$\begin{array}{c} \mathbf{C++\ STL} \\ \mathbf{Standard\ library\ (Functions\ etc.)} \end{array}$

Nathan Warner



Computer Science Northern Illinois University United States

Contents

1	\mathbf{C} +	+ Strings ($<$ string $>$)	2
	1.1		2
	1.2	Capacity	2
	1.3	Modifiers	2
	1.4	String Operations	3
	1.5	•	3
	1.6	Conversions (These are functions)	3
2	C-s	$strings (< cstring >) \dots $	4
	2.1		4
	2.2	Examination	4
	2.3	Searching	4
	2.4	Error	5
	2.5	Conversion	7
3	Nu	meric to string conversions <string></string>	8
4	\mathbf{Ch}	aracters (<ctype>)</ctype>	8
_	4.1		8
	4.2		8
5	C++		9
6		7. 1	
O	6.1	Style Arrays ($<$ algorithm $>$)	
	6.2	Using lambdas for these functions	
	6.3	Getting position of element from returned iterator	
	6.4	Using shuffle and sample	
	6.5	Sorting operations	
	6.6	Binary search operations (on sorted ranges)	
	6.7	Minimum/maximum operations	
7	Ite	$ m rators < iterator > \dots $	6
	7.1	Iterator generators:	
8	$\mathbf{V}_{\mathbf{P}}$	$\operatorname{\mathbf{ctors}}$	7
G	8.1	Element Access	
	8.2	Capacity	
	8.3	Modifiers	
	8.4	Emplace example <vector></vector>	
0	Set		
9			9
10	Ma	ID	0

1 C++ Strings (<string>)

Interlude. Before we begin with the C++ string methods, there are two things to know.

- size_t: an unsigned integral type, and it's designed to be able to represent the size of any object in bytes
- npos: (Constant) It's the largest possible value representable by the size type of std::string

Note:-

size t is used as the return value for methods such as "find" to indicate unsuccessful

1.1 Element Access

- at(r:size_t pos):→ char& Returns a reference to the character at the specified position. Store return value in char&
- $str.front():\mapsto char\&$ returns reference to the first character.
- str.back():→ char& returns a reference to the last character.
- str.c_str():→ const char* pointing to the null-terminated character array.
- str.data():→ const char* pointing to the underlying character array.

1.2 Capacity

- str.length():→ size_t Returns the number of characters in the string
- str.size():→ size t Returns the number of characters in the string
- str.empty():→ bool Returns true if the string is empty, false otherwise
- resize(r:size_t n, o:char c):→ void Resizes the string to contain n characters.
- capacity():→ size_t Returns the size of the storage space currently allocated
- reserve(size_t new_cap): → void Reserves storage (increases capacity).
- max_size():→ size_t Returns the maximum number of characters the string can hold
- shrink_to_fit():→ void Reduces memory usage by freeing unused memory

1.3 Modifiers

- append(string str):→ std::string& mutated_string String to append. (Has other overloads.) (Doesnt need to be saved in T&)
- $push_back(char c):\mapsto void$ Character to append.
- assign(string str, o:start,o:stop):→ std::string& mutated_string used to replace the current content of the string with a new set of characters. (Has other overloads.)
- insert(size_t pos, string str):→ std::string& mutated_str Position and string to insert. (Has other overloads.)
- replace(size_t pos, size_t len, string str):→ mutated_string Position, length, and string for replacement. (Has other overloads.)
- $swap(string str):\mapsto void String to swap with.$
- pop_back(): → void Removes the last character in the string

1.4 String Operations

- substr(pos, o:len): → string Generate a substr from a string
- copy(char[] dest, o:len, pos):→ size_t (number of characters that were copied) send contents of string to some character array
- find(substr, o:pos): → size_t=npos find a substr from within a string
- rfind(substr, o:pos):
 ⇒ size_t=npos find a substr form withing as string, starting search from the end
 of the string
- find_first_of(substr (or char), o:pos) → size_t=npos Find character in string (public member function)
- find_last_of(substr (or char), o:pos) → size_t=npos Find character in string from the end (public member function)
- find_first_not_of(substr (or char), o:pos) \mapsto size_t=npos Find absence of character in string (public member function)
- find_last_not_of(substr (or char), o:pos) → size_t=npos Find non-matching character in string from the end (public member function)

