Containers and algorithms — session 7

Tristan Brindle

Feedback

- We'd love to hear from you!
- The easiest way is via the *cpplang* channel on Slack — we have our own chatroom, *#cpplondonuni*
- Go to https://cpplang.now.sh/ for an "invitation"

Last week

- Algorithmic complexity

- Anatomy of a standard algorithm

This week

- Overview of the standard library algorithms

- End of module quiz

Last week's homework

- https://github.com/CPPLondonUni/algorithms_exercise

- Implement some standard algorithms

Solution

- https://github.com/CPPLondonUni/algorithms_exercise/tree/solution

Any questions before we move on?

Standard algorithms overview

- The standard library contains more than 90 algorithms which you can (and should!) use in your own programs

- We can’t possibly go over every algorithm in detail

- Instead, today I’ll give a rapid-fire tour over some of those that I find most useful

- cppreference.com has some nice usage examples of many algorithms

- See also Jonathan Boccara’s excellent “105 STL algorithms in less than an hour” video: https://youtu.be/bFSnXNIsK4A

all_of, any_of, none_of

- The functions `all_of`, `any_of` and `none_of` accept a range of elements and a unary predicate, and return a bool

- The return `true` if `pred(element)` is true for all of, any of, and none of the elements respectively

- These are surprisingly useful when combined with lambdas

Example


```cpp
const std::vector<int> vec{1, 2, 3, 4, 5};

bool less_than_ten(int i) { return i < 10; }

assert(std::all_of(vec.begin(), vec.end(), less_than_ten));

assert(std::any_of(vec.begin(), vec.end(),  [](int i) { return i == 3; });

bool is_lower_case(const std::string& s)
{
    return std::none_of(s.begin(), s.end(), ::isupper);
}
```

Searching

- There are several algorithms related to searching for one element (or range of elements) inside another
  - find — looks for a particular value in a sequence
  - find_if — looks for an element matching a predicate
  - find_if_not — looks for an element which does not match a predicate
  - find_first_of — takes two ranges. Looks for an element in range1 which is equal to one of the elements in range2
  - search — takes two ranges. Looks for the first occurrence of range2 as a subsequence of range1
  - find_end — takes two ranges. Looks for the last occurrence of range2 as a subsequence of range1

Examples


```cpp
const std::string str = "Hello C++ London Uni, Hello";

std::find(str.begin(), str.end(), ' ');
// returns an iterator to the first space character

std::find_if(str.begin(), str.end(), ::ispunct);
// returns an iterator to the first + character

std::find_if_not(str.begin(), str.end(), ::isupper);
// returns an iterator to 'e' in position 1

const std::vector<char> vowels{'a', 'e', 'i', 'o', 'u'};

std::find_first_of(str.begin(), str.end(), vowels.begin(), vowels.end());
// returns an iterator to 'e' in position 1

const std::string hello = "Hello";

std::search(str.begin(), str.end(), hello.begin(), hello.end());
// returns an iterator to position 0

std::find_end(str.begin(), str.end(), hello.begin(), hello.end());
// returns an iterator to position end()-5
```

Comparisons

- There are a few standard library algorithms which allow us to compare the elements in two ranges. Note that these don't require the ranges to be of the same type — only that we can compare the elements

- They also take optional predicates allowing us to define the meaning of "equal" or "less-than".

  - `equal` — returns true if they are the same length, and every element is equal

  - `lexicographical_compare` — compares each element in turn; returns true if the first range is "less than" the second

  - `mismatch` — returns a pair of iterators containing the first position in which the element of range1 does not equal the corresponding element in range2

Examples


```cpp
const std::string str = "Hello";
const std::vector<int> vec{'H', 'e', 'l', 'l', 'o'};

std::equal(str.begin(), str.end(), vec.begin(), vec.end());
// returns true

const std::string jones1 = "Jones, A";
const std::string jones2 = "Jones, B";

std::lexicographical_compare(jones1.begin(), jones1.end(),
                             jones2.begin(), jones2.end(),
                             std::greater<char>{});
// returns false!

const std::string hello1 = "Hello World";
const std::string hello2 = "Hello C++ London Uni";

auto pair = std::mismatch(hello1.begin(), hello1.end(),
                          hello2.begin(), hello2.end());

// pair.first is an iterator that points to 'W' in hello1
// pair.second is an iterator that points to 'C' in hello2
```

Copy/copy_if

- copy takes an input range and copies every element into an output range

- This is super optimised and very fast!

