

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address <EMAILADDRESS>. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- You must choose either this option
- Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- You could select this choice.
- You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

Please complete and submit these questions before the exam starts.

- (a) What is your full name?

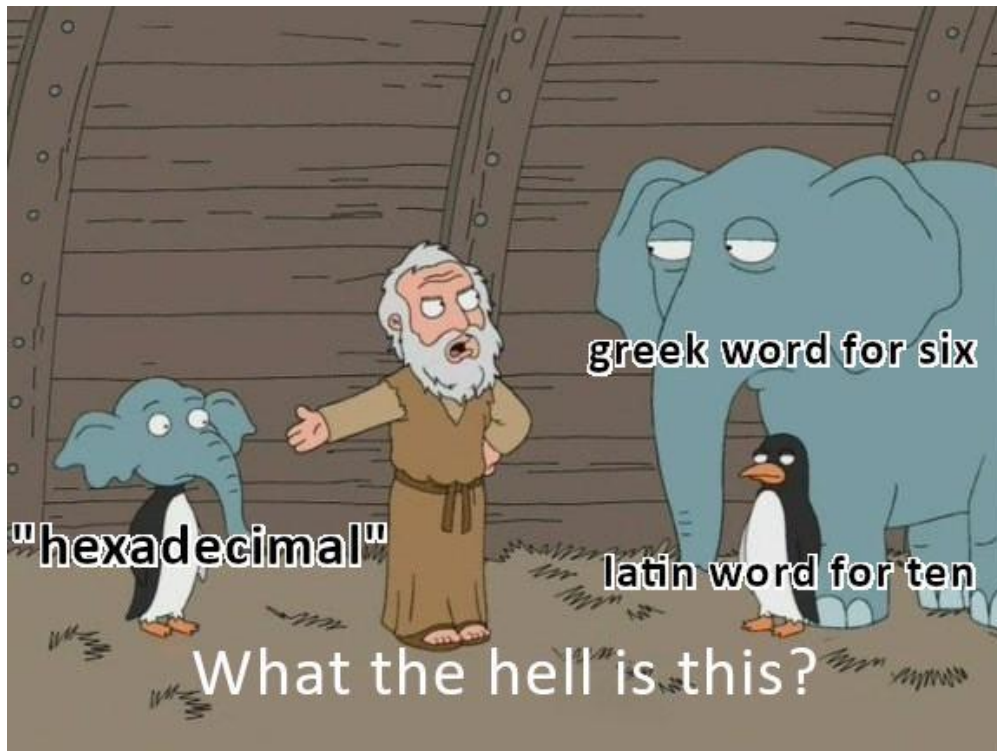
Solutions

- (b) What is your student ID number?

dQw4w9WgXcQ (This is a YouTube video)

- (c) If an answer requires hex input, make sure you only use capitalized letters! For example, 0xDEADBEEF instead of 0xdeadbeef. You will be graded incorrectly otherwise! Please always add the hex (0x) and binary (0b) prefix to your answers or you will receive 0 points. For all other bases, do not add the suffix or prefixes.

Some of the questions may use images to describe a problem. If the image is too small, you can click and drag the image to a new tab to see the full image. You can also right click the image and download it or copy its address to view it better. You can use the image below to try this. You can also click the star by the question if you would like to go back to it (it will show up on the side bar). In addition, you are able see a check mark for questions you have fully entered in the sidebar. Questions will auto submit about 5 seconds after you click off of them, though we still recommend you click the save button.



Good luck!

1. Number Fun

(a) Please fill out the following table. Write exactly “N/A” if the conversion is not possible. Some entries have already been filled out for you. You may assume all binary numbers are 8 bits. If you are writing your answer in hex or binary, make sure to include its prefix; you will not get credit if you forget! Also, do not include the suffix for any representation, i.e. for decimal, base 4, and base 8, just put in the raw number. (For example, if the answer for base 4 is 3210_4 , just enter 3210). Please include all necessary leading zeros for any base other than decimal. You must fully simplify your answers.

i. Convert 3210_4 in Two’s Complement to...

A. (0.5 pt) Decimal

-28

B. (0.5 pt) Binary (Two’s Complement)

0b11100100

C. (0.5 pt) Hex (Two’s Complement)

0xE4

D. (0.5 pt) Base 8 (Two’s Complement)

344

E.

F. (0.5 pt) Binary (Biased w/ added bias of -129)

0b01100101

G. (0.5 pt) Binary (Biased w/ added bias of -127)

0b01100011

ii. Convert -42_{10} to...

A. (0.5 pt) Binary (Two's Complement)

0b11010110

B. (0.5 pt) Hex (Two's Complement)

0xD6

C. (0.5 pt) Base 4 (Two's Complement)

3112

D. (0.5 pt) Base 8 (Two's Complement)

326

E.

F. (0.5 pt) Binary (Biased w/ added bias of -129)

0b01010111

G. (0.5 pt) Binary (Biased w/ added bias of -127)

0b01010101

iii. Convert 0x3C (Two's Complement) to...

A. (0.5 pt) Decimal

60

B. (0.5 pt) Binary (Two's Complement)

0b00111100

C. (0.5 pt) Base 4 (Two's Complement)

0330

D. (0.5 pt) Base 8 (Two's Complement)

074

E.

F. (0.5 pt) Binary (Biased w/ added bias of -129)

0b10111101

G. (0.5 pt) Binary (Biased w/ added bias of -127)

0b10111011

iv. Convert 88_{10} to...

A. (0.5 pt) Binary (Two's Complement)

0b01011000

B. (0.5 pt) Hex (Two's Complement)

0x58

C. (0.5 pt) Base 4 (Two's Complement)

1120

D. (0.5 pt) Base 8 (Two's Complement)

130

E.

F. (0.5 pt) Binary (Biased w/ added bias of -129)

0b11011001

G. (0.5 pt) Binary (Biased w/ added bias of -127)

0b11010111

v. Convert 225_8 (Two's Complement) to...

A. (0.5 pt) Decimal

-107

B. (0.5 pt) Binary (Two's Complement)

0b10010101

C. (0.5 pt) Hex (Two's Complement)

0x95

D. (0.5 pt) Base 4 (Two's Complement)

2111

E.

