CSCI-1200 Data Structures — Fall 2017
Homework 3 — GPS Tracking & Stack Hacking

In this assignment we will explore the use of pointer arithmetic, allocation of single value and array variables on the stack, passing arguments by value vs. by reference vs. by pointer, and the C calling convention. Please have your notes from Lecture 5 available for this homework.

Part 1: Working with GPS Tracking Coordinates

In this part of the problem you will write code using C-style arrays and pointers to process GPS coordinate data recorded by cell phone or smart watch running apps. We provide stater code for part 1 in the file named `part1.cpp`. You should not modify any of the provided code, just add the missing pieces.

Make a simple new class named `GPSdata` that contains three integers: the position coordinates x and y (measured in feet), and the current speed s (measured in feet per minute). The only constructor for this class is a default constructor that sets everything to zero. Write a `set_position` function that takes in two integer arguments. The speed will be calculated later. Note: Because this class is so tiny, we won't separate the declaration from the implementation from the main function. Keep everything in a single file. But your class should have private member variables and the necessary accessor and modifier functions.

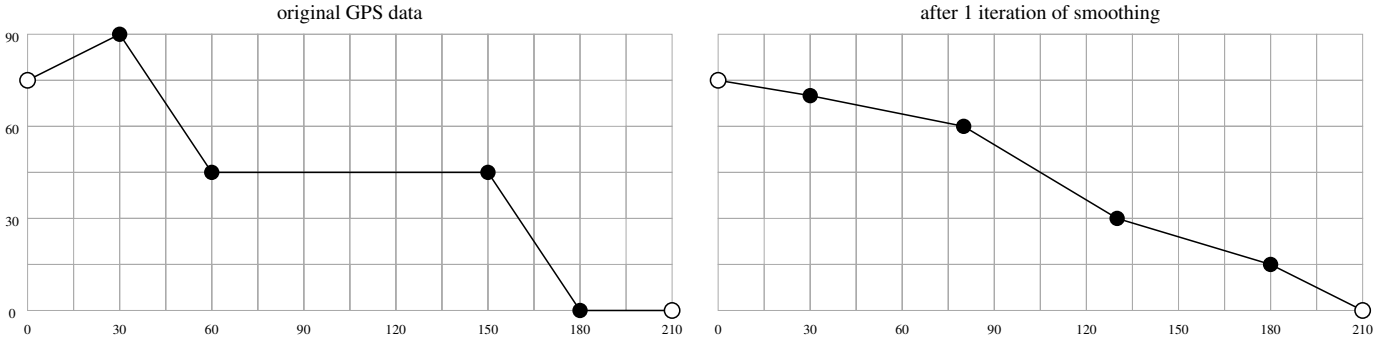
Then, write a non-member function named `distance` that takes as arguments a C-style array of `GPSdata` (*not* an STL `vector`) named `data`, an integer `n`, and a pass-by-reference integer parameter `avg_feet_per_minute`. This function calculates and stores with each GPS sample the per-sample speed (in feet per minute) and the average speed over the whole path. The function also calculates and returns the length of the path connecting these data points. The speeds and path lengths are rounded down to the nearest integer. Note that the input files we are working with contain one GPS coordinate sample every 4.25 seconds.

GPS data can be noisy, and the calculated path length likely overestimates the true distance. So let's write a function to named `filter` to *average or smooth* this data as shown in the diagram below. The function takes in three arguments, two C-style arrays of `GPSdata` named `input` and `output`, and an integer `n`. The function should process the data to smooth out noisy zigzags in the GPS path data. The end points of the output path are set to be the same as the endpoints of the input path. But each interior output path point should be the average of three input points:

$$\text{output}_i = \frac{\text{input}_{i-1} + \text{input}_i + \text{input}_{i+1}}{3}$$

The `filter` function should return the percentage change in the total path length:

$$\text{percentage change} = 100 * \frac{\text{input path length} - \text{output path length}}{\text{input path length}}$$



We provide the `recursive_filter` function which repeatedly calls your `filter` function until the percentage change in the path length drops below a target threshold. If you specify a very small percentage change target, the final path will be an approximately straight line between the two original endpoints. (Because we are rounding the GPS coordinates to the closest integer, the path won't be perfectly straight.)

To finish the code for Part 1, you'll have to write implement the printing helper functions. Then you should be all set to test and debug the code. We provide a few input data files of GPS coordinates. You'll call the program with 1, 2, or 3 arguments. The first is the name of the input file. The second (optional) parameter is the number of coordinate points from the file to process – by default it will process all of the points. And the third (optional) parameter is the percentage change stopping criteria threshold for recursive filtering.