1.5 Comparison

- compare(o:pos, o:len, str): \mapsto unsigned integral
 - 0: they compare equal
 - <0: Either the value of the first character that does not match is lower in the compared string, or all compared characters match but the compared string is shorter.</p>
 - − >0: Either the value of the first character that does not match is greater in the compared string, or all compared characters match but the compared string is longer.

1.6 Conversions (These are functions)

- $stoi(str, o:idx, o:base):\mapsto int (idx is a pointer to a size t object)$
- $stol(str, o:idx, o:base):\mapsto long$
- $stoul(str, o:idx, o:base):\mapsto unsigned long$
- $stoll(str, o:idx, o:base):\mapsto long long$
- $stoull(str, o:idx, o:base):\mapsto unsigned long long$
- $stof(str, o:idx):\mapsto float$
- $stod(str, o:idx):\mapsto double$
- $stold(str, o:idx):\mapsto long double$

2 C-strings (<cstring>)

2.1 Manipulation

- strncpy(char[] dest, char[] src, size_t n):→ const char* Copy up to n characters from the string src to dest.
- strcpy(char[] dest, char[] src) → const char* Copies the C string pointed by source into the array pointed by destination
- strncat(char[] dest, char[] src, size_t n):→ const char* Append up to n characters from the string src onto the end of dest.
- strcat(char[] dest, char[] src) → const char* Appends a copy of the source string to the destination string.

Note: cpy functions return a const char* comprised of the characters from the src array that were used.

2.2 Examination

- strnlen(char[] src, size_t maxlen) → size_t Return the length of the string (not including the null terminator).
- strlen(char[] src):→ size_t Return the length of the string (not including the null terminator).
- strncmp(char[] src1, char[] src2, size_t n):→ uint (value varies) Compare up to n characters of two strings.
- strcmp(char[| src1, char[| src2) → size_t (value varires) compare two strings

2.3 Searching

- strchr(char[] src, char c):→ char* get pointer to the first occurrence of character c in the string s, or nullptr if c is not found.
- $strrchr(char[] src, char c):\mapsto char^*$ get pointer to the last occurrence of character c in the string s.
- strstr(char[] src, char[] str):→ const char* Return a pointer to the first occurrence of the substring
- strspn(char[] src, char[] charset):→ size_t Returns the length of the initial portion of str1 which consists only of characters that are part of str2.
- strcspn(char[] src, char[] charset):→ size_t Get position of first character found from charset
- strpbrk(char[] src, char[] charset):→ const char* Return string consisting of first match form character set in string onward

2.4 Error

• strerror(errno) → const char* — Get pointer to error message string (we are literally passing in the defined variable errno)

```
std::ifstream inf("file.txt");
if (inf.fail()) { cout << strerrror(errno); } // No such file in directory
(Because that is what the error string is set to after inf.fail())</pre>
```

we also have control over the error we can pass in, errors that are defined in <cerrno>. Rather than leaving it up to error. We have:

E2BIG (C++11) Argument list too long (macro constant)

EACCES (C++11) Permission denied (macro constant)

EADDRINUSE (C++11) Address in use (macro constant)

EADDRNOTAVAIL (C++11) Address not available (macro constant)

EAFNOSUPPORT (C++11) Address family not supported (macro constant)

EAGAIN (C++11) Resource unavailable, try again (macro constant)

EALREADY (C++11) Connection already in progress (macro constant)

EBADF (C++11) Bad file descriptor (macro constant)

EBADMSG (C++11) Bad message (macro constant)

EBUSY (C++11) Device or resource busy (macro constant)

ECANCELED (C++11) Operation canceled (macro constant)

ECHILD (C++11) No child processes (macro constant)

