

# COS 217 Midterm with Solutions

Fall 2006

Please write your answers clearly in the space provided. For partial credit, show all work. State all assumptions. You have exactly 50 minutes for this exam. This midterm is open book, open notes. Put your name on every page. Write out and sign the Honor Code pledge just before turning in the test. *“I pledge my honor that I have not violated the Honor Code during this examination.”*

Question	Score
1	/30
2	/15
3	/15
4	/40
Total	/100

Name:

Honor Code:

# 1 Short Answer

Answer the following in only the space provided.

1. What is the difference between a dangling pointer and leaked memory?

**SOLUTION:**

A dangling pointer is a pointer pointing to a non-existent data object (e.g., freed memory in the heap segment, a local variable in an invalidated stack region, etc.). Leaked memory is dynamically allocated memory whose reference is lost and cannot be accessed.

**SCORING:**

This is a 4-point question.

Dangling pointer: Any sorts of pointers pointing to invalidated data are accepted.

Memory leak: A concept of lost reference constitutes a correct answer. A common insufficient answer is “Memory which is not freed”, and this gets 1 point deduction.

2. What is the difference between an ADT and a data structure?

**SOLUTION:**

An ADT is a specification of a set of data and the set of operations that can be performed on the data. Such a data type is abstract in the sense that it is independent of various concrete implementations. In C, an opaque pointer is a common choice to realize an ADT.

**SCORING:**

This is a 4-point question.

An explanation containing the basic concept of implementation independence, encapsulation, separation of usage and representation, or hiding/protecting the internals from clients is all right.

3. Give two reasons to have virtual memory?

**SOLUTION:**

Any two of the following reasons:

- To provide each process with a large ( $2^{32}$  byte), contiguous memory
- To provide each process with an image of exclusively protected memory
- To obviate the need for each program to implement its own memory and storage handling routines
- To enhance portability

**SCORING:**

This is a 4-point question.

Simply saying “swapping” is not enough because, i) the technique of swapping is not the purpose, but the implementation, of virtual memory, and ii) a non-virtualized memory system can also employ swapping.

4. Write C code that puts the string “Heap” into the heap segment.

**SOLUTION:**

```
char *pc;  
pc = (char *)malloc(strlen("Heap")+1);  
assert(pc != NULL);  
strcpy(pc, "Heap");
```

**SCORING:**

This is a 4-point question.

Common mistakes are assigning a string literal to the pointer, instead of using `strcpy()`. For example,

```
pc = "Heap";
```

This kind of solutions gets a small partial credit because it reveals that a student possesses some minimal understanding on the connection between dynamic memory and the heap segment. (What is actually happening, however, is making pc point to somewhere in the RODATA segment.)

5. Write C code that puts the string “Stack” into the stack segment.

**SOLUTION:**

```
void f(void) {  
    char pc[] = "Stack";  
}
```

**SCORING:**

This is a 4-point question.

Common mistakes are defining a character pointer (e.g., `char *pc`), not a character array, and assigning a string literal to the pointer. This kind of solutions gets a smallest partial credit only when the statements are residing in a local function. Again, we acknowledge that doing so reveals marginal understanding of the connection between local variables and the stack segment. (What is actually happening, however, is making pc point to somewhere in the RODATA segment.)

6. After the statement `unsigned char x = 0121;`, what is the value of `x` interpreted as an 8-bit signed integer value (expressed in decimal)?

**SOLUTION:**

0121 is an octal constant whose decimal value is  $1 * 8^2 + 2 * 8^1 + 1 * 8^0 = 64 + 16 + 1 = 81$ . One can also convert it to a binary value,  $01010001_2$ . Its decimal value can be also computed as  $1 * 2^6 + 1 * 2^4 + 1 * 2^0 = 64 + 16 + 1 = 81$ .

**SCORING:**

This is a 3-point question.

Some students made simple mis-calculation, such as  $1 * 8^2 + 2 * 8^1 + 1 * 8^0 = 64 + 16 + 8 = 88$ . Small partial credit is given to this case only when the conversion process is explained.

