! This class has been made inactive. No posts will be allowed until an instructor reactivates the class.

note @907 189 views

Actions

[Past Midterms] 2019

You can find the past exams here: https://cs61c.org/sp22/resources/exams/

When posting questions, please reference the semester, exam, and question in this format so it's easier for students and staff to search for similar questions:

Semester-Exam-Question Number

For example: **SP19-MT1-Q1**, or **FA19-MT2-Q3**

- Here's a video walkthrough by Daniel for the SP19 Final: https://www.youtube.com/watch?v=8DiN5Hu9x24&list=PLDoI-XvXO0apuEacxuUrUaBq2YDuKYPtV&index=2 (handout and timestamps in comments)
- Here's a video walkthrough for the SU19 Final made by Sunay.
  - Q1 Potpourri: https://youtu.be/FY5dAMrXvxo
  - Q2 FSM: https://youtu.be/gmHbw6LSeSw
  - Q3 C Coding: https://youtu.be/v4B1WTs5UNU
  - Q4 RISC-V: https://youtu.be/2VHjG-gy9Dk
  - Q5 Data-Level Parallelism: https://youtu.be/oG9Rrzmi0M4
  - Q6 RAID and ECC: https://youtu.be/rfCNTIzNZ2M
  - Q7 Caches: https://youtu.be/xojc8YZaO3Q
  - Q8 Spark: https://youtu.be/A37BFXRXmm0
  - Q9 Datapath: https://youtu.be/q-T4N3hBhUM
  - Q10 Digital Logic: https://youtu.be/3RI36lsDSg4
  - Q11 Virtual Memory: https://youtu.be/5_2fKsK4I34

exam exam/midterm

good note | 0

Updated 5 months ago by Jerry Xu and Caroline Liu

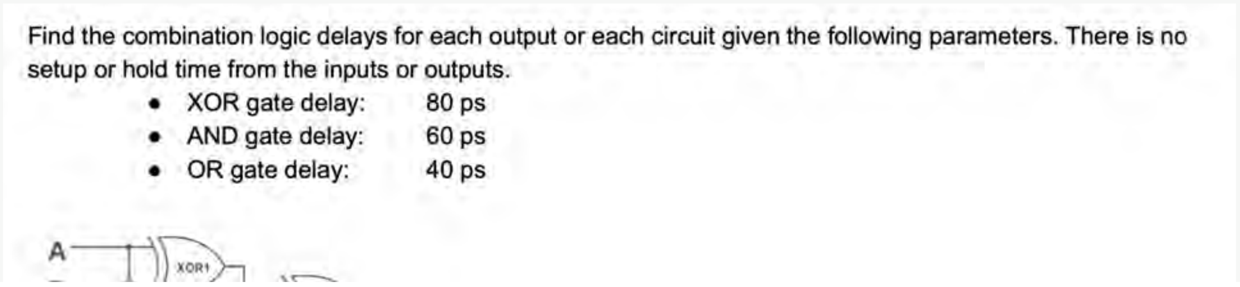
followup discussions, *for lingering questions and comments*

[✓ Resolved] Unresolved **@907_f1**

Anonymous Helix 7 months ago

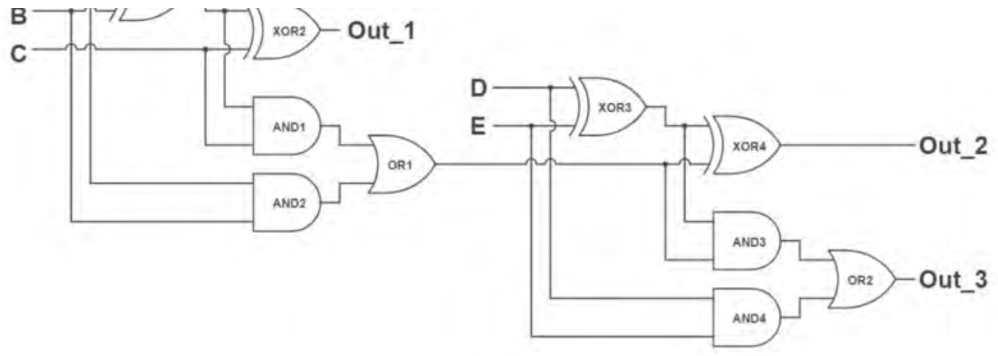
[su19-mt2-q3]

When we are calculating combination logic delay for out_2, do we need to still count for out_1? Can someone walk through how to get out_2 and out_3? Thanks!



Find the combination logic delays for each output or each circuit given the following parameters. There is no setup or hold time from the inputs or outputs.

- XOR gate delay: 80 ps
- AND gate delay: 60 ps
- OR gate delay: 40 ps



Out_1 Delay: 80ps + 80ps = 160ps

Out_2 Delay: 80ps + 60ps + 40ps + 80ps = 260ps

Out_3 Delay: 80ps + 60ps + 40ps + 60ps + 40ps = 280ps

helpful! 0

Adelson Chua 7 months ago
You trace the path from Out_2 to the inputs A, B, C, then pick which one is the longest.
You can also see from the diagram that the path to Out_2 is not connected to Out_1.
good comment 0

Reply to this followup discussion

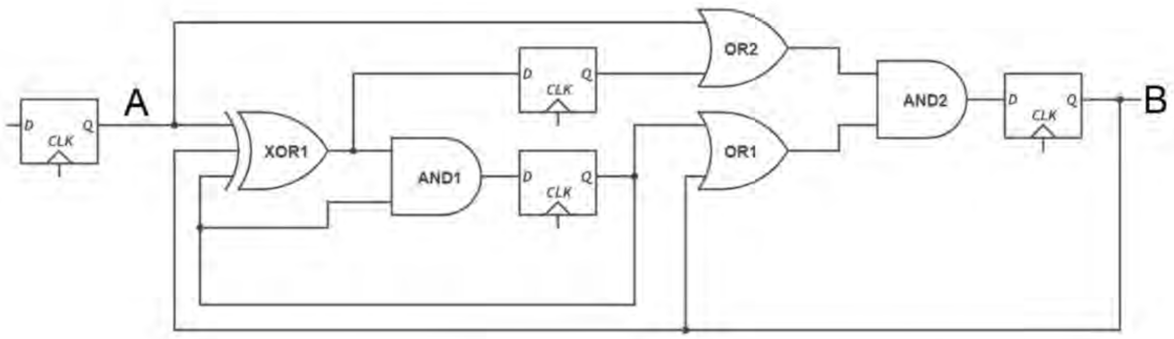
[✓ Resolved] Unresolved @907_f2

Anonymous Helix 7 months ago
[su19-mt2-q3]
Why the longest CL is just passing through AND and XOR? I'm thinking on the line of XOR1 + AND1 +

OR1 + AND2

For the next problems, consider the following pipelined circuit. Assume all registers have their clock inputs correctly connected to a global clock signal and that logic gates have the following parameters:

• XOR gate delay:	80 ps
• AND gate delay:	60 ps
• OR gate delay:	40 ps



When shopping for registers, we find two different models and want to determine which would be best for our circuit.

Register Type λ

• Setup Time:	40 ps
• Hold Time:	20 ps
• Clock-to-Q Delay:	30 ps

Register Type τ

• Setup Time:	10 ps
• Hold Time:	10 ps
• Clock-to-Q Delay:	80 ps

Critical Path = CLK_Q + XOR + AND + SETUP
Because this passes through 2 registers, our latency is 2 clock cycles.

