Cours de C++ 2

STL Containers

Cécile Braunstein

cecile.braunstein@lip6.fr

Introduction

Containers - Why ?

- Help to solve messy problems
- Provide useful function and data structure
- Consistency between containers

Containers

- Collection of objects
- Defined with the template classes : Seperate the container from the type of the data inside.
- Each `containers` type is optimized for a specific use (access/modification).
- Main containers:
  `list, vector, stack, queue, map`

Example : `vector`

Operations

Action	Method
Insert an element	`v.push_back()`
Remove an element	`v.pop_back()`
Remove all elements	`v.clear()`
Returns a value that denotes the first element	`v.begin()`
Returns a value that denotes (one past) the last element	`v.end()`
Returns a value that denotes the i^{th} element	`v[i]`
Take the vector size	`v.size()`
Check emptiness	`v.empty()`

Attention : The first element is indexed by 0 and the last by size-1.

Container's type

`list`

- Insert and remove anywhere in constant time
- Automatic memory management

`vector`

- General purpose
- Fast access by index (constant time)
- Remove an element at the end in constant time
- Other insert and remove in linear time

`set, map`

- Access an element by a key in constant time
- Fast search of an element

Containers constructors

```
#include <list>
using namespace std;
 list <int> one_list ;  //  empty  list  of  double
 list <double> second_list(10,4.22); // ten doubles with walue 4.22
 list <double> third_list(second_list) ;  //  a copy of the second list
```

```
#include <vector>
using namespace std;
vector<string> one_vector; //  empty vector of  string
vector<int> two_vector(4);  //  4  ints  with  undefined value
```

Containers constructors

`container<T> c;`	Empty container
`container<T> c(c2);`	Copy of c2
`container<T> c(n);`	With *n* elements value-initialized according to *T*
`container<T> c(n,t);`	With *n* elements copies of *t*
`container<T> c(b,e);`	Copy of the elements an other container between `[b,e)`

Container's properties

- Containers have their own elements
- Elements of a container have to support the copy and assigment instruction (=)
- All containers have a method `empty()` and `size()` in constant time
- All containers have a method `begin()` and `end()`

Which constructors are called ? 1/2

```
int main()
{
    Point p1(1,3);
    Point p2(2,3);
    Point p3(3,5);
    std::list<Point> liste_de_point;

    liste_de_point.push_back(p1);
    liste_de_point.push_back(p2);
    liste_de_point.push_back(p3);

    return 0;
}
```

Which constructors are called ? 2/2

```
int main()
{
    std::list<Point> liste_de_point;
    std::list<Point> liste_de_point_2(3);
    std::list<Point> liste_de_point_3(liste_de_point);

    liste_de_point.push_back(p1);
    liste_de_point.push_back(p2);
    liste_de_point.push_back(p3);

    liste_de_point_2 = liste_de_point;

    return 0;
}
```

How to choose ?

What is the purpose ?

- How we want ot access the element (randomly, in one order ...)
- Which modification on the collection of data (add/remove elements, sort ...)

Programm performance

- Access time/ Modification time
- Time depends on the number of elements
- Types of times : linear, log, exponential ...
- Memory usage ...

How to access element ?

Iterator Purpose

- Pointer generalization
- Use for a sequential access to elements
- Optimisation regarding the container's type

Iterator Definition

An iterator is a value that

- Identifies a container and an element in the container
- Lets us examine the value stored in that element
- Provides operations for moving between elements in the container
- Restricts the available operations to correspond to what the container can handle efficently

First example

```
vector<double> v;
//v is full
vector<double>::size_type
  i;

for(i = 0; i != v.size(); ++i)
{
   cout << v[i] << endl;
}
```

```
vector<double> v;
//v is full
vector<double>::iterator iter;

for( iter = v.begin();
   iter != v.end(); ++iter)
{
  cout << *iter << endl;
}
```

Iterator

Most general types

Every standard container defines two iterator types :

- `container-type::iterator`
- `container-type::const_iterator`

Operations

- Comparison ($\neq, = \ldots$)
- Incrementation
- Dereference $*$

Every operations that modify the containers invalid the iterator

Others examples - 1/2

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list <int> my_list;
    for (int i=1; i<10; i++)     my_list.push_back(i);

    list <int>:: iterator  it ;
    for( it = my_list.begin(); it != my_list.end(); it++)
    {
        if ( (*it )%2 == 1)
         it = my_list.erase(it );
    }
    for( it = my_list.begin(); it != my_list.end(); it++)
        std::cout << *it << endl;
    return 0;
}
```

Others examples - 2/2

```
vector<Point> my_vector(liste_de_point.begin(),liste_de_point.end());
vector<Point>:: iterator  it ;
for( it = my_vector.begin(); it != my_vector.end(); it++)
{
    cout << it->getX() << endl;
}
```

Type Inference (auto)

from C++03 to C++11

C++03

```cpp
int  a = 2;      // a is an interger
double b = 8.7; // b is a double
int  c = a;      // c is an integer
```

C++11

```cpp
auto a = 2;      // a is an interger
auto b = 8.7;   // b is a double
auto c = a;      // c is an integer
```

The keyword **auto** is very useful for reducing the verbosity of the code

```cpp
for (std::vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
```

```cpp
for (auto it = v.begin(); it != v.end(); ++it)
```

C++11 range-based for loops

Finally C++ has a convenient way to write a for loop over a range of values !

