COSC 3319 Lab #2 Spring 2017 Burris

From: Ima Your Boss
To: Love Cutting Code

Due: Thursday March 8th. Early submissions are encouraged. It is recommended you at least solve the lab in your head prior to the first test even if you do not complete the physical implementation. Read the hints at the bottom of the lab prior to starting all grading options.

An important part of the lab is to verify that you have implemented it correctly. I suggest you calculate by hand the correct answers for the first few overflows and verify your program results. I will also verify your results. If you are off by even a space, I will return the entire lab to you for corrections and deduct an appropriate penalty form your final grade. You must state on the first page of the lab which grading option you have completed. If you fail to state a grading option, the lab will be treated as a "C" option, i.e., I will not look for the additional features required for the "B" and "A" options while grading. In industry, you will typically work on more than one project at a time. Feel free to interleave work between labs in this class as well as other classes. Now is a good time to master multitasking with COSC 3319 labs.

Allocate the memory dynamically in your program as a single array in the ***system stack*** prior to processing data for the runtime stacks. You may not use heap memory for this storage space allocation! Your program should divide the memory equally between the stacks prior to processing any transactions.

"C" Option (best grade is a 75):
Implement algorithms Reallocate and MoveStack from the hymnal in Ada, C++, or Java to repack memory when overflow occurs out of stack I. *The desire is for your implementation to be oriented towards real time systems with hard real time constraints hence C++ and Ada are preferred. You must use a single contiguously allocated array for the stack space allocated at runtime in the system stack! All space used by your implementation must be allocated at runtime in the stack. You may not use heap memory. You may not use "Collections, Lists, vectors" or other features in languages like Ruby, Groovy, Python, PHP, C++ and other languages which are interpretive in nature*. I will consider use of additional languages but you must ask me individually for approval. Algorithms Reallocate and MoveStack should be written in a manner that will allow you to utilize them in future programming assignments, i.e., as functions or procedures. **You may not use global variables to communicate information between your main program and subprograms!** Limited use of global variables is occasionally justified in OOP "class" definitions. *However, if global variables are used improperly to communicate information within a class, you will not receive credit for the lab*! Your program should prompt the user for the total amount of memory (M) to allocate for all stacks combined, L_0, and the

number of stacks (N). Use memory locations $L_0 + 1$ through $L_0 + M$, e.g., if $L_0 =$ 100 and M = 200, you will actually store information in positions 101 through 200. Your program should divide the memory equally between the stacks prior to processing any transactions. When overflow occurs, assume that 13% of the available memory is to be divided equally between the stacks and that 87% of the available memory is to be divided based on growth. No penalty is to be suffered by stacks that occupy less space when overflow occurs than when the previous overflow occurred (they receive the equal allocation). **Print the contents of each transaction as you process it**. When overflow occurs, print the contents of BASE[J], TOP[J], and OLDTOP[J] before repacking for 1 <= j <= N. Print the contents of BASE[J] and TOP[J] after memory has been repacked. ***You must print the contents of memory when overflow occurs properly labeled with addresses (subscript locations prior to executing algorithm Reallocate) as well as after repacking is complete (after executing algorithm MoveStack)***. Clearly indicate the address of the base and top of each stack in your output. An error message should be printed and processing terminated when the available amount of memory free for distribution falls below 5% of the total memory allocation (from $L_0 + 1$ thru M) or all transactions have been processed. Print an appropriate message if an attempt is made to pop a stack that is empty but continue to process transactions.

Submit a printed copy of the code and all program inputs and outputs. Do not kill trees needlessly!

The most appropriate data type for the “names” below is string. You will learn the most by using string variables. You may however make the names an enumeration type if desired and utilize Ada’s ability to create I/O routines for programmer defined enumeration data types. In general, if you declare a string to be of length 10, Ada expects 10 characters for input. Variable length string libraries in the public domain are available. You may obtain and use one of these libraries if desired as long as you include it as part of your source and properly document its use in the code. VStrings.ada at T:\CSC\DSB\ada\Ada\StringHandling is such a library.

“C/B” data:

(The notation I<digit> <datum> means insert in stack <digit> the value <datum>; D<digit> means pop the top of stack <digit>). Print values as they are popped from the stack.

Allocate 4 stacks at runtime in an array with subscripts from -11 through 60. Use memory locations $L_0 := 4$ and M := 27 for the 4 stacks. Your stacks will physically occupy locations 4 through 24. Assume locations -11 through 4 and 28 through 60 are in use by other portions of your software. You must prompt the user at run time for the lower and upper bounds of the array (-11, 60), L_0 and M. <u>All space must be allocated dynamically at runtime in the system stack! You may not use heap memory in this program</u>.

