Iterator in C++

Iterator is a behavioral design pattern that allows sequential traversal through a complex data structure without exposing its internal details.

Thanks to the Iterator, clients can go over elements of different collections in a similar fashion using a single iterator interface.

Learn more about Iterator

Complexity:

Popularity:

Usage examples: The pattern is very common in C++ code. Many frameworks and libraries use it to provide a standard way for traversing their collections.

Identification: Iterator is easy to recognize by the navigation methods (such as `next`, `previous` and others). Client code that uses iterators might not have direct access to the collection being traversed.

Conceptual Example

This example illustrates the structure of the **Iterator** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```
/**
 * Iterator Design Pattern
 *
 * Intent: Lets you traverse elements of a collection without exposing its
```

```
 * underlying representation (list, stack, tree, etc.).
 */

#include <iostream>
#include <string>
#include <vector>

/**
 * C++ has its own implementation of iterator that works with a different
 * generics containers defined by the standard library.
 */

template <typename T, typename U>
class Iterator {
 public:
  typedef typename std::vector<T>::iterator iter_type;
  Iterator(U *p_data, bool reverse = false) : m_p_data_(p_data) {
    m_it_ = m_p_data_->m_data_.begin();
  }

  void First() {
    m_it_ = m_p_data_->m_data_.begin();
  }

  void Next() {
    m_it_++;
  }

  bool IsDone() {
    return (m_it_ == m_p_data_->m_data_.end());
  }

  iter_type Current() {
    return m_it_;
  }

 private:
  U *m_p_data_;
  iter_type m_it_;
};

/**
 * Generic Collections/Containers provides one or several methods for retrieving
 * fresh iterator instances, compatible with the collection class.
 */

template <class T>
class Container {
  friend class Iterator<T, Container>;

 public:
  void Add(T a) {
```

```
    m_data_.push_back(a);
  }

  Iterator<T, Container> *CreateIterator() {
    return new Iterator<T, Container>(this);
  }

 private:
  std::vector<T> m_data_;
};

class Data {
 public:
  Data(int a = 0) : m_data_(a) {}

  void set_data(int a) {
    m_data_ = a;
  }

  int data() {
    return m_data_;
  }

 private:
  int m_data_;
};

/**
 * The client code may or may not know about the Concrete Iterator or Collection
 * classes, for this implementation the container is generic so you can used
 * with an int or with a custom class.
 */
void ClientCode() {
  std::cout << "_________________Iterator with int_____________________________________" << s
  Container<int> cont;

  for (int i = 0; i < 10; i++) {
    cont.Add(i);
  }

  Iterator<int, Container<int>> *it = cont.CreateIterator();
  for (it->First(); !it->IsDone(); it->Next()) {
    std::cout << *it->Current() << std::endl;
  }

  Container<Data> cont2;
  Data a(100), b(1000), c(10000);
  cont2.Add(a);
  cont2.Add(b);
  cont2.Add(c);

  std::cout << "_________________Iterator with custom Class_____________________________" << 
```

```
  Iterator<Data, Container<Data>> *it2 = cont2.CreateIterator();
  for (it2->First(); !it2->IsDone(); it2->Next()) {
    std::cout << it2->Current()->data() << std::endl;
  }
  delete it;
  delete it2;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
_________________Iterator with int_______________________________________
0
1
2
3
4
5
6
7
8
9
_________________Iterator with custom Class_____________________________
100
1000
10000
```