CSC209H Worksheet: structs

1. Here is the beginning of a program involving structs. You will need to fill in missing bits. If you can work with a partner with a machine and actually compile your program at each step, do that. If not, work on paper.

```
#define MAX_POSITION_SIZE 16
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct player {
    char *name;
    char position[MAX_POSITION_SIZE];
    int home_runs;
    float avg;
};

int main() {
    // Declare a struct player called p1.

    struct player p1;j

    // Initialize it to represent the first baseman Justin Smoak,
    // whose batting average is 0.242. He has hit 25 home runs.

    p1.name = "Justin Smoak";

    // or if we wish the name to be mutable we could do this instead
    // p1.name = malloc(strlen("Justin Smoak") + 1);
    // strcpy(p1.name, "Justin Smoak"):

    strcpy(p1.position, "first base");
    p1.home_runs = 25;
    p1.avg = 0.242;
```

2. Here we have added a declaration for a pointer to a struct player. Allocate space for the struct on the heap and have p2 point to that memory.

   Error-checking: Look at the man page for `malloc` to see what it returns if it is unable to allocate memory. Now add code to check if the memory allocation for `p2` was successful, and if it failed, exit the program with a non-zero exit status.

```
    struct player *p2;

    if ((p2 = malloc(sizeof (struct player))) == NULL) {
        perror("malloc");
        exit(1);
    }
    // Set the values of p2 to represent centre fielder Kevin Pillar. His batting average is .260
    // and he has hit 15 home runs.
    p2->name = "Kevin Pillar";
    strcpy(p2->position, "centre field");
    p2->avg= 0.260;
    (*p2).home_runs = 15; // this notation is equivalent
```

CSC209H Worksheet: structs

3. Write a function `out_of_the_park` that increments the home-run count for the player passed as the function's argument. Think carefully about what the type of the function parameter should be.

```
void out_of_the_park (struct player *p) {
    p->home_runs++;
}
```

4. Show how to make calls to `out_of_the_park` using `p1` and `p2`.

```
out_ot_the_park(&p1);
out_of_the_park(p2);
```

5. Suppose we have the following function declaration.

```
void f(struct player p) { // Body hidden }
```

   Now suppose we call it from `main` using `f(p1)`. Draw the memory diagram of the program immediately after `f` is called, but before it starts executing. For extra practice, include `p2` and related memory in your diagram.

6. Something to think carefully about: can the body of `f` affect the local `p1` of `main`? In other words, after `f(p1)` exits, can any data associated with `p1` have changed?

   SOLUTION:
   Since local variable `p` is a copy of `p1`, the fields of the struct `p1` can not be changed by `f`. That means that `position, home_runs` and `avg` can not be changed. The field `name` cannot be changed either but since `name` is a pointer, the value to which it points **can** be changed. Of course, if `name` points to an immutable string literal it can't be changed. If we set the name to point to memory on the heap (as shown as an alternate for Justin Smoak), then function `f` could change this name.

7. On a new sheet of paper, repeat the previous two questions when you call `f(*p2)` instead.