CS 240: Programming in C
Midterm 2
Spring 2025

Practice Midterm 2

Version I

Name:

Username:

Read all instructions before beginning the exam.

- This exam is intended to be equally or more difficult than the past official midterm provided.
- You are encouraged to post on Ed Discussion, without hesitation, any sort of question you may have about this practice 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 less than 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.
- Read each question carefully and only do what is specifically asked for in that problem.
- Assume appropriate includes have been added to the code segments shown in the problems.
- Circle your answer in true or false questions.
- Some problems require several steps. Show all your work. Partial credit can only be rewarded to work shown.
- Write your username on EVERY page where indicated. Any page without a username will receive a zero for the material on that page.

Signature:

Do not open the examination booklet until instructed.

1. (30 points) Write brief answers to the following questions.

   (a) (3 points) What is the output of the following program? State the value displayed by `printf()`, undefined behavior, or error produced. Assume no program termination occurs due to assertions.

```
typedef struct {
    char *text;
} line_t;

int main() {
    line_t *record = malloc(sizeof(line_t));
    assert(record);

    if (!record->text) {
        record->text = malloc(15);
        assert(record->text);
        strcpy(record->text, "Dennis Ritchie");
    }

    printf("%s\n", record->text);
}
```

Undefined behavior

   (b) (3 points) The code segment in problem 1.a. has memory leaks. State the line(s) of code to be implemented in order, at the end of `main()`, to prevent them from occurring.

```
if (strcmp(“Dennis Ritchie”, record->text) == 0) {
  free(record->text);
}

free(record);
record = NULL;
```

(c) (1 points) True or [✓ False]: Considering a singly linked list, the expression `head + 1` is always functionally equivalent to `head->next`.

(d) (5 points) State whether the variables in the following code segment point to a section in the heap, stack, neither, or if it is undefined behavior, at the end of the function.

```
void change() {
    char *a = "Ken Thompson";
    int *b = calloc(1, sizeof(int) * 7);
    int *c = (int *)&a;
    int *d = (int *)&b;
    int *e = b;

    b = (int *) (b - c);
}
```

a: Neither

b: Undefined behavior

c: Stack

d: Stack

e: Heap

(e) (1 points) [✓ True] or False: The function in the code segment in problem 1.d. will produce memory leaks when called.

(f) (2 points) Briefly explain what the `bt` GDB command does.

Obtains the bracktrace of the program

(g) (3 points) What is the output of the following program? State the value displayed by `printf()`, undefined behavior, or error produced.

```
void modify(int **ptr) {
    int y = 20;
    *ptr = &y;
}

int main() {
    static int x;
    int *p = &x;
    int **pp = &p;

    modify(pp);
    printf("%d\n", x);
}
```

```
0
```

(h) (3 points) Briefly explain the problem that may arise from calling `free()` on a memory block that has already been deallocated.

```
It may deallocate another allocated region
in the heap by another program
```

(i) (3 point) Write the function prototype of `func` that takes in an integer and a constant string as arguments, and returns a function pointer that takes in and returns a `struct node` pointer.

```
struct node *(*func(int, const char *)) (struct node *);
```

(j) (1 point) True or [✓ False]: A function with the signature `void func(const int *p);` guarantees that the value pointed to by `p` will never change throughout the program.

(k) (5 points) In the code segment, there are 6 different arrow operators (`->`). Rewrite the function, replacing all arrow operators for pointer implementations (`*`).

```
void process(node_t *head) {
    while ((head) && (head->next)) {
        head->data += head->next->data;
        head = head->next->next;
    }
}
```

```
void process(node_t *head) {
    while ((head) && (*head).next)) {
        (*head).data += (*((*head).next)).data;
        head = (*((*head).next)).next;
    }
}
```

2. (30 points) The following questions deal with dynamic allocation and linked lists.

