I will summarize the knowledge about hash from two perspectives.
#source: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/12-hashing.pdf
#source: http://en.cppreference.com/w/cpp/container/unordered_map

First, from theoretical perspective.

Properties about hash function:
-load factor: number of elements stored / number of buckets
-uniformity: Pr[h(x)=i] = 1/m for all x and all i
-universal: Pr[h(x)=h(y) | x!=y] = 1/m
-near-universal: Pr[h(x)=h(y) | x!=y] <= 2/m
-k-uniform/k-universal: for any sequence of k disjoint keys and any sequence of k hash values, the probability that each
key maps to the corresponding hash value is 1/(m^k).
-Multiplicative Hashing: using modular arithmetic with prime numbers/ powers of two
a common example (near-universal): #define hash(a,x) ((a)*(x) >> (WORDSIZE-HASHBITS))
"a" (secret salt) is an odd integer smaller than the largest element, the shift only reserves the first several bits that can be fit in the hash table.

Methods for collision:
-chaining (array of linked lists)
-open addressing: set of hash functions, try them one by one (all the followings are examples):
-linear probing: hi(x) := (h(x) + i) mod m (looks for the next empty spot)
-binary probing: hi(x) := h(x) xor i
-quadratic probing: single hash function h and set hi(x) := (h(x) + i^2) mod m
-double hashing: two hash functions h and h_0 and set hi(x) := (h(x) + i · h_0(x)) mod m

About Perfect Hashing:
If we use a small hash table to keep the space usage down, even if we use ideal random hash functions, the resulting worst-case expected search time is Θ(log n/ loglog n) with high probability, which is not much better than a binary search tree.
On the other hand, we can get constant worst-case search time, at least in expectation, by using a table of roughly quadratic size, but that seems unduly wasteful. Fortunately, there is a fairly simple way to combine these two ideas to get a data structure of linear expected size, whose expected worst-case search time is constant. At the top level, we use a hash table of size m = n, but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the jth secondary hash table has size 2*(n_j)^2, where n_j is the number of items whose primary hash value is j. The worst-case search time in any secondary hash table is O(1). (If we discover a collision in some secondary hash table, we can simply rebuild that table with a new near-universal hash function.) Although this data structure apparently needs significantly more memory for each secondary structure, the overall increase in space is insignificant, at least in expectation.

Second, from the perspective of implementation of hash tables (unordered_map) in C++ STL.
It uses chaining.
You can specify the max_load_factor (default is 1). Resize happens when the load factor reaches the max_load_factor threshold.
During resize: roughly doubles its size. new_number_of_buckets = old_number_of_buckets * 2 + k (k is a small number that could change from the experience of experiment, sometimes 1).
O(n) operation happens, new hash function calculated. All iterators becomes invalid.
You can use rehash() to specify the number of buckets you want in advance, thus avoiding resizing. (Similar to reserve() in vector.) reserve() in unordered_map is the same as rehash(std::ceil(count / max_load_factor())).