```
BLG223E - Data Structures                                    16.11.2018
2018-2019 Fall, Bonus Homework
Submission Deadline: 07.12.2018 23:59
```

Tea Runtime Environment

Overview

In this homework you are going to write a simple runtime environment for a made-up language.

Problem

Software engineers at Cake Co. decided to write a simple language named tea. They spent lovely autumn nights without a sleep discussing the rules of the language. And eventually they have compromised. The compromise was that the language would be easy to write, so they set up simple rules. Tea programs would consist of subsequent lines of statements. Each statement would do a very simple job such as incrementing a variable.

What now? They need someone to write a runtime environment to run tea programs. You are hired by Cake Co. to make this. Your runtime environment is supposed to execute tea statements sequentially. But some statements cause jumps to other statements. So you need to be careful.

Definition

1 Program

a) A program written in tea language defines a tea program.

b) A tea program consists of subsequent lines of statements. There are 7 kinds of different statements. Those are *inc*, *dec*, *mul*, *div*, *function*, *call* and *return*.

2 Functions

c) These subsequent statements are so well organised that they constitute a sequence of function definitions. A function definition starts with *function* statement, and end with *return* statement. Between these statements, there may be other statements **except** *function* and *return*.

d) For a valid tea program, one function must *always* be defined with the label **main**. Because execution starts at function *main* in tea. But a program **does not have to** start off with the definition of function *main* at its first line.

3 Variables

Engineers at Cake Co. are interesting people. They hated local variable declarations. So, they did not allow custom variable declarations in their fresh language. Instead, they decided every function has access to 5 predefined variables of their **own**.

e) Whenever a function starts being executed, it has access to 5 predefined variables **initially zeroed** at hand. Those variables are: *a*, *b*, *c*, *d* and *e*. A function may use all of or some of these variables by accessing to and operating on them.

f) Those five variables are **specific to** a function.

g) Even if a function does not use one of *a*, *b*, *c*, *d* or *e*, that variable is still available. Think it of as a ready-to-use variable, staying zeroed.

4 Call Convention

Engineers at Cake Co. like single parameter functions much. So, they decided every function **other than** *main*, be single parameter function in their language. When you define a function in tea, it implicitly becomes defined with one parameter. In addition, **every** function returns value of a variable in tea.

h) Functions can call other functions. By the time caller calls callee, the caller passes one of its 5 variables *a*, *b*, *c*, *d*, *e* as **the argument** to the callee.

i) By the time a callee returns back to the caller, it returns value of one of its 5 variables *a*, *b*, *c*, *d* or *e*, to the caller as **the return value**.

j) Every function **also** *main*, returns value of a variable, ie. either value of *a*, *b*, *c*, *d* or *e*.

Table 1: An example tea program

Line	*Program*	*Value of a,b,c,d,e*	*Program's CPP-like counterpart*
0	`function main`		`int main( ) {`
1	`inc a 2`	2 0 0 0 0	`int a,b,c,d,e = 0;`
2	`inc b 3`	2 3 0 0 0	`a += 2;`
3	`call foo a`	2 0 0 0 0	`b += 3;`
4	`call bar b`	0 3 0 0 0	`a = foo(a); //because foo returns 'a' at 11th`
5	`inc d a`	3 3 4 3 0	`c = bar(b); //because bar returns 'c' at 15th`
6	`inc d c`	3 3 4 7 0	`d += a;`
7	`return d`	3 3 4 7 0	`d += c;`
8	`function foo`		`return d; }`
9	`inc a 4`	6 0 0 0 0	`int foo( int x ) {`
10	`div a 2`	3 0 0 0 0	`int a,b,c,d,e = 0;`
11	`return a`	3 3 0 0 0	`a = x; //because foo is called with 'a' at 3rd`
12	`function bar`		`a += 4;`
13	`inc c 7`	0 3 7 0 0	`a /= 2;`
14	`dec c b`	0 3 4 0 0	`return a; }`
15	`return c`	3 3 4 0 0	`int bar( int x ) {`
			`int a,b,c,d,e = 0;`
			`b = x; //because bar is called with 'b' at 4th`
			`c += 7;`
			`c -= b;`
			`return c; }`

