Iterator Design Pattern

Group number 9 : Võ Trần Thanh Lương - Nguyễn Trần Trọng Tâm – Tô Minh Thành

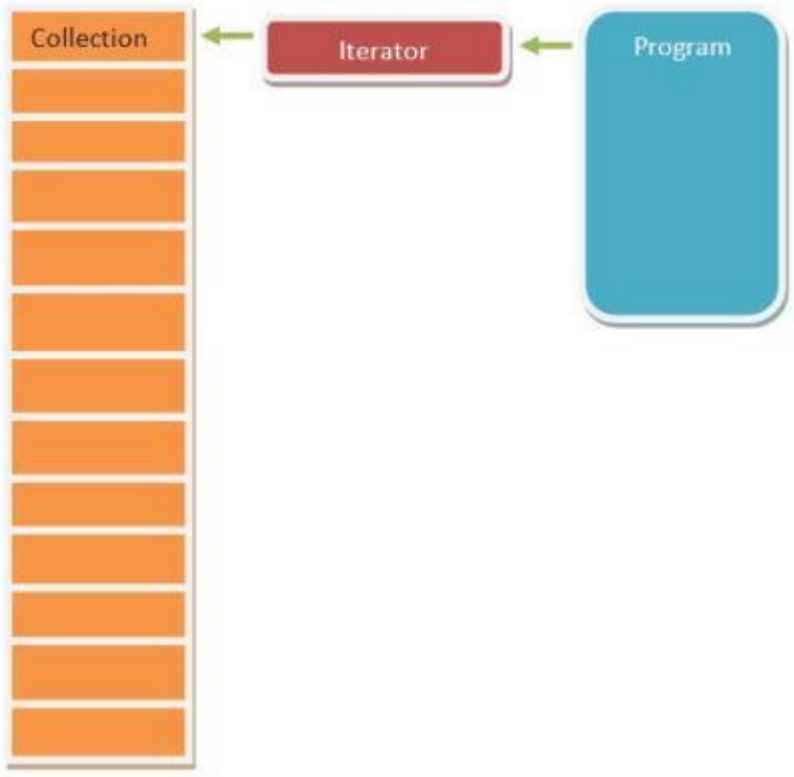
Table Of Contents

I. What is an iterator ?
II. What is iterator design pattern ?
III. Implementation in C++?

I. What is an iterator ?

Iterator is a pointer-like object. It can be increase with prefix or suffix increment, dereference with * and compared against another operator with !=. Iterators are often fundamental when learning C++ Standard Template Library as it provide a means for accessing data stored in containers. Declaring an iterator is easy:

```
vector<int> IntVector;
vector<int>::iterator IntVectorIterator;
```



Iterators are classified into five categories depending on the functionality they implement:

- Input iterator
- Output iterator
- Forward iterator
- Bidirectional iterator
- Random access itertor

> But why do we use iterators instead of indices ?

Because of Performance

Imagine you are standing at the beginning of a train. Your job is to run from the first compartment to the last compartment, to find a passenger A. For a train that has a certain number of compartment, the job is quite easy. But for a extremely long train, counting how many compartments/passengers there are would kill you before you even do the job. Also, sometimes you dont have to reach the end to find the passenger. So why do you have to know how many compartments/passengers there are ? Why dont you just start the job and save the time.

Iterator in Cpp inherits the same logic. With iterators, you won't have to store the size of the container (which is a not-always-fast operation)/ For a vector, using index

may still be fine (but not recommended) - your vector's size may be controllable. But for other containers like list, map, ... iterator is the best choice. It's more generic and plays well with algorithms. Also, some operations for the containers takes the iterator in as parameters. They are more precise and convenient than indices.

II. What is iterator design pattern ?

The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements in object-oriented programming. This pattern is used to get a way to access the elements of a collection object in sequential manner without exposing its underlying representation. Also, it promotes "full object status" to the traversal of a collection.

The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal method.

