CS105C: Computer Programming C++ Programming Assignment #3

Adrian Trejo Nuñez
atrejo@cs.utexas.edu

Due: 11:59pm October 8, 2017

The purpose of this assignment is to familiarize yourself with using STL container classes and other basic C++ constructs.

1 Description

We've talked about memory management in C++ and how it differs from a language like Java. Although manual memory management is efficient, it is sometimes error prone. A garbage collector is a runtime system that allows programmers to allocate pointers to objects whenever they need, but not require them to free the memory when it is no longer needed. We will write a simple garbage collector.

2 Directions and Hints

Download the the assignment file `assignment3.tar.gz` to the lab computers (physically or using `scp`). Unpack the tarball in a new directory:

```
> mkdir assignment3
> mv assignment3.tar.gz assignment3
> cd assignment3
> tar -xvf assignment3.tar.gz
```

Take a look at the skeleton code for the various classes. You will be modifying the `VirtualMachine` class and the `Heap` class.

2.1 Virtual Machine

The `VirtualMachine` class models a simple stack-based virtual machine similar to a stripped-down version of the JVM. It contains two segments of memory: the `Stack`

segment and the `Heap` segment, both of which are represented by a C++ class with the same name. The `Stack` class represents the program stack and for simplicity, we assume there is no limit on its capacity. The `Heap` class represents the program heap, which has limited capacity to be specified in its constructor.

The virtual machine supports the allocation of two different kinds of objects: integers and pairs. An integer object holds a single integer, and a pair object holds a pair of pointers to other objects. Objects can only be allocated on the heap, not the stack. The stack only holds pointers to heap objects.

2.2 Memory Representation

In our implementation, the `Heap` class allocates memory in units of a `MemoryCell`. Every `MemoryCell` has a unique memory address attached to it, and the heap is just a collection of memory cells.

Every `MemoryCell` is categorized into four different types:

- Null cell does not hold any valid data.
- Tag cell holds a tag value, which can be either an integer or a pair tag.
- Value cell holds an integer value.
- Pointer cell holds a pointer to another memory cell.

Integer and pair objects have different memory layouts on the heap and require a different number of memory cells to store. An integer object occupies two consecutive memory cells: the first cell holds an integer tag, and the second cell holds the integer value. A pair object occupies three consecutive memory cells: the first cell holds a pair tag, the second cell holds a pointer to the first component, and the third cell holds a pointer to the second component. The address of an object is the address of the first memory cell it occupies [1].

For example, here is the memory layout of a heap that contains a pair object and two int objects [2]. The two integer objects have values 10 and 11 and the pair object points to both of these integers.

```
{ a0 -> PAIR_TAG; a1 -> a3; a2 -> a5;
  a3 -> INT_TAG; a4 -> 10;
  a5 -> INT_TAG; a6 -> 11 }
```

The code for the `MemoryCell` class is provided for you. You can query the type of a memory cell object by calling `getCellType()`, and you can get or set the data associated with the cell through the exposed member functions. Note that in order to invoke `getTag()`, `getValue()`, or `getPointer()` on a memory cell, the cell must be a tag cell, a value cell, or a pointer cell, respectively. Otherwise, a runtime exception will be thrown.

[1] In our setting, this cell is always going to be a tag cell.

[2] $a0, ..., a6$ are consecutive addresses of memory cells.

2.3 Instructions

The virtual machine supports nine instructions, each of which is implemented by a corresponding member function in the `VirtualMachine` class:

- `pushInt(n)`: Creates an integer object on the heap and pushes the pointer on the stack.
- `pushPair()`: Pops two pointers from the top of the stack(p_2 then p_1) and creates a pair object on the heap with components p_1 and p_2. If the stack is empty or only contains one element, a `StackException` should be thrown.
- `pop()`: Pops an element off the stack. If the stack is empty, a `StackException` should be thrown.
- `dup()`: Push a copy of the top element of the stack. If the stack is empty, a `StackException` should be thrown.
- `loadFirst()`: Pops an element from the stack, which is assumed to be a pointer to a pair object, and pushes it's first component on the stack. If the stack is empty, a `StackException` should be thrown. If the top of the stack is not a pointer to a pair, a `RuntimeTypeError` should be thrown.
- `loadSecond()`: Pops an element from the stack, which is assumed to be a pointer to a pair object, and pushes it's second component on the stack. If the stack is empty, a `StackException` should be thrown. If the top of the stack is not a pointer to a pair, a `RuntimeTypeError` should be thrown.
- `storeFirst()`: Pops an element from the stack, which is assumed to point to an object, and sets it as the first component of the next element on top of the stack, which is assumed to point to a pair object. If the stack is empty or only contains one element, a `StackException` should be thrown. If the second element on the stack is not a pointer to a pair object, a `RuntimeTypeError` should be thrown.
- `storeSecond()`: Pops an element from the stack, which is assumed to point to an object, and sets it as the second component of the next element on top of the stack, which is assumed to point to a pair object. If the stack is empty or only contains one element, a `StackException` should be thrown. If the second element on the stack is not a pointer to a pair object, a `RuntimeTypeError` should be thrown.
- `add()`: Pops the top two elements from the stack, which are assumed to be pointers to integer objects, and allocates a new integer object on the heap with value equal to the sum of the two integers, and pushes the pointer to the new object onto the stack. If the stack is empty or only contains one element, a `StackException` should be thrown. If the elements on top of the stack are not pointers to integer objects, a `RuntimeTypeError` should be thrown

We will program our virtual machine using these instructions. Here is an example[3]:

[3] $b0, ..., b5$ are also consecutive addresses.

```
// Creates the vm with a heap that can hold 9 cells
VirtualMachine vm(9);
// stack: empty, heap: empty
vm.pushInt(1);
// stack: < a0 >, heap: { a0 -> INT_TAG; a1 -> 1 }
vm.pushInt(2);
// stack: < a0, a2 >
// heap: { a0 -> INT_TAG; a1 -> 1; a2 -> INT_TAG; a3 -> 2 }
vm.pushPair();
// stack: < a4 >
// heap: { a0 -> INT_TAG; a1 -> 1; a2 -> INT_TAG; a3 -> 2
//         a4 -> PAIR_TAG; a5 -> a0; a6 -> a2 }
vm.dup();
// stack: < a4, a4 >
// heap: { a0 -> INT_TAG; a1 -> 1; a2 -> INT_TAG; a3 -> 2
//         a4 -> PAIR_TAG; a5 -> a0; a6 -> a2 }
vm.storeFirst();
// stack: < a4 >
// heap: { a0 -> INT_TAG; a1 -> 1; a2 -> INT_TAG; a3 -> 2
//         a4 -> PAIR_TAG; a5 -> a4; a6 -> a2 }
vm.loadSecond();
// stack: < a2 >
// heap: { a0 -> INT_TAG; a1 -> 1; a2 -> INT_TAG; a3 -> 2
//         a4 -> PAIR_TAG; a5 -> a4; a6 -> a2 }
vm.pushInt(3);
// stack: < a2, a7 >
// heap: { a0 -> INT_TAG; a1 -> 1; a2 -> INT_TAG; a3 -> 2
//         a4 -> PAIR_TAG; a5 -> a4; a6 -> a2;
//         a7 -> INT_TAG; a8 -> 3 }
vm.add();
// We ran out of heap memory and cannot allocate a new integer object
// However, some of the heap objects can be reclaimed
// The garbage collector runs and the memory looks like this now:
// stack: < b0, b2 >
// heap: { b0 -> INT_TAG; b1 -> 2; b2 -> INT_TAG; b3 -> 3 }
// We can now proceed to perform the addition:
// stack: < b4 >
// heap: { b0 -> INT_TAG; b1 -> 2; b2 -> INT_TAG; b3 -> 3
//         b4 -> INT_TAG; b5 -> 5 }
vm.pop();
// stack: empty
// heap: { b0 -> INT_TAG; b1 -> 2; b2 -> INT_TAG; b3 -> 3
//         b4 -> INT_TAG; b5 -> 5 }
```

One of your tasks is to implement the nine member functions of `VirtualMachine` class. To facilitate that, you will also want to add to the `Stack` class and the `Heap` class. The `<<` operator has been overloaded for all three classes, so that you can directly print them out with `std::cout` or `std::cerr` to facilitate debugging. Please remember to remove all the debugging output before you submit your assignment.

