Homework Assignment 6

Due Date: 11:59, June 4, 2015

1 Introduction

In the class, you have learned about various containers and iterators provided by the C++ standard library. In this assignment, you are required to implement a container that "combines" the properties of `std::list<T>` and `std::vector<T>`.

In C++, the `std::vector<T>` class is simply a wrapper around a dynamically allocated array, and its `iterators` are simply pointers to the allocated array. Such iterators may be invalidated when the underlying array is reallocated due to insufficient space in the originally allocated memory.

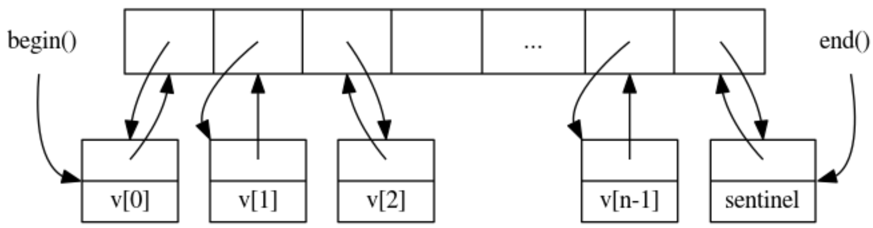
```
std::vector<int> v = {1, 2, 3, 4, 5};
std::vector<int>::iterator it = v.begin(); // *it == 1
while (v.size() < 10000) v.push_back(42);

std::cout << *it << std::endl; // Undefined behaivor if ‘v‘ was reallocated!
```

In the code above, repeatedly calling `push_back()` may trigger of a reallocation of the underlying arrays, thus invalidating all of the iterators created prior to the reallocation. As a result, dereferencing the now invalidated iterator `it` will likely crash the program, as it now points to memory locations that have been released.

On the other hand, a `std::list<T>` uses a doubly-linked list underhood, meaning that it never will invalidate iterators when its elements are inserted or removed due to reallocation. Such containers are said to be "stable", as iterators or references to their elements are valid as long as the element is not removed from the container. However, the linked list design of a `std::list<T>` implies that accessing elements not directly adjacent the current element requires linear traversal of the node pointers, an $O(n)$ operation, while `std::vector<T>` or traditional arrays which provide constant-time random access to all the elements.

As a result, Joaquín M. López Muñoz designed a hybrid of `std::vector<T>` and `std::list<T>`, the `stable_vector`: a STL-like container that provides fast random access to each element while also offering stability. A `stable_vector` of n elements is stored in the memory as the following:



In the example above, an array of $n+1$ pointers is allocated, with each pointer pointing to a node containing the actual element and a `up` pointer that points back to the array element. This allows each node to be fully stable when the array is reallocated, as we only have to "fix" the `up` pointers so that they point to the newly allocated array. Given an iterator to the node, we can quickly perform random access by following the `up` pointer to the array, perform the address calculation, and then follow the pointer down to the node.

Notice that for n elements, we need to allocate $n+1$ nodes, with the last one acting as the *sentinel* node to support the "one-past-the-end" semantics of iterators.

In this assignment, you are tasked to implement the class template `stable_vector<T>`, which exposes a STL-like container interface defined in the following sections. A skelton of the class template is already provided for you in the file `stable_vector.hpp`.

2 `stable_vector<T>`

All `T` will be default-constructable.

2.1 Public members

```
typedef T value_type;
typedef T* pointer;
typedef const T* const_pointer;
typedef T& reference;
typedef const T& const_reference;
typedef std::size_t size_type;
typedef std::ptrdiff_t difference_type;
class iterator;
class const_iterator;
```

Standard type defintions for STL containers and forward class declarations for `iterator` and `const_iterator`.

1. `stable_vector();`
   Construct an empty `stable_vector`. There should only be the sentinal node in the underlying vector.

2. `explicit stable_vector(const size_type n, const T& value = T());`
   Construct a `stable_vector` with `n` copies of `value`.

3. `template<typename InputIterator>`
   `stable_vector(InputIterator first, InputIterator last);`
   Construct a `stable_vector` from the range `[first, last)`.

   Note: In `stable_vector.hpp`, the last parameter is used to disable this template when an integral type is provided, so that it does not participate in the overload resolution to conflict with the constructor in (2). You may simply ignore (but not remove!) the third parameter.

4. `stable_vector(const stable_vector& rhs);`
   `stable_vector& operator=(stable_vector rhs);`
   `~stable_vector();`
   The copy constructor, copy-assignment operator and destructor. Both copy-assignment and destruction invalidates all `iterator`s originally stored in the `stable_vector`.