Make sure your output is debugged before moving on to the next parts.

Part 2: Diagramming Memory

What will happen if your `distance` or `filter` functions are incorrectly used and the length of the array(s) are not the same as n? What will happen if n is too small? If n is too big? What if the `filter` *input* array is bigger than the `filter` *output* array? Or vice versa? How might the order that the variables were declared in the `main` function impact the situation?

Following the conventions introduced in Lecture 5, draw paper & pencil diagrams of the variables, arrays, and pointers on the stack for the *correct* usage of these functions and at least two different *incorrect* use cases. Please be neat! Scan these drawings or take a picture of these drawings with your phone. You will upload these digital images with your homework, along with the well-written writeup of your expectations in the your `README.txt`. Complete this portion of the homework before moving on to Part 3. Note: You will not be graded on the correctness of your predictions, but rather on your thought process and clarity of your diagrams and discussion.

Part 3: Poking around in the Stack

Now let's print out the contents of memory and see what's going on. We have provided the `TheStack` class (in `the_stack.h` and `the_stack.cpp`) to help visualize the memory on the stack as we allocate local variables and call functions. Study the sample usage of this class in the provided file `part3.cpp`.

First, compile and run this program on your computer using `g++` or `clang++`. We recommend the following options:

```
g++ -Wall -m32 -O0 -g -o part3_example.out the_stack.cpp part3_example.cpp
```

`-O0` requests no compiler optimizations. This is important so that the compiler doesn't simplify away some of variables or functions that it deems unnecessary or useless. You're welcome to try with optimization (`-O3`), which may or may not produce a condensed visualization. `-m32` requests a 32-bit executable. (Yes! you can compile and run 32-bit programs even if you have a 64-bit machine). We strongly recommend using this flag for this homework because the 32-bit compiler & program usage of the stack is simpler and more predictable.

Now run the program and study the output. You should see 2 stack frames for the 2 functions (main & helper). You should see the local variables in each function. Because the compiler is allowed flexibility to re-order the local variables, their order may not match the code.

Note that the stack on x86 architectures is in descending order. You can see the elements of the array (which are required to be *continuous*). The first element of the array is required to be stored in the smallest memory address in the block of memory allocated for the array, so it looks upside down. We won't attempt to explain or understand every memory location on the stack. The extra space between the variables is due to temporary variables or padding inserted by the compiler to improve alignment. This extra space may be labeled as "garbage or float?" or it might contain old data values or addresses that appear to be legal and useful. The `TheStack` class does not attempt to visualize or display floating point values or strings, etc.

You should see the helper function arguments/parameters pushed on the stack just before the helper function frame. Note: `clang++` may make and use copies of the arguments. But you should still see the parameter values just before the return address. Here is sample output from GNU/Linux Ubuntu gcc/g++ 5.4:

```
size of intptr_t: 4
localvar address: 0xff94f070
x address: 0xff94f004
a address: 0xff94f010
y address: 0xff94f008
z address: 0xff94f00c
----------------------------------------
                location: 0xff94f0b0  VALUE:   1
                location: 0xff94f0ac  garbage or float?
                location: 0xff94f0a8  VALUE:   0
                location: 0xff94f0a4  garbage or float?
                location: 0xff94f0a0  garbage or float?
 return address location: 0xff94f09c  CODE: 0xf7403637
  FUNCTION MAIN location: 0xff94f098  VALUE:   0
                location: 0xff94f094  VALUE:   0
                location: 0xff94f090  POINTER: 0xff94f0b0
                location: 0xff94f08c  garbage or float?
                location: 0xff94f088  garbage or float?
                location: 0xff94f084  garbage or float?
                location: 0xff94f080  garbage or float?
                location: 0xff94f07c  garbage or float?
                location: 0xff94f078  VALUE:   8
                location: 0xff94f074  POINTER: 0xff94f07c
       localvar location: 0xff94f070  VALUE:   2017
                location: 0xff94f06c  garbage or float?
                location: 0xff94f068  garbage or float?
                location: 0xff94f064  garbage or float?
                location: 0xff94f060  VALUE:   1
                location: 0xff94f05c  garbage or float?
                location: 0xff94f058  POINTER: 0xff94f074
         param2 location: 0xff94f054  VALUE:   1863
         param1 location: 0xff94f050  VALUE:   1776
 return address location: 0xff94f04c  CODE: 0x804b7e7
FUNCTION HELPER location: 0xff94f048  POINTER: 0xff94f098  ----> FUNCTION MAIN
                location: 0xff94f044  VALUE:   0
                location: 0xff94f040  POINTER: 0xff94f070  ----> localvar
                location: 0xff94f03c  garbage or float?
                location: 0xff94f038  garbage or float?
                location: 0xff94f034  garbage or float?
                location: 0xff94f030  garbage or float?
                location: 0xff94f02c  garbage or float?
                location: 0xff94f028  VALUE:   1
                location: 0xff94f024  POINTER: 0xff94f02c
           a[4] location: 0xff94f020  VALUE:   14
                location: 0xff94f01c  VALUE:   13
                location: 0xff94f018  VALUE:   12
                location: 0xff94f014  VALUE:   11
           a[0] location: 0xff94f010  VALUE:   10
              z location: 0xff94f00c  VALUE:   98
              y location: 0xff94f008  POINTER: 0xff94f004  ----> x
              x location: 0xff94f004  VALUE:   72
                location: 0xff94f000  garbage or float?
                location: 0xff94effc  garbage or float?
                location: 0xff94eff8  POINTER: 0xff94f024
                location: 0xff94eff4  POINTER: 0xff94f00c  ----> z
                location: 0xff94eff0  garbage or float?
----------------------------------------
```