F. (0.5 pt) Binary (Biased w/ added bias of -129)

0b00010110

G. (0.5 pt) Binary (Biased w/ added bias of -127)

0b00010100

vi. Convert 172_8 (Two's Complement) to...

A. (0.5 pt) Decimal

122

B. (0.5 pt) Binary (Two's Complement)

0b01111010

C. (0.5 pt) Hex (Two's Complement)

0x7A

D. (0.5 pt) Base 4 (Two's Complement)

1322

E.

F. (0.5 pt) Binary (Biased w/ added bias of -129)

0b11111011

G. (0.5 pt) Binary (Biased w/ added bias of -127)

0b11111001

(b) Please answer the questions below, assume we are working with n bits.

i. **A. (1.0 pt)** It is possible to represent the same range of numbers with unsigned and 2's complement.

True

False

B. (1.0 pt) It is possible to represent the same range of numbers with unsigned and biased.

True

False

C. (1.0 pt) It is possible to represent the same range of numbers with biased and 2's complement.

True

False

D. (1.0 pt) It is possible to represent the same range of numbers with 1's complement and bias.

True

False

- ii. **A. (1.0 pt)** If I shift the mantissa bits of a floating-point number to the right by k bits, while keeping all other fields unchanged, it will divide the overall number by 2^k .
- True
- False
- B. (1.0 pt)** If I shift the exponent bits of a floating-point number to the right by k bits, while keeping all other fields unchanged, it will divide the overall number by 2^k .
- True
- False
- C. (1.0 pt)** $2 + 2$ can equal fish under the correct representation.
- True
- False

- (c) i. For this question, assume that we are using 8-bit numbers! Make sure you fully simplify your answers. Note these problems are in numerical terms, not in terms of magnitude.

A. (1.0 pt) What is the distance between the **largest** number in 2's complement and the **largest** number in Sign and Magnitude?

0

Largest 8-bit 2's: 127 Largest 8-bit sign: 127 $127 - 127 = 0$

B. (1.0 pt) What is the distance between the **smallest** number in 2's complement and the **smallest** number in Sign and Magnitude?

1

Smallest 8-bit 2's: -128 Smallest 8-bit sign: -127 $-128 + 127 = 1$

- ii. (1.0 pt) What is the distance between the **largest** number in 2's complement and the **largest** number in unsigned?

128

Largest 8-bit Unsigned: 255 Largest 8-bit 2's: 127 $255 - 127 = 128$

(d) Select all which is true for the following statements.

i. (1.0 pt) Which of the following interpretations allows for multiple different bit sequences to map to the same underlying value?

- Unsigned
- Sign and Magnitude
- One's Complement
- Biased (for at least 1 choice of bias)
- Two's Complement
- Floating Point
- None of the other options

ii. (1.0 pt) Which of the following interpretations allows us to deduce the sign just by looking at the most significant bit? (Ignore 0)

- Sign and Magnitude
- One's Complement
- Biased (for at least 1 choice of bias)
- Two's Complement
- Floating Point
- None of the other options

- (e) i. (1.0 pt) How many numbers are written the same way in 32-bit 2's complement and IEEE-754 single-precision floating point (32 bit)?

1

- ii. (1.0 pt) How many numbers are written the same way in 32-bit Sign and Magnitude and IEEE-754 single-precision floating point (32 bit)?

2

- iii. (1.0 pt) How many numbers are written the same way in 32-bit unsigned and IEEE-754 single-precision floating point (32 bit)?

1

(f) Does the resulting operation overflow given 6-bit, Two's Complement numbers?

i. (0.5 pt) $0b100010 + 0b011110$

- Correct
 Overflow

ii. (0.5 pt) $0b111110 + 0b100010$

- Correct
 Overflow

iii. (0.5 pt) $0b011111 + 0b000001$

- Correct
 Overflow

iv. (0.5 pt) $0b001111 + 0b001111$

- Correct
 Overflow

v. (0.5 pt) $0b010001 + 0b001111$

- Correct
 Overflow

vi. (0.5 pt) $0b101010 + 0b010101$

- Correct
 Overflow

2. Don't Float Away!

Suppose we use an 8-bit floating point format similar to IEEE-754, with 1 sign bit, 3 exponent bits, and 4 significand bits. Assume the bias is -3 and we add the bias. For ALL parts of this question, express your answer a) in decimal, and b) in hex. Make sure you add the prefix to your hex value, fully simplify your answers, and do NOT leave them as fractions. Feel free to plug your fraction into Google to turn it into a decimal value. For all answers, write the exact decimal value, not a rounded one. All solutions have a finite number of decimal digits without rounding!

Quick reminder about intervals: (and) are exclusive while [and] are inclusive.

- (a) i. Find the **largest positive normalized number** represented in this new format.

A. (1.0 pt) Decimal:

15.5

The largest possible exponent for normalized number is $6_{10} = 110_2$. $1.1111_2 * 2^{6-3} = 1.1111_2 * 2^3 = 1111.1_2 = 15.5_{10}$

B. (1.0 pt) Hex:

0x6F

In hex, sign = 0_2 , exp = 110_2 , significand = 1111_2 , and thus $01101111_2 = 0x6F$

ii. Find the **smallest positive non-zero normalized number** represented in this new format.

A. (1.0 pt) Decimal:

0.25

The smallest possible exponent for normalized number is $1_{10} = 001_2$. $1.0000_2 * 2^{1-3} = 1.0000_2 * 2^{-2} = 0.010000_2 = 0.25_{10}$

B. (1.0 pt) Hex:

0x10

In hex, sign = 0_2 , exp = 001_2 , significand = 0000_2 , and thus $00010000_2 = 0x10$

iii. Find the **largest positive denormalized number** represented in this new format.

A. (1.0 pt) Decimal:

0.234375

For a denormalized number, the exponent has to be $0_{10} = 000_2$. $0.1111_2 * 2^{0-3+1} = 0.1111_2 * 2^{-2} = 0.001111_2 = 0.234375_{10}$

B. (1.0 pt) Hex:

0x0F

In hex, sign = 0_2 , exp = 000_2 , significand = 1111_2 , and thus $00001111_2 = 0x0F$

iv. Find the smallest positive non-zero denormalized number represented in this new format.