5. `void assign(size_type n, const T& value);`
   Set the `stable_vector`'s elements to `n` copies of `value`. The original elements are discarded and all `iterator`s are invalidated.

6. `template<typename InputIterator>`
   `void assign(InputIterator first, InputIterator last);`
   Set the `stable_vector`'s elements from the range `[first, last)`. The original elements are discarded and all `iterator`s are invalidated.

   Note: the last parameter is provided to disable this template as in (3).

7. `reference at(const size_type pos);`
   `const_reference at(const size_type pos) const ;`
   Return a (const) reference to the element at `pos`, or throw a `std::range_error` exception if `pos` is out of range. As exceptions have not been covered in the class, a default implementation is already provided for you in the file `stable_vector.hpp`.

8. `reference operator[](const size_type pos);`
   `const_reference operator[](const size_type pos) const;`
   Return a (const) reference to the element at `pos` assuming `pos < size()`, undefined behavior otherwise.

9. `reference front();`
   `const_reference front() const;`
   `reference back();`
   `const_reference back() const;`
   Return a (const) reference to the first or last element assuming `!empty()`, undefined behavior otherwise.

10. `iterator begin();`
    `const_iterator begin() const;`
    `const_iterator cbegin() const;`
    Return an `iterator` or `const_iterator` to the first element, or the sentinel node if the `stable_vector` is empty.

11. `iterator end();`
    `const_iterator end() const;`
    `const_iterator cend() const;`
    Return an `iterator` or `const_iterator` to the sentinel node.

12. `bool empty() const;`
    Return whether the `stable_vector` is empty or not, that is, contains no other nodes except the sentinel.

13. `size_type size() const;`
    Return the number of elements in the `stable_vector`.

14. `void clear();`
    Empty the `stable_vector` so that `size() == 0`. All `iterator`s are invalidated.

15. `iterator insert(const_iterator pos, const T& value);`
    `template<typename InputIterator>`
    `iterator insert(const_iterator pos, InputIterator first, InputIterator last);`
    Insert `value` or values in the range `[first, last)` at the position `pos`. All `iterator`s should remain valid.

16. `iterator erase(const_iterator pos);`
    `iterator erase(const_iterator first, const_iterator last);`
    Remove the element at `pos` or the range `[first, last)` from the `stable_vector`. All `iterator`s except those erased should remain valid.

17. `void push_back(const T& value);`
    Insert `value` at the end of the `stable_vector`. All `iterator`s should be remain valid.

18. `void pop_back();`
    Remove the last element in the `stable_vector` assuming it is not empty, undefined behavior otherwise. All `iterator`s except the last one, which is popped off, should remain valid.

19. `void resize(size_type count, const T& value = T());`
    Resize the `stable_vector` so that it contains only `count` elements. If `count < size()`, excessive elements at the end of the `stable_vector` is removed. If `count > size()`, copies of `value` are added to the end of `stable_vector`. All `iterator`s that refer to the first `count` elements should remain valid.

20. `void swap(stable_vector& other);`
    Swap the elements of `*this` with the `other` one. All `iterator`s that refer to elements in both `stable_vector`s should remain valid.

21. ```
    friend bool operator==(const stable_vector& lhs, const stable_vector& rhs);
    friend bool operator!=(const stable_vector& lhs, const stable_vector& rhs);
    friend bool operator< (const stable_vector& lhs, const stable_vector& rhs);
    friend bool operator<=(const stable_vector& lhs, const stable_vector& rhs);
    friend bool operator> (const stable_vector& lhs, const stable_vector& rhs);
    friend bool operator>=(const stable_vector& lhs, const stable_vector& rhs);
    ```
    Compare two `stable_vector`'s in lexicographical order.

    Hint: you may use STL algorithms `std::equal` and `std::lexicographical_compare` to implement these relational operators.

2.2 Private members

There is no restriction as to how you may design the internals the `stable_vector`, as long as the required semantics listed above are fulfilled. However, you are strongly recommended to follow the guideline here in order to ease your implementation.

```
typedef std::vector<node*> vector_type;
vector_type v;

struct node {
    T datum;
    typename vector_type::iterator up;
};
```

The vector `v` stores an array of pointers to `node`'s. The `node` struct is used to store the actual element in the member `datum`, and the iterator `up` provides a reference to the corresponding pointer in the internal vector `v`.

3 `stable_vector<T>::iterator`

The `iterator` provides a stable reference to an element in the `stable_vector`, as well as random-access to other elements within the same `stable_vector`.

