CSCI 152, Performance and Data Structures, Exercise 2

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 a valid excuse for not making some part of the exercise.
- Submitted code will be checked for correctness, layout and style. We apply high standards, that make you fit for the real world.
- You are allowed to cooperate with other students during lab sessions 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 **queue** class that uses continuous memory, and which allocates its memory by itself. The implementation will be similar to the **stack class** of Exercise 1.

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

The implementation of **queue** is based on the class given below. As with **stack**, class **queue** has a method `ensure_capacity` that allocates its memory.

When an element is pushed on the queue, it is inserted at position `data + end`. When an element is popped from the queue, it is taken from position `data + begin`. The indices `begin,end` are always increased in circular fashion, and never decreased. When they reach `current_capacity`, they continue at `0`. Don't do this by yourself. Use `next(i)`.

If `begin == end`, there are two possibilities: Either the **queue** is empty, or it is full (`current_size == current_capacity`). We never allow `current_capacity` to be 0, because it has some complications that we don't want.

```
struct queue
{
   size_t current_size;
   size_t current_capacity;
```

[1] We still hope you are!

```
   size_t begin;
   size_t end;

   double* data;
      // INVARIANT: I wrote a function checkinvariant( )
      // which checks the invariant as much as possible. Use this
      // function in your tests.

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

   // Return i+1, but 0 if we reach current_capacity.

   size_t next( size_t i ) const
   {
      i = i + 1;
      if( i == current_capacity )
         i = 0;
      return i;
   }

   friend std::ostream& operator << ( std::ostream& , const queue& );
}
```

1. Implement the default constructor `queue( )`. Implement the copy constructor `queue( const queue& q )`, and the destructor `~queue( )`.

   Constructurs should always allocate something.

2. Implement assignment `const queue& operator = ( const queue& q )`. Use `ensure_capacity`. Don't start messing with the memory by yourself. It will give you a headache.

3. Implement `void push( double d )` and `void pop( )`.

   In `push( )`, you must use `ensure_capacity`.

   As explained in **advice.pdf**, it is totally fine to implement `pop` in such a way that it cheerfully crashes when the queue is empty. If this make you feel bad, you can write `throw std::runtime_error( "pop: queue is empty" );`

   Note that `pop( )` removes the element that is first in the queue.

4. Implement a constructor

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

   This constructor makes it possible to write initializers of form `queue q = { 1,2,3,4 };` Just call the default constructor, and use `push( )`.

5. Implement `void clear( )`.

6. Implement the function `double peek( ) const`, which allows you to see the element that is currently first in the queue. As with `pop( )`, you may assume that queue is non-empty, or throw an exception.

7. Implement `size_t size( ) const`, which returns the current size of the queue. Implement `bool empty( ) const` which returns **true** if the queue is empty.

8. Write a function `testqueue( )` that does some serious testing. It should contain initializers, copy constructors and assignment. Create a queue and push enough elements, so that there will be a few reallocations. Mix them with sufficiently many pops. Call `checkinvariant( )` after each operation in your tests. Test with `valgrind`, or another memory checking tool. This is just a small example:

```
void teststqueue( )
{
   queue q1 = { 1, 2, 3, 4, 5 };
   queue q2 = q1; // Copy constructor.

   for( unsigned int j = 0; j < 30; ++ j )
   {
      q1. push( j * j );
      std::cout << q1. peek( ) << "\n";
      q1. pop( );
   }
   q1 = q2; // Assignment.
   q1 = q1; // Self assignment.

   q1 = { 100,101,102,103 };
   std::cout << q1 << "\n";

   queue q3 = { 1,2,3,4,5,6 };
   std::cout << q3 << "\n";

   for( unsigned int i = 0; i < 10000; ++ i )
   {
      q3. push(i); q3. checkinvariant( );
      double d = q3. peek( ); q3. checkinvariant( );
      q3. pop( ); q3. checkinvariant( );
      q3. push(d); q3. checkinvariant( );
   }

   // Try to show in your solution that you enjoy programming!
}
```