I4 Zhou, I4 Wei, I1 Burris, I2 Zhou, I2 Shashidhar, I3 Deering, I2 An, I2 Deering, I3 Lester, I1 Yang, I3 Smith, I2 Wei, I2 Zhou, I2 Arcos, D2, I1 Wei, I2 Rabieh, D1, D1, I2 Song, I2 Cho, D3, I2 Varol, I3 Karabiyik, I1 Cooper, I1 Smith, I1 McGuire , I3 Najar, I2 An, I1 Zhou, D2, I2 Deering, I1 Burris, I2 Cho, I2 McGuire, I3 Hope, I3 Pray, I3 NoHope

"B" Option (maximum grade is 85):
Implement the "C" option. Minimize the amount of main memory utilized as overhead for algorithms Reallocate and MoveStack as described in the handout. I will specifically verify that you use only a single physical array to hold the contents of OLDTOP, GROWTH, and NEWBASE, i.e., OldTop[J] = Growth[J-1] = NewBase[J] for 1 <= J <= (N+1)! Clearly indicate in your code with a magic marker where the memory optimizations have been made. *Algorithms Reallocate and MoveStack must be implemented as a package (or class)*. You may add additional functionality (methods) if desired. Process the "C" option data.

"A" Option (maximum grade is 94):
Algorithms Reallocate and MoveStack must be implemented as generics with the highest degree of flexibility possible. Implement the "B" option requirements as part of the solution. You must implement the "B" option to receive credit for the "A" option. Process the C option data first. Now process the "A" option data below.

Use 3 stacks with L_0 = 4 and M = 13 (use memory locations 5 through M = 15) in an array with locations 0 through 50. You are only using a portion of the space in the array. This would be the kind of programming you would be expected to perform on the shuttle, fighter aircraft, and fire control centers on navel vessels. Month and date should be programmer defined types as defined below.

```
type MonthName is (January, February, March, April, May June, July, August,
        September, October, November, December);

type Data is record
        Month: MonthName;
        Day:   Integer range 1..31;
        Year:  Integer range 1400..2020;
end record;
```

I2 January 15 1956; I2 February 14, 1957; I3 September 16, 1946;

I2 September 17, 1842; I2 April 1, 2015; I1 December 24, 1996, D1, I3 March 16, 1992;

D1; I2 January 15, 1956; I3 April 4, 1492; I3 November 7, 1776;

I3 June 12, 1994; I2 July 4, 1776; I2 January 15, 2012; I3 December 6, 1991;

I3 March 5, 1886; I1 October 24, 1996; I1 November 23, 1996; I1 November 2, 1990;
I3 September 14, 1998

"A+" Option (maximum grade is 100):

Management would really be impressed if the user has the ability to specify the subscript (index) used to access the data space. If you implement this option, mark it with a colored pen and brag on yourself so that I do not overlook it while grading. Allocate memory space with L_0 = ‘a’, M = ‘n’, and terminate processing if the amount of available memory drops below 7% using the “C” option data set with 4 stacks. You are really using array locations b, c, d, e, f, g, h, I, j, k, l, m, and n. In Ada Pos(‘M’) – Pos(‘A’) is the number of locations between A and M. Note the “succ(‘a’)” is ‘b’ and the “pred(‘b’)” is ‘a’.

Management would be further impresses if you use class definitions allowing for inheritance. This is not however required to receive the maximum possible grade.

There is more pride in this part of the lab than points. How much pride do you have?

Hints:

Hint 1: Many languages have a library floor and ceiling operator, normally buried in a math function library that must be referenced for the compiler (with, import, include). They may be under a related name such as round and truncate or lower bound and upper bound. A good programmer however should not need a library, indeed the challenge is to write your own. Consider the following:

```
function floor(x: float) return integer is
        temp: integer;
begin
        temp := integer(x); -- force conversion to integer
        if( float(temp) <= x)
                return temp;
        else
                return temp – 1;
end;
```

Many library routines are very short sections of code.

Hint 2: There is more than one way to read and write strings in Ada. In most instances if you declare a string variable to be of length 10 (e.g., str: array(1..10) of character), Ada expects 10 characters in the input every time you execute “get(str).” You can use a publicly available variable length string library such as VString to ease reading and writing variable length strings. Look on the “T” drive for VString. When there are only a limited number of possible strings, such as in the lab for names, you can declare them as a programmer defined enumeration type and read / write them using I/O routines written by the compiler using generics. For example:

```
type NameType is (Joe, Betty, Sam, Bob, Mary);
package NameType_IO is new Ada.Text_IO.Enumeration_IO(NameType);
use NameType_IO;

N1: NameType;
get(N1);
put(N1);  -- "put" defaults to printing uppercase.  A simple parameter can fix this.
              -- In the generic instantiation for enumeration types:
          -- put(To: out String; Item: out Enum; Set: in Type_Set := Default_Setting);
          -- where Default_Setting: Type_Set := Upper Case;
```

A similar trick may be accomplished in C++ and Java by declaring the names to be an "enum" type then overloading the "<<" and ">>" operators in the stream I/O library. Unfortunately, C++ and Java make you write the code. Tisk, tisk, what did you expect.

Hint 3: Do not forget the hint in the first lab, i.e.,

```
Lab2 << FileIn >> Fileout
```

When this command is entered at the command line, all input is taken from FileIn and all output is redirected to FileOut. This technique can save you considerable effort during debugging.

Finally, I will calculate the results by hand. If your program has an error I will simply return it to you for correction but with no indication of what caused the error. You are a professional working for me! I expect professional performance. In part that means you will not submit software that does not meet the specification. I certainly do not expect to have to check my employee's work for accuracy. You are paid to be accurate (in this case your grade is dependent on your accuracy).