Exam statistics:
Mean: 19.9/25
Median: 20/25
StdDev: 3.7/25

Q1 Instructions and Pledge

1 Point

This exam consists of 5 multi-part questions (plus the pledge), and you have 60 minutes -- budget your time wisely.

This is a closed-book, closed-note exam, and "cheat sheets" are not allowed. During the exam you must not refer to the textbook, course materials, notes, or any information on the Internet. You may not compile or run any code on armlab or any other machine.

You are not allowed to communicate with any other person, whether inside or outside the class. You may not send the exam problems to anyone, nor receive them from anyone, nor communicate any information about the problems or their topics. *If you have technical issues or need to ask a clarifying question about the wording of some problem, please post a* ***private*** *message on Ed.*

You may use blank paper as scratch space, but you must enter your answer in the online system in order to receive credit.

This examination is administered under the Princeton University Honor Code, and by signing the pledge below you promise that you have adhered to the instructions above.

Please type out the Honor Code pledge exactly as follows, including this exact spelling and punctuation:

> *I pledge my honor that I have not violated the Honor Code during this examination.*

I pledge my honor that I have not violated the Honor Code during this examination.

Now type your name as a signature confirming that you have adhered to the Honor Code:

Q2 The problems just seem to multiply...

5 Points

We saw in class that the same addition and subtraction algorithms can be used for both unsigned and signed (two's complement) N-bit binary integers. The purpose of this question is to demonstrate an example of how this also holds for multiplication, provided that we truncate the result back to N bits. For this question, we will consider N = 3.

Q2.1

1 Point

What are the ***decimal*** interpretations of the 3-bit binary *unsigned* integers 110_{B} and 111_{B}?

- ○ 2 and 3
- ○ 3 and 4
- ○ 4 and 5
- ○ 5 and 6
- ◉ 6 and 7

EXPLANATION

$110_{\mathrm{B}} = 4 + 2 = 6.\ 111_{\mathrm{B}} = 4 + 2 + 1 = 7.$

Q2.2

1 Point

When those two *unsigned* integers are multiplied, what is the ***untruncated binary*** result? (It might be easier to multiply in decimal and then convert to binary.)

- ○ 110
- ○ 1100
- ○ 10100
- ○ 11110
- ◉ 101010

EXPLANATION

$6 \times 7 = 42 = 32 + 8 + 2 = 101010_{\mathrm{B}}.$

Q2.3

1 Point

Now, what would be the ***unsigned 3-bit*** result? (You may notice that the result is too big for 3 bits, and so it overflows and must be truncated.)

- ◉ 010
- ○ 100
- ○ 101
- ○ 110
- ○ 111

EXPLANATION

We take the least-significant (i.e., rightmost) 3 bits of 101010_{B}. Or, we can think of this as 42 modulo 2^3.

Q2.4

1 Point

What are the ***decimal*** interpretations of the 3-bit binary *two's complement signed* integers 110_{B} and 111_{B}?

- ○ -1 and 0
- ◉ -2 and -1
- ○ -3 and -2
- ○ -2 and -3
- ○ 2 and 3

EXPLANATION

We know that both 3-bit values are negative, because they start with a "1" bit. To negate each one, we complement the bits and then add 1. So, $110 \rightarrow 001 \rightarrow 010 = 2$, showing that 110_{B} is -2, and $111 \rightarrow 000 \rightarrow 001 = 1$, showing that 111_{B} is -1.

Q2.5

1 Point

When those two *signed* integers are multiplied, what is the ***3-bit two's complement signed*** result?

- ○ 000
- ◉ 010
- ○ 100
- ○ 110
- ○ 111

EXPLANATION

$-2 \times -1 = 2$, which is 010_{B}. Notice that this is the same result as in question 2.3.

Q3 What's past is prologue

5 Points

As we learned in class, the sizes of most C datatypes are not fixed by the language and can be machine-dependent. You've learned about the sizes on ARM64 machines, but the earliest

implementation of C was on the PDP-11, with the following sizes:

Type	Size in Bytes
`char`	1
`int`	2
`long`	4
pointer	2
`size_t`	2

* *Technically speaking, `size_t` wasn't introduced into the C language until long after the days of the PDP-11, but let's suspend disbelief.*

Given the above, what would be the value of each of the following expressions on the PDP-11 architecture?

Q3.1
1 Point

`sizeof(42)`

- ○ 1
- ◉ 2
- ○ 3
- ○ 4
- ○ 8
- ○ 42

EXPLANATION

`42` is an `int`.

Q3.2
1 Point

`sizeof("42")`

Hint: *`sizeof` applied to an array returns the size of the array, not the size of the pointer to the first element.*

- ○ 1
- ○ 2
- ◉ 3
- ○ 4
- ○ 8
- ○ 42

EXPLANATION

`"42"` is a `char` array consisting of `'4'`, `'2'`, and `'\0'`.