A. (1.0 pt) Decimal:

0.015625

For a denormalized number, the exponent has to be $0_{10} = 000_2$. $0.0001_2 * 2^{0-3+1} = 0.0001_2 * 2^{-2} = 0.000001_2 = 0.015625_{10}$

B. (1.0 pt) Hex:

0x01

In hex, sign = 0_2 , exp = 000_2 , significand = 0001_2 , and thus $00000001_2 = 0x01$ Note that the significand was not 0 because we wanted the smallest positive non-zero value.

(b) i. Give the nearest representation of π ($\approx 3.14159\dots$).

A. (3.0 pt) Decimal:

3.125

The first step is to break the number into two parts, the integer and the fractional value. So we want 3 so the integer value will be 0b11. Next we want to try to approximate the fractional value since we will not have enough bits to exactly represent π given our floating point spec. The first step is to add the largest of each fraction and see if the value is below. If it is, we keep it, if it is not, we do not keep it. If we find we are far off from a value, we will have to look at the slightly larger value to see if it is closer. We keep doing this until we go through all the fractional values. For example, we first try $\frac{1}{2} = 0.5$ which we see is above .14159265359... so we do not add the first fractional bit, next is $\frac{1}{4} = 0.25$ which is also above our desired value so we also skip it. Finally we add $\frac{1}{8} = 0.125$ which is really close though it is still above. Let's move on to the final bit: $\frac{1}{16} = 0.0625$. This is far from off from the value we wanted. Since this was not close but the previous bit was still high, we will see which one is closer and find that $\frac{1}{8}$ is closer. This means we will only keep that bit. Notice that we can only go to $\frac{1}{8}$ because we need to use 2 bits to represent the integer value. When we transform the number into its floating form, this means that part of the integer value must be encoded in the significand. $11.001_2 = 1.1001_2 * 2 = 3.125_{10}$

B. (3.0 pt) Hex (using our floating point representation):

0x49

In hex, sign = 0_2 , exp = 100_2 , significand = 1001_2 , and thus $01001001_2 = 0x49$

ii. Give the nearest representation of e ($\approx 2.71828\dots$).

A. (3.0 pt) Decimal:

2.75

Please read the explanation for how we calculated π as this is the same. In this case, we need to determine if $\frac{1}{2} + \frac{1}{4} = 0.75$ or $\frac{1}{2} + \frac{1}{8} = 0.625$ is closer to 0.7182818284 . In this case, $0.75 - 0.7182818284 = 0.0317181716$ and $0.7182818284 - 0.625 = 0.0932818284$ so the larger value is closer. Once again, we could only go to $\frac{1}{8}$ because we had two bits for the integer part. $10.11_2 = 1.0110_2 * 2 = 2.75_{10}$

B. (3.0 pt) Hex (using our floating point representation):

0x46

In hex, sign = 0_2 , exp = 100_2 , significand = 0110_2 , and thus $01000110_2 = 0x46$

iii. Give the nearest representation of $\sqrt{2}$ ($\approx 1.41429\dots$).

A. (3.0 pt) Decimal:

1.4375

Please read the explanation for how we calculated π as this is the same. In this case, we need to determine if $\frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 0.4375$ or $\frac{1}{4} + \frac{1}{8} = 0.375$ is closer to 0.41429. In this case, $0.4375 - 0.41429 = 0.02321$ and $0.41429 - 0.375 = 0.03929$ so the larger value is closer. Notice we can use up to $\frac{1}{16}$ because we only need one bit to represent the decimal portion. $1.0111_2 * 2^0 = 1.0111_2 * 1 = 1.4375_{10}$

B. (3.0 pt) Hex (using our floating point representation):

0x37

In hex, sign = 0₂, exp = 011₂, significand = 0111₂, and thus 00110111₂ = 0x37

iv. Give the nearest representation of δ (Feigenbaum constant) ($\approx 4.66920\dots$).

A. (3.0 pt) Decimal:

4.75

Please read the explanation for how we calculated π as this is the same. In this case, we lose an additional bit to the integer so we will only have two bits for the mantissa. In this case, we need to determine if $\frac{1}{2} + \frac{1}{4} = 0.75$ or $\frac{1}{2} = 0.5$ is closer to 0.66920 . In this case, $0.75 - 0.66920 = 0.0808$ and $0.66920 - 0.5 = 0.1692$ so the larger value is closer. $100.11_2 = 1.0011_2 * 2^2 = 4.75_{10}$

B. (3.0 pt) Hex (using our floating point representation):

0x43

In hex, sign = 0_2 , exp = 101_2 , significand = 0011_2 , and thus $01000011_2 = 0x53$

- (c) i. A. (1.5 pt) How many Floating Point numbers are in the interval of $(2^{-2}, 2^{-1})$? (Answer in decimal)

15

$$2^4 - 1 = 15$$

- B. (1.5 pt) How many Floating Point numbers are in the interval of $(2^1, 2^3)$? (Answer in decimal)

31

$$2^5 - 1 = 31$$

- C. (1.5 pt) How many Floating Point numbers are in the interval of $[2^{-2}, 2^{-1})$? (Answer in decimal)

16

$$2^4 = 16$$

- ii. A. (1.0 pt) What is the first integer that's not representable in this representation? (Answer in decimal)

16

- B. (1.0 pt) How many Floating Point numbers are in the interval of (0, 1)? (Answer in decimal)

47

$$(2^4 - 1) + (2^4) + (2^4) = 47$$

- C. (1.0 pt) How many positive non-zero denormalized Floating Point numbers can we represent? (Answer in decimal)

15

$$2^4 - 1 = 15$$

- iii. A. (1.5 pt) What's the gap (aka absolute value of the difference) between the **biggest positive** finite denorm and **smallest positive** non-zero norm? (Answer in decimal)

0.015625

$$1.0000_2 * 2^{1-3} - 0.1111_2 * 2^{0-3+1} = \frac{1}{4} - \frac{15}{64} = \frac{1}{64} = 0.015625$$