ECONNABORTED (C++11) Connection aborted (macro constant)

ECONNREFUSED (C++11) Connection refused (macro constant)

ECONNRESET (C++11) Connection reset (macro constant)

EDEADLK (C++11) Resource deadlock would occur (macro constant)

EDESTADDRREQ (C++11) Destination address required (macro constant)

EDOM Mathematics argument out of domain of function (macro constant)

EEXIST (C++11) File exists (macro constant)

EFAULT (C++11) Bad address (macro constant)

EFBIG (C++11) File too large (macro constant)

EHOSTUNREACH (C++11) Host is unreachable (macro constant)

EIDRM (C++11) Identifier removed (macro constant)

EILSEQ Illegal byte sequence (macro constant)

EINPROGRESS (C++11) Operation in progress (macro constant)

EINTR (C++11) Interrupted function (macro constant)

EINVAL (C++11) Invalid argument (macro constant)

EIO (C++11) I/O error (macro constant)

EISCONN (C++11) Socket is connected (macro constant)

EISDIR (C++11) Is a directory (macro constant)

ELOOP (C++11) Too many levels of symbolic links (macro constant)

EMFILE (C++11) File descriptor value too large (macro constant)

EMLINK (C++11) Too many links (macro constant)

EMSGSIZE (C++11) Message too large (macro constant)

```
ENAMETOOLONG (C++11) Filename too long (macro constant)
ENETDOWN (C++11) Network is down (macro constant)
ENETRESET (C++11) Connection aborted by network (macro constant)
ENETUNREACH (C++11) Network unreachable (macro constant)
ENFILE (C++11) Too many files open in system (macro constant)
ENOBUFS (C++11) No buffer space available (macro constant)
ENODATA (C++11) (deprecated in C++23) No message is available on the STREAM head read
    queue (macro constant)
ENODEV (C++11) No such device (macro constant)
ENOENT (C++11) No such file or directory (macro constant)
ENOEXEC (C++11) Executable file format error (macro constant)
ENOLCK (C++11) No locks available (macro constant)
ENOLINK (C++11) Link has been severed (macro constant)
ENOMEM (C++11) Not enough space (macro constant)
ENOMSG (C++11) No message of the desired type (macro constant)
ENOPROTOOPT (C++11) Protocol not available (macro constant)
ENOSPC (C++11) No space left on device (macro constant)
ENOSR (C++11)(deprecated in C++23) No STREAM resources (macro constant)
ENOSTR (C++11) (deprecated in C++23) Not a STREAM (macro constant)
ENOSYS (C++11) Function not supported (macro constant)
ENOTCONN (C++11) The socket is not connected (macro constant)
ENOTDIR (C++11) Not a directory (macro constant)
ENOTEMPTY (C++11) Directory not empty (macro constant)
ENOTRECOVERABLE (C++11) State not recoverable (macro constant)
ENOTSOCK (C++11) Not a socket (macro constant)
ENOTSUP (C++11) Not supported (macro constant)
ENOTTY (C++11) Inappropriate I/O control operation (macro constant)
ENXIO (C++11) No such device or address (macro constant)
EOPNOTSUPP (C++11) Operation not supported on socket (macro constant)
EOVERFLOW (C++11) Value too large to be stored in data type (macro constant)
EOWNERDEAD (C++11) Previous owner died (macro constant)
EPERM (C++11) Operation not permitted (macro constant)
EPIPE (C++11) Broken pipe (macro constant)
EPROTO (C++11) Protocol error (macro constant)
EPROTONOSUPPORT (C++11) Protocol not supported (macro constant)
EPROTOTYPE (C++11) Protocol wrong type for socket (macro constant)
ERANGE Result too large (macro constant)
EROFS (C++11) Read-only file system (macro constant)
ESPIPE (C++11) Invalid seek (macro constant)
ESRCH (C++11) No such process (macro constant)
ETIME (C++11)(deprecated in C++23) Stream ioctl() timeout (macro constant)
ETIMEDOUT (C++11) Connection timed out (macro constant)
ETXTBSY (C++11) Text file busy (macro constant)
EWOULDBLOCK (C++11) Operation would block (macro constant)
EXDEV (C++11) Cross-device link (macro constant)
```