Q3.3

1 Point

`sizeof(4 + 2 + 42L)`

- ○ 1
- ○ 2
- ○ 3
- ◉ 4
- ○ 8
- ○ 42

EXPLANATION

The additions involve `int` and `long` quantities, so the result is promoted to a `long`.

Q3.4

1 Point

`sizeof(sizeof(42))`

- ○ 1
- ◉ 2
- ○ 3
- ○ 4
- ○ 8
- ○ 42

EXPLANATION

The inner `sizeof` operator yields a value of type `size_t`.

Q3.5
1 Point

`sizeof(sizeof("42"))`

- ○ 1
- ◉ 2
- ○ 3
- ○ 4
- ○ 8
- ○ 42

EXPLANATION

The inner `sizeof` operator yields a value of type `size_t`.

Q4 So many ways to get it wrong
5 Points

The following functions are each intended to print a C string. They all use the `putchar` standard library function to print characters to `stdout` -- it has the following signature:

```
int putchar(int c);
```

Assume that `pc` is not `NULL`, and points to a correctly null-terminated string. For each function, indicate whether it succeeds, or how it fails.

Q4.1
1 Point

```
void fun1(const char *pc) {
    while (*pc != '\0')
        putchar(*pc); pc++;
}
```

- ○ Fails to compile.
- ◉ Enters an infinite loop.
- ○ Undefined behavior, may crash with a segmentation fault.
- ○ Runs and terminates, but produces incorrect output.
- ○ Correctly prints to `stdout` the string pointed to by `pc`.

EXPLANATION

The `pc++` is not part of the loop, despite the misleading placement on the same line as the `putchar`. So, `pc` is never incremented.

Q4.2

1 Point

```
void fun2(const char *pc) {
    while (*pc != '\0')
        putchar((*pc)++);
}
```

- ◉ Fails to compile.
- ○ Enters an infinite loop.
- ○ Undefined behavior, may crash with a segmentation fault.
- ○ Runs and terminates, but produces incorrect output.
- ○ Correctly prints to `stdout` the string pointed to by `pc`.

EXPLANATION

Fails to compile because of the attempt to increment `*pc`, which is declared `const`.

Q4.3

1 Point

```
void fun3(const char *pc) {
    while (*pc != '\0')
        putchar(*(pc++));
}
```

- ○ Fails to compile.
- ○ Enters an infinite loop.
- ○ Undefined behavior, may crash with a segmentation fault.
- ○ Runs and terminates, but produces incorrect output.
- ◉ Correctly prints to `stdout` the string pointed to by `pc`.

EXPLANATION

Executes correctly.

Q4.4

1 Point

```
void fun4(const char *pc) {
    while (pc != '\0')
        putchar(*(pc++));
}
```

- ○ Fails to compile.
- ○ Enters an infinite loop.
- ◉ Undefined behavior, may crash with a segmentation fault.
- ○ Runs and terminates, but produces incorrect output.
- ○ Correctly prints to `stdout` the string pointed to by `pc`.

EXPLANATION

Compares `pc`, not `*pc`, to `'\0'`. The latter is equivalent to 0, which is treated as a null pointer. But because `pc` is not `NULL`, this comparison is always false. As a result, `pc` is incremented until it points outside of the program's address space, leading to a segfault.

Q4.5
1 Point

```
void fun5(const char *pc) {
    while (*pc != '\0')
        putchar(*(++pc));
}
```

- ○ Fails to compile.
- ○ Enters an infinite loop.
- ○ Undefined behavior, may crash with a segmentation fault.
- ◉ Runs and terminates, but produces incorrect output.
- ○ Correctly prints to `stdout` the string pointed to by `pc`.

EXPLANATION

Fails to print the first character, and prints a `'\0'` at the end.

Q5 Scrambled scrambler
3 Points

Consider the following function:

```
/* On input: strings str1 and str2 must be the same length.
   After execution: the strings pointed to by str1 and str2
   have been modified in-place such that chars at *even* indices
```

```
   (counting from 0) have been swapped between str1 and str2. */
void scramble(char *str1, char *str2)
{
    /* Implementation goes here */
}
```

Here are some examples of the effect of the function, showing the values of strings `str1` and `str2` before and after a call to `scramble(str1, str2);`

str1 before	str2 before	str1 after	str2 after
"a"	"l"	"l"	"a"
"op"	"um"	"up"	"om"
"top"	"din"	"don"	"tip"
"fire"	"rock"	"rice"	"fork"

Your job is to determine the correct sequence of the following 9 lines of code (which are missing indentation in addition to being scrambled):

```
1   *str1 = *str2;
2   str1 += 2; str2 += 2;
3   }
4   if (*(str1 - 1) == '\0')
5   while (*str1 != '\0') {
6   char tmp = *str1;
7   assert(str1 != NULL); assert(str2 != NULL);
8   *str2 = tmp;
9   break;
```

Write the correct order in which those 9 lines of code should appear. Please ***separate the numbers by commas, and include no extra punctuation or white space***. For example, if you thought that the statements were exactly reversed, you would write `9,8,7,6,5,4,3,2,1` meaning that line 9 should appear first, line 8 second, etc.

