Associative Containers

Associative containers are set of pairs arranged in such a way that it can do quick lookup on the first element.

The associative containers can be grouped into two subsets:

1. sets: set of ascending/descending unique elements.
2. maps: (aka, dictionary): a key/value pair.

The **key** is used to order the sequence, and the **value** is somehow associated with that key.

Both *map* and *set* only allow **one** instance of the key to be inserted into the container; If multiple instances of elements are required, we need **multimap** or **multiset** containers respectively.

Associative containers are designed to be especially efficient in accessing its elements by their key, as opposed to sequence containers which are more efficient in accessing elements by their position.

Associative containers are guaranteed to perform operations of insertion, deletion, and testing whether an element is in it, in logarithmic time - O(log n).

Both maps and sets support bidirectional iterators.

Set

The set container stores **unique** elements following a **specific** order in which the value of an element identifies it (the value is the key).

Properties

- Associative: elements are referenced by their key and not by their absolute position at the container.
- Order: elements follow always a strict order. All inserted are given a position in this order.
- Set: the value of an element is also the key used to identify it.
- Unique keys: No multiple elements of the container can have the same key.
- Allocator-aware: memory is dynamically handled.

Methods provided by the class

- Constructor: creates the container.
- Destructor: frees the container and its elements.
- operator=: copy constructor.
- ::begin: points to the initial value.
- ::end: points to the final value.
- empty: tests whether container is empty.
- size: returns container size.
- max_size: returns maximum size.
- insert: inserts element.
- erase: erases elements.
- swap: swaps content.
- clear: clears content.
- count: returns the number of elements matching specific key.
- find: finds element with specific key.
- contains: checks if the container contains element with specific key. C++20

Using sets

1. Including the headers

As allways, the set container headers have to be included in order to use them in the code.

In [ ]:
```
#include <set>
```

2. Creating the container

The container definition have to specifically define the datatype of the objects to be stored.

The container can be created by calling its empty, parameter or copy constructor.

In [ ]:
```
std::set<int> myset;  //empty constructor

char mychars[]= {'a','b','c','d','e'};
std::set<char> otherset(mychars, mychars+5); //parameter constructor

std::set<char> otherchars(otherset); // copy constructor
```

3. Using the container

Which is the output of this code?:

In [ ]:
```
#include <iostream>
int myints[]={5,10,15};
std::set<int>::iterator it;

for (int i=0; i<=5; i++)
    myset.insert(i*10);

myset.insert(25);
myset.insert(24);
myset.insert(26);
myset.insert(myints, myints+3);

std::cout << "Elements of the set container: ";
for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << *it << " ";
std::cout << std::endl;
```

- 0 10 20 30 40 50 25 24 26 5 10 15
- 0 5 10 15 20 24 25 26 30 40 50

4. Clearing the container:

For removing the content of the container, we use the **clear** method.

In [ ]:
```
myset.clear();
otherset.clear();
otherchars.clear();
```

Map

Maps are associative containers that store elements formed by a combination of a **key** value and a **mapped** value, following a specific order.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key.

The types of key and mapped value may differ, and are grouped together in member type, **pair type**, combining both.

Properties

- Associative: elements are references by their key and not by their absolute position at the container.
- Order: elements follow always a strict order. All inserted are given a position in this order.
- Map: Each element associates a key to a mapped value.
- Unique keys: No multiple elements of the container can have the same key.
- Allocator-aware: memory is dynamically handled.

Some methods provided by the class

- constructor: constructs unordered_map.
- destructor: destroy unordered map.
- empty: tests whether container is empty.
- size: returns container size.
- max_size: return maximum size.
- ::begin : return iterator to beginning.
- ::end : return iterator to end.
- operator[]: access element.
- at: access element.
- find: gets iterator to element.
- count: counts elements with a specific key.
- insert: inserts elements.
- erase: erases elements.
- clear: clears content.
- swap: swaps content.

Pairs

Couples together a pair of values, which may be of different types.

The individual values can be accessed through its public members first and second.

A pair can be created by calling the constructor or by calling (once we have the reference) the create_pair() member function.

In [ ]:
```cpp
#include <utility> //std::pair and std::make_pair are there
#include <string>
#include <iostream>

//Empty constructor class pair
std::pair<std::string, double> product1;

//Parameter constructor
std::pair<std::string, double> product2("tomatoes", 2.30);

//Copy constructor
std::pair<std::string, double> product3(product2);

product1=std::make_pair(std::string("apple"), 1.99);
product2.first = "shoes";
product2.second = 39.90;

std::cout << "The price of " << product1.first << " is " << product1.second << "€" << std::endl;
std::cout << "The price of " << product2.first << " is " << product2.second << "€" << std::endl;
std::cout << "The price of " << product3.first << " is " << product3.second << "€" << std::endl;
```

Using maps

1. Including the headers

As shown before, the headers have to be included before first use of the container.

In [ ]:
```cpp
#include <map>
```

2. Creating the container

Map containes can be created in a similar way as the containers shown before.