- B. (1.5 pt) What's the gap (aka absolute value of the difference) between the **biggest positive** finite denorm and **biggest positive** finite norm? (Answer in decimal)

15.265625

$$1.1111_2 * 2^{6-3} - 0.1111_2 * 2^{0-3+1} = 15.5 - \frac{15}{64} = 15.265625$$

- C. (1.5 pt) What's the gap (aka absolute value of the difference) between the **smallest positive** non-zero denorm and **biggest positive** finite norm? (Answer in decimal)

15.484375

$$1.1111_2 * 2^{6-3} - 0.0001_2 * 2^{0-3+1} = 15.5 - \frac{1}{64} = 15.484375$$

- D. (1.5 pt) What's the gap (aka absolute value of the difference) between the **smallest positive** non-zero denorm and **smallest positive** non-zero norm? (Answer in decimal)

0.234375

$$1.0000_2 * 2^{1-3} - 0.0001_2 * 2^{0-3+1} = \frac{1}{4} - \frac{1}{64} = \frac{15}{64} = 0.234375$$

3. For this problem, assume all pointers and integers are **four bytes** and all characters are **one byte**. Consider the following C code (all the necessary `#include` directives are omitted). C structs are properly aligned in memory and all calls to `malloc` succeed. **For all of these questions, assume we are analyzing them right before `main` returns.**

(a) **Doubly Linked Trouble!**

```
typedef struct node {
    void *data;
    struct node *nxt;
    struct node *prv;
} node;

void push_back(node *list, void *data) {
    node *n = (node *) malloc(sizeof(node));
    n->data = data; n->nxt = list; n->prv = list->prv;
    list->prv->nxt = n; list->prv = n;
}

int main() {
    char *r    = "CS 61C Rocks!";
    char s[]   = "CS 61C Sucks!";
    node sentinel; sentinel.nxt = &sentinel; sentinel.prv = &sentinel;
    push_back(&sentinel, r);
    push_back(&sentinel, s);
    push_back(&sentinel, &sentinel);
    push_back(&sentinel, calloc(sizeof(s) + 1, sizeof(char)));
}
```

- i. Each of the following evaluate to an address in memory. In other words, they “point” somewhere. Where in memory do they **point**?

A. (0.75 pt) `&push_back`

- Code
 Static
 Stack
 Heap

B. (0.75 pt) `sentinel.nxt->data`

- Code
 Static
 Stack
 Heap

C. (0.75 pt) `sentinel.nxt->nxt->data`

- Code
 Static
 Stack
 Heap

D. (0.75 pt) `sentinel.prv->data`

- Code
- Static
- Stack
- Heap

E. (0.75 pt) `sentinel.prv->prv`

- Code
- Static
- Stack
- Heap

F. (0.75 pt) `sentinel.prv->prv->data`

- Code
- Static
- Stack
- Heap

G. (0.75 pt) `&sentinel`

- Code
- Static
- Stack
- Heap

- ii. (3.0 pt) How many bytes of memory are allocated but not `free()`d by this program, if any? (assuming we have not called `free_list`) (Leave your answers as an integer. Do not include the units, we are telling you it's bytes after all!)

63

Each node will be `sizeof(node) == 12` bytes. We have allocated 4 nodes so 48 bytes.

We also made a `calloc` of 15 bytes (Since the compiler knows the length of the `s` array since it is stored on the stack which is 13 characters plus a null terminator so 14 bytes long). This means that we will leak 63 bytes.

- iii. Say we had this free function:

```
void free_list(node *n) {
    if (n == NULL) return;
    node *c = n->nxt;
    for (; c != n;){
        node *tmp = c; c = c->nxt;
        free(tmp);
    }
}
```

- A. (1.75 pt) Given this free function, if we called `free_list(&sentinel)` **after all** the code in `main` is executed, this program would have well defined behavior.

True

False

This free function would free the `sentinel` node which is stored on the stack not the heap! This would result in undefined behavior.

- B. (1.75 pt) Given this free function, if we called `free_list(&sentinel)` **after all** the code in `main` is executed, we would have no memory leaks.

True

False

Even though we would have undefined behavior due to us freeing the `sentinel` node (stored on the stack), we would still have a memory leak since we do not free our `calloc`'ed data.

(b) Doubly Linked Trouble!

```

typedef struct node {
    void *data;
    struct node *nxt;
    struct node *prv;
} node;

void push_back(node *list, void *data) {
    node *n = (node *) malloc(sizeof(node));
    n->data = data; n->nxt = list; n->prv = list->prv;
    list->prv->nxt = n; list->prv = n;
}

int main() {
    char *r      = "CS 61C Rocks!!!!";
    char s[]     = "CS 61C Sucks!?!?";
    node sentinel; sentinel.nxt = &sentinel; sentinel.prv = &sentinel;
    push_back(&sentinel, s);
    push_back(&sentinel, r);
    push_back(&sentinel, calloc(sizeof(s) + 1, sizeof(char)));
    push_back(&sentinel, &sentinel);
}

```

i. Each of the following evaluate to an address in memory. In other words, they “point” somewhere. Where in memory do they **point**?

A. (0.75 pt) &main

- Code
- Static
- Stack
- Heap

B. (0.75 pt) sentinel.nxt->data

- Code
- Static
- Stack
- Heap

C. (0.75 pt) sentinel.nxt->nxt->data

- Code
- Static
- Stack
- Heap

D. (0.75 pt) `sentinel.prv->data`

- Code
- Static
- Stack
- Heap

E. (0.75 pt) `sentinel.prv->prv`

- Code
- Static
- Stack
- Heap

F. (0.75 pt) `sentinel.prv->prv->data`

- Code
- Static
- Stack
- Heap

G. (0.75 pt) `&sentinel`

- Code
- Static
- Stack
- Heap

- ii. (3.0 pt) How many bytes of memory are allocated but not `free()`d by this program, if any? (assuming we have not called `free_list`) (Leave your answers as an integer. Do not include the units, we are telling you it's bytes after all!)

66

Each node will be `sizeof(node) == 12` bytes. We have allocated 4 nodes so 48 bytes.

