Algorithms

CSE 232 - Dr. Josh Nahum

Reading:

Section 13.1 - Section 13.5

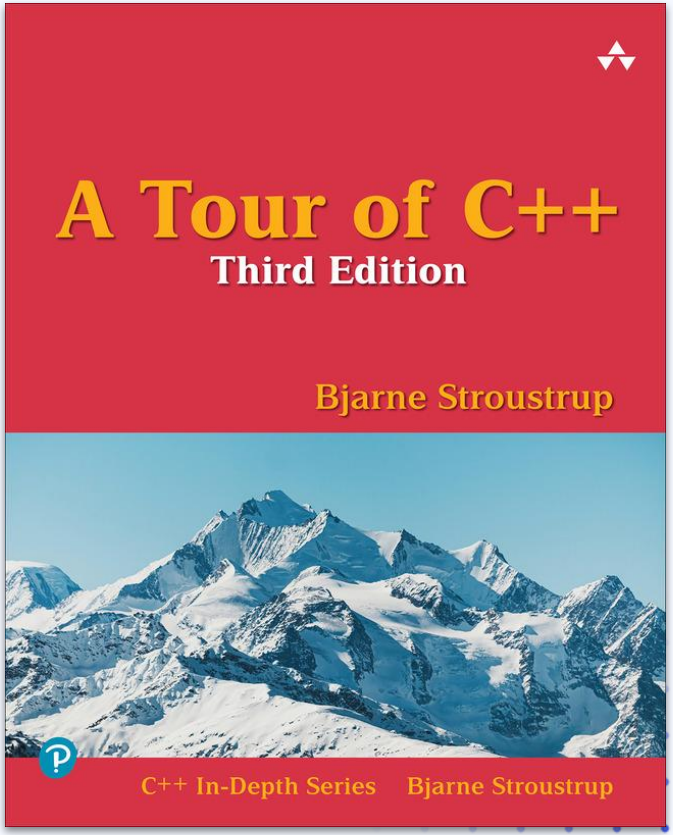

Table of contents

00

Iterator Types

Const and Non-const

`Container::iterator`

This iterator can change the container it is associated with

`Container::const_iterator`

This iterator can only read (not write to) its associated container. Similar to a pointer to a const array.

Forward Iterator

Dereference and Equality

Forward iterators can be dereferenced (`*iter`) and read from or assigned to. Support equality with other iterators on the same container (`iter == x` and `iter != x`)

Forward

They support pre and post-increment operators (`++iter` and `iter++`) which causes them to refer to the "next" element in a container.

Bidirectional Iterator

Forward

All the powers of a Forward Iterator!

Backward

They support pre and post-decrement operators (`--iter` and `iter--`) which causes them to refer to the "previous" element in a container.
Allows multi-pass algorithms!

Random Access Iterator

Bidirectional

All the powers of a Bidirectional Iterator!

Arithmetic

They support addition and subtraction with integers, and subscripting (indexing). Allows for "jumps" to a particular location adding the number of steps to an iterator. Can also use relational operators.

Container's Iterator

vector	list	deque	forward_list
Random Access	Bidirectional	Random Access	Forward
map	**set**	**stack**	**priority_queue**
Bidirectional	Bidirectional	No support	No support

01

Arrays

Pointers are iterators!

They are Forward Iterators because the following code is correct:

```
int array[3] = {2, 3, 2};
int * ar_begin = array;
int * ar_end = array + 3;
ar_begin++;
*ar_begin = 4;
bool b = ar_begin != ar_end;
```

Pointers are also Bidirectional Iterators because this code is also correct:

```
int * iter = ar_end--;
```

Pointers are Random Access Iterators!

They are Random Access Iterators because the following code is correct:

```
ar_begin -= 2;
iter = ar_begin + 1;
bool b2 = iter < ar_begin;
int i = iter[2];
```

02

Sort Example

Sort

Very Common

Sort is one of the most commonly used algorithms. You likely also had to use it on a lab.