Note after release we found 2 other interpretations to this question. 1 has just 1 critical path because it considers the latency to be just the top path A takes to B. The second also counts an extra clock to q to give A its value or propagate through the last register to B.

What is the minimum latency for the circuit from A to B if we use register type λ? 2 * (30ps + 80ps + 60ps + 40ps) = 420ps

helpful! 0

Adelson Chua 7 months ago
When you are calculating path delays, you start from the output of the register then stop at the input of a register. This is because the register only updates on the rising edge of the clock, so the register 'blocks' the delays once it arrives at its input port.

good comment 0

Anonymous Helix 7 months ago
Then why do we need to multiply it by 2? I'm still a bit confused by the comments in the solution

helpful! 0

Adelson Chua 7 months ago
Because we are trying to calculate the 'latency' of the output. Latency is defined as the amount of time we have to wait for the correct output to appear at the output port.
Because there are two registers that we have to pass through to properly update the output, the

latency is 2 clock cycles.

If we are using the minimum clock cycle time, then we need to consider two min clock cycle times for the correct output to appear.

good comment | 0

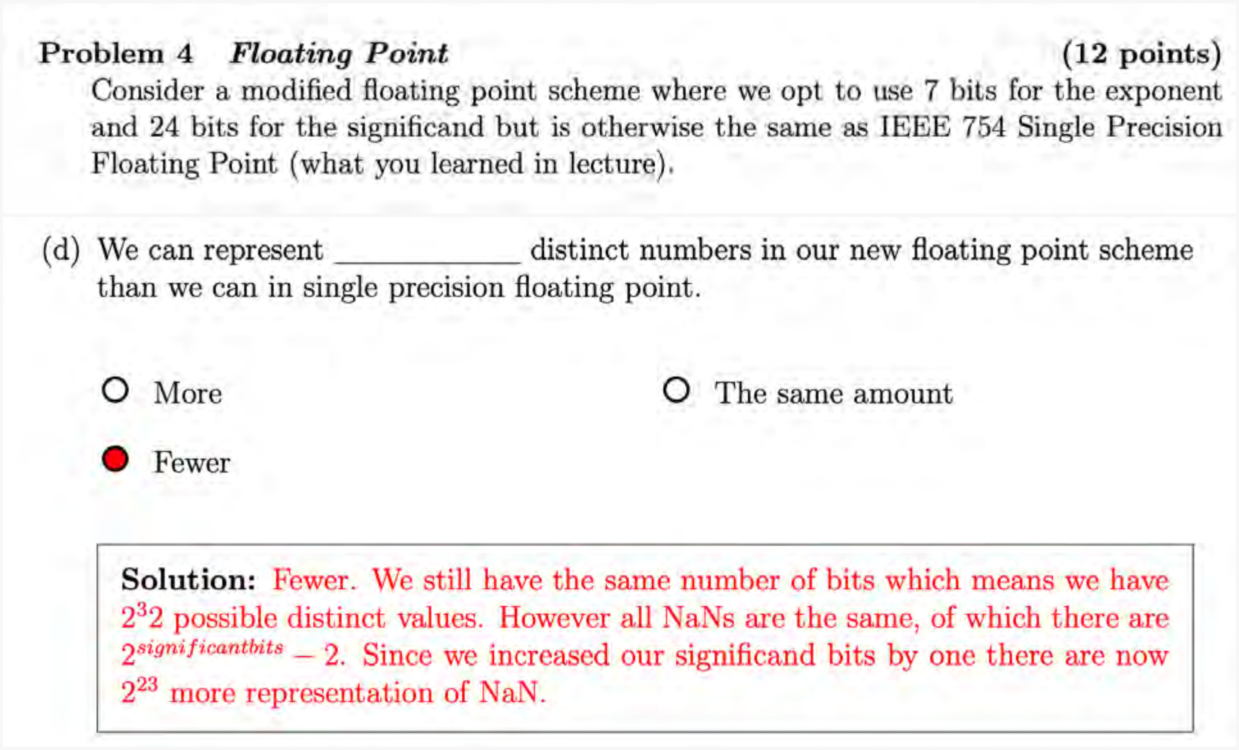
Reply to this followup discussion

[✓ Resolved] Unresolved @907_f3

Anonymous Helix 7 months ago

[sp19-mt2-q4.d]

Can someone explain how to interpret this solution? Thanks!



Problem 4 ***Floating Point*** **(12 points)**

Consider a modified floating point scheme where we opt to use 7 bits for the exponent and 24 bits for the significand but is otherwise the same as IEEE 754 Single Precision Floating Point (what you learned in lecture).

(d) We can represent ________ distinct numbers in our new floating point scheme than we can in single precision floating point.

More The same amount [✓ Fewer]

Solution: Fewer. We still have the same number of bits which means we have $2^{3}2$ possible distinct values. However all NaNs are the same, of which there are $2^{significantbits} - 2$. Since we increased our significand bits by one there are now 2^{23} more representation of NaN.

helpful! | 0

Adelson Chua 7 months ago

The number of combinations of exponent and mantissa remains the same.

However, as stated, increasing the number of significand/mantissa bits increases the number of NaNs that can be represented (review how NaNs are represented).

So the number of total combinations - the number of NaNs that can be represented decreases.

