Princeton University
COS 217: Introduction to Programming Systems
Fall 2008 Midterm Exam Answers

The exam was a 50-minute, open-book, open-notes exam.

Question 1 Part A

TRUE. All terms are true and logical AND yields TRUE.

Common mistake: If you used bitwise AND, the answer was FALSE.

Question 1 Part B

FALSE. This is evaluated first as 10 > x, which yields either a 0 or 1. This result is then compared to 5, so no matter what x is, the answer is FALSE.

Common mistake: If you got the evaluation order wrong and first evaluated x > 5, then you would have gotten TRUE.

Question 1 Part C

TRUE. This is simply 256 - 64, which is 192.

Common mistake: If you used octal instead of hexadecimal, you would have gotten an answer of zero, which is FALSE.

Question 1 Part D

TRUE. Whether the leftmost bits get two zeros or two ones, the answer is still non-zero.

Common mistake: If you said that the answer is undefined, that’s incorrect since we are only looking for true/false.

Question 1 Part E

FALSE. Since 5 < 10 is true, the overall expression takes on the first value after the question mark, 0.

Common mistake: Some people only reported the result of 5 < 10, rather than evaluating the whole expression.

Question 2 Part A

A dangling pointer refers to a particular programming error. When data is freed or otherwise deallocated (return from a function for data on the stack), pointers to data data are now invalid. Dereferencing that pointer is an error. Reading from will likely yield corrupt data, and writing to it can corrupt other program data. Any examples in the book are fine, such as using a pointer after freeing it, or returning the address of a variable allocated on the stack, etc.

Question 2 Part B

10.23.200.8 (with a newline at the end)

Explanation: 0x0a17c808 is a hexadecimal integer literal. >> is rightward bit shift and & is bitwise and. For any integer value v, v & 0xff selects the lowest byte. So 0xffff & 0xff = 0xff, 256 & 0xff = 0, and 255 & 0xff = 255 (255 and 0xff are representations of the same number). (x >> 24) shifts the value to the right 24 bits, resulting in 0x0a.

Selecting the low byte gives 0x0a, which is 10 in decimal. (x >> 16) shifts the value to the right 16 bits, resulting in 0x0a17. Selecting the low byte gives 0x17, which is 23 in decimal. (x >> 8) shifts the value 8 bits, resulting in 0x0a17c8. Selecting the low byte gives 0xc8, which is 200 in decimal. (x >> 0) is a no-op, giving us 0x0a17c808. Selecting the low byte gives 0x08, which is 8 in decimal. Thus the printf prints 10.23.200.8 followed by a newline.

Question 2 Part C

```
(*x)–>y = 5;
```

Note that this question requires reading both the structure declaration as well as the typedef. The typedef on the second line defines blah to be of type: (pointer to (struct blah)). Then x, the parameter of Func, is a pointer to blah, and hence pointer to a pointer: pointer to (pointer to (struct blah)). It must be dereferenced twice to access it: (**x).y = 5; or (*x)–>y = 5; (*x) gives you a pointer to struct blah. (**x) gives you a struct blah (**x).y accesses field y or (*x)–>y gives you a struct blah and access field y.

Question 2 Part D

It fails to compile – you can't dereference a void pointer

The problem here is that a pointer to void cannot be dereferenced. Void pointers cannot be dereferenced, as the type of data they point to is unknown to the compiler. This results in a compile time error.

Ideas for corrected versions:

```
   int *p = malloc(sizeof(int));
   *p = 1;

   char *p = malloc(1);
   *p = 1;

   void *p = malloc(1);
   *(char *)p = 1;
```

None of these are stylistically great, but they are not errors. The fact that the literal 1 has type int is a nonissue. It's the size of the type pointed to that matters (see examples below).

This is fine:
```
   char *p = malloc(1);
   *p = 1;
```

This is a problem:
```
   int *p = malloc(1);
   *p = 1;
```

This also is a problem:
```
   int *p = malloc(1);
   char c = 'a';
   *p = c;
```

Question 2 Part E

The stack is for local variables and formal parameters to functions, and space on the stack is deallocated when a function returns. The heap is for dynamically allocated memory, and space on the heap disappears when the program calls free() or exits.

Other important points:

stack -> function scope, automatic by compiler, frames, non-static, temporary or local, function parameters, declarations, vars removed on function return, code blocks, automatic variables, function calls, return values, return addresses, oldest at bottom, function duration, compile time

heap -> handled manually (in C), dynamic or runtime data, malloc free etc., persist between functions, program termination, unordered, depends on input

general -> grow towards each other

Question 2 Part F

```
int Function(char *buf, int maxLen);
int Function(char **buf, int maxLen);
```

Also, for the second declaration it's fine if the return type is a char* and *bytesUsed is a formal parameter. It's also a good idea to use size_t here instead of int, but you have to use it consistently, and should not mix/match them in a single declaration.

Question 2 Part G

```
Func2();
for (i--; i > 3; i--)
   Func2();
```

Question 3 Part A

1

Question 3 Part B

5

Question 3 Part A

Prints out all elements of vals[]

Question 3 Part A

2 3 5 7 11
In other words, this program finds the prime numbers.

Question 3 Part A

Change the "i++" to "i += 2" since even numbers beyond 2 are not prime.

Question 3 Part A

Add break after setting match to FALSE.

Question 4

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum {MAX_LINE = 99};

typedef struct LineBuf {
   char lb_line[MAX_LINE + 1];
   struct LineBuf *lb_next;
} LineBuf;

static LineBuf *head = NULL;

int main(int argc, char *argv[])
{
   char line[MAX_LINE + 1];
   LineBuf *walk;
   LineBuf *next;

   while (scanf("%s", line) == 1) {
      LineBuf *lb = malloc(sizeof(LineBuf));
      assert(lb != NULL);
      strcpy(lb->lb_line, line);
      lb->lb_next = head;
      head = lb;
   }

   for (walk = head; walk != NULL; walk = walk->lb_next)
      printf("%s\n", walk->lb_line);

   for (walk = head; walk != NULL; walk = next) {
      next = walk->lb_next;
      free(walk);
   }

   return(0);
}
```

1) lb_line and line should be of size MAX_LINE + 1.
2) scanf should use conversion specification "%s", not "%d".
3) Need to malloc sizeof(LineBuf) bytes, not sizeof(LineBuf *) bytes.
4) "lb->lb line = line" should be a strcpy.
5) Missing "head = lb;" at end of while loop.
6) "walk++" in for loop should be "walk = walk->lb_next"
7) Should free memory.