Getting to Know the Standard Library Session 2

Alex Voronov

Getting to Know the Standard Library

C++ London Uni

1. Introduction to unit testing with Catch2
2. Basic containers
   - **std::vector**
   - std::string
3. Lambda functions and std::function
4. Associative containers
   - std::map and std::unordered_map
   - std::set and std::unordered_set
   - Associative containers with custom types
   - Set algorithms
5. Overview of algorithms in the standard library

std::vector: Session plan

C++ London Uni

- Introduciton
  - Definition and basic properties
  - Big-O notation
- Basic operations
  - push_back/pop_back
  - Construction
  - Accessing elements
  - resize
- A few words about iterators
- insert and erase
  - Iterator invalidation
- Summary

std::vector

A container for efficiently storing and accessing variable-size sequences of elements of the same type

Does well

- Add and remove elements from the end of the sequence
- Access elements by their index in the sequence
- Iterate linearly over the sequence

Does not that well

- Lookup an element by value
- Add or remove elements in an arbitrary position

Algorithmic complexity: Big-O notation

Estimation of how an algorithm behaves with growths of the input size

- Two aspects: *running time* and *memory space*
- $f(x) = O(g(x))$ if $|f(x)| <= M \cdot g(x)$ for x in $[x0, +\infty)$

What can it be: O(1), O(logN), O(N), $O(N^2)$, $O(N^3)$, $O(2^N)$, O(N!), …

Examples for running time:

- Looking up a note in your notepad: all pages — linear, O(N)
- Looking up a word in a dictionary: always split in half — logarithmic, O(logN)
- Googling the meaning of the word: type and maybe click — constant, O(1)

std::vector: Session plan

C++ London Uni

- Introduciton
  - Definition and basic properties
  - Big-O notation
- **Basic operations**
  - push_back/pop_back
  - Construction
  - Accessing elements
  - resize
- A few words about iterators
- insert and erase
  - Iterator invalidation
- Summary

```
TEST_CASE("empty vector") {
    std::vector<std::string> fruit_basket;

    CHECK(fruit_basket.empty());
    CHECK(fruit_basket.size() == 0);
}
```

C++ London Uni

```
TEST_CASE("add an element and lookup it by index") {
    std::vector<std::string> fruit_basket;

    fruit_basket.push_back("banana");
    fruit_basket.push_back("apple");
    CHECK_FALSE(fruit_basket.empty());
    CHECK(fruit_basket.size() == 2);
    CHECK(fruit_basket[0] == "banana");
    CHECK(fruit_basket[1] == "apple");

    fruit_basket[1] = "orange";
    CHECK(fruit_basket[1] == "orange");
}
```

```
TEST_CASE("vector construction") {
    std::vector<std::string> fruit_basket{"banana", "apple", "orange"};
    CHECK(fruit_basket.size() == 3);
    CHECK(fruit_basket[0] == "banana");
    CHECK(fruit_basket[1] == "apple");
    CHECK(fruit_basket[2] == "orange");

    std::vector<std::string> basket_full_of_nothing(4);
    CHECK(basket_full_of_nothing.size() == 4);
    CHECK(basket_full_of_nothing == std::vector<std::string>{"", "", "", ""});

    std::vector<std::string> banana_basket(5, "banana");
    CHECK(banana_basket.size() == 5);
    CHECK(banana_basket ==
          std::vector<std::string>{"banana", "banana", "banana", "banana",
                                   "banana"});

    std::vector<std::string> twin_basket = fruit_basket;
    CHECK(twin_basket == fruit_basket);
}
```

C++ London Uni

```
TEST_CASE("accessing with [i] vs .at(i)") {
    std::vector<std::string> fruit_basket{"banana", "apple", "orange"};

    CHECK(fruit_basket[2] == "orange");
    CHECK(fruit_basket.at(2) == "orange");
    CHECK_THROWS_AS(fruit_basket.at(3), std::out_of_range);

    // fruit_basket[3] is undefined behavior
}
```

C++ London Uni

```
TEST_CASE("resize") {
    std::vector<std::string> fruit_basket{"banana", "apple"};

    fruit_basket.resize(3);
    CHECK(fruit_basket.size() == 3);
    CHECK(fruit_basket == std::vector<std::string>{"banana", "apple", ""});

    fruit_basket.resize(1);
    CHECK(fruit_basket.size() == 1);
    CHECK(fruit_basket == std::vector<std::string>{"banana"});
}
```

std::vector: Session plan

- Introduciton
  - Definition and basic properties
  - Big-O notation
- Basic operations
  - push_back/pop_back
  - Construction
  - Accessing elements
  - resize
- **A few words about iterators**
- insert and erase
  - Iterator invalidation
- Summary