5 Runtime

k) You will write a runtime environment. Your runtime runs given tea programs such as one at the second column of Table 1. **Your goal** is to print to the screen the return value of *main*. For example, the value of variable *d* of *main* at line 7th.

l) Value of variable *d* is 7 by the time *main* returns. Value of *d* is highlighted in red colour at that time. Thus, our runtime must **print** 7 to the screen.

6 Statements

Each statement has a type and one or two arguments. Also, each statement is associated with an address. We call address of a statement *StatementAddress*, type of a statement *StatementType*, and argument of a statement *StatementArgument*.

- *StatementType* is a string and it is equal to either one of “inc”, “dec”, “mul”, “div”, “function”, “call” or “return”.
- *StatementArgument* sometimes is a *variableName*, a *functionName* or a *scalarValue*.
  - *VariableName* is a string and it is either one of “a”, “b”, “c”, “d” or “e”.
  - *FunctionName* is a string, it is not equal to *the-empty-string* or it contains *white-spaces*.
  - *ScalarValue* is an integer value in range [-1773000, 1773000].
- *StatementAddress* is zero indexed line number of where a statement resides in tea file, ie. Zero-indexed line numbers are addresses for statements. See the first column of Table 1.

Table 2: Detailed definition of statements

Statement	*Definition*	*Example*
inc arg1 arg2	Increments arg1 by amount of arg2. arg1 is a *VariableName* arg2 is a *VariableName* or *ScalarValue*.	inc a 2 → a = a + 2 inc a b → a = a + b
dec arg1 arg2	Decrements arg1 by amount of arg2. arg1 is a *VariableName* arg2 is a *VariableName* or *ScalarValue*.	dec a 2 → a = a - 2 dec a b → a = a - b
mul arg1 arg2	Multiplies arg1 by amount of arg2. arg1 is a *VariableName* arg2 is a *VariableName* or *ScalarValue*.	mul a 2 → a = a * 2 mul a b → a = a * b
div arg1 arg2	Divides arg1 by amount of arg2. arg1 is a *VariableName* arg2 is a *VariableName* or *ScalarValue*.	div a 2 → a = a / 2 div a b → a = a / b
call arg1 arg2	Calls arg1 with argument arg2. Pushes contents of caller to *the callStack* such that we can remember the very same contents when callee comes to return back to the caller. Zeroes *a*,*b*,*c*,*d*,*e* except arg2 for callee's use. It jumps to and executes the next statement after where arg1 is defined. arg1 is a *FunctionName* arg2 is a *VariableName*	call foo a→ 1. Push contents of caller to *callStack.* 2. Zero *a*, *b*, *c*, *d*, *e* except *a*. 3. Jump to the **next** statement after where ***foo* is defined**.
return arg1	Returns arg1. When callee returned arg1, it jumps back to the next statement after where callee is called. Pops contents of caller from the *callStack* to remember contents of caller. arg1 is a Variable*Name*	return b → 1. Pop contents of the caller from *callStack*. 2. Jump to the **next** statement after where **callee is called** previously.

		3. Restore *a*, *b*, *c*, *d*, *e* except *b* by using contents you popped from *the callStack.*
function arg1	Defines a function with label arg1. arg1 is a *FunctionName*	function hello

Implementation

Engineers of tea language noticed that you are struggling with writing tea runtime. They came together and wrote some code for you, their welcome gift to you for joining them at Cake Co. Only thing **you need to do** is

- to implement template *Stack* in *runtime.h*
- and to implement one function named *executeStatement* in *runtime.cpp*
- *runtime.h* and *runtime.cpp* will be **given to you** alongside other source files.

