CS303 Data Structures Assignment #3

attachments and source available at https://github.com/alexskc/cs303

Aleksander Charatonik

September 30, 2018

1

A `const_iterator` is useful in preventing modifying the referenced value. It's simply about const correctness, and informs what the programmer should be able to do. `iterator`, by contrast, has read-write access, and is useful in scenarios where that is necessary.

2

a

An `iterator`. Regardless of whether you have an array-based structure, or a linked-list structure, you need to be able to change either the value of the next item, or the pointer to the next item.

b

`iterator` as well. You are modifying data, so you cannot be read-only.

c

For this one, a `cost_iterator` will suffice. We are not changing any data.

d

`iterator` as well. We can avoid changing the element pointed to if we're using a linked-list structure, but we still need to change the element before it to point to the new item. And of course, in an array-based structure, we're going to be moving elements around in the array to make space for the new element.

3

See attached `reverser.cpp`

4

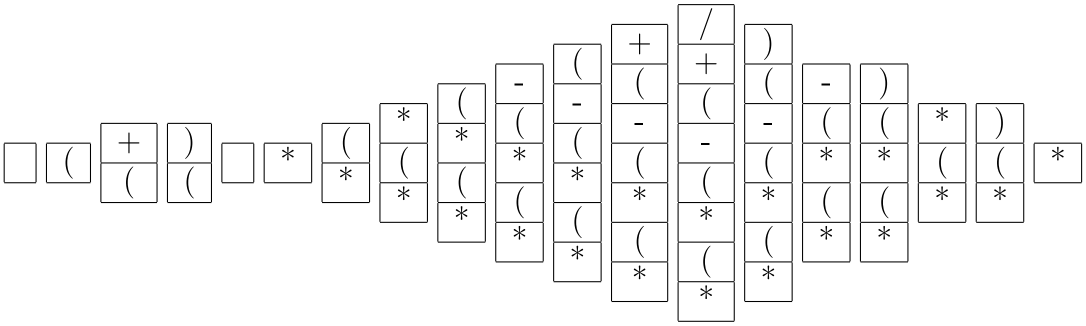
Expression	**Action**	**Stack**
$\underset{\uparrow}{10}\ \ 2\ \ *\ \ 5\ \ /\ \ 6\ \ 2\ \ 5\ \ *\ \ +\ \ -$	Push 10	10
$10\ \ \underset{\uparrow}{2}\ \ *\ \ 5\ \ /\ \ 6\ \ 2\ \ 5\ \ *\ \ +\ \ -$	Push 2	2 10
$10\ \ 2\ \ \underset{\uparrow}{*}\ \ 5\ \ /\ \ 6\ \ 2\ \ 5\ \ *\ \ +\ \ -$	Eval *	20
$10\ \ 2\ \ *\ \ \underset{\uparrow}{5}\ \ /\ \ 6\ \ 2\ \ 5\ \ *\ \ +\ \ -$	Push 5	5 20
$10\ \ 2\ \ *\ \ 5\ \ \underset{\uparrow}{/}\ \ 6\ \ 2\ \ 5\ \ *\ \ +\ \ -$	Eval /	4
$10\ \ 2\ \ *\ \ 5\ \ /\ \ \underset{\uparrow}{6}\ \ 2\ \ 5\ \ *\ \ +\ \ -$	Push 6	6 4
$10\ \ 2\ \ *\ \ 5\ \ /\ \ 6\ \ \underset{\uparrow}{2}\ \ 5\ \ *\ \ +\ \ -$	Push 2	2 6 4
$10\ \ 2\ \ *\ \ 5\ \ /\ \ 6\ \ 2\ \ \underset{\uparrow}{5}\ \ *\ \ +\ \ -$	Push 5	5 2 6 4
$10\ \ 2\ \ *\ \ 5\ \ /\ \ 6\ \ 2\ \ 5\ \ \underset{\uparrow}{*}\ \ +\ \ -$	Eval *	10 6 4
$10\ \ 2\ \ *\ \ 5\ \ /\ \ 6\ \ 2\ \ 5\ \ *\ \ \underset{\uparrow}{+}\ \ -$	Eval +	16 4
$10\ \ 2\ \ *\ \ 5\ \ /\ \ 6\ \ 2\ \ 5\ \ *\ \ +\ \ \underset{\uparrow}{-}$	Eval -	-12

5

y - 7 * 35 + 4 / 6 - 10
Operator Stack:

	-	* -	+	/ +	- +

(x + 15) * (3 * (4 - (5 + 7 / 2)))
Operator Stack:



6

We should be able to simply modify the OPERATORS string to also include `^`. We should note that C++ doesn't have a `^` operator, instead we use the `pow()` function in `cmath`, however our program doesn't need to concern itself with that.

7

If order doesn't matter, and the original queue doesn't matter, we can simply always read the front, print it, and then pop it. This would display all the elements, but in a backwards order, and the original queue would be gone.

If we want to display in the original order, and preserve the original queue, then what we should do is:
1. Read the front element
2. Push it to the back
3. Pop it from the front
4. Repeat until the queue is backwards

5. Read the front element
6. Print it out
7. Push it to the back
8. Pop it from the front
9. Repeat until the queue retains its original order.

8

```
void move_to_rear(queue<T> queue) {
  queue.push(queue.front())
  queue.pop();
  }
```