Create Rubric

55 points

Create your rubric now or come back to it later. You can also make edits to your rubric while grading.

Q1 Q1: You Belong with Me

6 points

Q1.1 a: max declaration

1 point

Rubric Settings

1 **+1.0**

Correct: the function *declaration* appears in the **interface** file.

The preprocessor handles `#include` statements that result in the interface's declarations being injected into the top of both the client and the implementation.

2 **+0.0**

The function *declaration* appears in the **interface** file.

The preprocessor handles `#include` statements that result in the interface's declarations being injected into the top of both the client and the implementation.

+ Add Rubric Item | Create Group | Import...

Q1.2 b: max definition header

1 point

Rubric Settings

1 +1.0

Correct: the *definition* for the function declared in the interface appears in the **implementation** file.

2 +0.0

The *definition* for the function declared in the interface appears in the **implementation** file.

+ Add Rubric Item | Create Group | Import...

Q1.3 c: main definition header

1 point

Rubric Settings

1 +1.0

Correct: the **client** file is the one with the `main` function for the program.

2 +0.0

The **client** file is the one with the `main` function for the program.

+ Add Rubric Item | Create Group | Import...

Q1.4 d: multiple inclusion guard

1 point

Rubric Settings

1 +1.0

Correct: this is the second part of the `#ifndef / #define / #endif` pattern that appears in **interface** files to guard against multiple inclusion.

2 +0.0

This is the second part of the `#ifndef / #define / #endif` pattern that appears in **interface** files to guard against multiple inclusion.

✚ Add Rubric Item | Create Group | Import...

Q1.5 e: max conditional

1 point

Rubric Settings

1 +1.0

Correct: this is the logic in the body of the `IntMath_max` function, which appears in the **implementation** file.

2 +0.0

This is the logic in the body of the `IntMath_max` function, which appears in the **implementation** file.

✚ Add Rubric Item | Create Group | Import...

Q1.6 f: max return value check

1 point

Rubric Settings

1 +1.0

Correct: this checks the return value from a *call* to the `IntMath_max` function, which is most likely to happen from the **client** program.

2 +0.0

This checks the return value from a *call* to the `IntMath_max` function, which is most likely to happen from the **client** program.

✚ Add Rubric Item | Create Group | Import...

Q2 Q2: Anti-Hero

5 points

Q2.1 a: ~(0xF << 2)

1 point

Rubric Settings

1 **+1.0**

Correct: the answer is -61.

0xF is 32 bits long, with the value 0...0001111.

left-shifting by two bits yields the value 0...0111100.

bitwise negating yields the value 1...1000011

this is a **negative** number, because the leftmost bit is 1.

we do the two's complement algorithm to find the *magnitude* of the negative number: 0...0111101 , which is 1+4+8+16+32 = 61.

2 **+0.0**

The answer is -61.

0xF is 32 bits long, with the value 0...0001111.

left-shifting by two bits yields the value 0...0111100.

bitwise negating yields the value 1...1000011

this is a **negative** number, because the leftmost bit is 1.

we do the two's complement algorithm to find the *magnitude* of the negative number: 0...0111101 , which is 1+4+8+16+32 = 61.

(3 was the most common incorrect answer, which results from doing the same process above but without acknowledging all the leading 0s initially: 1111 -> 111100 -> 000011 = 3.)

✚ Add Rubric Item | Create Group | Import...

Q2.2 b: (-~0xF) >> 3

1 point

Rubric Settings

1 **+1.0**

Correct: the answer is 2.

0xF is 32 bits long, with the value 0...0001111.

Bitwise negating yields 1...1110000.

Arithmetic negating requires doing the two's complement algorithm: 0...0010000.

This is a non-negative (because the leftmost bit is 0) signed integer, so right-shift is well-defined to replace from the left with 0s.

Thus, right-shifting by 3 bits yields: 0...0000010 = 2.

2 **+0.0**

The correct answer is 2.

0xF is 32 bits long, with the value 0...0001111.

Bitwise negating yields 1...1110000.

Arithmetic negating requires doing the two's complement algorithm: 0...0010000.

This is a non-negative (because the leftmost bit is 0) signed integer, so right-shift is well-defined to replace from the left with 0s.

Thus, right-shifting by 3 bits yields: 0...0000010 = 2.

