

# Homework 2

Due Sunday, February 20 by 10:00pm

Handout 5  
CSCI 334: Spring 2022

---

## Turn-In Instructions

---

For this assignment, create one separate source code file for each question (e.g., `q1.c`).

Supply a `Makefile` (20 points) with one rule per homework question. The naming convention for targets should be the name of the source file without the `.c` extension. For example, `q1.c` should compile to `q1`. You must also provide an `all` target that builds all targets and a `clean` target that removes all of the binary files generated by the build.

For full credit, be sure that your code compiles without emitting warnings even when using the `-Wall` flag. Note that if you use your own computer to do this assignment, you should check your assignment using a lab machine before submitting, since different compiler versions do not always behave the same way. The final authority will be the lab environment.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334lab2_<your user name>`. For example, my repository would be `cs334lab2_dbarowy`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

---

## Honor Code

---

This is a partner lab. You may work with another classmate if you wish, and you may co-develop solutions. Remember: although you can work on code together, you must each independently write up and submit your solution. No code copying is allowed. Be sure to tell me who your partner is by committing a `collaborators.txt` file to your repository (2 points).

This assignment is due on Sunday, February 20 by 10:00pm.

---

## Reading

---

1. **(Optional)** Read “A Brief Overview of C” from the course packet, especially if you’ve never seen C before.
2. **(Required)** Read “C: A Language Built Around a Memory Model” from the course packet.
3. **(Required)** Read “Passing Pointers by Value” from the course packet.
4. **(Optional)** Read “The Linked List: An Elegant, Recursive Data Structure” from the course packet if the definition of a linked list in the question below needs more explanation.

---

## Problems

---

### Q1. (15 points) ..... Allocating a zeroed array

Write a function that allocates an `int` array of size  $n$ , fills the array with the value 0, and returns a pointer to the array. `main` should obtain  $n$  from the command line. Demonstrate that you've correctly implemented your function by printing all of the values in the calling function. Be sure to correctly manage the lifetime of the array. In other words, given the following starter code, fill in the ??? sections of the implementation. Note that ??? lines may be filled in with implementations that are longer than one line.

```
1  ??? zero_fill(int n) {
2      // ???
3  }
4
5  int main(int argc, char **argv) {
6      if (argc != 2) {
7          printf("Usage: q1 <n>\n");
8          return 1;
9      }
10
11     int n = atoi(argv[1]);
12
13     ??? = zero_fill(n);
14
15     for(int i = 0; i < n; i++) {
16         printf("%d\n", ???);
17     }
18
19     ???
20
21     return 0;
22 }
```

Supply your answer as a C source code program called `q1.c`. You should also provide a “boxes and arrows” drawing as described in class and as shown in the reading, [Passing Pointers by Value](#), showing the current state of the computer’s memory at the end of the `zero_fill` function (line 3). You are welcome to draw this out on paper and to include a photo of it with your homework submission called `q1.png` (or `q1.jpg`). If you are looking for a challenge, make a “pretty” diagram using Keynote or PowerPoint.

**Hint:** Allocation should happen in `zero_fill`, which means that the allocated memory outlives the function in which it was requested.

### Q2. (63 points) ..... Allocation

Write code to perform the following. Refer to `man 3 rand` for documentation on sampling a random `int`.

- Generate a random `int` and copy it to every element of an array of length 10. The array should have automatic storage duration. Print the array.
- Generate a random `int` and copy it to every element of an array of length 10. The array should have allocated storage duration. Print the array. Be sure to deallocate when done.
- Generate a random C string (a `char *`) and copy it to every element of an array of length 10. Make sure that every element is a copy of the string, not a copy of the pointer. The array should have automatic storage duration. Print the array.

- (d) Generate a random C string (a `char *`) and copy it to every element of an array of length 10. Make sure that every element is a copy of the string, not a copy of the pointer. The array should have manual storage duration. Print the array. Be sure to deallocate when done.

Supply your answers as a C source code programs called `q2a.c`, `q2b.c`, `q2c.c`, and `q2d.c`. “Boxes and arrows” drawings should accompany each program; name these drawings `q2a.png` (or `q2a.jpg`), etc. Draw the diagram at the line number in your program that you think best demonstrates how your program utilizes memory. Be sure to indicate which line this is in your diagram.

