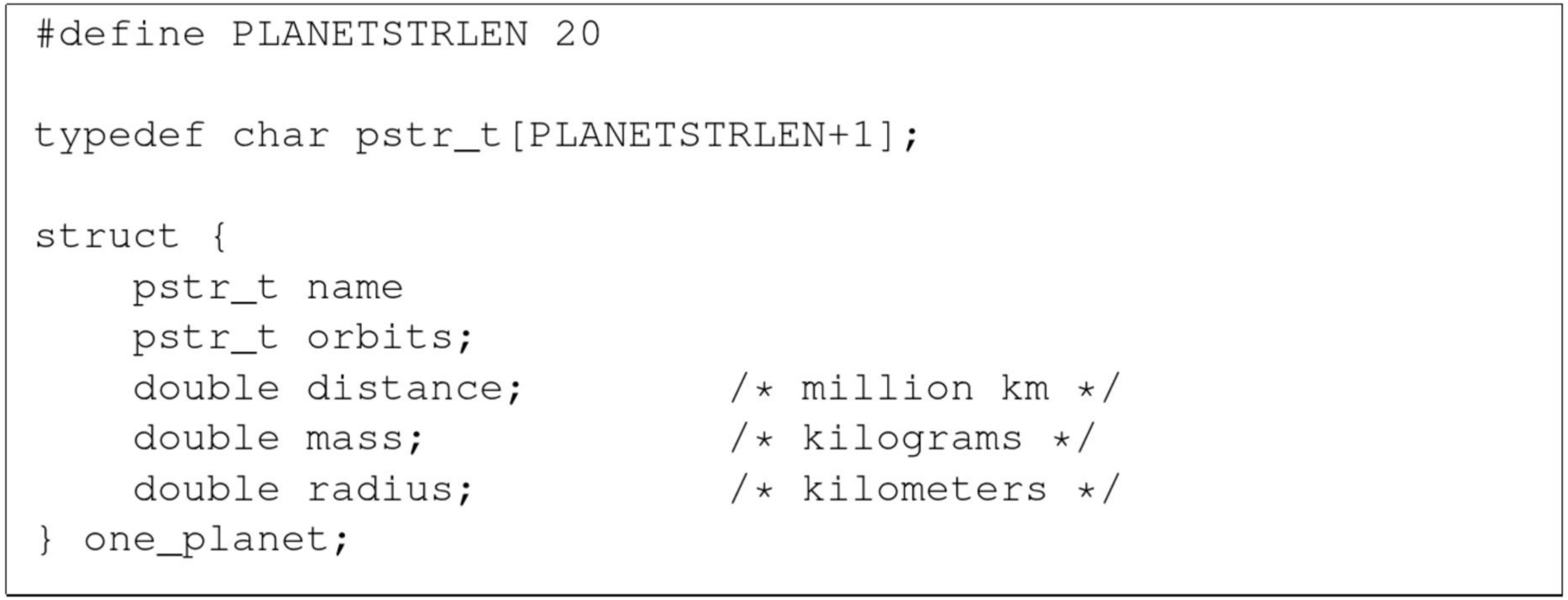
Structures

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

- Defining a structure:



