## Data Class, Stack: Run-time Stack Simulation

The purpose of this assignment is to demonstrate your ability to design and implement a simple data class, and to manage data records on a simple contiguous stack.

This project requires implementing a vastly simplified simulation of a run-time stack environment used to manage function calls for an executing process. When a function is called, an activation record is created and pushed on a stack. When the function call terminates, the corresponding activation record is popped from the stack, and execution returns to the calling function, whose activation record is now at the top of the stack.

The activation record contains all the information necessary to manage the call; in a real system this is quite complex; we will simplify considerably. For our purposes, an activation record will store the name of the called function (somewhat artificial but useful for debugging), the values of the parameters passed to the function, and the values of any local variables declared within the function. To simplify the implementation, we will impose a limit of 5 parameters, and 5 local variables, and further specify that all parameters and local variables must be of type `int`. The activation record may also contain any other reasonable values that you find useful.

In a real system, the activation record would have a number of additional components. For example, the address of the instruction to be executed when the call terminates must be stored, as well as any value returned by the function. And, of course, there are no restrictions on types of parameters or local variables, but all must have storage space allocated within the activation record. For now, we will ignore those complications.

Also, in a real system, each executing process will have a dedicated stack; i.e., the PCB would logically contain the process's stack. For now, we will only simulate the run-time stack for a single process.

There are two major elements to this project. First, you must implement a representation of an activation record; this is required to be a C++ class, with appropriate data and function members. Second, you must implement a stack to hold the activation records. The physical base for the stack must be an array (we limit ourselves to having no more than 10 activation records at any time). The stack must implement the stack ADT interface given in the course notes. Whether you implement the stack as a class is up to you (but that will be required in the next project).

On startup, your program will initialize the necessary data structures then read commands from a script file and log the results of processing those commands to an output file.

## Script file:

This project will involve only one input file, containing the commands that are to be processed. The script file, named `"RTStackScript.txt"`, contains a sequence of commands, one per line. Each command is to be processed as soon as it is read, so you will not need to read and store all of the commands. There will be a number of commands, described below:

**call**<tab><fn name><tab><# of parameters><tab><values of parameters><tab>
<# of locals><tab><values of locals><newline>

Create an activation record for the function and push it onto the run-time stack. If the run-time stack is full, then terminate the simulation with an error message. (That won't actually be tested in this project.) Log a message recording the initiation of the call. See the sample log for formatting.

**return**<tab><newline>

Pop an activation record off of the run-time stack. The stack will never be empty when this command is encountered. Log a message recording the termination of the function. See the sample log for formatting.

**set**<tab><position><tab><value><newline>

Set the value of the specified local variable to the given value. The position corresponds to the order in which the local variables are stored; they are numbered starting at one. If there is no such local variable, then log an error message (see sample log), and clear the run-time stack.

**clear**<tab><newline>
Pop all elements off the run-time stack. Log a confirmation message.

**display**<tab><newline>
Log the contents of the run-time stack, beginning with the top record. For each activation record, log the function name, the values of any parameters, and the values of any local variables. See the sample log for formatting.

**exit**<tab><newline>
Immediately stop reading the script and exit the program.

Legend:

| | |
|---|---|
| `fn name` | a character string, containing no tabs, with no more than 20 characters |
| `# of parameters` | a non-negative integer value |
| `# of locals` | a non-negative integer value |
| `values of parameters` | a tab-separated list of the specified number of integer values |
| `values of locals` | a tab-separated list of the specified number of integer values |
| `position` | a positive integer value possibly identifying a local variable |
| `value` | an integer value |

As a general rule, every command you process should result in a logged message, either confirming the command was carried out or indicating that an error occurred. There is no limit on the number of commands that may be given. Your program should be designed to stop when an input failure is reached. Here is a sample command file:

```
; Runtime Stack Project Script
;
call     main  0     3     1     2     3
call     Func01      1     42    2     17    3
display
;
call     Func02      0     0
call     Func03      0     0
;
display
;
return
display
return
;
set      2     8
display
;
set      5     23
display
;
exit
```

Since the execution of a process will always b gin with a call to `main()`, the first call in a script will always be to `main`.

The script file is guaranteed to conform to the specified syntax. However, it is always a good idea to design for invalid entries, such as an invalid command string.

## Log file:

Your program will write all of its output to a log file, named "`RTStackLog.txt`". Note that it is important to conform to the log file formatting specified in this document.

The log file will begin with some identification lines specifying the name of the programmer and the assignment. Then, for each scripted command, the log file will echo the complete command and then the resulting output. As always, be careful to match the fixed text shown below (e.g., "`Call to Func01 initiated.`").

