CS 240: Programming in C
Midterm Exam 2
Fall 2024

Name:

Username/email: @purdue.edu

Read all instructions before beginning the exam.

- This is a closed book examination. No material other than those provided for you are allowed.
- You need only a pencil and eraser for this examination. If you use ink, use either black or blue ink. If you use pencil, your writing must be dark and clearly visible.
- This examination contains an amount of material that a well-prepared student should be able to complete in well under one hour.
- This examination is worth a total of 100 points. Not all questions are worth the same amount. Plan your time accordingly.
- Write legibly. You should try to adhere to the course code standard when writing your solution(s). Egregious violations may result in point deductions. Function headers are not required.
- You may leave after you have turned in all pages of the examination booklet. You will not be able to change any answers after turning in your examination booklet.
- If you finish your exam with less than 10 minutes remaining, please remain seated until the exam is over. We will call you down by row to turn in your exam.
- Read each question *carefully* and *only do what is specifically asked for* in that problem.
- Some problems require several steps. Show all your work. Partial credit can only be rewarded to work shown.
- Do not attempt to look at other students' work. Keep your answers to yourself. Any violation will be considered academic dishonesty.
- Write your username on *EVERY* page where indicated. Any page without a username will receive a zero for the material on that page.
- Read and sign the statement below. Wait for instructions to start the examination before continuing to the next page.

"I signify that the answers provided for this examination are my own and that I have not received any assistance from other students nor given any assistance to other students."

Signature:

- Do not open the examination booklet until instructed.

1. (30 points) Provide a short answer to each of the following questions.

   (a) (2 points) Write a single line of code to allocate an array of 20 `ints` on the heap, and store the address of the allocated array into a pointer named `ptr`.

   (b) (2 points) Both `malloc` and `calloc` allocate a chunk of memory on the heap. Briefly describe how the allocated memory is different when allocated with `malloc` versus when it is allocated with `calloc`.

For parts (c) and (d), consider the following function. Assume we have declared a `struct room` containing a pointer to `struct room` named `next`, an `int` named `locked`, and an `int` named `found_key`. Additionally, assume that none of `cur_ptr`, `*cur_ptr`, or `dest` are `NULL`, and that `dest` will be eventually found by following the `next` pointers.

```
 1.    void goto_room(struct room **cur_ptr, struct room *dest) {
 2.      while (*cur_ptr)!= dest) {
 3.        struct room *cur = *cur_ptr;
 4.        if (cur->next->locked) {
 5.          if (cur->found_key) {
 6.            cur->next->locked = 0;
 7.          }
 8.          else {
 9.            break;
10.          }
11.        }
12.        cur = cur->next;
13.      }
14.    }
```

(c) (4 points) Rewrite the above function to convert arrow notation (->) into dot notation (.). It is not necessary to write the line numbers.

(d) (4 points) The `goto_room` function above will compile, but it will not run as expected. Write the line number of the single line that has the error, and briefly explain why it causes a problem. Then rewrite that line to fix the function. As in part (c), do not use arrow notation in your solution.

(e) (4 points) What does the following program print? Assume a 64-bit, little-endian system.

```
#include <stdio.h>

int main() {
  int arr[] = { 0x11223344, 0x55667788 };
  char *c_ptr = (char *)arr;
  short *s_ptr = (short *)arr;
  int *i_ptr = (int *)arr;
  printf("%x %x %x\n", *(c_ptr + 1), *(s_ptr + 1), *(i_ptr + 1));
  return 0;
}
```

Consider the following code snippet, and answer parts (f) and (g).

```
int arr[] = { 3, 2, 1, 0, -1, -2, -3 };
int *p = &(arr[3]);
int *q = arr + 4;
printf("%d", q[*(p + *(q + 1))]);
int *a = &((p + *q)[3]) + (*q - 2);
```

(f) (2 points) What is printed by the printf statement?

(g) (2 points) Which element of arr is pointed to by a?

