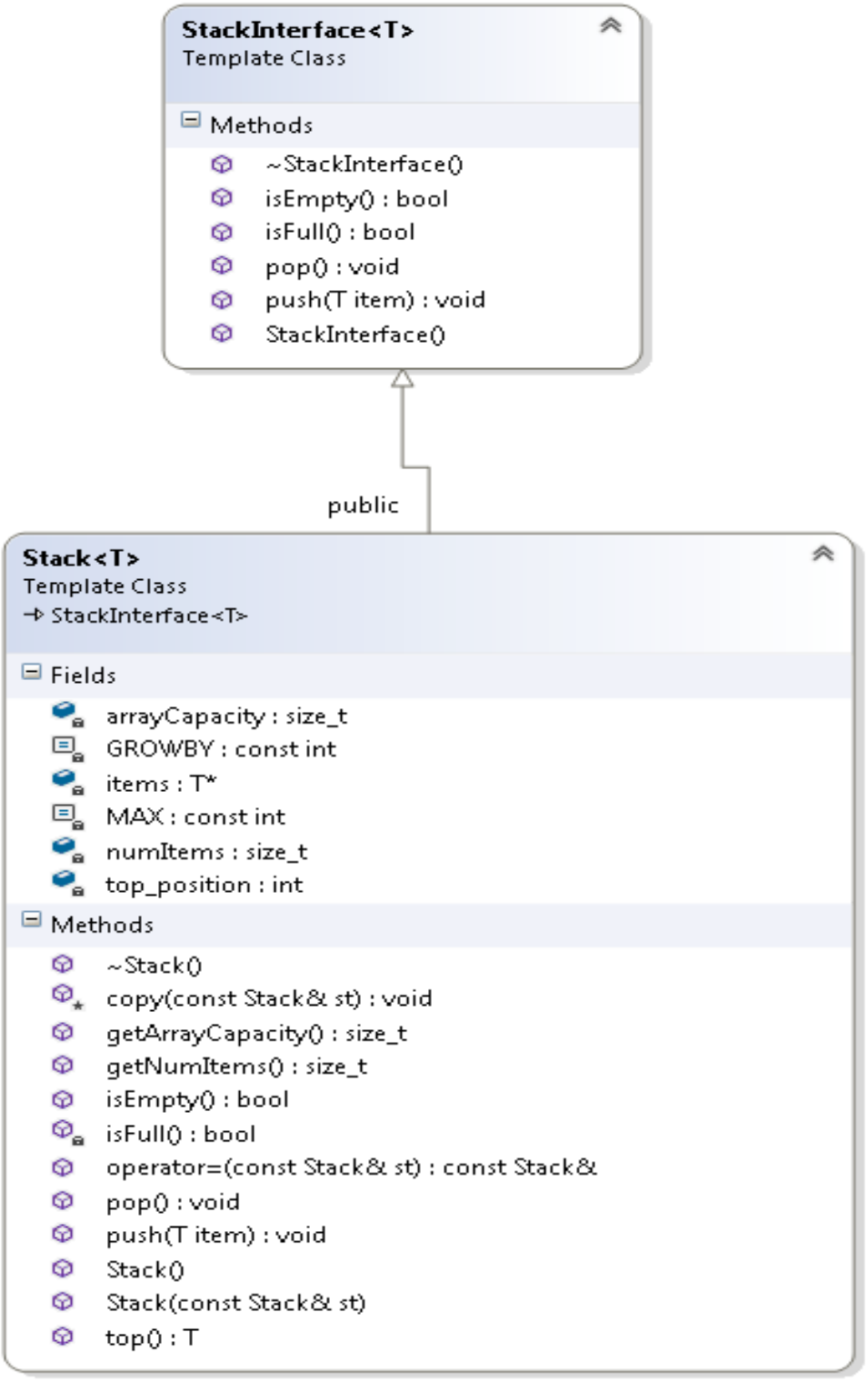
CS52 Lab 10(Self Growing Templated Stack)

Introduction

Templates are very useful in data structures because they allow us the flexibility of creating a data structures of many different data types in the same program. For example if templated stack was not available then one may end up writing as many stack classes as many data types for which stacks are needed. In this program you will write the template version of a self-expanding stack and demonstrate the functioning of this stack by pushing the characters, double values, integers, and strings, and on to the stack of their own type, and then unwind the stack and print the popped objects. Savitch describes stack and queue classes in section 13.2 in chapter 13. You can also read my chapter on stacks to get basic idea of this data structure.

