L9 - Pointers and Structures

November 1, 2019

0.1 Pointers and Structures

The & operator can be applied to structures:

```
In [ ]: point_t* ptr;

        point_t point1 = {320, 200};

        ptr = &point1;
```

The variable `ptr` now contains the address of the first byte in memory allocated to `point1`.
`*ptr` is the entire structure. `(*ptr).x` and `(*ptr).x` are the members of the structure pointed to. We need parentheses; the dot operator has higher precedence than the content-of operator.
If `ptr` is a pointer to a structure, then `ptr -> member` is a shorthand for `(*ptr).member`.

But why? Don't pass large structures as function arguments: * Forces pass-by-*value* semantics * Copying an entire structure requires time and memory
Instead, pass *pointers to structures* as function arguments.

0.1.1 addpoints

This function changes the structure pointed to by the parameter `ptr1`:

```
In [6]: #include <stdio.h>
        #include <stdlib.h>

        typedef struct{
            int x;
            int y;
        } point_t;

        void addpoints(point_t *ptr1, const point_t *ptr2) // const avoids changing points we
        {
            ptr1->x = ptr1->x + ptr2->x;
            ptr1->y = ptr1->y + ptr2->y;
        }

        int main()
```

```
{
    point_t a = {2, 3};
    point_t b = {4, 5};

    addpoints(&a, &b);

    printf("point a is now [%d, %d]", a.x, a.y);

    return EXIT_SUCCESS;
}
```

```
point a is now [6, 8]
```

Alternately, we can avoid modifying `a` if we give another struct, `sum`, to put the result in:

```
In [9]: #include <stdio.h>
        #include <stdlib.h>

        typedef struct{
            int x;
            int y;
        } point_t;

        void addpoints(const point_t* ptr1, const point_t* ptr2, point_t* sum) // const avoids
        {
            sum->x = ptr1->x + ptr2->x;
            sum->y = ptr1->y + ptr2->y;
        }

        int main()
        {
            point_t a = {2, 3};
            point_t b = {4, 5};
            point_t out; // output

            addpoints(&a, &b, &out); // so pointy

            printf("sum is [%d, %d]", out.x, out.y);

            return EXIT_SUCCESS;
        }
```

```
sum is [6, 8]
```

Returning a Pointer Rewrite `addpoints()` so that it returns a pointer to a `point_t` structure containing the sum of the two points.

```
In [14]: #include <stdio.h>
         #include <stdlib.h>
```

```c
typedef struct{
    int x;
    int y;
} point_t;

point_t* addpoints(const point_t* ptr1, const point_t* ptr2) // const avoids changing
{
    point_t sum;

    sum.x = ptr1->x + ptr2->x;
    sum.y = ptr1->y + ptr2->y;

    return &sum; // return the address of the sum
}

int main()
{
    point_t a = {2, 3};
    point_t b = {4, 5};

    point_t* result = addpoints(&a, &b); // so pointy

    printf("sum is [%d, %d]", (*result).x, (*result).y);

    return EXIT_SUCCESS;
}
```

```
/tmp/tmp9vfd7zu4.c: In function addpoints:
/tmp/tmp9vfd7zu4.c:16:12: warning: function returns address of local variable [-Wreturn-local-a
     return &sum; // return the address of the sum
            ^~~~
[C kernel] Executable exited with code -11
```

This fails to compile because `sum` is a local variable and will be *deallocated* as soon as `addpoints()` exits. This leaves a pointer to a structure that *does not exist!*.

The correct way to do this is to allocate the structure on the **heap**, rather than on the stack.