   (a) (4 points) Declare a structure named **flight** containing a character pointer named **id**, and a **float** named **boarding**. The structure should be a valid singly linked list node. Simultaneously create a type **flight_t** that refers to it.

```
typedef struct flight {
  char *id;
  float boarding;
  struct flight *next;
} flight_t;
```

(b) (4 points) The following code segment attempts to create a newly dynamically allocated linked list node of the structure created previously, populating it with argument values, where `id` must be not `NULL`, and `boarding` must be non-negative. There exists one problematic behavior in the code segment. Briefly explain what the problem is and propose a rewrite of one existing non-blank line to fix it.

```
1  flight_t *create_node(char *id, float boarding) {
2      assert(id);
3      assert(boarding >= 0.0);
4
5      flight_t *new = malloc(sizeof(flight_t));
6      assert(new);
7
8      new->boarding = boarding;
9      new->id = malloc(strlen(id) + 1);
10     assert(new->id);
11     strcpy(new->id, id);
12
13     return new;
14 }
```

```
Next pointer is not initialized, thus it contains garbage.

Line 5:
flight_t *new = calloc(1, sizeof(flight_t));
```

(c) (12 points) Write a function, `swap_flights()` that takes two arguments, the address of a pointer to a `flight_t` node named `fA`, and the address of a pointer to a `flight_t` node named `fB`. The two arguments reference pointers pointing to two flight nodes already in the linked list. Rearrange the linked list so that these two nodes exchange positions in the list without swapping their data fields. The function's return type is `void`. Use assertions where necessary.

You may assume that the arguments contain the address of the previous node's next pointer, or the head pointer if the first node is being replaced.

```
void swap_flights(flight_t **fA, flight_t **fB) {
  assert(fA);
  assert(*fA);
  assert(fB);
  assert(*fB);

  flight_t *a = *fA;
  flight_t *b = *fB;
  if (a == b) {
    return;
  }

  if (a->next == b) {
    a->next = b->next;
    b->next = a;
    *fA = b;
  }
  else if (b->next == a) {
    b->next = a->next;
    a->next = b;
    *fB = a;
  }
  else {
    flight_t *temp = a->next;
    *fA = b;
    *fB = a;
    a->next = b->next;
    b->next = temp;
  }
}
```

Continuation of 2.c.

(d) (10 points) Write a function, `inspect_flights()`, to perform a single linear traversal of the singly linked list, swapping every two nodes in order to maintain an ascending order of `boarding` values, where the head of the list has the smallest value. The function takes in two arguments, the address of the pointer to a `flight_t` node (the head of the linked list), and a function pointer to the `swap_flights()` function in problem 2.c. The return type is an integer value, where `1` is returned if the linked list was modified, or `0` if it was not modified.

Use the second argument to swap flight nodes. Make sure to call the function correctly, such that the list remains a valid singly linked list, and that the pointer passed in the first argument still references the head of the list upon completion. Use assertions where necessary. An empty list should not fail an assertion. Remember you are not expected to produce a final sorted list, as you only need to perform one linear traversal of the list. You may assume that `swap_flights()` was implemented appropriately.

```
int inspect_flights(flight_t **head,
  void (*swap)(flight_t **, flight_t **)) {

  assert(head);
  assert(swap);

  int check = 0;
  flight_t **cur = head;

  while (*cur && (*cur)->next) {
    if ((*cur)->boarding > (*cur)->next->boarding) {
      swap(cur, &((*cur)->next));
      check = 1;
    }
    cur = &((*cur)->next->next);
  }

  return check;
}
```

Continuation of 2.d.

3. (40 points) The following questions deal with dynamic allocation and binary trees.

   (a) (3 points) Declare an enumerated type named `type` that contains two named constants, `INT` and `PAIR`, in that order. Simultaneously create a type `type_t` that refers to it.

```
typedef enum type {
  INT,
  PAIR,
} type_t;
```

   (b) (7 points) You are a Purdue CS student, it is about time you create your own programming language. Represent the object types of a minimal programming language that has two types: integers, and pairs. A pair can be a pair of anything (e.g. two integers, an integer and another pair, another two pairs, etc).

