↳Add **Question 7.6**

✚ Add **Question 8**

Save Assignment

Exam statistics:
Mean: 25.13/33
Median: 26/33
StdDev: 4.97/33
Rough letter mapping:
27-33 is some kind of A, 20-26 is some kind of B, 13-19 is some kind of C.

Q1 Instructions and Pledge

1 Point

This exam consists of 6 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 Yes, it's the usual binary arithmetic question...

6 Points

Consider the following binary addition problem on 4-bit quantities:

$$1011_{\mathrm{B}} + 1101_{\mathrm{B}} = ?$$

Q2.1

1 Point

What is the ***4-bit binary*** result of the addition?
(Write only the 4 bits, with no extra white space, punctuation, or text.)

1000

EXPLANATION

The full result of the binary addition is 11000_{B}, which is truncated to the least-significant (rightmost) 4 bits.

Q2.2

1 Point

Interpreting the 4-bit result as an ***unsigned*** number, what is its *decimal*

value?
(Write only the value, with no extra white space, punctuation, or text.)

8

EXPLANATION

The addition problem is 11 + 13 = 24, and the truncation to 4 bits is equivalent to taking the result modulo 16. This results in 8, which is indeed the decimal value of 1000_{B}.

Q2.3

1 Point

If the addition is interpreted as ***unsigned***, it overflows:

- (•) True
- ( ) False

EXPLANATION

The correct result was too big to be represented in 4 bits.

Q2.4

1 Point

Interpreting the result as a 4-bit two's complement ***signed*** number, what is its *decimal* value?
(Write only the value, with no extra white space, punctuation, or text.)

-8

EXPLANATION

The addition problem is -5 + -3 = -8. Again, the latter corresponds to the 4-bit two's complement interpretation of 1000_{B}. Note that the result happens to be the "extra" negative two's complement number that has no positive counterpart in 4-bit two's complement.

Q2.5

1 Point

If the addition is interpreted as ***signed***, it overflows:

- ○ True
- ◉ False

EXPLANATION

The result is correctly representable in 4 bits.

Q2.6

1 Point

Overflow in an addition results in:

- ○ A compiler error
- ○ A compiler warning
- ○ A run-time crash
- ◉ None of the above

EXPLANATION

No checking for overflow is performed. Ever.

Q3 Mismanaged memory

4 Points

Consider the following code:

```
1  int main()
2  {
3      int *numbers;
4      numbers = malloc(sizeof(int));
5      numbers = calloc(10, sizeof(int));
6      numbers = realloc(numbers, 20 * sizeof(int));
7      return 0;
8  }
```

As a reminder, the memory allocation functions have the following calling conventions:

```
void *malloc(size_t size);
void *calloc(size_t n_elements, size_t size_per_element);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Q3.1

2 Points

Assuming that all appropriate header files have been included, and that ***all calls to memory management functions succeed***, how many bytes of memory are leaked by the original code?

- ○ 0
- ○ 4
- ○ 40
- ○ 44
- ○ 80
- ◉ 84
- ○ 124

EXPLANATION

The pointer from the first `malloc` is lost, so we leak 4 bytes there. The memory allocated by the `calloc` is correctly freed, if necessary, when the `realloc` succeeds. However, we never free memory from the `realloc`, so we leak an additional 20*4=80 bytes.

Q3.2

2 Points

Which statement/statements need to be added to prevent ***all*** memory leaks?

- ○ Add `free(numbers);` immediately after line 4
- ○ Add `free(numbers);` immediately after line 5
- ○ Add `free(numbers);` immediately after line 6
- ○ Add `free(numbers);` immediately after line 4 and line 5
- ◉ Add `free(numbers);` immediately after line 4 and line 6
- ○ Add `free(numbers);` immediately after line 5 and line 6

EXPLANATION

We must free the memory allocated by the `malloc` and `realloc`. However, we must not call `free` on the memory allocated by the `calloc`, since it is freed (if necessary) by `realloc`. Calling `free` after line 5 would result in a double free.

Q4 A hop, skip, and a jump

5 Points

Consider the following code:

```
1   struct Node {
2       int val;
3       struct Node *next;
```

```
4   };
5
6   struct List {
7       struct Node *first;
8   };
9
10  void List_insert(struct List *list, int new_val)
11  {
12      struct Node *new_node = malloc(sizeof(struct Node));
13      new_node->val = new_val;
14      new_node->next = list->first;
15      list->first = new_node;
16  }
17
18  void mystery(struct Node *node)
19  {
20      if (!node)
21          return;
22      printf("%d", node->val);
23      if (node->next)
24          mystery(node->next->next);
25      printf("%d", node->val);
26  }
27
28  int main()
29  {
30      struct List l = { NULL };
31      List_insert(&l, 1);
32      List_insert(&l, 2);
33      mystery(l.first);
34      List_insert(&l, 3);
35      mystery(l.first);
36      return 0;
37  }
```

Q4.1

1 Point

The implementation of `List_insert` runs in:

- [x] Constant time
- [ ] Linear time (i.e., proportional to the number of items in the list)
- [ ] Quadratic time

EXPLANATION

Because this implementation inserts onto the front of the list, it performs just a couple of simple operations. There is no traversal of the list, so it runs in constant time.

Q4.2

1 Point

The implementation of `List_insert` produces a list containing values in the *reverse* of the order in which they were inserted:

- [x] True
- [ ] False

EXPLANATION

Each call to `List_insert` inserts the value onto the front of the list. So, the List ends up in reverse order.

Q4.3

1 Point

Without changing the definitions of `struct Node` and `struct List`, it is possible to change `List_insert` such that it runs in constant time *and* produces a list containing values in the *same* order in which they were inserted:

- [ ] True
- [x] False

EXPLANATION

Producing a list in front-to-back order would require `List_insert` to insert `Node`s onto the back of the list. This requires traversing the list, and is inherently linear-time.

Q4.4

1 Point

What is printed by the call to `mystery` on line 33?
(Write only the digits printed, with no extra white space, punctuation, or text. If the code crashes or enters an infinite loop, write "crash".)

22

EXPLANATION

Before the first call to `mystery`, the list contains 2 and 1, in that order. The call to `mystery` prints 2, does a recursive call on `NULL` (which immediately returns), and prints 2 again.

Q4.5

1 Point

What is printed by the call to `mystery` on line 35?
(Write only the digits printed, with no extra white space, punctuation, or text. If the code crashes or enters an infinite loop, write "crash".)

3113

EXPLANATION

Before this call to `mystery`, the list contains 3, 2 and 1, in that order. The call to `mystery` prints 3, and does a recursive call on the `Node` containing the 1. The recursive call prints 1, skips doing another recursive call (since `node->next` is `NULL`), and prints 1 again. Finally, the original call to `mystery` prints 3 again.

Q5 Big cat

6 Points

For this problem, you will be using some `<string.h>` functions, which have the following signatures:

```
char *strcat(char *dest, const char *src);
char *strcpy(char *dest, const char *src);
int strcmp(const char *s1, const char *s2);
size_t strlen(const char *s);
char *strstr(const char *haystack, const char *needle);
```

Consider writing a function that concatenates *two* source strings, in order, onto a destination string. We shall call it `strtiger` (because it is bigger than just `strcat` (*groan*)).

Here is one implementation, to show the desired behavior in code:

```
char *strtiger1(char *dest, const char *src1, const char *src2)
{
    return strcat(strcat(dest, src1), src2);
}
```

Q5.1

1 Point

Now you want to write a version of `strtiger` that doesn't rely on `strcat`. Fill in the blanks with the appropriate expressions.

```
char *strtiger2(char *dest, const char *src1, const char *src2)
{
    char *p = /* __blank 1__ */;
    while (*p != '\0')
        p++;
    /* __blank 2__ */(p, src1);
    p += /* __blank 3__ */;
    strcpy(p, src2);
    return dest;
}
```

What should go in `__blank 1__`?

- ○ `src1`
- ◉ `dest`
- ○ `*src1`
- ○ `*dest`
- ○ `&src1`
- ○ `&dest`

EXPLANATION

We want to traverse to the end of the string that's currently in `dest`. Looking at the `while` loop, we realize that we're going to be incrementing `p` to walk through the string, and dereferencing `p` to determine whether we're at the end. So, we initialize `p` to `dest` itself, and not `*dest` or `&dest`.

Q5.2

1 Point

What should go in `__blank 2__`?

- [x] `strcpy`
- [ ] `strcmp`
- [ ] `strlen`
- [ ] `strstr`

EXPLANATION

`strcat` would have worked here, but of course we said that want to avoid that in `strtiger2`. After the `while` loop, `p` points at the `'\0'` character at the end of the old `dest` string, so this is the perfect location at which to *copy* `str1`.

Q5.3

1 Point

What should go in `__blank 3__`?

- [ ] `strlen(src1) - 1`
- [x] `strlen(src1)`
- [ ] `strlen(src1) + 1`
- [ ] `strlen(src2) - 1`
- [ ] `strlen(src2)`
- [ ] `strlen(src2) + 1`

EXPLANATION

We now wish to skip over the string we've just copied, and set ourselves up to copy `src2` at the new location of `p`. We want to move `p` forward by exactly `strlen(src1)` characters, since that will leave it pointing to the `'\0'` character at the end of the `str1` copy.

Q5.4

3 Points

We now have a skeleton of a third version of `strtiger`:

```
char *strtiger3(char *dest, const char *src1, const char *src2)
{
    char *p = dest;
    /* Insert code here */
    return dest;
}
```

but the lines that go in the middle have gotten scrambled:

```
1  while (*p++ = *src1++) ;
2  while (*p++ = *src2++) ;
3  while (*p)
4  p--;
5  p++;
```

Write the correct permutation of these 5 lines that would make the code work.
(Write only the 5 numbers representing the order of lines — e.g., 12345 or 54321 — with no extra white space, punctuation, or text.)

35142

EXPLANATION

We recognize lines 1 and 2 as loops that copy `src1` and `src2`, respectively, starting at the location of `p`. Note that these `while` loops have empty bodies — there is nothing between the conditions of the `while` and semicolons. Also note that they both copy the `'\0'` character at the end of the string, just as `strcpy` does — this will be important later on. So, we know that we will need to have line 1 and line 2 in that order, with some extra code to set up `p` correctly for the two copies. The first copy needs to have `p` pointing to the `'\0'` character at the end of `dest`, just as in `strtiger2`. Accomplishing that requires lines 3 and 5. (Note that the `while` on line 3 *does not* have an empty body on the same line, unlike the ones in lines 1 and 2. So, line 3 must be followed by either 4 or 5 to have a loop that makes sense.) Then, once the loop in line 1 is complete, `p` is pointing one character *past* the `'\0'` that just got copied (because of the postincrement). So, we need line 4 to move `p` back one character to set up for the loop in line 2.

Q6 Perplexing pointers and silly strings

6 Points

Consider the following code:

```
char princeton[] = "Tigers!";
const char *cos = "217!";
char *exam[9];
*exam = malloc(9);
strcpy(*exam, "Midterm!");
*(exam + 1) = (*exam) + 1;
```

Assuming that all appropriate header files have been included, and that the call to `malloc` succeeds, answer the following questions. ***Hint: definitely grab a sheet of scratch paper and draw out the variables and pointers! Also, read the definition of `exam` carefully!***

Q6.1

1 Point

What section of memory contains the characters `217!`?

- ○ heap
- ◉ rodata
- ○ stack
- ○ text
- ○ more than one of these

EXPLANATION

The string `"217!"` is allocated in rodata, with `cos` pointing to it.

Q6.2

1 Point

What section of memory contains the characters `Midterm!`?

- ○ heap
- ○ rodata
- ○ stack
- ○ text
- ◉ more than one of these

EXPLANATION

Initially, there is one copy of `"Midterm!"` located in rodata. A pointer to its first character is passed to `strcpy`, and afterwards the string `"Midterm!"` is also present in the memory allocated by `malloc` — i.e., on the heap.

Q6.3

1 Point

For each of the following expressions, indicate whether it results in a compiler error or warning, or what it evaluates to otherwise. ***Hint: the compiler will warn about comparisons between different pointer types.***

`princeton[6] == cos[3]`

- ○ Results in a compiler error or warning
- ○ Evaluates to `0` (FALSE)
- ◉ Evaluates to `1` (TRUE)

EXPLANATION

Character 6 (counting from 0, of course) of `princeton` is `'!'`, as is character 3 of `cos`.

Q6.4

1 Point

`exam[7] == cos[3]`

- ◉ Results in a compiler error or warning
- ○ Evaluates to `0` (FALSE)
- ○ Evaluates to `1` (TRUE)

EXPLANATION

This is the tricky one. It requires you to understand that `exam` is declared as an array of `char *`, and not just an array of `char`. So, `exam[7]` isn't a character — it's a *pointer*, unlike `cos[3]`. Trying to compare a pointer to a `char` will result in a compiler warning.

Q6.5

1 Point

`*(exam + 1) == (princeton + 1)`

- ○ Results in a compiler error or warning
- ◉ Evaluates to `0` (FALSE)
- ○ Evaluates to `1` (TRUE)

EXPLANATION

Unlike the previous question, this at least is a valid comparison between two pointers-to-`char`. The first one was set by the last line in the given code to point at the `'i'` in the string `"Midterm!"` that was copied into `*exam`. The second one is a pointer to the `'i'` in `"Tigers!"`. Even though both pointers point to an `'i'`, they are nonetheless different *pointers* and contain different memory locations.

Q6.6

1 Point

