Programming Assignment 5

Set, and Map

# 

[Approved Includes](#_jngx5a99g2jd)

[Code Coverage](#_776g0839yxxo)

[Starter Code](#_jmrk4oe3zs75)

[Files to Submit](#_van3nv2hgzl6)

[Task 1: Set](#_hutw5taun68d)

[Requirements](#_s8k1975o5yba)

[Files](#_j0z3mne6g52h)

[Class](#_1j8b25wynm5v)

[Member types (private)](#_x5g5vluuhfh1)

[Member types (public)](#_i9qk1nd11nlk)

[Functions (public)](#_irh9q2qguf30)

[Constructors & Rule of Three](#_32qdffc7ce6m)

[Element Access](#_h8nep8djt6vd)

[Iterators](#_dpbgwjsnmuea)

[Capacity](#_yul7axaodkmy)

[Modifiers](#_27i4q9sbml6i)

[Lookup](#_butoibcbonym)

[Visualization](#_sxas3vuupsd)

[Optional](#_j03u6m8q9u5a)

[Example](#_qb8p1482n8pe)

[Example Output](#_q0h16vf5wfyc)

[Task 2: Map](#_x32vt81852y9)

[Requirements](#_eq1kgxk16ah6)

[Files](#_u9amgla8l7ub)

[Class](#_3tagllbv3zs5)

[Member types (private)](#_hce93m1gwkz9)

[Member types (public)](#_nc1iipxoq7xz)

[Functions (public)](#_zbpky3ri2768)

[Constructors](#_uu8h3pu4dhvb)

[Element Access](#_qdyui8z7uk1s)

[Iterators](#_arqeg6eyn4ai)

[Capacity](#_hq2rmkt39wrj)

[Modifiers](#_kckb3cczpjsn)

[Lookup](#_3cymio1oy8ei)

[Visualization](#_77erk5vp33cv)

[Optional](#_hnc0rirfh7nb)

[Example](#_d2u6v7vh0f21)

[Example Output](#_kvoqq1a75ry)

[Iterators](#_ugslozrkaq7l)

[Traversal with Iterators and a Note about auto](#_yhvrpl284ww0)

# 

# Approved Includes

<cstddef>

<iostream>

<sstream>

<stdexcept>

<utility>

<tuple>

"my\_set.h"

"my\_map.h"

# Code Coverage

You must submit a test suite for each task that, when run, covers at least 90% of your code. You should, at a minimum, invoke every function at least once. Best practice is to also check the actual behavior against the expected behavior, e.g. verify that the result is correct. You should be able to do this automatically, i.e. write a program that checks the actual behavior against the expected behavior.

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests.

# Starter Code

compile\_test\_set.cpp

compile\_test\_map.cpp

Makefile

map\_tests.cpp

my\_map.h

my\_set.h

set\_tests.cpp

## Files to Submit

map\_tests.cpp

my\_map.h

my\_set.h

set\_tests.cpp

# Task 1: Set

Implement a Set.

*I recommend that you use an AVL tree for this. I found using a Red-Black tree to be very challenging. If done properly, the Red-Black tree is actually more efficient. However, I found that the Red-Black tree implementation to achieve this was not worth the effort for this assignment. Part of that had to do with the implementation of the iterators. So, I further recommend that you think carefully about how you want to implement the iterators, either by threading or by including extra pointers in each node.*

## Requirements

### Files

my\_set.h - contains the template definitions

set\_tests.cpp - contains the test cases and test driver (main)

### Class

template <class Comparable>

class Set;

### Member types (private)

typedef Set\_Node<Comparable> Node;

### Member types (public)

typedef Set\_const\_iterator<Comparable> const\_iterator;

typedef Set\_iterator<Comparable> iterator;

[See Section on Iterators](#_ugslozrkaq7l)

### Functions (public)

#### Constructors & Rule of Three

**Set()** - makes an empty set

**Set(const Set&)** - constructs a copy of the given set

**~Set()** - destructs this set

**Set& operator=(const Set&)** - assigns a copy of the given set

#### Element Access

N/A

#### Iterators

**iterator begin()** - return an iterator that points to the first element of the set

**const\_iterator begin() const** - return a constant iterator that points to the first element of the set

**iterator end()** - return an iterator that points to just past the end of the set

**const\_iterator end() const** - return a constant iterator that points to just past the end of the set

#### Capacity

**bool is\_empty() const** - returns Boolean true if the set is empty

**size\_t size() const** - returns the number of elements in the set

#### Modifiers

**void make\_empty()** - remove all values from the set

**std::pair<iterator,bool> insert(const Comparable&)** - insert the given lvalue reference into the set and return an iterator to the inserted element (or the element which prevented insertion) and boolean which indicates whether the insertion was successful (if the value was newly inserted)

