This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

« 2 Sequential containers Course materials 4 Associative Containers »
ELEC-A7151 / Module 2: Containers / 3 Iterators

3 Iterators¶

- **You can download the template for the programming tasks of the module as a** `zip` **file from** `this link`.

Contents

C++ Primer Chapter: 9.2.1 (Iterators)

Iterator is a pointer to an element within a container. For some types of containers, iterator is the only way to access the container, when direct indexing using `[]` or function is not possible, such as with linked list. Commonly iterators are used to walk through the elements in container, for example in for loop, but they can be used were ever individual elements need to be pointed. For example, some function calls use iterators as parameters.

C++ Primer Chapter: 9.2.3 (Begin and end members)

Iterator is type named in the scope of the container namespace, and each container type has a dedicated iterator type. For example, the basic iterator type for *std::vector<int>* is `std::vector<int>::iterator`, and the constant iterator is `std::vector<int>::const_iterator`. One can obtain the iterator corresponding to a container class by using the **begin** function of the container, which returns an iterator that points to the first element of the container. Alternatively, **end** returns iterator pointing to the end of the container, more precisely one member after the last member, i.e., trying to access an element at the *end* iterator is an error.

Iterators can be compared for equality. If `c.begin() == c.end()`, it means that *c* is empty. Here is a basic example of using iterators, a rewrite of the *LargestNumber* function from Module 1, in a better way:

```
1#include <vector>
2#include <iostream>
3#include <limits>
4
5// Note that argument numbers is declared as const: the contents of the container
6// cannot be modified by this function
7int LargestNumber(const std::vector<int>& numbers) {
8    int largest = std::numeric_limits<int>::lowest(); // the lowest value of an int
9
10    // we will have to use 'const_iterator', because argument numbers is const
11    // Note that we could have use 'auto' type as well, to save typing
12    for (std::vector<int>::const_iterator it = numbers.begin(); it != numbers.end(); it++) {
13
14        // iterator it is dereferenced as if it was a pointer, using the * operator
15        if (*it > largest) {
16            largest = *it;
17        }
18    }
19    return largest;
20}
21
22int main(void) {
23    std::vector<int> numbers = { 1, 2, 3 };
24
25    numbers.push_back(5);
26    numbers.push_back(7);
27
28    std::cout << "Size: " << numbers.size() << std::endl
29        << "Largest: " << LargestNumber(numbers) << std::endl;
30}
```

Line 12 first declares a new iterator variable inside the *for* statement. This is of **const_iterator** type, because vector *number* is a const argument in the function. I.e., the function cannot modify the vector contents. The regular *iterator* type can only be used for modifiable containers. The for loop initializes iterator *it* using the *begin* function. The loop ends when iterator *it* becomes equal to `numbers.end()`.

Operators `++`, `--`, `+` and `-` are overloaded to move the iterator forward or backward by one or more steps, behaving quite similarly to pointer arithmetics in C. With this being told, the functionality of the iterator *it* in the for loop in the *LargestNumber* function should now be more apparent: the loop simply walks through all members of vector *number*.

For accessing the element pointed by iterator, the dereference operator `*` is used, as if the iterator was a traditional C pointer (which it is not). The *LargestNumber* example shows this on lines 15 and 16.

C++ Primer Chapter: 9.3.3 (Erasing elements)

Iterator can be used as a parameter in functions, as any other type, when individual data elements in a container need to be pointed at. For example, the *vector* type has a function, **erase**, that takes an iterator as a parameter. This function causes the pointed item to be deleted from the vector (example follows a bit later).

Because an iterator behaves much like a pointer, iterator arithmetics also work similarly to pointer arithmetics, also for larger increments and decrements than one. The below example shows the *erase* function in use, together with pointer arithmetics. Test it, and try modifications if you wish.

```
1#include <string>
2#include <iostream>
3
4int main() {
5    // string is just a vector of characters
6    std::string str = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'};
7
8    // could use 'auto' type, if we wanted. 'it' is now at the beginning
9    std::string::iterator it = str.begin();
10
11    // remove first character, 'B' will be the first
12    it = str.erase(it);
13    std::cout << "After removing str.begin(): " << str << std::endl;
14
15    // remove second and third character (after 'B')
16    str.erase(it + 1, it + 3);
17    std::cout << "After removing more characters: " << str << std::endl;
18}
```

Iterators can be used also for *string*, which can be thought of as a vector of characters. Line 9 in the above example defines an iterator we use for processing *str*.

The *erase* function can take one or two iterators as a parameter. The one-parameter variant on line 12 removes one element from a container, as pointed by the iterator. The function returns an iterator that points to the location *after* the removed element. Using the return value is important, because **removal may invalidate the previously used iterator**.

Line 16 shows an *erase* variant with two iterators as parameter. This function will remove multiple elements from the container, as indicated by the start and end iterators, and return the iterator pointing to element following the removed block. The iterator marking the end of range is not included in the removed set. I.e., line 16 removes characters 'C' and 'D'.

Addition and removal of elements in the container may invalidate iterators and references/pointers to the container elements. For example, sometimes the space allocated for vector needs to be reallocated for addition, which makes the earlier references invalid. The behavior depends on the container type, and how it internally stores the data. To understand this, it might be helpful to recall (e.g. from the C course) how dynamically allocated arrays or linked lists are implemented.

Another container function that uses iterator as a parameter is `insert_after(it, e)`, where *it* is the iterator after which a new element *e* is inserted. It returns an iterator that points to the last inserted element.

3.1 Range for¶

C++ Primer Chapter: 5.4.3 (Range for statement)

To make things easier, C++ defines a simpler form of the for loop that can be applied to containers or other sequential constructs. The **range for** can be applied to container types, and it is a short way of telling the compiler that 'we want to go through all elements in container'.