Iterators

Iterator is an object pointing to an element of a sequence that supports two operations

- Dereference: `*it` — access the object
- Advance: `++it` or `it++` — move to next object

- Most of the standard library iterator types support equality comparison `it1 == it2` and `it1 != it2`

Flavours of iterators

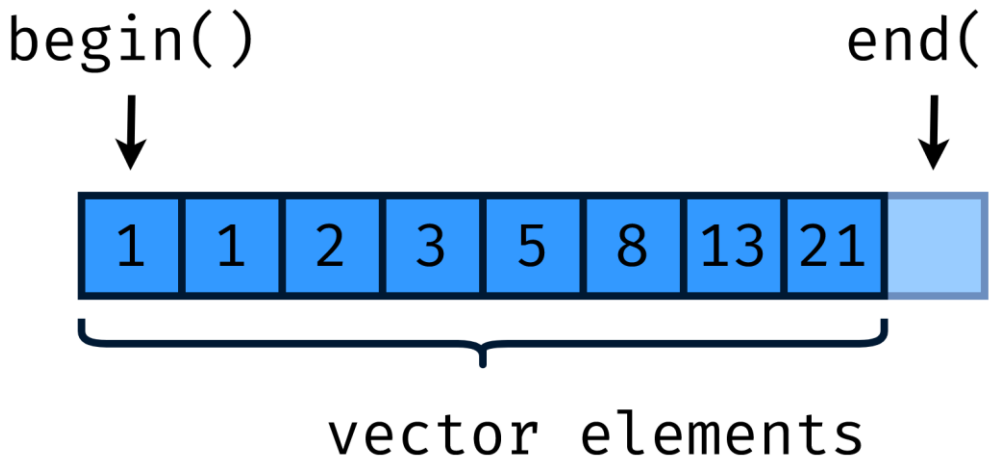
- Forward iterator: `*it`, `++it` and often `it1 == it2`
- Bidirectional iterator: all of the above and `--it` (previous element)
- Random access iterator: all of the above and
  - Arithmetic: `it + N`, `it - N`, `it2 - it1`
  - Ordered comparison: `it1 < it2` (also `<=`, `>`, `>=`)
  - Dereference operator: `it[N]` — equivalent to `*(it + N)`

Iterators can be constant (read-only) and mutable (read-write)

`std::vector` iterators are random access

```cpp
size_t count_positive(const std::vector<int> &numbers) {
    size_t count = 0u;
    for (auto number : numbers) {
        if (number > 0) {
            ++count;
        }
    }

    return count;
}

size_t count_positive_using_iterators(const std::vector<int> &numbers) {
    size_t count = 0u;
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        if (*it > 0) {
            ++count;
        }
    }

    return count;
}
```

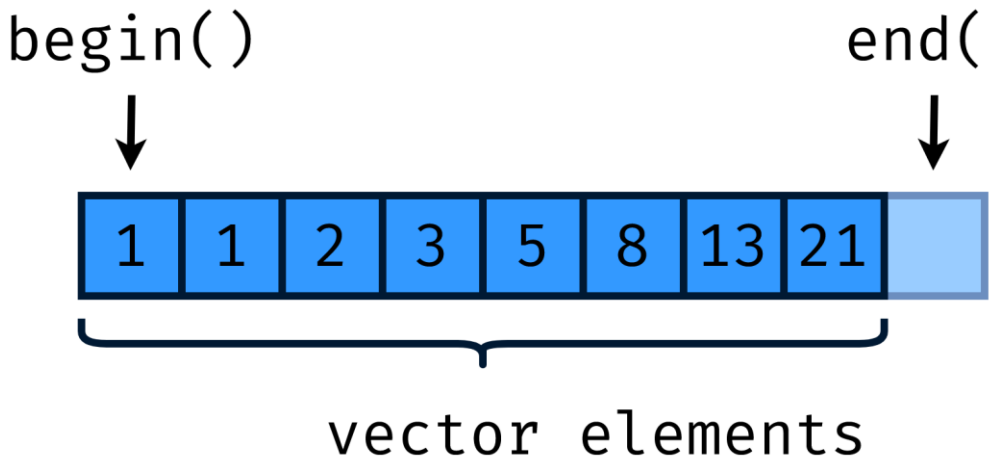
```cpp
size_t count_positive(const std::vector<int> &numbers) {
    size_t count = 0u;
    for (auto number : numbers) {
        if (number > 0) {
            ++count;
        }
    }

    return count;
}

size_t count_positive_using_iterators(const std::vector<int> &numbers) {
    size_t count = 0u;
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        if (*it > 0) {
            ++count;
        }
    }

    return count;
}
```




```
size_t count_positive_using_iterators(const std::vector<int> &numbers) {
    size_t count = 0u;
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        if (*it > 0) {
            ++count;
        }
    }

    return count;
}
```

