Containers

June 20, 2023

1 Containers and iterators

Containers and iterators are part of the C++ STL.

The **containers** are data structures which contain data. Can be seen as a collection of objects of a certain datatype.

C++ provides 2 different types of containers depending on how the information is ordered in the allocation: **sequence containers** and **associative containers**.

The **iterators** are used to step through the elements of the collection of objects.

1.1 Sequence containers

The elements stored in the container can be accessed sequentially.

Three types of sequence containers: **array**, **vector**, **deque** and **list**.

[ ]:
```
#include <iostream>
#include <string>

#define N 6
```

1.1.1 1. Array

- Implements a compile-time non-resizeable array and encapsulates fixed size arrays.
- It does not keep any data other than the elements it contains, even not its size.

Properties

- **Sequence**: The elements are ordered in a strict linear sequence. Individual elements are accessed only by their position.
- **Contiguous storage**: The elements are stored in contiguous memory locations. Pointers to an element can be offset to access other elements.
- **Fixed-size**: Has a fixed size and do not manage the allocation of its elements through an allocator: it can not be expanded or contracted.

Using arrays

1. Including the header Array has to be included in order to being able to use it:

[ ]:
```
#include <array>
```

2. Creating the array container The array container can be created ba calling its parameter constructor or calling the copy constructor.

[ ]:
```
std::array<int, 3> a1{1, 2, 3};
std::array<std::string, 4> a2={"a", "b", "c", "d"};
```

3. Using the array container We can use the container as a normal datatype.

[ ]:
```
for (const auto& i:a1)          //for (int i=0; i<3; i++)
    std::cout << i << " ";      // std::cout << a1[i] << " ";
std::cout << std::endl;

for (const auto& s: a2)
    std::cout << s << " ";
std::cout << std::endl;
```

1.1.2 2. Vector

- The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements.
- Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Vectors are good at: * Accessing individual elements by their position index. * Iterating over the elements in any order. * Add and remove elements from its end.

Properties

- **Sequence**: Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
- **Dynamic array**: Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.
- **Allocator-aware**: The container uses an allocator object to dynamically handle its storage needs.

Methods provided by the class

- Constructor: creates the container.
- Destructor: frees the container and its elements.
- operator=: copy constructor.
- ::begin : points to the initial value (iterator).
- ::end : points to the final value (iterator).
- size: number of items.
- empty: boolean value to know if it is empty or not.
- operator[] : element at a certain position.
- at: returns the element at position n in the vector.

- push_back: inserts an element at the end.
- pop_back: removes the element at the end.

Using the vector class

1. Including the header Vectors have to be included before its first use:

[ ]:
```
#include <vector>
```

2. Creating the vector container The vector object can be created ba calling its empty, parameter or copy constructor.

[ ]:
```
std::vector<int> myVector;
std::vector<int> v = {7, 5, 16, 8};
std::vector<int> v2 (v);
```

3. Using the vector container We can use the container as a normal datatype.

[ ]:
```
for (int i=0; i<N; i++)
    myVector.push_back(i);

std::cout << "First: " << myVector.front() << " Last: " << myVector.back()
          << " Middle: " << myVector.at(N/2) << " Size: " << myVector.size()
          << std::endl << std::endl;

v.push_back(25);
v.push_back(13);

std::cout << "Vector v: ";
for (int n : v)
    std::cout << n << " ";
std::cout << std::endl;

std::cout << "Vector v2: ";
for (int n : v2)
    std::cout << n << " ";
std::cout << std::endl;
```

Open question What happens in this example?:

[ ]:
```
std::vector<int> myVector (5);

for (int i=0; i<N; i++)
    myVector.push_back(i);

std::cout << "First: " << myVector.front() << " Last: " << myVector.back()
```

```
          << " Middle: " << myVector.at(N/2) << " Size: " << myVector.size() <<␣
 ↪std::endl;
```

[ ]:
```
for (int n : myVector)
    std::cout << n << ", ";
std::cout << std::endl;
```

1.1.3 3. List

- Lists allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists

Lists are good at: * Inserting, extracting and moving elements in any position within the container. * Resizing: made without moving all the elements to a new container.

The main drawback of lists compared to vector containers is that they lack direct access to the elements by their position.

Properties

- **Sequence**: Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
- **Doubly-linked list**: Each element keeps information on how to locate the next and the previous elements, allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access.
- **Allocator-aware**: The container uses an allocator object to dynamically handle its storage needs.

Methods provided by the class

- Constructor: creates the container
- Destructor: frees the container and its elements.
- operator=: copy constructor.
- ::begin : points to the initial value.
- ::end : points to the final value.
- size: number of items.
- empty: boolean value to know if it is empty or not.
- push_front, push_back: inserts an element at head or end.
- pop_front, pop_back: retrieves the element at the head or end.

Using the list class

1. Including the header Lists have to be included before its first use:

[ ]:
```
#include <list>
```

2. Creating the list container The list object can be created by calling its empty, parameter or copy constructor.

[ ]:
```
std::list<int> first;
std::list<int> second (4,100);
std::list<int> third (second.begin(), second.end());
std::list<int> fourth(third);
```

3. Using the list container The list object can be used as a normal datatype.

[ ]:
```
//Empty list of int elements
for (int i=0; i<N; i++)
    first.push_back(i);

// List of elements from 0 to 5
std::cout << "First: ";
for (int n : first)
    std::cout << n << " ";
std::cout << std::endl;

// List with 4 elements = 100
std::cout << "Second: ";
for (int n : second)
    std::cout << n << " ";
std::cout << std::endl;

// Iterating from second
std::cout << "Third: ";
for (int n : third)
    std::cout << n << " ";
std::cout << std::endl;

// Copied from third
std::cout << "Fourth: ";
while (!fourth.empty())
{
    std::cout << fourth.front() << " ";
    fourth.pop_front();
}

std::cout << std::endl;
```

1.2 Iterators

An iterator provides a general method of successively accessing each element within any sequential or associative container type. * ++iter: advances the iterator to address the next element of the container. * *iter: returns the value of the element addressed by the iterator.

Each container provides a begin() and an end() member function. * begin(); returns an iterator that addresses the first element of the container * end(); returns an iterator that addresses 1 element past the last element of the container.

1.2.1 Declaration

Iterators are declared by indicating, to which type of container it should refer to:

[ ]:
```
std::vector<int>::iterator it;
```

For accessing the elements of the container, we have to iterate over the elements:

[ ]:
```
std::cout << "Elements of the vector container: ";
for (it=myVector.begin(); it!=myVector.end(); ++it)
    std::cout << *it << " ";
std::cout << std::endl;
```

The iterators concept gives us a completely transparent way of accessing the elements of the container.

Compare the code above with the same used for visiting the elements of the List container from above:

[ ]:
```
std::list<int>::iterator it2;

std::cout << "Elements of the list container: ";
for (it2=first.begin(); it2!=first.end(); ++it2)
    std::cout << *it2 << " ";
std::cout << std::endl;
```

Open questions

1. How would you implement a reverse ordering of a vector/list?
2. How could you just print the elements in odd position of your choosed container?

[ ]:
```

```