2.5 Conversion

- atoi(char[]) Converts a C-string to an int.
- atol(char[]) Converts a C-string to a long.
- atoll(char[]) Converts a C-string to a long long.
- atof(char[]) Converts a C-string to a double.
- strtol(char[], entdptr, base) Converts a C-string to a long int, with error checking and more flexibility with base representations.
- strtoul(char[], entdptr, base) Converts a C-string to an unsigned long int.
- strtoll(char[], entdptr, base) Converts a C-string to a long long int.
- strtoull(char[], entdptr, base) Converts a C-string to an unsigned long long int.
- strtof(char[], entdptr) Converts a C-string to a float.
- strtod(char[], entdptr) Converts a C-string to a double.
- strtold(char[], entdptr) Converts a C-string to a long double.

```
Note: Consider the codeblock
const char* a = "12.322string";
char* ptr;
double val = strtod(a, &ptr); // ptr will hold the value "string"
```

In this example, you can see that we are making use of the endptr optional parameter, <code>endptr</code> is the address to a char pointer. It allows us to send all the string data that is not able to be converted to the requested data type . It allows us to send all the string data that is not able to be converted to the requested data type to this pointer. Only data AFTER the converted data will be sent.

3 Numeric to string conversions <string>

• to_string(n): → string — converts numeric type to c++ string

4 Characters (<ctype>)

4.1 Character Classification

- isalpha(char c): Checks if the character is an alphabet (either uppercase or lowercase).
- **isdigit(char c)**: Checks if the character is a digit (0-9).
- isalnum(char c): Checks if the character is either an alphabet or a digit.
- isspace(char c): Checks if the character is a whitespace character (like space, tab, newline, etc.).
- isupper(char c): Checks if the character is uppercase.
- islower(char c): Checks if the character is lowercase.
- ispunct(char c): Checks if the character is a punctuation character.
- **isprint(char c)**: Checks if the character is printable.
- iscntrl(char c): Checks if the character is a control character.

4.2 Character Conversion:

- toupper(char c): Converts the character to uppercase (if it's lowercase).
- tolower(char c): Converts the character to lowercase (if it's uppercase).

5 C++ Arrays <array>

Note: C++ Arrays are defined

std::array<T,N> name;

- at(size_t)→ T&: Accesses the element at the specified position with bounds checking.
- front() \mapsto T&: Returns a reference to the first element.
- $back() \mapsto T\&:$ Returns a reference to the last element.
- $data() \mapsto T^*$: Returns a direct pointer to the first element in the array, usefull if we want to use our std::array as if it were a c-style array
- $empty() \mapsto bool$: Checks if the container has no elements.
- size() → size t: Returns the number of elements in the container.
- max_size() → size_t: Returns the maximum number of elements the container can hold (same as size).
- fill(value) → void: Fills the array with the specified value.
- $swap(other) \mapsto void$: Swaps the contents of the array with those of *other*.
- **begin()** → **Iterator**: Returns an iterator pointing to the first element.
- cbegin() → Const_Iterator: Returns a const iterator pointing to the first element.
- end()→ Iterator: Returns an iterator pointing to one-past-the-last element.
- cend() → Const_Iterator: Returns a const iterator pointing to one-past-the-last element.
- rbegin() → Reverse_Iterator: Returns a reverse iterator pointing to the last element.
- crbegin()→ Const_Reverse_Iterator: Returns a const reverse iterator pointing to the last element.
- rend()→ Reverse_Iterator: Returns a reverse iterator pointing to one-past-the-first element.
- crend()→ Const_Reverse_Iterator: Returns a const reverse iterator pointing to one-past-the-first element.

