Assignment 4 – Stack frame

Question 1: Stack frame and stack pointer

Consider the code below, and give answers to the following questions.

```
char *qbuf(int n) {
   char buf[16];
   sprintf(buf, "%d", n);
   return buf;
}

int main(...) {
   int exitstatus = 0;
   char *tokens[32];
   int n_tokens = parse(..., tokens, 32, ...);
   ...
   for (i = 0; i < n_tokens; i++)
       if (!strcmp(tokens[i], "$?"))
        tokens[i] = qbuf(exitstatus);
   for (i = 0; i < n_tokens; i++)
       printf("%s\n", tokens[i]);
}
```

Please explain in terms of the stack frame layout and stack pointer location when calling `qbuf()` and `printf()`.

a). **Is it guaranteed to operate properly?** I.e. given a line "echo $? X Y" will it print out the lines "echo", "0", "X", and "Y"?

b). **Is it likely/unlikely to crash in printf?** Note that it would only crash if a bad pointer was passed to printf - it won't crash if it points to the wrong data.

Question 2 x86-64 assembly and stack frame

We have talked about registers and assembly in class. Below are some details. You need to understand them to answer questions.

A. Registers (64-bits):

```
%rip: contain the address of the next instruction to run
%rsp: point to the top of the stack
%rax: a general-purposed register, also used for holding return value
%rdi: a general-purposed register, also used for holding the first argument
```

B. Basic x86-64 assembly instructions:

(1) `movq PLACE1, PLACE2`

means "move 64-bit quantity from PLACE1 to PLACE2". the places are usually registers or memory addresses, and can also be immediates (constants).

(2) `pushq %rax`

Push the content of %rax (which is 8B) to the stack. This is equivalent to two instructions:

```
subq $8, %rsp       // subtract %rsp by 8
movq %rax, (%rsp)   // move the content of %rax to
                    //   the memory pointed by %rsp
```

(3) `popq %rax`

Pop the 8B content on top of the stack and copy it to %rax. This is equivalent to two instructions:

```
movq (%rsp), %rax   // move memory content pointed by %rsp
                    //   to %rax
addq $8, %rsp       // %rsp = %rsp + 8
```

(4) call foo

Invoke function foo ("foo" is a function pointer) [what is a function pointer? Functions are part of the code, which locates in memory. A function pointer is a pointer to the first instruction of the function in memory.] This is equivalent to two instructions:

```
pushq %rip        // push %rip to the stack
movq foo, %rip    // jump to the first instruction of foo
```

(5) ret

Return from a function. This is equivalent to:

```
popq %rip   // pop the top of the stack and copy it to the %rip,
            // meaning the next instruction to run will be the pointer
            // on top of the stack
```

C. Questions:

(a). Consider an initial state of a stack as follows: (Each slot is 8bytes, indicated by "[8B]")

```
high
    |  ... [8B] | <- %rsp
    |  [=?]      |
    |  [=??]     |
    |            |
low
```

What will the stack look like after running these four instructions:

```
movq $36, %rax
pushq %rax
movq $99, %rax
pushq %rax
```

Draw the stack state like above with %rsp and contents in each slot
(what are in "[=?]" and "[=??]").

Continue with the above stack, and we run the following instructions: (A–G are addresses of instructions. You will need them when "pushq %rip")

```
A:  movq $50, %rdi
B:  call foo
C:  movq %rax, (%rsp)
     ...

  foo:
D:  subq $8, %rsp
E:  movq %rdi, (%rsp)
F:  popq %rax
G:  ret
```

(b). **What is the stack state after finishing instruction "E" and before running "F"? Draw the stack state below.**

(c). **What's the stack state when finishing instruction "C"? Draw the stack state below.** (note: you should include %rsp and everything on stack.)