Exam statistics:
Mean: 36.7/55
Median: 38/55
StdDev: 6.1/55

90th %ile: 44/55
75th %ile: 41/55
60th %ile: 38.5/55
40th %ile: 36.6/55
25th %ile: 32/55
10th %ile: 28.3/55

Q1 Instructions and Pledge

2 Points

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

This is a partially "open-book" exam, with these ***allowances***:

- you may refer to the course's textbooks and the course materials found on or linked directly from the course's website.
- you may refer to your own notes and assignment solutions.
- you may use blank paper as scratch space, but you must enter your answer in the online system in order to receive credit.

In contrast to the allowances above, this exam has these ***restrictions***:

- You may **not** access other 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. Because students may be taking the exam late, this applies from when you take the exam until the sample solution has been published.
- The ***only*** form of allowable help from any person is that if you have technical issues or need to ask a clarifying question about the wording of some problem, you may post a **private** message on Ed for the course staff to answer at their discretion.

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 Teeny-tiny Computer

10 Points

Suppose we have a 6-bit computer. Answer the following questions.

Q2.1

2 Points

What is the **largest unsigned** number that can be represented in 6 bits?

In Binary:

111111

EXPLANATION

Unsigned integers are interpreted as just the value in binary, so the largest value possible is the value with all 1 bits. This is the decimal value 63 -- recall that the value of a bitstring of k 1 bits is $2^k - 1$.

In Hexadecimal:

3F

EXPLANATION

Each hexit is four bits, so the rightmost four bits, `1111` are the hexit `F` and the two remaining bits `11` are equivalent to the four bits `0011`, and thus are the hexit `3`.

Q2.2

2 Points

What is the **smallest** (i.e., most negative) **signed** number represented in 2's complement in 6 bits?

In Binary:

100000

EXPLANATION

Signed integers in 2's complement start at -1 (all 1 bits) and continue until the leftmost bit is 1 and all others are 0.

In Decimal:

-32

EXPLANATION

To compute the decimal value, we note that the leftmost bit is a 1, so it is a negative number, and then compute the value by the two's complement algorithm: flip all the bits to the left of the rightmost 1 (or flip all the bits and add one). In this case, we get back the same number, `100000`, which is 32.

Q2.3

6 Points

When doing **signed arithmetic** using 2's complement representation in 6 bits, consider the following three operations, where the first number is -12 and the second number is some *positive* number.

For each operation, indicate (Yes/No) whether it is possible for the result to **overflow**?

If an overflow is possible, provide an *example* that shows the second number in *decimal* (you don't need to show the result).
If an overflow is not possible, say "None" (no explanation is needed).

1. **Addition**

- ○ Yes
- ◉ No

Example of second number:

None

EXPLANATION

Signed overflow with addition happens when you add a positive and a positive and get a negative or add a negative and a negative and get a positive. When operands have opposite signs, their sum will never overflow.

2. **Subtraction**

- (●) Yes
- ( ) No

EXPLANATION

Signed overflow with subtraction happens when you subtract a negative from a positive and get a negative or subtract a positive from a negative and get a positive.

Example of second number:

21-31, inclusive

EXPLANATION

-32 is the most negative number that can be represented, so $-12 - x$ would overflow for $x > 20$.

3. **Multiplication**

- (●) Yes
- ( ) No

Example of second number:

3-31, inclusive

EXPLANATION

-32 is the most negative number that can be represented, so $-12 \times x$ would overflow for $x > 2$.

Q3 Any Portability in a Storm

10 Points

Each subquestion should be considered in the context of appearing in `main` after the following definitions:

```
struct S {
   char c;
   int i;
};

int iPlusOne(struct S s) {
   s.i++;
   return s.i;
}

int main(void) {
   char c;
   int i, j;
   unsigned int k;
   long l;
   struct S s = {'a', 2};
```

Q3.1

1 Point

This set of subquestions considers the following code:

```
   i = iPlusOne(s) + iPlusOne(s);
```

Executing on `armlab`, what is the value of the `int` `i` after its assignment?