6 C-Style Arrays (<algorithm>)

Prereq: Vocab

- An **InputIterator** is a type of iterator that can be used to read data from a sequence of elements. It supports operations like incrementing (to move to the next element in the sequence), dereferencing (to access the value of the element it currently points to), and comparing with other iterators (to check for the end of the sequence). Input iterators are the least powerful, but most widely applicable kind of iterator, as they only require single-pass, read-only access.
- An **ForwardIterator** is a type of iterator that has all the capabilities of an InputIterator but with some additional properties:
 - It can be incremented multiple times and will always give the same sequence of results (multi-pass guarantee).
 - It supports both it++ and ++it operations with the same effect.
 - Two dereferenced copies of a Forward Iterator are guaranteed to reference the same element (if neither is modified).

ForwardIterator is used in contexts where an algorithm might need to pass over a range of elements multiple times.

- A UnaryPredicate is a function or a function object that takes a single argument and returns a bool. It is used to test whether a certain condition is true for the elements in a sequence. The predicate can be a regular function, a lambda expression, or an object of a class that overloads the function call operator.
- A UnaryOperator refers to a function or a function object that takes a single argument and returns some
 value.
- An **OutputIterator** is a type of iterator that can be used to write to a sequence of elements. It's a concept from the C++ Standard Library that defines the requirements for an iterator that can be used to output or write data to a container.

Note: A standard iterator can be used in place of a outputiterator, however, here is how we can use a outputiterator to write contents of an array to the output buffer 1br

std::pair type: std::pair is a class template that provides a way to store two heterogeneous objects as a single unit. The way we access the elements is with .first and .second

6.1 Non-Modifying

- all_of(InputIterator, InputIterator, UnaryPredicate) → bool: Returns true if the predicate is true for all elements in the given range.
- any_of(InputIterator, InputIterator, UnaryPredicate) → bool: Returns true if the predicate is true for any element in the range.
- none_of(InputIterator, InputIterator, UnaryPredicate) → bool: Returns true if the predicate is false for all elements in the range.
- for_each(InputIterator, InputIterator, Function) → Function: Applies a function to each element in the range.

- count(InputIterator, InputIterator, const T&) → size_t: Counts elements equal to the specified value.
- count_if(InputIterator, InputIterator, UnaryPredicate) → size_t: Counts elements for which the predicate is true.
- mismatch(InputIterator1, InputIterator1, InputIterator2) \mapsto pair<InputIterator1, InputIterator2>: Finds the first position where two ranges differ.
- find(InputIterator, InputIterator, const T&) \mapsto InputIterator: Finds the first element equal to the specified value.
- find_if(InputIterator, InputIterator, UnaryPredicate) → InputIterator: Finds the first element for which the predicate is true.
- find_if_not(InputIterator, InputIterator, UnaryPredicate) → InputIterator: Finds the first element for which the predicate is false.
- find_end(InputIterator, InputIterator, InputIterator, InputIterator) \mapsto InputIterator: Finds the last occurrence of a subsequence.
- find_first_of(InputIterator, InputIterator, InputIterator, InputIterator) \mapsto InputIterator: Finds the first element that matches any element in another range.
- adjacent_find(InputIterator, InputIterator, o:UnaryPredicate) → InputIterator: Finds the first two adjacent items that are equal (or satisfy a given predicate).
- search(InputIterator, InputIterator, InputIterator, InputIterator) \mapsto InputIterator: Searches for the first occurrence of a subsequence.
- search_n(InputIterator, InputIterator, size_t, const T&) → InputIterator: Searches for a sequence of repeated elements.

8 Note:

The for each function returns the function that was used to map each element.