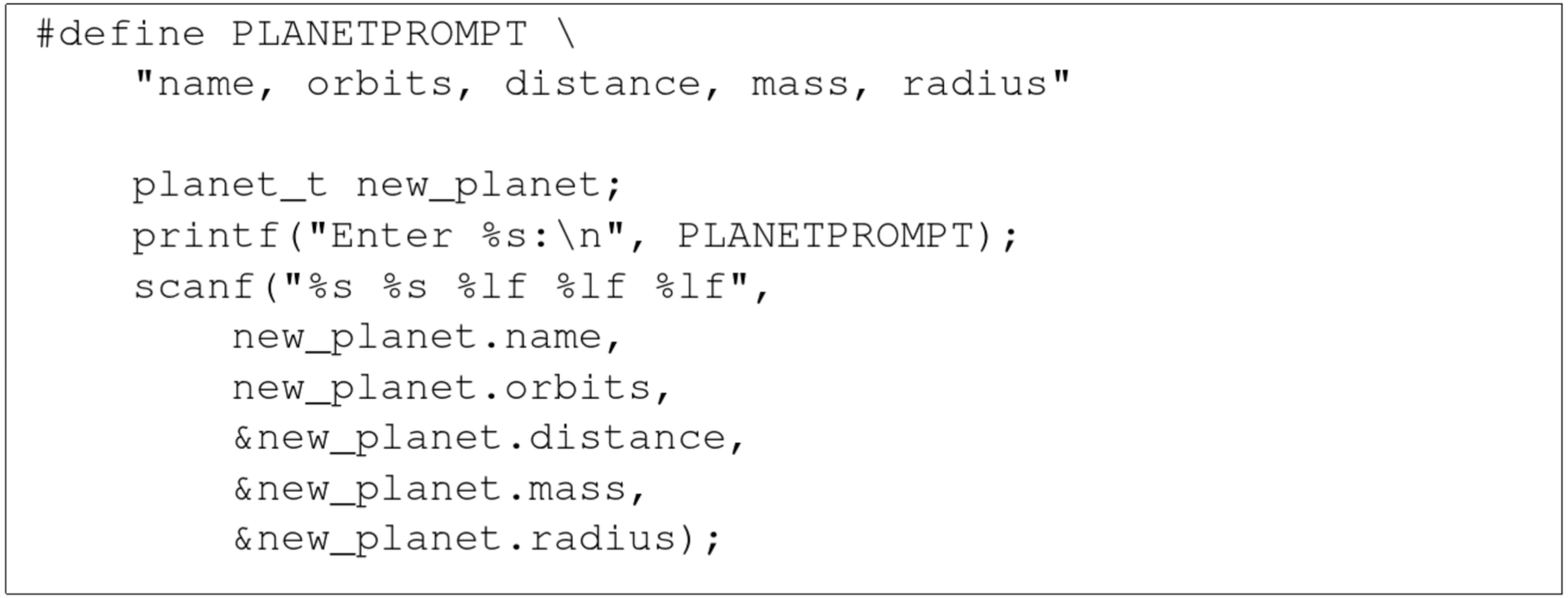
```
#define PLANETSTRLEN 20

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:

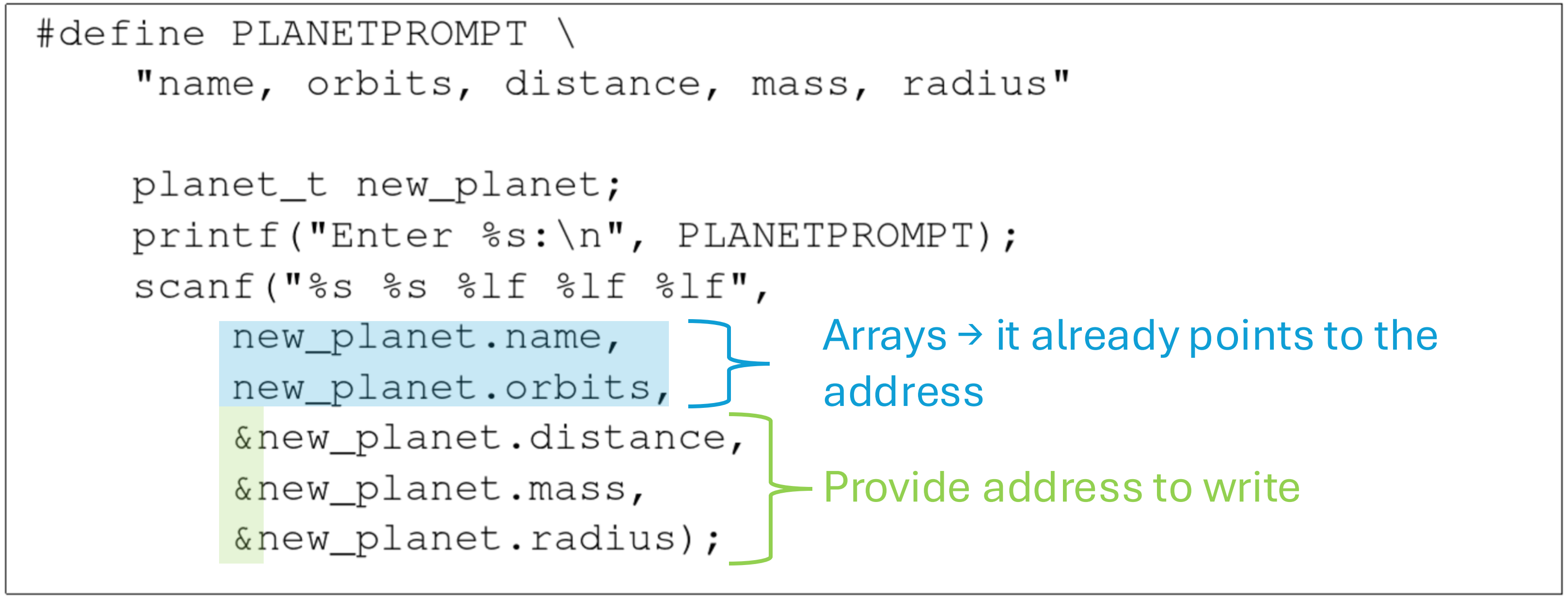


```
#define PLANETPROMPT \
    "name, orbits, distance, mass, radius"

