# Chapter 1 – Containers

## Section 1.1 – Standard Sequence Containers

Standard sequence containers implement data structures that can be sequentially accessed. These encompass vectors, lists, and deques.

### 1.1.0 Member Functions for All Sequence Containers

#### Overview

These member functions are a part of all sequence containers and have identical purpose.

#### Initialization

CONTAINER\_NAME()

Empty container.

Valid container names are vector, list, and deque.

CONTAINER\_NAME(size\_type n, const T& value = T())

Sequential container of ‘n’ elements of value ‘value’ (defaulted to default value of the type T).

Valid container names are vector, list, and deque.

1. CONTAINER\_NAME(const CONTAINER\_NAME& other)

2. operator=

Sequential container that is a deep copy of ‘other’.

Valid container names are vector, list, and deque.

#### Access

operator[]

Accesses specified element.

front() back()

Accesses the first / last element.

#### Iterators

begin() rbegin()

Returns an iterator / reversed-iterator to the beginning / end.

begin faces forwards (from first to last), while rbegin faces backwards (from last to first)

end() rend()

Returns an iterator / reversed-iterator to the end / beginning.

#### Capacity

empty()

Returns boolean, indicating if the container is empty (true) or not empty (false).

size()

Returns the number of elements as an unsigned long long.

### 1.1.1 Vector

#### Overview

Vectors represent contiguous, dynamic arrays. Available through: #include <vector>

#### Modifiers

clear()

Clears the contents.

insert(iterator position, const T& value)

Inserts element of value ‘value’ at iterator ‘position’.

1. erase(iterator position)

2. erase(iterator first, iterator last)

Removes elements from [1] ‘position’ or from [2] ‘first’ (inclusive)to ‘last’ (exclusive).

push\_back(const T& value);

Appends the given element.

pop\_back()

Removes the last element.

resize(size\_type count) resize(size\_type count, T value = T())

Resize the container to contain ‘count’ elements, [optional] with a value to set new elements to.

swap(vector& other)

Exchanges the contents of the vector with those of vector ‘other’.

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the vector based on lexicographical order.

### 1.1.2 List

#### Overview

Lists represent doubly-linked list. Available through: #include <list>

#### Modifiers

clear()

Clears the contents.

insert(iterator position, const T& value)

Inserts element of value ‘value’ at iterator ‘position’.

1. erase(iterator position)

2. erase(iterator first, iterator last)

Removes elements from [1] ‘position’ or from [2] ‘first’ (inclusive)to ‘last’ (exclusive).

push\_back(const T& value) push\_front(const T& value)

Appends / prepends the given element.

pop\_back() pop\_front()

Removes the last / first element.

resize(size\_type count) (size\_type count, T value = T())

Resize the container to contain ‘count’ elements, [optional] with a value to set new elements to.

swap(list& other)

Exchanges the contents of the container with those of container ‘other’.

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the list based on lexicographical order.

1. merge(list& other)

2. merge(list& other, Compare comp)

Merges two sorted lists into one sorted list, [optional] with an comparator for merging the lists.

remove(const T& value) remove\_if(UnaryPredicate p)

Removes all elements satisfying the criteria (equal to some value OR satisfies some operator ‘p’).

reverse()

Reverses the order of the elements in the container.

unique()

Removes all consecutive duplicate elements from the container, leaving only unique values in list.

sort() sort(Compare comp)

Sorts the elements in ascending order, [optional] with a comparator for sorting the list.

splice(const\_iterator pos, list& other)

splice(const\_iterator pos, list& other, const\_iterator it)

splice(const\_iterator pos, list& other, const\_iterator A, const\_iterator B)

Transfers elements from one list to another. Elements are inserted before the element pointed at by ‘pos’.

### 1.1.3 Deque

#### Overview

Deques represent double-ended queues, an indexed seuqnece container that allows fast insertion and deletion at both its beginning and its end. Available through: #include <deque>

#### Modifiers

clear()

Clears the contents.

insert(iterator position, const T& value)

Inserts element of value ‘value’ at iterator ‘position’.

1. erase(iterator position)

2. erase(iterator first, iterator last)

Removes elements from [1] ‘position’ or from [2] ‘first’ (inclusive)to ‘last’ (exclusive).

push\_back(const T& value) push\_front(const T& value)

Appends / prepends the given element.

pop\_back() pop\_front()

Removes the last / first element.

resize(size\_type count) (size\_type count, T value = T())

Resize the container to contain ‘count’ elements, [optional] with a value to set new elements to.

swap(deque& other)

Exchanges the contents of the container with those of container ‘other’.

