Assignment 1: Object-Oriented C

C++ Programming Course, Winter Term 2016

0: Prerequisites

0-1: GoogleTest

Install googletest in your home directory:

```
$ mkdir ~/build
$ mkdir ~/opt
$ git clone https://github.com/google/googletest.git ~/build/googletest.git
$ cd ~/build/googletest.git
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/opt/gtest
$ make install
$ echo "export GTEST_BASE=$HOME/opt/gtest" >> ~/.zshrc
```

0-2: Tools

On the test system (VM image or SSH account on server), `tmux` and `vim` (actually `nvim`) are installed and configured nicely. If you have no previous experience with these programs, make yourself familiar with both to make your life easier.

0-3: Notes on Running Tests

Source files for your implementation are located in `assignment-01/solution`. To build and run your code and the test suite, use:

```
$ cd assignment-01/test
$ make run
# same as
$ make clean ; make && ./testsuite.bin
```

To run individual test cases, specify the parameter `--gtest_filter` when running the test suite:

```
$ ./testsuite.bin --gtest_filter="VectorTest.StandardConcept"
$ ./testsuite.bin --gtest_filter="StackTest.*"
```

Write basic implementations in the `solution/*.c` files first, the build will fail in the linker stage otherwise because of undefined references.

If you modify the test suites for debugging, remember to validate your solution using the original test implementation before submitting.

1: Classes in C - "Wax on, Wax off"

Remember the implementation of the `String` class presented in the last course session. Let's implement some more.

We first define the `Standard` concept that specifies standard operations which must be provided for any type:

.	.
`<T>__new() -> o`	creates a default instance of type
`<T>__copy(r) -> o`	creates a new instance `o` as a copy of instance `r`
`<T>__delete(o)`	destroys instance `o` and releases its memory
`<T>__assign(o,r) -> o'`	assigns instance `r` to instance `o`
`<T>__equals(o,r) -> b`	`1` if `o` and `r` are equal, otherwise `0`

Obviously, the functions implementing these operations for a specific type are named according to the type, such as `wombat__new` for type `Wombat`.

1-1: Vector Class in C

Implement a class in C that satisfies the `Vector` concept with the following semantics:

.	Semantics
`vector__size(v) -> n`	number of elements contained in vector `v`
`vector__empty(v) -> b`	`0` if vector `v` has no elements, otherwise 1
`vector__begin(v) -> i`	iterator `i` referencing the first element in `v`
`vector__end(v) -> i`	iterator `i` referencing past the final element in `v`
`vector__push_back(v,e)`	append value `e` as last element in vector `v`
`vector__pop_back(v) -> e`	append value `e` as last element in vector `v`

.	Semantics
`vector__push_front(v,e)`	append value `e` as first element in vector `v`
`vector__pop_front(v) -> e`	append value `e` as first element in vector `v`
`vector__at(v,o) -> e`	returns value `e` at offset `o` in vector `v`

Validate your implementation with the test specification in test suite `VectorTest`.

Some thoughts on this:

- We use regular pointers as iterators. What if we implemented a list?

  Answer 1: Iterators

  A list which stores the adress of every element in the vector, would enable us to call every element very fast. Thus, it is quite complex to implement and update.

- Look up the interface of `std::vector` at http://cppreference.com. The vector concept in the STL does not specify some of the methods above. Which ones? Why? If the answer does not seem obvious to you, implement all methods first. You'll see.

  Answer 2: Interface

  The `std::vector` does not implement `push__front` and `pop__front`. It is very inefficient since it constantly shuffling the existing data up every time we do an insertion. A `deque` is designed to perform these operations.

1-2: Deque Class in C

Implement a class in C that satisfies the `Deque` concept with the following semantics:

.	Semantics
`deque__size(d) -> n`	number of elements contained in deque `d`
`deque__empty(d) -> b`	`0` if deque `d` has no elements, otherwise 1
`deque__push_back(d,e)`	append value `e` as last element in deque `d`
`deque__push_front(d,e)`	append value `e` as first element in deque `d`
`deque__pop_back(d) -> e`	append value `e` as last element in deque `d`
`deque__pop_front(d) -> e`	append value `e` as first element in deque `d`

Validate your implementation with the test specification in test suite `DequeTest`.

Keep in mind that you already have an implementation of the `Vector` concept

that might come handy. Note that it's not about efficiency, yet. See bonus assignments below for more on this.

1-3: Stack Class in C

Implement a class in C that satisfies the `Stack` concept with the following semantics:

.		Semantics
`stack__size(s)`	`-> n`	number of elements contained in stack `s`
`stack__empty(s)`	`-> b`	`0` if stack `s` has no elements, otherwise 1
`stack__push(s,e)`		put value `e` on the stack
`stack__pop(s)`	`-> e`	remove value `e` from the stack

Validate your implementation with the test specification in test suite `StackTest`.

1-4: Some Thoughts to Meditate Over

Is there a `stack` container in the STL? How about a `deque` container? Go to `cppreference.com` and find out.

Answer 3: Stack & Deque in CPP

Yes there is a `Stack` "is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure."
"std::deque (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements. "

Now, see if you can find something special about their definitions. Can you think of a reason for their differences?

Answer 4: Iterators

`Stack` is designed to perform FILO and `Deque` to insert in the front or back while keeping the memoryadress of the rest of the elements.

1-5: Reverse Algorithm

This assignment is just about experimenting:

Implement the methods `vector__reverse(v)`, `deque__reverse(v)` and `stack__reverse` that reverse the element order of the respective container instance. For this:

- Try to formulate the semantics of the general reverse operation `<T>__reverse` with respect to existing methods specified above.
- Extend the test suites by tests of the reverse-methods.
- Finally, implement the reverse-methods and run the test suite to check their correctness.

What do you notice about the similarities and differences in your implementations of these functions?

Answer 5: Reverse

Since I use a quit easy but expensive way to implement the `reverse` functions those are completly similiar. I create a new object and insert the last element of the original object to the start of the new one.

Now, have another look at the question in 1-4. If you did not find an answer already, can you now think of a reason for the curious definition of `std::stack` in the STL?

1-6: Outlook to Upcoming Course Session

Assuming you had to implement the test cases yourself.

What feature are you missing in C that would drastically simplify the implementation of the test cases for `Vector`, `Deque` and `Stack`?

Answer 6: Outlook

A templated function which uses a Type and/or a word which is inserted into a templated test-case. In C I implemented the concepts very similiar since the test-cases do not check whether the memory allocation (etc) is completly according to the concept.

Hint: You might have noticed that the test suites are quite identical. Why? Is there an advantage in this? Can you exploit this in C?

1-X: Bonus Assignment: Efficient Memory Management

Your implementation of `Vector` and `Deque` from assignments 1-1 and 1-2 should have correct semantics, but a basic "just formally correct" implementation is quite inefficient with respect to memory management.

- Refactor `deque__push_front` and `deque__pop_front` such that worst-case complexity O(1) is maintained.
- Refactor `vector__push_back` and `vector__pop_back` to improve their average case complexity.
- Your modified implementation should still pass the test suites, of course.

As usual, you should have a close look at cppreference.com first.