```
vector<int> vec;
vec.push_back( 10 );
vec.push_back( 20 );

for (int i : vec)
{
    cout << i;
}
```

```
map<string, string> address_book;
for (auto address_entry : address_book)
{
    cout << address_entry.first ;
    cout << " < " ;
    cout << address_entry.second ;
    cout << ">" << endl;
}
```

To modify the values in the container or to avoid to copy large objects

```
vector<int> vec;
vec.push_back( 1 );
vec.push_back( 2 );

for (int& i : vec )
{
    i++;
}
```

Associative containers

Goal

What happen if we want to find a given value into a sequential containers ?

- Look at each element one by one
- Sort the container and use a fast search algorithm

Both solutions are quite slow or need sophistical algorithm

Alternative : using associative containers

- Arrange elements that depends on the value of the element
- Exploit the ordering to locate element quickly
- It contains more information : key

Associative Array

`map` example

`< key , value >`

When we put this pair into the data structure, the key will be associate to this value until we delete the pair.

Works as `vector`

But

- Key doesn't need to be an integer; it can be any value that we can compare in order to keep them ordered
- Unique key values
- Associative containers is self-ordering : our program must not change the order of elements

Using associative containers

The class `pair`

- Simple data structure that holds to element : `first` and `second`
- Each element of `map` is a `pair`
- `first` : key ; `second` : the associated value

Iterator

For a `map` with a key of type *K* and a value of type *T* the associated pair is :

```
pair<const K, T>
```

Access the key and the value with an iterator is :

```
map<char,value>::iterator ite;
ite->first;
ite->second;
```

Compare fonction

How to compare keys ?

- When built-in type or type with comparaison function : use the defined ones
- When no comparaison exists: programmer have to write one

`map<K,V> m;`	Empty map with keys of type `const K` and value of type `V`
`map<K,V,T_fpt> m(fpt);`	Map with the comparaison function as pointer function `fpt` with the prototype `T_fpt`
`map<K,V,Comp> m;`	Map with a comparison object to be used for the ordering

Main operations

`map<Key,t>`	declaration
`begin()` `end()`	Return iterator to beginning Return iterator to end
`empty()` `size()`	Test whether container is empty Return container size
`operator[]`	Access element
`insert(pair elt)` `erase`	Insert element Erase elements
`find`	Get iterator to element
`lower_bound` `upper_bound`	Return iterator to lower bound Return iterator to upper bound

Use of an associative container

```
map<string,double> m;
m["Abie"] = 2.5;
m["Sarah"] = 6.8;
m["Michael"]= 7.5;
m.insert(pair<string,double>("Thomas",5.2));
map<string,double>::iterator it;
for(map<string,double>::iterator it = m.begin(); it != m.end(); ++it)
{
    cout << it->first << "\t" << it-> second << endl;
}
```

Use of an associative container-constructor 1/2

```
bool compare(const string& s1, const string& s2)
{
    return s1.size() >= s2.size();
}
int main()
{
    bool(*fn_pt)(const string&,const string&) = compare;

    map<string,double,bool(*)(const string&, const string&)> m(fn_pt);
     ...
}
```

Use of an associative container-constructor 2/2

```
struct classcomp {
  bool operator() (const string& s1, const string& s2) const
  {return s1.size() <= s2.size() ;}
};

int main()
{
    map<string,double,classcomp> m;
     ...
}
```

Use of an associative container-constructor for class type

```
bool compare(const Point& p1, const Point &p2)
{
    return p1.cosinus() <= p2.cosinus();
}
int main()
{
    Point p1(1,3);
    Point p2(2,3);
    Point p3(3,5);

    set<Point,bool(*)(const Point&,const Point&)> set_point(compare);
     ...
}
```

Using associative containers

Example

1. Use a `map` to count the number of word's occurences in a sentence. Print the result (word,number).
2. We have a list of names, we want to decompose this list in as many list as we have different first letters.

Standard library for containers

STL Algorithm

Defines a set of function specially design to be used with a containers of elements.

- Elements must be accessible with iterators or pointers
- Operates on elements
- Never affect the containers structure

Not all function works with all containers types (depends on the operation)

Functions example

`for_each`	Apply a function to range
`find`	find value in a range
`copy`	copy range of elements
`replace`	replace value in a range
`rotate`	rotate elements in a range
`set_union`	Union of two sorted range
`min_elements`	return the smallest element in a range

How it works

Iterators exists for all containers, hence the functions have access to the element through the iterators.

Example

```
template<class InputIterator, class Function>
  Function for_each(InputIterator  first ,  InputIterator  last , Function f)
  {
    for ( ;  first != last ;  ++first  )  f(* first ) ;
    return f ;
  }
```

```
int add_1(int& a){return a++;}
vector<int> my_vector;
// fill the vector
for_each(my_vector.begin(), my_vector.end(), add_1);
```