7. After the statement `unsigned char x = 0xF0;`, what is the value of `x` interpreted as an 8-bit signed integer value (expressed in decimal)?

**SOLUTION:**

0xF0 is a hexadecimal constant whose binary representation is  $11110000_2$ . Since the sign bit is 1, this is a negative integer. Therefore, by applying the “invert-and-plus-one” rule, one can obtain  $-(00001111_2 + 1_2) = -(00010000_2) = -16$ .

**SCORING:**

This is a 4-point question.

Simple mis-calculation is given partial credit only when the conversion process is explained.

8. Describe one property a good hash function should have. Why should it have this property?

**SOLUTION:**

Any property that is synonymous with one of the followings:

- Uniform (even, random) distribution over the hashing range
- Similar inputs shouldn't generate similar hash values

Why?

To minimize collisions

**SCORING:**

This is a 3-point question.

## 2 Sign Extension

Prove that sign extending the 3-bit two's-complement number  $a_2a_1a_0$  to the 4 bit two's-complement number  $b_3b_2b_1b_0$  preserves its value.

### SOLUTION:

There are many valid approaches to this problem. Here are two simple ones.

#### Method 1: Algebraic

The value of the two's complement number  $a_2a_1a_0 = -2^2a_2 + 2^1a_1 + 2^0a_0 = -4a_2 + 2a_1 + a_0$ . Likewise, the value of the two's complement number  $b_3b_2b_1b_0 = -2^3b_3 + 2^2b_2 + 2^1b_1 + 2^0b_0 = -8b_3 + 4b_2 + 2b_1 + b_0$ .

The single bit sign extension of the 3-bit two's complement number  $a_2a_1a_0$  is the two's complement number  $a_2a_2a_1a_0$ . Therefore,  $b_3 = a_2$ ,  $b_2 = a_2$ ,  $b_1 = a_1$ , and  $b_0 = a_0$ .

By substitution:

$$b_3b_2b_1b_0 = a_2a_2a_1a_0 = -8a_2 + 4a_2 + 2a_1 + a_0 = -4a_2 + 2a_1 + a_0$$

$$a_2a_1a_0 \text{ also equals } -4a_2 + 2a_1 + a_0 \text{ (see above)}$$

Therefore, the two's complement numbers  $b_3b_2b_1b_0$  and  $a_2a_1a_0$  represent the same value when  $b_3b_2b_1b_0$  is the sign extension of  $a_2a_1a_0$ .

#### Method 2: Exhaustive

Let's enumerate all possible values of  $b_3b_2b_1b_0$ , the sign extension of  $a_2a_1a_0$ .

$a$ value	$a_2a_1a_0$	$b_3b_2b_1b_0$	$a$ value
3	011	0011	3
2	010	0010	2
1	001	0001	1
0	000	0000	0
-1	111	1111	-1
-2	110	1110	-2
-3	101	1101	-3
-4	100	1100	-4

Since the  $a$  value is the same as the  $b$  value for all settings of  $a_2a_1a_0$ , the two's complement numbers  $b_3b_2b_1b_0$  and  $a_2a_1a_0$  represent the same value when  $b_3b_2b_1b_0$  is the sign extension of  $a_2a_1a_0$ .

#### Scoring

5 points for demonstrating an understanding of sign extension.

5 points for demonstrating an understanding of the value of a two's complement number.