    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

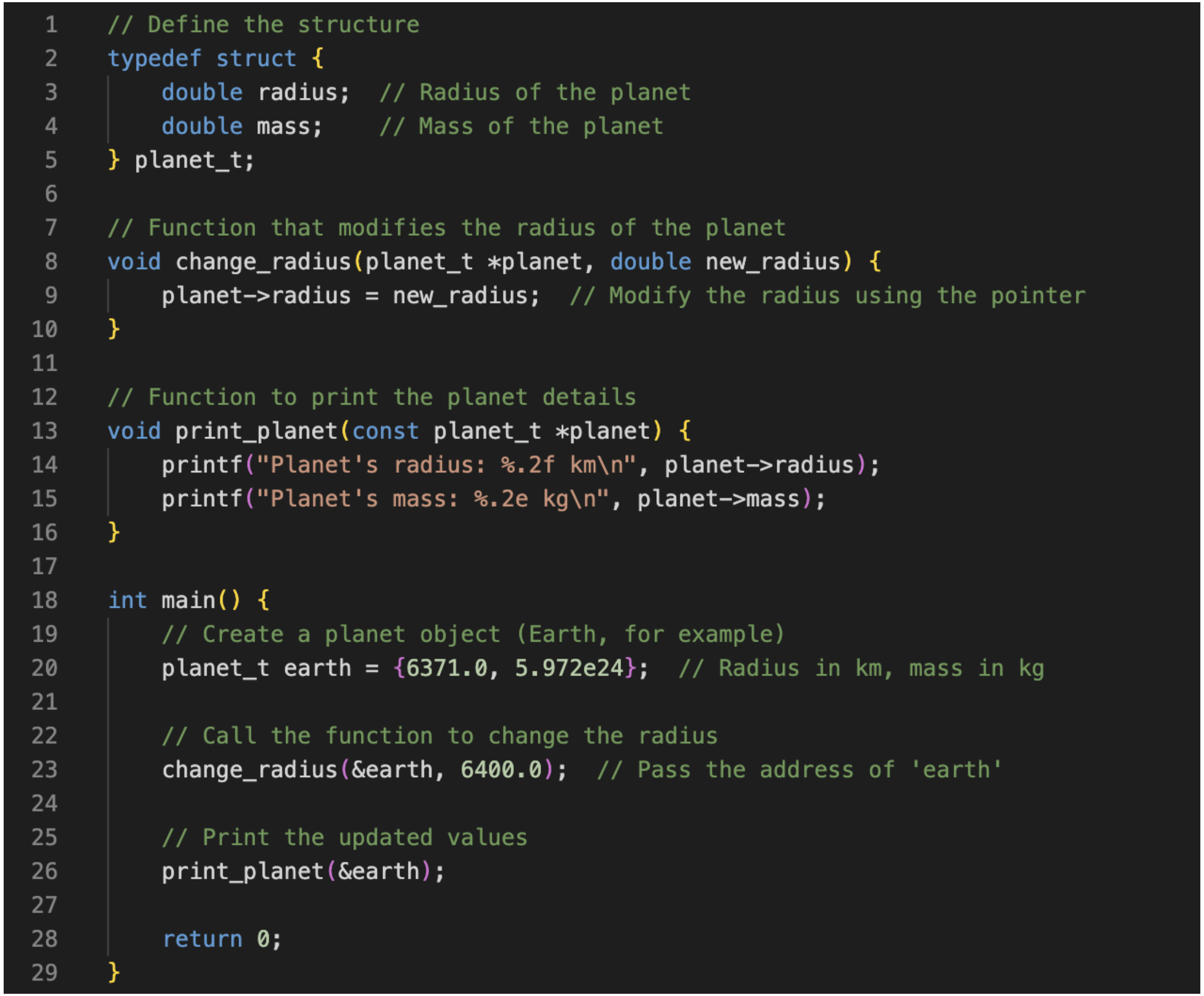
- Structures can be passed into and returned from functions
- Tend to pass a structure pointer to avoid making only local changes

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
freq	int

already defined as `word_t[MAXCHARS+1]`

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 (use `strcpy` and `strcmp`)
- 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

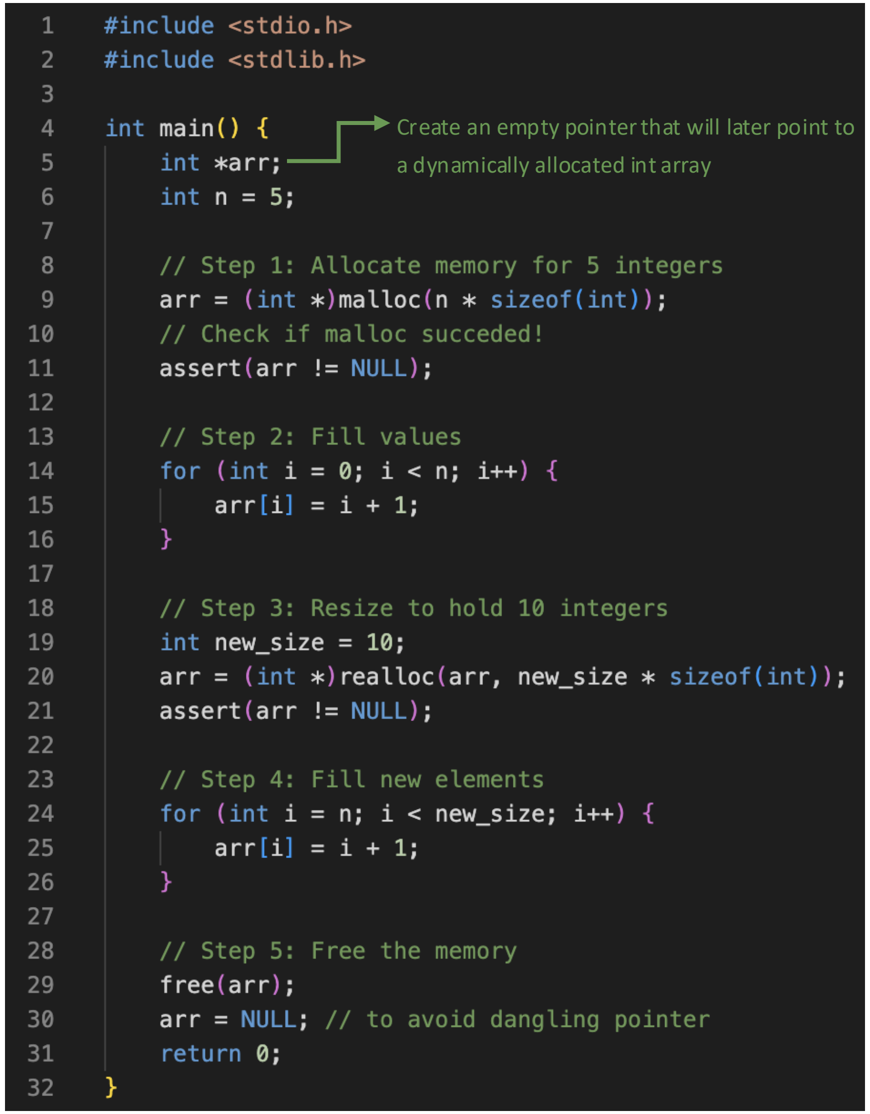
3. Update the `main` function to use `add_freq` and add the second print statement