- copy(InputIterator, InputIterator, OutputIterator) → OutputIterator: Copies a range of elements to a new location.
- copy_if(InputIterator, InputIterator, OutputIterator, UnaryPredicate) → OutputIterator: Copies elements satisfying a condition to a new location.
- copy_n(InputIterator, Size, OutputIterator) → OutputIterator: Copies a number of elements to a new location.
- $copy_backward(BidirectionalIterator1, BidirectionalIterator1, BidirectionalIterator2) \mapsto BidirectionalIterator2 \mapsto BidirectionalIterator2 \mapsto BidirectionalIterator3 \mapsto Bidi$
- move(InputIterator, InputIterator, OutputIterator) → OutputIterator: Moves a range of elements to a new location.
- move_backward(BidirectionalIterator1, BidirectionalIterator1, BidirectionalIterator2) \mapsto BidirectionalIterator1. Moves elements in reverse order.
- fill(ForwardIterator, ForwardIterator, const T&) \mapsto void: Assigns a value to all elements in a range.
- fill_n(OutputIterator, Size, const T&) \mapsto OutputIterator: Assigns a value to a number of elements.
- transform(InputIterator, InputIterator, OutputIterator, UnaryOperation) \mapsto OutputIterator: Applies a function to a range of elements.
- generate(ForwardIterator, ForwardIterator, Generator) → void: Fills a range with the results of successive function calls.
- generate_n(OutputIterator, Size, Generator) → OutputIterator: Fills a range with N results of successive function calls.
- remove(InputIterator, InputIterator, const T&) → InputIterator: Removes elements equal to a
 value.
- remove_if(InputIterator, InputIterator, UnaryPredicate) \mapsto InputIterator: Removes elements satisfying a condition.
- remove_copy(InputIterator, InputIterator, OutputIterator, const $T\&) \mapsto OutputIterator$: Copies elements not equal to a value.
- remove_copy_if(InputIterator, InputIterator, OutputIterator, UnaryPredicate) \mapsto OutputIterator: Copies elements not satisfying a condition.
- replace(InputIterator, InputIterator, const T&, const T&) → void: Replaces all values equal to a specified value.
- replace_if(InputIterator, InputIterator, UnaryPredicate, const $T\&) \mapsto \text{void}$: Replaces values satisfying a condition.
- replace_copy(InputIterator, InputIterator, OutputIterator, const T&, const T&) \mapsto OutputIterator: Copies and replaces values.
- replace_copy_if(InputIterator, InputIterator, OutputIterator, UnaryPredicate, const $T\&) \mapsto$ OutputIterator: Copies and replaces values based on a condition.
- $swap(T\&, T\&) \mapsto void$: Swaps the values of two objects.
- swap_ranges(ForwardIterator1, ForwardIterator1, ForwardIterator2) → ForwardIterator2: Swaps two ranges of elements.
- iter swap(ForwardIterator1, ForwardIterator2) \mapsto void: Swaps elements pointed to by two iterators.
- reverse(BidirectionalIterator, BidirectionalIterator) → void: Reverses the order of elements in a range.

- reverse_copy(BidirectionalIterator, BidirectionalIterator, OutputIterator) \mapsto OutputIterator: Creates a reversed copy of a range.
- rotate(ForwardIterator, ForwardIterator, ForwardIterator) \mapsto ForwardIterator: Rotates the order of elements in a range.
- rotate_copy(ForwardIterator, ForwardIterator, ForwardIterator, OutputIterator) \mapsto OutputIterator: Copies and rotates a range of elements.
- shift_left(ForwardIterator, ForwardIterator, Size) → ForwardIterator: Shifts elements in a range to the left.
- shift_right(ForwardIterator, ForwardIterator, Size) → ForwardIterator: Shifts elements in a range to the right.
- random_shuffle(RandomAccessIterator, RandomAccessIterator) → void: Randomly re-orders elements in a range.
- shuffle(RandomAccessIterator, RandomAccessIterator, URNG&) \mapsto void: Randomly re-orders elements in a range using a generator.
- sample(InputIterator, InputIterator, OutputIterator, Size, URNG&) → OutputIterator: Selects N random elements from a sequence.
- unique(ForwardIterator, ForwardIterator) → ForwardIterator: Removes consecutive duplicate elements in a range.
- unique_copy(InputIterator, InputIterator, OutputIterator) → OutputIterator: Creates a copy of a range without consecutive duplicates.
- partition(BidirectionalIterator, BidirectionalIterator, UnaryPredicate)→ Iterator: Partition range in two
- partition_copy(BidirectionalIterator, BidirectionalIterator, UutputIterator) \mapsto OutputIterator: Partition range in two
- $stable_partition(BidirectionalIterator, BidirectionalIterator, UnaryPredicate) \mapsto Iterator: Partition range in two stable ordering$

