CS-354 Midterm (Spring '17 @ Epic)

Bugs, Bugs, Everywhere, and Not A Debugger To Invoke

Please Read All Questions Carefully!

There are nineteen (19) total numbered pages.

Please put your FULL NAME on THIS page only.

Name: Answers McSheet

This exam is about bugs. When you write C code (or, x86 assembly), you create a lot of them! (well, sometimes). And so, to be a really good programmer, you have to learn how to identify them quickly and accurately.

Most questions introduce some code (C, or x86 assembly) and ask you to identify if the code is buggy or correct, and to explain why. That's it! Well, not quite; sometimes some other related questions are asked.

IMPORTANT: For ALL questions, assume we are running on a **32-bit** x86 processor. Addresses are 32 bits (4 bytes) long. Integers are also 4 bytes in size.

Each of the 16 numbered questions is worth the same number of points. Thus, do the questions you think are easiest first, so you don't run out of time, and maximize your score.

Score[1]: **Perfect***

[1]The score is filled in by us, not you, alas

* unless you find a problem

1. **Getting The Right Size**

   This code tries to allocate room for some memory of varying sizes using `malloc()`. The first thing written down is the programmer's intent (e.g., "one character" means the programmer wishes to allocate room for one character). The second thing is the code the programmer wrote to try to match the intention.

   (a) **Put an X** through code that you think is definitely buggy.

   (b) **Circle** code that will work (i.e., not crash or lead to undefined behavior) but you think is poor style or otherwise problematic.

   (c) Leave correct code alone (do not circle or X it).

   In all cases, **EXPLAIN YOUR ANSWER.**

   ASSUME FOR ALL OF THESE THAT MALLOC SUCCEEDS (i.e., will return a valid pointer to memory of the size requested).

   (a) One character, pointed to by pointer-to-character `cp`:

   ```
   char* cp = malloc(sizeof(char*));
   ```

   sizeof(char)

   works because 4 bytes alloc'd (1 asked for)

   (b) One character, pointed to by pointer-to-character `cp`:

   ```
   char *cp = malloc(sizeof(char));
   ```

   correct (matches spec)

   (c) One character, pointed to by pointer-to-character `cp`:

   ```
   char* cp = malloc(1);
   ```

   sizeof(char)

   OK as long as sizeof(char) is 1; poor form

   (d) One integer, pointed to by pointer-to-integer `ip`:

   ```
   char* cp = (char*) malloc(sizeof(int));
   int* ip = (int*) cp;
   ```

   → (int*) malloc (sizeof(int));

   works but verbose, hard to read

(e) One integer, pointed to by pointer-to-integer ip:

```
int* ip = malloc(sizeof(int*));
```

int

works because sizeof(int) equals sizeof(int*)

(f) One integer, pointed to by pointer-to-integer ip:

```
int* ip = malloc(sizeof(char*));
```

int

works (again, same size)

(g) One integer, pointed to by pointer-to-integer ip:

```
int ip = malloc(sizeof(int*));
```

int

as above

(h) One integer, pointed to by pointer-to-integer ip: int

```
int i = (int) malloc(sizeof(int*));
int *ip = (int *) i;
```

verbose

but works again

(i) Ten integers, pointed to by pointer-to-integer ip:

```
int* ip = malloc(10);
```

← X not 0

only 10 bytes, needs 40

bug! (too small)

(j) Ten integers, pointed to by pointer-to-integer ip:

```
int* ip = malloc(10 * sizeof(ip));
```

int

actually works (OK if you said it shouldn't)

(k) Ten integers, pointed to by pointer-to-integer `ip`:
```
int* ip = malloc(10 * sizeof(int));
```

correct

exactly what was asked for

(l) Ten integers, pointed to by pointer-to-integer `ip`:
```
int* ip = malloc(10 + sizeof(int));
```

* not +

too small (broken)

(m) Ten integers, pointed to by pointer-to-integer `ip`:
```
int* ip = malloc(40);
```

10 * sizeof(int)

works when sizeof(int) is 4 (as it is here)

(n) Ten integers, pointed to by pointer-to-integer `ip`:
```
int* ip = malloc(44);
```

(as above)

too big but works

(o) Ten integers, pointed to by pointer-to-integer `ip`:
```
int* ip = malloc(44); ip++;
```

as above

dangerous, seems like it might leak memory, but works

2. **Remembering Memory Chunks.**

   (a) **Put an X** through code that you think is definitely buggy.