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



```
#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

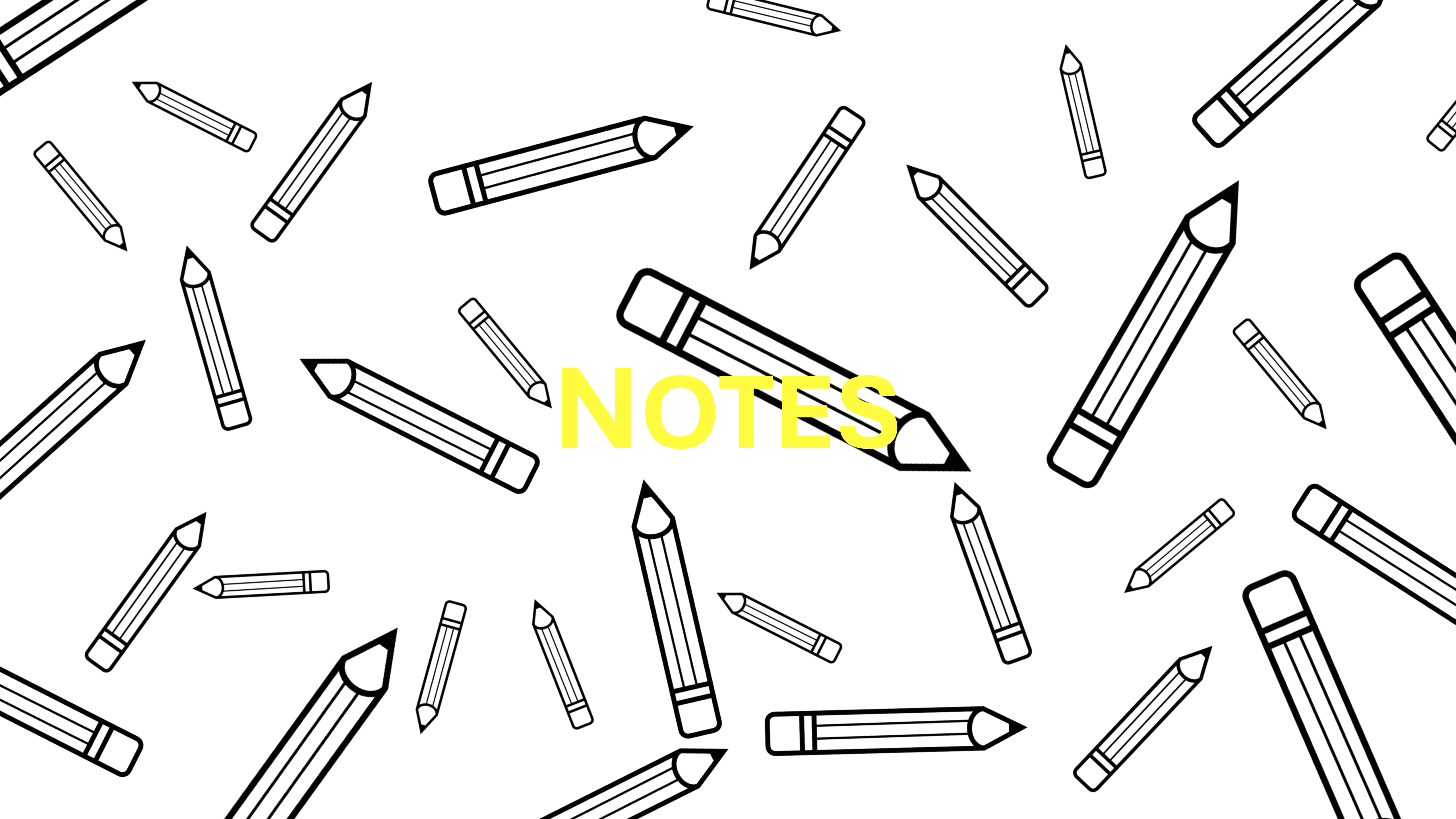
Exercise 6: *Free all the memory used by the set of strings*. Note, each string must be freed individually, and then S itself must be freed as well.

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)

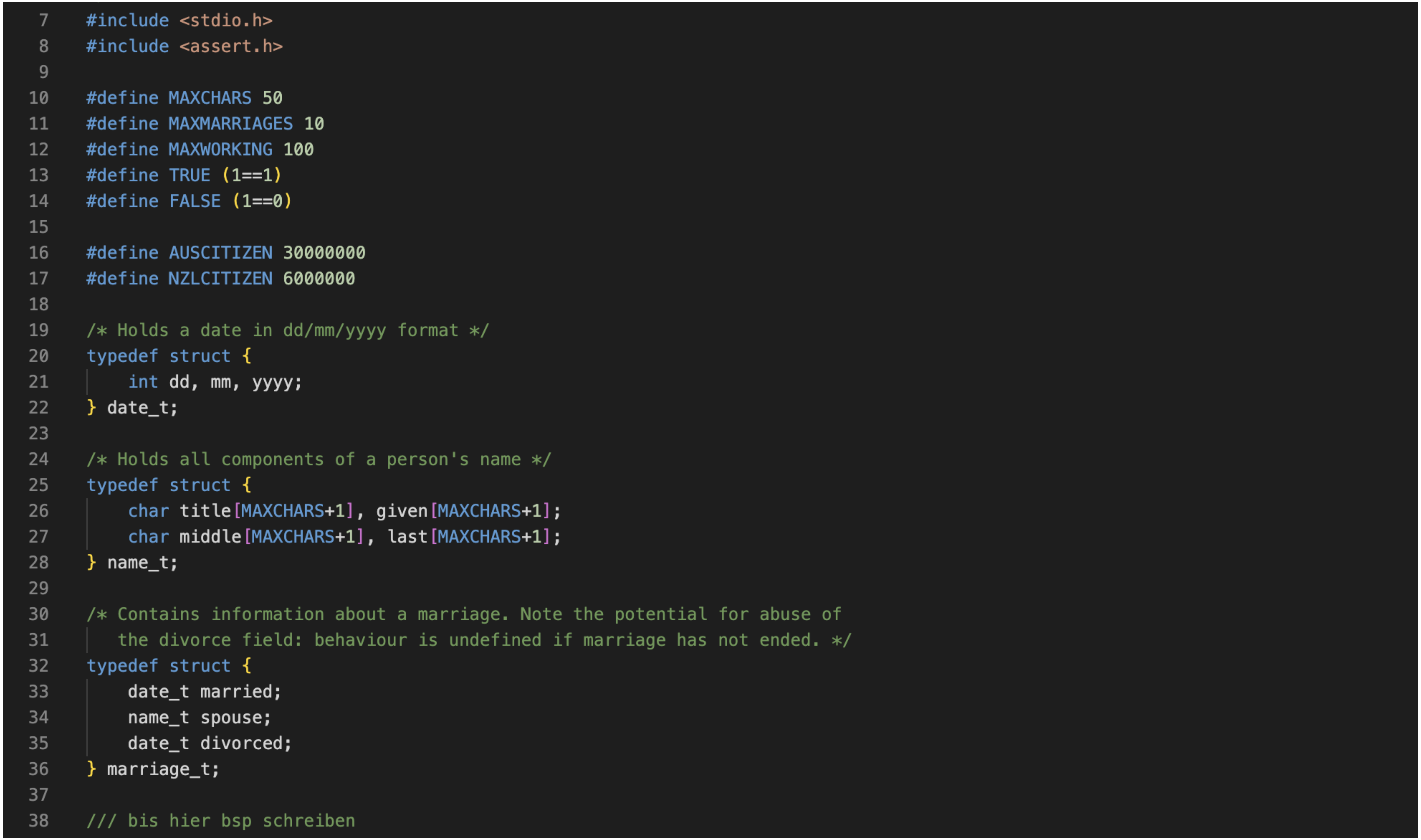
NOTES



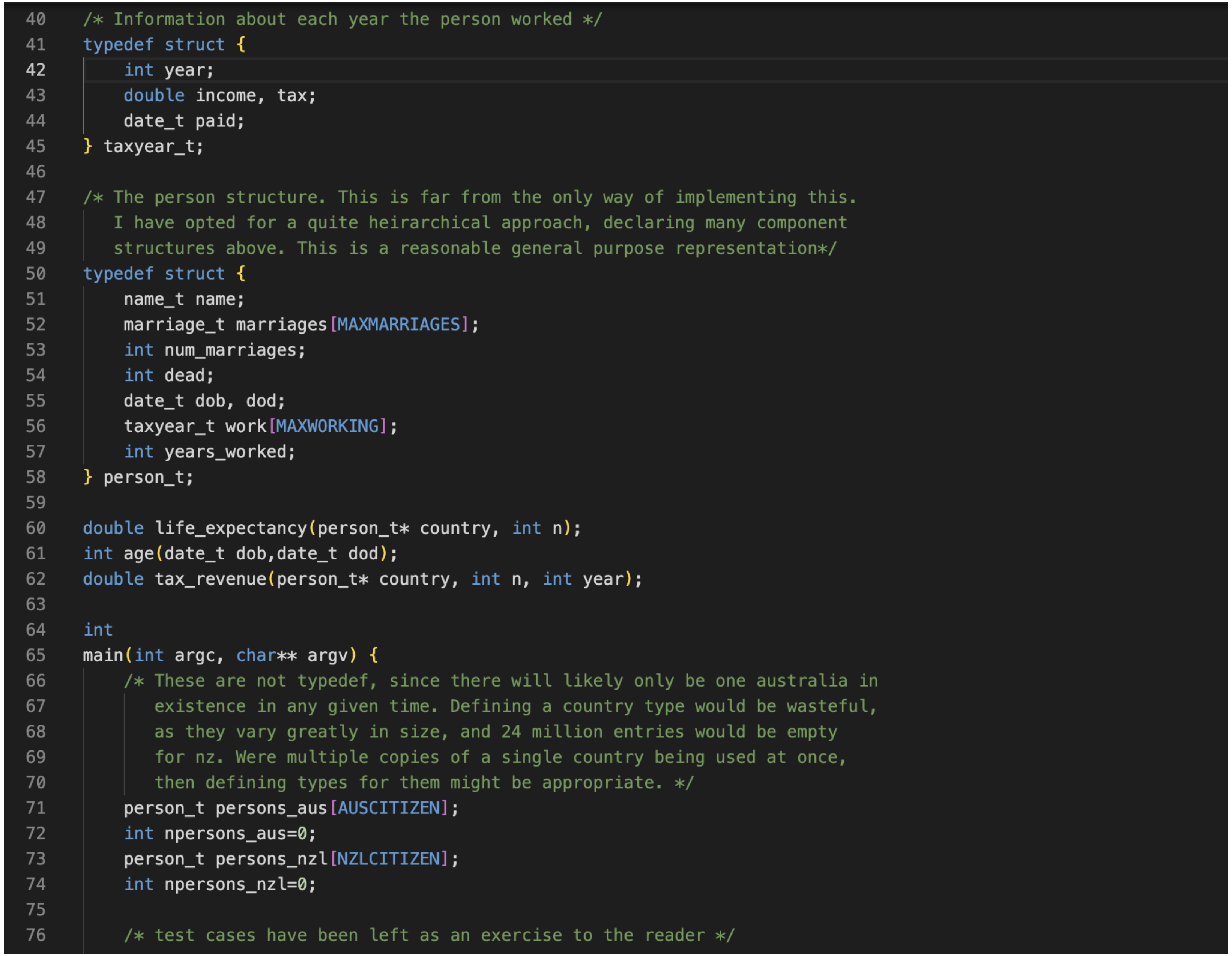