We also made a `calloc` of 18 bytes (Since the compiler knows the length of the `s` array since it is stored on the stack which is 16 characters plus a null terminator so 17 bytes long). This means that we will leak 66 bytes.

- iii. Say we had this free function:

```
void free_list(node *n) {
    if (n == NULL) return;
    node *c = n->nxt;
    for (; c != n;) {
        node *tmp = c; c = c->nxt;
        free(tmp);
    }
}
```

- A. (1.75 pt) Given this free function, if we called `free_list(&sentinel)` **after all** the code in `main` is executed, this program would have well defined behavior.

True

False

This free function would free the `sentinel` node which is stored on the stack not the heap! This would result in undefined behavior.

- B. (1.75 pt) Given this free function, if we called `free_list(&sentinel)` **after all** the code in `main` is executed, we would have no memory leaks.

True

False

Even though we would have undefined behavior due to us freeing the `sentinel` node (stored on the stack), we would still have a memory leak since we do not free our `calloc`'ed data.

(c) Doubly Linked Trouble!

```

typedef struct node {
    void *data;
    struct node *nxt;
    struct node *prv;
} node;

void push_back(node *list, void *data) {
    node *n = (node *) malloc(sizeof(node));
    n->data = data; n->nxt = list; n->prv = list->prv;
    list->prv->nxt = n; list->prv = n;
}

int main() {
    char *r      = "CS 61C is super fun!";
    char s[]     = "CS 61C is super bad!";
    node* sentinel = (node*) malloc(sizeof(node));
    sentinel->nxt = sentinel; sentinel->prv = sentinel;
    push_back(sentinel, calloc(sizeof(s) + 1, sizeof(char)));
    push_back(sentinel, sentinel);
    push_back(sentinel, r);
    push_back(sentinel, s);
}

```

i. Each of the following evaluate to an address in memory. In other words, they “point” somewhere. Where in memory do they **point**?

A. (0.75 pt) &push_back

- Code
- Static
- Stack
- Heap

B. (0.75 pt) sentinel->nxt->data

- Code
- Static
- Stack
- Heap

C. (0.75 pt) sentinel->nxt->nxt->data

- Code
- Static
- Stack
- Heap

D. (0.75 pt) `sentinel->prv->data`

- Code
- Static
- Stack
- Heap

E. (0.75 pt) `sentinel->prv->prv`

- Code
- Static
- Stack
- Heap

F. (0.75 pt) `sentinel->prv->prv->data`

- Code
- Static
- Stack
- Heap

G. (0.75 pt) `&sentinel`

- Code
- Static
- Stack
- Heap

- ii. (3.0 pt) How many bytes of memory are allocated but not `free()`d by this program, if any? (assuming we have not called `free_list`) (Leave your answers as an integer. Do not include the units, we are telling you it's bytes after all!)

82

Each node will be `sizeof(node) == 12` bytes. We have allocated 5 nodes so 60 bytes.

We also made a `calloc` of 22 bytes (Since the compiler knows the length of the `s` array since it is stored on the stack which is 20 characters plus a null terminator so 21 bytes long). This means that we will leak 82 bytes.

- iii. Say we had this free function:

```
void free_list(node *n) {
    if (n == NULL) return;
    node *c = n->nxt;
    for (; c != n;){
        node *tmp = c; c = c->nxt;
        free(tmp);
    }
}
```

- A. (1.75 pt) Given this free function, if we called `free_list(sentinel)` **after all** the code in `main` is executed, this program would have well defined behavior.

- True
 False

Since all nodes of the linked list, including the `sentinel` node are stored on the heap, this free function would free all of the heap-allocated node structs.

- B. (1.75 pt) Given this free function, if we called `free_list(sentinel)` **after all** the code in `main` is executed, we would have no memory leaks.

- True
 False

We would still have a memory leak since we do not free our `calloc`'ed data.

(d) Doubly Linked Trouble!

```
typedef struct node {
    void *data;
    struct node *nxt;
    struct node *prv;
} node;

void push_back(node *list, void *data) {
    node *n = (node *) malloc(sizeof(node));
    n->data = data; n->nxt = list; n->prv = list->prv;
    list->prv->nxt = n; list->prv = n;
}

int main() {
    char *r      = "CS 61C is supper [sic]!";
    char s[]     = "CS 61C is super boring!";
    node* sentinel = (node*) malloc(sizeof(node));
    sentinel->nxt = sentinel; sentinel->prv = sentinel;
    push_back(sentinel, sentinel);
    push_back(sentinel, calloc(sizeof(s) + 1, sizeof(char)));
    push_back(sentinel, s);
    push_back(sentinel, r);
}
```

- i. Each of the following evaluate to an address in memory. In other words, they “point” somewhere. Where in memory do they **point**?

A. (0.75 pt) &main

- Code
 Static
 Stack
 Heap

B. (0.75 pt) sentinel->nxt->data

- Code
 Static
 Stack
 Heap

C. (0.75 pt) sentinel->nxt->nxt->data

- Code
 Static
 Stack
 Heap

D. (0.75 pt) sentinel->prv->data

- Code
- Static
- Stack
- Heap

E. (0.75 pt) sentinel->prv->prv

- Code
- Static
- Stack
- Heap

F. (0.75 pt) sentinel->prv->prv->data

- Code
- Static
- Stack
- Heap

G. (0.75 pt) sentinel

- Code
- Static
- Stack
- Heap

- ii. (3.0 pt) How many bytes of memory are allocated but not `free()`d by this program, if any? (assuming we have not called `free_list`) (Leave your answers as an integer. Do not include the units, we are telling you it's bytes after all!)

85

Each node will be `sizeof(node) == 12` bytes. We have allocated 5 nodes so 60 bytes.

We also made a `calloc` of 25 bytes (Since the compiler knows the length of the `s` array since it is stored on the stack which is 23 characters plus a null terminator so 24 bytes long). This means that we will leak 85 bytes.

- iii. Say we had this free function:

```
void free_list(node *n) {
    if (n == NULL) return;
    node *c = n->nxt;
    for (; c != n;){
        node *tmp = c; c = c->nxt;
        free(tmp);
    }
}
```

- A. (1.75 pt) Given this free function, if we called `free_list(sentinel)` after all the code in `main` is executed, this program would have well defined behavior.