✚ Add Rubric Item | Create Group | Import...

Q2.3 c: -(0xF + !~0xF)

1 point

Rubric Settings

1 +1.0

Correct: the answer is 1.

0xF is 32 bits long, with the value 0...0001111.

Bitwise negating yields 1...1110000.

Adding the original 0...0001111 with the bitwise negated 1...1110000 yields 1...1111111

Arithmetic negating requires doing the two's complement algorithm: 0...0000001 = 1.

2 +0.0

The correct answer is 1.

0xF is 32 bits long, with the value 0...0001111.

Bitwise negating yields 1...1110000.

Adding the original 0...0001111 with the bitwise negated 1...1110000 yields 1...1111111

Arithmetic negating requires doing the two's complement algorithm: 0...0000001 = 1.

3 +1.0

The correct answer is 1.

0xF is 32 bits long, with the value 0...0001111.

Bitwise negating yields 1...1110000.

Adding the original 0...0001111 with the bitwise negated 1...1110000 yields 1...1111111

Arithmetic negating requires doing the two's complement algorithm: 0...0000001 = 1.

You were asked to give the answer in base 10. This answer is in binary, and has an insufficient number of bits, but full credit for the correct value.

✚ Add Rubric Item | Create Group | Import...

Q2.4 d: 0xF & ~(1 << 3)

1 point

Rubric Settings

1 +1.0

Correct: the answer is 7.

0xF is 32 bits long, with the value 0...0001111.

1 is also 32 bits long, with the value 0...0000001.

Left-shifting 1 by 3 bits yields 0...0001000.

Bitwise negating the result of the shift yields 1...1110111.

Bitwise ANDing 0...0001111 with 1...1110111 yields 0...0000111 = 7.

2 +0.0

The correct answer is 7

0xF is 32 bits long, with the value 0...0001111.

1 is also 32 bits long, with the value 0...0000001.

Left-shifting 1 by 3 bits yields 0...0001000.

Bitwise negating the result of the shift yields 1...1110111.

Bitwise ANDing 0...0001111 with 1...1110111 yields 0...0000111 = 7.

3 **+1.0**

The correct answer is 7

0xF is 32 bits long, with the value 0...0001111.

1 is also 32 bits long, with the value 0...0000001.

Left-shifting 1 by 3 bits yields 0...0001000.

Bitwise negating the result of the shift yields 1...1110111.

Bitwise ANDing 0...0001111 with 1...1110111 yields 0...0000111 = 7.

You were asked to give the answer in base 10. This answer is in binary, and has an insufficient number of bits, but full credit for the correct value.

+ Add Rubric Item | Create Group | Import...

Q2.5 e: ~(0xF >> !0xF)

1 point

Rubric Settings

1 **+1.0**

Correct: the answer is -16.

0xF is 32 bits long, with the value 0...0001111.

This is a non-zero value, so as a logical value it is

interpreted as "true". Thus, logical negation will produce "false", which yields the value 0.

Right-shifting by 0 bits does not make any change, so the expression in the parentheses (still) evaluates to 0xF.

Bitwise negating 0xF yields 1...1110000.

The leftmost bit of this number is a 1, so this is a **negative** number.

To find the *magnitude* of this negative number, complete the 2's complement algorithm: 0...0010000 = 16.

2 **+0.0**

The correct answer is -16

0xF is 32 bits long, with the value 0...0001111.

This is a non-zero value, so as a logical value it is interpreted as "true". Thus, logical negation will produce "false", which yields the value 0.

Right-shifting by 0 bits does not make any change, so the expression in the parentheses is still 0xF.

Bitwise negating 0xF yields 1...1110000.

The leftmost bit of this number is a 1, so this is a **negative** number.

To find the *magnitude* of this negative number, complete the 2's complement algorithm: 0...0010000 = 16.

+ Add Rubric Item | Create Group | Import...

Q3 Q3: I Know Places | Bigger than the Whole Sky

12 points

Q3.1 (1-4) Section

4 points

Rubric Settings

1 +1.0

`ai`, `ui`, `aiDigits`, and `pui` are all parameters or local variables of a function, and go in the Stack section in the activation record (aka stackframe) for a function call of that function.