good comment | 0

Reply to this followup discussion

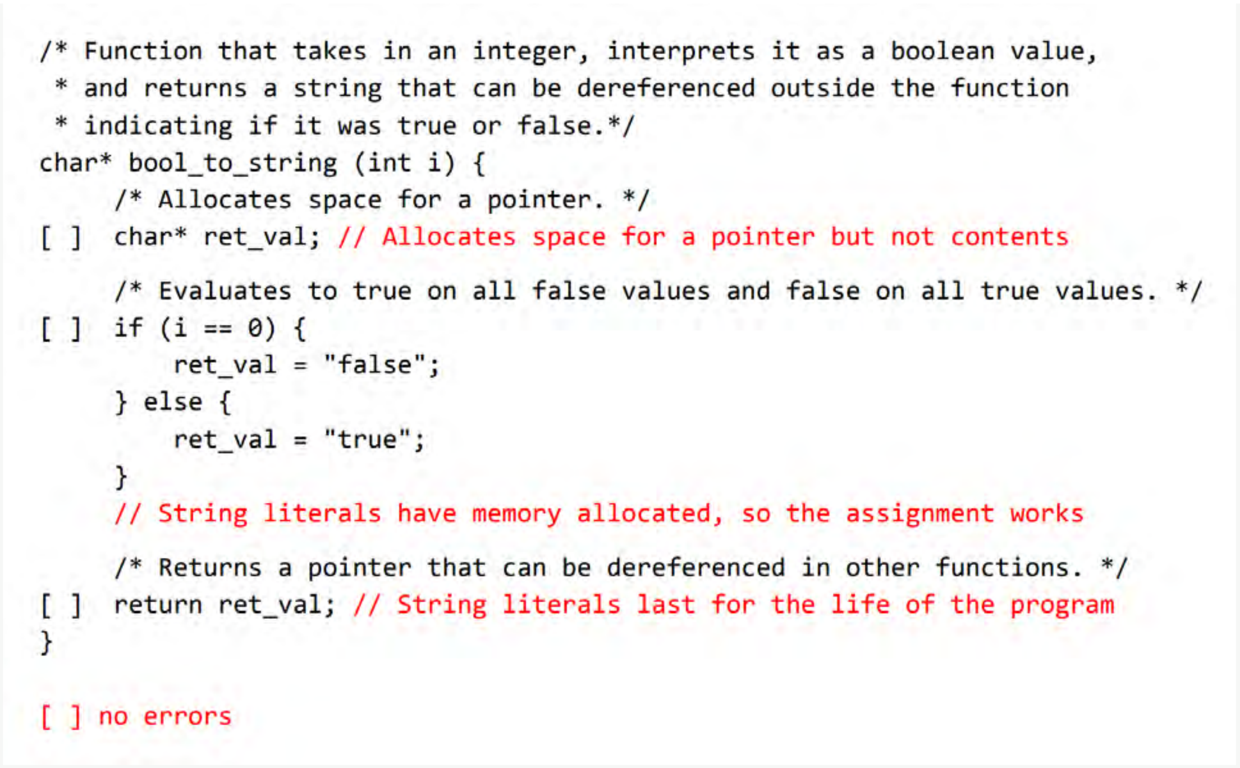
[✓ Resolved] Unresolved @907_f4

Anonymous Beaker 7 months ago

SU19-MT1-Q3.2

Is there an error in these solutions? The second comment and the code seem to mismatch; the code assigns false to false values and true to true values...

```
2.
```



```
/* Function that takes in an integer, interprets it as a boolean value,
 * and returns a string that can be dereferenced outside the function
 * indicating if it was true or false.*/
char* bool_to_string (int i) {
     /* Allocates space for a pointer. */
[ ]  char* ret_val; // Allocates space for a pointer but not contents

     /* Evaluates to true on all false values and false on all true values. */
[ ]  if (i == 0) {
         ret_val = "false";
     } else {
         ret_val = "true";
     }
     // String literals have memory allocated, so the assignment works

     /* Returns a pointer that can be dereferenced in other functions. */
[ ]  return ret_val; // String literals last for the life of the program
}

[ ] no errors
```

helpful! | 0

Anonymous Beaker 7 months ago

Also, is it possible to deference a pointer to a string literal without error?

helpful! | 0

Peyrin Kao 7 months ago

I think what that comment is saying is that `i == 0` evaluates to true on all false values and false on all true values, which is what you'd want in the case checking whether we want to return the string "false".

String literals are stay in static memory for the life of the program, so it's okay to dereference and read that memory. You just can't modify the string literal in static memory.

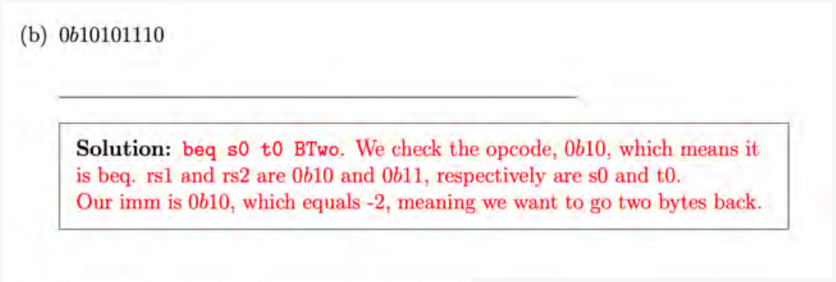
good comment | 0

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f5

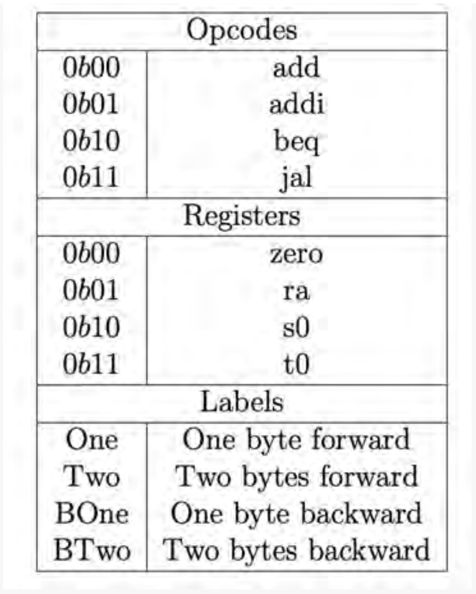
Anonymous Comp 7 months ago

SP19-MT1-Q5



(b) 0b10101110

Solution: beq s0 t0 BTwo. We check the opcode, 0b10, which means it is beq. rs1 and rs2 are 0b10 and 0b11, respectively are s0 and t0.
Our imm is 0b10, which equals -2, meaning we want to go two bytes back.



Opcodes	
0b00	add
0b01	addi
0b10	beq
0b11	jal
Registers	
0b00	zero
0b01	ra
0b10	s0
0b11	t0
Labels	
One	One byte forward
Two	Two bytes forward
BOne	One byte backward
BTwo	Two bytes backward

Shouldn't it be -4 bytes? Because we don't encode the last zero in the immediate for jumping and branching?

helpful! | 0

Anonymous Scale 7 months ago
Not an instructor but I don't think that concept applies here. The reason we don't encode the last zero in standard RISC-V is because all instructions are 4 bytes apart from each other. So we could omit the last two bytes but to support 16-bit CPUs, we retain one of the two possible omissions. In this problem, we're operating in like a 2-bit system so we can't really omit a 0.

helpful! | 0

Anonymous Scale 7 months ago
Edit: meant to say "last two bits**" not "last two bytes".

helpful! | 0

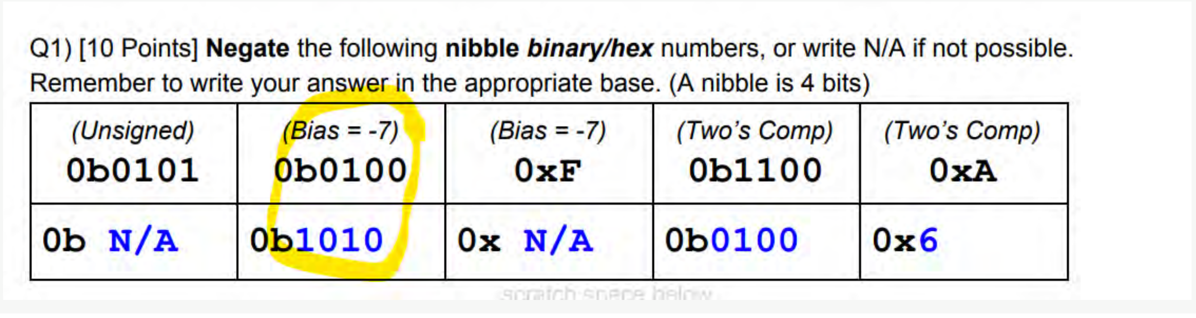
Adelson Chua 7 months ago
Yeah, the exclusion of bit 0 is specific to RISC-V. It doesn't apply here where we are technically defining a new instruction encoding.

good comment | 0

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f6

Anonymous Atom 7 months ago



Q1) [10 Points] **Negate** the following **nibble *binary/hex*** numbers, or write N/A if not possible. Remember to write your answer in the appropriate base. (A nibble is 4 bits)

(Unsigned) 0b0101	*(Bias = -7)* 0b0100	*(Bias = -7)* 0xF	*(Two's Comp)* 0b1100	*(Two's Comp)* 0xA
0b N/A	0b1010	0x N/A	0b0100	0x6

scratch space below

The answer key got to this answer by getting -3 and negating this, but I thought we subtract the bias to put the number in biased notation? Doesn't this mean we should add 7 to 4 (0b0100) and then negate 11? How do we know if we're starting with the biased or unbiased notation?

helpful! | 0

Anonymous Atom 7 months ago

[FA19-QUEST-Q1] ^

helpful! | 0

Adelson Chua 7 months ago

@106

biased encoding + bias = unbiased encoding.

