Tutorial 6: C continue

Question 1: Pointer

1. Here are two ways we could store a collection of words in C.

```
char hello[]   = "Hello";
char cat[]     = "Cat";
char world[]   = "World";
char* dict[] = {hello, cat, world};
```

What do each of the following values describe? If possible, also give the value itself. If the value is undefined, explain why.

(a) hello

Solution. A memory address for the where string "Hello" lives in memory. The actual value depends on where the compiler decides to put it.

(b) *hello

Solution. The value of the byte that the pointer hello points at. In this case, it's the first character of the string, or the character 'H', with value 72.

(c) &hello

Solution. A memory address for where the pointer hello itself resides in memory. The actual value depends on where the compiler decides to put it.

(d) hello+1

Solution. The memory address of the second character in the string "Hello", namely, wherethe 'e' lives.

(e) dict

Solution. The memory address of where the array of three char pointers lives. The actual value depends on where the compiler decides to put it.

(f) *dict

Solution. The value of the first item in the array dict[], which is the pointer hello. The value of hello itself depends on where the compiler decided to store the string "Hello".

(g) **dict

Solution. What the first element of dict[] points at. The first element of dict[] is hello, which is pointing at the character 'H'.

(h) dict[2][3]

Solution. Equivalent to *(*(dict+2)+3). Now, *(dict+2) resolves as the pointer world, and *(world + 3) is the character 'l'.

(i) world[10]

Solution. Whatever random data happens to be in possibly unallocated memory 5 bytes pastthe last character 'd' in the string "World".

(j) cat[0][0]

Solution. cat[0] resolves as 'C', or the value 67. So (depending on the compiler) this could either try to fetch whatever is in the 67th byte of memory, or more likely, will not compile in the first place as the compiler won't let you use a char as a pointer.

2. If you had a pointer to a char array, (like hello in the above example), and we passed this pointer to a function to compute the length of the array, how would the function known when to stop?
   Solution. All strings in C actually use up one more byte of data than the string itself, as there is a special character at the end, called a **null terminator** (with value 0) that marks the end of the string. Note the null terminator character (usually denoted '\0' is not to be confused with the character for zero, namely '0', which actually has value 30 (check an ASCII lookup table).
3. What about if we have an array of integers?
   ```
   int arrayptr[] = {4,2,7,2,9,8};
   ```
   Given arrayptr, is it possible to compute how many numbers in the array that arrayptr points to?
   Solution. Arrays don't have a terminator, as there'd be no way to distinguish the number zero in the array from a null terminator, as they have the same value. You'd have to record the size of the array somewhere, and pass that information along.
4. Consider the following two functions.
   ```
   int addone(int x)
   {
       return x + 1;
   }

   void addone2(int* x)
   {
       *x = *x + 1;
   }
   ```
   We call the first style of function *pass-by-value* and the second *pass-by-reference*.[1]
   What do each of these functions do? Identify the differences.
   Solution.
   The addone function takes an integer as input, and returns a new integer that is one bigger. The original value still exists, unless specifically overwritten by the return of the function, e.g.,
   ```
   int x = 10;
   x = addone(x);
   ```
   The addone2 function modifies the original integer in place, by being passes a pointer to the integer,and overwriting the original value with the one plus the original value.
   ```
   int x = 10;addone2(&x);
   // now x=11
   ```
5. Suppose you wanted to write a function to swap the value of two integers.
   ```
   int x = 10;
   int y = 20;
   // call the swap function on x and y
   // now x = 20, y = 10
   ```

---

[1] I'm lying here, pass by reference requires the ability to be able to modify the input to a function. C can't actually dothis, but merely *emulates* pass-by-reference by passing a non-mutable pointer to a value, which can then be modified bywriting data back to the memory address specified by the pointer.

Write such a function. Should you write it as a pass-by-value, or as a pass-by-reference[2]?

Solution.

It should be pass-by-reference, as we need to modify the original values. We emulate this by writinga function take takes two pointers and swaps the dereferenced values.

```
void swap(int* x, int* y)
{
      int tmp = *x;
      *x = *y;
      *y = tmp;
}
```

Question 2: Pointer and array

Write a program that creates an array of type float. How you populate this array is up to you. Print out the content of the array and the corresponding memory addresses for each entry. (you may want to try with both equivalent ways of referencing an array to make sure your syntax is up to speed).

Take note of the numeric difference between successive address entries. What does that suggest regarding the way the array is stored to memory? What is the size of a float entry?

```c
#include <stdio.h>

int main() {
   float arr[] = {1.22, 2.33, 3.44, 4.55};
   int size = sizeof(arr)/sizeof(float);

   for (int i = 0; i < size; i++) {
      printf("Entry: %f Memory address %p\n", arr[i], &arr[i]);
   }

   printf("\n");

   for (int i = 0; i < size; i++) {
      printf("Entry: %f Memory address %p\n", *(arr+i), arr+i);
   }

   return 0;
}
```

Question 3: Structure

Implement, using structures and functions as appropriate, a program which requires you to enter multiple points in 3 dimensions. The points will have a name (one alphanumeric character) and three coordinates x, y, and z. Find and implement a suitable way to enter the points. The program, through an appropriate function, should calculate the centre of gravity of the point cloud as a new point (i.e., the average of each coordinate for all points entered)

The output should show a list of the points entered and the centre of gravity as a new point. (i.e., name, x, y, and z)

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

struct point
{
    char name[10];
    float x;
    float y;
    float z;
};

struct point find_centre(struct point *point_arr, int num);

int main() {
    struct point point_arr[3];
    int size = sizeof(point_arr)/sizeof(struct point);

    for (int i = 0; i < size; i++) {
        printf("Please enter the point name, x, y and z in sequence: \n");
        scanf("%s", point_arr[i].name);
        scanf("%f", &point_arr[i].x);
        scanf("%f", &point_arr[i].y);
        scanf("%f", &point_arr[i].z);
    }

    struct point centre = find_centre(point_arr, size);

    for (int i = 0; i < size; i++) {
            printf("%s\t%f\t%f\t%f\n",point_arr[i].name,point_arr[i].x,point_arr[i].y,point_arr[i].z);
    }
    printf("%s\t%f\t%f\t%f\n",centre.name,centre.x,centre.y,centre.z);

    return 0;
}

struct point find_centre(struct point *point_arr, int num) {
    struct point centre;
    float mean_x;
    float mean_y;
    float mean_z;

    for (int i = 0; i < num; i++) {
        mean_x += (point_arr+i)->x;
        mean_y += (point_arr+i)->y;
        mean_z += (point_arr+i)->z;
    }

    strcpy(centre.name, "centre");
    centre.x = mean_x/num;
    centre.y = mean_y/num;
    centre.z = mean_z/num;

    return centre;
}
```