In this case, one other constructor has been included to see other possible options:

In [ ]:
```
std::map<std::string, std::string> firstmap; // empty constructor
std::map<std::string, std::string> secondmap({{"apple","red"},{"lemon","yellow"}}); // init constructor
std::map<std::string, std::string> thirdmap (secondmap); // copy constructor
std::map<std::string, std::string> fourthmap (thirdmap.begin(),thirdmap.end()); // range constructor
```

3. Using the container

Lets see some examples about how to use the map containers.

1. **Inserting an element into an existing map container object:**

   In this case, we insert to elements which contain a letter and a number.

In [ ]:
```
std::map<char, int> mymap;

mymap.insert(std::pair<char,int>('a',100) );
mymap.insert(std::pair<char,int>('z',200) );
```

2. **Inserting and element which already was in the container:**

   Since the properties of the map containers, there can not be two elements with the same key value:

In [ ]:
```
#include <iostream>
#include <utility>

std::pair<std::map<char, int>::iterator,bool> ret;
ret=mymap.insert(std::pair<char, int>('z',500));
if (ret.second==false)
    std::cout << "Element " << ret.first->first << " is already in the map with a value of " << ret.first->second <
```

The element 'z' can not be inserted again in the container.

3. **Interting elements in the container from a certain position on:**

In this case, the two new pair elements will be inserted at the beginning of the container:

In [ ]:
```
std::map<char,int>::iterator it = mymap.begin();
mymap.insert(it, std::pair<char,int>('b',300));
mymap.insert(it, std::pair<char,int>('c',400));
```

4. **Inserting elements from other container until a certain value:**

   In this case, the *anothermap* container will store a copy of the values of the original container until it finds the element with the key equal to 'c'.

In [ ]:
```
std::map<char,int> anothermap;
anothermap.insert(mymap.begin(), mymap.find('c'));
```

5. **Printing the elements of a container:**

   We print the content of both containers with the iterator, as usual.

In [ ]:
```
std::cout << "mymap contains: " << std::endl;
for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " -> " << it->second << std::endl;

std::cout << "anothermap contains: " << std::endl;
for (it=anothermap.begin(); it!=anothermap.end(); ++it)
    std::cout << it->first << " -> " << it->second << std::endl;

// C++11
for (const auto& kv : mymap) {
    std::cout << kv.first << " has value " << kv.second << std::endl;
}

// C++17
for (auto& [key, value]: mymap) {
    std::cout << key << " has value " << value << std::endl;
}
```

6. **Clearing the container:**

   For clearing the container (usually not needed), we call the **clear** method.

In [ ]:
```
mymap.clear();
anothermap.clear();
```

1.- **Is it the same using?:**

- std::map<std::string, int>

as using:

- std::map<int, std::string>

2.- **Can we use:** std::vector<std::pair<X, Y>> **instead of:** std::map<X, Y> **?**

1. From the point of view of Performance
    - map: ordered structure with respect to the key; vector: sequence of pairs where the order keeps constant.
    - map: does not allow duplicated keys; vector: can hold any number of duplicated.
    - map: fast search O(log n); vector: brute force search O(n).
2. From the point of view of results or correctness:
    - if well implemented, no differences.

Multiset

The multiset container is an associative container that contains a sorted set of objects of type *Key*. Unlike set, multiple keys with equivalent values are allowed.

The sorting is done using the key comparison function; searching, inserting, and erasing operations have logarithmic complexity.

It must be taken into account that the value of the elements in a multiset cannot be modified once in the container (the elements are always const), but they can be removed and re-inserted from the container if needed.

Properties

- Associative: elements are references by their key and not by their absolute position at the container.
- Order: elements follow always a strict order. All inserted are given a position in this order.
- Set: the value of an element is also the key used to identify it.
- Multiple keys: multiple elements of the container can have equivalent key.
- Allocator-aware: memory is dynamically handled.

Some methods provided by the class

- constructor: construct the map.
- destructor: destroys the map.
- operator=: assigns content to the map.
- empty: tests whether container is empty.
- size: returns container size
- ::begin : return iterator to beginning.
- ::end : returns iterator to end.
- operator[]: accesses element at certain position.
- at: accesses element.
- find: gets iterator to element if it is found in the container.
- count: counts elements with a specific key.
- insert: inserts elements in the container.
- erase: erases elements from the container.
- clear: clears the content.
- swap: swaps elements of the container.

Using multiset containers

1. Including the headers

In order to be able to use the container, the headers have to be included. Otherwise, the compiler does not recognise, what the container is.

In [ ]:
```
#include <set>
```

2. Creating the container

The multiset containers can be created in a similar way as the containers shown before.

In [ ]:
```
std::multiset<int> mymultiset; // empty constructor

int myints[]= {10,20,30,20,20};
std::multiset<int> secondmulti (myints,myints+5); // range constructor

std::multiset<int> thirdmulti (secondmulti); // copy constructor

std::multiset<int> fourthmulti (secondmulti.begin(), secondmulti.end());  // iterator constructor.
```

3. Using the container

Lets see some examples on how to work with this container.

