TP C++ - 4

Thierry Géraud, Edwin Carlinet

Contents

1 The C++ standard library

The C++ standard library is a collection of functions, constants, classes, objects and templates that extends the C++ language. It can be divided in three kinds of libraries:

- C library
- STL
- Miscellaneous libraries

To avoid name conflicts and provide homogeneous names, every standard function of the C++ standard library belongs to the `std` namespace.

1.0.1 C Library

The C standard library is included in the C++ one. C++ standard headers change their name when they come from the C standard library. They are prefixed with the 'c' character and do not have the `.h` extension. Thus, available headers are:

```
cassert, cctype, cerrno, cfloat, ciso646, climits, clocale, cmath, csetjmp, csignal, cstdarg,
cstdbool, cstddef, cstdint, cstdio, cstdlib, cstring, ctime, cuchar, cwchar, cwctype
```

1.0.2 STL

The concept of *template* is the basis of the STL (Standard Template Library), which has inspired a big part of the C++ standard library.

This part aims at showing you some features of the C++ Standard Library. It is just a glimpse at the global contents of the standard library, which has numerous other features you are encouraged to discover by yourself.

Today, we will mainly focus on containers and iterators, which are foundamental concepts introduced by the STL and included in the C++ standard library.

1.0.3 Miscellaneous Libraries

In the C++ standard library, there are many libraries which are not part of the C library or the STL. They are very useful. Well-known miscellaneous libraries are the Strings library, the Input/Output Stream library, the Exceptions library, etc.

1.1 Documentation

You will quickly notice that using the C++ without its documentation is quite hard. Some of the websites where you can find good documentation are listed below (you can also find documentation about the other libraries of the C++ Standard Library): `http://en.cppreference.com`.

1.2 Containers

A main purpose of the C++ Standard Libraryis to provide generic data structures, which can hold any kind of data: this kind of structure is called a *container*. It helps developers not to reimplement the same classical data structures and then factorizes performance improvements and bug tracking.

Containers are split into two main categories, depending on their functionalities:

- A *sequence* container is a variable-sized container holding elements in a sequential way; thus, they are identified by their position.

  ```
  vector, deque, list, forward_list, bit_vector
  ```

- An *associative* container is a variable-sized container that supports efficient retrieval of elements (values) based on keys. It supports insertion and removal of elements, but differs from a *sequence* in that it does not provide a mechanism for inserting an element at a specific position.

  ```
  set, map, multiset, multimap, hash_set, hash_map, hash_multiset,
  hash_multimap, hash
  ```

To allow uniformization across algorithms working on these structures, containers define the type value_type. This is the type of the elements which must be inserted, but it is generally the same as the one they are templated with.

1.3 Iterators

Each container defines its own iterator, allowing to generically iterate over their content. Iterators are an abstraction of pointers for containers.

The two most important operators for dealing with iterators are:

- `*` accesses the item pointed by the iterator
- `++` jumps to the next item in the container

Since iterators are not the objects themselves (but accessors to them), they are comparable to pointers and are semantically equivalent.

Each container defines two kinds of iterators:

- `iterator`, the type of an iterator on the container
- `const_iterator`, the type of an iterator pointing to a content which cannot be modified through it. When dereferenced, the pointed item has the type `const T`

Each container also defines:

- `begin` which returns an iterator to the first element of the container
- `end` which returns an iterator referring to the *past-the-end* element in the vector container

One can also use `rbegin` and `rend` to get reverse iterators. Thus, iterating on a container is usually done in this way:

```
#include <iostream>
#include <vector>

typedef std::vector<int> Cont;

int main()
{
  Cont v;

  // insert elements in v ...

  for (Cont::iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it << std::endl;
}
```

1.3.1 C++11 range-based `for` loop

C++11 introduced a new `for` loop syntax which allows easy iteration through a container that defines `begin` and `end` methods:

```
std::vector<int> array(4, 100);
for (int x: array)
{
   // do something
}
```

1.4 Comparison operators on containers

Containers can be compared with `==` and `!=`. To be equal, two containers must satisfy the following properties:

- they must have the same size
- their elements must be equal two by two

Comparing two containers with `>` or `<` uses the same operators, if they are defined, on their elements. They operate by comparing elements two by two, returning the result of the comparison between the first two different elements (this is the lexicographical ordering).

