

C Structs for REAL Dummies

You are probably experienced with writing Java classes, which probably look something like:

```
class Player {
    int id;
    int hp;
    boolean onGround;
    String username;
    double x, y, z;
}
```

While programming, we use “Player” like any other type: similar to how we can declare variables of type “int” or of type “Object”, we can declare variables of type “Player”. The Player type is a type that has the fields id, hp, username, etc. An equivalent structure in C might look like:

```
struct {
    int id;
    int hp;
    int onGround; // C has no boolean type, so we use `char` instead.
    char* username;
    double x, y, z;
}
```

There are a few things to notice here: first, the structure’s name is gone. This structure has no name, or as C programmers say, this is an anonymous struct. More on this later. Secondly, boolean and String have become int and char*, respectively.

In C, ‘true’ and ‘false’, don’t exist: instead, we say ‘non-zero’ or ‘zero’. In an if-statement, any expression that evaluates to zero is false, and everything else (nonzero) is true. Likewise, we don’t have ‘boolean’ and just substitute it with a number, typically 1 or 0.

The String has become a char*. Think about it: a String is just a sequence of characters, right? It makes sense to think about a String as just a char array, which as we learned is equivalent to a char*. The pointer just points to the first character of the string, and the rest of the characters follow it sequentially in memory. Another important convention that we follow in C is that strings are terminated with a zero byte. This lets us know where the string actually stops. We call strings in this form “C strings”.

54	68	69	73	20	69	73	20	61	20	43	20	73	74	72	69	6E	67	2E	00
T	h	i	s		i	s		a		C		s	t	r	i	n	g	.	

^ char* points here

Likewise, in C we can declare variables of this type:

```
struct {  
    int id;  
    int hp;  
    int onGround;  
    char* username;  
    double x, y, z;  
} myPlayer;
```

And it would be perfectly valid to write code such as `myPlayer.x = 5`; similar to Java. The key observation is that this compound “structure type” can be used like any other type we are familiar programming with, such as ints, doubles, pointers, etc. It’s just that this struct type is one that happens to have the fields `id`, `hp`, `username`, and so on.

Hello! My Name Is... (Typedefs)

But writing that explicit struct type declaration out each time is a chore. Let’s assign a name to this compound type so we can refer to it easily:

```
typedef struct {  
    int id;  
    int hp;  
    int onGround;  
    char* username;  
    double x, y, z;  
} Player_t;
```

Now, our previous declaration of `myPlayer` becomes:

```
Player_t myPlayer;
```

We’ve assigned a name, “`Player_t`” to our previously-anonymous type. To do that, we used the **typedef** keyword. Typedef simply creates an alias for a type; in this case, it aliased “`Player_t`” to that explicit structure type. In other words, typedef defines types. Here’s another example of typedef:

```
typedef bool_t int;  
#define true 1  
#define false 0  
bool_t myBool = false;
```

This is actually a way you can “invent” your own boolean type, since C doesn’t provide one for you. The meaning of `#define` is not important right now, but it simply creates compile-time macros. For instance, here ‘`true`’ is a macro for 1, and ‘`false`’ is a macro for 0.

You also may be wondering why we have added a “`_t`” at the end of the type’s name. It’s a common naming convention among C programmers to add “`_t`” to indicate that a type is a typedef.

Memory Layout

Anyways, let's get back to the main topic. Where are these values actually stored in memory? As it so happens, basically every C compiler stores them sequentially starting from the base address of the struct. For example, our `Player_t` struct looks like this in memory:

address	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57	89	A	B	C	D	E	F	0	1	2	3	4	5	6	7
013AA148	7B	00	00	00	64	00	00	00	01	00	00	00	90	7B	3A	01	{...d...	{:..												
013AA158	00	00	00	00	00	80	4A	40	00	00	00	00	00	44	8F	40	J@	D	@										
013AA168	6F	12	83	C0	CA	21	09	40	00	00	00	00	00	00	00	00	o.	!.@												

Erm...this is a bit hard to read.

address	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53	54	55	56	57
013AA148	7B	00	00	00	64	00	00	00	01	00	00	00	90	7B	3A	01
013AA158	00	00	00	00	00	80	4A	40	00	00	00	00	00	44	8F	40
013AA168	6F	12	83	C0	CA	21	09	40	00	00	00	00	00	00	00	00