Order:

7,5,6,1,8,2,4,9,3

EXPLANATION

Here's a guide to a possible thought process behind how to unscramble the code. We start by observing that, in general, variable declarations and asserts need to come first. We also recognize that we're going to need a loop - the only candidate is the `while` on line 5, paired with the closing brace on line 3. The `while` loop is checking `*str1`, so it's likely that something within the loop is going to need to advance the `str1` pointer - the only candidate is line 2. We also know that the loop needs to have a swap, and we recognize lines 6, 1, and 8 as a standard pattern for swapping two locations using a temporary variable. This, incidentally, is something that tripped up a number of students - as we saw a few times in the class, it is perfectly fine to declare a variable at the beginning of a *block*, such as the one after the `while`, and not just at the beginning of the function. Because there's no way to write the swap without line 6 inside the loop, we know that there won't be any variable declarations at the beginning of the function - and so the function will begin with the `assert`s on line 7. At this point, we have an outline that looks vaguely like 7,5,__,6,1,8,__,2,__,3, where we still haven't accounted for lines 4 and 9, and need to verify that 6,1,8 and 2 are in the correct order relative to each other. By tracing through some simple cases, we see that the code so far meets the specification for even-length strings, but line 2 steps past the end of odd-length strings. Adding in 4,9 between 2 and 3 handles this special case.

Q6 ?desreveR

6 Points

Contemplate the following program:

```
1   #include <stdio.h>
2   #include <stdlib.h>

3   struct Node {
4       char c;
5       struct Node *next;
6   };

7   struct Node *read_it(void) {
8       struct Node *curr = NULL;
9       int c;
10      while ((c = getchar()) != EOF) {
11          struct Node *newnode = calloc(1, sizeof(struct Node));
12          newnode->c = c;
13          newnode->next = curr;
14          curr = newnode;
15      }
16      return curr;
17  }

18  void print_it(struct Node *curr) {
19      if (!curr)
20          return;
21      print_it(curr->next);
22      putchar(curr->c);
23  }
```

```
24  int main(void) {
25      print_it(read_it());
26      return EXIT_SUCCESS;
27  }
```

Recall that `getchar` returns a character read from `stdin`, or `EOF` on end of file, and that `calloc(n, s)` allocates storage for `n` elements of size `s` and fills the newly-allocated memory with zeros. ***Assume that memory allocation always succeeds, and that the maximum recursion depth is never exceeded.***

Q6.1
2 Points

What does the program do?

- (●) Reads characters from `stdin` and then writes them to `stdout` in the order in which they were read
- ( ) Reads characters from `stdin` and then writes them to `stdout`, in reverse order
- ( ) Crashes on all inputs
- ( ) None of the above

EXPLANATION

`read_it` creates a linked list with the characters in reversed order. But then the ordering of the recursive call in `print_it`, relative to the `putchar`, causes the linked list to be printed in reverse order. The net effect is that characters are printed in the same order in which they were read.

Q6.2
2 Points

Suppose you now wish to print characters in the opposite order (i.e., reversed if the code prints them in the original order, or in the original order if the code prints them reversed). Which of the following changes, if applied individually to the code without other modifications, would achieve this? Select all changes that would work. ***Hint:*** *this is subtle. Trace through what happens on, say, a two-character input.*

- [ ] Swap lines 13 and 14
- [ ] Replace lines 13 and 14 with `curr->next = newnode;` and `curr = curr->next;`
- [x] Swap lines 21 and 22
- [ ] Replace lines 21 and 22 with `print_it(curr);` and `putchar(curr->next->c);`

EXPLANATION

The first option creates a mangled data structure, with nodes pointing to themselves. The second option *almost* works to construct the linked list from front to back, but it dereferences `NULL` the first time it is executed and returns the wrong node from `read_it`. The third option prints out the linked list in order, hence printing out the original characters in reversed order. The last option enters an infinite loop.

Q6.3

2 Points

The code as written has a memory leak. How could you fix it?

- ○ Add `free(curr);` immediately above line 16
- ○ Add `free(curr);` immediately above line 19
- ○ Add `free(curr);` immediately above line 21
- ○ Add `free(curr);` immediately above line 22
- ◉ Add `free(curr);` immediately above line 23
- ○ Change `main` to save the return value of `read_it()`, then call `free` on that value after passing it to `print_it`
- ○ None of the above will work

EXPLANATION

The correct answer frees each node after it will no longer be accessed again. Adding `free` at a different location can end up accessing nodes after freeing them. The "change `main`" option frees the first node in the linked list, but does not free the remaining nodes.