- Examples

We will discuss the examples of different associative containers for comparative analysis in this lesson.

WE'LL COVER THE FOLLOWING

Example 1 - Unordered Map

```
//unorderedMap.cpp
#include <iostream>
#include <map>
#include <string>
#include <unordered_map>

int main(){

  std::cout << std::endl;

  // using the C++ map
  std::map<std::string, int> m { {"Dijkstra", 1972}, {"Scott", 1976}, {"Wilkes", 1967}, {"Ham
  for(auto p : m) std::cout << '{' << p.first << ", " << p.second << '}';
  std::cout << std::endl;
  std::cout << "m[Scott]: " << m["Scott"] << std::endl;
  m["Ritchie"] = 1983;
  m["Scott"]= 1988;
  for(auto p : m) std::cout << '{' << p.first << ", " << p.second << '}';
```

```
  std::cout << "\n\n";

  // using the C++11 unordered_map

  std::unordered_map<std::string, int> um { {"Dijkstra", 1972}, {"Scott", 1976}, {"Wilkes", 1
  for(auto p: um) std::cout << '{' << p.first << ", " << p.second << '}';
  std::cout << std::endl;
  std::cout << "um[Scott]: " << um["Scott"] << std::endl;
  um["Ritchie"] = 1983;
  um["Scott"]= 1988;
  for(auto p : um) std::cout << '{' << p.first << ", " << p.second << '}';

  std::cout << std::endl;

  std::cout << std::endl;
}
```

Explanation

- In the example above, we have defined an `std::map` `m` and an `std::unordered_map` `um` and stored the same data in both containers.
- Values in `std::map` are stored depending on the alphabetical value of their associated keys.
- Values in `std::unordered_map` are stored depending on the hash values of their associated keys.
- Values can be accessed using their associated keys.

Example 2 - Unordered Multimap

```
// unorderedMapMultimap.cpp
#include <iostream>
#include <map>
#include <unordered_map>

int main(){

  std::cout << std::endl;

  long long home= 497074123456;
  long long mobile= 4916046123356;

  // constructor
  std::unordered_multimap<std::string, long long> multiMap{{"grimm", home}, {"grimm", mobile}
  std::unordered_map<std::string, int> uniqMap{{"bin", 1}, {"root", 20}, {"nobody", 65834}, {

  // show the unordered maps
  std::cout << "multiMap: ";
```

```
std::cout << "multiMap: ";
for(auto m : multiMap) std::cout << '{' << m.first << ", " << m.second << '}';

std::cout << std::endl;

std::cout << "uniqMap: ";
for(auto u : uniqMap) std::cout << '{' << u.first << ", " << u.second << '}';
std::cout << std::endl;

std::cout << std::endl;

// insert elements
long long work= 4970719754513;

multiMap.insert({"grimm", work});
// will not work
//multiMap["grimm-jaud"]=4916012323356;

uniqMap["lp"]=4;
uniqMap.insert({"sshd", 71});

std::map<std::string, int> myMap{{"ftp", 40}, {"rainer", 999}};
uniqMap.insert(myMap.begin(), myMap.end());

// show the unordered maps
std::cout << "multiMap: ";
for(auto m : multiMap) std::cout << '{' << m.first << ", " << m.second << '}';

std::cout << std::endl;

std::cout << "uniqMap: ";
for(auto u : uniqMap) std::cout << '{' << u.first << ", " << u.second << '}';
std::cout << std::endl;


std::cout << std::endl;
// search for elements

// only grimm
auto iter= multiMap.equal_range("grimm");
std::cout << "grimm: ";
for(auto itVal= iter.first; itVal !=iter.second;++itVal){
  std::cout << itVal->second << " ";
}

std::cout << std::endl;

std::cout << "multiMap.count(grimm): " << multiMap.count("grimm") << std::endl;

auto it= uniqMap.find("root");
if ( it != uniqMap.end()){
  std::cout << "uniqMap.find(root): " << it->second << std::endl;
  std::cout << "uniqMap[root]: " << uniqMap["root"] << std::endl;
}

// will create a new entry
std::cout << "uniqMap[notAvailable]: " << uniqMap["notAvailable"] << std::endl;

std::cout << std::endl;

// remove
int numMulti= multiMap.erase("grimm");
int numUniq= uniqMap.erase("rainer");
```

```
    std::cout << "Erased " << numMulti << " times grimm from multiMap." << std::endl;
    std::cout << "Erased " << numUniq << " times rainer from uniqMap." << std::endl;

    // all
    multiMap.clear();
    uniqMap.clear();

    std::cout << std::endl;

    std::cout << "multiMap.size(): " << multiMap.size() << std::endl;
    std::cout << "uniqMap.size(): " << uniqMap.size() << std::endl;

    std::cout << std::endl;

}
```