1. **Inserting elements into an existing container object:**

In [ ]:
```
for (int i=0; i<=5; i++)
    mymultiset.insert(i*10);
```

We can also insert elements from a certain position on using iterators:

In [ ]:
```
std::multiset<int>::iterator it;

it = mymultiset.insert(25);
it = mymultiset.insert(it, 27); //max efficiency by inserting the element.
it = mymultiset.insert(it, 29);
it = mymultiset.insert(it, 24); //less efficiency inserting.
```

Lets insert now elements from an array:

In [ ]:
```
mymultiset.insert(myints, myints+4);
```

What do you think, it will be the output of this code?:

In [ ]:
```
std::cout << "The multiset container contains: ";
for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
    std::cout << *it << " ";
std::cout << std::endl;
```

- 0 10 20 30 40 50 25 27 29 24 10 20 30 20
- 0 10 10 20 20 20 24 25 27 29 30 30 40 50
- 0 10 20 30 40 50 25 27 29 24

2. **Searching elements in an existing container object:**

   The operation is, as expected, completely straightforward

In [ ]:
```
it=mymultiset.find(20);
mymultiset.erase (it);

mymultiset.erase (mymultiset.find(40));

std::cout << "The multiset container after removing some elements contains: ";
for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
    std::cout << *it << " ";
std::cout << std::endl;
```

3. **Clearing the container:**

In [ ]:
```
mymultiset.clear();
```

Multimap

The concept of multimap does not differ a lot from the concept of multiset but just applied to maps and not to sets containers.

Multimaps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order, and where multiple elements can have equivalent keys.

In a multimap, the key values are generally used to sort and uniquely identify the elements, whilethe mapped values store the content associated to this key.

The types of key and mapped value may of course differ.

Properties:

- Associative: elements are references by their key and not by their absolute position at the container.
- Order: elements follow always a strict order. All inserted are given a position in this order.
- Map: Each element associates a key to a mapped value.
- Multiple keys: multiple elements of the container can have equivalent key.
- Allocator-aware: memory is dynamically handled.

Some methods provided by the class

Lets see a little list, what we can do with multimap containers:

- constructor: Constructs multimap.
- destructor: Multimap destructor.
- operator= : Copy the container content.
- ::begin : returns the iterator to beginning of the container.
- ::end : return the iterator to end.
- empty: tests whether container is empty or not.
- size: return the container size.
- insert: insert element into the container.
- erase: erases elements from the container.
- swap: swaps content between elements of the container.
- clear: clears content.
- emplace: constructs and inserts element in the container.
- emplace_hint: construct and insert element with hint.
- find Get iterator to element.
- count Count elements with a specific key.
- lower_bound Return iterator to lower bound.
- upper_bound Return iterator to upper bound.
- equal_range Get range of equal elements.

Using multimap containers

Lets do some examples to understand the use of the multimap containers:

1. Including the headers

The procedure to include the headers of this container is the same as the one shown with the other containers:

In [ ]:
```
#include <map>
```

2. Creating the container

The multimap containers can be created in a similar way as all the containers shown before.

Lets see some examples:

In [ ]:
```
std::multimap<char,int> first;                              // Using the empty constructor
std::multimap<char,int> second (first.begin(),first.end()); // Using the iterator constructor
std::multimap<char,int> third (second);                     // Using the copy constructor
```

3. Using the container

Lets try to understand this complete example:

What should be the output of this code?:

In [ ]:

```
std::multimap<char, int> mymultimap;
std::multimap<char, int>::iterator it;

mymultimap.insert(std::pair<char, int>('a',100));
mymultimap.insert(std::pair<char, int>('z',150));
it = mymultimap.insert(std::pair<char, int>('b',75));

mymultimap.insert(it, std::pair<char, int>('c',300));
mymultimap.insert(it, std::pair<char, int>('z',400));

std::multimap<char, int> another;
another.insert(mymultimap.begin(), mymultimap.find('c'));

std::cout << "Mymultimap contains: ";
for (it=mymultimap.begin(); it!=mymultimap.end(); ++it)
    std::cout << "(" << it->first << "," << it->second << ") ";   //it->first is equal to (*it).first...
std::cout << std::endl;

std::cout << "Another contains: ";
for (it=another.begin(); it!=another.end(); ++it)
    std::cout << "(" << it->first << "," << it->second << ") ";
std::cout << std::endl;
```

And from this one?:

In [ ]:

```
std::multimap<char,int> mymm;

mymm.insert(std::make_pair('x',50));
mymm.insert(std::make_pair('y',100));
mymm.insert(std::make_pair('y',150));
mymm.insert(std::make_pair('y',200));
mymm.insert(std::make_pair('z',250));
mymm.insert(std::make_pair('z',300));

for (char c='x'; c<='z'; c++)
{
    std::cout << "There are " << mymm.count(c) << " elements with key " << c << ":";
    std::multimap<char,int>::iterator it;
    for (it=mymm.equal_range(c).first; it!=mymm.equal_range(c).second; ++it)
        std::cout << ' ' << (*it).second;
    std::cout << '\n';
}
```

In [ ]: