# Learning C++: Using Sets and Multisets

There are times when you need a container that holds only unique values that are ordered by some criteria. The set class of the Standard Template Library has these features. If your requirements allow the container to hold duplicate, ordered values, then the multiset class is the container you want to use. In this article I'll discuss how to use these two classes and how to modify the way the data in these containers can be ordered. I'll cover the set class first and then talk about the difference when you use the multiset class instead.

## Creating Sets

The set class is imported into your program with this header file:

#include <set>

The class is a template class so when you declare a set you must provide a data type. A set can be of any data type that is comparable according to a sorting criterion. This means you can have a set of a user-defined type if some type of sorting function is defined for that type.

A set declaration must include the data type for the set elements:

set<string> names;

set<int> grades;

You can also declare a sorting criterion in the declaration as a second template parameter. I'll demonstrate how this works later in the article.

You can declare a set with initial data by supplying an initializer list. The list does not have to be sorted as its insertion into the set will automatically sort it. For example:

int main()

{

set<int> numbers = {3,1,2};

for (const int n : numbers) {

cout << n << " "; // displays 1 2 3

}

return 0;

}

## Adding Data to a Set

The primary way to add data into a set is the insert function. This function places an element into its proper place in the set:

set<int> numbers;

numbers.insert(3);

numbers.insert(1);

numbers.insert(2);

The function returns both the position where the element was inserted and, for a set, whether the insertion was successful. This return value is a pair structure where the first field is the insertion position and the second field is a boolean value showing the success or failure of the insertion.

Here is an example of how to handle the return value of the insert function:

int main()

{

set<int> numbers;

pair<set<int>::iterator, bool> success;

success = numbers.insert(3);

if (success.second) {

cout << "3 was inserted at position "

<< \*success.first << "." << endl;

}

success = numbers.insert(1);

if (success.second) {

cout << "1 was inserted at position "

<< \*success.first << "." << endl;

}

success = numbers.insert(2);

if (success.second) {

cout << "2 was inserted at position "

<< \*success.first << "." << endl;

}

return 0;

}

The output from this program is:

3 was inserted at position 3.

1 was inserted at position 1.

2 was inserted at position 2.

There are other versions of the insert function as well. You can provide a suggested insertion point to the function to speed up the insertion. Here is an example:

#include <iostream>

#include <set>

using namespace std;

int main()

{

set<int> numbers = {8, 1, 2, 4, 5, 7, 3};

int value = 6;

auto iter = numbers.begin();

while (\*iter < value) {

++iter;

}

numbers.insert(iter, value);

for (const int n : numbers) {

cout << n << " "; // 1 2 3 4 5 6 7 8

}

return 0;

}

You can also insert a list of elements into a set, as demonstrated in the program below:

int main()

{

set<int> numbers = {2,4,6};

numbers.insert({1,3,5});

for (const int n : numbers){

cout << n << " "; // 1 2 3 4 5 6

}

return 0;

}

## Removing Elements from a Set

Elements are removed from a set with the erase function. The first version of this function removes a specific value and returns the number of elements removed:

int main()

{

set<int> numbers = {1,3,2,5,4};

int removed = 3;

int count = numbers.erase(removed);

cout << count << " element was removed." << endl << endl;

for (const int n : numbers) {

cout << n << " "; // displays 1 2 4 5

}

return 0;

}

The second version of erase removes the element via an iterator that points to a position in the set. Its return value is the position after the element removed. In the example below I ignore the return value:

int main()

{

set<int> numbers = {1,3,2,5,4};

int removed = 3;

auto iter = numbers.begin();

while (\*iter != removed) {

iter++;

}

numbers.erase(iter);

for (const int n : numbers) {

cout << n << " ";

}

return 0;

}

## Searching a Set

There are several member functions in the set class for searching. The primary function for searching is find. This function takes a value as a parameter and searches a set for the value. If the value is found, the function returns an iterator pointing at the value. If the value isn't found, the function returns end(), or the position just past the last element of the set.