### Q3. (personal satisfaction + 5 bonus points) ..... Optional: Linked Lists C

One of the most fundamental data structures in computer science is the linked list. A linked list is a dynamic data structure, unlike an array. A linked list can be used to store data in situations where the total amount of data that needs to be stored is not known ahead of time. The word dynamic refers to the fact that the size of the data structure can change while the program runs.

Linked lists have an elegant, recursive definition. A linked list is either:

- NULL, or
- a list node that stores a data element and points to a linked list (the “tail”).

In this question, you will implement a linked list that stores a C string (`char *`) in each list node. You will also implement a small collection of utility functions for manipulating the list.

- (a) Design a list node data structure using `struct`. Use `typedef` to name this data structure `listnode` so that you can refer to a list node pointer with `listnode *`.
- (b) Write a list prepend function with the following type signature:

```
listnode *prepend(char *data, listnode *list);
```

`prepend` stores the given `data` in a new `listnode` where the given `listnode *` is the tail of the new list. `prepend` should `malloc` a `listnode`, store `data` in that node, store `list` in the tail, and return a pointer to the new node. The function should work even if `list` is NULL; in other words, a user can create a new list just by appending to NULL, for example:

```
listnode *list = prepend("hello", NULL);
```

The `prepend` function should not make a copy of the given C string; instead, it should just store a pointer to that string.

- (c) Write a `head` function that takes a `listnode *` and returns the `data` item for the `listnode` at the head of the list. Calling `head` on a NULL list should return NULL.

```
char *head(listnode *list);
```

- (d) Write a `tail` function that takes a `listnode *` and returns the tail of the list. Note that the tail of a list is the original list without the head `listnode`. Calling `tail` on a NULL list should return NULL.

```
listnode *tail(listnode *list);
```

- (e) Write a `printlist` function with the following type signature:

```
void printlist(listnode *list, char *sepy);
```

that prints each stored C string, separated by the string specified by the `sepby` C string, in order. The function should not print a trailing `sepby`; instead, it should print a newline at the end of the list. For example, given the list that contains the strings "hello" and "world" and where `sepby` is "zzz", the function should print:

```
hellozzzworld[new line]
```

- (f) Write a `delete` function that takes a `listnode *` and **free**s all the `listnodes` in the given list, but does not free the memory pointed to by the `data` field of each `listnode`. If the rationale for this behavior seems unclear to you, have a look at part **Q3.k**.

```
void delete(listnode *list);
```

- (g) Write a `reverse` function that takes a `listnode *` and returns a new list in the reverse order. As with other functions in this assignment, it should not make a copy of each C string.

```
listnode *reverse(listnode *list);
```

- (h) Write a `length` function that takes a `listnode *` and returns an **unsigned int** representing the length of the list.

```
unsigned int length(listnode *list);
```

- (i) Write a `fromfile` function that takes a `char *` representing the name of a file, reads each whitespace-separated word of the file into a `listnode` and returns the linked list representing the sequence of words in the file, in order.

```
listnode *fromfile(char *filename);
```

- (j) Write a `main` function that allows a user to pass in the name of a file from the command line, and prints the following information:

- The number of words in the file.
- The first and last word from the file (properly handling the case that a file may be empty).

- (k) Finally, answer the following question in a file called `q3.md`.

- i. Why shouldn't the `delete` function **free** data pointed to in the list? For completeness, you should consider the following two cases: 1) data stored in the list comes from string literals, and 2) a user reverses `list1`, yielding `list2`, and then **deletes** both lists. You will need to do a little research on your own to determine how string literals are stored in C.

Supply your complete solution as a single C source code program called `q3.c`. For full credit, ensure that all storage with allocated duration is deallocated and that all files are closed before the program terminates.

#### Q4. (1 point) ..... Optional: Feedback

How hard was this assignment on a scale of 1 to 5? (where 1 is easy and 5 is difficult).

Do you have any additional comments or feedback that you would like me to know?

Please supply your answer as a `feedback.txt` file.