CSO: Recitation 3

Goktug Saatcioglu

09.12.2019

1 Memory Layout

We begin by talking about how different types in C are placed in memory. As we know from lecture, structs and arrays are contigiously laid out in the memory. Furthermore, if we only declare a variable then its contents are placed on the stack. On the other hand, we can also use pointers to point to memory located on the stack or the heap. Let's consider the two different ways we can utilize nested struct behavior and what the memory implication of each are.

The first program is given below.

```c
#include <stdio.h>

struct Point_2D {
  int x;
  int y;
};

struct Point_3D {
  struct Point_2D xy;
  int z;
};

int norm_squared(struct Point_3D point) {
  struct Point_2D xy_coordinates = point.xy;
  return xy_coordinates.x * xy_coordinates.x + xy_coordinates.y * xy_coordinates.y +
    point.z * point.z;
}

int main() {
  struct Point_2D point_a = {3, 4};
  struct Point_3D point_b = {point_a, 5};

  int b_norm_squared = norm_squared(point_b);

  printf("Norm squared is : %d\n", b_norm_squared);

  return 0;
}
```

Questions:

- How is `point_a` represented in memory? It is in the stack and two integers are laid out contigiously.
- How about `point_b`? Well `point_b` has two fields that must be laid out contigiously which then has a field of type `Point_2D` which is also laid out contigiosuly. Overall, we will have 3 fields which are all integers.
- When `point_b` is intialized what does the field `xy` get? Well it expects a full struct and not an address of a struct (i.e. pointer) so it must get a copy of `point_a`.

- Do the two different print statements differ? No because we created a copy of `point_a` when initializing `point_b`.

Now let's consider a new program.

```c
#include <stdio.h>

struct Point_2D {
  int x;
  int y;
};

struct Point_3D {
  struct Point_2D* xy;
  int z;
};

int norm_squared(struct Point_3D point) {
  struct Point_2D xy_coordinates = *point.xy;
  return xy_coordinates.x * xy_coordinates.x + xy_coordinates.y * xy_coordinates.y +
    point.z * point.z;
}

int main() {
  struct Point_2D point_a = {3, 4};
  struct Point_3D point_b = {&point_a, 5};

  int b_norm_squared = norm_squared(point_b);

  printf("Norm squared is : %d\n", b_norm_squared);

  point_a.x = 2;

  b_norm_squared = norm_squared(point_b);

  printf("Is this the same norm squared : %d?\n", b_norm_squared);

  return 0;
}
```

Questions:

- How is `point_a` represented in memory? It is in the stack and two integers are laid out contigiously. It is the same representation as above so nothing has changed.
- How about `point_b`? Now `point_b` has two fields that must be laid out contigiously where one of them is a pointer. So that is all that there is to this `point_b`.
- When `point_b` is intialized what does the field `xy` get? Now it expects an address to a struct so it gets the address of `point_a` and the field points to that struct.
- Do the two different print statements differ? Now they do because of the pointer.

We modify the above program to look like this now.

```c
#include <stdio.h>

struct Point_2D {
  int x;
  int y;
};

struct Point_3D {
  struct Point_2D* xy;
  int z;
};

int norm_squared(struct Point_3D point) {
  struct Point_2D point_a = {3, 4};
  point.xy = &point_a;
  struct Point_2D xy_coordinates = *point.xy;
  return xy_coordinates.x * xy_coordinates.x + xy_coordinates.y * xy_coordinates.y +
    point.z * point.z;
}

int main() {
  struct Point_3D point_b;
  point_b.z = 5;

  int b_norm_squared = norm_squared(point_b);

  printf("Norm squared is : %d\n", b_norm_squared);

  struct Point_2D tmp = *point_b.xy;

  printf("The (x,y) co-ordinates are : (%d, %d)", tmp.x, tmp.y);

  return 0;
}
```

Questions:

- In terms of memory layout has anything changed? No.
- What can go wrong with this program? When the stack frame of `norm_squared` is popped off after the function call we get a pointer that references invalid memory as the memory that `point_a` was located in is freed due to the popping of the stack frame.

Let's try to fix the problem by allocating the object on the heap instead of on the stack. The solution looks as follows.

