COMP6771
Advanced C++ Programming

4.2 – Custom Iterators

In this Lecture

Why?

- We sometimes need our custom types to be iterable.
- We must define that functionality ourselves

What?

- Iterator Revision
- How to Make a Custom Iterator
- Iterator Invalidation

Iterator Revision

- Iterator is an abstract notion of a pointer.
- Iterators are types that abstract container data as a linear sequence of objects.
- They allow containers and algorithms to interface generically.
  - Designers of algorithms need not care about how a container is implemented.
  - Designers of containers need not provide extensive operations.
- Iterators fall into distinct categories.
  - Output, Input, Forward, Bidirectional, Random-access, Contiguous

```
auto v = std::vector{1, 2, 3, 4, 5};
const auto cv = v;

// vector<int>'s non-const iterator
++(*v.begin());

// vector<int>'s const iterator
*cv.begin();

// vector<int>'s const iterator
v.cbegin();
```

Custom Iterators

- A custom iterator is a class type that heavily uses operator overloading to provide the same syntactical operations as a pointer.
- A custom iterator must define certain traits for the compiler.
- Each category of iterator defines its set of operations.
  - Base Iterator Requirements
  - Output Iterator Requirements
  - Input Iterator Requirements
  - Forward Iterator Requirements
  - Bidirectional Iterator Requirements
  - Random-Access Iterator Requirements
  - Contiguous Iterator Requirements

Iterator Traits

- Every iterator has certain required type members.
  - Iterator category
  - Value type
  - Reference type
  - Pointer type
    - Not strictly required
  - Difference type
    - Used to count the number of elements between two iterators
- You must define these yourself in your custom iterators.

```cpp
// iterator traits for an iterator
// modelling an int*

// <iterator> contains the category tags
#include <iterator>

class iter {
public:
  using iterator_category
       = std::contiguous_iterator_tag;

  using value_type = int;

  using reference_type = value_type&;

  using pointer_type = value_type*;
  // could also do pointer_type = void;

  // usually std::pointerdiff_t is sufficient
  using difference_type = std::pointerdiff_t;
};
```

Random-Access Iterator Interface

```
struct random_iter {
  random_iter(); // must be default constructible.
  random_iter(const random_iter &); // must be copy constructible.
  random_iter& operator=(const random_iter &); // must be copy assignable.

  reference operator*() const; // must be dereferenceable and return a reference.
  pointer operator->() const; // only useful if this was an iterator to a class type

  random_iter &operator++();    // must be pre-incrementable.
  random_iter  operator++(int); // must be post-incrementable.

  random_iter &operator--();    // must be pre-decrementable.
  random_iter  operator--(int); // must be post-decrementable.

  random_iter &operator+=(int n); // can progress n spots
  random_iter &operator-=(int n); // can regress n spots

  reference operator[](int); // get the nth element ahead from this position (setter version).
  const reference operator[](int) const; // get the nth element ahead from this position (getter version).

  friend random_iter operator+(random_iter, int n); // new iter n spots ahead
  friend random_iter operator+(int n, random_iter); // new iter n spots ahead (reverse order)

  friend random_iter operator-(random_iter, int n); // new iter n spots behind
  friend difference_type operator-(random_iter, random_iter); // get the distance between two iterators

  auto operator<=>(random_iter) const; // all six comparison functions are needed.
};
```

From a Container to a Range

- A range is a container with certain member types and functions.
  - Particularly, a range can be used in a ranged for-loop.
- Member types:
  - `iterator`
  - `const_iterator`
  - Bidirectional and greater iterators also require:
    - `reverse_iterator`
    - `const_reverse_iterator`
- Member functions:
  - `begin(), end()`
  - `cbegin(), cend()`
  - Bidirectional and greater iterators also require:
    - `rbegin(), rend()`
    - `crbegin(), crend()`

```
class vector {
  struct iter { /* implementation */ };
public:
  using iterator = iter;
  using const_iterator = /* to be defined */;
  using reverse_iterator = /* to be defined */;
  using const_reverse_iterator = /* to be defined */;

  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
  const_iterator cbegin() const;
  const_iterator cend() const;

  reverse_iterator rbegin();
  reverse_iterator rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
  const_reverse_iterator crbegin() const;
  const_reverse_iterator crend() const;
};
```

UNSW SYDNEY

`iterator` & `const_iterator`

- The only practical difference between an `iterator` and a `const_iterator` is that the `value_type` of a `const_iterator` is const-qualified.
- This creates a potential problem of code duplication between const and non-const iterators.
- Solutions:
  - Accept the duplication? ✕
  - Give only one kind of iterator? ⚠
    - For some containers (like a set), only a const_iterator makes sense.
  - Use a template? ☑
    - We will cover templates later in the course.
    - Single-iterator types don't need templates

```cpp
class vector {
  template <typename ValueType>
  struct iter {
    // Instead of hardcoding a type
    // directly, use the type parameter.
    //
    // Most other member types can be
    // written in terms of value_type.
    using value_type = ValueType;
    Using reference = value_type&;
    // more implementation...
  };

  using iterator = iter<int>;
  using const_iterator = iter<const int>;

  // more implementation...
};
```

Automatic Reverse Iteration

- Reverse iterators can be created by using `std::reverse_iterator` .
- Requires a bidirectional iterator or greater.
- `rbegin()` stores `end()`, so `*rbegin` is actually `*(--end())` .

```
class vec {
public:
  using iterator = /* ... */;
  using const_iterator = /* ... */;
  using reverse_iterator =
        std::reverse_iterator<iterator>;
  using const_reverse_iterator =
        std::reverse_iterator<const_iterator>;

  iterator begin() { /* ... */ }
  iterator end() { /* ... */ }

  reverse_iterator rbegin() {
    return reverse_iterator{end()};
  }

  reverse_iterator rend() {
    return reverse_iterator{begin()};
  }

  // similar for other reverse iterator methods
};
```

Iterator – Container Relationship

- Designers of containers usually provide the iterators also.
- The iterator must be at least publically default constructible, but usually has at least one private constructor.
- The container uses the private constructor to intialise the iterator to the start (if `[cr]begin()`) or end (if `[cr]end()`) of the range.
- This means that the container must be a `friend` of the iterator.
- Usually, the iterator class is defined as in *inner class* in the container.

`int` Stack Example: Container

Live Demo

`int` Stack Example: Iterator

Live Demo

Iterator Invalidation

- An iterator is an abstract notion of a **pointer**.
  - If the object a pointer points to moves, that pointer *dangles* and it is no longer valid to dereference.
- If the data an iterator references moves, it can dangle too.
- This is called **iterator invalidation**
  - No longer valid to dereference the iterator.
- Iterator invalidation is the consequence of (usually) adding or removing elements.
  - Element modification virtually never results in invalidation.

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

// the call to push_back may result in v’s data
// being expanded and moved to a new location.
// if v.size() == v.capacity() when push_back()
// is called, it will expand.

// if it did not expand, only variables holding
// the old v.end() are invalidated.

// if it did expand, all iterator variables are
// invalidated and cannot be used.
```

Iterator Invalidation in the STL

- Iterators from array-backed containers (vector, unordered_map, unordered_set, etc.) are invalidated when the array needs to grow or shrink.
  - For `std::vector`, this happens most often through `push_back()`.
  - For the unordered containers, this happens most when a rehash of elements is needed.
- Iterators from linked data structures (`list`, `map`, `set`, etc.) are only invalidated when elements are removed.
  - Size changes result in internal pointers being adjusted, but these don't affect the iterators.

Feedback (stop recording)