Associative Containers: The Hash Function

The hash function maps a potentially infinite number of keys onto a finite number of buckets. What is the strategy of the C++ runtime, and how can you tailor it to your needs? Let's disucss these questions further.

WE'LL COVER THE FOLLOWING

Which requirements must the key and the value of an unordered associative container fulfill?

The **key** must be

- comparable with the equality function,
- available as a hash value,
- copyable and moveable.

The **value** must be

- by default constructible,
- copyable and moveable.

A **hash function** is good if two factors are present: one, if the mapping from the keys to the values produces few collisions; two, if the hash values are uniformly distributed among the buckets. Since the execution time of the hash function is *constant*, the access time of the elements is also constant. Instead, the access time in the bucket is *linear*, meaning that the overall access time of a value depends on the number of collisions in the bucket.

The Hash Function

A hash function

A hash function

- is available for the fundamental data types, such as booleans, integers, and floating points.
- is available for the data types `std::string` and `std::wstring`,
- creates a hash value of the pointer address for C string const `char*`
- can be defined for user-defined data types.

By applying the theory to user-defined data types, which we want to use as a key of an unordered associative container, our data type needs to fulfill *two* requirements:

1. It should have a hash function and
2. an equality function.

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

Rehashing

The hash function decides the bucket to which the key goes. Because the hash function reduces a potentially infinite number of keys on a finite number of buckets, the various keys can go to the same bucket. This event is called collision. The keys in each bucket are typically stored as a singly linked list. With this in mind, you can reason how quickly the access time on a key is in an unordered associative container. The application of the hash function is a constant operation, and the searching of the key in the singly linked list is a linear operation. Therefore, the goal of the hash function is it to produce a few collisions.

The number of buckets is called the capacity. The average number of elements in each bucket is the load factor. By default, the C++ runtime creates new buckets if the load factor goes beyond 1. This process is called rehashing. It can be explicitly triggered by setting the capacity of the unordered associative container to a higher value.

Capacity: Number of buckets.

Load factor: Average number of elements (keys) per bucket.

Rehashing: Creation of new buckets if load factor > 1.

Rule of Thumb 👍

If you know how large your unordered associative container will become, you can start with a reasonable number of buckets. Therefore, you spare expensive rehashings since each rehashing includes memory allocation and the new distribution of all keys. This leaves us with the question, what is reasonable? You must start with a bucket size similar to the number of your keys. Therefore, your load factor is close to 1.

In the next lesson, we will compare the performance of ordered and unordered associative containers using an interesting example.