

CS 3721: Programming Languages Lab

Lab #12: Parametric Polymorphism and the Standard Template Library

Due date: April 29, 3:30pm. At the beginning of the next recitation.

Goals of this lab:

- Be able to write programs in C++ using classes, templated functions, templated classes, and the Standard Template Library.

1. *Templates.* We can use templates to achieve *parametric polymorphism* in C++. By using templates, we can create special functions whose functionality can be adapted to more than one type. In this way, we avoid repeating the entire code for each type, and these special functions can operate with *generic types*.

The **syntax** for declaring function templates with type parameters is:

```
template <typename iden_1, ..., typename iden_n> function_declaration
```

An example:

```
template <typename T>
T Max(T x, T y)
{
    return (x>y ? x : y);
}
```

Using the above function template, we don't need to repeat the code for different types such as int, float, double, string, and even for user-defined types (as long as the "operator >" is overloaded. We will discuss *overloading operators* later).

You can call this function like calling any other ordinary function:

```
int resultInt, a = 4, b = 9;
resultInt = Max(a, b); // int

double resultDouble, x = 4.3, y = 6.1;
resultDouble = Max(x, y); // double
```

Here's the question: The following is the insertion sort function that operates on integer values. Rewrite a *generic* version of it using *templates* so that the new templated function operates with integers, floats, doubles, and strings (C++ overloads the comparison operator >, so don't worry about it.).

```
void insertion_sort_Array(int array[], int array_length)
{
    int i, j;
    int key;
    for(j = 1; j < array_length; j++)
    {
        key = array[j];
        for(i = j - 1; (i >= 0) && (array[i] > key); i--)
        {
            array[i+1] = array[i];
        }
    }
}
```

```

        array[i+1] = key;    //Insert key into proper position
    }
    return;
}

```

You can test your function using the following statements in the main() function:

```

string mystrings[] = {"wsa","dbf","vhd", "jhd"};
insertion_sort_Array(mystrings, 4); //calling the generic function on strings
for (int i=0; i < 4; i++)
    cout << " " << mystrings[i];

double mydoubles[] = {16.3,2.2,77.1,29.5};
insertion_sort_Array(mydoubles, 4); //calling the generic function on doubles
for (int i=0; i < 4; i++)
    cout << " " << mydoubles[i];

```

2. *Standard Template Library.* The Standard Template Library (STL) is a library included in the C++ Standard Library providing *containers, iterators algorithms, and functors*. Some of the containers are: vector (dynamic array), list (doubly-linked list), deque, set, map. We will cover the container, *vector*.

The STL makes use of *templates*, so they can be used with any built-in type and with any user-defined type that supports some elementary operations such as *copying and assignment*.

The container, *vector*, has many useful member functions. Some of them are:

size: Returns the number of elements in the vector container

operator[]: Accesses an element in the vector.(similar to accessing an element in an array.)

push_back: Adds an element at the end.

Interested students may see C++ references for more information about STL.

An Example:

```

#include <iostream>
#include <vector>
using namespace std;
int main ()
{
    vector<int> myVector;
    for (int i = 5; i < 10; i++)
        myVector.push_back(i); // adds the number, i, at the end
    int length = myVector.size();
    for (int i = 0; i < length; i++)
        myVector[i]++;
    for (int i = 0; i < length; i++)
        cout << myVector[i] << " "; // accessing
    cout << endl;
    return 0;
}

```

The above program creates a vector named *myVector*, adds numbers from 5 to 9 to the vector, increments the elements in the vector, and finally displays the contents of the vector.

Question: Rewrite your generic function that you implemented in Question 1 using vectors

instead of arrays. Since you can access the number of elements in the vector using the member function *size*, you don't need to pass the size of the vector to the function.

The prototype of the function will be:

```
void insertion_sort( vector <T> &array);
```

You can test your function using the following statements in the main() function:

```
\\For doubles:
vector<double> doubleVect;
doubleVect.push_back(16.3);
doubleVect.push_back(2.2);
doubleVect.push_back(77.1);
doubleVect.push_back(29.5);
insertion_sort(doubleVect);
for (int i=0; i < doubleVect.size(); i++)
    cout << " " << doubleVect[i];
cout << endl;
// For strings:
vector<string> stringVect;
stringVect.push_back("wsa");
stringVect.push_back("dbf");
stringVect.push_back("vhd");
stringVect.push_back("jhd");
insertion_sort(stringVect);
for (int i=0; i < stringVect.size(); i++)
    cout << " " << stringVect[i];
cout << endl;
```

3. *Operator Overloading*. We can overload the function of most built-in operators in C++. Consider the standard "+" operator. The addition of two integers is not implemented in the same way as the addition of two double numbers. Suppose that we have a class for complex numbers. However, C++ does not support addition of two complex numbers. Therefore, we should overload "+" operator to achieve this addition.

