Programming Exercise 4

EE312 Fall 2014

To complete during recitation, Oct 2/3

TWO POINTS

You will be provided with a simple program that contains at least one error. Your challenge during this exercise is to design a test program that confirms the existence of that error. You are encouraged to pursue a test program that, if all tests were passed, would confirm the absence of errors.

STEP1: Understanding the program that you're testing

The PUT, or "program under test" for this exercise is the Vector case study that we will eventually do in class. You're not expected to understand how the design of a Vector data structure works for this exercise. In fact, good tests can often be written without any knowledge of the PUT implementation, just the PUT's specification. So, if you don't know how it works, so much the better!

A Vector is a data structure that's supposed to work a lot like an array. Our Vector simulates an array of N elements. To access the k^{th} element, we use the get() and set() functions. The get() function reads an element and the set() function writes an element. However, unlike ordinary arrays, our Vector data structure performs bounds checking on every read and write access. A correct implementation of the Vector should crash (assert failure) if a get() or set() is invoked trying to access an array element that is out-of-bounds.

If a Vector can perform bounds checking, then a Vector must know how many elements are supposed to be in the array it's simulating. That's convenient for client programmers – if the client forgets (or simply never knew) how much data was supposed to be in a Vector he/she is working with, then a call to the size() function can be used to determine how many elements are in the Vector. If you look at Source.cpp in the Exercise, and take a look at the printVector function, you can see how printVector uses the get and size functions to read all the elements in a Vector (and prints them to the screen).

In addition to knowing its own size and performing bounds checking, Vectors have one other very important advantage over arrays. A Vector can be resized. A new element can be added (to the end of the array) by calling the push_back function. A Vector with 10 elements becomes a Vector with 11 elements when push_back is called. The value "pushed" becomes the new last element in the Vector. Similarly, the pop_back function will reduce the size of a Vector by one. The complete list of functions for the Vector abstract data type is contained in Vector.h. Read this file and become familiar with how these functions are supposed to work.

STEP2: Writing some real tests
There is a fairly obvious bug in the push_back function. Design several tests that call push_back and see if you can confirm the bug. **Remember, the goal is not to find the bug by looking at the Vector push_back function!!!** It's there, trust me. The goal is to write a test that should work, but that fails because of this bug. As I said, the bug is pretty obvious, so once you've got a test that proves that the bug exists, go ahead and fix the bug, then move on to STEP3.

What's a test? A test is a function that you write that calls one or more of the Vector functions in some sequence. In order for the test to be useful, your function must produce some output and/or do some calculations that determine whether or not the Vector functions you called worked correctly. For example, here's a simple test:

```
bool testCreate(void) {
	Vector v = createVector(10);
	assert(size(&v) == 10); // confirm that v has 10 elements
	for (uint32_t k = 0; k < 10; k += 1) {
		assert(get(&v, k) == 0); // confirm that each element is zero
	}
	return true;
}
```

This function tests both the createVector function (with one specific input) and the get function. The function expects the Vector v to be created with 10 elements all set to zero, and it checks that this actually happened. Note that the function doesn't check very much. What if createVector works correctly for vectors with 10 elements, but not correctly when creating a Vector with 0 elements? Does the get function correctly do the bounds checking?

A *Test Suite* is a collection of functions (a collection of tests) that check all of the features of your PUT. Write a small test suite that attempts to prove the existence of the bug in push_back.

STEP3: Writing some thorough tests
There is at least one other bug that I know about, it's subtle, and the bug only bites when at least two of the Vector functions are used in a specific combination. I'll give you this hint, to reveal this bug, you will need to call push_back. But, you'll have to call other functions too. Developing good tests is hard, and it's both art and science (ECE offers an entire course just on this topic!) For starters, your test_suite should try to call at least every function, and should call most of the functions with every interesting combination of parameters. What does "interesting" mean? Well, for push_back, "interesting" could mean calling push_back when the Vector's capacity and length should be equal, calling push_back when the Vector is empty. For a function like pop_back, "interesting" could mean removing the last element from a Vector.

SPECIAL NOTE ON DESTROY AND MEMORY LEAKS
We're still debating how we want you to track memory leaks. Since we don't have a definite plan, for this exercise, we want you to ignore memory leaks. The program may have bugs related to how the destroy function works. Please do not go in search of those bugs.