Classes to be Written

The UML diagram on next page shows the design of template StackInterface and Stack classes. StackInterface can be an abstract class. That means that all its member functions, other than constructor would be pure virtual functions. Destructor will be virtual.



Do not use above diagram to copy function signatures. The diagram is only schematic and is not usable for member function headers for writing implementation outside the class. The diagram is not source code.

The template class StackInterface and Stack in this assignment would be written with syntax as below:

```
template<typename  T>
class StackInterface{
//code
};
```

For Stack class, the code for .h and .cpp file is to be placed in one file. That is because when template classes are shipped to client, there is no information hiding. Client needs the implementation code of all class member functions. The Stack class would schematically look like below:

```
template<typename  T>
class Stack{
//code
};
```

As you can see from UML diagram on page 2, the items pointer would now be type T * or a pointer of type T, where T is the template parameter. The functions that get the template parameter T as argument are push and top functions. Latter returns the template parameter T as the value of item on top. It is to be remembered that outside the class, the qualified name of the Stack class is Stack<T>. Thus in implementation part of the assignment operator, for example, you would write the header as below:

```
template<typename  T>
const Stack<T>   Stack<T>::Operator =  (const Stack<T> & Other)
{
 //code
}
```

Both class definition and member functions can be in the same file. However, the class member functions must be outside the class.

Be aware of conforming to template syntax because compiler warning would be too cryptic if syntactic errors are made in writing template d member functions of a class.
Description of various classes and member functions is as follows:

StackInterface	
Member Name	*Member Description*
StackInterface( )	Constructor. Has empty body.
~StackInterface( )	Destructor. Must be declared virtual
bool isEmpty( )	Pure virtual function
bool isFull( )	Pure virtual function
void push (T data)	Pure virtual function
void pop ()	Pure virtual function
T top( )	Pure virtual function. (Not shown in UML diagram)

Stack	
Class data member name	*Description*
T * items	items is a pointer that would store the address of the dynamically created array of type T.
const static int MAX	Initial capacity of the array created and pointed to by pointer items. Choose a suitable number. const static int data should be initialized in the same line.
const static int GROWBY	When stack is filled and new push operation is attempted, the array used by the stack grows by additional capacity GROWBY and then push operation is completed
size_t arrayCapacity	Array capacity at a particular time during program execution. At start arrayCapacity = MAX, but it will increase, each time array grows by amount GROWBY
size_t numItems	number of items added by the user to the stack.
int top_position	Index managing the top of the stack. Starting value could be -1 or zero depending upon the implementation of push function.
Class member functions	***Description***
stack ( )	Constructor creates a dynamic array of type T and stores the return address in items. Size of dynamically created array is MAX. Sets arrayCapacity to MAX. Sets numItems to zero. It sets top_position to either -1 or zero, depending upon the implementation of push function.
void push (T data)	Overrides pure virtual function in base class StackInterface. Below is not an algorithm. You have to come up with the algorithm. Below are only the ideas about what may be going on in this function. 1. Pushes data on to stack if numItems<arrayCapacity. Otherwise does followings: 2. Creates a dynamic array of size arrayCapacity + GROWBY 3. Copies elements from old stack to this new array (with increased capacity). 4. Adds data to stack. 5. Does required numItems and top_position adjustments.
T top ( )	If stack is not empty, then returns ONLY a copy of the item on the top of stack. Otherwise throws an exception using throw statement. You can write your own exception class if you like, or use C++ exception class.
void pop( )	Overrides the pure virtual function in base class StackInterface. Removes top item from the stack. Adjusts top_position and numItems accordingly.
bool isEmpty()	Returns true if stack is empty, else returns false. Overrides the base class pure virtual function.
bool isFull( )	Code this function to return true, because of its expandable nature, stack will not be full. Overrides the base class pure virtual function.
size_t getArrayCapacity()	returns the arrayCapacity data member.
size_t getNumItems()	returns numItems data member
Rule of three enforcement member functions	
copy function	Makes a deep copy of Stack passed to it into this stack.
operator ==	makes a deep copy of stack passed to it into the left-hand operand. Should allow cascaded assignment. Makes use of copy function
Destructor	Deallocates the memory assigned by the constructor
Copy constructor	Makes a deep copy of the argument passed to it into this object. Makes use of copy function.

Testing Your Templated Stack Class

In main function or using the helper functions create objects of type string, int, char, and double and push them on the stack of their respective type. Then you top and pop each stack and print popped objects to the console. Confirm that your template is working correctly. At least five data of each type must be added to the stack and then topped and popped and contents of topped items displayed on console output.