Lec07 | Exercise 1-3

People have titles, a given name, a middle name, and a family name, all of up to 50 characters each. People also have dates of birth (dd/mm/yyyy), dates of marriage and divorce (as many as 10 of each), and dates of death (with a flag to indicate whether or not they are dead yet). Each date of marriage is accompanied by the name of a person. Assuming that people work for less than 100 years each, people also have, for each year they worked, a year (yyyy), a net income and a tax liability (both rounded to whole dollars), and a date when that tax liability was paid. **Countries** are collections of people. Australia is expected to contain as many as 30,000,000 people; New Zealand as many as 6,000,000 people.

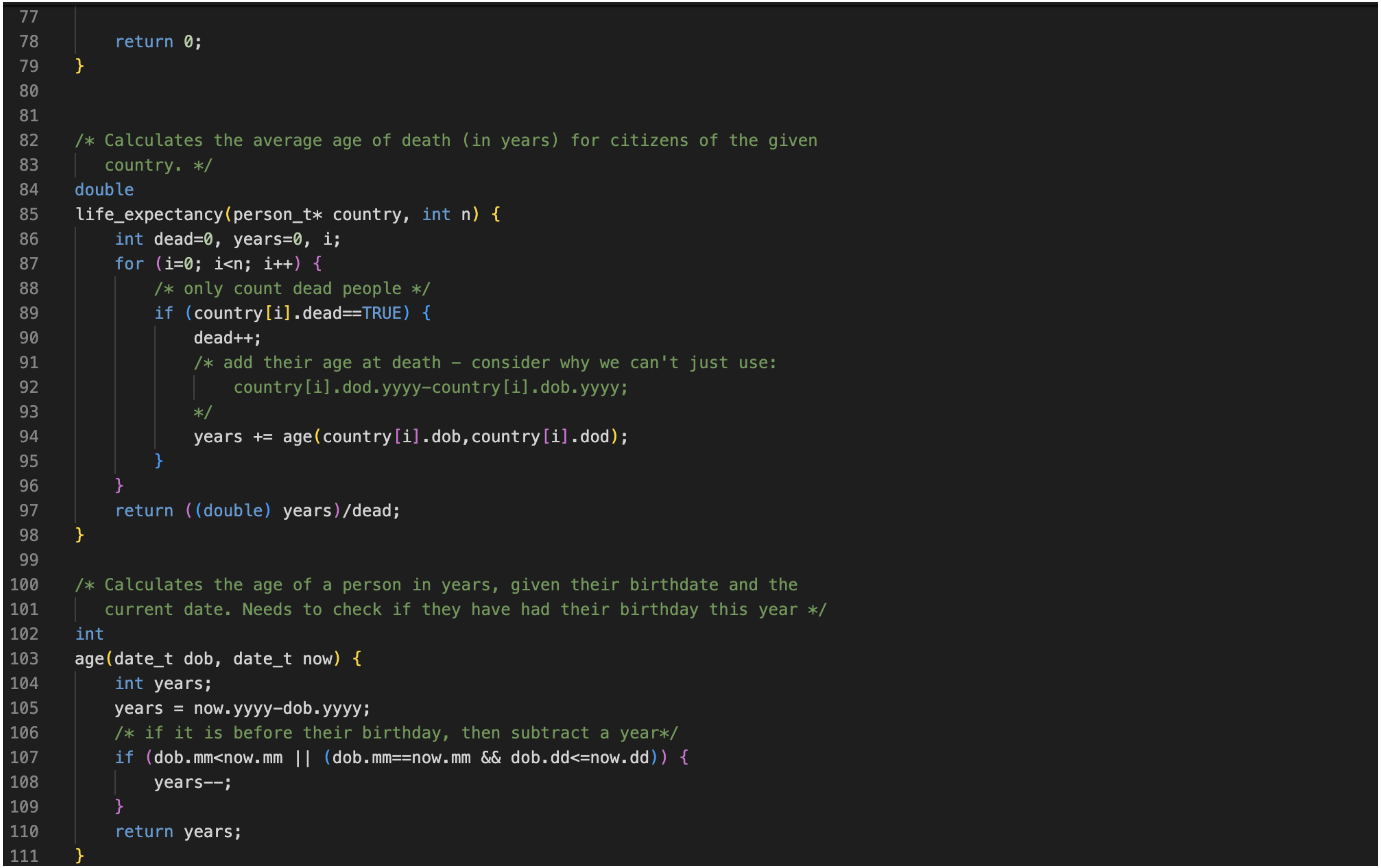
1. Give declarations that reflect the data scenario that is described.