3.1 Public members

```
typedef stable_vector::difference_type difference_type;
typedef stable_vector::value_type value_type;
typedef stable_vector::pointer pointer;
typedef stable_vector::reference reference;
typedef std::random_access_iterator_tag iterator_category;
```

Type defintions for an iterator required for STL algorithms to work. You do not need to modify these definitions.

1. `iterator(node* const n_ = nullptr);`
   Construct an `iterator` that internally stores a pointer to `n_`.
```

2. `iterator(const iterator& rhs);`
   `iterator& operator=(const iterator& rhs);`
   `~iterator();`
   The copy constructor, copy-assignment operator and destructor for `iterator`. The copied `iterator` should refer to the same element as `rhs`.

3. `reference operator*() const;`
   Dereference the `iterator`. A reference to the element is returned.

4. `pointer operator->() const;`
   The overload of the `->` operator, which returns a pointer to the element. As the syntax of this overloaded operator is not covered in the class, a default implementation is already provided for you in the file `stable_vector.hpp`.

5. `friend iterator operator+(iterator it, const difference_type n);`
   `friend iterator operator+(const difference_type n, iterator it);`
   `friend iterator operator-(iterator it, const difference_type n);`
   Return an `iterator` to elements in the referenced `stable_vector` offset by `n` (or `-n` in the case of `operator-`) positions. Note that `difference_type` is defined as `std::ptrdiff_t`.

   For example, given an iterator `it`, `*(it + 3)` should return a reference to the element that is 3 positions ahead of `*it`.

6. `friend difference_type operator-(const iterator lhs, const iterator rhs);`
   Return the number of elements between two iterators `lhs` and `rhs`.

7. `iterator& operator+=(const difference_type n);`
   `iterator& operator-=(const difference_type n);`
   Compound assignment versions of `operator+` and `operator-`.

8. `iterator& operator++();`
   `iterator operator++(int);`
   `iterator& operator--();`
   `iterator operator--(int);`
   The prefix/postfix increment/decrement operators, which modify the iterator so that it points the previous/next element. Note that postfix versions should return the original iterator to be consistent with the semantics of C and C++.

9. `reference operator[](const difference_type n);`
   `const_reference operator[](const difference_type n) const;`
   Return a reference to the element offseted by `n` positions. Recall that both C and C++ defines `a[n]` as `*(a + n)`.

10. `operator const_iterator() const;`
    Implicitly convert an `iterator` into a `const_iterator`.

11. `friend bool operator==(const iterator lhs, const iterator rhs);`
    `friend bool operator!=(const iterator lhs, const iterator rhs);`
    `friend bool operator< (const iterator lhs, const iterator rhs);`
    `friend bool operator<=(const iterator lhs, const iterator rhs);`
    `friend bool operator> (const iterator lhs, const iterator rhs);`
    `friend bool operator>=(const iterator lhs, const iterator rhs);`
    Compare two iterators `lhs` and `rhs` within the same `stable_vector`.

3.2 Private members

The `iterator` should internally store a pointer to the `node`. As the `node` remains valid in the `stable_vector` unless removed, the `iterator` is valid as long as the element is not erased.

```
node* n;
```

4 `stable_vector<T>::const_iterator`

The interface and semantics of `const_iterator` is mostly the same as `iterator`, except that it cannot modify the referenced element. The interface is listed in the appendix for completeness.

5 Notes and Hints

- Your `stable_vector` should not impose any artifical limits on the number of elements. In other words, your `stable_vector` should be able to store as many elements as memory allocation allows, and fixed-sized allocations should be avoided.
- Follow the **Don't Repeat Yourself** (DRY) principle to greatly decrease the number of functions you need to implement. For example, `push_back(v)` may be implemented as `insert(end(), v)`, `operator>=` can be implemented by inverting the result of `operator<`, and the increment operators `operator++` for iterators may be implemented in terms of `operator+=`.
- Your classes must correctly manage all allocated memory during copy construction, copy assignment, destruction or any other operations. There should be no memory leaks, dangling pointers or uses of uninitialized memory within your implementation. If the presence of memory-related bugs cause the test program to crash, you will receive **zero** credits for the test case.
- Test all edge cases of your implementation, for example, when handling an empty `stable_vector`. You should also test your implementation with multiple compilers and optimization settings, which may help revealing bugs within your program.
- You should provide a default **no-op** implementation if you have any un-implemeneted functions, so that your source file can still be successfully compiled and linked by TAs.
- Print your own debug messages to standard error by using `std::cerr`. TAs will ignore all outputs on the standard error stream.

6 Submission

Please archive your homework into a single zip file and submit the zipped file to E3. The zipped file must be named by your student ID (e.g. `0123456.zip`) and **must** contain the following file(s):

- `stable_vector.hpp`: Your implementation of `stable_vector`.

You are allowed to submit and include other header files you have written. TAs will compile your homework using the following command on a Linux workstation:

```
$ clang++ -std=c++14 -Wall -Wextra -pedantic -O2 main.cpp
```

where `main.cpp` will `#include` your `stable_vector.hpp` to test your implementation.

7 External References

- Iterator invalidation rules in C++0x: `http://stackoverflow.com/a/6442829/509880`
- Using Valgrind to detect memory errors: `http://valgrind.org/docs/manual/quick-start.html`

8 Sample Code

```cpp
#include <cassert>
#include <algorithm>
#include <iostream>
#include <string>

#include "stable_vector.hpp"

int main() {
    std::string s = "able was I ere I saw elba";
    stable_vector<int> v1;
    const stable_vector<int> v2(5, 42); // v2 = {42, 42, 42, 42, 42}
    stable_vector<char> v3(s.begin(), s.end());
    stable_vector<char> v4;

    assert(v1.empty());
    assert(v1.size() == 0);
    assert(v1.begin() == v1.end());

    assert(v2.begin() + v2.size() == v2.end());
    for (stable_vector<int>::const_iterator it = v2.begin(); it < v2.end(); ++it)
        assert(*it == 42);

    v4.assign(v3.begin(), v3.end());
    std::reverse(v4.begin(), v4.end());
    assert(v3 == v4);
    std::sort(v3.begin(), v3.end());
    for (stable_vector<char>::const_iterator it = v3.begin(); it != v3.end(); ++it)
        std::cout << *it; // "      IIaaaabbeeeellrssww"
    std::cout << std::endl;

    stable_vector<int> u1(v2); // u1 = {42, 42, 42, 42, 42}
    stable_vector<int> u2; u2 = u1;
    stable_vector<int>::iterator it = --u2.end();
    *it = 1; // u2 = {42, 42, 42, 42, 1}
    assert(u1 > u2); //               ^it
    for (int i = 0; i < 10000; ++i)
        u2.push_back(i); // u2 = {42, 42, 42, 42, 1, 0, 1, 2, 3, ...}
    std::cout << *it << std::endl; // 1           ^it
    u2.erase(u2.begin(), u2.begin() + 4); // u2 = {1, 0, 1, 2, 3, ...}
    std::cout << it[4] << std::endl; // 3          ^it         ^(it + 4)
    u1.swap(u2);                   // u1 = {1, 0, 1, 2, 3, ...}
    std::cout << *it << std::endl; // 1     ^it

    return 0;
}
```

9 Appendix

10 stable_vector<T>::const_iterator

```
class const_iterator {
public:
    typedef stable_vector::difference_type difference_type;
    typedef stable_vector::value_type value_type;
    typedef stable_vector::pointer pointer;
    typedef stable_vector::reference reference;
    typedef std::random_access_iterator_tag iterator_category;

    const_iterator(const node* const n_ = nullptr);
    const_iterator(const const_iterator& rhs);
    const_iterator& operator=(const const_iterator& rhs);
    ~const_iterator();

    friend const_iterator operator+(const_iterator it, const difference_type n);
    friend const_iterator operator+(const difference_type n, const_iterator it);
    friend const_iterator operator-(const_iterator it, const difference_type n);
    friend difference_type operator-(const const_iterator lhs, const const_iterator rhs);

    const_iterator& operator+=(const difference_type n);
    const_iterator& operator-=(const difference_type n);

    const_reference operator*() const;
    const_pointer operator->() const { return std::addressof(operator*()); }

    const_iterator& operator++();
    const_iterator operator++(int);

    const_iterator& operator--();
    const_iterator operator--(int);

    const_reference operator[](const difference_type n) const;

    friend bool operator==(const const_iterator lhs, const const_iterator rhs);
    friend bool operator!=(const const_iterator lhs, const const_iterator rhs);
    friend bool operator< (const const_iterator lhs, const const_iterator rhs);
    friend bool operator<=(const const_iterator lhs, const const_iterator rhs);
    friend bool operator> (const const_iterator lhs, const const_iterator rhs);
    friend bool operator>=(const const_iterator lhs, const const_iterator rhs);

private:
    const node* n;
};
```