COMP6771 Week 7.2

Custom Iterators

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

- 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

```cpp
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);
  }
}
```

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 std::vector v{1, 2, 3, 4, 5};
2 // Copy all 2s
3 for (auto it = v.begin(); it != v.end(); ++it) {
4   if (*it == 2) {
5     v.push_back(2);
6   }
7 }
```

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

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

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
- std::map/set does not invalidate iterators upon insertion (why?)

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     auto copy{*this};
13     ++(*this);
14     return copy;
15   }
16   // This one isn't strictly required, but it's nice to have.
17   pointer operator->() const { return &(operator*()); }
18
19   friend bool operator==(const Iterator& lhs, const Iterator& rhs) { ... };
20   friend bool operator!=(const Iterator& lhs, const Iterator& rhs) { return !(lhs == rhs); }
21 };
```

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    using iterator = ...;
5
6    // Need to define these.
7    iterator begin();
8    iterator end();
9
10   // If you want const iterators (hint: you do), define these.
11   const_iterator begin() const { return cbegin(); }
12   const_iterator cbegin() const;
13   const_iterator end() const { return cend(); }
14   const_iterator cend() const;
15 };
```

Dissecting IntStack

- The iterator traits
- The overloaded operators (*, ->)
- The equality operators
- The constructor (default to nullptr)
- The private data
  - The iterator is defined inside the class, so gets access to private data
  - Iterator defines the container as a friend class for the constructors
- Key points in the List Class:
  - begin() – returns an Iterator object
  - end() – returns an Iterator object (with nullptr as private data)
- Note: The Iterator Class does not modify the List/Node data except through returning references.

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

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

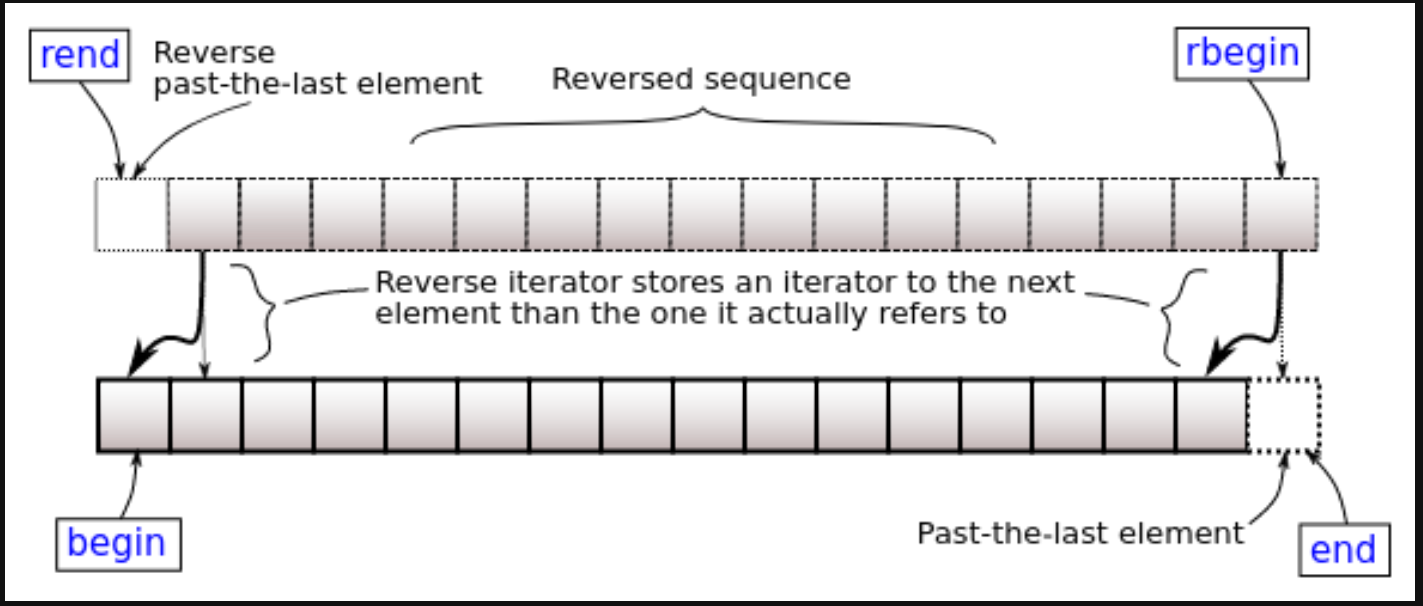
```
class Container {
  // Make the iterator using these.
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;

  // 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

```
class Iterator {
  ...
  using reference = T&;
  using difference_type = int;

  Iterator& operator+=(difference_type rhs) { ... }
  Iterator& operator-=(difference_type rhs) { return *this += (-rhs); }
  reference operator[](difference_type index) { return *(*this + index); }

  friend Iterator operator+(const Iterator& lhs, difference_type rhs) {
    Iterator copy{*this};
    return copy += rhs;
  }
  friend Iterator operator+(difference_type lhs, const Iterator& rhs) { return rhs + lhs; }
  friend Iterator operator-(const Iterator& lhs, difference_type rhs) { return lhs + (-rhs); }
  friend difference_type operator-(const Iterator& lhs, const Iterator& rhs) { ... }

  friend bool operator<(Iterator lhs, Iterator rhs) { return rhs - lhs > 0; }
  friend bool operator>(Iterator lhs, Iterator rhs) { return rhs - lhs < 0; }
  friend bool operator<=(Iterator lhs, Iterator rhs) { !(lhs > rhs); }
  friend bool operator>=(Iterator lhs, Iterator rhs) { !(lhs < rhs); }
}
```

See legacy requirements for random access iterators