2. Write a function that calculates, for a specified country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2), the average age of death. Do not include people that are not yet dead.
3. Write a function that calculates, for the country indicated by a pointer argument (argument 1) with a number of persons in it (argument 2) the total taxation revenue in a specified year (argument 3).



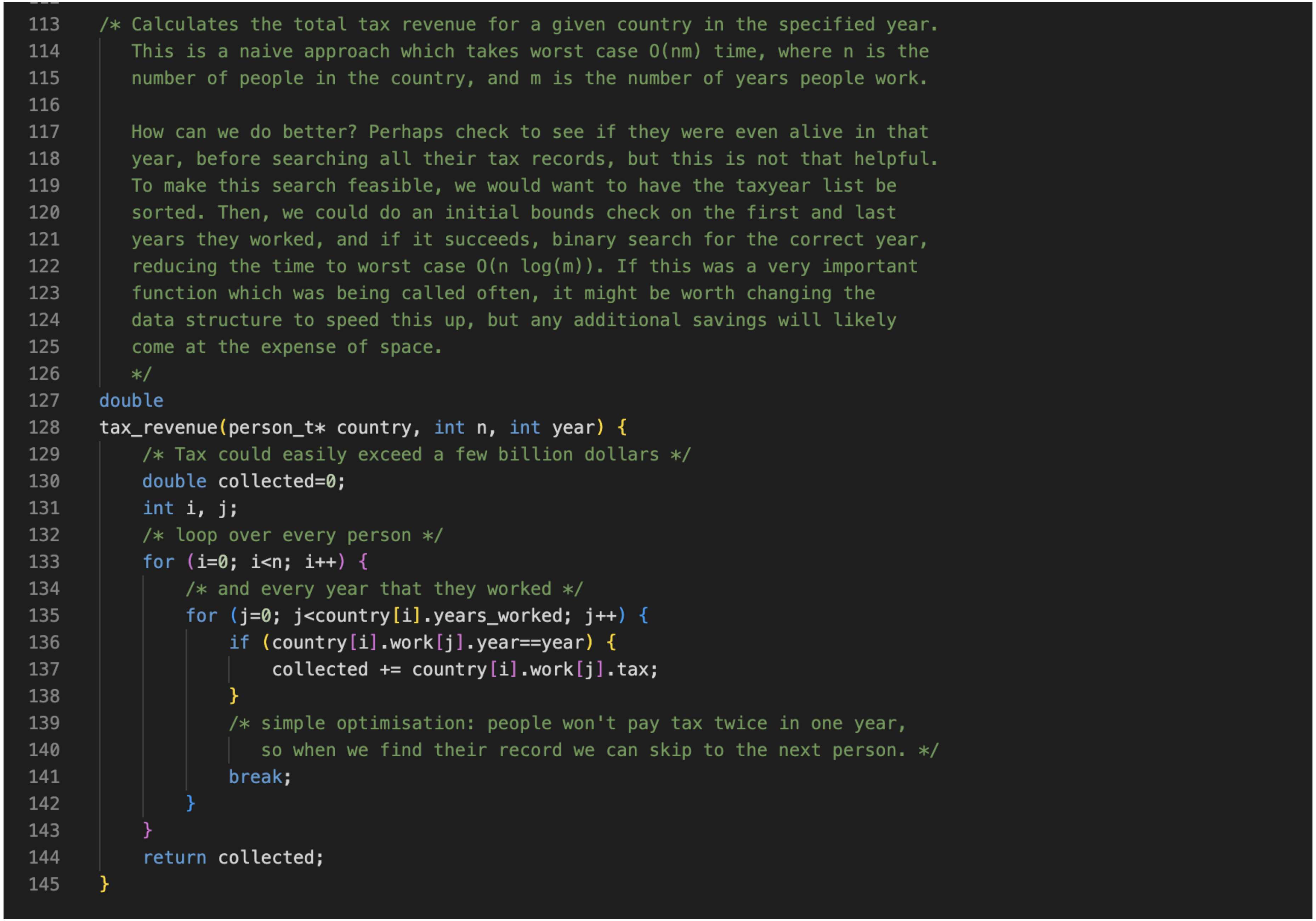
```c
#include <stdio.h>
#include <assert.h>