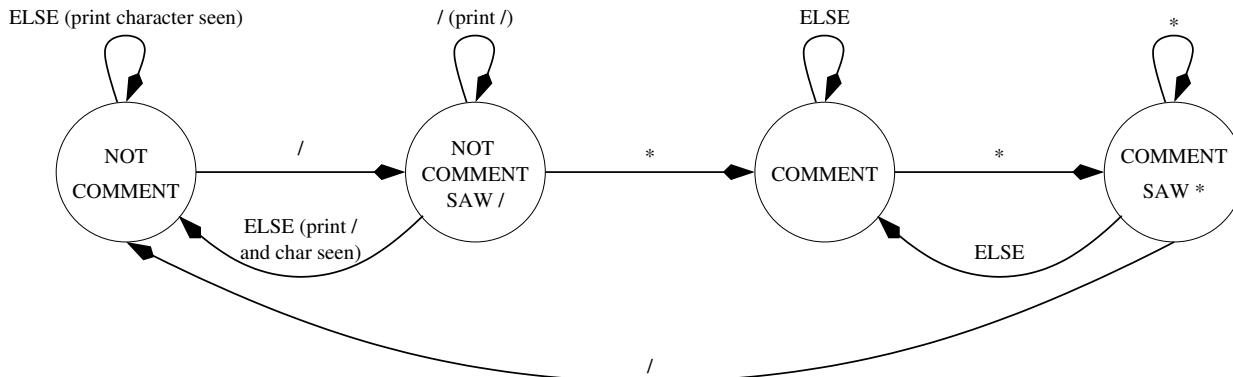
5 points for a correct proof.

### 3 DFA

Draw a DFA with actions to print only the content of string literals found in C code (not in comments and without interpreting control sequences). Please clearly indicate your start state. Things to consider include:

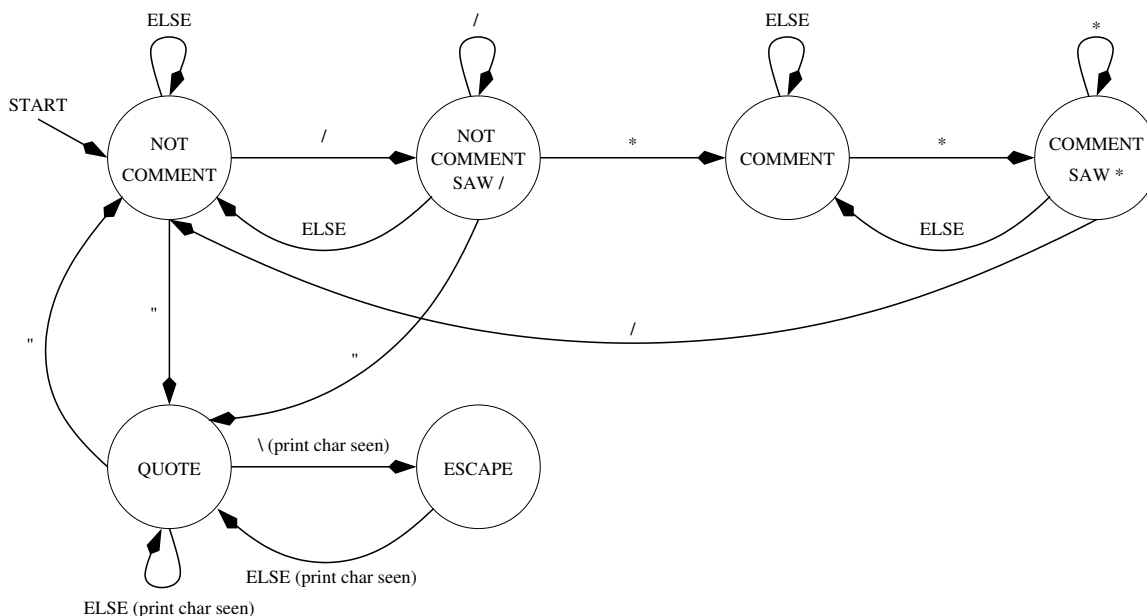
1. While the "-" character may exist inside comments, strings do not exist inside comments. In other words, no part of a comment should be printed.
2. Just as strings cannot start in comments, comments cannot start inside strings.
3. Valid strings may include the control sequence "\\" as part of the string. Do not interpret this control sequence. (Interpreting it would just print a ". Your DFA should print both the \ and " characters in sequence.)
4. Comments start with /\* and may have many /\*'s inside of them.
5. Comments end with \*/ but may not have any \*/'s inside of them.
6. If you find the following figure useful, you may use it as part of a larger DFA which constitutes your answer. Note: you may need to modify, not just add to this figure. This figure is not guaranteed to perform any specific task, and it does not have a start state.