6

EXPLANATION

`struct` parameters are passed by value, so the function `iPlusOne` gets its own copy of `s`, and the increment doesn't change the value of the `s` declared in `main`. Thus, the first call returns 3, as does the second, so their sum is 6.

Q3.2

1 Point

Is this resulting value of `i` portable (i.e., would it be the same on any system running C90)?

- (•) yes
- ( ) no

EXPLANATION

C90 specifies that `struct`s are passed by value, and there is no risk of the value 6 overflowing an `int`, which must be at least as large as a `char`, and thus must be able to store integer values up to at least 127.

Q3.3

1 Point

Is the size of `i` portable (i.e., would it be the same on any system running C90)?

- ( ) yes
- (•) no

EXPLANATION

The size of an `int` is system dependent. See the relevant **handout** from precept 4.

Q3.4

1 Point

Is the size of `s` portable (i.e., would it be the same on any system running C90)?

- ( ) yes
- (•) no

EXPLANATION

The size of a `struct` is *at least* as large as the sum of the sizes of its fields, but its exact size depends on padding. Padding within a `struct` is system dependent based on architectural alignment preferences implemented by the compiler.

Q3.5

1 Point

This set of subquestions considers the following code:

```
c = '\0';
c--;
j = c;
```

Executing on `armlab`, what is the value of the `int` `j` after its assignment?

255

EXPLANATION

`char`s are 1 byte. On `armlab`, a `char` is equivalent to an `unsigned char`. Thus, when we subtract 1 from `'\0'`, which is ASCII value `0`, we get `255`.

Q3.6

1 Point

Is this value of `j` portable to any system with character encodings using ASCII?

- ○ yes
- ◉ no

EXPLANATION

Whether `char` is equivalent to `signed char` or `unsigned char` is system dependent. If `char` were `signed char`, the resulting value would be `-1`.

Q3.7

1 Point

Is the size of the `char` `c` portable (i.e., would it be the same on any system running C90)?

- (●) yes
- ( ) no

EXPLANATION

The size of `char` is defined by the C standard to be 1 byte.

Q3.8

1 Point

This set of subquestions considers the following code:

```
k = 0U;
l = 1L;
if(k = l-1)
   k++;
else
   k--;
++k;
```

Executing on `armlab`, what is the value of the `unsigned int` `k` after this snippet?

0

EXPLANATION

- `l-1` yields `0L`.
- Inside the conditional, `k` gets assigned the value `0U`, which is automatically type-converted from `0L`.
- This assigned value is emitted to evaluate the conditional and is considered as "false".
- So the `k--` in the alternative is executed, resulting in decrementing `k` to yield the largest `unsigned int` value ($2^{32} - 1$).
- Unconditionally re-incrementing `k` overflows to yield `0U` again. We accepted either `0` or `0U`.

Q3.9
1 Point

Is this resulting value of `k` portable (i.e., would it be the same on any system running C90)?

- (•) yes
- ( ) no

EXPLANATION

- `1L-1` is always `0L`
- `0U-1` always results in the largest possible `unsigned int`; and the largest possible `unsigned int` plus `1` is always `0U`.
- Thus, we do always get `0U`, and our computation is portable -- perhaps surprising, since the value of the largest possible `unsigned int` itself is not the same on every system, as it depends on the size of an `unsigned int`.

Q3.10
1 Point

Is the size of the `long` `l` portable (i.e., would it be the same on any system running C90)?

- ( ) yes
- (•) no

EXPLANATION

The size of a `long` is system dependent, with the only requirement that `sizeof(long) >= sizeof(int)`. See the relevant **handout** from precept 4.

