EPL Spring 2016, Project 1
Vector Container Basics
Development Phases A and B are described here

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, and to give you a little experience thinking about the difference between copies and moves. We're not experts with templates yet, and so we'll be leaving off some important functionality for this data structure. We might come back and have you add that functionality later on in the semester.

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. Methods and design requirements that will be tested in Phase A are marked with (A). Methods and design requirements that will be tested in Phase B are marked with (B) – note that Phase B testing will also include all Phase A tests.
- Some requirements are marked with one or two asterisk, such as (A*), (B**), etc. When a requirement is marked with one asterisk, then that requirement is considered moderately difficult. Students taking the course for EE379K credit should complete all of the unmarked requirements (zero asterisks), and most if not all of the single-asterisk requirements. Students taking the course for EE379K credit who find it challenging or impossible to complete the un-marked (no asterisks) requirements are not succeeding in the class and may be in danger of failing. Similarly, students taking the course for EE380L.5 credit should be able to complete all of the unmarked requirements, all of the single-asterisk requirements and many if not all of the double-asterisk requirements. If a requirement is marked in multiple ways, then all of the markings apply. For example, (A**/B*) implies that the requirement is expected during Phase A for EE380L.5 students and is expected during Phase B for EE379K students.

- (A**/B*) Do not initialize objects of type T (do not run the constructor for type T) unless necessary. Copy construct the objects in place when push_back is invoked. Destruct the objects immediately when pop_back is invoked. Note: when initializing a vector or when resizing the vector (with amortized doubling), you cannot use “new T[capacity]” to allocate storage. Using new[] with “operator syntax” will invoke T::T(void) for each object in the newly allocated space. Your newly allocated space must remain uninitialized. Only when an object is “push_back” (or push_front) to the object can you construct the object.
- (B) You should not need to use T::operator=(T) for this project. You will use both a copy constructor T::T(const T&) and a move constructor for T::T(T&&) Consider the following:
    - (B*) When resizing the array, objects from the old (smaller) storage block should be moved into the new (larger) storage block that you’ve allocated. The new block is uninitialized, so you will be using the move constructor for T – T::T(T&&)
    - (A**/B*) When doing push_back (or push_front) you may get an lvalue reference (use a const T& to capture the lvalue arg). In this case, you’ll be using the copy constructor for T – T::T(const T&)
    - (B*) When doing push_back (or push_front) you may instead get an rvalue reference (use a T&& to capture an rvalue reference arg). In this case, you should move the argument into your vector with T::T(T&&)
- You should provide a copy constructor, a move constructor, a copy assignment operator and a move assignment operator. You’ll also need the destructor – copy semantics are Phase A, move semantics are Phase B.
- (A) The pop_back and pop_front methods should not reallocate space. Just note the extra capacity.
- (A**/B*) The pop_back and pop_front methods should run the destructor on the object that was popped.
- The push_front function is an oddity. It’s included in this project to (1) give you an interesting design challenge to think about, and (2) to remove any temptation to solve this project by copying someone else’s design. When you do push_front, you should bear in mind that push_front changes the index positions assigned to all the elements currently in the array. The new value added will become position 0 (i.e., x[0] will be the new value added after x.push_front(v) is called). That means the previous x[0] is now x[1] and so on. It’s not hard to support this functionality – your operator[] will just calculate the distance from the true front of the array. You’ll have a pop_front which will do the inverse. Here are some details on push_front
    - push_front is an (A*/B) functionality. It is an advanced requirement for phase A, but should be included among the minimal functionality implemented in Phase B.
    - (A*/B) Until the first push_front is called, all excess capacity should be available for push_back to use. That doesn’t mean you have to

force the capacity to be zero at the front (or at the back). It does mean that the first push_back on an empty vector must not require a reallocation. The first push_front can require a reallocation (although it might not require one depending on the implementation). Once push_front is called, there is no guarantee of available capacity at the back.
    - (A*/B) You can have as much capacity at the front as you wish, provided the following is true. If push_front is called an infinite number of times in succession (with no intervening calls to push_back), then the time complexity must be linear. Note that this rule allows you to take a large vector (built with push_back), that has zero capacity at the front and as push_front is called, you can first add a small amount of capacity at the front, and then gradually double that capacity as needed. If push_back is called, you can do anything you want to the capacity in the front at that point.
    - (A*/B) The same is true for push_back. If an unbounded number of calls to push_back are made at any time (with no intervening calls to push_front) the time complexity must be linear.

Method List

Constructors:

- (A) vector(void) – creates an array with some minimum capacity (8 is fine) and length equal to zero. Must not use T::T(void). In fact, as long as Vector::Vector(int) is not called, you should never use T::T(void)
- (A) explicit Vector(uint64_t n) – create an array with capacity and length exactly equal to n. Initialize the n objects using T::T(void). As a special case, if n is equal to zero, you can duplicate the behavior of vector(void).
- (A) copy and (B) move constructors for vector<T> arguments (same type)
- (A) copy and (B) move assignment operators for vector<T> arguments (same type)
- (A) destructor
- (A) uint64_t size(void) const – return the number of constructed objects in the vector
- (A) 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
- (A*) const T& operator[](uint64_t k) const – same as above, except for constants.
- (A) 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.
- (B) void push_back(T&&) – same as above, but move construct the argument.
- (A*/B) void push_front(const T&) and (B) void push_front(T&&) -- similar to push_back, but add the element to the front of the Vector

- (A) void pop_back(void) and (B) void pop_front(void) – if the array is empty, throw a std::out_of_range exception (with any reasonable string error message you want). Otherwise, destroy the object at the end (or beginning) of the array and update the length (and any pointers you need). Do not reallocate storage, even if the vector becomes empty. Obviously, the available capacity increases by one. It is possible that a vector can have available capacity at both the front and back simultaneously.

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.
- Similarly if one or more successive calls to pop_front succeed and are followed immediately by an equal number of push_front calls, then no memory 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.
- Similarly, N push_front operations must complete in O(N) time (as N approaches infinity) regardless of the current state of the vector, provided the push_front operations are performed in sequence with no other push or pop operations.