The following example illustrates overloading the plus (+) operator:

```
#include <iostream>
using namespace std;
class complx
{
    double real, imag;
public:
    complx() {}
    complx( double r, double i) {real = r; imag = i;}
    const complx operator+(const complx& rhs)
    {
        complx result;
        result.real = (this->real + rhs.real);
        result.imag = (this->imag + rhs.imag);
        return result;
    }
};
```

```

    }
};

int main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}

```

Question: You're working for a phone service provider company. You have a file having first names, last names, and phone numbers of some customers in an unordered fashion. The format of the file is:

```

<first name> <last name> <phone number>
<first name> <last name> <phone number>
.
.
.
<first name> <last name> <phone number>

```

Your boss wants you to print the contents of the file sorted according to the last names. Your colleague, Sally, gives you the following *cpp* file. (She prepared the interfaces for you, and wants you to implement the functions according to the comments.):

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

class Record
{
public:
    Record() {firstName = lastName = phoneNo = "";}
    ~Record() {}
    bool operator>(const Record& rhs)    // Overloading operator>
    {
        // Overload this operator so that we can make use of the
        // insertion sort in question 2.

        // return true if !(this->last < other.last ||
        // (this->last == other.last && this->first < other.first)).
    }
    void display()
    {
        // print the contents of this record
    }
}

```

```

    string firstName;
    string lastName;
    string phoneNo;
};

class YellowBook
{
public:
    YellowBook() {}
    ~YellowBook() {}
    void insert(Record &rec)
    {
        //insert the record to the back of the vector, array.
    }
    void sort()
    {
        // make use of the insertion_sort function.
        // You don't need to make it generic.
        // It should work on our vector (Record type).
    }
    void display()
    {
        // display the contents of the vector.
        // You can make use of the display method
        // of the Record class.
    }
private:
    vector<Record> array;
};

int main()
{
    YellowBook yellowBook;
    ifstream input;
    try {
        input.open("yellowBook.dat");
        // Read the values from the file.
        // Stick them into the vector in
        // the yellowBook using the insert
        // method of the YellowBook class.
        input.close();
    }
    catch(ifstream::failure e) {
        cout << "Exception opening/reading file: " << e.what() << endl;
    }
    // printing the values in memory. Testing
    yellowBook.sort(); // Sorts the records.
    yellowBook.display(); // Displays the sorted records.
    return 0;
}

```

```
}
```

Your output should be the following:

```
Leandro Barbosa 4327698
Boris Diaw 4557688
Tim Duncan 2851039
Michael Finley 2984623
Manu Ginobili 2569387
Steve Nash 4527635
Shaquille O'neal 4446356
Fabricio Oberto 2318797
Tony Parker 2657487
Amare Stoudemire 4657687
```

4. *Class Templates.* A class template definition is very similar to a regular class definition, but it should be prefixed by the keyword *template*. The following is a class template for a stack:

```
template <typename T>
class Stack
{
public:
    Stack(int = 10) ;
    ~Stack() { delete [] stackPtr ; }
    int push(const T&);
    int pop(T&) ;
    int isEmpty() const { return top == -1 ; }
    int isFull() const { return top == size - 1 ; }
private:
    int size ; // number of elements on Stack.
    int top ;
    T* stackPtr ;
} ;
```

As you guess, the above class can operate on any type such as integers, floats, doubles, etc.

Question: Rewrite the class `YellowBook` so that it works for the types other than `Record`.

If you want to test your new templated class `YellowBook`, you should change the first line of the main function in question 3 to make it work for the type *Record*:

```
YellowBook yellowBook; ==> YellowBook<Record> yellowBook;
```

PS: Submit your code electronically via email to btas@cs.utsa.edu

Please create different directories for each question. These directories will contain your programs in *cpp* files. Tar the directories under <your name> .tar. Make sure that the programs are compiled and tested. (Use the given code snippets for testing)