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

struct Point_2D {
  int x;
  int y;
};
```

```c
struct Point_3D {
  struct Point_2D* xy;
  int z;
};

struct Point_2D* allocate_2d() {
  struct Point_2D* point_a = malloc(sizeof(struct Point_2D));

  if (point_a == NULL)
    return NULL;

  point_a->x = 3;
  point_a->y = 4;

  return point_a;
}

int norm_squared(struct Point_3D* point) {
  point->xy = allocate_2d();
  if (point->xy == NULL)
    return -1;

  struct Point_2D* xy_coordinates = point->xy;
  return xy_coordinates->x * xy_coordinates->x + xy_coordinates->y * xy_coordinates->y +
    point->z * point->z;
}

int main() {
  struct Point_3D point_b;
  point_b.z = 5;

  int b_norm_squared = norm_squared(&point_b);

  printf("Norm squared is : %d\n", b_norm_squared);

  struct Point_2D tmp = *point_b.xy;

  printf("The (x,y) co-ordinates are : (%d, %d)\n", tmp.x, tmp.y);

  return 0;
}
```

Now a lot has changed in our program. Firstly, we created a new function that allocates a `Point_2D` on the heap and if it is unable to then it ill return `NULL`. Next, in the `norm_squared` function we start by first allocating our inner struct and then making sure the allocation was successful. This is something you should always do. Then we calculate the norm by accessing the pointer itself. Upon return from the method we will now have the field `xy` on the heap and we are pointing to it so we haven't lost anything so our program no longer crashes. Finally, notice how we also had to change the argument `norm_squared` to a pointer. Why did we do this? Well if hadn't changed it to a pointer then we would have lost the reference to the `Point_2D` when the stack frame popped off. This is because C has call by value semantics in that case which means we create a copy of `point_b` before passing it in. By switching to reference semantics we can ensure that we modify the same struct rather than its copy.

Questions:

- What does the memory layout look like now? There is a struct on the heap now. Other than that not much should change.
- There is still something wrong with this program. Can you figure it out? We are doing what is known as polluting the heap. We forgot to de-allocate the struct we had allocated! We are also not checking whether the struct was actually allocated in the `main` method.

The final fixed program is given below.

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

struct Point_2D {
  int x;
  int y;
};

struct Point_3D {
  struct Point_2D* xy;
  int z;
};

struct Point_2D* allocate_2d() {
  struct Point_2D* point_a = malloc(sizeof(struct Point_2D));

  if (point_a == NULL)
    return NULL;

  point_a->x = 3;
  point_a->y = 4;

  return point_a;
}

int norm_squared(struct Point_3D* point) {
  point->xy = allocate_2d();
  if (point->xy == NULL)
    return -1;

  struct Point_2D* xy_coordinates = point->xy;
  return xy_coordinates->x * xy_coordinates->x + xy_coordinates->y * xy_coordinates->y +
    point->z * point->z;
}

int main() {
  struct Point_3D point_b;
  point_b.z = 5;

  int b_norm_squared = norm_squared(&point_b);

  if (b_norm_squared == -1)
    return 7;
```

```
  printf("Norm squared is : %d\n", b_norm_squared);

  struct Point_2D tmp = *point_b.xy;

  printf("The (x,y) co-ordinates are : (%d, %d)\n", tmp.x, tmp.y);

  free(point_b.xy);

  return 0;
}
```

Make sure you get a strong understanding of the difference between these programs!

2 Parsing C Types

To correctly read a C type declaration we start at the name and working outwards according to the rules of precedence. So `int *x[10]` is read as `x` is an array of pointers while `int (*x)[10]` is read as `x` is a pointer to an array of `int`s. The general idea is, read the name and work right-wards. Once reaching the very right work left-wards until you are done. If the name is enclosed in a bracket then first parse the bracket then the remaining type declaration.

3 Pointer Arithmetic

Suppose we have the following C code.

```
int a[100];
int *ptr;
```

Then the two statements `ptr = a;` and `ptr = &a[0];` are equivalent. That is, we can view arrays as either a collection of contigious memory or a pointer to some starting address which we can then increment to get the other elements of the array. If the array starts at address 400 then the above two statements would assign 400 to the `ptr` variable. Following this, the two statements `ptr = a + 1;` and `ptr = &a[1];` are equivalent and both will assign 404 to `ptr`. Then, we can sum up the elements of an array by either doing

```
int sum = 0;
for (ptr = a; ptr < &a[100]; ++ptr)
  sum += *ptr;
```

or

```
int sum = 0;
for (i = 0; i < 100; ++i)
  sum += *(a + i);
```

and the array access notation `a[b];` we are used to is basically synctactic sugar for `*(a+b)`.