SFU CMPT-212 2008-1 Topic: Standard Template Library

Ján Maňuch

E-mail: jmanuch@sfu.ca

Monday 10th March, 2008

STL — Standard template library

- *generic programming* — concentrates on algorithms, but uses abstraction (of data types and algorithms) and the creation of reusable code through *templates* and *concepts*
- important components (concepts) of STL are:
  - *container* — is an object that can hold several values of particular type
  - *iterator* — is an object used to move through a particular container, stream, etc.
  - *function object* — is an object that acts like a function
  - *algorithm* — is a generic function accomplishing a particular task (e.g., sorting, finding element)

- *concepts*:
  - are not part of C++ language
  - are sets of requirements
  - implementation of the concept is called a *model*

  Example: the *concept* of the *input iterator* specifies the following requirements: an input iterator object can be
  - dereferenced for reading (`*` operator)
  - can be incremented to the next element in the container (`++` operator)
  - can be tested for equality (`==` and `!=` operators)

  a pointer to an array is an example of a model for the input iterator concept (where the container is an array)

Example of a container: vector

- defined in `<vector>` header file in namespace `std`
- it's a class template with one parameter: the type stored in the container

```
1 vector<int> v;
```

- acts as an array

```
1 vector<int> v(10); // can hold 10 elements
2 v[0]=5;
3 cout << v[9];
```

- can be easily enlarged

```
1 v.push_back(20); // the size is now 11 and
2                  // the last element v[10] is 20
```

- method `size()` holds the number of elements

Example: vector.cpp

- the amount of memory which is `vector` allocating is not always equal to `size()`

```
1 vector<double> d; // empty array
2 for (int i=0; i<30; i++) {
3   d.push_back(1.0);
4   cout <<"capacity: "<<d.capacity()
5   // number of allocated elements
6        <<"  size: "<<d.size()<<endl;
7 }
```

Example: vector1.cpp

- if you know how many elements you are going to add, you can "help" `vector` by giving it this information

```
1 vector<unsigned int> v(10);
2 // planning to add another 10 elements:
3 v.reserve(v.size()+10);
4 cout <<v.size();     // prints 10
5 cout <<v.capacity(); // prints 20
```

- `resize()` method will change the number of elements

```
1 vector<int> v(10);
2 for (int i=0; i<10; i++)
3   v[i]=i;
4 v.resize(20);
5 cout <<v.size(); // prints 20
6 v.resize(5);
7 cout <<v.size(); // prints 5
8 // note: values of v[5]..v[19] are lost
```

- copy constructor and assignment operator are properly overloaded:

```
1 vector<int> w=v; // copy the values
2 cout <<w.size(); // prints 5
3 cout <<w[2];     // prints 2
```

- removing last element:

```
1 w.pop_back(); // decrease size to 4
```

Note: `push_back()` and `pop_back()` are efficient operations for `vector`

Example of an iterator

- each container has its own iterator, “kind of pointer” which can point at elements of the container

```
1 vector<int>::iterator i;
2 // declaring an iterator of container vector<int>
3 vector<int> v(10);
4 i=v.begin(); // i points at the first element stored in v
5 i=v.end();   // i points at past-the-end location of v
```

- easy to make loops accessing elements of a container:

```
1 for (i=v.begin(); i!=v.end(); i++)
2   cout <<*i;
3 // cout <<v[i]; // a compiler error! i is not an integer
```

advantage: the same code will work for any container, independently on implementation of the container (dynamic array, linked list, binary tree, etc)

- iterator of a vector is a *random access iterator*: we can do with it the same things as with pointers (dereference, increment, decrement, compare, but also add or subtract random integer)

```
1 i=v.begin()+2; // i points the third element of v
2 i=v.end()-1;   // i points at the last element of v
```

- inserting elements to a container

```
1 vector<int> v(5,12); // 5 elements each initialized to 12
2 v.insert(v.begin()+2,3,13);
3 // insert at the third position 3 copies of 13
4 // v: [12, 12, 13, 13, 13, 12, 12, 12]
5 v.insert(v.begin(),11); // insert 11 in front
6 // v: [11, 12, 12, 13, 13, 13, 12, 12, 12]
```

- inserting elements from one container to another

```
1 vector<int> v(5,12); // 5 elements each initialized to 12
2 vector<int> w(2);    // 2 elements
3 w[0]=2; w[1]=4;
4 v.insert(v.end(),w.begin(),w.end());
5 // append content of w to the end of v
6 // v: [12, 12, 12, 12, 12, 2, 4]
```