True

False

Since all nodes of the linked list, including the `sentinel` node are stored on the heap, this free function would free all of the heap-allocated node structs.

- B. (1.75 pt) Given this free function, if we called `free_list(sentinel)` after all the code in `main` is executed, we would have no memory leaks.

True

False

We would still have a memory leak since we do not free our `calloc`'ed data.

4. RISC-V!

For each of the following, write a simple RISC-V function with one argument. Follow calling convention, use register mnemonic names (e.g., refer to t0 rather than x6), and add commas and a single space between registers/arguments (e.g. `addi a0, a1, 2`). If you do not follow this, you may be misgraded!

- (a) Arithmetically negate a Two's Complement 32-bit integer without using the `sub`, `mul` or pseudo instructions.

```
negate:
  __<CODE INPUT 1>__
  __<CODE INPUT 2>__
  ret
```

Fill in the following:

- i. (0.75 pt) <CODE INPUT 1>

```
xori a0, a0, -1
```

- ii. (0.75 pt) <CODE INPUT 2>

```
addi a0, a0, 1
```

- (b) Find the length of a null-terminated string in bytes. The function should accept a pointer to a null-terminated string and return an integer. Your solution must be recursive!

```
strlen:
  __<CODE INPUT 1>__
  beq t0, zero, basecase
  __<CODE INPUT 2>__
  __<CODE INPUT 3>__
  __<CODE INPUT 4>__
  jal strlen
  __<CODE INPUT 5>__
  __<CODE INPUT 6>__
  __<CODE INPUT 7>__
  ret
basecase:
  __<CODE INPUT 8>__
  ret
```

Fill in the following:

- i. (0.75 pt) <CODE INPUT 1>

```
lb t0, 0(a0)
```

- ii. (0.75 pt) <CODE INPUT 2>

```
addi sp, sp, -4
```

- iii. (0.75 pt) <CODE INPUT 3>

```
sw ra, 0(sp)
```

- iv. (0.75 pt) <CODE INPUT 4>

```
addi a0, a0, 1
```

- v. (0.75 pt) <CODE INPUT 5>

```
addi a0, a0, 1
```

- vi. (0.75 pt) <CODE INPUT 6>

```
lw ra, 0(sp)
```

- vii. (0.75 pt) <CODE INPUT 7>

```
addi sp, sp, 4
```


viii. (0.75 pt) <CODE INPUT 8>

```
addi a0, zero, 0
```

(c) Leave your answers fully simplified as integers. **Do not leave powers of 2 in your answer!** Feel free to use a calculator to simplify your answer.

- i. You want to build a mini RISC-V instruction architecture that only supports 16 registers, which allows the length of the register fields to be shortened. Assuming that you use the extra bits to extend the immediate field, what is the range of 32-bit instructions that can be reached using a branch instruction in this new format? [<lower bound>, <upper bound>]

A. (0.75 pt) <lower bound>

-4096

$[-2^{12}, 2^{12} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

B. (0.75 pt) <upper bound>

4095

$[-2^{12}, 2^{12} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

- ii. You want to build a mini RISC-V instruction architecture that only supports 16 registers, which allows the length of the register fields to be shortened. Assuming that you use the extra bits to extend the immediate field, what is the range of half-word instructions that can be reached using a branch instruction in this new format? [<lower bound>, <upper bound>]

A. (0.75 pt) <lower bound>

-8192

$[-2^{13}, 2^{13} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

B. (0.75 pt) <upper bound>

8191

$[-2^{13}, 2^{13} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

- iii. You want to build a mini RISC-V instruction architecture that only supports 16 registers, which allows the length of the register fields to be shortened. Assuming that you use the extra bits to extend the immediate field, what is the range of 32-bit instructions that can be reached using a jal instruction in this new format? [<lower bound>, <upper bound>]

A. (0.75 pt) <lower bound>

-524288

$[-2^{19}, 2^{19} - 1]$ Since we now only need 4 bits for the register fields, for the jal instruction, the immediate field will have 21 bits.

B. (0.75 pt) <upper bound>

524287

$[-2^{19}, 2^{19} - 1]$ Since we now only need 4 bits for the register fields, for the jal instruction, the immediate field will have 21 bits.

- iv. You want to build a mini RISC-V instruction architecture that only supports 16 registers, which allows the length of the register fields to be shortened. Assuming that you use the extra bits to extend the immediate field, what is the range of half-word instructions that can be reached using a jal instruction in this new format? [<lower bound>, <upper bound>]

A. (0.75 pt) <lower bound>

-1048576

$[-2^{20}, 2^{20} - 1]$ Since we now only need 4 bits for the register fields, for the jal instruction, the immediate field will have 21 bits.

B. (0.75 pt) <upper bound>

1048575

$[-2^{20}, 2^{20} - 1]$ Since we now only need 4 bits for the register fields, for the jal instruction, the immediate field will have 21 bits.

(d) i. A. (1.0 pt)

```
auipc t0, 0xABCDE # Assume this instruction is at 0x100
addi t0, t0, 0xABC
```

Write down the value of t0 in hex. Reminder: include the prefix in your answer!

0xABCDDBBC

B. (1.0 pt)

```
auipc t0, 0x12345 # Assume this instruction is at 0x200
addi t0, t0, 0xDEF
```

Write down the value of t0 in hex. Reminder: include the prefix in your answer!

0x12344FEF

ii. A. (2.0 pt)

```
li t0, 0xABCDEFAD
sw t0, 0(s0)
lb t0, 0(s0)
```

Write down the value of t0 in hex. Assume big-endianness. Reminder: include the prefix in your answer!

0xFFFFFFFFAB

B. (2.0 pt)

```
li t0, 0xABCDEFAD
sw t0, 0(s0)
lb t0, 1(s0)
```

Write down the value of t0 in hex. Assume big-endianness. Reminder: include the prefix in your answer!

0xFFFFFFFFCD

C. (2.0 pt)

```
li t0, 0xABCDEFAD
sw t0, 0(s0)
lb t0, 2(s0)
```

Write down the value of t0 in hex. Assume big-endianness. Reminder: include the prefix in your answer!