**iterator insert(const\_iterator, const Comparable&)** - insert the given lvalue reference into the set just after the specified position (if the hint is accurate, otherwise insert in the correct place) and return an iterator to the inserted element (or the element which prevented insertion)

**size\_t remove(const Comparable&)** - remove the specified value from the set and return the number of values removed (0 or 1)

**iterator remove(const\_iterator)** - remove the specified value (by position) from the set and return an iterator to the value after the removed value. Throw std::invalid\_argument if the iterator is invalid.

#### Lookup

**bool contains(const Comparable&) const** - returns Boolean true if the specified value is in the set and false otherwise

**iterator find(const Comparable& key)** - return an iterator that points to value in the set, or end() if the value is not found

**const\_iterator find(const Comparable&) const** - return a constant iterator that points to value in the set, or end() if the value is not found

#### Visualization

**void print\_set(std::ostream&=std::cout) const** - pretty print the set (as a curly brace-enclosed comma-separated list) to the specified output stream (default std::cout). Print “<empty>” if the set is empty.

#### Optional

**Set(Set&&)** - move constructs a copy of the given (rvalue) set

**Set& operator=(Set&&)** - move assigns a copy of the given (rvalue) set

**std::pair<iterator,bool> insert(Comparable&&)** - insert the given rvalue reference into the set using move semantics

**iterator insert(const\_iterator, Comparable&&)** - insert the given rvalue reference into the set just after the specified position (if the hint is accurate, otherwise insert in the correct place) and return an iterator to the inserted element (or the element which prevented insertion)

**void print\_tree(std::ostream&=std::cout) const** - pretty print the underlying tree

### 

### Example

// make an empty set

std::cout << "make a set" << std::endl;

Set<int> set;

std::cout << "is empty? " << std::boolalpha << set.is\_empty() << std::endl;

// insert 8 values (5 unique) into the set

std::cout << "insert 9, 6, 10, 2, 6, 5, 6, 10 " << std::endl;

set.insert(9);

set.insert(6);

set.insert(10);

set.insert(2);

set.insert(6);

set.insert(5);

set.insert(6);

set.insert(10);

{

// print the set

std::cout << "print set: ";

std::ostringstream ss;

set.print\_set(ss);

std::cout << ss.str() << std::endl;

}

std::cout << "set has " << set.size() << " elements" << std::endl;

std::cout << "is empty? " << std::boolalpha << set.is\_empty() << std::endl;

std::cout << "contains 2? " << std::boolalpha << set.contains(2) << std::endl;

// remove the root

std::cout << "contains 9? " << std::boolalpha << set.contains(9) << std::endl;

std::cout << "remove 9 " << std::endl;

set.remove(9);

std::cout << "contains 9? " << std::boolalpha << set.contains(9) << std::endl;

// find 6

std::cout << "find 6" << std::endl;

Set<int>::iterator iter = set.find(6);

std::cout << "found " << \*iter << std::endl;

std::cout << "increment iterator" << std::endl;

++iter;

std::cout << "now at " << \*iter << std::endl;

{

// print the set

std::cout << "print set: ";

std::ostringstream ss;

set.print\_set(ss);

std::cout << ss.str() << std::endl;

}

// make empty

std::cout << "make empty" << std::endl;

set.make\_empty();

std::cout << "is empty? " << std::boolalpha << set.is\_empty() << std::endl;

{

// print the set

std::cout << "print set: ";

std::ostringstream ss;

set.print\_set(ss);

std::cout << ss.str() << std::endl;

}

#### Example Output

make a set

is empty? true

insert 9, 6, 10, 2, 6, 5, 6, 10

print set: {2, 5, 6, 9, 10}

set has 5 elements

is empty? false

contains 2? true

contains 9? true

remove 9

contains 9? false

find 6

found 6

increment iterator

now at 10

print set: {2, 5, 6, 10}

make empty

is empty? true

print set: <empty>

# Task 2: Map

Implement a Map.