1 Implement Stack in runtime.h

```
template<typename T> class Stack;
```

This is a stack. You can implement it however you like. You will use your stack in implementing **`executeStatement`** function. Note that one of its parameters is your stack, `Stack`.

2 Implement executeStatement in runtime.cpp

```
AddressValue executeStatement( const Statement & statement,
                               const AddressValue currentAddress,
                               ScalarValue variables [5],
                               Stack<ScalarValue> & callStack,
                               const AddressValue addressOfCalleeIfCall,
                               bool & isDone );
```

This function returns **address of the next statement** (think of the first column on the left of Table 1) your runtime is supposed to execute. You will be knowing:

- `const Statement & statement is the current statement being executed at a time. You can ask for statement.type, statement.arg1 and statement.arg2 by using it.`
- `const AddressValue currentAddress` is the current address your runtime executes at. Reasonably it is the address of `statement`.
- `ScalarValue variables [5]` is for variables *a*, *b*, *c*, *d*, *e*.
- `Stack<ScalarValue> & callStack` is the call stack. Use it to remember contents of caller function when you need to return back from callee function. What contents you need to put into `callStack` is up to you.
- `const AddressValue addressOfCalleeIfCall` When `statement` is a *call* statement, `addressOfCalleeIfCall` has the address of callee, ie. `statement.arg1` Otherwise it will always equal to zero.
- `bool & isDone` has always value of false, but you can set it to true. Once you set it to true,

your runtime environment halts. Possibly you would like to do this by the time you find out the return value of the main function of your tea program and get it printed onto the screen, which is your goal.

Evaluation

You will be given 3 files: *runtime.cpp*, *runtime.h* and *runtime.a*

*Y*ou will only modify 2 of them, *runtime.cpp and* runtime.h

Your homework will be evaluated in terms of your implementation. Your program should be compilable on ITU's Linux Server by the following command. Yes, you are allowed to use C++11 features in this homework if you wish!

```
>g++ -std=c++0x -Wall -Wextra -Werror runtime.cpp -L. -l:runtime.a -o runtime
```

Table 3: An example how we run your program

```
>ls
easy.tea  runtime.a  runtime.cpp  runtime.h
>cat easy.tea
function main
return a
>g++ -std=c++0x -Wall -Wextra -Werror runtime.cpp -L. -l:runtime.a -o runtime
>./runtime easy.tea
0
```

m) You will **not** be given tea programs with recursive calls in any way. For example, *foo* calls *foo*, *bar* calls *bar*, or *foo* calls *bar* then *bar* calls *foo*; etc.

n) There will **not** be duplicate function definitions, ie. function definitions with the same label. For example with respect to the example at Table 1 above, in a tea program there will be only one *main* function, or one *foo* function, or one *bar* function etc.

o) You are not allowed to include **any STL container**, such as **std::stack<T>, std::vector<T> or others**. If you use it in your implementation, your code will not be graded even if **it compiles** or **produces correct output**. You need to implement your own stack and use that stack as your *callStack*. You are free to consult to course slides for help about stacks.

p) Your homework will be evaluated by using **black-box techniques**.

Policy

- You may discuss the problem addressed by the homework at an abstract level with your classmates, but you should not share or copy code from your classmates or from the Internet. You should submit your own, individual homework.
- Academic dishonesty including but not limited to cheating, plagiarism, collaboration is unacceptable and subject to disciplinary actions.

Submission

- Please submit your files through Ninova e-Learning System.
- You must **only** submit runtime.*cpp* and *runtime.h*. Do **not** send us runtime.*a* or anything else.