LEGEND:



SOLUTION:

LEGEND:



## 4 Bug Hunt!

The following code on the next page attempts to implement a very simple editor. The pages following that contains many pages for your convenience. Find eight *distinct* bugs (there are more than eight total bugs in the program) using the guide below. Keep in mind that some lines may participate in more than one bug. Please list line numbers when describing each bug. Write your answer to this part on this page only. Note that the code on the next page compiles (with a warning) and executes, so don't worry about finding syntax errors.

1. The compiler (gcc with no arguments other than the C file name) prints “warning: initialization from incompatible pointer type” as its only message due to a bug on one line in the function `main`. What is the line number and what is the problem?

### SOLUTION:

The compiler generates the warning for line 37. This line declares `fn` as an array of 5 pointers to functions taking 3 parameters (of types `char *`, `int`, and `char*`, respectively) and returning an `int`. Additionally, this line initializes the first 4 elements of this array with the addresses of functions `exit` (from the standard C library), `insert_text`, `search`, and `print`. The warning comes from the assignment of the address of function `exit` to the first element of `fn`. The reason is that `exit` has `void` return type, and takes one formal parameter of type `int`.

### Scoring

This is a 6-point question. Answers with the correct line number and with no or incorrect explanation received 3 points.

2. Describe a bug related to variable scope in the `insert_text` function which may lead to a segmentation fault at run time for some inputs.

### SOLUTION:

Function `insert_text` receives the `buf` pointer from `main` and `reallocs` it. This may change the buffer's location in memory but, although the local pointer `buf` in `insert_text` is updated properly, this value is not propagated back to `main`. Therefore `main`'s `buf` pointer will still point to the old location, which is no longer allocated to this buffer.

### Scoring

This is a 6-point question. Many students answered the out-of-bounds access to the `buf` array and got 3 points for that.

3. Describe a bug which may lead to a segmentation fault at run time for some inputs.

### SOLUTION:

Here is partial list of correct answers:

- (a) The code does not check the return value of `malloc` (line 42) and `realloc` (line 8) to make sure they are not `NULL`, i.e. to check that memory was properly allocated.
- (b) The code does not check if the value of variable `command` provided by the user (line 48) is between 0 and 3. This can cause a segmentation fault at line 56.
- (c) The code does not check for buffer overrun when reading the `text` variable (line 53). Inputting a string with more than 99 characters can cause a segmentation fault.
- (d) The call to `realloc` (line 8) does not allocate enough space to store the old content of the string pointed by `buf` and the inserted text. This can cause segmentation fault in lines 12 and 13.
- (e) The code in function `insert_text` does not make sure that the resulting string has the `'0'` at the end. This can lead to segmentation fault when executing lines 24, 25 or 33.
- (f) The call to `printf` in line 33 is unsafe and can result in segmentation fault if `buf` contains formatting sequences.
- (g) Function `print` fails to return a value, so a position out of the allocated buffer may be stored in `pos` (line 56), which can cause segmentation fault later when using `pos` to index the buffer.

### Scoring

These are 6-point questions each. In many cases, partial credit was given for correctly pointing a bug location but failing to fully explain what the bug was.

4. Describe another bug which may lead to a segmentation fault at run time for some inputs.

**SOLUTION:**

Same as the above.

**Scoring**

No credit was given for listing the same instance of a bug repeatedly.

5. Which paragraph of code makes printing and exiting needlessly less convenient for the user?

**SOLUTION:**

The paragraph in lines 50-54 makes printing and exiting needlessly inconvenient for the user because it requires the user to input a string which is not necessary (and not used) for these options.

**Scoring**

This is a 6-point question. Because what is a “paragraph” of code is not a very standard definition, full credits were also given for correct answers that listed the whole loop in lines 44-57.