Here is a program that demonstrates how the find function works:

#include <iostream>

#include <set>

#include <cstdlib>

#include <ctime>

using namespace std;

void buildSet(set<int> &st) {

srand(time(0));

for (int i = 1; i <= 25; i++) {

st.insert(rand() % 100 + 1);

}

}

void printSet(const set<int> st) {

for (int n : st) {

cout << n << " ";

}

}

int main()

{

set<int> numbers;

buildSet(numbers);

printSet(numbers);

cout << endl << endl;

int value;

cout << "Enter a value to find: ";

cin >> value;

auto found = numbers.find(value);

if (found != numbers.end()) {

cout << "Found " << value << "." << endl;

}

else {

cout << value << " was not found in the set. " << endl;

}

return 0;

}

Here is the output from two runs of the program:

14 21 24 27 35 38 42 44 52 57 58 60 68 71 80 84 86 89 92 96

Enter a value to find: 57

Found 57.

7 10 11 15 22 23 34 36 40 53 56 60 62 63 65 72 76 79 84 93 94 96 97

Enter a value to find: 77

77 was not found in the set.

## Changing the Sort Order

As I mentioned earlier, the primary sorting criterion for sets is less<>. You can change this by specifying a different criterion when you declare a new set. If you aren't familiar with the different sorting criteria available in C++, [here](https://www.cplusplus.com/reference/functional/less/) is a reference guide for you.

To demonstrate how changing the sorting criterion works, I'll create a new set that is sorted using the greater<int> criterion. Here is the program:

#include <iostream>

#include <set>

#include <cstdlib>

#include <ctime>

using namespace std;

void buildSet(set<int, greater<int>> &st) {

srand(time(0));

for (int i = 1; i <= 25; i++) {

st.insert(rand() % 100 + 1);

}

}

void printSet(const set<int, greater<int>> st) {

for (int n : st) {

cout << n << " ";

}

}

int main()

{

set<int, greater<int>> numbers;

buildSet(numbers);

printSet(numbers);

return 0;

}

The output from this program is:

90 87 85 77 65 62 50 40 39 38 31 29 26 25 24 21 20 19 15 14 8 6 1

Notice that I had to specify the second parameter in all the function parameters. Templated functions will fix this and I'll address this in a future article.

## Using the multiset Class

If you need the ordering features of a set but you want to allow duplicates in the container, you should use the multiset class. The functions we used for sets also work for multisets.

You can still reference the set class in your preprocessor directives to use a multiset. The program below creates a new multiset and displays the values in the container:

#include <iostream>

#include <set>

#include <cstdlib>

#include <ctime>

using namespace std;

void buildSet(multiset<int> &st) {

srand(time(0));

for (int i = 1; i <= 25; i++) {

st.insert(rand() % 100 + 1);

}

}

void printSet(const multiset<int> st) {

for (int n : st) {

cout << n << " ";

}

}

int main()

{

multiset<int> numbers;

buildSet(numbers);

printSet(numbers);

return 0;

}

Here is the output from one run of this program:

9 11 15 18 18 28 45 46 54 63 64 67 68 68 71 77 79 79 84 84 87 88 91 93 95

Notice there are duplicate 18s, 68s, and 84s.

All the functions I demonstrated earlier with sets work with multisets. One function you can productively use just with multisets is the count function, which counts the numbers of a specified value in a set. I didn't demonstrate this with set instances because its known that only one value is stored in a set.

Here is a program that counts the number of values found in a multiset:

int main()

{

multiset<int> numbers;

buildSet(numbers);

printSet(numbers);

int value;

cout << endl << endl;

cout << "Enter a value to count: ";

cin >> value;

auto found = numbers.find(value);

if (found != numbers.end()) {

int numVals = numbers.count(value);

cout << "Found " << numVals << " " << value

<< "s in the set." << endl;

}

return 0;

}

Here is one run of the program:

5 20 21 21 22 22 25 29 30 33 33 50 57 66 70 70 73 77 81 82 82 89 89 93 94

Enter a value to count: 70

Found 2 70s in the set.

## When to Use Sets and Multisets

The primary reason to use a set or a multiset is when you need to keep your data in a sorted order. The underlying data structure for sets and multisets is the balanced binary tree, which means that keeping the data in order is efficient. However, using binary trees also creates a problem because you can't change the value of an element in place because doing so would invalidate the ordering. So, making changes involves removing an element from the container and then inserting a new element. This is more inefficient than doing so in another container so you will want to choose a different container if you need to make lots of changes to the data in your application.

Thanks for reading this article and please email me with comments and suggestions.