- This can be used to print a sequence using `std::ostream_iterator`

- `copy_if` is similar, but only copies those elements for which a given predicate returns true

Examples


```cpp
#include <algorithm>
#include <iostream>
#include <iterator>

const std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::copy(vec.begin(), vec.end(),
          std::ostream_iterator(std::cout, " "));
// Prints every element separated by a space

std::vector<int> out{};

bool is_even(int i) { return i % 2 == 0; }

std::copy_if(vec.begin(), vec.end(), std::back_inserter(out),
             is_even);
// out contains [2, 4, 6, 8, 10]
```

Remove/remove_if

- `remove` takes a range and a value, and "removes" those elements which compare equal to the value

- But it doesn't actually *remove* anything! In fact it *moves* the elements to the back of the container, and returns an iterator to the new "end" of the container

- The actual container (e.g. `vector`) is *still the same size* — its elements have just been moved around

- To actually delete the elements, we need to use the container's `erase()` method — the so-called *erase-remove idiom*

Examples


```cpp
std::vector<int> vec{1, 1, 1, 2, 3, 4, 5};

auto it = std::remove(vec.begin(), vec.end(), 1);
// vec now contains [2, 3, 4, 5, 1, 1, 1]
// it is an iterator which points to the first '1' (end() - 3)
// this is the new "end" of the container

// To actually erase the elements, we need to call vec.erase()
vec.erase(it, vec.end())
// erases everything from it to vec.end()
// vec now contains [2, 3, 4, 5]

// We can do this all in one step as well
std::string str = "AbCdEfG";
str.erase(std::remove_if(str.begin(), str.end(), ::islower), str.end());
// str now contains "ACEG"
```

Transform

- `transform` is perhaps the most general function in the whole library, and one of the most useful

- It takes an input range, a unary function, and an output range. It applies the function to every element of the input in turn, placing the result in the output range

- This operation is often called “map” in functional programming languages

- There is also a binary transform version which takes two input ranges and a binary function

Example


```cpp
const std::string in = "hello";
std::string out;

std::transform(in.begin(), in.end(), std::back_inserter(out), ::toupper);
// out now contains "HELLO"

// The input and output ranges can overlap
std::vector<int> ints{1, 2, 3, 4, 5};
int square(int i) { return i * i; }

std::transform(ints.begin(), ints.end(), ints.begin(), square);
// ints now contains {1, 4, 9, 16, 25)

// There is a binary (two input) version too
const std::vector<int> in1{1, 2, 3, 4, 5};
const std::vector<int> in2{5, 4, 3, 2, 1};
std::vector<int> out(5);

std::transform(in1.begin(), in1.end(),
               in2.begin(), in2.end(),
               out.begin(), std::multiplies<int>{});
// out contains [5, 8, 9, 8, 5]
```

Numeric algorithms

- As well as <algorithm>, the header <numeric> contains extra algorithms

- Probably the two most useful are `iota` and `accumulate`

- `iota` takes a range and a starting value, and fills the range with ++value

- `accumulate` computes the sum of the elements in a range

- We can also customise the operation that `accumulate` performs — this is equivalent to a left fold in functional languages

Examples


```cpp
std::vector<int> ints(5);

std::iota(ints.begin(), ints.end(), 1);
// ints now contains [1, 2, 3, 4, 5]

auto sum = std::accumulate(ints.begin(), ints.end(), 0);
// sum is 1+2+3+4+5 = 15

auto product = std::accumulate(ints.begin(), ints.end(), 0,
                               std::multiples<int>{});
// product is 1*2*3*4*5 = 5! = 120


// We can also do really clever things with accumulate
auto dash_fold = [](std::string a, int b) {
    return std::move(a) + '-' + std::to_string(b);
};

std::string s = std::accumulate(std::next(ints.begin()), ints.end(),
                                std::to_string(v[0]), // start with first element
                                dash_fold);
// s contains 1-2-3-4-5
```

And that’s it!

Quiz

- It’s quiz time!

- http://bit.do/cppmodulesandalgorithms

- or https://goo.gl/forms/S3XQ7NDfjb3a31ZW2

Next week

- NO CLASS next week (5th Feb) due to C++ on Sea

- The week after (12th Feb) we start a new module — object orientated programming

Online resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)