*I recommend that you copy and modify your Set to function as a Map. Recall that a Map is a Set where the values have type std::pair<const KeyType, ValueType> (key-value pairs). The keys are used for insertion, removal, and lookup; the iterator yields the whole pair.*

## Requirements

### Files

my\_map.h - contains the template definitions

map\_tests.cpp - contains the test cases and test driver (main)

### Class

template <class Key, class Value>

class Map;

### Member types (private)

typedef Map\_Node<Key, Value> Node;

### Member types (public)

typedef Map\_const\_iterator<Key, Value> const\_iterator;

typedef Map\_iterator<Key, Value> iterator;

[See Section on Iterators](#_ugslozrkaq7l)

### Functions (public)

#### Constructors

**Map()** - makes an empty map

**Map(const Map&)** - constructs a copy of the given map

**~Map()** - destructs this map

**Map& operator=(const Map&)** - assigns a copy of the given map

#### Element Access

**Value& at(const Key&)** - access value at specified key with bounds checking, throw std::out\_of\_range if key is not in map.

**const Value& at(const Key&) const** - access value at specified key with bounds checking, throw std::out\_of\_range if key is not in map.

**Value& operator[](const Key&)** - access or insert specified value at specified key, updates values if key already exists or inserts otherwise, returns a reference to the value

**const Value& operator[](const Key&) const** - access or insert specified value at specified key, updates values if key already exists or inserts otherwise, returns a constant reference to the value

#### Iterators

**iterator begin()** - return an iterator that points to the first element of the map

**const\_iterator begin() const** - return a constant iterator that points to the first element of the map

**iterator end()** - return an iterator that points to just past the end of the map

**const\_iterator end() const** - return a constant iterator that points to just past the end of the map

#### Capacity

**bool is\_empty() const** - returns Boolean true if the map is empty

**size\_t size() const** - returns the number of elements in the map

#### Modifiers

**void make\_empty()** - remove all key-value pairs from the map

**std::pair<iterator,bool> insert(const std::pair<const Key, Value>&)** - insert the given lvalue reference into the map and return an iterator to the inserted element (or the element which prevented insertion) and boolean which indicates whether the insertion was successful

**iterator insert(const\_iterator hint, const std::pair<const Key, Value>&)** - insert the given lvalue reference into the set just after the specified position (if the hint is accurate, otherwise insert in the correct place) and return an iterator to the inserted element (or the element which prevented insertion)

**size\_t remove(const Key&)** - remove the specified key (and its value) from the map and return the number of values removed (0 or 1)

**iterator remove(const\_iterator)** - if the iterator is valid, remove the specified key-value pair (by position) from the map and return an iterator to the value after the removed value, otherwise throw std::invalid\_argument.

#### Lookup

**bool contains(const Key&) const** - returns Boolean true if the specified key is in the map and false otherwise

**iterator find(const Key& key)** - return an iterator that points to the key-value pair in the map, or end() if the value is not found

**const\_iterator find(const Key& key) const** - return an iterator that points to the key-value pair in the map, or end() if the value is not found

#### Visualization

**void print\_map(std::ostream&=std::cout) cons**t - pretty print the map (as a curly brace-enclosed comma-separated list of key: value pairs) to the specified output stream (default std::cout). Print “<empty>” if the map is empty.

#### Optional

**Map(Map&&)** - move constructs a copy of the given (rvalue) map

**Map& operator=(Map&&)** - move assigns a copy of the given (rvalue) map

**std::pair<iterator,bool> insert(std::pair<const Key, Value>&&)** - insert the given rvalue reference into the map (using move semantics) and return an iterator to the inserted element (or the element which prevented insertion) and boolean which indicates whether the insertion was successful

**iterator insert(const\_iterator hint, std::pair<const Key, Value>&&)** - insert the given rvalue reference into the map just after the specified position (if the hint is accurate, otherwise insert in the correct place) and return an iterator to the inserted element (or the element which prevented insertion)

**void print\_tree(std::ostream&=std::cout) const** - pretty print the underlying tree

### 

### Example

// make an empty map

std::cout << "make an empty map" << std::endl;