The logged output for each command must be delimited for easy reading, as shown. Each echoed command must have a descriptive label, as shown.

For a start command, log that the process was added or that it was not, as shown in the sample log. For a kill command, log that the process was found and removed, or that it was not found, as shown. For a ps command, log the shown information for the relevant process(es). For a switch command, log which process has been moved to the run state, as shown. Finally, for the exit command, just log that the program is exiting, as shown.

Here is a sample log file:

```
Programmer: Bill McQuain
CS 1704:    Run-time Stack Simulation

Command: call  main  0     3     1     2     3
Call to main initiated.
--------------------------------------------------------------------------------
Command: call  Func01      1     42    2     17    3
Call to Func01 initiated.
--------------------------------------------------------------------------------
Command: display
Func01               42      17       3
main                  1       2       3
--------------------------------------------------------------------------------
Command: call  Func02      0     0
Call to Func02 initiated.
--------------------------------------------------------------------------------
Command: call  Func03      0     0
Call to Func03 initiated.
--------------------------------------------------------------------------------
Command: display
Func03
Func02
Func01               42      17       3
main                  1       2       3
--------------------------------------------------------------------------------
Command: return
Call to Func03 terminated.
--------------------------------------------------------------------------------
Command: display
Func02
Func01               42      17       3
main                  1       2       3
--------------------------------------------------------------------------------
Command: return
Call to Func02 terminated.
--------------------------------------------------------------------------------
Command: set   2     8
Local variable 2 set in function Func01
--------------------------------------------------------------------------------
Command: display
```

```
Func01                42      17       8
main                   1       2       3
--------------------------------------------------------------------------
Command: set   5     23
Access violation on local variable 5 in function Func01
Terminating process execution.
--------------------------------------------------------------------------
Command: display
Stack is empty.
--------------------------------------------------------------------------
Command: exit
Exiting script execution.
--------------------------------------------------------------------------
```

## Evaluation:

Do not waste submissions to the Curator in testing your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that are provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 100, based upon the runtime testing performed by the Curator System. We will also be evaluating your submission of this program for documentation style and a few good coding practices. This will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

As stated in the course policies, the submission that receives the highest score will be evaluated. If two or more of your submissions are tied for the highest score, then the earliest of those submissions WILL be evaluated. Therefore, DO NOT make undocumented, incomplete submissions to the Curator and then complain that you didn't want those to be evaluated. I will grant NO exceptions.

Your implementation must meet the following requirements:

- Choose descriptive identifiers when you declare a variable or constant. Avoid choosing identifiers that are entirely lower-case.
- Use named constants instead of literal constants when the constant has logical significance.
- Use C++ streams for input and output, not C-style constructs.
- Use C++ `string` variables to hold character data, not C-style character pointers or arrays.
- You must make appropriate use of functions. You should have a function for servicing each of the commands. You should have a function that initializes the data arrays to sensible starting values. There are certainly other good candidates for functions. Aside from some testing functions, my solution uses seven procedural functions besides `main()` and the member functions of classes.
- None of your functions can include more than 30 executable statements. An executable statement is one that causes something to happen. Comments, constant and variable declarations (even if they initialize) do not count as executable statements.
- When you pass an array to one of your functions, use pass by constant reference unless pass by reference is logically necessary. (Remember that for array parameters, pass by reference is the default.)
- You must NOT use dynamic allocation anywhere in your solution.
- But, it is OK (but not required) to use pointers for communication purposes.

You must implement the activation record type as a C++ class. The class should have appropriate data and function members. Try to provide a useful class interface, but also practice good information hiding. The stack may be implemented as a class, or in purely procedural code. If it is a class, you should again practice good information hiding.

Since the implementation of the stack is a major element, the use of the Standard Library stack type is forbidden. Likewise, you may not implement the stack as a template.

Read the *Programming Standards* page on the CS 1704 website for general guidelines. You should comment your code in some detail. In particular:

- You should have a header comment identifying yourself, and describing what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every major block of code should have a comment describing its purpose.
- Every function implementation must have a header comment. The format of the header comment is described in the course notes, as well as on the *Programming Standards* page on the course website.
- Every class declaration must have a header comment, describing the purpose of the class and anything a client would need to know about the limitations of the class.
- Adopt a consistent indentation style and stick to it.

Understand that the list of requirements here is not a complete repetition of the *Programming Standards* page on the course website. It is possible that requirements listed there will be applied, even if they are not listed here.

## Submitting your program:

You will submit this assignment to the Curator System (read the *Student Guid*e), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guid*e.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* and submission link can be found at: http://www.cs.vt.edu/curator/

## Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the course website in the header comment for your program.

**Failure to include the pledge statement may result in a substantial grade penalty.**