2 +2.0

`ai`, `ui`, `aiDigits`, and `pui` are all parameters or local variables of a function, and go in the Stack section in the activation record (aka stackframe) for a function call of that function.

3 +3.0

`ai`, `ui`, `aiDigits`, and `pui` are all parameters or local variables of a function, and go in the Stack section in the activation record (aka stackframe) for a function call of that function.

4 +4.0

Correct: `ai`, `ui`, `aiDigits`, and `pui` are all parameters or local variables of a function, and go in the Stack section in the activation record (aka stackframe) for a function call of that function.

+ Add Rubric Item | Create Group | Import...

Q3.2 (1) Number of Bytes

1 point

Rubric Settings

1 +1.0

Correct: arrays are passed into functions as pointers to the 0th element of the array, so the function's parameter

is a pointer type. Pointers on armlab are 8 bytes.

2 +0.0

Arrays are passed into functions as pointers to the 0th element of the array, so the function's parameter is a pointer type. Pointers on armlab are 8 bytes.

✚ Add Rubric Item | Create Group | Import...

Q3.3 (2) Number of Bytes

1 point

Rubric Settings

1 +1.0

Correct: on armlab, `int`s (of any signedness) are 4 bytes.

2 +0.0

On armlab, `int`s (of any signedness) are 4 bytes.

✚ Add Rubric Item | Create Group | Import...

Q3.4 (3) Number of Bytes

1 point

Rubric Settings

1 +1.0

Correct: `aiDigits` is an array of 4 `int`s, each of which is allocated 4 bytes on armlab. There is no array overhead, so the total memory allocated for `aiDigits` is:
$4 * 4 = 16$

2 +0.0

`aiDigits` is an array of 4 `int`s (and **only** 4 -- unlike with `char` arrays representing strings, there is no trailing sentinel value in arbitrary arrays) each of which is allocated 4 bytes on armlab. There is no array overhead, so the total memory allocated for `aiDigits` is:
$4 * 4 = 16$

+ Add Rubric Item | Create Group | Import...

Q3.5 (4) Number of Bytes

1 point

Rubric Settings

1 +1.0

Correct: on armlab, all pointer types are 8 bytes.

2 +0.0

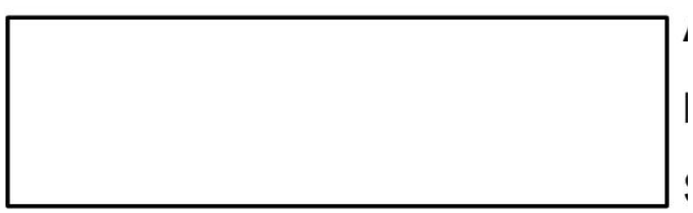
On armlab, all pointer types are 8 bytes.

+ Add Rubric Item | Create Group | Import...

Q3.6 (5) Section

1 point

Rubric Settings



1 +1.0

Correct: `malloc` allocates space from the **heap** and returns a pointer to that space.

2 +0.0

`malloc` allocates space from the **heap** and returns a pointer to that space.

+ Add Rubric Item | Create Group | Import...

Q3.7 (5) Number of Bytes

1 point

Rubric Settings

1 +1.0

Correct: `pui` is an `unsigned int *`, thus `*pui` (i.e., the thing that `pui` points to) is an `unsigned int`. On armlab, `int`s (of any signedness) are 4 bytes.

2 +0.0

`pui` is an `unsigned int *`, thus `*pui` (i.e., the thing that `pui` points to) is an `unsigned int`. On armlab, `int`s (of any signedness) are 4 bytes.

+ Add Rubric Item | Create Group | Import...

Q3.8 (6) Section

1 point

Rubric Settings

1 +1.0

Correct: string literals are allocated as arrays of characters in the RODATA section.

2 +0.0

String literals are allocated as arrays of characters in the RODATA section.

Add Rubric Item | Create Group | Import...

Q3.9 (6) Number of Bytes

1 point

Rubric Settings

1	+1.0	Correct: the contents of the RODATA array of characters are: `{'%', 'u', '\n', '\0'}`: a total of 4 bytes.
2	+0.0	The contents of the RODATA array of characters are: `{'%', 'u', '\n', '\0'}`: a total of 4 bytes. Common close-but-incorrect answers included 3 bytes (likely due to forgetting the `'\0'`) and 5 bytes (likely due to interpreting `'\n'` as two separate characters rather than the C character constant name for the newline character)

✚ Add Rubric Item | Create Group | Import...