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the deque based on lexicographical order.

## Section 1.2 – Adapted Sequence Containers

Adapted sequence containers implement data structures that can be accessed in some specialized sequential order. These encompass stacks, queues, and priority queues.

### 1.2.1 Stack

#### Overview

Stacks represent the LIFO (last-in, first-out) data structure.

Available through: #include <stack>

#### Initialization

stack()

Empty stack.

1. stack(const SEQUENTIAL\_CONTAINER\_NAME& other)

2. operator=

Stack that is a deep copy of ‘other’.

Valid container names are vector, list, deque, stack, and queue.

#### Access

top()

Accesses the top element of the stack.

#### Capacity

empty()

Returns boolean, indicating if the container is empty (true) or not empty (false).

size()

Returns the number of elements as an unsigned long long.

#### Modifiers

push(const value\_type& value)

Inserts element at the top.

pop()

Removes element from the top.

swap(stack& other)

Exchanges the contents of the stack with those of ‘other’.

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the vector based on lexicographical order.

### 1.2.2 Queue

#### Overview

Queues represent the FIFO (first-in, first-out) data structure.

Available through: #include <queue>

#### Initialization

queue()

Empty queue.

1. queue(const SEQUENTIAL\_CONTAINER\_NAME& other)

2. operator=

Queue that is a deep copy of ‘other’.

Valid container names are vector, list, deque, stack, and queue.

#### Access

front()

Accesses the first element.

back()

Accesses the last element.

#### Capacity

empty()

Returns boolean, indicating if the container is empty (true) or not empty (false).

size()

Returns the number of elements as an unsigned long long.

#### Modifiers

push(const value\_type& value)

Inserts element at the back (as the last element).

pop()

Removes the first element.

swap(queue& other)

Exchanges the contents of the queue with those of ‘other’.

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the vector based on lexicographical order.

### 1.2.2 Priority Queue

#### Initialization

priority\_queue()

priority\_queue(Compare& cmp = Compare(), Container& c = container())

Empty priority queue.

Make empty int priority queue: priority\_queue<int>

Make empty int min-heap: priority\_queue<int, vector<int>, greater<int>>

1. priority\_queue(const priority\_queue& other)

2. operator=

Priority queue that is a deep copy of ‘other’.

#### Access

top()

Accesses the top element, as determined by the priority queue comparator.

#### Modifiers

push(const value\_type& value)

Inserts element and sorts the underlying container.

pop()

Removes the top element.

swap(priority\_queue& other)

Exchanges the contents of the queue with those of ‘other’.

## Section 1.3 – Ordered Associative Containers

Ordered associative containers implement sorted data structures that can be quickly searched in O(logn) time. These encompass ordered sets and ordered maps.

### 1.3.0 Member Functions for All Ordered Associative Containers

#### Iterators

begin() rbegin()

Returns an iterator / reversed-iterator to the beginning / end.

begin faces forwards (from first to last), while rbegin faces backwards (from last to first)

end() rend()

Returns an iterator / reversed-iterator to the end / beginning.

#### Capacity

empty()

Returns boolean, indicating if the container is empty (true) or not empty (false).

size()

Returns the number of elements as an unsigned long long.

#### Lookup

count(const Key& key)

Returns the number of elements matching the key value.

find(const Key& key)

Returns iterator to the element matching the key value. If no element matches, returns end().

lower\_bound(const Key& key)

Returns iterator to the first element that is not less than the key value.

upper\_bound(const Key& key)

Returns iterator to the first element that is greater than the key value.

equal\_range(const Key& key)

Returns a pair of iterators, indicating the range of all elements matching the key value.

#### Modifiers

clear()

Clears the contents.

insert(const T& value) insert(iterator position, const T& value)

Inserts element of value ‘value’, [optionally] at iterator ‘position’.

1. erase(iterator position)

2. erase(iterator first, iterator last)

3. erase(const key\_type& key)

Removes elements from [1] ‘position’ or from [2] ‘first’ (inclusive)to ‘last’ (exclusive), or [3] a single element matching the key value.

swap(ASSOCIATIVE\_CONTAINER\_NAME& other)

Exchanges the contents of the associative container with those of ‘other’.

### 1.3.1 Ordered Set

#### Overview

Ordered sets represent a sorted collection of unique objects of some type.

Shares majority of functionality with ordered maps, except for element access/assignment.

Available through: #include <set>

#### Initialization

set()

Empty set.

1. set(const set& other)

2. operator=

Set that is a deep copy of ‘other’.

set(iterator first, iterator last)

Constructs the container with the contents from ‘first’ (inclusive) to ‘last’ (exclusive).