6.2 Using lambdas for these functions

```
int arr[] = {1,2,3};
std::for_each(arr, arr+3, [](int& a) -> int { return ++x; });
```

6.3 Getting position of element from returned iterator

Concept 1: If we subtract the iterator (which is the address of the array element) from the beginning of the array, we get the distance between the two pointers (so the index position of the iterator value)

```
idx = it - std::begin(array);
```

6.4 Using shuffle and sample

For these functions we need two components, a seed and a random engine

```
#include <chrono>
#include <random>

unsigned seed std::chrono::system_clock::now().time_since_epoch().count();

std::default_random_engine engine(seed);
```

We then use the engine variable in place of URNG&

6.5 Sorting operations

- is_sorted(ForwardIterator, ForwardIterator) → bool: Checks whether a range is sorted into ascending order.
- is_sorted_until(ForwardIterator, ForwardIterator) → ForwardIterator: Finds the largest sorted subrange.
- sort(RandomAccessIterator, RandomAccessIterator) → void: Sorts a range into ascending order.
- partial_sort(RandomAccessIterator, RandomAccessIterator, RandomAccessIterator, comp)
 → void: Sorts the first N elements of a range.
- partial_sort_copy(InputIterator, InputIterator, RandomAccessIterator, RandomAccessIterator, comp) → RandomAccessIterator: Copies and partially sorts a range of elements.
- stable_sort(RandomAccessIterator, RandomAccessIterator) → void: Sorts a range of elements while preserving order between equal elements.
- nth_element(RandomAccessIterator, RandomAccessIterator, RandomAccessIterator) → void: Partially sorts the given range making sure that it is partitioned by the given element.

6.6 Binary search operations (on sorted ranges)

- lower_bound(ForwardIterator, ForwardIterator, const T&) → ForwardIterator: Returns an iterator to the first element in the range [first, last) that is not less than (i.e., greater or equal to) the given value.
- upper_bound(ForwardIterator, ForwardIterator, const T&) \mapsto ForwardIterator: Returns an iterator to the first element in the range [first, last) that is greater than the given value.
- binary_search(ForwardIterator, ForwardIterator, const T&) → bool: Determines if an element equal to the given value exists within the range [first, last).
- equal_range(ForwardIterator, ForwardIterator, const T&) \mapsto pair<ForwardIterator, ForwardIterator>: Returns a range containing all elements equivalent to the given value in the range [first, last).

6.7 Minimum/maximum operations

- $max(const \ T\&, const \ T\&) \mapsto T$: Returns the greater of the two given values.
- max_element(ForwardIterator, ForwardIterator) → ForwardIterator: Returns an iterator to the largest element in the range [first, last).
- $min(const T\&, const T\&) \mapsto T$: Returns the smaller of the two given values.
- min_element(ForwardIterator, ForwardIterator) → ForwardIterator: Returns an iterator to the smallest element in the range [first, last).
- $minmax(const\ T\&,\ const\ T\&) \mapsto pair<T$, T>: Returns a pair consisting of the smaller and larger of the two elements.
- minmax_element(ForwardIterator, ForwardIterator) → pair<ForwardIterator, ForwardIterator>: Returns a pair of iterators to the smallest and the largest elements in the range [first, last).
- clamp(const T&, const T&, const T&) → T: Clamps a value between a pair of boundary values. If the value is less than the lower bound, it returns the lower bound. If it's greater than the upper bound, it returns the upper bound. Otherwise, it returns the value itself.