Q4 Q4: Forever & Always

12 points

Q4.1 a: a0++

1 point

Rubric Settings

○ ○

1	+1.0	Correct: it is not legal to increment an array name.
2	+0.0	

It is not legal to increment an array name.

✚ Add Rubric Item | Create Group | Import...

Q4.2 b: (*a0)++

1 point

Rubric Settings

○ ○

1 **+1.0**

Correct:

Dereferencing `a0` yields the character '2'
Incrementing that result will place '3' in `a0[0]`

2 **+0.0**

Dereferencing `a0` yields the character '2'
Incrementing that result will place '3' in `a0[0]`

✚ Add Rubric Item | Create Group | Import...

Q4.3 c: p1++

1 point

Rubric Settings

○ ○

1 **+1.0**

Correct:

The pointer `p1` points to `a0[0]`.
Incrementing `p1` updates the pointer to point to `a0[1]`.

2 **+0.0**

The pointer `p1` points to `a0[0]`.
Incrementing `p1` updates the pointer to point to `a0[1]`.

✚ Add Rubric Item | Create Group | Import...

Q4.4 d: (*p1)++

1 point

Rubric Settings

1 +1.0

Correct:

Dereferencing `p1` yields the character `'2'`.
Incrementing that causes the character `'3'` to be placed in `a0[0]`

2 +0.0

Dereferencing `p1` yields the character `'2'`.
Incrementing that causes the character `'3'` to be placed in `a0[0]`

✚ Add Rubric Item | Create Group | Import...

Q4.5 e: (&p1)++

1 point

Rubric Settings

1 +1.0

Correct:

The address of `p1` is an address to the memory location on the stack where `p1` is. The address does not exist as a variable that we can increment.

2 +0.0

The address of `p1` is an address to the memory location on the stack where `p1` is. The address does not exist as a variable that we can increment.

✚ Add Rubric Item | Create Group | Import...

Q4.6 f: (*(&p1))++

1 point

Rubric Settings

○ ○

1 +1.0

Correct:

The `&` and `*` operators are inverses of each other. If we take the address of `p1` and then dereference that address, we are back at the original pointer, `p1`. So this statement is the same as 4.c.

2 +0.0

The `&` and `*` operators are inverses of each other. If we take the address of `p1` and then dereference that address, we are back at the original pointer, `p1`. So this statement is the same as 4.c.

✚ Add Rubric Item | Create Group | Import...

Q4.7 p2++

1 point

Rubric Settings

○ ○

1 +1.0

Correct:

The pointer `p2` points to `a0[0]`.
Incrementing `p2` updates the pointer to point to `a0[1]`.
It is the [illegible] that is [illegible], not the pointer, so it is fine to

It is the `char` that is `const`, not the pointer, so it is fine to increment the pointer.

2 **+0.0**

The pointer `p2` points to `a0[0]`.
Incrementing `p2` updates the pointer to point to `a0[1]`.
It is the `char` that is `const`, not the pointer, so it is fine to increment the pointer.

+ Add Rubric Item | Create Group | Import...

Q4.8 (*p2)++

1 point

Rubric Settings

1 **+1.0**

Correct:

Dereferencing `p2` gives us a `const char`. We cannot increment a `const char`.

2 **+0.0**

Dereferencing `p2` gives us a `const char`. We cannot increment a `const char`.

+ Add Rubric Item | Create Group | Import...

Q4.9 *(p3++)

1 point

Rubric Settings

1 **+1.0**

Correct:

`p3` is a `const` pointer. We cannot increment a `const` pointer.

2 **+0.0**

`p3` is a `const` pointer. We cannot increment a `const` pointer.

✚ Add Rubric Item | Create Group | Import...

Q4.10 (*p3)++

1 point

Rubric Settings

1 **+1.0**

Correct:

Dereferencing `p3` yields the character '2'.
Incrementing that causes the character '3' to be placed in `a0[0]`.
It is the pointer that is `const`. It is OK to increment the `char`.

2 **+0.0**

Dereferencing `p3` yields the character '2'.
Incrementing that causes the character '3' to be placed in `a0[0]`.
It is the pointer that is `const`. It is OK to increment the `char`.

