Structures

- Structure types allow related variables to be grouped together into a single compound value
- Defining a structure:

```
#define PLANETSTRLEN 20

typedef char pstr_t[PLANETSTRLEN+1];

struct {
    pstr_t name
    pstr_t orbits;
    double distance;        /* million km */
    double mass;            /* kilograms */
    double radius;          /* kilometers */
} one_planet;
```

Page 130 of the textbook

Structures

- Structure types allow related variables to be grouped together into a single compound value
- Writing a structure:

```
#define PLANETPROMPT \
    "name, orbits, distance, mass, radius"

    planet_t new_planet;
    printf("Enter %s:\n", PLANETPROMPT);
    scanf("%s %s %lf %lf %lf",
        new_planet.name,
        new_planet.orbits,
        &new_planet.distance,
        &new_planet.mass,
        &new_planet.radius);
```

Page 132 of the textbook

Structures

- Structure types allow related variables to be grouped together into a single compound value
- Writing a structure:

```
#define PLANETPROMPT \
    "name, orbits, distance, mass, radius"

    planet_t new_planet;
    printf("Enter %s:\n", PLANETPROMPT);
    scanf("%s %s %lf %lf %lf",
        new_planet.name,
        new_planet.orbits,
        &new_planet.distance,
        &new_planet.mass,
        &new_planet.radius);
```

Arrays → it already points to the address

Provide address to write

Page 132 of the textbook

Structures

- Structures can be passed into and returned from functions
- Tend to pass a structure pointer to avoid making only local changes

Structures

- Structures can be passed into and returned from functions
- Tend to pass a structure pointer to avoid making only local changes

Dereference structures

- Consider a function with input argument `planet_t *planet`
- To access the `radius` variable, use `planet -> radius`

```c
// Define the structure
typedef struct {
    double radius;  // Radius of the planet
    double mass;    // Mass of the planet
} planet_t;

// Function that modifies the radius of the planet
void change_radius(planet_t *planet, double new_radius) {
    planet->radius = new_radius;  // Modify the radius using the pointer
}

// Function to print the planet details
void print_planet(const planet_t *planet) {
    printf("Planet's radius: %.2f km\n", planet->radius);
    printf("Planet's mass: %.2e kg\n", planet->mass);
}

int main() {
    // Create a planet object (Earth, for example)
    planet_t earth = {6371.0, 5.972e24};  // Radius in km, mass in kg

    // Call the function to change the radius
    change_radius(&earth, 6400.0);  // Pass the address of 'earth'

    // Print the updated values
    print_planet(&earth);

    return 0;
}
```

Solving Ex8.08 Word frequencies with structs

1. Define the following structs:

freq_table_t	
freqs	word_freq_t
num_distinct	int
total_num_words	int

word_freq_t	
word	word_t → already defined as `word_t[MAXCHARS+1]`
freq	int

2. Extract the linear search logic into a new function:

```
void add_freq(word_t target, freq_table_t *table)
```

   - This function iterates through `table` to check if `target` exists in the table already
   - If so, it increments `freq`, else it appends the new word
   - Always increment `total_num_words`
   - NOTE! `freq_table_t` is passed as a pointer. To access or modify its members use the `->` operator

3. Update the `main` function to use `add_freq` and add the second print statement

Dynamic Memory Allocation in C

When do we need it?

- When we don't know in advance how much memory is required (e.g., based on user input or data file size).
- To create data structures (arrays, structs, linked lists) whose size can change at runtime.
- To avoid wasting memory from large static allocations.

Functions	
`malloc()`	Allocates a block of memory (uninitialized)
`realloc()`	Changes the size of a previously allocated block
`free()`	Releases memory that was previously allocated

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    int n = 5;

    // Step 1: Allocate memory for 5 integers
    arr = (int *)malloc(n * sizeof(int));
    // Check if malloc succeded!
    assert(arr != NULL);

    // Step 2: Fill values
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    // Step 3: Resize to hold 10 integers
    int new_size = 10;
    arr = (int *)realloc(arr, new_size * sizeof(int));
    assert(arr != NULL);

    // Step 4: Fill new elements
    for (int i = n; i < new_size; i++) {
        arr[i] = i + 1;
    }

    // Step 5: Free the memory
    free(arr);
    arr = NULL; // to avoid dangling pointer
    return 0;
}
```

Create an empty pointer that will later point to a dynamically allocated int array

Scaffold: https://github.com/Msch167/Comp10002/blob/main/Ex10x1.c

Solving Ex10.x1 Dynamic memory allocation

Exercise 4: *Duplicate a single string*. Since we don't know in advance how long the string will be, we need to allocate the memory ourselves.

Allocate memory with `malloc`

Exercise 5: *Duplicate a set of strings*. We use our previous function for each individual string, but we must allocate more memory to point to them all.

Refresher: What is the ** ?

- we want to duplicate a set of strings ⇒ an array of strings
- `char **S` is a pointer to a pointer to a `char` (array of strings).
- each `char *` in the array is a pointer to a string

Exercise 6: *Free all the memory used by the set of strings*. Note, each string must be freed individually, and then S itself must be freed as well.

Memory Management in C:

In C, memory that has been dynamically allocated with `malloc` (or similar functions) **needs to be manually freed when it is no longer needed**.

- For every `malloc`, there should be a corresponding `free`
- Set each freed pointer to `NULL` (safety measure to prevent accidental access to memory that has been freed)