Map<std::string, int> map;

std::cout << "is empty? " << std::boolalpha << map.is\_empty() << std::endl;

EXPECT\_TRUE(map.is\_empty());

// insert 8 values (5 unique) into the set

const std::string keys[] = {"nine", "six", "ten", "two", "six", "five", "six", "ten"};

const int values[] = {9, 6, 10, 2, 9, 5, 60, -10};

const int correct\_values[] = {9, 6, 10, 2, 6, 5, 6, 10};

Map<std::string, int>::iterator iter = map.end();

for (size\_t index = 0; index < 8; index++) {

bool success = false;

const std::string& key = keys[index];

int value = values[index];

std::cout << "insert {"<<key<<", "<<value<<"}" << std::endl;

std::tie(iter, success) = map.insert({key, value});

std::cout << "success? " << std::boolalpha << success << std::endl;

int correct\_value = correct\_values[index];

if (value == correct\_value) {

EXPECT\_TRUE(success);

} else {

EXPECT\_FALSE(success);

}

std::cout << "iterator points to " << iter->first << ": " << iter->second << std::endl;

EXPECT\_EQ(iter->first, key);

EXPECT\_EQ(iter->second, correct\_value);

}

{

// print the map

std::cout << "print map: ";

std::ostringstream ss;

map.print\_map(ss);

std::cout << ss.str() << std::endl;

EXPECT\_EQ(ss.str(), "{five: 5, nine: 9, six: 6, ten: 10, two: 2}");

}

// get size

std::cout << "map has " << map.size() << " elements" << std::endl;

EXPECT\_EQ(map.size(), 5);

std::cout << "is empty? " << std::boolalpha << map.is\_empty() << std::endl;

EXPECT\_FALSE(map.is\_empty());

std::cout << "contains \"seven\"? " << std::boolalpha << map.contains("seven") << std::endl;

EXPECT\_FALSE(map.contains("seven"));

// remove the root?

std::cout << "contains \"six\"? " << std::boolalpha << map.contains("six") << std::endl;

EXPECT\_TRUE(map.contains("six"));

std::cout << "remove \"six\" " << std::endl;

size\_t cnt = map.remove("six");

std::cout << cnt << " values removed" << std::endl;

EXPECT\_EQ(cnt, 1);

std::cout << "contains \"six\"? " << std::boolalpha << map.contains("six") << std::endl;

EXPECT\_FALSE(map.contains("six"));

// find "nine"

std::cout << "find \"nine\"" << std::endl;

iter = map.find("nine");

ASSERT\_NE(iter, map.end());

std::cout << "found " << iter->first << ": " << iter->second << std::endl;

EXPECT\_EQ(iter->first, "nine");

EXPECT\_EQ(iter->second, 9);

std::cout << "increment iterator" << std::endl;

++iter;

ASSERT\_NE(iter, map.end());

std::cout << "now at " << iter->first << ": " << iter->second << std::endl;

EXPECT\_EQ(iter->first, "ten");

EXPECT\_EQ(iter->second, 10);

{

// print the map

std::cout << "print map: ";

std::ostringstream ss;

map.print\_map(ss);

std::cout << ss.str() << std::endl;

EXPECT\_EQ(ss.str(), "{five: 5, nine: 9, ten: 10, two: 2}");

}

// make empty

std::cout << "make empty" << std::endl;

map.make\_empty();

std::cout << "is empty? " << std::boolalpha << map.is\_empty() << std::endl;

EXPECT\_TRUE(map.is\_empty());

{

// print the map

std::cout << "print map: ";

std::ostringstream ss;

map.print\_map(ss);

std::cout << ss.str() << std::endl;

EXPECT\_EQ(ss.str(), "<empty>");

}

// use operator[]

std::cout << "contains \"what\"? " << std::boolalpha << map.contains("what") << std::endl;

EXPECT\_FALSE(map.contains("what"));

std::cout << "access map[\"what\"]" << std::endl;

map["what"];

EXPECT\_EQ(map["what"], 0);

std::cout << "contains \"what\"? " << std::boolalpha << map.contains("what") << std::endl;

EXPECT\_TRUE(map.contains("what"));

std::cout << "map has " << map.size() << " elements" << std::endl;

EXPECT\_EQ(map.size(), 1);

{

// print the map

std::cout << "print map: ";

std::ostringstream ss;

map.print\_map(ss);

std::cout << ss.str() << std::endl;