vector iterators

- `begin(), end()` – a default pair of iterators
- `cbegin(), cend()` – constant iterators (can't write with them)
- `rbegin(), rend()` –reverse-order iterators
- `crbegin(), crend()` – constant reverse-order iterators




```
size_t count_positive_using_iterators(const std::vector<int> &numbers) {
    size_t count = 0u;
    for (auto it = numbers.cbegin(); it != numbers.cend(); ++it) {
        if (*it > 0) {
            ++count;
        }
    }

    return count;
}
```

std::vector: Session plan

C++ London Uni

- Introduciton
  - Definition and basic properties
  - Big-O notation
- Basic operations
  - push_back/pop_back
  - Construction
  - Accessing elements
  - resize
- A few words about iterators
- **insert and erase**
  - Iterator invalidation
- Summary

```
TEST_CASE("insert elements") {
    std::vector<std::string> fruit_basket{"banana", "apple"};

    fruit_basket.insert(fruit_basket.begin() + 1, "persimmon");
    CHECK(fruit_basket ==
          std::vector<std::string>{"banana", "persimmon", "apple"});

    std::vector<std::string> bag_of_oranges(3, "orange");
    fruit_basket.insert(fruit_basket.end(), bag_of_oranges.begin(),
                        bag_of_oranges.end());
    CHECK(fruit_basket ==
          std::vector<std::string>{"banana", "persimmon", "apple", "orange",
                                   "orange", "orange"});
}
```

```
TEST_CASE("erase and clear") {
    std::vector<std::string> fruit_basket{"banana", "persimmon", "apple"};

    SECTION("erase one") {
        fruit_basket.erase(fruit_basket.begin() + 1);
        CHECK(fruit_basket == std::vector<std::string>{"banana", "apple"});
    }

    SECTION("erase range") {
        fruit_basket.erase(fruit_basket.begin(), fruit_basket.begin() + 2);
        CHECK(fruit_basket == std::vector<std::string>{"apple"});
    }

    SECTION("clear") {
        fruit_basket.clear();
        CHECK(fruit_basket.empty());
    }
}
```

```
std::vector<std::string> erase_persimmon(std::vector<std::string> fruit) {
    for (auto it = fruit.begin(); it != fruit.end(); ++it) {
        if (*it == "persimmon") {
            fruit.erase(it);
        }
    }
    return fruit;
}



TEST_CASE("erase persimmon") {
    std::vector<std::string> fruit_basket{
            "banana", "orange", "persimmon", "apple", "persimmon"};
    CHECK(erase_persimmon(fruit_basket) ==
          std::vector<std::string>{"banana", "orange", "apple"});
}
```

```
std::vector<std::string> erase_persimmon(std::vector<std::string> fruit) {
    for (auto it = fruit.begin(); it != fruit.end(); ++it) {
        if (*it == "persimmon") {
            fruit.erase(it);
        }
    }
    return fruit;
}
```

Not going to end well

```
TEST_CASE("erase persimmon") {
    std::vector<std::string> fruit_basket{
            "banana", "orange", "persimmon", "apple", "persimmon"};
    CHECK(erase_persimmon(fruit_basket) ==
          std::vector<std::string>{"banana", "orange", "apple"});
}
```

```
std::vector<std::string> erase_persimmon(std::vector<std::string> fruit) {
    for (auto it = fruit.begin(); it != fruit.end();) {
        if (*it == "persimmon") {
            it = fruit.erase(it);
        } else {
            ++it;
        }
    }
    return fruit;
}

TEST_CASE("erase persimmon") {
    std::vector<std::string> fruit_basket{
            "banana", "orange", "persimmon", "apple", "persimmon"};
    CHECK(erase_persimmon(fruit_basket) ==
          std::vector<std::string>{"banana", "orange", "apple"});
}
```