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the vector based on lexicographical order.

### 1.3.2 Ordered Map

#### Overview

Ordered maps represent a sorted collection of key-value pairs, sorted by keys, which are unique.

Available through: #include <map>

#### Initialization

map()

Empty map.

1. map(const map& other)

2. operator=

Map that is a deep copy of ‘other’.

#### Access and Assignment

operator[]

Accesses or inserts the specified element. Returns reference to the value that is mapped to the key value, performing an insertion if such a key does not already exist.

Associate value 3 with key 5: map<int, int> m; m[5] = 3;

#### Operators

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the vector based on lexicographical order.

## Section 1.4 – Unordered Associative Containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched in O(1) time. These encompass unordered sets and unordered maps.

### 1.4.0 Member Functions for All Unordered Associative Containers

#### Iterators

begin() rbegin()

Returns an iterator / reversed-iterator to the beginning / end.

begin faces forwards (from first to last), while rbegin faces backwards (from last to first)

end() rend()

Returns an iterator / reversed-iterator to the end / beginning.

#### Capacity

empty()

Returns boolean, indicating if the container is empty (true) or not empty (false).

size()

Returns the number of elements as an unsigned long long.

#### Lookup

count(const Key& key)

Returns the number of elements matching the key value.

find(const Key& key)

Returns iterator to the element matching the key value. If no element matches, returns end().

equal\_range(const Key& key)

Returns a pair of iterators, indicating the range of all elements matching the key value.

#### Modifiers

clear()

Clears the contents.

insert(const T& value) insert(iterator position, const T& value)

Inserts element of value ‘value’, [optionally] at iterator ‘position’.

1. erase(iterator position)

2. erase(iterator first, iterator last)

3. erase(const key\_type& key)

Removes elements from [1] ‘position’ or from [2] ‘first’ (inclusive)to ‘last’ (exclusive), or [3] a single element matching the key value.

swap(ASSOCIATIVE\_CONTAINER\_NAME& other)

Exchanges the contents of the associative container with those of ‘other’.

### 1.4.1 Unordered Set

#### Overview

Unordered sets represent an unsorted collection of unique objects of some type.

Shares majority of functionality with unordered maps, except for element access/assignment.

Available through: #include <unordered\_set>

#### Initialization

unordered\_set()

Empty unordered set.

1. unordered\_set(const unordered\_set& other)

2. operator=

Set that is a deep copy of ‘other’.

unordered\_set(iterator first, iterator last)

Constructs the container with the contents from ‘first’ (inclusive) to ‘last’ (exclusive).

#### Operators

operator==, operator!=

Compares contents of the unordered set based on lexicographical order.

### 1.4.2 Unordered Map

#### Overview

Unordered maps represent an unsorted collection of key-value pairs, where the keys are unique.

Available through: #include <unordered\_map>

#### Initialization

unordered\_map()

Empty map.

1. unordered\_map(const unordered\_map& other)

2. operator=

Unordered map that is a deep copy of ‘other’.

#### Access and Assignment

operator[]

Accesses or inserts the specified element. Returns reference to the value that is mapped to the key value, performing an insertion if such a key does not already exist.

#### Operators

operator==, operator!=

Compares contents of the unordered map based on lexicographical order.

# Chapter 2 – I/O

## Section 2.1 – Input and Output Stream

Program data is received through standard input from cin objects. Results are presented through standard output from cout objects. Available through: #include <iostream>

### Standard Input

#### Overview

For competitions, program data is received through standard input from cin objects. As a result, all data is fed through cin, which are standard istream objects.

operator>>(value\_type& value)

Extracts and parses characters from the input stream, interpreting them as the representation of a value of the proper type. Known as the extraction operator.

Read until End-of-File: int n; while (cin >> n) { /\* Do stuff with input \*/ }

### Standard Output

#### Overview

For competitions, results are presented through standard output from cout objects, which are standard ostream objects.

operator<<(value\_type& value)

Generates a sequence of characters with the representation of the given value and inserts them into the output stream. Known as the insertion operator.

endl

Inserts a new-line character and flushes the stream.

## Section 2.2 – Input and Output Manipulation

Standard manipulators exist to adjust how data is extracted and inserted from the stream.

Available through: #include <iomanip>

### Parametric Manipulators

#### Overview

For competitions, results may have to be presented in unique manners. These parametric manipulators covered will adjust standard output from cout objects.

setbase(int base)

Sets the base-field to three possible bases: octal (base = 8), decimal (base = 10), hex (base = 16).