Iterators

Like most algorithms, it takes two iterators denoting the range of elements to sort. (The iterators must be Random Access.)

Example

```
string const original =
  "My dog is named Mal.";
string copy{original};
sort(copy.begin(), copy.end());
cout << copy << endl;
// prints: "    .MMaaddegilmnosy"
```

You can sort a string by passing in the begin and end iterators. This changes the string!

Note: the default sort compares elements using operator< (which treats char as their ASCII values).

Example (continued)

```
copy = original;
std::string::iterator start =
  copy.begin() + 5;
sort(start, copy.end());
cout << copy << endl;
// prints: "My do   .Maadegilmns"
```

All algorithms take iterators (not containers). And the iterators don't have to be the begin and end iterators. Here only part of the string is sorted.

Example (continued)

```
bool CaseInsensitiveLess(
    char left, char right) {
  return tolower(left) < tolower(right);
}
// ...
copy = original;
sort(copy.begin(), copy.end(),
  CaseInsensitiveLess);
cout << copy << endl;
// prints: "    .aaddegilMmMnosy"
```

Sort can take an optional predicate function that provides a different operator< used to compare elements. Here, `CaseInsensitiveLess` is used to compare two chars after both are made lowercase, which results in a sort that isn't sensitive to case. Here 'M' and 'm' compare as equal.

Example (continued)

```
copy = original;
sort(copy.begin(), copy.end(),
  [](char left, char right){
    return tolower(left) < tolower(right);
  });
cout << copy << endl;
// prints: "    .aaddegilMmMnosy"
```

Here a lambda expression is used to accomplish the exact same task.

<ranges>

Almost every algorithm has a traditional iterator interface and a counterpart that can accept a container directly.

```
sort(copy.being(), copy.end());
sort(copy);
```

To use the range version of an algorithm, you have to specify the range namespace.

```
copy = original;
std::ranges::sort(copy);
// or
using std::ranges::sort;
sort(copy);
cout << copy << endl;
// prints: "    .MMaaddegilmnosy
```

03

Important Algos

“The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements.”

—cppreference.com

Noteworthy Algorithms

for_each

applies a function to a range of elements

for_each_n

applies a function object to the first N elements of a sequence

all_of / any_of

checks if a predicate is true for all or any of the elements in a range

find

searches for an element equal to value

find_if

searches for an element for which predicate returns true

adjacent_find

finds the first two adjacent items that are equal

Noteworthy Algorithms

`count/count_if`

returns the number of elements satisfying specific criteria

`equal`

determines if two sets of elements are the same

`search`

searches for a range of elements

`copy/copy_if`

copies a range of elements to a new location

`transform(unary)`

applies a unary function to a range of elements, storing results in a destination range

`transform(binary)`

applies a binary function to two ranges of elements

Noteworthy Algorithms

`generate`

assigns the results of successive function calls to every element in a range

`reverse`

reverses the order of elements in a range

`shuffle`

randomly re-orders elements in a range (more on this later)

`partition`

divides a range of elements into two groups

`sort`

sorts a range into ascending order

`stable_sort`

sorts a range of elements while preserving order between equal elements

Noteworthy Algorithms

`max_element`

returns the largest element in a range

`min_element`

returns the smallest element in a range

`max`

returns the greater of the given values

`min`

returns the smaller of the given values

Noteworthy Algorithms (on sorted ranges)

`includes`

returns true if one sequence is a subsequence of another

`set_union`

computes the union of two sets

`set_intersection`

computes the intersection of two sets

`set_difference`

computes the difference between two sets

`merge`

merges two sorted ranges

Removing Algos

`remove`

removes elements that match a value

`remove_if`

removes elements that match a predicate

`unique`

removes consecutive duplicate elements in a range

These algorithms sound like they change the size of their container, but that is impossible. Instead they change the container in-place and return an iterator to the new "end" of the range. You can use the erase member function of vector to change the size of vector to only include the characters in the range.

Attribution

Please ask questions via Piazza

Dr. Joshua Nahum
www.nahum.us
EB 3504

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**