Now let's use the `TheStack` visualization with the code we wrote for Part 1. In order to accommodate 32-bit and 64-bit operating systems, the `TheStack` class uses the type `intptr_t` in place of `int` and all pointers. On a 32 bit OS/compiler, this will be a standard 4 byte integer and on a 64 bit OS/compiler, this will be a 8 byte integer type. Copy your code from Part 1 to a new file named `part3.cpp`. Search and replace all use of `int` to be `intptr_t`.

Now, follow the examples in the `part3_example.cpp` file to label the array named `input` storing the initial GPS data. Print the stack before and after the call to the `distance` function and confirm that you can see the `speed` member variables being set appropriately. Each "slot" of this array stores an instance of the `GPSdata` class. The compiler is required to pack together the member variables for each class instance, and they are required to match the member variable order in the class declaration. What happens if the value you pass in for n does *not* match the actual length of the `input` array?

Similarly, label the `filtered` array on the stack and print the stack before and after the first call to the `filter` function. Confirm that you see the data calculated and stored in the array. Now test your hypotheses from Part 2. What happens if the arrays are not the same length? Discuss your findings in your README.txt. Paste small samples of the stack visualization into your README.txt to support your investigation. Make sure to exaggerate the errors so that memory is misused or clobbered and correct program behavior is disrupted.

NOTE: The `set_label` function expects a memory address of type `intptr_t*`, so you may encounter compiler errors/warnings similar to: "`cannot convert `GPSData*' to `intptr_t*'`" or "`cannot initialize a parameter of type 'intptr_t *' (aka 'long*') with an rvalue of type 'GPSData *`". Normally, you don't want to mix pointers of different types, but for this homework the conversion is simple and safe. To fix these errors/warnings, use an *explicit cast* to override the compiler check:

```
thestack.set_label((intptr_t*)&tmp[0],"tmp[0]");
```

Part 4: Visualizing Pass-by-Value vs. Pass-by-Reference

Next let's look more carefully at the arguments to the distance function. You'll probably want to comment out all of the stack labeling and printing code you added for the previous part. And let's also comment out the calls to filter and recursive filter. Inside of the distance function you can label the distance function stack frame, the associated return address, and the 3 arguments to the function. When you run the code and study the visualization, focus on the difference between the pass-by-value argument `n` and the pass-by-reference argument `avg_feet_per_minute`. Pass-by-reference actually uses a pointer so that we can modify the original local variable in calling function! (It'll be helpful to label that variable too!) Save this code as `part4.cpp` and paste the relevant portion of the output in your README.txt along with any additional observations you made.

Part 5: Visualizing a Recursive Function

Finally, let's visualize the stack frames generated by the execution of a recursive function. Remove or comment out the stack visualization for the previous part(s), and re-enable the call to the `recursive_filter` function. Add code at the top of the `recursive_filter` function to label the start of each stack frame, the associated return address, and the local `tmp` array that stores the intermediate results of each call to `filter`. Now when you run this code with `input1.txt` you should see multiple stack frames labeled for the same function. And when you adjust the `percentage_change_threshold` you should see more or fewer stack frames for that function, as appropriate. Also note that the *return address* pointer to the code that made each call is the *same* for all of the `recursive_filter` function frames.

Save this code as `part5.cpp`, and paste a small portion of the stack frame visualization into your README.txt, along with your final thoughts.