```
**(exam + 1) == *(princeton + 1)
```

- ○ Results in a compiler error or warning
- ○ Evaluates to `0` (FALSE)
- ◉ Evaluates to `1` (TRUE)

EXPLANATION

We now dereference both pointers described in the previous answer, and are comparing two `char`s, each equal to `'i'`.

Q7 !sgub eht dniF

5 Points

The following functions are each intended to print a C string ***backwards***. 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 whose length fits into an `int`. For each function, indicate whether it succeeds, or how it fails.

Q7.1

1 Point

```
void fun1(const char *pc)
{
    int i;
    for (i = strlen(pc) - 1; i >= 0; i--)
        putchar(pc[i]);
}
```

- ○ Results in a compiler error.
- ○ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ○ Runs and terminates, but produces incorrect output.
- ○ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ◉ Runs correctly and efficiently (within a constant factor of optimal).

EXPLANATION

The loop correctly sets `i` equal to indices ranging from the last character in the string to the first. Although we've pointed out in class the perils of having `strlen` in the ***condition*** of a `for` loop, where it gets called each time through the loop, having `strlen` in the ***initialization*** of the loop (i.e., before the first semicolon) means it gets executed only once. So, the code is not particularly inefficient.

Q7.2

1 Point

```
void fun2(const char *pc)
{
    int i;
    int n = strlen(pc);
    for (i = n; i >= 0; i--)
        putchar(pc[i]);
}
```

- ○ Results in a compiler error.
- ○ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ◉ Runs and terminates, but produces incorrect output.
- ○ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ○ Runs correctly and efficiently (within a constant factor of optimal).

EXPLANATION

Prints out a spurious `'\0'` character the first time through the loop.

Q7.3

1 Point

```
void fun3(const char *pc)
{
    int i;
    int n = strlen(pc - 1);
    for (i = n; i > 0; i--)
        putchar(pc[i]);
}
```

- ○ Results in a compiler error.
- ◉ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ○ Runs and terminates, but produces incorrect output.
- ○ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ○ Runs correctly and efficiently (within a constant factor of optimal).

EXPLANATION

Calls `strlen` on a pointer pointing to a memory location before the start of the string at `pc`. This not only gives the wrong value of `n`, but, more seriously, may cause `strlen` to read memory that it wasn't entitled to read. Although in practice the program most likely won't crash, the rules of C say that this is an illegal memory access that *may* result in a crash.

Q7.4

1 Point

```
void fun5(const char *pc)
{
    int i;
    int n = strlen(pc) - 1;
    for (i = n; i >= 0; i--)
        putchar((*pc) + i);
}
```

- ○ Results in a compiler error.
- ○ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ◉ Runs and terminates, but produces incorrect output.
- ○ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ○ Runs correctly and efficiently (within a constant factor of optimal).

EXPLANATION

The expression `(*pc)` is the first character in the string. The increment by `i` then increments the numeric value of the *character*, not the pointer. To be correct, this would have to read `putchar(*(pc + i));`

Q7.5

1 Point

```
void fun6(const char *pc)
{
    int i;
    for (i = 0; i < strlen(pc); i++)
        putchar(pc[strlen(pc) - i - 1]);
}
```

- ○ Results in a compiler error.
- ○ Compiles and runs, but accesses memory it shouldn't and may crash at run-time.
- ○ Runs and terminates, but produces incorrect output.
- ◉ Produces correct output, but is grossly inefficient (by more than a constant factor).
- ○ Runs correctly and efficiently (within a constant factor of optimal).

EXPLANATION

This produces correct output, but calls `strlen` twice per iteration of the loop. This has turned a linear-time algorithm into a quadratic-time one.