

CS 14: Data Structures and Algorithms

Lab 3

Peter H. Fröhlich
phf@cs.ucr.edu

Wagner Truppel
wagner@cs.ucr.edu

Out on: January 20, 2003

Due by: January 26, 2003

Updated: January 22, 2003

Remember to follow the instructions and policies for assignments given on the course website. This week's assignment consists of **three** separate parts:

- an **in lab** programming exercise (1)
- an **at home** programming exercise (2)
- two **at home written** exercises (3 & 4)

The “in lab” part is due at the **end** of your **lab section**. The “at home” parts are due **before 8:00 pm** on the date given above. You have to use the department's electronic turnin service for **everything**, including the **written** part. Put your solution for the latter into the **text file** `solution-3.txt` and turn it in together with the other files.

1 In-Lab: Testing

You can download an archive for this lab from the course website. Inside, you will find a file `weirdcounter` which contains an implementation of the `Counter` interface. Your task is to develop a **test driver** according to the specification of ADT `Counter` given in Figure 1. That is, write a file `testweird.cpp` with a `main` function that creates `WeirdCounter` objects and calls their functions in order to determine whether the class is **indeed** an implementation of the ADT `Counter`. **Do not look at the source code for `WeirdCounter`!**

Hints: You are allowed to use last week's test driver `testcounter.cpp` as a starting point, but you should remember that it contains only a few of the necessary test cases. Also, it might be useful to define additional functions besides `main` to avoid repetitive code.

2 At-Home: Arrays

The archive you downloaded also contains a file `array`. It defines an *abstract base class* describing the **interface** of the ADT `Array` we discussed in class (see Figure 2). There's also a program `testarray.cpp` which tries to create an object of the class `SimpleArray` implementing the `Array` interface. Your task is to complete the partial implementation given in the file `simplearray-skel` (for “skeleton,” but your final file should be called `simplearray` of course). Your class `SimpleArray` should conform to the ADT `Array` and pass all the tests performed in `testarray.cpp`. Note that the given test cases do **not** cover all the required functionality. Some of the tests also target **C++ specific** features such as a working copy constructor and assignment operator...

Hints: Once again, your class should be derived publicly from `Array`, and you should use exceptions to signal various errors that can occur. Throwing the exception `std::domain_error` is as good a choice as any for now, but there's also another exception called `std::out_of_range` that you could use.

3 At-Home: Variables

The notion of a **variable** in programming languages like C++ can be thought of as an abstract data type. Using the notation from class, your task is to define an ADT `Variable` that **accurately** captures what a variable is.

Hints: We did the same thing for **arrays** in the lecture, and you should follow a similar approach: Identify

<p>adt Counter aka Subrange</p> <p>uses Any, Integer</p> <p>defines Counter</p> <p>operations new: Integer × Integer → Counter lower: Counter → Integer upper: Counter → Integer get: Counter → Integer set: Counter × Integer → Counter inc: Counter → Counter dec: Counter → Counter</p> <p>preconditions new(l, u): $l \leq 0 \leq u$ set(c, v): $\text{lower}(c) \leq v \leq \text{upper}(c)$ inc(c): $\text{get}(c) < \text{upper}(c)$ dec(c): $\text{lower}(c) < \text{get}(c)$</p> <p>axioms get(new(l, u)) = 0 lower(new(l, u)) = l upper(new(l, u)) = u get(set(c, v)) = v lower(set(c, v)) = lower(c) upper(set(c, v)) = upper(c) get(inc(c)) = get(c) + 1 lower(inc(c)) = lower(c) upper(inc(c)) = upper(c) get(dec(c)) = get(c) - 1 lower(dec(c)) = lower(c) upper(dec(c)) = upper(c)</p>	<p>adt Array aka DynamicArray</p> <p>uses Any, Integer</p> <p>defines Array<T>: Any></p> <p>operations new: Integer × T → Array<T> length: Array<T> → Integer put: Array<T> × Integer × T → Array<T> get: Array<T> × Integer → T</p> <p>preconditions new(l, t): $0 < l$ put(a, i, t): $0 \leq i < \text{length}(a)$ get(a, i): $0 \leq i < \text{length}(a)$</p> <p>axioms length(new(l, t)) = l get(new(l, t), i) = t length(put(a, i, t)) = length(a) get(put(a, i, t), j) = if i = j then t else get(a, j)</p>
--	---

Figure 1: The Abstract Data Type Counter

Figure 2: The Abstract Data Type Array

- Why are we **able** to define these operators in the abstract base class already?
- What is the **conceptual** problem of defining these operators in the abstract base class?

Hint: A careful study of the comment in file `counter` gives a lot of this away, but **don't** just “rip off” the comment!

the essential operations, determine whether there are any preconditions, and finally state sensible axioms that you expect to be true whenever you work with variables.

4 At-Home: Counter

The abstract base class `Counter` expresses the **interface** of ADT `Counter` using C++ constructs. Besides several **pure** virtual member functions, it also defines `operator++` and `operator--`. Answer the following in **one short sentence each**:

- Why would we **want** to define these operators in addition to `inc` and `dec`?