Name: ________Answers__________________ Login: cs61c-____

Question 1: Potpourri: hard to spell, nice to smell… (14 pts, 36 min)

Questions (a) and (b) refer to the C code to the right; pretend you don't know about MIPS yet.

```
#define val 16
char arr[] = "foo";
void foo(int arg){
    char *str = (char *) malloc (val);
    char *ptr = arr;
}
```

a) In which memory sections (code, static, heap, stack) do the following reside?

arg __stack__ arr __static__

*str __heap__ val __code__

b) Name a C operation that would treat `arr` and `ptr` differently: __sizeof sizeof(arr) != sizeof(ptr)__ __++ (arr++ crashes, ptr++ does not)__

You peek into the *text* part of an `a.out` file and see that the left *six* bits of an instruction are `0x02`. As a result of executing this instruction…

c) What's the *most* that your PC could change? Be exact. __opcode=0x02 ➔ jump 2^28 – 4__

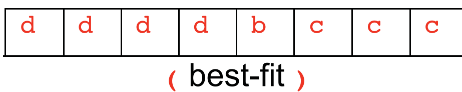
d) What is the *least*? __0__

e) Write a `getPC` function, which returns the address of the `jal` instruction **calling it**. (two instructions should be sufficient)

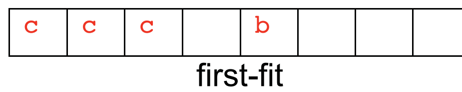
```
getPC: addiu $v0, $ra, -4
       jr $ra
```

f) Which of the *best*-, *first*-, *next*-fit schemes would succeed for **all 5** of the following sequence of `malloc` and `free` requests on a `malloc`-able region of memory only 8 bytes long? Circle those that would and show the resulting contents of memory for each one. E.g., After the "`a=malloc(4)`" call, all schemes should have the *leftmost* 4 boxes labeled "a". A pencil is useful (or draw "a" lightly).

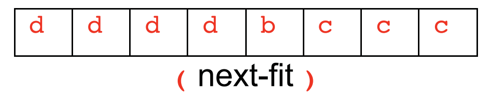
`a = malloc(4); b = malloc(1); free(a); c = malloc(3); d = malloc(4);`



(best-fit)



first-fit



(next-fit)

g) In one sentence, why can't we use automatic memory management in C?

__C is weakly typed; any variable could be a pointer.__

h) To reduce complexity for your software company, you delete the *Compiler*, *Assembler* and *Linker* and replace them with a single program, `CAL`, that takes all the source code in a project and does the job of all three for *all* the files given to it. Overall, is this a good idea or bad idea? Why or why not?

__BAD idea! A change to only one file requires recompiling/reassembling all!__

Name: ___Answers___ Login: cs61c-____

Question 2: Player's got a brand new bag… (15 pts, 36 min)

We want to add an inventory system to the adventure game so that the player can collect items. First, we'll implement a *bag* data structure that holds *items* in a linked list. Each `item_t` has an associated `weight`, and each `bag_t` has a `max_weight` that determines its holding capacity (see the definitions below). In the left text area for `item_node_t`, define the necessary data type to serve as the nodes in a **linked list** of items, and in the right text area, add any necessary fields to the `bag_t` definition.

```
typedef struct item {
    int weight;
    // other fields not shown
} item_t;
```

```
typedef struct item_node {
    // (a) FILL IN HERE

    item_t *item;
    struct item_node *next;

} item_node_t;
```

```
typedef struct bag {
    int max_weight;
    int current_weight;
    // add other fields necessary
    // (b) FILL IN HERE

    item_node_t *contents;

} bag_t;
```

c) Complete the `add_item()` function, which should add `item` into `bag` **only** if adding the item would not cause the weight of the bag contents to exceed the bag's `max_weight`. The function should return `0` if the item *could not* be added, or `1` if it succeeded. Be sure to update the bag's `current_weight`. You do not need to check if `malloc()` returns `NULL`. Insert the new item into the list wherever you wish.

```
int add_item(item_t *item, bag_t *bag) {
    if ( item->weight + bag->current_weight > bag->max_weight ) {
        return 0;
    }
    item_node_t *new_node = (item_node_t *) malloc( sizeof(item_node_t) );

    // Add more code below…

    new_node->item = item;
    new_node->next = bag->contents;
    bag->contents = new_node;
    bag->current_weight += item->weight;

    return 1;
}
```

(d) Finally, we want an `empty_bag()` function that frees the bag's linked list but **NOT** the memory of the items themselves and **NOT** the bag itself. The bag should then be "reset", ready for `add_item`. Assume that the operating system immediately fills any freed memory with garbage. Fill in the functions below.

```
void empty_bag(bag_t *bag) {
    free_contents( bag->contents );

    // FILL IN HERE
    bag->current_weight = 0;
    bag->contents = NULL;

}
```

```
void free_contents( item_node_t *c ) {

    // FILL IN HERE

    if (c == NULL) return;
    free_contents(c->next);
    free(c);

}
```

(e) Now suppose that we care about the order of `items` in our `bag`. However, because we're clumsy, the only possible way for us to rearrange `items` is to reverse their order in the list.

```
void reverse_list(bag_t *bag) {
    item_node_t *next, *node = bag->contents;
    bag->contents = NULL;

    while (node) {
        next = node->next;             // Keep track of the next node.
        node->next = bag->contents;    // Current node points to what's
                                       // currently reversed.
        bag->contents = node;          // Now current node is head of
                                       // currently reversed list.
        node = next;                   // Examine the next node, which we
                                       // saved.
    }
}
```

Bonus: You have five jars of pills. All the pills in one jar only are "contaminated." The only way to tell which pills are contaminated is by weight. A regular pill weighs 10 grams; a contaminated pill is 9 grams. You are given a scale and allowed to make just one measurement with it How do you tell which jar is contaminated?

Take out 1 pill from jar 2, 2 pills from jar 3, 3 pills from jar 4, and 4 pills from jar 5. Put them all on the scale. If it reads 100 grams, then none of the pills you took out was contaminated, so jar 1 is the culprit. If it reads 99, jar 2 is contaminated, 98 corresponds to jar 3, 97 jar 4, 96 jar 5.