(h) (4 points) Given the following function:

```
char *do_something() {
  struct some_data {
    char str1[5];
    char *str2;
  } data = { "Hi", NULL };

  data.str2 = malloc(5);
  assert(data.str2 != NULL);
  strcpy(data.str2, "Hey");

  return data.str2;
}
```

Is the string pointed to by data.str1 on the stack or the heap? What about data.str2? Will the returned string still exist after the function returns?

(i) (3 points) Briefly describe what a "double pointer" is, and give one example of a situation where it is needed.

(j) (3 points) Write a function prototype for a function named `register_callback`, which returns an `int` and takes two arguments. The first argument is an `int` and the second argument is a pointer to a function that returns `void` and takes two arguments, a pointer to `int` and a pointer to a pointer to `char`. You may use a `typedef` if you wish.

2. (40 points) The following questions deal with structures that are dynamically allocated and form a singly-linked list.

   (a) (5 points) You are in charge of maintaining the history of an ancient empire. Among your responsibilities is the daunting task of cataloging each of the various rulers, and for how long they ruled. You decide it would be best to store this information in a linked list.

   Declare a structure named `era` containing a pointer to a string named `ruler` and a `float` named `years`. The structure should be a valid singly-linked list node. You may declare your own type for this structure if you wish to slightly simplify the remaining questions.

(b) (10 points) Write a function, `create_era()`, that accepts two arguments — a pointer to a string (the ruler of the new era) and a `float` (how many years they ruled, which must be > 0). It should return a pointer to the newly allocated `struct era`, containing a copy of the data pointed to by the first argument and the second argument. Be sure to properly initialize all fields of the structure, and use `asserts` where necessary!

(c) (10 points) Recent discoveries have revealed that a particular era was actually run by a secret ruler, unbeknownst to the public of that time. You will need to update your linked list.

Write a function, `replace()`, that accepts two arguments. The first argument is the *address* of a pointer to a `struct era` node (the old node to be replaced), and the second argument is a pointer to a `struct era` node (the new node that replaces the old node). Remove the old node from the linked list and insert the new node in its place. Ensure that all links are properly updated! After removing the old node from the list, deallocate all of its associated memory. The function's return type should be `void`. Use `asserts` where necessary.

Do **not** simply swap the data in each node. You must replace the node itself.

Hint: you may assume that the first argument contains the address of the previous node's `next` pointer (or the `head` pointer if the first node is being replaced).

Additional space on the next page...

Work area for 2.c continued...

(d) (3 points) In order to use your `replace()` function, you must first find the node that needs to be replaced. Additionally, you want to support other similar operations alongside `replace()`.

Declare a new type, `update_func`, as an alias of a pointer to a function with the same signature as the `replace()` function from part (c).

(e) (12 points) Write a function, `find_and_update()`, to search the linked list for a particular ruler and update the first node found with that ruler. The function takes four arguments. The first argument is the *address* of a pointer to a `struct era` node (the head of the linked list). The second argument is a pointer to a `struct era` node (the new node used for updating). The third argument is a pointer to a string (the ruler to search for), and the fourth argument is a function pointer of type `update_func` (as declared in part (d)). The return type is `int`.

Traverse the linked list, starting from the head, and look for any node with a `ruler` equal to the string pointed to by the third argument. Update the first node that fits this criteria by invoking the function pointed to by the fourth argument. Use the second argument of `find_and_update()` as the second argument to the update function. Return `1` if a node was updated, or `0` otherwise.

