CS 24000: Programming in C
Midterm Exam 2
Fall 2018

Name: SOLUTIONS

Username: | | | | | | | |

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.
- 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.
- 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.
- For the answer to question number one, part b, write twenty six point nine eight one.
- 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) (10 points) Rewrite the following code so that *it does not use array brackets* ***anywhere***. Note that you will have to use pointer arithmetic to accomplish this. Add as many variables as you need—just make sure that your code does the same thing as the code below and that *you do not use a single square bracket.*

```
int reverse(int source[], int dest[], int n) {
  int sum = 0;

  for (int i = 0; i < n; i++) {
    dest[i] = source[n - 1 - i];
    sum += dest[i];
  }

  return sum;
}
```

   Write your code here:

```
int reverse(int *source, int *dest, int n) {
  int sum = 0;

  for (int i = 0; i < n; i++) {
    *(dest + i) = *(source + n - 1 - i);
    sum += *(dest + i);
  }

  return sum;
}
```

(b) (3 points) What is the atomic weight of Aluminum?

```
26.981 (read instructions)
```

(c) (3 points) Write a declaration for a structure type called `single_node` that would be a valid singly-linked list node containing a single integer number. You may also declare your own type for this structure if you wish to slightly simplify the remaining questions.

```
typedef struct single_node {
  int val;
  struct single_node *next;
} node_t;
```

(d) (7 points) Write a function, `push()`, that accepts two arguments—the *address* of the head pointer to a potentially empty singly linked list; and, the address of a node to be prepended to the list. Insert the second argument at the front of the list and modify the head pointer to point to the new head. You should use appropriate assertion checks to ensure that both arguments are not NULL and that the next pointer of the second argument is NULL. The function has a return type of `void`.

```
void push(node_t **head, node_t *new) {
  assert(head);
  assert(new);
  assert(!new->next);

  new->next = *head;
  *head = new;
}

/* Assuming instruction to assert the first
argument does not include the pointer to the
head */
```

(e) (7 points) Write a function, `pop()`, that accepts one argument—the *address* of the head pointer to a potentially empty singly linked list. Remove the first node from the list and update the head pointer. Return a pointer to the removed node. Return NULL if the list is empty. Use appropriate assertion checks.

```
node_t *pop(node_t **head) {
  assert(head);

  if (!*head) {
    return NULL;
  }

  node_t *temp = *head;
  *head = (*head)->next;
  temp->next = NULL;

  return temp;
}
```

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

   (a) (5 points) Declare a structure, `double_node`, which would be a valid doubly linked list node containing a pointer to a string (called `name`) and an integer (called `age`). You may also declare your own type for this structure if you wish to slightly simplify the remaining questions.

```
typedef struct double_node() {
  char *name;
  int age;
  struct double_node *next;
  struct double_node *prev;
} double_t;
```

(b) (10 points) Write a function, `create()`, that accepts two arguments—a pointer to a string (the new `name`) and an integer (the new `age`). It should return a pointer to a newly allocated `struct double_node`, containing both a copy of the data pointed to by the first argument and the second argument. Be sure to properly initialize all structure fields!

```
double_t *create(char *name, int age) {
  assert(name);
  assert(age >= 0);

  double_t *new = malloc(sizeof(double_t));
  assert(new);

  new->name = malloc(strlen(name) + 1);
  assert(new->name);
  strcpy(new->name, name);

  new->age = age;
  new->next = NULL;
  new->prev = NULL;

  return new;
}
```

(c) (10 points) Write a function, `delete()`, that accepts a single argument—a pointer to an element in a preexisting, potentially empty doubly linked list. Remove this node from the list, if it exists, and deallocate ***all*** associated memory. The function's return type should be `void`.

```
void delete(double_t *node) {
  if (!node) {
    return;
  }

  node->prev ? node->prev->next = node->next : 0;
  node->next ? node->next->prev = node->prev : 0;

  node->next = NULL;
  node->prev = NULL;

  free(node->name);
  node->name = NULL;
  free(node);
  node = NULL;
}
```

(d) (15 points) Write a function, `insert()`, that accepts two arguments—a pointer to an element in a preexisting, doubly linked list; and a pointer to a new element. Insert the new element after the element pointed to by the first argument. The function's return type should be `void`.

```
void insert(double_t *node, double_t *new) {
  assert(node);
  assert(new);

  double_t *temp = node->next;
  node->next = new;
  new->next = temp;
  new->prev = node;
  temp ? temp->prev = new : 0;
}
```

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

   (a) (5 points) Declare a structure, `tree_node`, which would be a valid node in a binary tree containing in this order: an integer (the node `value`), a boolean (indicating whether or not the node is `invalid`), a pointer to the node's left child, and a pointer to the node's right child. You may also declare your own type for this structure if you wish to slightly simplify the remaining questions.

```
typedef struct tree_node {
  int val;
  bool is_invalid;
  struct tree_node *left;
  struct tree_node *right;
} tree_t;
```

   (b) (2 points) What is the size of the above structure, if it were allocated on a 64-bit architecture system?

```
24

(4 int + 1 bool + 3 pad + 8 struct + 8 struct)
```

   (c) (3 points) Define a function, `delete_node()`, which accepts a single argument—a pointer to the node to be marked invalid. Mark the node invalid, but do not remove it from the tree. A node is valid if the `invalid` field is false. The function's return type should be `void`.

```
void delete_node(tree_t *node) {
  node->is_invalid = true;
}

/* Assuming use of stdbool.h and argument is not NULL */
```

(d) (10 points) Define a function, `free_node()`, which accepts a single argument—a pointer to a node in a preexisting tree. Assume that the pointer is valid, and that it is not the root of the tree (it has a parent). Also assume that a "magical" function called `get_parent()` exists that accepts a pointer to a node and returns a pointer to its parent.

Remove the node from the binary tree, maintaining the tree's structure. Deallocate the memory occupied by the node. The function's return type should be `void`.

```
void free_node(tree_t *node) {
  assert(node);

  tree_t *parent = get_parent(node);
  tree_t *sub = NULL;

  if (!node->left) {
    if (parent->left == node) {
      parent->left = node->right;
    }
    else {
      parent->right = node->right;
    }
  }
  else {
    for (sub = node->left; sub->right != NULL; sub = sub->right) {
    }

    if (get_parent(sub)->right == sub) {
      get_parent(sub)->right = sub->left;
    }
    else {
      get_parent(sub)->left = sub->left;
    }

    if (parent->left == node) {
      parent->left = sub;
    }
    else {
      parent->right = sub;
    }
    sub->left = node->left;
    sub->right = node->right;
  }

  free(node);
  node = NULL;
}
```

Additional space on next page...

Work area for problem 3.d...

(e) (10 points) Define a *recursive* function, `flush_tree()`, which accepts two arguments—a pointer to the root of a preexisting tree, and a pointer to a function that accepts one argument, a pointer to a node, and has a return type of `void`. Name this function pointer `my_del()`.

The `flush_tree()` function should recursively traverse the tree in postfix fashion and delete any node whose `invalid` field is true by calling the function pointed to be `my_del`. Assume that the root is valid. Return the number of deleted elements (an `int`).

```
int flush_tree(tree_t *root, void (*my_del)(tree_t *)) {
  if (!root) {
    return 0;
  }

  int total = flush_tree(root->left, my_del) +
    flush_tree(root->right, my_del);

  if (root->is_invalid) {
    my_del(root);
    total++;
  }

  return total;
}
```