Once you finish writing these functions, you will implement the garbage collector. If your implementation of `VirtualMachine` class is correct, then setting the heap capacity to be large enough should always execute correctly. However, if we shrink the heap to a certain extent, it may run out of memory cells due unclaimed garbage. You may choose to add a new `GarbageCollector` class that handles all garbage collection, or you can integrate garbage collection directly into the `Heap` class. Read section 2.5 for more information on how the collection algorithm can be implemented.

2.4 Helper Functions

The following four member functions of `VirtualMachine` are required so that I can grade your submission:

- `getStackSize()`: Returns the number of elements currently stored on the stack.
- `getHeapSize()`: Returns the number of memory cells currently stored on the heap.
- `getStackReference(i)`: Returns a reference to the object pointed to by the i-th element on the stack. The bottom stack element has the index 0 and the top stack element has the index `getStackSize() - 1`. If i is greater than or equal to the current stack size, a `StackException` should be thrown.
- `gc()`: Force the virtual machine to perform garbage collection on the heap immediately, regardless of whether the heap has run out of space or not. Read section 2.5 for more information on how the collector can be implemented.

The implementation of the first three functions are given in the skeleton codes. You only need to fill in the implementation of the last function.

2.5 Garbage Collection Algorithm

We will write what is called a stop-the-world, single-generation, compacting collector. If you are interested in other types of garbage collectors, you can search online.

The heap is just a vector of memory cells. Any allocation is done by assigning the next two or three unused memory cells to a newly created object, if there are enough free cells left on the heap.

Of all the nine instructions supported by the virtual machine, only three of them try to heap-allocate new object: `pushInt(n)`, `pushPair()`, and `add()`. The virtual machine needs to check whether there are enough cells on the heap for the new objects. If not,

the garbage collection procedure should be triggered. If we are still short of memory cells, then an `OutOfMemoryException` should be thrown.

When the garbage collector starts, it should scan the entire stack and get the set of objects that are pointed to from the stack, which we call the *root set*. The collector finds all objects that are transitively *reachable* from the stack:

- For a reachable integer object, no other objects are reachable from it.
- For a reachable pair object, both the objects pointed to are reachable from it.

After the set of reachable cells is discovered, we move and compact them consecutively on the heap. The easiest way to do this is to allocate a new heap, copy the reachable cells from the old heap to the new heap, and discard the old heap[4].

You should be careful about how you implement the copy procedure: **Make sure that objects on the new heap can only reference other objects on the new heap.** For example, suppose we start with the following heap:

```
{ a0 -> INT_TAG; a1 -> 1;
  a2 -> PAIR_TAG; a3 -> a0; a4 -> a2 }
```

The new heap should look like this:

```
{ b0 -> INT_TAG; b1 -> 1;
  b2 -> PAIR_TAG; b3 -> b0; b4 -> b2 }
```

In particular, cell b_3 and b_4 should reference b_0 and b_2 respectively, not a_0 and a_2.

3 Submission and Grading

- You will submit a single file `assignment3.tar.gz` on Canvas containing all of the files from the assignment, except for `main.cpp`.
- Make sure your final submission compiles without any external dependencies other than the C++ standard libraries. I will compile it using the included Makefile with the command `make`. If your code does not compile, you will receive zero points for this assignment.
- I will automate the grading process by running your program against several test cases (including the example in this PDF). Make sure the output of your program conforms to the format described in the description section. You should write several test cases yourself and check that your program outputs the correct solution.

[4]Of course, this is a simplification of the compaction problem. Real-world machines cannot create a new heap. Instead they partition the heap into two 'semi-spaces', and reserve one semi-space for the purpose of move-and-compact only.

- In particular, I will use my own make file that creates several test cases and makes sure your implementation is good. You should use the main file to create your own test cases.
- For the grading script to work, do not change any function signature or remove any existing source files, classes or methods from the skeleton project. You may add extra files, classes, or methods, as long as you modify the Makefile accordingly.
- A small portion of your grade is style, so please make sure you comment your file and you adhere to good coding practices. In particular, you should be consistent throughout your code.
- If you have any questions about the assignment, please ask on Piazza.