   For this, a structure can be declared that stores an integer, or two pointers to two of the same structure type (the pair of pointers wrapped together in another struct) in a shared memory space, as to represent either an integer or a pair type respectively. Whether the struct holds an integer or a pair will be determined by the value of the enumerated type created in problem 3.a.

   Declare a structure named `object` containing a `type_t` enum named `type`, an unsigned character named `mark`, and, for the storage of the object's value, an integer named `value` and a pair of `struct object` pointers named `pair_first` and `pair_second`. The pair should be wrapped around another internal struct and share the same memory space as the integer. Simultaneously create a type `object_t` that refers to it.

   Hint: Use a union inside of the object struct.

```
typedef struct object {
  type_t type;
  unsigned char mark;
  union {
    int value;
    struct {
      struct object *pair_first;
      struct object *pair_second;
    };
  };
} object_t;
```

Continuation of 3.b.

(c) (8 points) Write a function, `new_object()`, which accepts two parameters, a `type_t` enum, and an integer. The first argument determines the type of the object being created, and the second argument corresponds to the integer value that the new object holds. The second argument must only be used if the first argument indicates that the object is an integer type (`INT`), otherwise it must be ignored. Return a pointer to a newly allocated `object_t` object. Assert for the first argument to be valid, and assert in other places where it may be necessary (do not assert the second argument). Be sure to initialize all fields.

```
object_t *new_object(type_t type, int value) {
  assert(type >= 0 && type <= 1);

  object_t *new = calloc(1, sizeof(object_t));
  assert(new);

  new->type = type;

  if (type == INT) {
    new->value = value;
  }

  return new;
}
```

(d) (10 points) Pair type objects provide two references to two other objects, that may or may not be of the same type. This layout can mirror a binary tree, where the left and right child are the `pair_first` and the `pair_second` pointers respectively, turning the associated objects into nodes.

Write a recursive function named `set_marks()` that takes one argument, a pointer to an `object_t` node, the root of the objects' reference tree. Traverse the tree in postfix order, and set the `mark` character to `1` in each object node. Return an integer of the number of nodes that were modified. Consider that some nodes may already have `mark` set to `1` in which case they should not be counted as modified nodes.

```
int set_marks(object_t *node) {
  if (node == NULL)
    return 0;

  int modified_count = 0;

  if (node->type == PAIR) {
    modified_count += set_marks(node->pair_first);
    modified_count += set_marks(node->pair_second);
  }

  if (node->mark == 0) {
    node->mark = 1;
    modified_count++;
  }

  return modified_count;
}
```

(e) (12 points) Write a recursive function named `sweep_marks()` that takes one argument, a pointer to the address of an `object_t` node, the root of the objects' reference tree. For every node that has `mark` set to `0`, set all of its children nodes' `mark` to `0`. This sweeping operation should end up with all nodes emerging from a parent node, that has its `mark` set to `0`, until the associated leaf nodes, with their `mark` also being set to `0`.

Additionally deallocate, maintaining a valid binary tree structure, all node objects of type `INT` that have their mark set to 0 by the end of the sweeping operation. Return the integer sum of the deallocated nodes' values.

```
int sweep_marks(object_t **root) {
  if (*root == NULL) {
    return 0;
  }

  object_t *cur = *root;
  int sum = 0;

  if (cur->mark == 0 && cur->type == PAIR) {
    if (cur->pair_first) {
      cur->pair_first->mark = 0;
    }
    if (cur->pair_second) {
      cur->pair_second->mark = 0;
    }
  }

  if (cur->type == PAIR) {
    sum += sweep_marks(&cur->pair_first);
    sum += sweep_marks(&cur->pair_second);
  }

  if (cur->type == INT && cur->mark == 0) {
    sum += cur->value;
    free(cur);
    *root = NULL;
  }

  return sum;
}
```

Continuation of 3.e.