Course C^{++}

Exercise List 3

Deadline: 14.03.2013

Topic of this task are the *life cycle* methods.

1. Define (in a file `stack.h`) a class

```
class stack
{
   unsigned int current_size;
   unsigned int current_capacity;
   double* tab;
      // class invariant is that tab is always
      // allocated with a block with current_capacity.

   void ensure_capacity( unsigned int c );
      // Ensure that stack has capacity of at least c.

public:
   stack( );               // Constructs empty stack.

   stack( const stack& s );  // These are the 3 life cycle methods:
   ~stack( );
   void operator = ( const stack& s );

   void push( double d );  // Use ensure_capacity, so that
                           // pushing is always possible, as
                           // long as memory is not full.

   reset( unsigned int s ); // Resets the stack to length of
                            // s < size( ).

   double operator [ ] ( unsigned int i ) const;
   double& operator [ ] ( unsigned int i );
      // Be careful, s[0] is equal to top of stack.
      // s[ s. size( ) - 1 ] is the deepest element.

   double top( ) const;
```

```
   double& top( );

   void pop( );
      // Remove one element from the stack. It's OK to write
      // code that crashes, as long as you write clearly what are
      // your preconditions, so:
      // PRECONDITION: The stack is not empty.

   unsigned int size( ) const { return current_size; }
   bool nonempty( ) const { return current_size; }

};
```

This is the definition of `ensure_capacity()`. Write the other methods by yourself (in a file `stack.cpp`)

```
stack::ensure_capacity( unsigned int c )
{
   if( current_capacity < c )
   {
      // New capacity will be the greater of c and
      // 2 * current_capacity.

      if( c < 2 * current_capacity )
         c = 2 * current_capacity;

      double* newtab = new double[ c ];
      for( unsigned int i = 0; i < c; ++ i )
         newtab[i] = tab[i];

      current_capacity = c;
      delete[] tab;
      tab = newtab;
   }
}
```

2. If you wrote the copy constructor, the assignment operator, and the destructor correctly, then your class has *object semantics*. This means that your class is as easy to handle as any primitive type, that it can be put in a standard container, that it can be passed as parameter, and returned by a function without restriction. Always make sure that your classes have object semantics, unless there is a very good reason not to do so. Lazyness is not a good reason.

It is time to check that your implementation of stack has no memory leaks. The easiest way to test this, is by implementing the following program:

```
for( unsigned int i = 0; i < 1000000; ++ i )
{
   stack s1;
   s1. push_back(45); s1. push_back(45); s1. push_back(46);

   stack s2 = s1;
   stack s2. push_back( 2000 ); s2. push_back(100);

   s1 = s2;
}
```

Use the `top` command in Linux, to ensure that the memory use of your program is not increasing.

3. Write

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

Make it a friend of `class stack`, or use `size( )` and `operator[]`.

4. Write some tests, that show that you understand the difference between `operator[ ] ( unsigned int )` and `operator[ ] ( unsigned int ) const` .