Range-based for loop in C++

Range-based for loop in C++ is added since C++ 11. It executes a for loop over a range. Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

**Syntax :**

**for ( range\_declaration : range\_expression )**

**loop\_statement**

**Parameters :**

**range\_declaration :**

a declaration of a named variable, whose type is the

type of the element of the sequence represented by

range\_expression, or a reference to that type.

Often uses the auto specifier for automatic type

deduction.

**range\_expression :**

any expression that represents a suitable sequence

or a braced-init-list.

**loop\_statement :**

any statement, typically a compound statement, which

is the body of the loop.

C++ implementation :

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Illustration of range-for loop  // using CPP code  #include <iostream>  #include <vector>  #include <map>    //Driver  int main()  {      // Iterating over whole array      std::vector<int> v = {0, 1, 2, 3, 4, 5};      for (auto i : v)          std::cout << i << ' ';        std::cout << '\n';        // the initializer may be a braced-init-list      for (int n : {0, 1, 2, 3, 4, 5})          std::cout << n << ' ';        std::cout << '\n';        // Iterating over array      int a[] = {0, 1, 2, 3, 4, 5};      for (int n : a)          std::cout << n << ' ';        std::cout << '\n';        // Just running a loop for every array      // element      for (int n : a)          std::cout << "In loop" << ' ';        std::cout << '\n';        // Printing string characters      std::string str = "Geeks";      for (char c : str)          std::cout << c << ' ';        std::cout << '\n';        // Printing keys and values of a map      std::map <int, int> MAP({{1, 1}, {2, 2}, {3, 3}});      for (auto i : MAP)          std::cout << '{' << i.first << ", "                    << i.second << "}\n";  } |

Output:

0 1 2 3 4 5

0 1 2 3 4 5

0 1 2 3 4 5

In loop In loop In loop In loop In loop In loop

G e e k s

{1, 1}

{2, 2}

{3, 3}

Hashing | Set 1 (Introduction)

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With**balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.  
This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need O(m \* 10n) space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

*Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.*

**[Hash Function](http://en.wikipedia.org/wiki/Hash_function" \t "_blank):** A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.  
A good hash function should have following properties  
1) Efficiently computable.  
2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

**[Hash Table](http://en.wikipedia.org/wiki/Hash_table" \t "_blank):** An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

**Collision Handling**: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

* **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
* **Open Addressing:**In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

**What is Collision?**  
Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

**What are the chances of collisions with large table?**  
Collisions are very likely even if we have big table to store keys. An important observation is [Birthday Paradox](https://www.geeksforgeeks.org/birthday-paradox/). With only 23 persons, the probability that two people have the same birthday is 50%.

**How to handle Collisions?**  
There are mainly two methods to handle collision:  
1) Separate Chaining  
2) Open Addressing  
In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post.

**Separate Chaining:**  
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

**[C++ program for hashing with chaining](https://www.geeksforgeeks.org/c-program-hashing-chaining/)**

**Advantages:**  
1) Simple to implement.  
2) Hash table never fills up, we can always add more elements to the chain.  
3) Less sensitive to the hash function or load factors.  
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

**Disadvantages:**  
1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.  
2) Wastage of Space (Some Parts of hash table are never used)  
3) If the chain becomes long, then search time can become O(n) in the worst case.  
4) Uses extra space for links.