EXPECT\_EQ(ss.str(), "{what: 0}");

}

std::cout << "assign value 1 to map[\"what\"]" << std::endl;

map["what"] = 1;

std::cout << "map has " << map.size() << " elements" << std::endl;

EXPECT\_EQ(map.size(), 1);

std::cout << "map[\"what\"] = " << map["what"] << std::endl;

EXPECT\_EQ(map["what"], 1);

#### Example Output

make an empty map

is empty? true

insert {nine, 9}

success? true

iterator points to nine: 9

insert {six, 6}

success? true

iterator points to six: 6

insert {ten, 10}

success? true

iterator points to ten: 10

insert {two, 2}

success? true

iterator points to two: 2

insert {six, 9}

success? false

iterator points to six: 6

insert {five, 5}

success? true

iterator points to five: 5

insert {six, 60}

success? false

iterator points to six: 6

insert {ten, -10}

success? false

iterator points to ten: 10

print map: {five: 5, nine: 9, six: 6, ten: 10, two: 2}

map has 5 elements

is empty? false

contains "seven"? false

contains "six"? true

remove "six"

1 values removed

contains "six"? false

find "nine"

found nine: 9

increment iterator

now at ten: 10

print map: {five: 5, nine: 9, ten: 10, two: 2}

make empty

is empty? true

print map: <empty>

contains "what"? false

access map["what"]

contains "what"? true

map has 1 elements

print map: {what: 0}

assign value 1 to map["what"]

map has 1 elements

map["what"] = 1

# Iterators

Iterators must support the operations

* Default constructor, creates an iterator which points to null
* Pre- and post-increment, ++iter and iter++, moves the iterator to the next element
* Equals, iter1 == iter2, compares iterators for equality (point to same node)
* Not Equals, iter1 != iter2, compares iterators for inequality (point to different nodes)
* Dereference, \*iter, returns a reference to the value stored in the node to which it points

Other useful operations for iterators:

* Construct from pointer, iterator iter = node, creates an iterator which points to the node
* Arrow, iter->member, dereferences the iterator and returns the *address* of the value stored in the node to which it points.

Note: dereferencing the end iterator is undefined (could be SEGFAULT, could be weird behavior) in the STL. **For this assignment, I want you to instead throw a std::runtime\_error exception if the user tries to dereference the end iterator.**

Note: incrementing the end iterator should result in the end iterator. I.e. end()++ goes nowhere.

## Traversal with Iterators and a Note about auto

Iterators make life fun. Once you have a working iterator for Set, you can do this:

Set<T> set;

// insert stuff

for (T value : set) {

// do something to value

}

Recall, this is equivalent to

Set<T> set;

// insert stuff

for (Set<T>::iterator iter = set.begin(); iter != set.end(); iter++) {

int value = \*iter;

// do something to value

}

Have you heard of auto? Please don’t use it for simple things. It would look like this:

Set<T> set;

// insert stuff

for (auto iter = set.begin(); iter != set.end(); iter++) {

auto value = \*iter;

// do something to value

}

But auto is useful sometimes, like when the proper type is quite long and/or complex. Or, when the language literally requires that it be used. Consider this range-based for loop to traverse a Map:

Map<K,V> map;

// insert stuff

for (const auto& [key, value] : map) {

// do something with key and/or value

}

This gives access to the already dereferenced and split key-value pairs without having to go through the hoops of pulling each pair out and manually decomposing it:

Map<K,V> map;

// insert stuff

for (const std::pair<const K, V>& key\_value : map) {

const auto& [key, value] = key\_value;

// do something with key and/or value

}

Even that is a bit of a “hack”, since we just push the fancy bit (called *structured binding*) into the body of the loop. To get rid of auto entirely, we have to write the code like this:

Map<K,V> map;

// insert stuff

for (const std::pair<const K, V>& key\_value : map) {

const K& key = key\_value.first;

const V& value = key\_value.second;

// do something with key and/or value

}

Structured binding requires the use of auto. Only when auto is *required* by the language are you allowed to use it. Otherwise, you must use the correct name of the type. Knowing the types of your variables will help you write better code and you will spend less time debugging.

Iterator Help:

<https://www.internalpointers.com/post/writing-custom-iterators-modern-cpp>