✚ Add Rubric Item | Create Group | Import...

Q4.11 p4++

1 point

Rubric Settings

1 +1.0

Correct:

`p4` is a `const` pointer. We cannot increment it.

2 +0.0

`p4` is a `const` pointer. We cannot increment it.

✚ Add Rubric Item | Create Group | Import...

Q4.12 (*p4)++

1 point

Rubric Settings

1 +1.0

Correct:

Dereferencing `p4` gives us a `const char`. We cannot increment a `const char`.

2 +0.0

Dereferencing `p4` gives us a `const char`. We cannot increment a `const char`.

✚ Add Rubric Item | Create Group | Import...

Q5 Q5: I Did Something Bad

12 points

Q5.1 a

Rubric Settings

2 points

Rubric Settings

☐ ☐ ☐ ☐ ☐ ☐

1 **+2.0**

Correct

The array `ai` was not allocated using `malloc()`, `calloc()` or `realloc()` and thus it is incorrect to use it as an argument in a call to `free()`.

2 **+1.0**

The correct answer is D.
`free(ai)` tries to free memory that was not allocated with `malloc()`, `calloc()` or `realloc()`. It does not try to access it by dereferencing.

3 **+0.0**

The correct answer is D.
The array `ai` was not allocated using `malloc()`, `calloc()` or `realloc()` and thus it is incorrect to use it as an argument in a call to `free()`.

4 **+1.0**

The correct answer is D.
`free(ai)` tries to free memory that was not allocated with `malloc()`, `calloc()` or `realloc()`. But there are no double frees in this code snippet.

5 **+1.0**

The correct answer is D.
`free(ai)` tries to free memory that was not allocated with `malloc()`, `calloc()` or `realloc()`. But there are no memory leaks in this code snippet.

+ Add Rubric Item | Create Group | Import...

Q5.2 b

2 points

Rubric Settings

☐ ☐ ☐ ☐ ☐ ☐

1 **+2.0**

Correct

When we execute pi1 = ai; we lose the pointer that pi1 used to hold for the memory allocated in the Heap, thus creating a memory leak.

2 **+0.0**

The correct answer is C.
When we execute pi1 = ai; we lose the pointer that pi1 used to hold for the memory allocated in the Heap, thus creating a memory leak.

3 **+1.0**

The correct answer is C.
When we execute pi1 = ai; we lose the pointer that pi1 used to hold for the memory allocated in the Heap, thus creating a memory leak. But we are not accessing unallocated memory.

4 **+0.0**

The correct answer is C.
It cannot be both C and None.

+ Add Rubric Item | **Create Group** | **Import...**

Q5.3 c

2 points

Rubric Settings

☐ ☐ ☐ ☐ ☐ ☐

1 **+2.0**

Correct

2 +0.0

The correct answer is **None**.

The space allocated in the first `malloc` is freed by the first `free`. Then `p1` is set to be an alias to `p2`, and is used in the second `free` to free the space allocated by the second `malloc` .

+ Add Rubric Item | Create Group | Import...

Q5.4 d

2 points

Rubric Settings

☐ ☐ ☐ ☐ ☐ ☐

1 +2.0

Correct

`pi2 = pi1;` creates a memory leak.
`free(pi2);` frees the memory pointed to by both `pi1` and `pi2`, so `*pi1` accesses freed memory.

2 +0.0

The correct answer is to choose both B and C.
`pi2 = pi1;` creates a memory leak.
`free(pi2);` frees the memory pointed to by both `pi1` and `pi2`, so `*pi1` accesses freed memory.

3 +1.0

The correct answer is to choose both B and C.
`pi2 = pi1;` creates a memory leak.
`free(pi2);` frees the memory pointed to by both `pi1` and `pi2`, so `*pi1` accesses freed memory.

4 **+1.0**

The correct answer is to choose both B and C.
`pi2 = pi1;` creates a memory leak.
`free(pi2);` frees the memory pointed to by both `pi1` and `pi2`, so `*pi1` accesses freed memory.
But there is no accessing of unallocated memory.

✚ Add Rubric Item | Create Group | Import...

Q5.5 e

2 points

Rubric Settings

☐ ☐ ☐ ☐ ☐ ☐

1 **+2.0**

Correct.

