EPL Spring 2016, Project 1
Vector Container Basics
Development Phase 0 (will not be graded!)

Implement the container vector. A vector is a data structure that provides O(1) random access (via an operator[]), and amortized constant append (via push_back). Your vector will differ from the standard library vector in three minor ways. You will not be required to implement all of the C++ standard functions, your operator[] will perform bounds checking, and you will have to implement a "push_front" and "pop_front".

The purpose of this project is to get you programming in C++, to get you thinking about designing a template class, make sure you're up-to-speed on copy constructors, assignment operators, memory allocation, member functions, operators and general C++ programming. This phase of project 1 will NOT be graded.

Requirements:

- Name your template class epl::vector. Create a namespace called "epl" and include the vector template as a member of this namespace. The template should (for now) have one argument – the element type for the vector. In this document we'll use the name "T" for this template argument – i.e., epl::vector<T>
- Implement all of the functions defined below. When there are both const and non-const versions of a function, be sure to define both.

- You should provide a copy constructor and a (copy) assignment operator. You'll also need a destructor.
- The pop_back method should not reallocate space. Just note the extra capacity.

Method List

Constructors:

- vector(void) – creates an array with some minimum capacity (8 is fine) and length equal to zero.
- explicit vector(uint64_t n) – create an array with capacity and length exactly equal to n. As a special case, if n is equal to zero, you can duplicate the behavior of vector(void).
- copy constructor for vector<T> arguments
- copy assignment operator for vector<T> argument
- destructor
- uint64_t size(void) const – return the number of constructed objects in the vector

- T& operator[](uint64_t k) – if k is out of bounds (equal to or larger than length), then you must throw std::out_of_range("subscript out of range"); or a similar error message (the string is arbitrary). If k is in bounds, then you must return a reference to the element at position k
- const T& operator[](uint64_t k) const – same as above, except for constants.
- void push_back(const T&) – add a new value to the end of the array, using amortized doubling if the array has to be resized. Copy construct the argument.
- void pop_back(void) – if the array is empty, throw a std::out_of_range exception (with any reasonable string error message you want). Otherwise, update the length (and any pointers you need) so that the last element in the vector is no longer considered to be an element of the vector. Do not reallocate storage, even if the vector becomes empty. Obviously, the available capacity increases by one.

You must implement amortized doubling (or amortized "trebling" if you prefer) whenever you increase the allocated space of your vector.

- If a call to pop_back succeeds (does not throw an exception) and is followed immediately by a call to push_back, then you must not perform any memory allocations. More generally, if two or more consecutive calls to pop_back succeed and are followed by an equal number of calls to push_back, then no allocations are performed.
- For any vector with K elements ($K \geq 0$), regardless of capacity and regardless of any prior sequence of push_back, pop_back, push_front and pop_front invocations, if N push_back operations are invoked consecutively (with no intervening push or pop operations), then the time complexity to complete all N push_back operations must be O(N) as N approaches infinity.