#define MAXCHARS 50
#define MAXMARRIAGES 10
#define MAXWORKING 100
#define TRUE (1==1)
#define FALSE (1==0)

#define AUSCITIZEN 30000000
#define NZLCITIZEN 6000000

/* Holds a date in dd/mm/yyyy format */
typedef struct {
    int dd, mm, yyyy;
} date_t;

/* Holds all components of a person's name */
typedef struct {
    char title[MAXCHARS+1], given[MAXCHARS+1];
    char middle[MAXCHARS+1], last[MAXCHARS+1];
} name_t;

/* Contains information about a marriage. Note the potential for abuse of
   the divorce field: behaviour is undefined if marriage has not ended. */
typedef struct {
    date_t married;
    name_t spouse;
    date_t divorced;
} marriage_t;

/// bis hier bsp schreiben
```



```c
/* Information about each year the person worked */
typedef struct {
    int year;
    double income, tax;
    date_t paid;
} taxyear_t;

/* The person structure. This is far from the only way of implementing this.
   I have opted for a quite heirarchical approach, declaring many component
   structures above. This is a reasonable general purpose representation*/
typedef struct {
    name_t name;
    marriage_t marriages[MAXMARRIAGES];
    int num_marriages;
    int dead;
    date_t dob, dod;
    taxyear_t work[MAXWORKING];
    int years_worked;
} person_t;

double life_expectancy(person_t* country, int n);
int age(date_t dob,date_t dod);
double tax_revenue(person_t* country, int n, int year);

int
main(int argc, char** argv) {
    /* These are not typedef, since there will likely only be one australia in
       existence in any given time. Defining a country type would be wasteful,
       as they vary greatly in size, and 24 million entries would be empty
       for nz. Were multiple copies of a single country being used at once,
       then defining types for them might be appropriate. */
    person_t persons_aus[AUSCITIZEN];
    int npersons_aus=0;
    person_t persons_nzl[NZLCITIZEN];
    int npersons_nzl=0;

    /* test cases have been left as an exercise to the reader */
```



```c

    return 0;
}


/* Calculates the average age of death (in years) for citizens of the given
   country. */
double
life_expectancy(person_t* country, int n) {
    int dead=0, years=0, i;
    for (i=0; i<n; i++) {
        /* only count dead people */
        if (country[i].dead==TRUE) {
            dead++;
            /* add their age at death - consider why we can't just use:
                country[i].dod.yyyy-country[i].dob.yyyy;
            */
            years += age(country[i].dob,country[i].dod);
        }
    }
    return ((double) years)/dead;
}

/* Calculates the age of a person in years, given their birthdate and the
   current date. Needs to check if they have had their birthday this year */
int
age(date_t dob, date_t now) {
    int years;
    years = now.yyyy-dob.yyyy;
    /* if it is before their birthday, then subtract a year*/
    if (dob.mm<now.mm || (dob.mm==now.mm && dob.dd<=now.dd)) {
        years--;
    }
    return years;
}
```



```c
/* Calculates the total tax revenue for a given country in the specified year.
   This is a naive approach which takes worst case O(nm) time, where n is the
   number of people in the country, and m is the number of years people work.

   How can we do better? Perhaps check to see if they were even alive in that
   year, before searching all their tax records, but this is not that helpful.
   To make this search feasible, we would want to have the taxyear list be
   sorted. Then, we could do an initial bounds check on the first and last
   years they worked, and if it succeeds, binary search for the correct year,
   reducing the time to worst case O(n log(m)). If this was a very important
   function which was being called often, it might be worth changing the
   data structure to speed this up, but any additional savings will likely
   come at the expense of space.
   */
double
tax_revenue(person_t* country, int n, int year) {
    /* Tax could easily exceed a few billion dollars */
    double collected=0;
    int i, j;
    /* loop over every person */
    for (i=0; i<n; i++) {
        /* and every year that they worked */
        for (j=0; j<country[i].years_worked; j++) {
            if (country[i].work[j].year==year) {
                collected += country[i].work[j].tax;
            }
            /* simple optimisation: people won't pay tax twice in one year,
               so when we find their record we can skip to the next person. */
            break;
        }
    }
    return collected;
}
```