0xFFFFFFFFEF

D. (2.0 pt)

```
li t0, 0xABCDEFAD
sw t0, 0(s0)
lb t0, 3(s0)
```

Write down the value of t0 in hex. Assume big-endianness. Reminder: include the prefix in your answer!

0xFFFFFFFFAD

5. (a) CALL!

Consider the following assembly code (Note these are the addresses the assembler give each of the instructions):

Address	Assembly
0x00	add t0, x0, x0
0x04	addi t1, x0, 4
0x0C	loop: beq t0, t1, end
0x10	add a0, a0, t0
0x14	jal ra, square
0x18	jal ra, printf
0x1C	n: addi t0, t0, 1
0x20	j loop
0x24	end: ecall
0x28	square: mul a0, a0, a0
0x2C	ret

i. (0.5 pt) This code is the output of...

- Compiler
- Assembler
- Linker
- Loader
- None of the other options

Assuming an isolated assembler, create the symbol table after the first pass (top to down). If a line of the symbol table is not used, enter N/A.

ii. A.

B. (0.25 pt) First label:

loop

C. (0.25 pt) First address:

0x0C

D.

E. (0.25 pt) Second label:

n

F. (0.25 pt) Second address:

0x1C

G.

H. (0.25 pt) Third label:

end

I. (0.25 pt) Third address:

0x24

J.

K. (0.25 pt) Fourth label:

square

L. (0.25 pt) Fourth address:

0x28

iii. (2.0 pt) After the first pass of a top to bottom assembler, which of the following instructions have their addresses fully resolved?

- beq t0, t1, end
 jal ra, square
 jal ra, printf
 j loop
 None of the other options

iv. (1.0 pt) No address needs to be resolved at the linker stage.

- True
 False

printf need to be resolved at the linker stage

v. (1.0 pt) A poorly written but correct assembler won't seriously slow down the speed of the compiled program.

- True
 False

The assembler is only replacing pseudo instructions and offsets, so won't affect the speed

vi. (4.0 pt) Translate the instruction at address 0x20 into machine code (in hex).

0xFEDFF06F

	Imm[20 10:1 11 19:12]	rd	opcode
0b	1 1111110110 1 11111111	00000	1101111

vii. (1.5 pt) Apple recently announced that it is switching from Intel processors to ARM ones, which have a different ISA (a RISC one!). To ensure that old programs can still run on these new devices, which stage(s) of the CALL stack do they need to re-run to create the executable binaries?

- Compiler
 Assembler
 Linker
 Loader
 None of the other options

(b) CALL!

Consider the following assembly code (Note these are the addresses the assembler give each of the instructions):

Address	Assembly
0x0C	add t0, x0, x0
0x10	addi t1, x0, 4
0x14	loop: beq t0, t1, end
0x18	add a0, a0, t0
0x1C	jal ra, square
0x20	jal ra, printf
0x24	n: addi t0, t0, 1
0x28	j loop
0x2C	end: ecall
0x30	square: mul a0, a0, a0
0x34	ret

i. (0.5 pt) This code is the input of ..

- Compiler
 Assembler
 Linker
 Loader
 None of the other options

Assuming an isolated assembler, create the symbol table after the first pass (top to down). If a line of the symbol table is not used, enter N/A.

ii. A.

B. (0.25 pt) First label:

loop

C. (0.25 pt) First address:

0x14

D.

E. (0.25 pt) Second label:

n

F. (0.25 pt) Second address:

0x24

G.

H. (0.25 pt) Third label:

`end`

I. (0.25 pt) Third address:

`0x2C`

J.

K. (0.25 pt) Fourth label:

square

L. (0.25 pt) Fourth address:

0x30

iii. (2.0 pt) After the first pass of a top to bottom assembler, which of the following instructions do **NOT** have their addresses fully resolved?

- beq t0, t1, end
- jal ra, square
- jal ra, printf
- j loop
- None of the other options

iv. (1.0 pt) No address needs to be resolved at the linker stage.

- True
- False

printf need to be resolved at the linker stage

v. (1.0 pt) A poorly written but correct assembler can seriously slow down the speed of the compiled program.

- True
- False

The assembler is only replacing pseudo instructions and offsets, so won't affect the speed

vi. (4.0 pt) Translate the instruction at address 0x1C into machine code (in hex).

0x014000EF

Imm[20 10:1 11 19:12]	rd	opcode
0b 0 0000001010 0 00000000	00001	1101111

vii. (1.5 pt) Apple recently announced that it is switching from Intel processors to ARM ones, which have a different ISA (a RISC one!). To ensure that old programs can still run on these new devices, which stage(s) of the CALL stack do they need to re-run to create the executable binaries?

- Compiler
- Assembler
- Linker
- Loader
- None of the other options

(c) CALL!

Consider the following assembly code (Note these are the addresses the assembler give each of the instructions):

Address	Assembly
	.data
0x80	n: .word 9
	.text
0x00	main: add t0, x0, x0
0x04	addi t1, x0, 1
0x08	la t3, n
0x10	lw t3, 0(t3)
0x14	fib: beq t3, x0, finish
0x18	add t2, t1, t0
0x1C	mv t0, t1
0x20	mv t1, t2
0x24	addi t3, t3, -1
0x28	j fib
0x2C	finish: addi a0, x0, 1
0x30	addi a1, t0, 0
0x34	ecall # Print int
0x38	addi a0, x0, 10
0x3c	ecall # Terminate ecall

i. (0.5 pt) This code is the input of...

- Compiler
 Assembler
 Linker
 Loader
 None of the other options

Assuming an isolated assembler, create the symbol table after the first pass (top to down). If a line of the symbol table is not used, enter N/A.

ii. A.

B. (0.25 pt) First label:

n

C. (0.25 pt) First address:

0x80

D.

E. (0.25 pt) Second label:

main

F. (0.25 pt) Second address:

0x00

G.

H. (0.25 pt) Third label:

fb

I. (0.25 pt) Third address:

0x14

J.

K. (0.25 pt) Fourth label:

finish

L. (0.25 pt) Fourth address:

0x2C

iii. (2.0 pt) After the first pass of a top to bottom assembler, which of the following instructions have their addresses fully resolved?

