Introduction to STL (Standard Template Library)

Rajanikanth Jammalamadaka

<rajani@ece.arizona.edu>

A template is defined as "something that establishes or serves as a pattern" Websters

In C++, a template has more or less the same meaning. A template is like a skeleton code which becomes "*alive*" when it is instantiated with a type.

An algorithm is a *well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time* [Schneider].

A class holding a collection of elements of some type is commonly called a *container class*, or simply a *container* [Stroustrup].

A class is a user defined type which is very similar to the pre-defined types like `int`, `char`, etc. So, the standard template library (STL) is a collection of generic algorithms and containers which are orthogonal to each other. By the word orthogonal, we mean that any algorithm can be used on any container and vice-versa.

In C++, there are various generic classes like `vector`, `string`, etc.

Since STL is a very large topic to be covered in an article or two, we will focus on the most commonly used generic classes: `vector` and `string`. Before we discuss the standard containers let us take a simple example to understand the word *template*.

```
// template.cpp

#include<iostream>
using std::cout;

template<typename T>
         // Declares T as a name of some type
/* It is also common to see template<class T>.
   The two mean the same */

/* The following template defines a function
   which takes two constant references of type
   T and returns the maximum value of type T.
   */

T Max(const T& a, const T& b) {
  return (a > b)? a : b;
}
```

```
int main() {
  cout << Max('i', 'r') << "\n";
  cout << Max(1, 3) << "\n";
}
// Output of template.cpp
r
3
```

In the above example, a function `Max` is defined which takes two constant references of type `T` and returns the maximum value. But in the `main` function there are two function calls, one is `Max('i','r')` and the other one is `Max(1,3)`. We did not get any compilation errors because we have used the template mechanism in this function. At run-time, the compiler resolves what types are being passed to the `Max` function and hence knows which output to return. It should be noted that the final compiled binary will be larger as the compiler has to create a function for every type in the code passed to it.

Now, let us start with the example of the `vector` container.

A `vector` is very similar to an array but has more advanced features, some of which are utilized in the example below.

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

using namespace std;

int main() {
  /* I'll now create a vector of integers. It
     is instantiated here, so vint can hold
     integers */
  vector<int> vectorint;
  vectorint.reserve(5);

  /* reserve pre-allocates memory for holding
     five integers*/

  for(int j = 1; j < 6; j++)
    vectorint.push_back(j);

  typedef vector<int> Vector_Of_Ints;
  for(Vector_Of_Ints::const_iterator i
                           = vint.begin();
      i != vint.end(); ++i)
    cout << *i << "\t";
    cout << endl;
}
```

It should be noted that the `reserve` function allocates memory but the allocation is more akin to that of an array than using new. The capacity of a vector is defined as the *minimum number of objects that it can hold*. The reserve method makes sure that the vector has a capacity *greater than or equal to its argument* [Sutter]. In the above example, after this statement

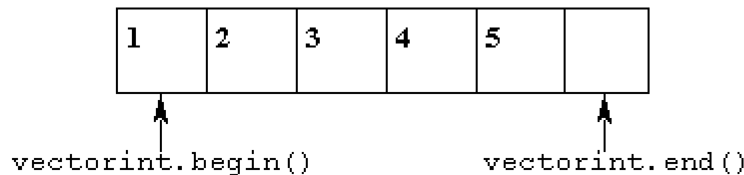
```
vectorint.reserve(5);
```

the `vector` `vectorint` has a capacity of at least 5.

The `push_back()` method function pushes the five integers into the `vector`. An iterator behaves in the same way as a pointer but is more generic. Note that the iterator is made a constant in this example because an iterator seldom modifies the contents of its `vector`.

In the above example, `vectorint.begin()` points to the first element of the `vector`, which is 1. `vectorint.end()` points to an element which is after the last element of the `vector`, which is 5 in this case. Therefore, we cannot dereference `vectorint.end()`. This is as shown below:

When we dereference the iterator using the `*` operator, we get the value stored at the address to which the iterator points to.



When the above code is executed, gives the following output:

```
1    2    3    4    5
```

An even more convenient way of printing a `vector` would be to use the copy algorithm as shown below:

```
copy(vint.begin(), vint.end(),
     ostream_iterator<int>(cout, "    "));
```

which basically means: *copy the contents of the* `vector` *starting from* `vint.begin()` *to* `vint.end() -1` (*remember that* `vint.end()` *points to an element* ***after*** *the last element of the* `vector`) *to the standard output* (`cout`) *separating each of the numbers with four space characters.*

The header `<algorithm>` must be included for the copy algorithm to work.

Of course, the `vector` can be of any type, `int`, `char` or even another class such as `string`, and they are all handled in the same way. This is possible because a `vector` is a generic container, or in other words, a `vector` is a template class. For example after the statement

```
vector<T> foo;
```

`foo` can hold objects of type `T`. Therefore, `T` can even be any class.

Consider a normal class `foo`. It will have a standard constructor, destructor, some methods, and a couple of variables.