   (b) **Circle** code that will work (i.e., not crash or lead to undefined behavior) but you think is poor style or otherwise problematic.

   (c) Leave correct code alone (do not circle or X it).

   In all cases, **EXPLAIN YOUR ANSWER.**

   ASSUME FOR ALL OF THESE THAT MALLOC SUCCEEDS (i.e., will return a valid pointer to memory of the size requested).

   (a)
```
int* p = malloc(sizeof(int));
p = 0;
```

   setting p to zero (NULL) is a memory leak as previously alloc'd memory is lost

   (b)
```
int* p = malloc(sizeof(int));
*p = 0;
```

   correct
   → alloc's int
   → sets it to zero

   (c)
```
int* p = malloc(sizeof(char));
p = 0;
```

   weird, likely too small

   → memory leak

   (d)
```
int* p = malloc(sizeof(char));
*p = 0;
```

   can't write an int (4 bytes) when only 1 byte alloc'd

(e)
```
int* p = malloc(sizeof(int));
p = 0;
*p = 0;
```
leak

null ptr deref (likely crash)

(f)
```
int* p = malloc(sizeof(int));
*p = 0;
free(p);
*p = 0;
```
write to free'd memory

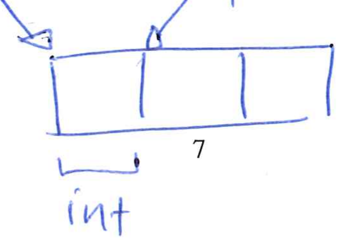
(g)
```
int* p = malloc(sizeof(int));
*p = 0;
free(p);
p = 0;
```
correct (ok to set p to NULL after freeing)

(h)
```
int* p = malloc(sizeof(int));
p = malloc(2 * sizeof(int));
*p = 0;
p++;
*p = 0;
```
leak

works if 2 ints are alloc'd (but not 1)

(i)
```
int* p = malloc(3 * sizeof(int));
*p = 0;
p++;
*p = 0;
free(p);
```
allocated

p (can't free from middle)



3. **Computing Some Expressions**

In this question, we have a different type of bug: assume the C compiler you are using doesn't quite have the right rules of precedence implemented. Specifically, **all arithmetic operations are at the same level of precedence** (with left-to-right associativity).

Your task: write the following expressions in C assuming these broken precedence rules. **Important: Use the minimal number of parentheses in doing so.**

(a) Adding integers x and y, dividing the sum by 2:

x + y / 2

(b) Dividing y by 2 and adding x to it, then dividing whole thing by z:

y / 2 + x / z

(c) Multiplying x and y, multiplying a and b, and then summing these two products:

x * y + (a * b)

(d) Computing $2 \times a^2$:

2 * a * a

(e) Your thoughts: Does changing the precedence rules (as above) make C programming harder, easier, or not much difference?

Probably harder as it doesn't match algebra (many answers accepted)

4. **Factorial**

The following code is meant to compute a **factorial**. The factorial of N, sometimes written as $N!$, is the product of all positive integers less than or equal to N. For example, $4! = 4 \times 3 \times 2 \times 1 = 24$. Also, by definition, $0!$ is 1. Trying to compute the factorial of a negative number is considered undefined (i.e., don't worry about it).

```
int factorial(int n) {

    int r = 0;

    for (i = 0;   i < n;   i++)

        r = r * n;

    return r;
}
```

Correct the code above (if needed) to work as desired.

5. **Double Swap**

The following code is meant to swap the value of two doubles. For example, if there are two doubles x and y, calling swap_double(&x,&y) should result in x now containing the previous value of y, and y the previous value of x.

```
void swap_double(double* a, double* b) {

    double   tmp   =   a;

    a   =  b;

    b   =   tmp;

}
```

Correct the code above (if needed) to work as desired.

6. **Fizz Buzz**

Some programming interviews ask for this very simple programming test:
Write a program that prints the numbers from 1 to 100 (inclusive), one per line. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Here are a bunch of attempts at writing such code. For each:

(a) **Put an X** through parts of the code that you think are definitely buggy.

(b) Leave correct code alone (do not X it).