6. Describe another bug or a serious style problem.

**SOLUTION:**

Here is a partial list of correct answers, besides the bugs listed in the items above:

- (a) Lack of comments in the code.
- (b) Lack of clarity in various parts of the code.
- (c) Use of magic numbers in various places.
- (d) Failure to include `string.h` and `stdio.h`
- (e) Array `fn` is declared with more elements than necessary.
- (f) Variable `ch` should be declared as `int` because it is assigned the result of `getchar`.
- (g) In function `insert_text`, variables used to index potentially large string should be of type `size_t` instead of `int`.
- (h) Line 33 is unsafe; it should be `printf(“%s”, buf);`.
- (i) There are unused arguments in functions `search` and `print`.
- (j) Function `print` is declared to return an `int` but no value is returned.

**Scoring**

These are 5-point questions each. In many cases, partial credit was given for correctly pointing the location of a problem but failing to fully explain what the problem was.

7. Describe another bug or a serious style problem.

**SOLUTION:**

Same as the above.

**Scoring**

No credit was given for listing the same instance of a bug or style problem repeatedly.

```

0  #include <stdlib.h>
1
2  /* Insert text at pos in buf, shifting existing text to make room */
3  int insert_text(char *buf, int pos, char *text) {
4      int size, i;
5
6      /* Allocate memory for the new text */
7      size = strlen(text);
8      buf = (char *)realloc((void *)buf, size);
9
10     /* Insert the text and shift text in buffer to make room */
11     for(i = 0; i < size; i++) {
12         buf[i+pos+size] = buf[i+pos];
13         buf[i+pos] = text[i];
14     }
15
16     return pos + size;
17 }
18
19 /* Find pattern in buf and return its starting position */
20 int search(char *buf, int pos, char *pattern) {
21     char *match;
22
23     /* Search for string pattern, return end position if not found */
24     if(!(match = strstr(buf, pattern)))
25         return strlen(buf);
26
27     return match - buf;
28 }
29
30 int print(char *buf, int pos, char *junk) {
31     /* Print the buffer */
32     printf("The editor's buffer contains:\n");
33     printf(buf);
34 }
35
36 int main(int argc, char *argv[]) {
37     int (*fn[5])(char*, int, char*) = {exit, insert_text, search, print};
38     int command, text_pos, pos = 0;
39     char *buf, ch, text[100];
40
41     /* Create some space for the initial buffer to get things going */
42     buf = (char *) malloc(1);
43
44     while (1) {
45         /* Give menu of commands and get command */
46         printf("0:Exit, 1:Insert at Position %d, ", pos);
47         printf("2:Search for String, 3:Print Buffer \n > ");
48         scanf("%d\n", &command);
49
50         /* Get text for command and execute */
51         text_pos=0;
52         while((ch = getchar()) != '\n')
53             text[text_pos++] = ch;
54         text[text_pos] = '\0';
55
56         pos = (*fn[command])(buf, pos, text);
57     }
58     return 0;
59 }

```

These man pages are included for your convenience.

#### NAME

strstr - locate a substring

#### SYNOPSIS

```
#include <string.h>
```

```
char *strstr(const char *haystack, const char *needle);
```

#### DESCRIPTION

The `strstr()` function finds the first occurrence of the substring `needle` in the string `haystack`. The terminating `'\0'` characters are not compared.

#### RETURN VALUE

The `strstr()` function returns a pointer to the beginning of the substring, or `NULL` if the substring is not found.

#### NAME

exit - cause normal program termination

#### SYNOPSIS

```
#include <stdlib.h>
```

```
void exit(int status);
```

#### DESCRIPTION

The `exit()` function causes normal program termination and the the value of `status & 0377` is returned to the parent (see `wait(2)`). All functions registered with `atexit()` and `on_exit()` are called in the reverse order of their registration, and all open streams are flushed and closed. Files created by `tmpfile()` are removed.

The C standard specifies two defines `EXIT_SUCCESS` and `EXIT_FAILURE` that may be passed to `exit()` to indicate successful or unsuccessful termination, respectively.

#### RETURN VALUE

The `exit()` function does not return.

## NAME

realloc - reallocate dynamic memory

## SYNOPSIS

```
#include <stdlib.h>
```

```
void *realloc(void *ptr, size_t size);
```

## DESCRIPTION

realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

## RETURN VALUE

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails the original block is left untouched - it is not freed or moved.