The given 0100 is already in biased form.

0100 (4) + -7 = -3

-3 negated is 3.

unbiased encoding - bias = biased encoding

3 - (-7) = 10 => 1010

Typically, when you are given a binary representation, that's in biased encoding already.

Biased encoding is always unsigned. If you are given a number that was negative, that means it is in unbiased encoding.

Just like in the floating-point representation, the exponent is in biased notation. Exponent field-127 = actual exponent.

The exponent field is typically written as bits, that's in biased encoding. The actual exponent can be negative right? That's the unbiased form.

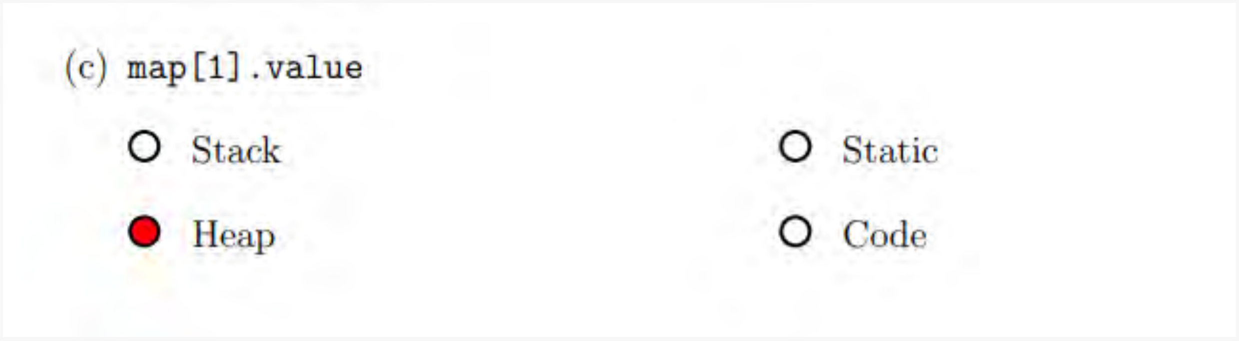
So it follows the formula biased encoding + bias = unbiased encoding.

good comment | 0

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f7

Anonymous Mouse 7 months ago


(c) `map[1].value`

○ Stack [✓ Heap] ○ Static ○ Code

SP19-MT1-Q2

helpful! | 0

Peyrin Kao 7 months ago

`struct mapentry *map = malloc(sizeof(struct mapentry) * 10);` allocates space for all the structs on the heap, and `map[1].value` is a field in one of the structs.

good comment | 0

Reply to this followup discussion

Resolved Unresolved @907_f8

Anonymous Mouse 7 months ago

For SP19-MT1-Q2 part e, how do we approach these types of problems do we have to draw out the stack/heap to understand the comparisons?

helpful! | 0

Peyrin Kao 7 months ago

We allocated space for 10 structs, and each struct is 8 bytes each, for a total of 80 bytes. Then we allocated space for 10 characters (each character is 1 byte). We didn't free any of this space, so we're leaking 90 bytes in total.

good comment | 0

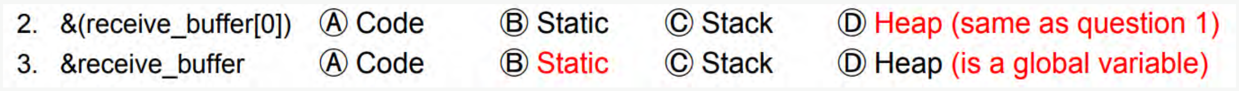
Reply to this followup discussion

Resolved Unresolved @907_f9

Anonymous Calc 7 months ago

SP19-MT1-Q2

Could someone explain how number 2.2 is heap and 2.3 is static?



2.	&(receive_buffer[0])	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (same as question 1)
3.	&receive_buffer	Ⓐ Code	Ⓑ Static	Ⓒ Stack	Ⓓ Heap (is a global variable)

helpful! | 0

Peyrin Kao 7 months ago

I think this is SU19, not SP19.

The pointer `receive_buffer` is defined outside of a function, so `&receive_buffer` is in static memory. The pointer `receive_buffer` points into the heap, so if we dereference it with `receive_buffer[0]`, we get some data on the heap. The address of this data on the heap `&(receive_buffer[0])` is on the heap.

