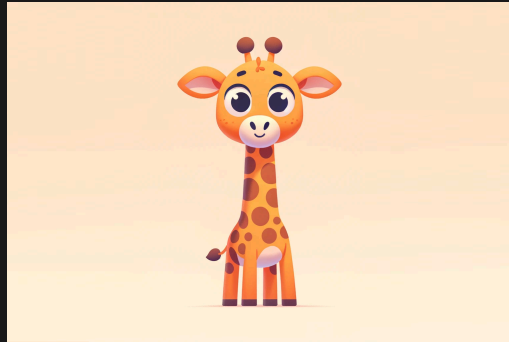
COMP6771



Custom Iterators

Lecture 4.3

Author(s): Hayden Smith



[\(Download as PDF\)](#)

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
 - 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

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



Iterator Invalidation

- What happens when we modify the container?
 - What happens to iterators?
 - What happens to references to elements?
- Using an invalid iterator is undefined behaviour

```
1 #include <vector>
2
3 int main()
4 {
5     std::vector v { 1, 2, 3, 4, 5 };
6     // Copy all 2s
7     for (auto it = v.begin(); it != v.end(); ++it) {
8         if (*it == 2) {
9             v.push_back(2);
10        }
11    }
12    // Erase all 2s
13    for (auto it = v.begin(); it != v.end(); ++it) {
14        if (*it == 2) {
15            v.erase(it);
16        }
17    }
18 }
```

iterator-invalidate.cpp



Iterator Invalidation

- `push_back`

- 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."

```
1 #include <vector>
2
3 int main()
4 {
5     std::vector v { 1, 2, 3, 4, 5 };
6     // Copy all 2s
7     for (auto it = v.begin(); it != v.end(); ++it) {
8         if (*it == 2) {
9             v.push_back(2);
10        }
11    }
12 }
```

`vector-invalidate.cpp`



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

```
1 #include <vector>
2
3 int main()
4 {
5     std::vector v { 1, 2, 3, 4, 5 };
6     // Erase all even numbers (C++11 and later)
7     for (auto it = v.begin(); it != v.end();) {
8         if (*it % 2 == 0) {
9             it = v.erase(it);
10        } else {
11            ++it;
12        }
13    }
14 }
```

erase-invalidate.cpp



Iterator Invalidation

- In 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

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

custom-iter-class.cpp



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`

```
1 class Container {
2     // Make the iterator using one of these by convention.
3     class iterator {
4         ...
5     };
6     using iterator = ...;
7
8     // Need to define these.
9     iterator begin();
10    iterator end();
11
12    // If you want const iterators (hint: you do), define these.
13    const_iterator begin() const { return cbegin(); }
14    const_iterator cbegin() const;
15    const_iterator end() const { return cend(); }
16    const_iterator cend() const;
17 };
```

[container-requirements.cpp](#)



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:

```
1 class Container {
2     // Make the iterator
3     class reverse_iterator {
4         ...
5     };
6     // or
7     using reverse_iterator = ...;
8
9     // Need to define these.
10    reverse_iterator rbegin();
11    reverse_iterator rend();
12
13    // If you want const reverse iterators (hint: you do), define these.
14    const_reverse_iterator rbegin() const { return crbegin(); }
15    const_reverse_iterator crbegin();
16    const_reverse_iterator rend() const { return crend(); }
17    const_reverse_iterator crend() const;
18 };
```

custom-bidirectional.cpp



Automatic Reverse Iterators

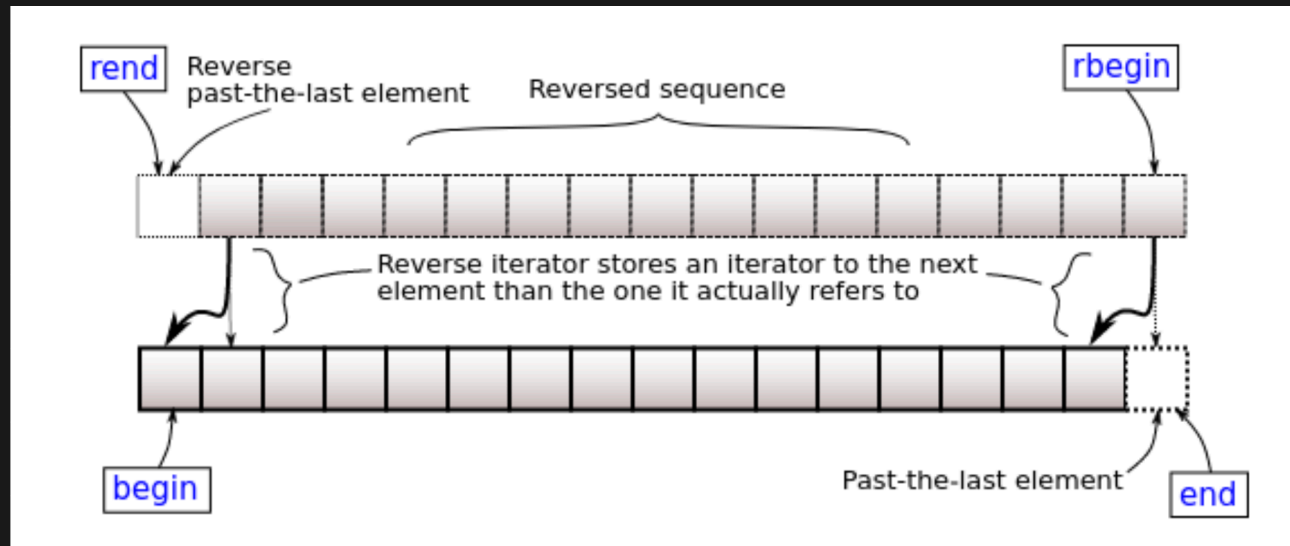
- Reverse iterators can be created by `std::reverse_iterator`
 - Requires a bidirectional iterator
- You should be able to just copy-and-paste the following code

```
1 class Container {
2     // Make the iterator using these.
3     using reverse_iterator = std::reverse_iterator<iterator>;
4     using const_reverse_iterator = std::reverse_iterator<const_iterator>;
5
6     // Need to define these.
7     reverse_iterator rbegin() { return reverse_iterator { end() }; }
8     reverse_iterator rend() { return reverse_iterator { begin() }; }
9
10    // If you want const reverse iterators (hint: you do), define these.
11    const_reverse_iterator rbegin() const { return crbegin(); }
12    const_reverse_iterator rend() const { return crend(); }
13    const_reverse_iterator crbegin() const { return const_reverse_iterator { cend() }; }
14    const_reverse_iterator crend() const { return const_reverse_iterator { cbegin() }; }
15 };
```

auto-reverse.cpp

⏪ Automatic Reverse Iterators

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





Random Access 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

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

[random-access.cpp](#)

Feedback



Or go to the [form here](#).