Explanation

- In lines 14-15, we have defined an `std::unordered_multimap` named `multiMap` and an `std::unordered_map` named `uniqMap`. `std::unordered_multimap` can have multiple values for the same key, but `std::unordered_map` can only have one associated value for a key.

- In lines 19 and 24, we print the key/value pairs for both the maps.

- In line 32, we insert another value, `work`, associated with the key `grimm` using the built-in function `insert()`. This is the only way to insert new values in the `std::unordered_multimap`.

- There are multiple methods used to insert values in `std::unordered_map`. They are shown in lines 36 - 37 and 39 - 40. Ensure that the key/value pairs are of the same data type as the `std::unordered_map`.

- In lines 57 - 60, we display all the values associated with `grimm` in `multiMap`.

- In line 79, we have erased the values associated with `grimm` in `multiMap`, and they turn out to be in 3. In line 80, we erased all the values associated with `rainer` in `uniqMap` and they turn out to be in 1. We have used the built-in function `erase()` for `multiMap`.

Example 3 - Unordered Map with Hash

```cpp
// unorderedMapHash.cpp
#include <iostream>
#include <ostream>
#include <unordered_map>

struct MyInt{
  MyInt(int v):val(v){}
  bool operator== (const MyInt& other) const {
    return val == other.val;
  }
  int val;
};

struct MyHash{
  std::size_t operator()(MyInt m) const {
    std::hash<int> hashVal;
    return hashVal(m.val);
  }
};

std::ostream& operator << (std::ostream& strm, const MyInt& myIn){
  strm << "MyInt(" << myIn.val << ")";
  return strm;
}

int main(){

  std::cout << std::endl;

  std::hash<int> hashVal;

  // a few hash values
  for ( int i= -2; i <= 1 ; ++i){
    std::cout << "hashVal(" << i << "): " << hashVal(i) << std::endl;
  }

  std::cout << std::endl;

  typedef std::unordered_map<MyInt, int, MyHash> MyIntMap;

  std::cout << "MyIntMap: ";
  MyIntMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

  for(auto m : myMap) std::cout << '{' << m.first << ',' << m.second << '}';

  std::cout << std::endl << std::endl;

}
```

Explanation

- The class `MyInt` is a small wrapper for an `int`.

- To use instances of `MyInt` as a key in an associative container, `MyInt` must support the hash function and the equal operator.
- The class `MyHash` implements the hash function for the key `MyInt` by using the hash value for the wrapped `int` (line 17). Line 34 shows a few hash values for `int`'s. In contrast, the equal operator is implemented inside the class (line 8).
- Line 39 defines the type `MyIntMap`, which uses the hash function `MyHash` as a template argument.

Example 4 - Performance Comparison

> 💡 The following code will take some time to compile

```
// unorderedOrderedContainerPerformance.cpp
#include <chrono>
#include <iostream>
#include <map>
#include <random>
#include <unordered_map>

static const long long mapSize= 1000000;
static const long long accSize= 500000;

int main(){

  std::cout << std::endl;

  std::map<int, int> myMap;
  std::unordered_map<int, int> myHash;

  for ( long long i=0; i < mapSize; ++i ){
    myMap[i]=i;
    myHash[i]= i;
  }

  std::vector<int> randValues;
  randValues.reserve(accSize);

  // random values
  std::random_device seed;
  std::mt19937 engine(seed());
  std::uniform_int_distribution<> uniformDist(0, mapSize);
  for ( long long i=0 ; i< accSize ; ++i) randValues.push_back(uniformDist(engine));

  // I know i have to pay for the randValues access.
```

```
  auto start = std::chrono::system_clock::now();
  for ( long long i=0; i < accSize; ++i){
    myMap[randValues[i]];

  }
  std::chrono::duration<double> dur= std::chrono::system_clock::now() - start;
  std::cout << "time for std::map: " << dur.count() << " seconds" << std::endl;

  auto start2 = std::chrono::system_clock::now();
  for ( long long i=0; i < accSize; ++i){
    myHash[randValues[i]];
  }
  std::chrono::duration<double> dur2= std::chrono::system_clock::now() - start2;
  std::cout << "time for std::unordered_map: " << dur2.count() << " seconds" << std::endl;

  std::cout << std::endl;
}
```

Explanation

- The key difference between `std::map` and `std::unordered_map` is that the `std::map` has logarithmic access time, but the `std::unordered_map` has amortized constant access time. The example provides numbers to this performance difference.

