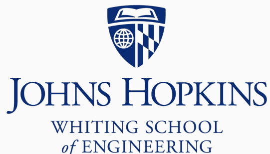
Containers

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Containers

We saw STL containers and learned what they have in common:

- `container<...>::iterator` is the iterator type
  - Also `::const_iterator`, `::reverse_iterator`, ...
- `.begin()` return iterator "pointing" to beginning
- `.end()` return iterator "pointing" just past end
  - `.cbegin()` / `.cend()` for `const_iterator`
  - `.rbegin()` / `.rend()` for `reverse_iterator`

Containers

```
#include <iostream>
#include <vector>

using std::cout;   using std::endl;
using std::vector;

int main() {
    vector<int> vec = {2, 4, 6, 8};
    // Iterate forward
    for(vector<int>::iterator it = vec.begin();
        it != vec.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl;
    // Iterate backward
    for(vector<int>::reverse_iterator it = vec.rbegin();
        it != vec.rend(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl;
    return 0;
}
```

Containers

```
$ g++ -c iter_eg.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o iter_eg iter_eg.cpp
$ ./iter_eg
2 4 6 8
8 6 4 2
```

Containers

STL containers also have:

- `container<...>::size_type`
  - Integer type for indexing container items
  - Almost always `size_t`
- `container<...>::value_type`
  - Type of items in the container
  - `std::vector<T>::value_type` is `T`
  - `std::map<K, V>::value_type` is `std::pair<K, V>`

Implementing an iterator

How would we *implement* an iterator for a container?

Begin with simple `vector`-like `class`:

Implementing an iterator

```
template <typename T>
class Vec {
  public:
    // TODO: iterator typedefs: ::iterator, ::const_iterator, ...

    typedef size_t size_type;
    typedef T value_type;

    Vec() { ... }

    void push_back(const T& item) { ... }
    T pop() { ... }

    // TODO: iterator accessors: begin(), end(), cbegin(), ...

  private:
    size_t size;     // # elts in vector
    size_t reserved; // # elts allocated in data array
    T *data;         // points to beginning of data array
};
```

Implementing an iterator

Can we simply use pointers? Let's try

Iterator type is T*:

```
template <typename T>
class Vec {
  public:
    typedef T* iterator;
    typedef const T* const_iterator;
  ...
};
```

Implementing an iterator

.begin() and .end() return pointers to the appropriate locations

```
template <typename T>
class Vec {
  public:
  ...

    T* begin() { return data; }
    T* end() { return data + size; }

    const T* cbegin() const { return data; }
    const T* cend() const { return data + size; }

  private:
    size_t size;     // # elts in vector
    size_t reserved; // # elts allocated in data array
    T *data;         // points to beginning of data array
};
```

Implementing an iterator

```
int main() {
    Vec<int> vec;
    vec.push(2); vec.push(4); vec.push(6); vec.push(8);
    // Iterate forward
    for(Vec<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << ' ';
    }
    cout << endl;
    return 0;
}
```

```
$ g++ -o vec_iter vec_iter.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./vec_iter
2 4 6 8
```

Implementing an iterator

Simply returning a pointer works as long as elements are laid out consecutively in memory

- Fine for `Vec` or `std::vector`
- Not fine for `std::map`, `std::list`, ...

Pointers also don't work for `reverse_iterator`

- `++ptr` goes forward, but ++ for `reverse_iterator` should go backward

Implementing an iterator

Alternative is to define a new `class` that handles iterating for `Vec`

Good example of where *nested* `classes` are useful:

```
template <typename T>
class Vec {
  public:

    class iterator {
        // ...
    };
    class const_iterator {
        // ...
    };
    class reverse_iterator {
        // ...
    };
  ...
};
```

Implementing an iterator

Nested `class` has access to members of the enclosing class, including `private` members

We don't really need that here; each `iterator` `class` simply wraps a layer of operator overloads around a pointer

Implementing an iterator

```
template <typename T>
class Vec {
  public:

    class iterator {
        T* it;

      public:
        iterator(T* initial) : it(initial) { }

        // *** Overloads ***

        ...
    };
   ...
 };
```

Implementing an iterator

What do we need to overload? At least:

- `operator++` for `++it`
- `operator*` for `*it`
- `operator!=` for `while(it != container.end())`

That's all we need for today, but a real-world iterator might additionally handle:

- `operator==`
- `operator->`

Implementing an iterator

```
template <typename T>
class Vec {
  public:

    class iterator {
        T* it;

      public:
        iterator(T* initial) : it(initial) { }

        iterator& operator++() { ??? }

        bool operator!=(const iterator& o) const { ??? }

        T& operator*() { ??? }
    };
  ...
};
```

Implementing an iterator

```
template <typename T>
class Vec {
  public:

    class iterator {
        T* it;

      public:
        iterator(T* initial) : it(initial) { }

        iterator& operator++() {
            ++it;              // advance pointer 1 slot
            return *this;
        }

        bool operator!=(const iterator& o) const {
            return it != o.it; // if pointers are different, iterators
                               // are pointing to different slots
        }

        T& operator*() {
            return *it;        // dereferencing iterator is
                               // dereferencing pointer
        }
    };
  ...
};
```

Implementing an iterator

Study questions:

- How would `const_iterator` implementation differ?
  - Hint: `operator*` needs to change
- How would `reverse_iterator` implementation differ?
  - Hint: `operator++` needs to change