Vector: iterator invalidation

A method that changes the vector size is likely to invalidate the iterators

- push_back/pop_back
- insert/erase
- resize
- clear
- …

std::vector: Session plan

C++ London Uni

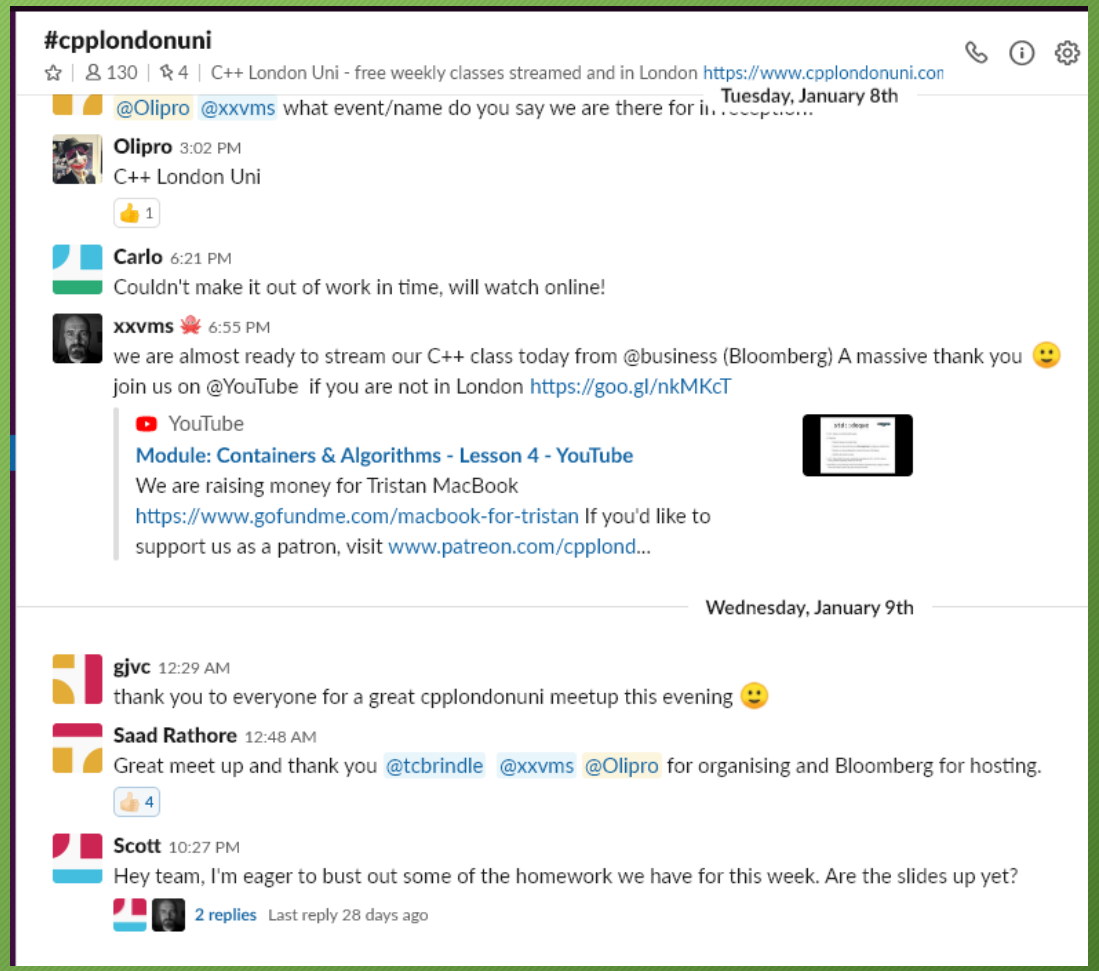
- Introduciton
  - Definition and basic properties
  - Big-O notation
- Basic operations
  - push_back/pop_back
  - Construction
  - Accessing elements
  - resize
- A few words about iterators
- insert and erase
  - Iterator invalidation
- **Summary**

Summary: std::vector

- It's a container for efficiently storing variable size sequences
- Creating a vector of a given size and adding/removing elements from the back is the most efficient
- `.at()` reports an error when accessing an element out of bounds
- Vector allows `insert/erase` in arbitrary location and `resize`
- Iterators are widely-used in the standard library
- Beware of iterator invalidation

Feedback

- We'd love to hear from you!
- The easiest way is via the *CPPLang* Slack organisation. Our chatroom is #cpplondonuni
- If you already use Slack, don't worry, it supports multiple workgroups!
- Go to https://slack.cpp.al to register.



Thank You!

As usual, we will be going to the pub! Support us @ https://patreon.com/CPPLondonUni