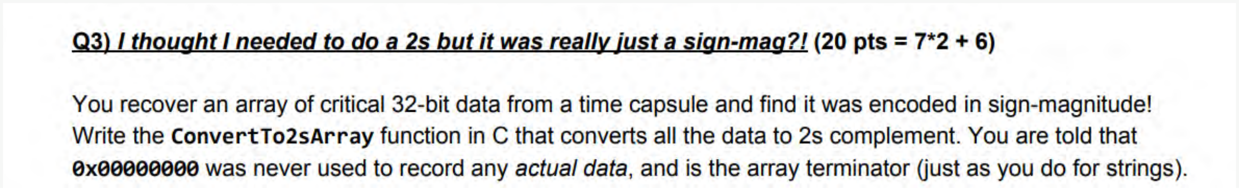
good comment | 0

Reply to this followup discussion

Resolved Unresolved @907_f10

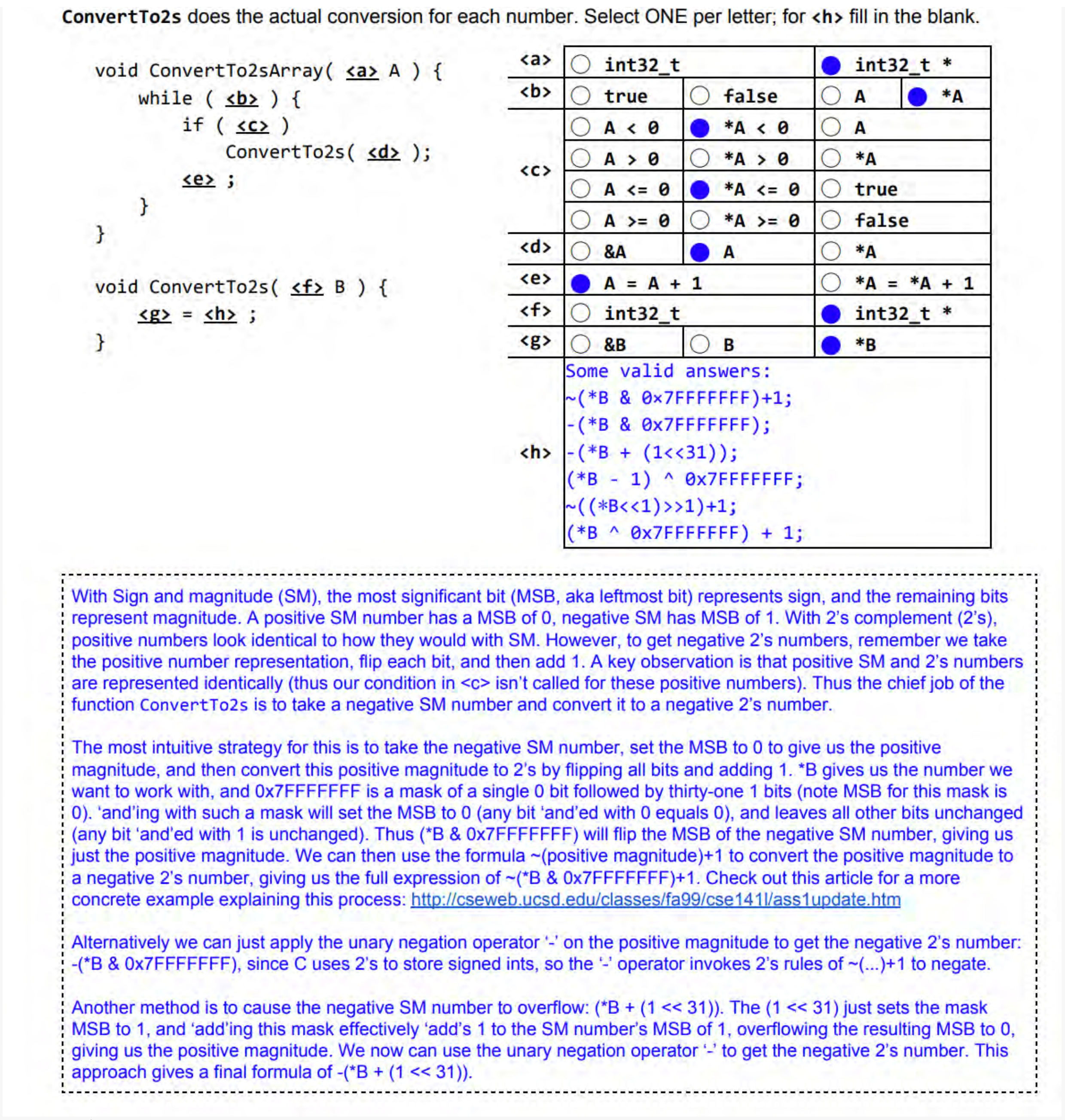
Anonymous Gear 7 months ago

We used hex digits in h), so I was wondering if we were dealing with a binary array instead, would we use binary digits or would we still use binary converted to hex?



Q3) I thought I needed to do a 2s but it was really just a sign-mag?! **(20 pts = 7*2 + 6)**

You recover an array of critical 32-bit data from a time capsule and find it was encoded in sign-magnitude! Write the **`ConvertTo2sArray`** function in C that converts all the data to 2s complement. You are told that **`0x00000000`** was never used to record any *actual data*, and is the array terminator (just as you do for strings).



ConvertTo2s does the actual conversion for each number. Select ONE per letter; for **<h>** fill in the blank.

```
void ConvertTo2sArray( <a> A ) {
    while ( <b> ) {
        if ( <c> )
            ConvertTo2s( <d> );
        <e> ;
    }
}

void ConvertTo2s( <f> B ) {
    <g> = <h> ;
}
```

<a>	int32_t		[✓ int32_t *]	
<b>	true	false	A	[✓ *A]
<c>	A < 0	[✓ *A < 0]	A	
	A > 0	*A > 0	*A	
	A <= 0	[✓ *A <= 0]	true	
	A >= 0	*A >= 0	false	
<d>	&A	[✓ A]	*A	
<e>	[✓ A = A + 1]		*A = *A + 1	
<f>	int32_t		[✓ int32_t *]	
<g>	&B	B	[✓ *B]	

<h>	Some valid answers:
	~(*B & 0x7FFFFFFF)+1;
	-(*B & 0x7FFFFFFF);
	-(*B + (1<<31));
	(*B - 1) ^ 0x7FFFFFFF;
	~((*B<<1)>>1)+1;
	(*B ^ 0x7FFFFFFF) + 1;			

With Sign and magnitude (SM), the most significant bit (MSB, aka leftmost bit) represents sign, and the remaining bits represent magnitude. A positive SM number has a MSB of 0, negative SM has MSB of 1. With 2's complement (2's), positive numbers look identical to how they would with SM. However, to get negative 2's numbers, remember we take the positive number representation, flip each bit, and then add 1. A key observation is that positive SM and 2's numbers are represented identically (thus our condition in <c> isn't called for these positive numbers). Thus the chief job of the function `ConvertTo2s` is to take a negative SM number and convert it to a negative 2's number.

The most intuitive strategy for this is to take the negative SM number, set the MSB to 0 to give us the positive magnitude, and then convert this positive magnitude to 2's by flipping all bits and adding 1. *B gives us the number we want to work with, and 0x7FFFFFFF is a mask of a single 0 bit followed by thirty-one 1 bits (note MSB for this mask is 0). 'and'ing with such a mask will set the MSB to 0 (any bit 'and'ed with 0 equals 0), and leaves all other bits unchanged (any bit 'and'ed with 1 is unchanged). Thus (*B & 0x7FFFFFFF) will flip the MSB of the negative SM number, giving us just the positive magnitude. We can then use the formula ~(positive magnitude)+1 to convert the positive magnitude to a negative 2's number, giving us the full expression of ~(*B & 0x7FFFFFFF)+1. Check out this article for a more concrete example explaining this process: http://cseweb.ucsd.edu/classes/fa99/cse141l/ass1update.htm

Alternatively we can just apply the unary negation operator '-' on the positive magnitude to get the negative 2's number: -(*B & 0x7FFFFFFF), since C uses 2's to store signed ints, so the '-' operator invokes 2's rules of ~(...)+1 to negate.

Another method is to cause the negative SM number to overflow: (*B + (1 << 31)). The (1 << 31) just sets the mask MSB to 1, and 'add'ing this mask effectively 'add's 1 to the SM number's MSB of 1, overflowing the resulting MSB to 0, giving us the positive magnitude. We now can use the unary negation operator '-' to get the negative 2's number. This approach gives a final formula of -(*B + (1 << 31)).

helpful! 0

Adelson Chua 7 months ago

Binary and hex represent the same thing. Hex is just a 'shorthand' version of binary.

It's like saying 'one' and '1', they're the same, just being written differently.

good comment 0

Anonymous Gear 7 months ago

So if I did use binary instead of hex, would it still be correct then?

helpful! 0

Adelson Chua 7 months ago

Yes, it's just not good to look at, especially if there's too many bits involved.