- The central idea of the performance test is to create a map/unordered_map with 1000000 elements and to read 500000 arbitrary elements.

- The lines 18 - 21 fill the `std::map` and the `std::unordered_map`.

- `randValues` is the vector with 1000000 entries from 0 to 1000000. The values are uniformly distributed (line 29).

- The lines 34 - 36 use the `std::map` and the lines 41 - 43 use the `std::unordered_map`. Due to the new time library, it is easy to measure the two time points and get the past time.

Example 5 - Unordered Set with Hash

```
// unorderedSetHashInfo.cpp
#include <iostream>
#include <random>
#include <unordered_set>

void getInfo(const std::unordered_set<int>& mySet){
```

```
void getInfo(const std::unordered_set<int>& mySet){

  std::cout << "mySet.bucket_count(): " << mySet.bucket_count() << std::endl;
  std::cout << "mySet.load_factor(): " << mySet.load_factor() << std::endl;

}

void fillMySet(std::unordered_set<int>& h, int n){

  std::random_device seed;
  // default generator
  std::mt19937 engine(seed());
  // get random numbers 0 - 1000
  std::uniform_int_distribution<> uniformDist(0, 1000);

  for (int i = 1; i<= n; ++i){
    h.insert(uniformDist(engine));
  }

}

int main(){

  std::cout << std::endl;

  std::unordered_set<int> mySet;
  std::cout << "mySet.max_load_factor(): " << mySet.max_load_factor() << std::endl;

  std::cout << std::endl;

  getInfo(mySet);

  std::cout << std::endl;

  // only to be sure
  mySet.insert(500);
  // get the bucket of 500
  std::cout << "mySet.bucket(500): " << mySet.bucket(500) << std::endl;

  std::cout << std::endl;

  // add 100 elements
  fillMySet(mySet, 100);
  getInfo(mySet);

  std::cout << std::endl;

  std::cout << "---------------" << std::endl;

  auto numBuck = mySet.bucket_count();

  std::cout << "mySet.bucket_count(): " << mySet.bucket_count();

  std::cout << "\n\n\n";

  for (std::size_t i = 0; i < numBuck; ++i){
      std::cout << "mySet.bucket_size(" << i << "): " << mySet.bucket_size(i) << std::endl;
      for (auto it = mySet.begin(i); it != mySet.end(i); ++it) std::cout << *it <<  " ";
      std::cout << std::endl;
   }

  std::cout << " ---------------"  << "\n\n";
```

```
    // at least 500 buckets
    std::cout << "mySet.rehash(500): " << std::endl;
    mySet.rehash(500);

    std::cout << std::endl;

    getInfo(mySet);

    std::cout << std::endl;

    // get the bucket of 500
    std::cout << "mySet.bucket(500): " << mySet.bucket(500) << std::endl;

    numBuck = mySet.bucket_count();
    std::cout << "mySet.bucket_count(): " << mySet.bucket_count() << std::endl;

    std::cout << std::endl;

}
```

Explanation

- The example shows the internal working of `std::unordered_set`. The behavior would be similar for each other unordered associative container.
- The function `getInfo` (lines 6 - 11) is a convenience function which returns both the number of buckets and the load factor for a given `std::unordered_set`.
- The program shows the `max_load_factor`, which returns the value when a rehashing would occur.
- In line 41, the program inserts 500 into the `mySet`, and line 43 returns the bucket in which 500 is stored.
- In line 48, the function call `fillMySet` causes 100 elements between 0 and 1000 to be added to `mySet`.
- Due to the number of buckets `numBuck`, it is possible to show how many and which elements are inside each bucket. This occurs in lines 61 - 65.
- A call `mySet.rehash(500)` in line 71 creates at least 500 buckets for `mySet`, meaning that all elements are distributed in new buckets.

Let’s solve an exercise for associative containers in the next lesson.