Universidad
Carlos III de Madrid

Industrial Informatics I Lab Session

Bachelor in Industrial Electronics and Automation

2016-2017

Lab 6

Teaching staff :

Juan Carlos Gonzalez Victores

Mohamed Abderrahim

Lab 6 – Templates

Templates are one of the most powerful and sophisticated features of C++. Templates allow creating **generic functions and classes** that can be applied to different data types without having to modify the code for each type.

1. Generic functions

A generic function is a set of operations that can be applied to different data types. The data type, on which the function needs to operate, is passed as a parameter. In essence, when you create a generic function, you are automatically creating an overloaded function.

The keyword `template` *must be utilized for defining a template, and the generic form of defining a template function is:*

```
template <class Type,...> return_type name_function(parameter list) {
      //body of function
}
```

`Type` is a reference of the data type to be used within the function that is passed as function parameters and/or the return type of the function. The compiler substitutes the real data type corresponding to each call, during compilation.

Consider an example in which you create a generic function for swapping values of two variables. Since the process of swapping values of variables is independent of the data type of variable, this example fits very well for demonstrating the use of templates.

```
#include <iostream>

using namespace std;

template <class Type>
void swaping(Type & a, Type & b){
      Type aux = a;
      a = b;
      b = aux;
};

int main(int argc, char *argv[]) {
      int i = 10, j = 20;
      double x = 10.2, y = 43.11;
      char a = 'v', b = 's';

      cout << "Original i, j: " << i << ' ' << j << '\n';
      cout << "Original x, y: " << x << ' ' << y << '\n';
      cout << "Original a, b: " << a << ' ' << b << '\n';

      swaping<int>(i,j);
      swaping<double>(x,y);
      swaping<char>(a,b);

      cout << "Swaped i, j: " << i << ' ' << j << '\n';
```

```
	cout << "Swaped x, y: " << x << ' ' << y << '\n';
	cout << "Swaped a, b: " << a << ' ' << b << '\n';

	return 0;
}
```

In this example, Type is the generic type that represents the data type of the values to be swapped. The compiler will replace it with int, double and char, as necessary, in the function template instance.

What is the output of this program?

In a template you can use more than one generic data type (and hence the ellipsis [...] in the above generic definition of a function template). Implement a generic function that displays values of two variables of different types, which are passed as parameters. Try it out. The generic function header is:

```
...
template<class type1, class type2> void print(type1 a, type2 b);
...
```

Recalling from the past lab sessions, as is in the case of classes, templates can also be overloaded. Furthermore, a mixture of standard data types and generic data types can be used

Continue with the previous example, and implement a generic function corresponding to:

```
...
template<class type1, class type2> void print(type1 a, type2 b, double c);
...
```

What happens if you execute `print("string", 2.5, 'a');`? Why?

In short, generic functions must perform the same action for all versions, differing only in the data type it performs the actions on. In contrast, function overloading "normal" allows each implementation to behave in a completely different way depending on the number and type of parameters.

2. Generic classes

A generic class defines all the algorithms (Methods) that a class will use, and the data type that will be handled by the class is specified as a parameter when an object of the class is instantiated. A generic class is useful when implementing a logic that can be generalized for different data types.

The generic form of defining a template class is as follows:

```
template <class Type,...> class class_name {
	...
}
```

The `Type` that was mentioned for generic function is also applicable for generic classes.

An object of the generic class type is created in the following way:

```
class_name <type> object;
```

Where, `type` is the data type that the class handles. Following is an example of a generic class that implements a rudimentary data structure stack. This stack can handle any data type. Is there a mistake in the above code? If so, find it.

```
#include <iostream>

using namespace std;

template<class Type>
class Stack {
    public:
        Stack() : _size(10), _index(0) { // another way to initialize variables
            _data = new Type[_size];
        }
        void push(Type data);
        Type pop();

    private:
        int _size, _index; // stack size
        Type *_data;
};

template<class Type>
void Stack<Type>::push(Type data) {
      if(_index == _size) {
            cout << "Full stack" << endl;
      }
      else {
            this->_index++;
            this->_data[this->_index] = data;
      }
}

template<class Type>
Type Stack<Type>::pop() {
      if(this->_index == 0) {
            cout << "Empty stack" << endl;
            return 0;
      }
      this->_index--;
      return this->_data[this->_index];
}

int main() {
      Stack<char> S;
      S.pop();
      S.push('z');
      S.push("q");
      S.push(2.1);
}
```

In the definition of a template class, you could also use what is commonly thought to be a standard argument. For example, in the above example, if we want the size of the stack to be defined during the instantiation of a template class object, it is defined as follows:

```
...
template<class T, int size> class Stack {
      T data[_size];
}
...
```

So, what in the implementation of the member functions of the generic class should be changed?

Note: Keep in mind that in C++ only usage of "standard arguments" of integer type, pointer or reference are allowed in templates, and variables are not allowed.

3. STL

The Standard Template Library (STL) of C++ is a library that provides classes to implement general purpose data structures that are well known and widely used. They are based on class and function templates. In STL there are three main elements: Containers that store objects, Algorithms that act on the containers and Iterators for traversing the containers in various ways.

Before you go ahead and develop your own data structures and algorithms (as we did with the stacks example) make sure that you cannot use any of the existing and well known templates included in STL. As an example, let us see how to use the Standard Template Class Vector, and its basic operations.

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

using namespace std;

int main() {
      // creating vector with size 10
    unsigned int i;
      vector<char> myvector(10);
      cout << "size = " << myvector.size() << endl;

      // assign elements and showing
      for(i=0;i<10;i++) {
            myvector[i] = 'z' - i;
      }
      for(i=0;i<myvector.size();i++) {
            cout << myvector[i] << ' ';
      }
      cout << endl << endl;

      // add new elements if necessary
      for(i=0;i<10;i++) {
            myvector.push_back('z'-i-10);
      }
      for(i=0;i<myvector.size();i++) {
            cout << myvector[i] << ' ';
```

```
	}
	cout << endl << endl;

	// using an iterator
	vector<char>::iterator it;
	it = myvector.begin();
	while(it != myvector.end()) {
		*it = toupper(*it);
		it++;
	}
	for(i=0;i<myvector.size();i++) {
		cout << myvector[i] << ' ';
	}
	cout << endl <<endl;

	// sorting the vector
	sort(myvector.begin(), myvector.end());
	for(i=0;i<myvector.size();i++) {
		cout << myvector[i] << ' ';
	}
	cout << endl << endl;

	cin.get(); // wait an INTRO

	return 0;
}
```

Remember:

When you are typing the example code, uses the auto-completion option (Ctrl+Space) to see all the available possibilities.

There are several sources from where you can get more information about the STL. For example you can check out any of the recommended books on the subject or on the following links:

- http://www.cplusplus.com/reference/stl/
- http://www.cppreference.com/wiki/stl/start

Exercise

Since templates can be applied to virtually any data type, including user-defined data types, we will apply what we learned today on our Computer Simulator project, which we have been working on since the past few lab sessions.

We will augment the project by creating a list of all strings received by the Processor. When the Processor receives a special command `history`, it must display on the monitor screen, all the strings that has been processed until that point.