- `la t3, n`
 `beq t3, x0, finish`
 `j fib`
 None of the other options

iv. (1.0 pt) No address needs to be resolved at the linker stage.

- True
 False

For `la t3, n` address of `n` needs to be resolved at the linker stage

v. (1.0 pt) A poorly written but correct assembler won't seriously slow down the speed of the compiled program.

- True
 False

The assembler is only replacing pseudo instructions and offsets, so won't affect the speed.

vi. (4.0 pt) Translate the instruction at address 0x14 into machine code (in hex).

0x000E0C63

	Imm[12 10:5]	rs1	func3	imm[4:1 11]	opcode
0b	0 000000	00000	11100	000	1100 0

vii. (1.5 pt) Apple recently announced that it is switching from Intel processors to ARM ones, which have a different ISA (a RISC one!). To ensure that old programs can still run on these new devices, which stage(s) of the CALL stack do they need to re-run to create the executable binaries?

- Compiler
 Assembler
 Linker
 Loader
 None of the other options

(d) CALL!

Consider the following assembly code (Note these are the addresses the assembler give each of the instructions):

Address	Assembly
	.data
0x90	n: .word 9
	.text
0x10	main: add t0, x0, x0
0x14	addi t1, x0, 1
0x18	la t3, n
0x20	lw t3, 0(t3)
0x24	fib: beq t3, x0, finish
0x28	add t2, t1, t0
0x2C	mv t0, t1
0x30	mv t1, t2
0x34	addi t3, t3, -1
0x38	j fib
0x3C	finish: addi a0, x0, 1
0x40	addi a1, t0, 0
0x44	jal ra, printf
0x48	addi a0, x0, 10
0x4c	ecall # Terminate ecall

i. (0.5 pt) This code is the output of...

- Compiler
 Assembler
 Linker
 Loader
 None of the other options

Assuming an isolated assembler, create the symbol table after the first pass (top to down). If a line of the symbol table is not used, enter N/A.

ii. A.

B. (0.25 pt) First label:

n

C. (0.25 pt) First address:

0x90

D.

E. (0.25 pt) Second label:

main

F. (0.25 pt) Second address:

0x10

G.

H. (0.25 pt) Third label:

fb

I. (0.25 pt) Third address:

0x24

J.

K. (0.25 pt) Fourth label:

finish

L. (0.25 pt) Fourth address:

0x3C

iii. (2.0 pt) After the first pass of a top to bottom assembler, which of the following instructions have their addresses fully resolved?

- `la t3, n`
 `beq t3, x0, finish`
 `j fib`
 `jal ra, printf`
 None of the other options

iv. (1.0 pt) Two addresses need to be resolved at the linker stage.

- True
 False

`printf` and `la` need to be resolved at the linker stage

v. (1.0 pt) A poorly written but correct assembler can seriously slow down the speed of the compiled program.

- True
 False

The assembler is only replacing pseudo instructions and offsets, so it won't affect the speed.

vi. (4.0 pt) Translate the instruction at address 0x38 into machine code (in hex).

`0xFEDFF06F`

Imm[20 10:1 11 19:12]	rd	opcode
0b 1 111110110 1 11111111	00000	1101111

vii. (1.5 pt) Apple recently announced that it is switching from Intel processors to ARM ones, which have a different ISA (a RISC one!). To ensure that old programs can still run on these new devices, which stage(s) of the CALL stack do they need to re-run to create the executable binaries?

- Compiler
 Assembler
 Linker
 Loader
 None of the other options

6. A Generic C Question

In object-oriented programming languages such as Java, the concept of a Generic data type exists. This means that, in a class definition of an object, we can leave the data types of chosen variables as an “unknown” type that is instead expected to be provided during instantiation of the object. In this problem, we will implement generics in C for a LinkedList. Remember, though, that we do not have objects to instantiate in C, so instead our GenericLinkedList should simply be versatile enough to accept any given data type without error or compiler warnings. **A user should not need to do any form of explicit or implicit casting when working with this new data type, except for when dealing with the void* pointer returned by the alloc functions.** For the following, assume we have included the correct includes.

- (a) (3.0 pt) You may assume that our GenericLinkedList only has to account for 3 data type choices: `char`, `uint16_t`, `uint32_t`, where the # in `uint#_t` represents the number of bits the data type contains. It also supports structs and unions. In addition, we are working on a 32-bit addressable memory space, structs are word-aligned and padded appropriately, and all calls to `malloc()`, `calloc()`, and `realloc()` succeed. Fill in the skeleton for a `GenericLink`. **Your solution must use the minimum amount of space possible. A sub-optimal solution may not receive credit. You may not use void* in your approach.**

```
typedef struct {
    <YOUR CODE HERE>
} GenericLink;
```

```
union { char val1; uint16_t val2; uint32_t val3; } value; GenericLink* next;
```

- (b) (1.0 pt) What does `sizeof(GenericLink)` evaluate to?

```
8
```

$32\text{bits (value)} + 32\text{bits (next)} = 64\text{ bits} = 8\text{ bytes}$

- (c) I now want to store a String as a GenericLinkedList, i.e. each link should hold one char of the string, with the links ordered the same way as the chars in the string. You may assume that the length of the string is > 1 . You do not need to worry about storing the null terminator. Please fill in the following function implementations:

- i. (2.0 pt)

```
GenericLink* store_char(char c) {
    /* store_char takes in a char, and returns a
       pointer to a link containing this char */

    <YOUR CODE HERE>
}
```

```
GenericLink* link = (GenericLink*) calloc(sizeof(GenericLink)); link->value.val1
= c; return link;
```

ii. (6.0 pt)

```
GenericLink* store_string(char* str) {  
    /* store_string takes in a string, and returns a  
    pointer to the 'head' of the GenericLinkedList  
    holding the string, i.e. the link containing the first char*/  
  
    <YOUR CODE HERE>  
}
```

```
GenericLink* string = store_char(str++); GenericLink curr = string; while (str)  
{ curr->next = store_char(str); curr = curr->next; str++; } return string;
```

No more questions.