In all cases, **EXPLAIN YOUR ANSWER.**

accidentally forgot this throughout

7. Attempt #1:

```
int i;
for (i = 1; i <= 100; i++) {
    if ((i / 3 == 0) && (i / 5 == 0)) {
        printf("FizzBuzz\n");
    } else if (i / 3 == 0) {
        printf("Fizz\n");
    } else if (i / 5 == 0) {
        printf("Buzz\n");
    } else {
        printf("%d\n", i);
    }
}
```

should be % not /

8. Attempt #2:

```
int i;
for (i = 1; i <= 100; i++) {
    if ((i % 3 == 0) || (i % 5 == 0)) {
        printf("FizzBuzz\n");
    } else if (i % 3 == 0) {
        printf("Fizz\n");
    } else if (i % 5 == 0) {
        printf("Buzz\n");
    } else {
        printf("%d\n", i);
    }
}
```

and not or

9. Attempt #3:

```
int i, p;
for (i = 1; i <= 100; i++) {
    p = 0;
    if (i % 3 == 0) {
        printf("Fizz"); p = 1;
    }
    if (i % 5 == 0) {
        printf("Buzz"); p = 1;
    }
    if (p == 0) printf("%d", i);
    printf("\n");
}
```

correct

prints Fizz or Buzz or both w/o newline OR prints num w/o newline then newline

10. Attempt #4:

```
int i;
for (i = 1; i <= 100; i++) {
    if (i % 15 == 0) {
        printf("FizzBuzz\n");
    } else if (i % 3 == 0) {
        printf("Fizz\n");
    } else if (i % 5 == 0) {
        printf("Buzz\n");
    } else {
        printf("%d\n", i);
    }
}
```

correct

takes %3 and %5 as special case of %15 == 0

11. Attempt #5:

→ 1 → <=

```
int i;
for (i = 0; i < 100; i++) {
    if (i % 15 == 0)
        printf("FizzBuzz\n");
    } else if (i % 3 == 0) {
        printf("Fizz\n");
    } else if (i % 5 == 0) {
        printf("Buzz\n");
    } else {
        printf("%d\n", i);
    }
}
```

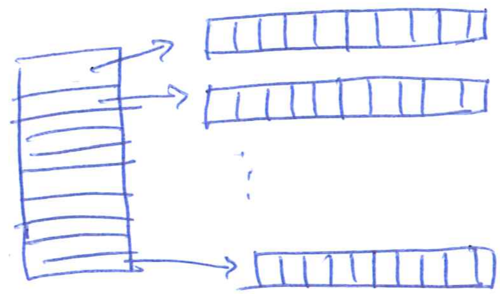
sometimes you just gotta stare at all the characters

12. **Matrix Fun**

One programmer decides to create a two-dimensional array of integers as follows, of size 10×10:

```
int array[10][10];
```

Another programmer does the following:



```
int** array;
array = malloc(10 * sizeof(int *));
int i;
for (i = 0; i < 10; i++)
    array[i] = malloc(10 * sizeof(int));
```

(a) Is either one of these buggy? If so, why? If not, why not?

No, both allocate 100 ints and can be accessed as desired (array[x][y])

(b) How are these code snippets similar? (explain)

as above

(c) How are these code snippets different? (explain)

1) int array[10][10] => contiguous in memory on stack or global

2) not contiguous — heap

(d) Write code that initializes each value of this two-dimensional array to the product of its indices, i.e., array[i][j] should be set to the product of i and j (that is, to $i \times j$).

```
int i, j;
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        array[i][j] = i*j;
```

13. **Structs**

Structs are used throughout C programs. Here, a programmer is trying to access fields of a struct using various means. The first thing written down is the programmer's intent. The second thing is the code the programmer wrote to try to match the intention.

(a) **Put an X** through code that you think is definitely buggy.

(b) **Circle** code that will work (i.e., not crash or lead to undefined behavior) but you think is poor style or otherwise problematic.

(c) Leave correct code alone (do not circle or X it).