Also, as said above, since iterators are very generic, the concept of iterators is essential to “generic programming”. The iterator design pattern utilizes this as a way to help you with “generic programming”, gives you the right tool to support multiple “data structures” and “algorithms” without any hassle.

A real world example of when to apply iterator design pattern :

Suppose we have a list of Radio channels and the client program want to traverse through them one by one or based on the type of channel. For example some client programs are only interested in English channels and want to process only them, they don’t want to process other types of channels.

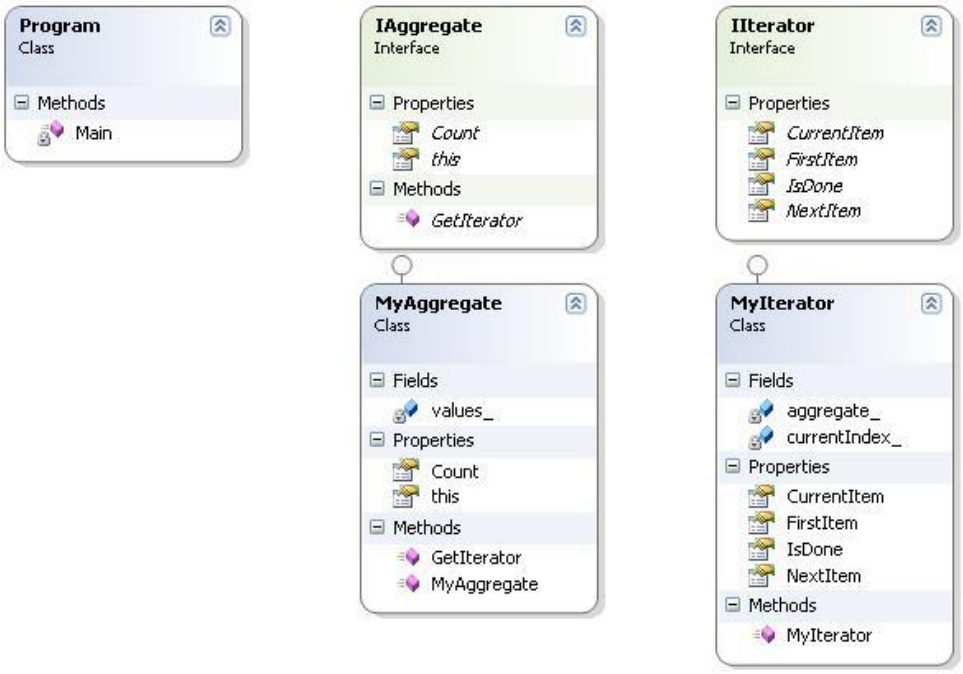
So we can provide a collection of channels to the client and let them write the logic to traverse through the channels and decide whether to process them. But this solution has lots of issues such as client has to come up with the logic for traversal.

We can’t make sure that client logic is correct. Furthermore if the number of client grows then it will become very hard to maintain.

Here we can use Iterator pattern and provide iteration based on type of channel. We should make sure that client program can access the list of channels only through the iterator.

III. Implementation in C++?

C++ implements iterators with the pointers (or exactly, smart pointers). All of the pointer operations can be overloaded, which is why iterator can be implemented in such a way that it relates to pointer. Iterators enhances pointer’s dereference, increment, and decrement. This has the advantage that C++ algorithms such as std::sort can immediately be applied to plain old memory buffers, and that there is no new syntax to learn. However, it requires an "end" iterator to test for equality, rather than allowing an iterator to know that it has reached the end.



Every STL container in C++ implements their own iterators. STL algorithms are defined in terms of iterators.

Java implements iterators design pattern through Iterator interface which narrates navigation method and a Container interface

which returs the iterator. Concrete classes implementing the Container interface will be responsible to implement Iterator interface and use it.

There is a large conceptual difference between the implementation of C++ and Java. C++ utilizes different "classes" of iterators. Some are used for random access, some are used for forward access, others are used for writing data. It is more powerful than Java iterator overall.