Q4 Places with Character (or not!)
20 Points

Consider the following C program, where some lines (`21-24`) are commented out initially. Assume that all calls to `malloc` succeed (and you don't need to check this).

```
01 #include <stdio.h>
02 #include <stdlib.h>
```

```c
#include <string.h>

int main(void) {
  enum {ARRAY_LENGTH = 11};
  int k = ARRAY_LENGTH / 2;
  char place1[ARRAY_LENGTH] =
    {'P', 'r', 'i', 'n', 'c', 'e', 't', 'o', 'n', '\0'};
  char *place2 = "Kingston";
  const char *place3 = "Queenston";
  char *pp, *pq;

  pp = (char *) malloc((ARRAY_LENGTH + 1) * sizeof(char));
  strcpy(pp, place1);
  pq = (char *) malloc((ARRAY_LENGTH + 1) * sizeof(char));
  strcpy(pq, place3);
  pp = pq;

  if (place1[k] == pp[k])
    printf("I love %s.\n", place1);
  else
    printf("I love %s.\n", place2);

  /* free(pp);  */
  /* pq = NULL; */
  /* printf("Where is %s?!", pq);*/
  /* free(place2); */

  return 0;
}
```

As a reminder:

- For sizes of various `types` on `armlab`, see the relevant **handout** from precept 4.
- The library functions have the following calling conventions:

```c
void *malloc(size_t size);
void free(void *ptr);
char *strcpy (char *destination, const char *source);
```

Q4.1

1 Point

How many bytes are allocated on the **stack frame** for the `main` function in this program? (Don't worry about alignment or padding.)

47

EXPLANATION

- `int k`: 4
- `char place1[ARRAY_LENGTH]`: 11
- `char *place2`: 8
- `const char *place3`: 8
- `char *pp`: 8
- `char *pq`: 8
- note that the no memory is allocated on line 5, as no variable is declared when declaring the anonymous enumeration type with enumerator constant `ARRAY_LENGTH`.

Q4.2

1 Point

How many bytes are allocated on the **heap** in this program? (Don't worry about alignment or padding, and assume that all calls to `malloc` succeed.)

24

EXPLANATION

Each of the two calls to `malloc` allocates 12 bytes (`ARRAY_LENGTH`+1) from the heap.

Q4.3

1 Point

In which section of the memory is the variable `pp` stored **before** line `12`?

Stack

EXPLANATION

It is a local variable. Until precept 14, when we see the `static` keyword applied to local variables, we only know one place for local variables: the stack.

Q4.4

1 Point

In which section of the memory is the variable `pp` stored **after** line `12`?

Stack

EXPLANATION

It is *still* a local variable. Its value at this point is now a pointer into the heap, but the memory for `pp` itself is still on the stack.

Q4.5

6 Points

In which section(s) of the memory are the following strings stored:

A. `Princeton`

Stack, Heap

EXPLANATION

- On line 7, `place1` is a compile-time defined array local variable, and thus its contents -- the string "Princeton" defined one character at a time by the initializer list -- are contained on its function's stackframe.
- On line 13, the contents of the `place1` array are copied into the location pointed to by `pp` via the call to `strcpy`. `pp` points at an address returned from `malloc` on line 12, and thus, the string "Princeton" is copied into the heap.

B. `Kingston`

RODATA

EXPLANATION

On line 9, `"Kingston"` is a string literal. String literals are found in the RODATA section. The local variable `place2` is on the stack, but its value is not the contents of the string but instead the address in RODATA.

C. `Queenston`

RODATA, Heap

EXPLANATION

- On line 10, `"Queenston"` is a string literal, and thus found in the RODATA section.
- On line 15, the contents of the string to which `place3` points (the string in the RODATA section) are copied into the location `pq` points, which is an address returned from `malloc` on line 14 and thus in the heap.

Q4.6

1 Point

What is the value of the variable `k` after line `06`?

5

EXPLANATION

Integer division `11/2` truncates 5.5 to 5.

Q4.7

1 Point

What is printed to `stdout` by lines `17-20` in the program ?

I love Kingston.

EXPLANATION

- Notice that in line 16, `pp` is pointed to the same thing `pq` points to. Thus, it no longer points to the string `"Princeton"`, but instead the string `"Queenston"` in the heap.
- `place1[k]` is the `'e'` in `"Princeton"`. `pp[k]` is the `'s'` in `"Queenston"`. Thus, we take the alternative in the if statement and print the statement about `place2`.

