The Details

We'll finish this section by learning the terminology present in unordered associative containers.

The unordered associative containers store their indices in buckets. In which bucket the index goes depends on the hash function, which maps the key to the index. If different keys are mapped to the same index, it's called a collision. The hash function tries to avoid this.

Indices are typically stored in the bucket as a linked list. Since the access to the bucket is constant, the access to the linked list in the bucket is linear. The number of buckets is called *capacity*, the average number of elements for each bucket is called the *load factor*. In general, the C++ runtime generates new buckets if the load factor is greater than 1. This process is called *rehashing* and can also be triggered explicitly:

```
// hashInfo.cpp
#include <iostream>
#include <unordered_set>
using namespace std;

void getInfo(const unordered_set<int>& hash){
  cout << "hash.bucket_count(): " << hash.bucket_count();
  cout << "hash.load_factor(): " << hash.load_factor();
}

int main(){

  // Create an unoredered set and initialize it with the array
        // Set will contain only random elements
  int arr[100];
  for(int i=0; i<100; i++)
    arr[i] = (rand() % 100) + 1;
  unordered_set<int> hash(arr, arr + sizeof(arr) / sizeof(int));
  cout << "hash.max_load_factor():\t" << hash.max_load_factor() << endl; // hash.max_load_fac

  getInfo(hash);
  //hash.bucket_count(): 103hash.load_factor(): 0.660194
  cout<<endl;

  hash.insert(500);
  cout << "hash.bucket(500):\t" << hash.bucket(500) << endl; // 88

  getInfo(hash);
```

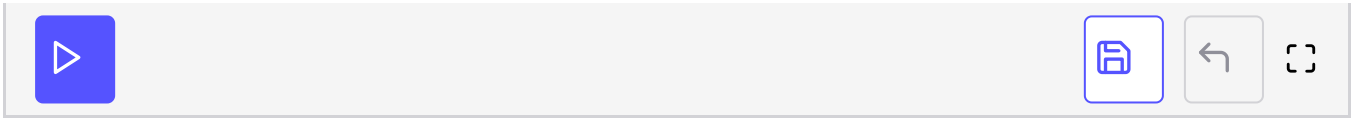
```
  cout<<endl;
  //hash.bucket_count(): 103hash.load_factor(): 0.669903


  hash.rehash(500);

  getInfo(hash);
  //hash.bucket_count(): 503hash.load_factor(): 0.137177500

  cout << endl << "hash.bucket(500):\t" << hash.bucket(500); //hash.bucket(500):        500

  return 0;
}
```



Details to the hash function

With the method `max_load_factor`, you can read and set the load factor in order to influence the probability of collisions and rehashing. I want to emphasize one point in the short example above: the key 500 is in the 5th bucket at first, but after rehashing, it is in the 500th bucket.

In the next lesson, we'll solve an exercise related to `std::unordered_set` and `std::unordered_multiset`.