7 Iterators <iterator>

- advance(Iterator&, Distance) → void: Advances the iterator by the given distance.
- distance(InputIterator, InputIterator) \mapsto typename iterator_traits<InputIterator>::difference_type: Returns the number of steps between two iterators.
- begin(Container&) → Iterator: Returns an iterator pointing to the first element of the container.
- end(Container&) → Iterator: Returns an iterator pointing to the past-the-end element of the container.
- prev(BidirectionalIterator) → BidirectionalIterator: Returns an iterator pointing to the element immediately before the given iterator.
- next(ForwardIterator) → ForwardIterator: Returns an iterator pointing to the element immediately after the given iterator.

7.1 Iterator generators:

- back_inserter(Container&) → BackInsertIterator: Constructs a back-insert iterator that appends new elements to the end of the specified container.
- front_inserter(Container&) → FrontInsertIterator: Constructs a front-insert iterator that prepends new elements to the beginning of the specified container.
- inserter(Container&, Iterator) → InsertIterator: Constructs an insert iterator for inserting new elements into the container at the position specified by the given iterator.
- make_move_iterator(Iterator) → MoveIterator: Converts a regular iterator into a move iterator, causing the elements to be moved rather than copied.

8 Vectors

8.1 Element Access

- at(size_type n) \mapsto T& Accesses the element at the specified position n in the container, with bounds checking. Throws an out-of-range exception if n is not within the range of the container.
- front() → T& Accesses the first element in the container. Undefined behavior if the container is empty.
- back() → T& Accesses the last element in the container. Undefined behavior if the container is empty.
- data() → T*: Provides direct access to the underlying array in the container. Returns a pointer to the array.

8.2 Capacity

- empty() → bool: Checks whether the container is empty. Returns true if the container is empty, false otherwise.
- $size() \mapsto size_type$: Returns the number of elements in the container.
- max_size() → size_type: Returns the maximum number of elements that the container can hold.
- capacity() \mapsto size_type: Returns the number of elements that can be held in currently allocated storage.
- reserve(size_type n) → void: Reserves storage to make the container capable of holding at least n elements. If n is greater than the current capacity(), new storage is allocated, otherwise the method does nothing.
- shrink_to_fit() → void: Reduces memory usage by freeing unused memory. This is a non-binding request to reduce capacity() to size(). The implementation is allowed to optimize otherwise or ignore the request.

8.3 Modifiers

- push_back(const T& value) → void Adds an element to the end of the container.
- pop_back() → void: Removes the last element from the container.
- append_range(InputIterator first, InputIterator last) → void: Adds a range of elements from first to last to the end of the container.
- insert(iterator pos, const T& value) → iterator: Inserts an element at the position given by pos.
- insert_range(iterator pos, InputIterator first, InputIterator last) → iterator : Inserts a range of elements from first to last at the position given by pos.
- clear() → void: Clears the contents of the container. After this call, size() returns zero.
- erase(iterator pos) → iterator: Erases the element at the position given by pos. Returns an iterator following the last removed element.
- emplace(iterator pos, Args... args) → iterator: Constructs an element in-place at the position given by pos. The arguments args are forwarded to the constructor of the element type.
- emplace_back(Args... args) \mapsto T&: Constructs an element in-place at the end of the container. The arguments args are forwarded to the constructor of the element type.
- resize(size_type sz) → void: Changes the number of elements stored. If sz is smaller than the current size, the container is reduced to its first sz elements.
- swap(Container& other) → void: Swaps the contents with *other* container. The swap operation is generally very fast and never fails.

8.4 Emplace example <vector>

```
#include <iostream>
#include <vector>
3 #include <string>
5 class MyClass {
   public:
       MyClass(int x, double y) : x(x), y(y) {}
       void print() const {
            std::cout << "MyClass(" << x << ", " << y << ")\n";
10
11
12
   private:
       int x;
14
        double y;
15
   };
16
17
   int main() {
18
       std::vector<MyClass> vec;
20
       // Construct MyClass objects directly at the end of the vector
       vec.emplace_back(10, 3.14);
^{22}
        vec.emplace_back(20, 6.28);
23
       for (const auto& obj : vec) {
25
            obj.print();
27
       return 0;
29
  }
30
```

9 Set

10 Map