Also, writing in binary is more prone to a careless mistake. Miss a 1 or a 0, you're wrong.

good comment 0

Peyrin Kao 7 months ago
On an exam, we'll usually specify whether we want an answer in binary or hex.

good comment 0

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f11

Anonymous Poet 7 months ago
For SP19-MT1-Q3, can we assume that the a register arguments at the beginning of SearchAST are in the same order as the C SearchAST?

helpful! 0

Peyrin Kao 7 months ago
Yeah, that seems like a safe assumption because the code says "Arguments follow the RISC-V calling convention."

good comment 0

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f12

Anonymous Helix 2 7 months ago

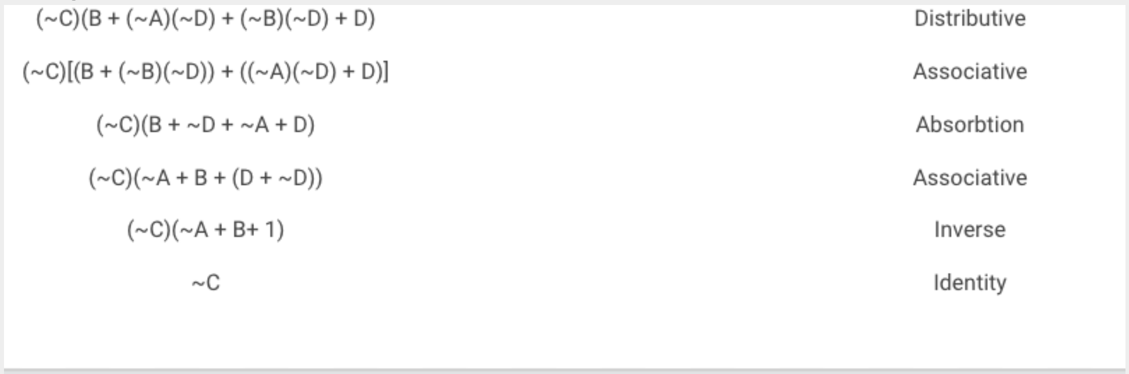
can someone explain to me Absorption law??

helpful! 0

Adelson Chua 7 months ago
Sounds like Boolean logic. Can you post the expression? This should be easily done through Boolean algebra.

good comment 0

Anonymous Helix 2 7 months ago



(~C)(B + (~A)(~D) + (~B)(~D) + D)	Distributive
(~C)[(B + (~B)(~D)) + ((~A)(~D) + D)]	Associative
(~C)(B + ~D + ~A + D)	Absorbtion
(~C)(~A + B + (D + ~D))	Associative
(~C)(~A + B+ 1)	Inverse
~C	Identity

can you just explain what absorbtion do?

helpful! 0

Adelson Chua 7 months ago
I... don't know what you want me to explain actually.

What I see is a complex Boolean logic and then some Boolean algebra is applied then it got simplified.

What actually is your question?

good comment 0

Anonymous Helix 2 7 months ago

nvm

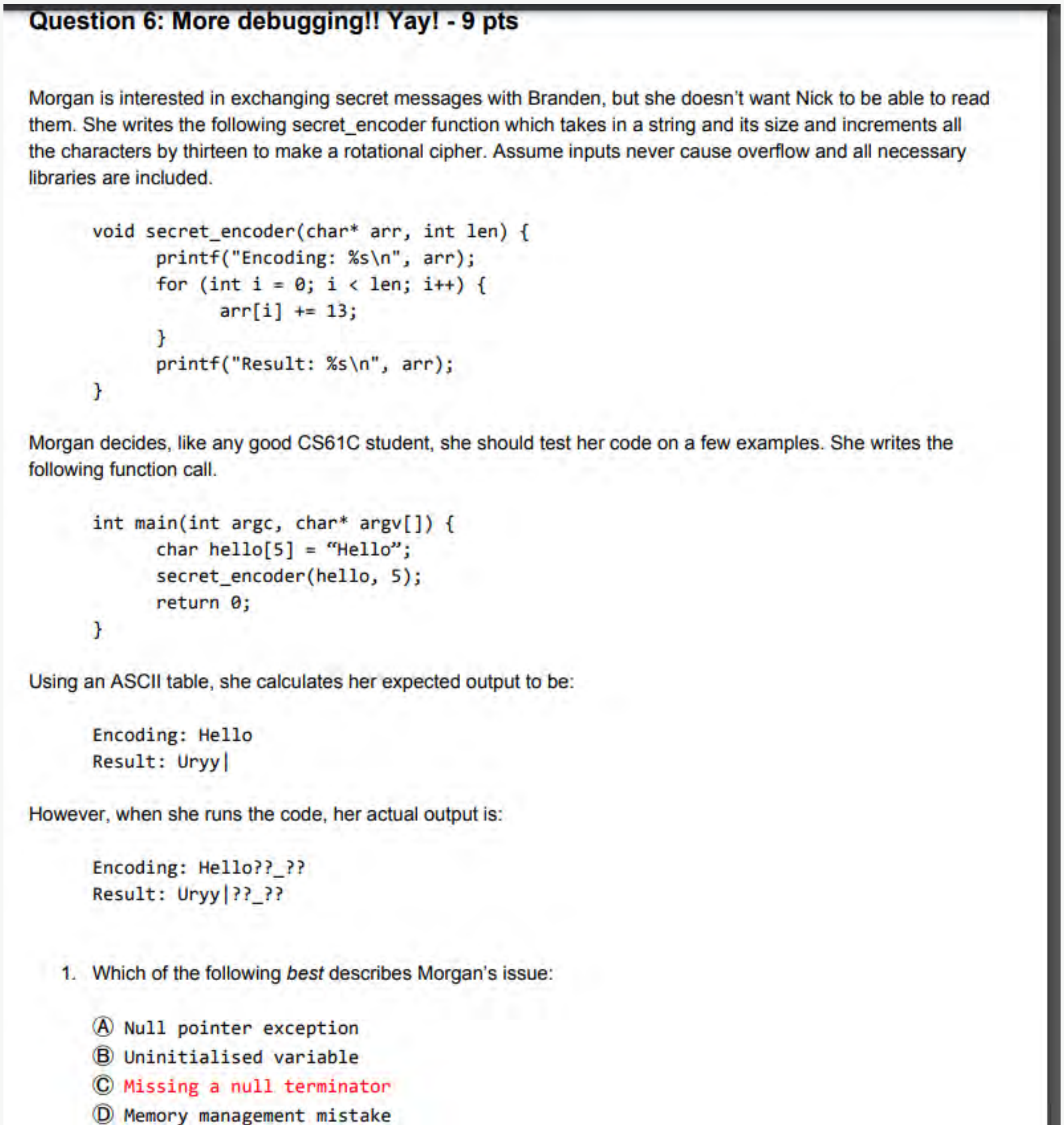
helpful! 0

Reply to this followup discussion

[✓ Resolved] Unresolved **@907_f13**

Anonymous Beaker 2 7 months ago

SU19-MT1-Q6 b)



Question 6: More debugging!! Yay! - 9 pts

Morgan is interested in exchanging secret messages with Branden, but she doesn't want Nick to be able to read them. She writes the following secret_encoder function which takes in a string and its size and increments all the characters by thirteen to make a rotational cipher. Assume inputs never cause overflow and all necessary libraries are included.

```
void secret_encoder(char* arr, int len) {
      printf("Encoding: %s\n", arr);
      for (int i = 0; i < len; i++) {
            arr[i] += 13;
      }
      printf("Result: %s\n", arr);
}
```

Morgan decides, like any good CS61C student, she should test her code on a few examples. She writes the following function call.

```
int main(int argc, char* argv[]) {
      char hello[5] = "Hello";
      secret_encoder(hello, 5);
      return 0;
}
```

Using an ASCII table, she calculates her expected output to be:

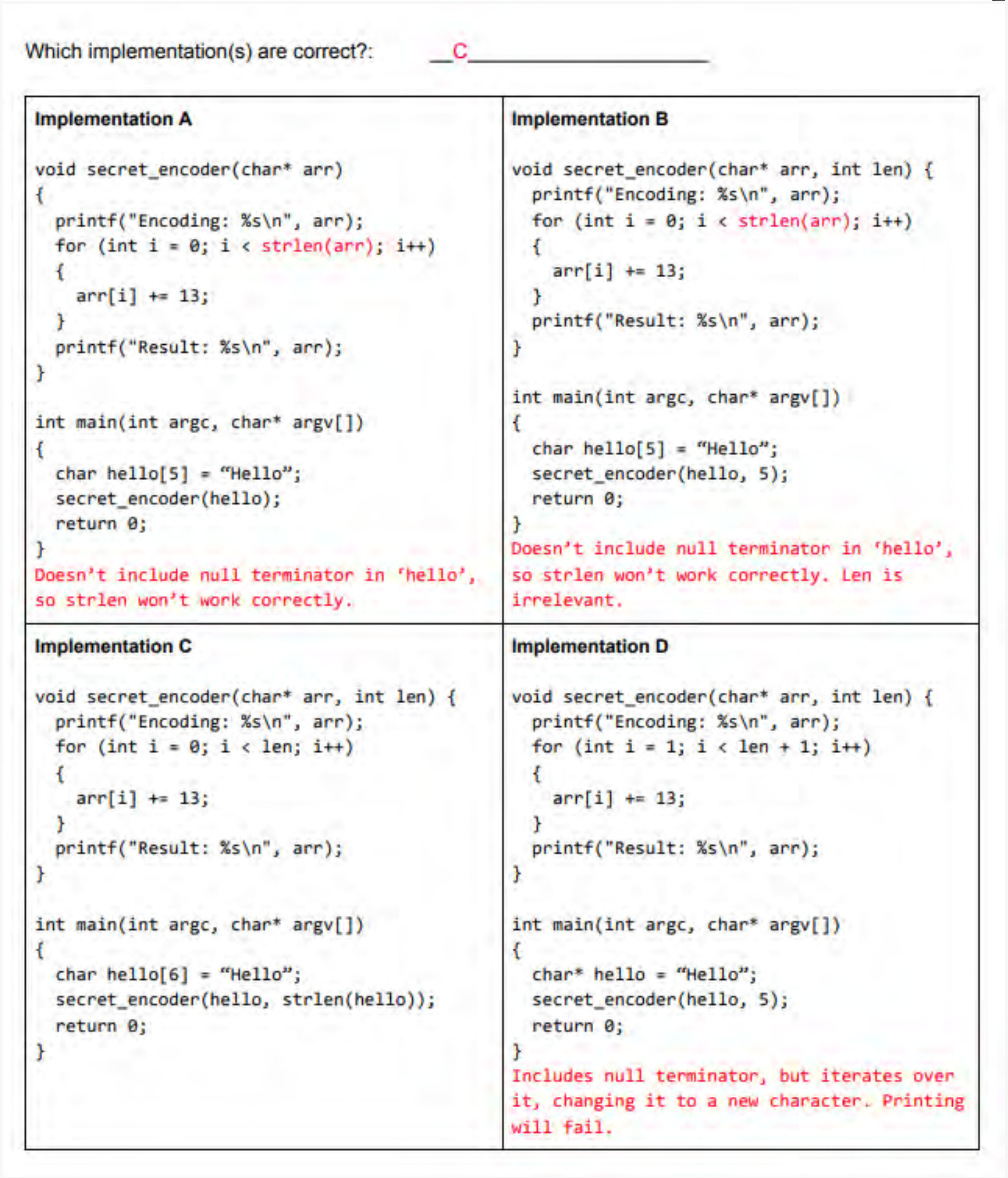
```
Encoding: Hello
Result: Uryy|
```

However, when she runs the code, her actual output is:

```
Encoding: Hello??_??
Result: Uryy|??_??
```

1. Which of the following *best* describes Morgan's issue:

   Ⓐ Null pointer exception
   Ⓑ Uninitialised variable
   [✓ Ⓒ Missing a null terminator]
   Ⓓ Memory management mistake



Which implementation(s) are correct?: __C________________

Implementation A

```
void secret_encoder(char* arr)
{
  printf("Encoding: %s\n", arr);
  for (int i = 0; i < strlen(arr); i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char hello[5] = “Hello”;
  secret_encoder(hello);
  return 0;
}
Doesn’t include null terminator in ‘hello’,
so strlen won’t work correctly.
```

Implementation B

```
void secret_encoder(char* arr, int len) {
  printf("Encoding: %s\n", arr);
  for (int i = 0; i < strlen(arr); i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char hello[5] = “Hello”;
  secret_encoder(hello, 5);
  return 0;
}
Doesn’t include null terminator in ‘hello’,
so strlen won’t work correctly. Len is
irrelevant.
```

Implementation C

```
void secret_encoder(char* arr, int len) {
  printf("Encoding: %s\n", arr);
  for (int i = 0; i < len; i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char hello[6] = “Hello”;
  secret_encoder(hello, strlen(hello));
  return 0;
}
```

Implementation D

```
void secret_encoder(char* arr, int len) {
  printf("Encoding: %s\n", arr);
  for (int i = 1; i < len + 1; i++)
  {
    arr[i] += 13;
  }
  printf("Result: %s\n", arr);
}

int main(int argc, char* argv[])
{
  char* hello = “Hello”;
  secret_encoder(hello, 5);
  return 0;
}
Includes null terminator, but iterates over
it, changing it to a new character. Printing
will fail.
```

I get why the null terminator problem is fixed in Implementation C, but I don't get why there isn't a memory management issue as well.

When we pass in hello to secret_encoder, aren't we passing in a pointer to a location on the stack, since hello[6] is on the stack? I know we aren't supposed to return pointers to things on the stack and assumed that we also could not pass in pointers to things on the stack to functions.

Is this thinking wrong?

helpful! | 0

i **Rosalie Fang** 7 months ago

That's a great question! The reason we usually try not to pass in a pointer on the stack is that once that stack space closes, the space in that pointer could that overwritten and therefore change our result. However, in this case, the variable "char hello[6]" will be written to the stack within main's stack frame. Before this stack frame closes, secret_encoder is called and finishes running. So, in the entire time that this pointer is used, it's in a secure stack space, and we don't need to worry about it being overwritten.