Q4.8

1 Point

How many bytes of **memory leak** does this program have (as shown)?

24

EXPLANATION

There are no calls to `free`, thus all the dynamically allocated memory is leaked.

Q4.9

1 Point

Suppose we *uncomment* the code at line `21`.
How many bytes of memory leak does the program have now?

12

EXPLANATION

Calling `free` on line 21 would free the memory allocated on line 14, because `pp` is now an alias to `pq`. The memory allocated on line 12, to which `pp` originally pointed, is still leaked.

Q4.10

2 Points

Suppose we now uncomment the code at line `23` also, i.e., lines `21` and `23` are both uncommented.
Choose ***all*** the following options that apply.

- [ ] The program compiles with a warning.
- [ ] There is no bug and the program executes successfully (i.e., without failure).
- [ ] There is at least one bug, and the program definitely aborts.
- [x] There is at least one bug, but the program may not abort.
- [x] The program may print something to `stdout` at line `23`.
- [x] The program may print `Where is Queenston?!` to `stdout` at line `23`.

EXPLANATION

- `pp` is an alias to `pq`, and thus the `free` on line 21 freed the string `pq` points to.
- Thus, passing `pq` to `printf` will result in dereferencing a dangling pointer: a memory management bug.
- The contents of the freed memory, however, is not guaranteed. It is possible that it may still contain the string "Queenston".

Q4.11

2 Points

Suppose we now uncomment the code at line `22` also, i.e., lines `21–23` are uncommented.
Choose ***all*** the following options that apply.

- [ ] The program compiles with a warning.
- [ ] There is no bug and the program executes successfully (i.e., without failure).
- [ ] There is at least one bug, and the program definitely aborts.
- [x] There is at least one bug, but the program may not abort.
- [x] The program may print something to `stdout` at line `23`.
- [ ] The program may print `Where is Queenston?!` to `stdout` at line `23`.

EXPLANATION

- `pp` is an alias to `pq`, and thus the `free` on line 21 freed the string `pq` points to.
- Setting `pq` to `NULL` on line 22 averts the risk of dereferencing a dangling pointer.
- But passing `pq` to `printf` is still a bug, however, since `pq` is no longer a pointer to a string.
- The result will be compiler-dependent. `gcc` prints `(null)` as the result for the format specifier `%s`, but other implementations may unconditionally dereference the argument, resulting in a segmentation fault.

Q4.12

2 Points

Finally, suppose we now uncomment the code at line `24` also, i.e., all lines `21–24` are uncommented.
Choose ***all*** the following options that apply.

- [ ] The program compiles with a warning.
- [ ] There is no bug and the program executes successfully (i.e., without failure).
- [x] There is at least one bug, and the program definitely aborts.
- [ ] There is at least one bug, but the program may not abort.
- [x] The program may print something to `stdout` at line `23`.
- [ ] The program may print `Where is Queenston?!` to `stdout` at line `23`.

EXPLANATION

- Calling `free` on something not in the heap is a memory management error. In this case, a value in RODATA is passed to `free`, which will certainly crash.

Q5 No stresslessness for "possessionlessness"

5 Points

In each subquestion you will consider an implementation of the following function, which is another member of the `string.h` interface that you worked with in Assignment 2:

```
char* strrchr(const char* s, int c)
```

This function locates the last (i.e., highest address) occurrence of `c` (converted to a `char`) in the string pointed to by `s`. It returns a pointer to the located character or `NULL` if the character does not appear in the string. The terminating nullbyte is considered to be part of the string; therefore if `c` is `'\0'`, the function locates the terminating `'\0'`.