If none of these three are used, the base is defaulted to decimal.

Note: Output should be adjusted by the programmer if a different base from these three is desired.

setprecision(int n)

Sets the decimal precision to be used to format floating-point values on output operations.

Note: Does not round decimals. Instead, prints up to the specified decimal location.

# Chapter 3 – Strings

## Section 3.1 – Character Handling

Character handling considers the classification and transformation of individual characters. Available through: #include <cctype>

### 3.1.1 Character Classification

#### Overview

Characters are classifiable based on criteria. These are competition-relevant classifications.

isalnum(char ch)

Returns non-zero (true) if character is either a digit or a letter, and zero (false) otherwise.

isalpha(char ch)

Returns non-zero (true) if character is a letter, and zero (false) otherwise.

isblank(char ch)

Returns non-zero (true) if character is a space, and zero (false) otherwise.

iscntrl(char ch)

Returns non-zero (true) if character is a control character, and zero (false) otherwise.

An example of a control character would be newline, ‘\n’.

ispunct(char ch)

Returns non-zero (true) if character is a punctuation, and zero (false) otherwise.

Punctuation characters are those that can be visibly printed to a console, but are not alphanumeric.

islower(char ch)

Returns non-zero (true) if character is a lowercase letter, and zero (false) otherwise.

isupper(char ch)

Returns non-zero (true) if character is an uppercase letter, and zero (false) otherwise.

isxdigit(char ch)

Returns non-zero (true) if character is a hexadecimal character, and zero (false) otherwise.

### 3.1.2 Character Conversion

#### Overview

Two simple character conversions exist within the standard library, tolower and toupper.

tolower(char ch)

Returns a lowercase version of ‘ch’.

toupper(char ch)

Returns an uppercase version of ‘ch’.

## Section 3.2 – Basic Strings

Basic strings, in essence, implement dynamic sequences of char-like objects. Similar to those of sequence containers, the elements of a basic string are stored contiguously. For competitions, the only basic string to consider is std::basic\_string<char>, or simply std::string.

### 3.2.1 String

#### Overview

Strings represent dynamic sequences of characters. It is the most commonly used version of the basic string, and the only version to consider for competition.

Available through:

#### Initialization

string()

Empty string.

string(size\_type n, CharT ch)

String of ‘n’ characters with value ‘ch’.

1. string(const string& other)

2. operator=

String that is a deep copy of ‘other’.

#### Access

operator[]

Accesses specified character.

front() back()

Accesses the first / last character.

data()

Returns a pointer to the first character.

#### Iterators

begin() rbegin()

Returns an iterator / reversed-iterator to the beginning / end.

begin faces forwards (from first to last), while rbegin faces backwards (from last to first)

end() rend()

Returns an iterator / reversed-iterator to the end / beginning.

#### Capacity

empty()

Returns boolean, indicating if the container is empty (true) or not empty (false).

size()

Returns the number of elements as an unsigned long long.

#### Operations

clear()

Clears the contents.

1. insert(iterator position, const CharT ch)

2. insert(size\_type index, size\_type count, CharT ch)

3. insert(size\_type index, const string& str)

[1] Inserts characters of value ‘ch’ at iterator ‘position’.

[2] Inserts ‘count’ characters of value ‘ch’ starting at ‘index’.

[3] Inserts string ‘str’ starting at ‘index’.

1. erase(iterator position)

2. erase(iterator first, iterator last)

Removes characters from [1] ‘position’ or from [2] ‘first’ (inclusive)to ‘last’ (exclusive).

push\_back(CharT ch);

Appends the given character.

pop\_back()

Removes the last character.

operator+=

Appends the given character or string.

resize(size\_type count) resize(size\_type count, T value = T())

Resize the container to contain ‘count’ elements, [optional] with a value to set new elements to.

swap(vector& other)

Exchanges the contents of the vector with those of vector ‘other’.

compare(const string& str)

Returns -1 if str > this (either by length or lexicographical comparison) and vice versa for returning 1. Returns 0 if str equals this for both length and lexicographical comparison.

substr(size\_type pos = 0, size\_type count = 0)

Returns a substring from the given ‘pos’ (inclusive) to ‘pos + count’ (exclusive).

If the upper bound goes out of range, then a substring is generated up to the end of the string.

find(const string& str, size\_type pos = 0) find(CharT ch, size\_type pos = 0)

rfind(..., size\_type pos = npos) rfind(..., size\_type pos = npos)

Finds the first / last substring equal to the given string or character, beginning at ‘pos’.