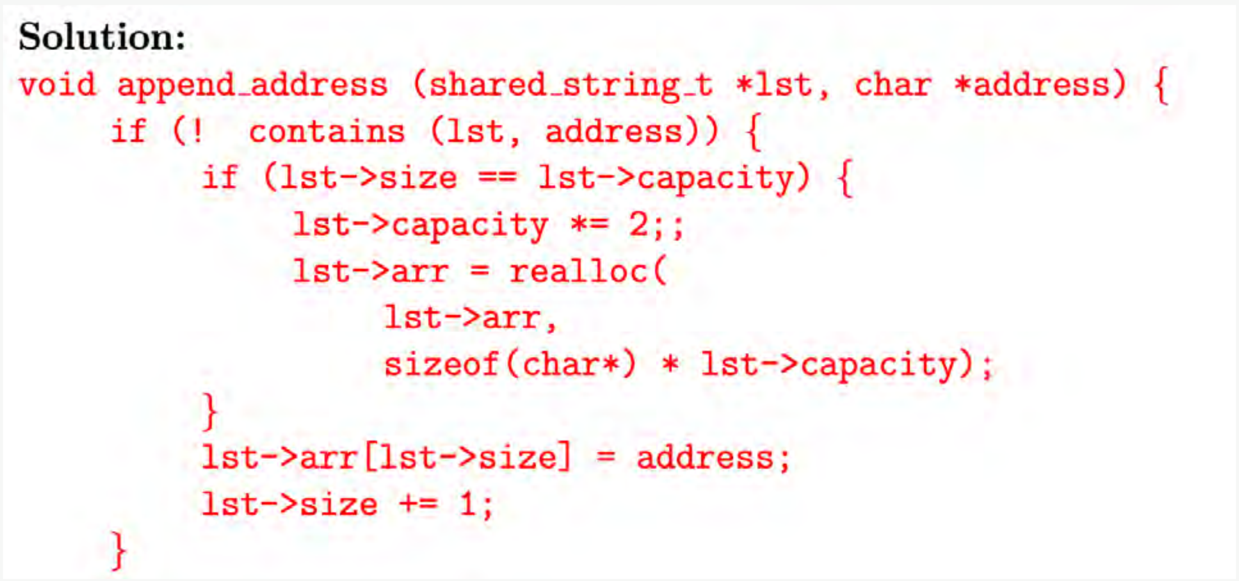
good comment | 1

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f14

Anonymous Comp 2 7 months ago
sp19-mt1-Q1

what is size and capacity represent?
why are we multiplying by 2 for lst->capacity?



Solution:

```
void append_address (shared_string_t *lst, char *address) {
    if (!  contains (lst, address)) {
        if (lst->size == lst->capacity) {
            lst->capacity *= 2;;
            lst->arr = realloc(
                lst->arr,
                sizeof(char*) * lst->capacity);
        }
        lst->arr[lst->size] = address;
        lst->size += 1;
    }
```

helpful! | 0

Adelson Chua 7 months ago
Capacity is the size of the currently allocated memory for arr.
Size is the current size of the filled entries within arr.

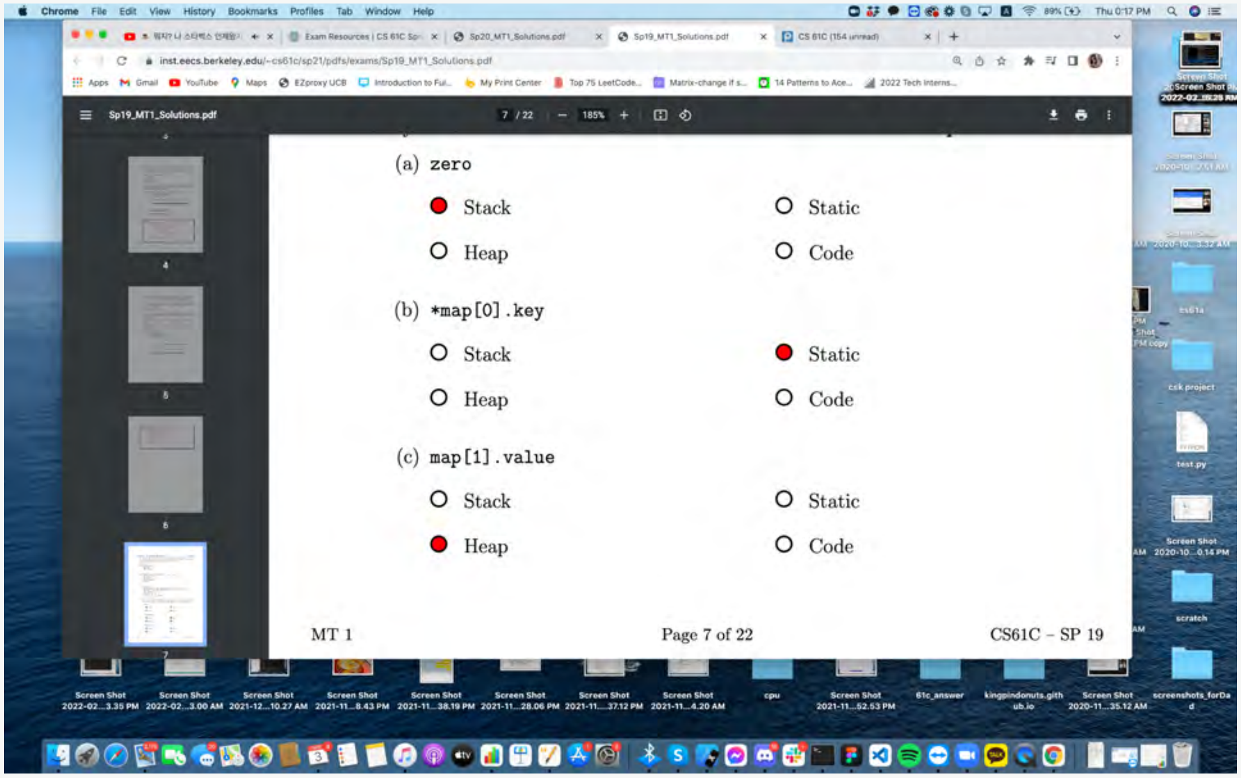
When the filled elements (size) = the size of allocated memory (capacity), the code reallocates more memory to store more strings. The reallocation just doubles the size every time the limit is reached.
Not sure if there's a specific reason regarding the doubling of size, though.

good comment | 1

Reply to this followup discussion

[✓ Resolved] Unresolved @907_f15

YoonWha Won 7 months ago



(a) `zero`

[✓ Stack] Static

Heap Code

(b) `*map[0].key`

Stack [✓ Static]

Heap Code

(c) `map[1].value`

Stack Static

[✓ Heap] Code

MT 1 Page 7 of 22 CS61C – SP 19

sp19-mt1-2(a)

why map[1].value is in heap instead of stack? i thought since value is defined as an array it's basically a pointer in the stack.
Also, I'm struggling to understand the difference between *map[0] in (b) and map[1] in (c). What difference does the * sign in front of map makes?

Thanks

helpful! | 0

Peyrin Kao 7 months ago

`map` is a pointer to an array of structs all allocated in the heap. `map[1].value` is a field in one of those structs, so it's on the heap. This is different from `value[20]`, which is on the stack but a completely different variable (that just happens to have the same name).

`map[1].value` gives you the `value` field in the struct, which is a pointer stored in the heap. `*map[0].key` dereferences the `key` field in the struct, which dereferences the pointer and goes to the actual string in static memory.

good comment | 0

Caroline Liu 7 months ago

`map[1].value` is in heap because we dynamically allocated a chunk of memory for our two pointers; those two pointers reside in heap, but their **values** can reside elsewhere (static, stack, etc…) so if we had done `*map[1].value`, then our answer would've been stack since we have `char value[20]` declared on the stack.

The difference between the star versus not having the star is that the star **dereferences** the pointer. That means, we access the value stored in there. `*map[0].key` accesses the `"k"` that is stored at the memory address (pointer) provided by `key` for the first `map` entry. That value is stored in static memory because it is a `char*`, AKA a pointer to a string stored in static memory.

The second is looking at the variable itself, and not examining the contents.

good comment | 0

i **Caroline Liu** 7 months ago
Darn… LaTeX-ing made Peyrin beat me to it.

good comment | 0

Reply to this followup discussion

Start a new followup discussion

Compose a new followup discussion