

EPL Spring 2015, Project 1, Phase C

Vector Container Extensions

Due Feb 19, 2015.

Add some new functionality to your `Vector<T>` template class, making it more compliant with the C++ standard, and applying some of the additional knowledge you've gained about templates.

In the requirements below, the asterisk characters indicate increased challenge/complexity of the requirement. All students enrolled in EPL are expected to be easily able to complete all of the non-asterisk requirements. In addition, students enrolled in EE379K are expected to complete most or all of the single-asterisk (*) requirements. Students enrolled in EE380L.5 are expected to complete all of the single-asterisk (*) requirements and most or all of the double-asterisk (**) requirements.

Requirements:

- Create a random-access iterator type for your `Vector`. Be sure that at least `value_type` and `iterator_category` work correctly for your iterator type (you can assume that `iterator_traits` is used to access `value_type` and `iterator_category` – do not change or specialize the standard `iterator_traits` template to make it work with your iterator, make your iterator work with `iterator_traits`).
 - Provide `begin/end` functions for your `Vector`.
 - (*) Provide both `iterator` and `const_iterator` (and `begin/end` for each).
 - (**) Ensure that an `iterator` can be converted (without warnings or type casts) to `const_iterator`. Ensure that `const_iterator` cannot be converted to `iterator`.
 - (*) Design your `iterator` so that it throws the exception `epl::invalid_iterator` whenever the value of an invalid `iterator` is used. “Using the value” of an `iterator` includes comparison operations (with other `iterators`), dereferencing the `iterator`, incrementing the `iterator`, etc. Assigning to an `iterator`, for example, is not “using the value” of the `iterator`, it is assigning a new value to the `iterator` (and should not throw an exception).
 - For this project, an `iterator` must be invalid if there are any `push_back`, `pop_back`, `push_front` or `pop_front` operations applied to the vector, or if the vector is assigned a new value, or if the vector is “moved”.
 - (**) `epl::invalid_iterator` has three severity levels.
 - If the `iterator` references a position that no longer exists (i.e., the old position is out-of-bounds), the exception you throw must use the level `SEVERE`.

- If the iterator reference a position that is in-bounds, but the memory location for that position may have been changed (e.g., a reallocation has been performed because of a `push_back`, or a new assignment has been performed to the `Vector`), then the exception you throw must have the level `MODERATE`.
 - If the iterator is invalidated for any other reason, the exception must have the level `MILD`.
- (*) Write an `emplace_back` variadic member template function for your vector that constructs the object in place.
- ~~Create a member template constructor that will initialize a `Vector<T>` using a `Vector<T2>` as an argument. This constructor must compile without warnings or errors when objects of type `T` can be constructed using objects of type `T2`. i.e., if `T::T(T2)` exists (even if its explicit), then your constructor member template must compile and produce the obvious behavior.~~
 - ~~○ If `T` cannot be constructed using `T2`, then your member template should fail with a compile time error.~~
- ~~Create a member template assignment operator with equivalent behavior to the constructor above.~~
- Please put some thought (we won't grade this) into the question of whether the member template constructor and member template assignment operator are good or bad things (see above requirements that have been struck out – not required Spring 2015).
- (*) Create a member template constructor that takes an iterator pair `b` and `e` and initializes the `Vector` to contain copies of the values from `[b, e)`.
 - (**) Design your template constructor (using specialization/overloading if necessary) so that
 - When `b` and `e` are random-access iterators, only one allocation is necessary to construct the vector
 - When `b` and `e` are not random-access iterators, treat them as “input iterators”. That is, you must invoke `++b` only once for each position in the range `[b, e)` – specifically, you cannot save a copy of `b`, increment `b` to `e` (counting the number of elements) and then expect your copy to still reference the first element of the source input.
- Create a constructor that will initialize a `Vector` from a `std::initializer_list<T>`
 - Please note that for `Vector<int>` this constructor conflicts with the explicit `Vector::Vector(int)` constructor. When testing, it may be worthwhile to know that:


```
Vector x{42}; // uses Vector::Vector(std::initializer_list<int>)
```

```
Vector x(42); // uses Vector::Vector(int)
```