Note that find works forwards, rfind searches backwards. Returns string::npos if not found.

find\_first\_of(const string& str, size\_type pos = 0)

find\_first\_of(Char Tch, size\_type pos = 0)

find\_first\_not\_of(const string& str, size\_type pos = 0)

find\_first\_not\_of(Char Tch, size\_type pos = 0)

find\_last\_of(const string& str, size\_type pos = npos)

find\_last\_of(Char Tch, size\_type pos = npos)

find\_last\_not\_of(const string& str, size\_type pos = npos)

find\_last\_not\_of(Char Tch, size\_type pos = npos)

Finds first occurrence of / first non-occurrence of / last occurrence of / last non-occurrence of the given string or character. Returns string::npos if not found.

Note that …\_first\_... works forwards and …\_last\_... works backwards.

#### Operators

operator+

Concatenates two strings or a string and a char.

operator==, operator!=, operator<, operator<=, operator>, operator>=

Compares contents of the vector based on lexicographical order.

#### Numeric Conversions

stoi(const string& str, size\_type pos = 0, int base = 10)

stol(const string& str, size\_type pos = 0, int base = 10)

stoll(const string& str, size\_type pos = 0, int base = 10)

Returns the result of converting string to a signed integer / long / long long.

stoul(const string& str, size\_type pos = 0, int base = 10)

stoull(const string& str, size\_type pos = 0, int base = 10)

Returns the result of converting string to a unsigned long (unsigned integer) / long long.

stof(const string& str, size\_type pos = 0, int base = 10)

stod(const string& str, size\_type pos = 0, int base = 10)

stold(const string& str, size\_type pos = 0, int base = 10)

Returns the result of converting string to a float / double / long double.

to\_string(value\_type value)

Returns the result of the conversion of the numeric value to a string.

# Chapter 4 – Algorithms

## Section 4.1 – Numerical Computations

### 4.1.1 Common Mathematical Functions

#### Overview

This library consists of common mathematical functions. Available through: #include <cmath>

#### Exponential Functions

exp(value\_type arg)

Computes e (Euler’s number, 2.7182818…) raised to the given power.

log(value\_type arg) log10(value\_type arg) log2(value\_type arg)

Computes the natural (base-e) / common (base-10) / base-2 logarithm of the given number.

#### Power Functions

pow(value\_type base, value\_type exp)

Computes the value of ‘base’ raised to the power of ‘exp’.

sqrt(value\_type arg) cbrt(value\_type arg)

Computes the square / cubic root of the given number.

hypot(value\_type A, value\_type B)

Computes the square root of the sum of the squares of the two given numbers.

sin(value\_type x) cos(value\_type x) tan(value\_type x)

asin(value\_type x) acos(value\_type x) atan(value\_type x)

Computes sine / cosine / tangent / arc-sine / arc-cosine / arc-tangent of the given number.

#### Rounding Functions

ceil(value\_type arg) floor(value\_type arg)

Computes the smallest integer not less than / not greater than the given number.

round(value\_type arg) lround(value\_type arg) llround(value\_type arg)

Computes the nearest integer to the given number as a floating-point / long / long long.

trunc(value\_type arg)

Computes the nearest integer not greater in magnitude than the given number.

### 4.1.2 Numeric Algorithms

#### Overview

This library consists of common numeric algorithms. Available through: #include <numeric>

#### Numeric Operations

accumulate(iterator first, iterator last, value\_type value)

Computes the sum of the given value and the elements from ‘first’ (inclusive) to ‘last’ (exclusive).

adjacent\_difference(iterator first, iterator last, iterator output\_first)

Computes the difference between adjacent items in a container, placing results into the location indicated by output iterator.

## Section 4.2 – General Algorithms

### 4.1.1 Non-Modifying Sequence Operations

all\_of(iterator first, iterator last, UnaryPredicate p)

any\_of(iterator first, iterator last, UnaryPredicate p)

none\_of(iterator first, iterator last, UnaryPredicate p)

Checks if the predicate returns true for all / any / none of the elements in the range [first, last).

for\_each(iterator first, iterator last, UnaryFunction f)

Applies the given function to the elements in the range [first, last).

count(iterator first, iterator last, const T& value)

count\_if(iterator first, iterator last, UnaryPredicate p)

Counts elements that are equal to the value / returns true for the predicate from [first, last).

find(iterator first, iterator last, const T& vaue)

find\_if(iterator first, iterator last, UnaryPredicate p)

find\_if\_not(iterator first, iterator last, UnaryPredicate p)

Returns the first element that equals value / returns (true / false) for predicate from [first, last).

### 4.1.2 Modifying Sequence Operations