In all cases, **EXPLAIN YOUR ANSWER.**

First, assume you have the following structure definition:

```
struct foo {
    int a;
    int b;
    int c;
};
```

(a) Initialize the elements of a struct foo.

```
struct foo x = {0, 0, 0};
```

✓

(b) Initialize the elements of a struct foo.

```
struct foo x; x.a = 0; x.b = 0; x.b = 0;
```

X

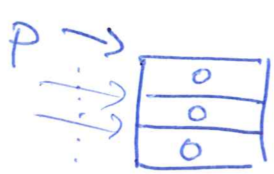
x.c not init'd (sorry)

(c) Initialize the elements of a struct foo.

```
struct foo x; int* p=(int *)&x; *p=0; p++; *p=0; p++; *p=0;
```

works

(perhaps poor style?)



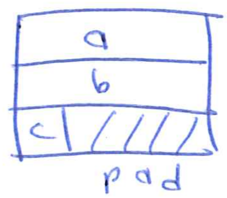
Now assume we have the following, slightly different, structure.

```
struct foo2 {
    int a;
    int b;
    char c;
};
```

We now ask a few questions about this new structure:

(d) Assuming usual packing rules, does this structure occupy more, less, or the same amount of space as `foo`? (explain)

same



(e) Which code sequences from (a), (b), and (c) above also will work with the newly defined structure, and which won't? Explain.

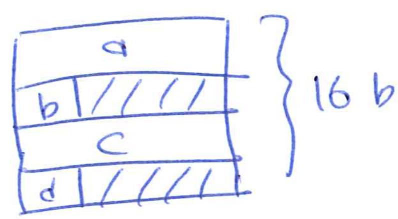
a) works   b) still doesn't

c) kind of (over writes pad)

Finally, assume we have the following definition of a new struct and an array:

```
struct foo3 {
    int a;
    char b;
    int c;
    char d;
};

struct foo3 farray[100];
```

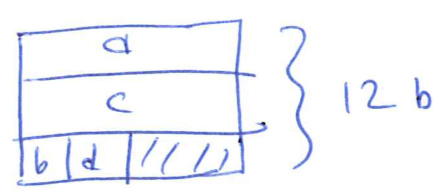


(f) How much space in memory will `farray` use?

100 * 16 => 1600 bytes

(g) How can you rewrite the structure definition to save space?

```
struct foo3 {
    int a, c;
    char b, d;
}
```



14. **Functions and the Stack**

The following functions are reported to be likely candidates for serious bugs. For each, diagnose the problem (i.e., describe why it is a bug). Be concise if you can. Of course, it is possible that the code works fine (i.e., is not buggy). In those cases, just say so.

(a) This code is supposed to be a quick way to allocate space for an integer and return a pointer to that integer:

```
int* ialloc() {
    int newmem;
    return &newmem;
}
```

What is wrong with this code? How would you fix it?

wrong: returns addr of soon-to-be dealloc'd stack memory

correct return malloc(sizeof(int));

(b) This code is supposed to copy n bytes from the source C string to the destination C string.

```
void astrcpy(char* dst, char* src, int n) {
    int i;
    for (i = 0; i < n; i++)
        dst[i] = src[i];
}
```

What is wrong with this code? How would you fix it?

wrong: should stop copying if src encounter \0 (end of string)

correct: stops copy at that point (but makes sure \0 is in dst too)

also ensures that if n reached, dst properly \0 terminated

(c) This routine takes a pointer to a string and fills it with the word "hi". If the string is NULL, it will allocate memory for it.

```
void hifill(char* str) {
    if (str == NULL)
        str = (char *) malloc(3);
    str[0] = 'h';
    str[1] = 'i';
    str[2] = '\n';
}
```

clearly wrong ⇒ \0

What is wrong with this code? How would you fix it?

wrong: as above and cannot change str (copy on stack)

correct: \n ⇒ \0 and char ** str as arg

15. **Reversing x86**

Each of the following questions has an assembly fragment shown first. Unfortunately, the C equivalent (probably buggy anyhow) has been lost. Your job is to **write the C that best matches the assembly**.

