Part 1

1) After implementing the "Hello World" program in 1.c, I ran ls -l on the executable, and the overall size is 16696 bytes. Running size on 1.out showed me that the text size is 1568 bytes, the data segment size is 600 bytes, and the bss is 8 bytes.
2) The overall size of 2.out is 16728 bytes. When I ran size on 2.out after declaring a global array, the text and data segments were the same as 1.out, but 2.out's bss is 4032 bytes.
3) The overall size of 3.out is 20744 bytes. The text segment size is 1568 bytes, the data segment size is 4616 bytes, and the bss segment is 8 bytes.
4) The overall size of 4.out is 16744 bytes. After running size 4.out on the executable, the text segment size is 1772 bytes, the data segment size is 608 bytes, and the bss is 8 bytes. Data is not defined locally in a function stored in the executable, but it matters whether or not it is initialized.
5) When 5.c was compiled for debugging and optimization, the debugging file size was larger than the optimization executable (19320 bytes for 5d.out v. 16744 bytes for 5o.out). However, each of the segment sizes for both executables were the same.

Part 2

1) After running stack_hack_1.out, the stack's top was near 0x7ffe939a7264.
2) After creating a few variables and running stack_hack_2.out, the data segment was near the memory address 94785325178896 in decimal, the text segment was near 140729248614264, and the heap within the data segment was near 140729248614272.
3) When I created a test function with local arrays, the stack top was near the memory address 140728416773716 in decimal.

Part 3

1)

*No local variables
*a(1)
*No prev frame
* return to line 5
%int i
%If i>0, a –i
%main frame
% return to line 5 if I is not 0, else return to line 14
^no local variables
^printf statement
^function a frame
^no return

The stack is made of stack frames for each function call: main (*), a (%), and printf (^). Main sees the function a with an argument 1, so the stack moves down and main goes to line 5. The function a returns

to itself until i = 0, in which case the function goes to the printf stack frame. The print statement is then return back through a to the main function, where it is returned through main's a function call.

2) I compiled a.out for debugging, then placed breakpoints at each function call. As I ran through each breakpoint, entering info frame to see if my stack frame by hand was correct, I first was in the main stack frame, at the highest memory address of 140737488349520 in decimal. Then, the program went to the function a's frame, repeating the function until i = 0. When I ran info frame on this function, it was referenced as coming from the main frame, and a's frame memory address was 140737488349504. Then, a new stack frame was made for the print statement, who was referenced as coming from the a frame, and whose memory address was 140737488349472. The print statement was then returned through the a frame to the main frame, where it was executed.