1.5 Sequence containers

Sequences are containers which aim to store objects to treat them in a predefined order. Since there is no key to identify the objects they contain, they have no search feature. Generally speaking, sequences only implement insertion and suppression methods, as well as iteration on their elements.

The following examples use the `list` container, but they are common to most other sequences. Obviously performances differ depending on the chosen container.

Construction and initialization Sequences implement several constructors and two overloads of the `assign` method which allow to assign several elements at a time.

```cpp
#include <list>

int main()
{
  // l1 = {}
  std::list<int> l1;

  // l2 = {5, 5, 5}
  std::list<int> l2(3, 5);

  // l3 = l2 => l3 = {5, 5, 5}
  std::list<int> l3(l2.begin(), l2.end());

  // l1 = {2, 2, 2, 2}
  l1.assign(4, 2);

  // l2 = {2, 2, 2, 2}
  l2.assign(l1.begin(), l1.end());
}
```

Adding and removing elements The `insert` method has various flavours.

Below is an example of the different ways of using the `insert` method:

```cpp
#include <list>

int main()
{
  // l1 = {}
  std::list<int> l1;

  // l1 = {5}
```

```
  std::list<int>::iterator it;

  it = l1.insert(l1.begin(), 5);

  // l1 = {5, 3, 3}
  l1.insert(it, 2, 3);

  // l2 = {}
  std::list<int> l2;

  // l2 = {5, 3, 3}
  l2.insert(l2.begin(), l1.begin(), l1.end());
}
```

Similarly, two flavours of the erase method allow to specify in two different ways how elements are suppressed from a sequence:

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

For example, removing all elements from a list can be achieved like this:

```
std::list<int> l;

// ...

// Get an iterator on the first element of the list
std::list<int>::iterator it = l.begin();

while (it != l.end())
{
  // Assigning the result of erase is very important since
  // this method invalidates it and then ++it would be
  // corrupted otherwise
  it = l.erase(it);
}
```

1.5.1 Lists

The list class implements a doubly linked list structure. It is especially designed for algorithms that iterate on a structure in a sequential way.

They implement bidirectional iterators. Then, they cannot be accessed randomly and one can only jump from an element to another; but they allow constant complexity insertion and suppression, without invalidating iterators or references on list elements. When a suppression occurs, only the iterators and references to the suppressed element are invalidated.

In one word, it is designed for efficient insertion and suppression, but not for random access.

Lists allow front and back insertion and suppression, thanks to the following methods:

```
push_front, push_back, pop_front, pop_back
```

There are also specific methods which allow to apply special treatments to lists:

```
remove, remove_if, unique, splice, sort, merge, reverse
```

1.5.2 Vectors

The vector class provides a data structure with semantics near to the classical C/C++ array. Random access is then done with a constant complexity, but insertion and suppression is noticeably slower.

Suppressing an element only invalidates iterators and references to the elements past the suppressed one.

One can access the first (resp. last) item using `front` (resp. `back`). But, unlike lists, vectors only define `push_back` and `pop_back`, since they do not allow quick insertion and suppression at the first position.

The vector class also defines the `at` method which takes an index (`i`) as its only argument and returns a reference (possibly constant) on the vector itself. It might throw an `out_of_range` exception. The `[]` operator is redefined as an alias of the `at` method but does not perform bound checking.

1.5.3 Adding and removing elements

Insertion methods on associative containers do not allow to insert elements anywhere (unlike sequences). They are listed in the documentation.

Refer to `http://en.cppreference.com/w/cpp/container/map/insert` and `http://en.cppreference.com/w/cpp/container/map/erase` for more information.

They also implement the `clear` method which produces the same result as the equivalent method for sequences.