The correct answer is A.
We allocated room for 3 `int` for `pi2`. That means we did not allocate memory for a fourth array cell, `pi2[3]`.

2 **+0.0**

The correct answer is A.
We allocated room for 3 `int` for `pi2`. That means we did not allocate memory for a fourth array cell, `pi2[3]`.

3 **+1.0**

The correct answer is A.
We allocated room for 3 `int` for `pi2`. That means we did not allocate memory for a fourth array cell, `pi2[3]`.
There is no memory leak.

✚ Add Rubric Item | Create Group | Import...

Q5.6 f

2 points

Rubric Settings

☐ ☐ ☐ ☐ ☐ ☐

1 **+2.0**

Correct

When we assign `pi2 = pi1;` after freeing pi1, and then assign `*pi2 = ai[2];` we are creating a memory leak (the area p_reviously allocated to `pi2`) and accessing freed memory. Then, when we `free(pi2);` (which now points to the original `pi1` memory area) we have a double free.

2 **+0.0**

The correct answer is to choose B, C and E.
When we assign `pi2 = pi1;` after freeing pi1, and then assign `*pi2 = ai[2];` we are creating a memory leak (the area p_reviously allocated to `pi2`) and accessing freed memory. Then, when we `free(pi2);` (which now points to the original `pi1` memory area) we have a double free.

3 **+1.0**

The correct answer is to choose B, C and E.
When we assign `pi2 = pi1;` after freeing pi1, and then assign `*pi2 = ai[2];` we are creating a memory leak (the area p_reviously allocated to `pi2`) and accessing freed memory. Then, when we `free(pi2);` (which now points to the original `pi1` memory area) we have a double free.

4 **+1.0**

The correct answer is to choose B, C and E.
When we assign `pi2 = pi1;` after freeing pi1, and then assign `*pi2 = ai[2];` we are creating a memory leak (the area p_reviously allocated to `pi2`) and accessing freed memory. Then, when we `free(pi2);` (which now points to the original `pi1` memory area) we have a double free.

There is no unallocated memory being accessed.

5 **+1.0**

The correct answer is to choose B, C and E.
When we assign `pi2 = pi1;` after freeing pi1, and then assign `*pi2 = ai[2];` we are creating a memory leak (the area p_reviously allocated to `pi2`) and accessing freed memory. Then, when we `free(pi2);` (which now points to the original `pi1` memory area) we have a double free.

There is no unallocated memory being freed.

✚ Add Rubric Item | Create Group | Import...

Q6 Glitch

8 points

Q6.1 Bugs

8 points

Rubric Settings

:. Bug 2 Line Number	d. Bug 2 Correction

1 **+0.0**

The three bugs were:

Line 6: `sizeof(pcSrc)` calculates the number of bytes of the pointer, which will not vary based on the string's length. The argument to `malloc` should have been `Str_getLength(pcSrc)` (or `strlen`). It would be fine to allocate 1 more byte than that for the `'\0'`, which fits our general pattern, however it is not necessary in this case since the extra byte will never be filled by the loop on lines 12-13.

Line 16: this compares the addresses of these two pointer variables, not their values (i.e., where they point). The correct loop sustaining condition should be `(pcSrc >= pcSrcStart)`.

Line 18: this attempts to `free` memory at an address that was not returned by `malloc`, because we have

that was not returned by `malloc`, because we have advanced `pcTemp` to the end of the string. Thus, we should use the variable meant to keep the place of the beginning of the string instead: `free(pcTempStart);`.

2 **+8.0**

Correct line numbers and corrections for both bugs.

3 **+4.0**

Correct bug #1 and correction.

4 **+3.0**

Correct bug #1. 2/3 partial credit for correction.

5 **+2.0**

Correct bug #1. 1/3 partial credit for correction.

6 **+1.0**

Correct bug #1. Incorrect correction.

7 **+4.0**

Correct bug #2 and correction.

8 **+3.0**

Correct bug #2. 2/3 partial credit for correction.

9 **+2.0**

Correct bug #2. 1/3 partial credit for correction.

0 **+1.0**

Correct bug #2. Incorrect correction.

+0.0

Incorrect or Blank line number and correction for other bug.

+0.0

Incorrect or Blank line numbers and corrections for both bugs.

+ Add Rubric Item | Create Group | Import...