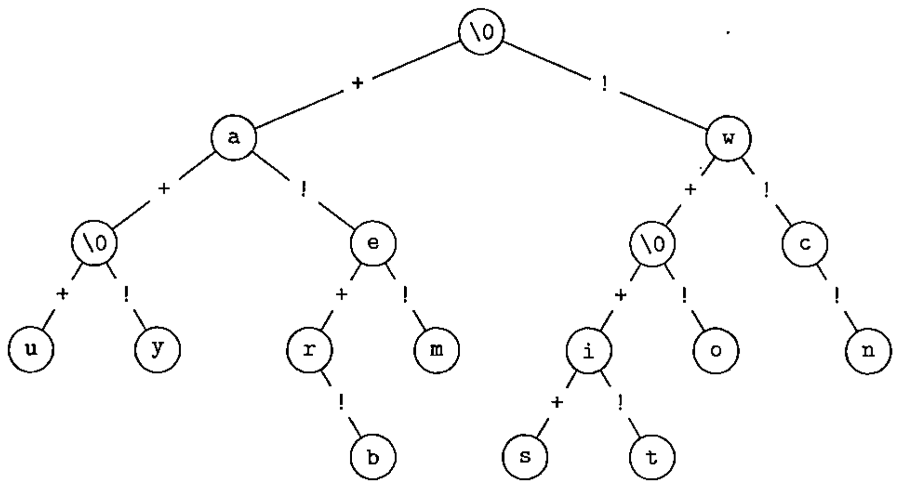
Be sure to use the update function correctly (see the hint for part (c)) so that the linked list remains intact. Use `assertions` where necessary. An empty list should not fail an `assertion`.

Additional space on the next page...

Work area for 2.e continued...

3. (30 points) The following questions deal with structures that are dynamically allocated and form a binary tree.

   (a) (5 points) The ancient culture you are studying has a fascinating written language consisting of only two characters, which closely resemble the plus sign (+) and the exclamation mark (!). Recent studies have discovered ways to map sequences of these characters to modern alphanumeric symbols (similar to Morse code). These mappings can be stored in a binary tree to facilitate decoding these ancient texts. Such a tree is shown below.



In this tree, each edge represents either a + (left) or a ! (right) symbol, and each node stores a character that maps to the sequence of + and ! symbols along the path from the root to that node. For example, given the string "+!!", then starting from the root we follow the + edge, then the ! edge, and finally the ! edge to arrive at the letter m. We can thus conclude that the sequence "+!!" represents the letter m. Note that some sequences, such as "++", do not map to any letter, thus the corresponding node stores the NUL terminator.

Declare a structure named `tree` to represent a node in the binary tree. It should store a `char` as well as the appropriate child pointers — one pointing to the `plus` subtree and one pointing to the `exclamation` subtree. You may declare your own type for this structure if you wish to slightly simplify the remaining questions.

(b) (5 points) You would like to determine which characters have already been mapped by scholars before you. Write a **recursive** function named `traverse()` that takes one argument, a pointer to a `struct tree` node, and returns `void`. Traverse the tree in postfix order, printing each non-`NUL` character found in the tree, separated by spaces. For this traversal, you may consider the `plus` subtree to be "left" and the `exclamation` subtree to be "right".

(c) (2 points) Given the tree from part (a), what will be printed by calling `traverse()` on the root of the tree?

(d) (8 points) In order to read inscriptions in this language, you need to be able to use the tree to decode them. Write a **recursive** function named `decode()` that accepts two arguments and returns a `char`. The first argument is a pointer to a `struct tree` node, and the second argument is a pointer to a string of + and ! symbols.

Using the second argument, recursively traverse the tree to find the character that is mapped to the string and return it. If no character maps to the string, return the `NUL` terminator. You may assume the second argument is not `NULL` and points to a `NUL`-terminated string that only contains + and ! characters.

(e) (10 points) After many sleepless nights of research, you've finally managed to map an additional letter. You will need to update the tree. Write an **iterative** function named `insert()` that takes three arguments and returns `void`. The first argument is a pointer to a `struct tree` node (the root of the tree), the second argument is a pointer to a NUL-terminated string containing only + and ! symbols, and the third argument is a `char`.

Iteratively traverse the tree and allocate and insert a new `struct tree` node containing the third argument in the correct location according to the second argument. If you encounter a NULL pointer before reaching the insertion point, allocate and insert any intermediate nodes as necessary, storing the NUL terminator in each.

You may assume the second argument is not NULL and does not already map to an existing character (but it may correspond to an existing node with the NUL terminator). You may also create helper functions if you wish. Use assertions as needed. Do not use recursion.

Additional space on the next page...

Work area for 3.e continued...

Additional space on the next page...

Work area for 3.e continued...