CSCI 152, Performance and Data Structures, Exercise 1

Rules for Exercises:

- You are expected to read the complete exercise, and to make every part of it. 'I did not see this part', will never be considered valid excuse for not making some part of the exercise.
- Code must be in proper lay out. Graders are allowed to ignore code that is not properly indented. Therefore, badly formatted code may result in loss of points.
- You are allowed to cooperate with other students during labs, and to ask for advice for lab assistants. It however is not allowed to share code, or to hand in identical solutions. See the syllabus for more rules. You may be invited for life grading, for discussion of the quality of your code, or its authorship.

In this exercise you will implement a stack class that uses consecutive memory, and that allocates its memory by itself. It was explained in the lecture, that classes must handle their memory management in such a way that it is not visible from outside. Main goal of this exercise is to practice this. Second goal is to understand stack as *data structure*, and see an example of its use.

During lecture, you have seen two other implementations of stack, an implementation that relies on `std::vector`, and an implementation using *linked lists*.

Download the files **stack.h**, **stack.cpp**, **main.cpp**. If you use Linux[1], also download **Makefile**.

The implementation of `stack` is based on the `class` below. In order to avoid frequent reallocations, `stack`, allocates more memory than it needs. This is handled by the method `ensure_capacity`. `data` is a pointer to the allocated memory. The memory segment between `0` and `current_size` is the current contents of the stack. The segment between `current_size` and `current_capacity` is not in use, but reserved for later use.

```
class stack
{
```

[1] We hope you are!

```
   size_t current_size;
   size_t current_capacity;

   double* data;
      // INVARIANT: data has been allocated with size current_capacity.
      // 0 <= current_size <= current_capacity.

   void ensure_capacity( size_t c );
      // Ensure that stack has capacity of at least c.
      // This function is given, so you don't have to write it.
};
```

Each of the functions below can be implemented in file **stack.h** or **stack.cpp**. We suggest that you implement short functions (shorter than 5 lines) in file **stack.h**, longer functions in file **stack.cpp**.

1. Implement the default constructor `stack( )`.
2. Implement the copy constructor `stack( const stack& s )`.
3. Implement the destructor `~stack( )`.
4. Implement assignment `const stack& operator = ( const stack& s )`. Use the function `ensure_capacity`.
5. Implement a constructor

   ```
   stack( std::initializer_list<double> d );
   ```

   This constructor makes it possible to write initializers of form `stack st = { 1,2,3,4 };`
6. Implement `void push( double d )`. Use `ensure_capacity`.
7. Implement `void pop( )`. As explaned in **advice.pdf**, it is totally fine to implement `pop` in such a way that it cheerfully crashes when the stack is empty. If this make you feel bad, you can write `throw std::runtime_error( "pop: stack is empt`
8. Implement `void clear( )`.
9. Implement `void reset( size_t s );`, which shrinks the stack to size `s`. You may assume as precondition that `s <= size( )`.
10. Implement `double peek( ) const`, which allows you to see the top element of the stack. As with `pop`, you may assume that stack is non-empty, or throw an exception.
11. Implement `size_t size( ) const`, which returns the current size of the stack.
12. Implement `bool empty( ) const` which returns **true** if the stack is empty.

13. Write

```
std::ostream& operator << ( std::ostream& , const stack& s );
```

14. Write a function `teststack( )` that does some serious testing. It should contain initializers, copy constructors and assignment. Create a stack and push enough elements, so that there will be a few reallocations. Test it with `valgrind`, or another memory checking tool. For example:

```
void teststack( )
{
   stack s1 = { 1, 2, 3, 4, 5 };
   stack s2 = s1; // Copy constructor.

   for( unsigned int j = 0; j < 30; ++ j )
      s2. push( j * j );

   s1 = s2; // Assignment.
   s1 = s1; // Self assignment.

   s1 = { 100,101,102,103 };
      // Works because the compiler inserts constructor and
      // calls assignment with the result.

   std::cout << s1 << "\n";
}
```

15. If your stack is correct, then you can evaluate expressions in Reverse Polish Notation, for example `15 7 1 1 + - / 3 * 2 1 1 + + -` (Outcome is 5), or `2 sqrt 2 sqrt *`. Outcome is 2.