Q5.1

1 Point

```
01  char* strrchr1(const char* s, int c) {
02    char* ret = NULL;
03    assert(s != NULL);
04    while(*s) {
```

```
05      if(*s == c)
06        ret = s;
07      s++;
08    }
09    if(!c)
10      ret = s;
11    return ret;
12  }
```

When compiled with `gcc217`, `strrchr1`:

- ○ compiles with no warnings or errors
- ○ produces an error on line 4
- ○ produces an error on line 9
- ◉ produces warnings on lines 6 and 10
- ○ produces a warning on line 11
- ○ more than one of the above

EXPLANATION

- Lines 6 and 10 assign `s`, which is a `const char*` into `ret`, which is a `char*`. This is reducing the `const` restriction, and thus we must "cast away constness" in order to avoid a warning.
- Other than that, the function works perfectly:
- line 4 bounds the traversal to stop at the nullbyte (because `'\0'` has value 0, an which is false)
- line 5 keeps track of the most recent location of `c` in the pointer `ret`
- line 7 advances `s` one character at a time in the traversal
- after the loop, there is a special-case check for if `c` is the nullbyte (because the loop ends when s points to `'\0'`)
- line 11 returns `ret`, which will be the most recently found `c`, or `NULL` if `c` was never found.

Q5.2

1 Point

```
01  char* strrchr2(const char s[], int c) {
02    size_t ret = 0;
03    size_t i = 0;
```

```
04    assert(s != NULL);
05    while(s[i] != '\0') {
06      if(s[i] == c)
07        ret = i;
08      i++;
09    }
10    if(c == '\0')
11      ret = i;
12    return (char*) &s[ret];
13  }
```

`strrchr2` produces the correct answer for:

- ○ all possible strings `s` and characters `c`
- ○ all possible strings `s` and characters `c`, but not efficiently (as defined in A2)
- ◉ some, but not all, possible strings `s` and characters `c`
- ○ only the empty string `s`
- ○ no strings `s`

EXPLANATION

- Correct behavior: any string that contains `c`.
- Incorrect behavior: any string that does not contain `c`. In this case, it returns the address of the 0th character.

Q5.3

1 Point

```
01  char* strrchr3(const char s[], int c) {
02    size_t i = 0;
03    assert(s != NULL);
04    while(s[i] != '\0') {
05      if(s[i] == c)
06        return (char*) &s[i];
07      i++;
08    }
09    if(c == '\0')
10      return (char*) &s[i];
11    return NULL;
12  }
```

`strrchr3` produces the correct answer for:

- ○ all possible strings `s` and characters `c`
- ○ all possible strings `s` and characters `c`, but not efficiently (as defined in A2)
- ◉ some, but not all, possible strings `s` and characters `c`
- ○ only the empty string `s`
- ○ no strings `s`

EXPLANATION

- Correct behavior: any string that does not contain `c` or contains `c` only 1 time.
- Incorrect behavior: any string that contains `c` multiple times. In this case, it returns the address of the *first* instance, not the *last* instance of `c`.

Q5.4

1 Point

```
01  char* strrchr4(const char s[], int c) {
02    size_t i = strlen(s);
03    char* p = NULL;
04    for( ; i > 0; i--) {
05      if(s[i] == c)
06        return (char*) &s[i];
07    }
08    if(s[0] == c)
09      return (char*) &s[i];
10    return p;
11  }
```

`strrchr4` produces the correct answer for:

- ○ all possible strings `s` and characters `c`
- ◉ all possible strings `s` and characters `c`, but not efficiently (as defined in A2)
- ○ some, but not all, possible strings `s` and characters `c`
- ○ only the empty string `s`
- ○ no strings `s`

EXPLANATION

This algorithm does work correctly, but `strlen` traverses the entire string `s` then the `for` loop traverses `s` again in the opposite direction, making this a two-pass algorithm for something that can be done in a single traversal.