Below is a rewrite of the earlier *LargestNumber* function as an example how the **range for** loop works. In addition it uses the *auto* type to make the for statement even shorter:

```
int LargestNumber(const std::vector<int>& numbers) {
    int largest = std::numeric_limits<int>::lowest();
    for (auto i : numbers) { // i is an 'int'
        if (i > largest)
            largest = i;
    }
    return largest;
}
```

The for loop now uses the *auto* declaration specifier to automatically specify the right type for variable *i*. In addition to this, we just use colon (':'), and the name of the container (in this case *numbers*) to be processed in the loop. The range for statement will then go through each element in the container, with variable *i* holding an integer in the vector. Note that *i* now stands for plain integer in the vector, not an iterator pointing to an integer value.

3.2 Programming tasks¶

Points **10 / 10** My submissions **11** Deadline Friday, 8 October 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 17 December 2022, 19:59).

Vector iterators

Objective: Learn to use iterators with vectors.

Implement three functions that operate on integer vectors in the following manner:

ReadVector that will read numbers from user and stores them as integer vector that is returned by the function. The function stops reading when a non-numeric value is given by the user.

Hint

C++ Primer: Chapter 1.4.3 (Reading an unknown number of inputs)

PrintSum1 that calculates sums of pair of numbers in the vector and prints them on the screen. The function will print the sum of two consecutive numbers in vector, separated by space. For example, if the parameter vector contains the following numbers: *1 2 3 4*, the output should be *3 5 7*, followed by a space and a newline character. As can be seen, the output will have one number less than the input.

PrintSum2 that calculates sum of the first and last item in the vector, then the sum of second and second-last item in the vector, and so on, until all integers in the vector have been processed. For example, with the input *1 2 3 4* the output will be *5 5*, followed by a space and a newline.

Use iterators for walking through the vectors.

Instructions on how to run and test your programs locally are available in Getting Started Module.

vector_it.cpp

Choose File | No file chosen

Submit

Points **10 / 10** My submissions **7** Deadline Friday, 8 October 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 17 December 2022, 19:59).

List

Objective: Understanding the basic handling of list, and using input stream as a source.

This exercise reads series of lines into a list container from input stream, that can be the standard input, or for example a file. You will need to implement the following functions:

- **GetLines(is, list)** that will empty the given list *list*, read lines from the given input stream *is*, and add each line to list *list*. You will find function getline useful for reading lines from the input stream. The function will read as long as the input stream is readable. See istream reference for ideas how to check the input stream state.

  Note

  1. A common problem is that program tries to add one more item to the list after a newline character that is at the end of the file. This should not be done.
  2. Grader tests do not ensure that the list is empty before calling the function. Ensure that the list is empty (see *clear()* function) before you start reading lines.
- **Print(list)** that will print each string in the list on a separate line to the standard output stream.
- **SortAndUnique(list)** that will sort the list into alphabetical order and remove duplicate strings. See list reference for further information.

Instructions on how to run and test your programs locally are available in Getting Started Module.

list.cpp

Choose File | No file chosen

Submit

Points **11 / 15** My submissions **6** Deadline Friday, 8 October 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 17 December 2022, 19:59).

Matrix rotation

Objective: Use nested vectors for a two-dimensional matrix.

This exercise operates on two-dimensional integer matrix that is composed of vector containers. Conceptually the idea is not much different from two-dimensional arrays in C, but now we use C++ vector along with its functions and iterators. Like C arrays, vectors can be accessed using the subscript operator, and when two nested vectors are used for two-dimensional matrix, you need to provide two indexes, for both dimensions respectively.

Implement the following functions:

ReadMatrix(n) that will read a square matrix from user (i.e., standard input). Argument *n* (int) gives the width and height of the matrix, i.e., you must read *n^2* integers, and then return the resulting Matrix. Pay attention to how the Matrix type is defined in *matrix.hpp*.

Rotate90Deg(m) that will rotate Matrix *m* clockwise, and return the resulting matrix. The function should not modify the old matrix, but create a new one.

Print(m) that will output Matrix *m* to the screen, starting with a line `Printing out a n x n matrix:` where *n* is replaced by the width and height of the matrix. Following that, print the rows of the matrix: each integer is followed by one space. Each row must be printed on separate line.

Instructions on how to run and test your programs locally are available in Getting Started Module.

matrix.cpp

Choose File | No file chosen

Submit

Points **30 / 30** My submissions **20** Deadline Friday, 8 October 2021, 19:59 To be submitted alone

⚠ This course has been archived (Saturday, 17 December 2022, 19:59).

Library

Objective: More practice with containers.

In this exercise, you will implement three classes:

- **Book** which holds information about a book
- **Customer** which holds information about a customer
- **Library** which collects together books, and customers loaning the books

You will need to start from finishing the definitions in the *.hpp* files according to the instructions in the comments, and then implement the functions in the *.cpp* files.

The tests for *Customer* and *Library* require full points from Book tests and *Library* tests requires full points from *Customer* tests. Therefore, the recommended order of implementing is *Book*, *Customer*, and then *Library*. Getter functions are also required to get full points for the other tests to be run. Note that you still need to at least make dummy declarations and definitions for the methods or you will not be able to compile the tests. You could also test them yourself using the main function, in which case you do not need dummy definitions for unused methods.

Instructions on how to run and test your programs locally are available in Getting Started Module.

book.hpp

Choose File | No file chosen

book.cpp

Choose File | No file chosen

customer.hpp

Choose File | No file chosen

customer.cpp

Choose File | No file chosen

library.hpp

Choose File | No file chosen

library.cpp

Choose File | No file chosen

Submit