(a) Assume the address of variable `i` is in register `%eax`.

```
movl (%eax), %ecx
addl %ecx, %ecx
movl %ecx, (%eax)
```

Write some C to match this assembly code:

i = i + 1;

(b) Assume the address of variable `x` is in `%eax`.

```
movl (%eax), %ecx
cmpl $0x10, %ecx
jg after
addl $10, %ecx
after:
movl %ecx, (%eax)
```

Write some C to match this assembly code:

or 16

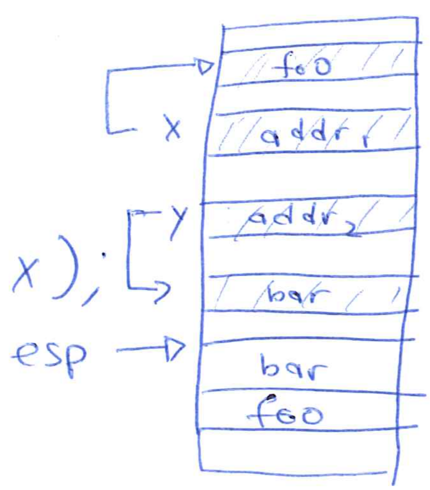
if (x ≤ 0x10)
x = x + 10;

(c) Assume the addresses of variables `x` and `y` are in registers `%ebx` and `%edi`, and that variable `rc`'s address is in `%esi`.

```
movl (%ebx), %eax
pushl (%eax)
movl (%edi), %eax
pushl (%eax)
call compute
movl %eax, (%esi)
```

Write some C to match this assembly code:

rc = compute(*y, *x);



(d) The following assembly, you've been told, defines a function that takes an address to a variable as a parameter.

```
subsome:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
movl (%eax), %ecx
subl $10, %ecx
movl %ecx, (%eax)
movl %ebp, %esp
popl %ebp
ret
```

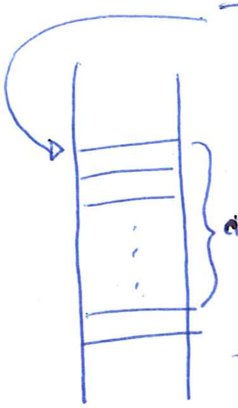
arg on stack

arg treated as pointer

Write some C to define this function:

```
void subsome(int *x) {
    *x = *x - 10;
}
```

(e) The following assembly, you've been told, defines a function that does something to an array.



```
clearit:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
movl 12(%ebp), %ecx
movl $0, %edx
cmpl %edx, %ecx
je alldone
top:
movl %edx, (%eax,%edx,4)
incl %edx
cmpl %edx, %ecx
jg top
alldone:
movl %ebp, %esp
popl %ebp
ret
```

size

Write some C to define this function:

```
void clearit(int *a,
             int size) {
    int i;
    for(i=0; i<size; i++)
        a[i] = i;
}
```

16. **x86 Call/Return**

The x86 cdecl calling convention (as discussed in class) is quite complex. Here are the full set of steps needed:

Step 1. Save "caller-save" registers
Step 2. Push arguments onto the stack (reverse order)
Step 3. Call function
Step 4. Establish new base pointer
Step 5. Make room for local variables
Step 6. Save "callee-save" registers
Step 7. (execute body of function)
Step 8. Restore "callee-save" registers
Step 9. Free stack space that was allocated for locals
Step 10. Restore old base pointer
Step 11. Return from function
Step 12. Deallocate space for arguments
Step 13. Restore "caller-save" registers

However, someone tells you that is it (i.e., not buggy) to sometimes skip some of these steps. Your job here is to answer questions about skipping some of these steps, as described here:

(a) Skip steps 1 and 13. Is this ever OK? When?

can skip (if caller has no live values in those registers)

(b) Skip steps 2 and 12. Is this ever OK? When?

can (if no args to func)

(c) Skip steps 4 and 10. Is this ever OK? When?

can (if no local vars, args used)

(d) Skip steps 5 and 9. Is this ever OK? When?

can (if no locals used)

(e) Skip steps 6 and 8. Is this ever OK? When?

can if no callee-saved regs used in func

(f) All of the above skipped pairs of steps. What would happen in those cases if one of the steps was skipped, but not the other? Could that ever work?

usually broken ⇒
e.g. push w/o pop will grow stack over time in weird way

(g) The **order** of the steps of the convention also matter, in some cases more than others. Specifically, why do we order 1 before 2? Does this order matter?

→ allows stable, non changing reference to args
→ otherwise, would have to save all args

(h) Similarly, why do we order 5 before 6? Does this order matter?

(same really)

(i) Does the order that arguments are pushed onto the stack in step 2 matter? (could we have changed the convention to do it the other way?)

order is just convention
(could do it differently)