Q5.5

1 Point

```
01  char* strrchr5(const char* s, int c) {
02    const char* ret = NULL;
03    assert(s != NULL);
04    while(*s != '\0') {
05       if(*s == c)
06         ret = s;
07       s++;
08    }
09    if(c == '\0')
10      return (char*) s;
11    return (char*) ret;
12  }
```

`strrchr5` produces the correct answer for:

- (●) all possible strings `s` and characters `c`
- ( ) all possible strings `s` and characters `c`, but not efficiently (as defined in A2)
- ( ) some, but not all, possible strings `s` and characters `c`
- ( ) only the empty string `s`
- ( ) no strings `s`

EXPLANATION

This is a correct implementation, very similar to `strrchr1` in Q5.1 except with:

- less terse comparisons on lines 4 and 9
- the `const` keyword on the traversing pointer `ret`
- correct casting away of `const`ness where necessary

Consider two cases: `c` is `'\0'`, and `c` is anything else:

- In the general case, the loop tracks the last occurrence of `c` in `ret`, and breaks at the terminating nullbyte, after which we cast away `const`ness from `ret` and return.
- For the special case, since the loop was advancing `s` during its traversal, the function can just return `s` (again after casting away `const`ness) since it points to the character at which the loop broke: the trailing nullbyte.

Q6 Sort out this mess!

8 Points

The following function `isSorted` should return `1` (true) if the argument array `a` of `n` integer elements is sorted in ascending order, i.e., its elements are in non-decreasing order, and `0` (false) otherwise.
However, it has several bugs.

```
1  int isSorted(int *a, size_t n){
2    size_t i;
3    int check;

4    /* add assert statement(s) here */

5    for (i=0; i < n; i++)
6       check = a[i] <= a[++i];
7    return check;
8  }
```

Q6.1

2 Points

To start with, add any missing ***assert*** statements at line `4`. You can assume that the header file `assert.h` has been included.

```
assert(a != NULL);
```

(Note: the missing assert(s) are not bugs *per se* -- adding them is recommended for testing code modularly.)

EXPLANATION

- The function dereferences `a` by accessing `a[i]`, thus we must `assert` that `a` is not `NULL`.
- This is the only validation we should do, because we cannot validate that `a` has `n` elements, and there is no sensible check on `n`'s value, as it is unrestricted.
- Note that because `n` is of type `size_t`, checking `n >= 0` results in a compiler warning that the comparison is always true.

Q6.2

6 Points

Now identify up to ***three*** different bugs.
For each bug, describe the effect of the bug (concisely) and suggest a fix.

Assume that all required C header files are included, and don't worry about comments, efficiency, error checking, or style.

Bug 1:
Effect (concise explanation):

Suggested fix (show the fixed code, with line numbers):

Bug 2:
Effect (concise explanation):

Suggested fix (show the fixed code, with line numbers):

Bug 3:
Effect (concise explanation):

Suggested fix (show the fixed code, with line numbers):

EXPLANATION

There are four definite bugs in this code and one additional potential bug.

- Bug 0: `check` is not initialized, so if the loop never executes, i.e., the array is a 0-element array or a 0- or 1-element array after fixing bug 2, the value returned will not be well defined. The fix is to initialize `check` to `1` (or any other true value) before the loop. We choose a true value because an array with 0 or 1 elements is trivially sorted.
- Bug 1. `i` updates twice per iteration, which results in skipping some comparisons (e.g. elements 0 and 1 are compared, and elements 2 and 3 are compared, but elements 1 and 2 are not.) The easiest fix is accessing the subsequent element using `i+1` not `++i` on line 6. An alternative equivalent fix would be to remove the update step from the loop signature on line 5.
- Bug 2. The loop bounds in line 5 will cause the array to be accessed out of bounds in accessing the subsequent element, which will be `a[n]` in line 6 (whether as `a[++i]` or `a[i+1]`). The fix is to bound the loop by `i < n-1`.
- Bug 3. The value of `check` is reset each time through the loop, so the value returned is the `check` from the final comparison only. There are two easy ways to fix this: accumulate `check` with & or && or * so that if any individual comparison is ever false `check` will remain false forever; or immediately return false if `check` ever becomes 0, so the loop will only complete if all comparisons are true.
- Possible Bug 4. The compiler emits a warning that the order of accessing the two sides of the comparison in line 6 is not defined by the C standard. If the comparison is processed left-to-right it gets the desired behavior (which is the case on `armlab`), but if the comparison is processed right-to-left it ends up comparing each item with itself.

THE END