- pointers are also a model of iterators for ordinary arrays

```
1 int array[]={2,4,6,8};
2 vector<int> v; // note v.begin()=v.end()
3 v.insert(v.end(),array,array+4);
4 // v: [2, 4, 6, 8]
5 v.insert(v.begin()+3,array+1,array+4);
6 // v: [2, 4, 6, 4, 6, 8, 8]
```

- erasing elements:

```
1 v.erase(v.begin()+1); // erase second element
2 // v: [2, 6, 4, 6, 8, 8]
3 v.erase(v.end()-2,v.end()); // erase last two elements
4 // v: [2, 6, 4, 6]
5 v.clear(); // erase all
6 // v: []
```

- **Note:** `insert()` and `erase()` are not as efficient methods as `push_back()` and `pop_back()` (linear time is required to perform these operations)

Examples of algorithms

Example 1: `sort(iterator,iterator)`

- define in header file `<algorithm>`
- works with any *random access iterators* (pointing at the first and pass-the-end elements of a container which should be sorted)
- in addition, it's required that the elements stored in container can be compared with `operator<()`

```
1  int a[]={5,4,3,2,1};
2  sort(a,a+5);
3  // a: [1,2,3,4,5]
4
5  vector<int> v(a+2,a+5); // use iterators to initialize vector
6  // v: [3,4,5]
7  v.insert(v.end(),a,a+2);
8  // v: [3,4,5,1,2]
9  sort(v.begin()+1,v.end()-1);
10 // v: [3,1,4,5,2]
```

Example 2: `for_each(iterator,iterator,function)`

- works with any *input iterators* (the values can be read) and any unary function object which can be applied on the type stored in container

```
1  void print(int i) // unary function on int
2  {
3    cout <<"The value: "<<i<<endl;
4  }
5  ...
6  vector<int> v(10); // size 10
7  for (int i=0; i<10; i++)
8    v[i]=10-i;
9
10 for_each(v.begin(),v.end(),print);
11 // applies print() function on every element of v
```

How is an algorithm implemented?

- using function templates
- iterators are either pointers or objects of some class (defined in the container), i.e. they are variables of some type
- function objects are either pointers to ordinary functions (recall if you use the name of function without parenthesis, you get the address of the function) or object of a class with overloads `operator()`, i.e. they are variables of some type
- when you use function templates, compiler will automatically guess the types; you don't need to know the type, only the requirements what those types should be able to do (which methods can be applied on them), i.e., to know their concepts

Example: `for_each(iterator,iterator,function)`

- the header of template:

```
1 template <typename Iterator,typename Function>
2 void foreach(Iterator b,Iterator e,Function f)
```

Specifications:

  - `Iterator` is an *input iterator* (dereferencing for reading, `++` and `==` operators) pointing at elements of some type `T`
  - `Function` is a *function object* which takes one parameter of type `T`

- the body of template:

```
3 {
4   Iterator i;
5   for (i=b; i!=e; i++)
6     f(*i);
7 }
```

Example: foreach.cpp

Examples of function objects

- ordinary functions (e.g., `print()`) — difficult to parametrize
- for instance, assume you want to print different strings before and after the integer value; for each combination you would need a new version of `print()`
- let's have objects of one class which behave like functions:

```
1  class Print {
2    const char* prefix;
3    const char* suffix;
4   public:
5    Print(const char *pre="",const char* suf="\n")
6      : prefix(pre), suffix(suf) {}
7    void operator()(int i) {
8      cout <<prefix<<i<<suffix;
9    }
10 };
```

Example: foreachprint.cpp

Supplemental material

Useful links on STL (references):

- `http://www.sgi.com/tech/stl/table_of_contents.html`
- `http://www.cppreference.com/index.html`

Sample questions

- List four most important concepts of STL. For each give a short description.
- What is a concept and what is a model in STL?
- Create a vector of double and fill it with 50 values 2.5 followed by 100 values 3.5. Do not use loops. Your code should be short.
- What is advantage of iterators over pointers?
- Write a template for generic algorithm `sum` which takes two iterators `a` and `b` and returns the sum of the elements (using `operator+`) in the interval `[a,b)` (include `*a`, exclude `*b`). As a specification for your algorithm assume that the elements stored in the container (to which `a` and `b` points) can be summed using `operator+`.

  Hint: Each iterator class has a member type `value_type` which is the type stored in the container with which the iterator class is

associated. *Solution:* sum.cpp

- Write a code for a class which can be used for function objects computing a liner transformation on a `double`. That is objects of the class can be used in place of a function taking a `double` and returning a `double`, where the return value is `a*x+b`, where `x` is the argument to function object and `a` and `b` are `double` constants fixed when creating the object. *Solution:* linear.cpp