- All your implementation must be in C++, and we must be able to compile and run it on ITU’s Linux Server (you can access it through SSH).
- For Windows users: If you wish, you can use WinSCP to upload and edit your source code into ITU SSH Server, and use PuTTY to compile and run your algorithm. If does not, please make sure that your code is able to **be compiled** and runned on ITU’s Linux Server.
- Your code **has to be compiled successfully**. Otherwise you will not be graded. Which will result a grade of **zero** for the homework.
- You should be aware that the Ninova e-Learning System clock may not be synchronized with your computer, watch, or cell phone. Do not e-mail the teaching assistant or the instructors about your submission after the Ninova site is closed for submission. If you have submitted to Ninova once and still want changes in your report, you should do this before the Ninova submission system closes. Your changes will not be accepted by e-mail. Connectivity problems in the last minute about the internet or about Ninova are not valid excuses for being unable to submit. You should not risk your project by leaving its submission to the last minute. After uploading to Ninova, make sure that your homework appears there.

START EARLY.

If a point is not clear, discuss it or e-mail kcengiz@itu.edu.tr

Examples

1 Example 1

```
 0 function main
 1 inc a 2
 2 inc b 3
 3 call foo a
 4 call bar b
 5 inc d a
 6 inc d c
 7 return d
 8 function foo
 9 inc a 4
10 div a 2
11 return a
12 function bar
13 inc c 7
14 dec c b
15 return c
```

Executed line	Next line **after** execution	Address of Callee if Statement is "Call"	*a,b,c,d,e* **after** execution	*callStack* **after** execution
1	2	0	2 0 0 0 0	[]
2	3	0	2 3 0 0 0	[]
3	9	8	2 0 0 0 0	[2 3 0 0 0 3]
9	10	0	6 0 0 0 0	[2 3 0 0 0 3]
10	11	0	3 0 0 0 0	[2 3 0 0 0 3]
11	4	0	3 3 0 0 0	[]
4	13	12	0 3 0 0 0	[3 3 0 0 0 11]
13	14	0	0 3 7 0 0	[3 3 0 0 0 11]
14	15	0	0 3 4 0 0	[3 3 0 0 0 11]
15	5	0	3 3 4 0 0	[]
5	6	0	3 3 4 3 0	[]
6	7	0	3 3 4 7 0	[]
7		0	3 3 4 **7** 0	[]

2 Example 2

```
0  function allOperations
1  inc a -5
2  dec b 5
3  mul a b
4  div a c
5  call hello a
6  call world b
7  inc a b
8  return a
9  function hello
10 mul a 7
11 inc b 6
12 div b 3
13 div a b
14 call world a
15 return b
16 function world
17 inc b 6
18 mul b b
19 inc a b
20 inc b a
21 return b
22 function main
23 inc c 5
24 call allOperations c
25 inc b a
26 return b
```

Executed line	Next line **after** execution	Address of Callee if Statement is “Call”	*a,b,c,d,e* **after** execution	*Element count in callStack* **after** execution
23	24	0	0 0 5 0 0	0
24	1	0	0 0 5 0 0	6
1	2	0	-5 0 5 0 0	6
2	3	0	-5 -5 5 0 0	6
3	4	0	25 -5 5 0 0	6
4	5	0	5 -5 5 0 0	6
5	10	9	5 0 0 0 0	12
10	11	0	35 0 0 0 0	12
11	12	0	35 6 0 0 0	12
12	13	0	35 2 0 0 0	12
13	14	0	17 2 0 0 0	12
14	17	16	17 0 0 0 0	18
17	18	0	17 6 0 0 0	18
18	19	0	17 36 0 0 0	18
19	20	0	53 36 0 0 0	18
20	21	0	53 89 0 0 0	18
21	15	0	17 89 0 0 0	12
15	6	0	5 89 5 0 0	6
6	17	16	0 89 0 0 0	12
17	18	0	0 95 0 0 0	12
18	19	0	0 9025 0 0 0	12
19	20	0	9025 9025 0 0 0	12
20	21	0	9025 18050 0 0 0	12
21	7	0	5 18050 5 0 0	6

7 8 25	8 25	0 0 0	18055 18050 5 0 0 18055 0 5 0 0 18055 **18055** 5 0 0	6 0 0