Below is an example of the different ways of using the `insert` and `erase` method:

```
#include <map>
#include <string>

int main()
{
  typedef std::map<int, std::string> int2string;

  // Association integer -> string
  int2string m;

  // m = {<2, "Two">}
  m.insert(int2string::value_type(2, "Two"));

  // m = {<2, "Two">, <3, "Three">}
  // res.first = -> <3, "Three">
  // res.second = true (a new element was inserted)
  std::pair<int2string::iterator, bool> res =
    m.insert(int2string::value_type(3, "Three"));

  // One can also specify the index where the insertion will
  // be done
  m.insert(res.first, int2string::value_type(5, "Cinq"));

  // Suppresses the element which has the key "2"
  // m = {<3, "Three">}
  m.erase(2);

  // Erase the element "Three" using an iterator pointing to it.
  // m = {}
```

```
  m.erase(res.first);
}
```

1.5.4 Search methods

There are many powerful search methods for associative containers. They are all described in the documentation. We strongly encourage you to take the time to read it.

1.6 Containers and references

Please notice that containers of the STL can have object or pointers, but no references. In fact, a reference must be initialized. But, the size of a vector is computed dynamically, so we cant initialize each item of the container.

Example:

```
std::vector<MyClass>  a; // Correct
std::vector<MyClass*> b; // Correct
std::vector<MyClass&> c; // Incorrect, will not compile
```

1.7 Debugging containers

GNUs libc++ offers some neat features to debug containers and iterators. Just use the _GLIBCXX_DEBUG macro when compiling your code to enable all these features.

With this macro enabled, iterators are automatically checked and bound-checking is performed on accesses.

2 Exercices

2.1 Repertory

2.1.1 Contact Class

Implement a class `Contact` that will represent an entry of the repertory that will contain (at least) the firstname, lastname and number phone of the contact.

2.1.2 Main program

Implement a program that will allow a user to enter contacts in a repertory. These contacts will be stored in a map and associated with an id. At the end of the program, the repertory will have to be displayed.

Here is an example of what your program could do:

```
$ ./repertory
New contact:
Enter firstname: Xavier
Enter lastname: Login
Enter number phone: 0613374221
Continue (y,n): n

Repertory:
Entry 1: Xavier Login (0613374221)
```

2.1.3 Unique

Write an algorithm that detects if a vector has all unique elements.

Examples:

```
input {1,2,3} -> return true;

input {1,2,1} -> return false;
```

2.1.4 List of pairs = maps

Write a program which inserts pairs of type ‘¡int, string¿‘ in a `list`. For example, insert `<1, "One">‘, ‘<2, "Two">‘`

Construct a `map`, initializing it with the elements of the list built previously. Then, display the content of the map, i.e. the *key-value* relations.

2.1.5 Palindrome vector

A palindrome is a word, phrase, number or other sequence of characters which reads the same backward or forward.

Write a templated function which determines whether a given container holds a palindrome or not. Use an iterator and a reverse iterator.

Example:

```
# include <iostream >
# include <list >
# include <vector >
int main ()
{
  std::vector <char > v1;
  std::vector <std::string > v2;
  std::vector <std::pair <int , bool >> v3;

  // Gimme gimme a man!
  v1.push_back('a');
  v1.push_back('b');
  v1.push_back('b');
  v1.push_back('a');

  // Another one
  v2.push_back("Win");
  v2.push_back("the");
  v2.push_back("yes");
  v2.push_back("the");
  v2.push_back("Win");

  v3.push_back(std::make_pair (1, true));
  v3.push_back(std::make_pair (2, false));
  v3.push_back(std::make_pair (1, false));

  std::cout << "v1 is" << (palindrome(v1) ? "" : "not") << "a palindrome\n";
  std::cout << "v2 is" << (palindrome(v2) ? "" : "not") << "a palindrome\n";
  std::cout << "v3 is" << (palindrome(v3) ? "" : "not") << "a palindrome\n";
}
```