```
class foo {
public:
  foo();
  foo(int a, int b);
  foo(int a, double b);
  ~foo() {};
  int showValue() {
    return something;
  }
private:
  int something;
};
```

A template class can be thought of as being the same, except that now when we instantiate it, we do so with an unspecified type `T` rather than an explicit `int`, `char`, etc. Everything else remains the same (more or less).

```
template <typename A, typename B>
class foo {
public:
  foo();
  foo(A a, A b);
  foo(A a, B b);
  ~foo() {};
  A showValue() {
    return something;
  }
private:
  A something;
};
```

Next let us consider the `string` class. A `string` container is very similar to that of a `vector` class. A `string` object can be declared as follows:

```
std::string sobject;
```

A `string` object can be initialized in some of the following ways (there are plenty of other ways)

1. `string sobject("hello");`
2. `string sobject = "hello";`
3. `string s1 = "Hello, ";`
4. `string sobject = s1;`
5. `string s2 = "World";`
6. `string sobject =s1 + s2;`

In the first form of initialization the constructor of the `string` class is invoked with the value of "`hello`" and therefore the `sobject` has the value of "`hello`" once this line is executed.

In the second, third and fifth forms of initialisation, a `string` is assigned to the `string` object and so the `string` objects will hold the respective strings after these statements are executed.

In the fourth and fifth form of initialization, the copy constructor of the `string` class is invoked in order to copy the contents of the strings at the right hand side to the `string` objects.

Let us take another example which uses both the `string` and `vector` class to understand how both of them work together.

```
// vecstringSimple.cpp

#include<iostream>
#include<string>
#include<vector>
#include<iterator>

using namespace std;

int main() {
  vector<string> vector_of_strings;
  vector_of_strings.reserve(5);
  string text;
  cout << "\n";
  cout << "Enter the strings\n";
  cout << "\n";
  for (int a = 0; a < 5; ++a) {
    cin >> text;
    vector_of_strings.push_back(text);
  }

  cout << "\n";
  cout << "Output of the program\n";
  cout << "\n";
  for(vector<string>::iterator i
          = vector_of_strings.end()-1;
      i >= vector_of_strings.begin();
      i--)
    cout << *i << "\n";
}
```

In the above example a `vector` of `string`s is created and `string`s are read into the `vector` until the input is end. The `string`s are then output in the reverse order in which they were entered, as shown below

```
Enter the strings:
```

```
Rajanikanth
Ravikanth
Srimannarayana
VijayaLakshmi
hello

Output of the program:

hello
VijayaLakshmi
Srimannarayana
Ravikanth
Rajanikanth
```

Let us take a more complex example which uses both the `string` and `vector` classes.

```
// vecstr.cpp

#include<iostream>
#include<vector>
#include<algorithm>
#include<iterator>
#include<string>

using namespace std;

int main() {
  vector<string> vector_of_strings;
  string s;

  cout << "Enter the strings to be sorted: "
       << "\n";

  while(getline(cin,s) && s != "end")
    vector_of_strings.push_back(s);

  sort(vector_of_strings.begin(),
       vector_of_strings.end());
  vector<string>::const_iterator pos
         = unique(vector_of_strings.begin(),
                  vector_of_strings.end());
  vs.erase(pos, vector_of_strings.end());
  copy(vector_of_strings.begin(),
       vector_of_strings.end(),
       ostream_iterator<string>(cout,
                                "\t"));
  cout << '\n';
}
```

Let us first understand how this code works and then we will look at how it runs.

First of all, the line

```
vector<string> vector_of_strings;
```

declares vector_of_strings as a vector of strings.

Next, the condition

```
while(getline(cin,s) && s != "end")
```

means read a line of input from stdin as a string until the input is end. So, "end" cannot be an input string for this program. Next, the program pushes each of these strings into the vector vector_of_strings.

The algorithm sorts the strings in the vector vector_of_strings in ascending order. The unique algorithm moves all but the first string for each set of consecutive strings to the end of the unique set of strings in the vector container. It returns an iterator which points to the end of the unique set of strings; in our code this iterator is stored in pos. Next, we call the erase iterator which actually deletes the duplicate elements from pos to vector_of_strings.end().

The copy algorithm then copies the unique set of strings to the standard output.

Following is the output of the program vecstr.cpp

```
Enter the strings to be sorted:
Srimannarayana Jammalamadaka
VijayaLakshmi Jammalamadaka
Rajanikanth Jammalamadaka
Ravikanth Jammalamadaka
aaaaa
a
a
k
k
end

Rajanikanth Jammalamadak
Ravikanth Jammalamadaka
Srimannarayana Jammalamadaka
VijayaLakshmi Jammalamadaka
a
aaaaa
k
```

We will discuss a more complicated program using the vector and string containers in the next article.

References

[Schneider] Schneider, M. and J. Gersting (1995), *An Invitation to Computer Science*, West Publishing Company, New York, NY, p. 9.

[Stroustrup] Stroustrup, B., *The C++ Programming Language (Special Edition)*, Addison Wesley, p. 41.

[Sutter] Sutter, H., *Exceptional C++ Style : 40 New Engineering Puzzles, Programming Problems, and Solutions (C++ in Depth Series)*, Pearson Education; (July, 2004).