`id = 0x7b = 123` `hp = 100` `onGround = 1 (true)`

`username = 0x013a7b90 -> "you expected..."`

This is a bit better. Though... wouldn't it be nice if there was a tool to help dissect the structure for us? Luckily, there's ReClass for that:

```

▼ 013AA148 (1) Class Player_t [40] //
  0000 013AA148 Int32 id = 123 //
  0004 013AA14C Int32 hp = 100 //
  0008 013AA150 Int32 onGround = 1 //
  000C 013AA154 PCHAR username = 'you expected george p. burdell, but it was me, Dio!'
  0010 013AA158 double x = 53 //
  0018 013AA160 double y = 1001 //
  0020 013AA168 double z = 3.142 //
  
```

The important thing to take away here, is that if we understand the layout of a structure, we can find the address of all of its member data fields if we know the base address of the structure. And vice versa: if we know the address of one of the fields, and we know its offset, we can calculate the base address.

base address + offset = field address

field address - offset = base address

Or in C:

```

&myPlayer + 4 = &myPlayer.hp
&myPlayer.hp - 4 = &myPlayer
  
```

What's My Size?

In C, every structure has a size. The size of a structure is just the sum of all of its members' sizes. Let's calculate the size of our `Player_t` structure:

- `int id` (4 bytes)
- `int hp` (4 bytes)
- `int onGround` (4 bytes)
- `char* username` (4 bytes on 32-bits)
- `double x` (8 bytes)
- `double y` (8 bytes)
- `double z` (8 bytes)

$4+4+4+4+8+8+8 = 40$ bytes.

Now, it's useful to be able to know how large a structure is. Imagine we had a function that copied some amount of memory from a source to a destination. Since the function copies memory, we'll call this function "memcpy". Now suppose we wanted to copy a struct. It would be useful to know how large our struct is, so we know how much memory to copy!

Luckily, there is the **sizeof** operator. `sizeof` simply evaluates to (at compile-time) how many bytes a struct is. So, for example: `printf("%d\n", sizeof(Player_t));` would print "40". And if we were to use `memcpy`, we would call it like so: `memcpy(&myPlayerCopy, &myPlayer, sizeof(Player_t));` Note the arguments: destination pointer, source pointer, and size.

It's Turtles All the Way Down

The **real** fun begins when we start putting structs inside other structs. Remember the `x`, `y`, `z` fields from our `Player_t` structure? Well, let's suppose our game wants to use a unified format across the whole engine to store positions, to make things simple when calculating stuff. It would make sense to wrap those three fields into a struct!

```
typedef struct {  
    double x, y, z;  
} Vec3_t;
```

```
typedef struct {  
    int id;  
    int hp;  
    int onGround;  
    char* username;  
    Vec3_t pos;  
} Player_t;
```

The memory layout is the same as before, but importantly, as programmers we can now treat "pos" as a single piece of data, since now it's just a single variable of type `Vec3_t`:

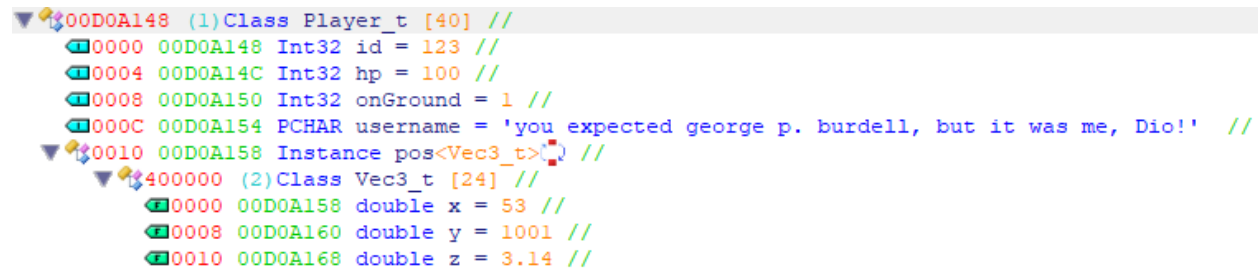
```
myPlayer.pos.z = 3.14;
```

The size of our structure is still the same:

- int id (4 bytes)
- int hp (4 bytes)
- int onGround (4 bytes)
- char* username (4 bytes on 32-bits)
- Vec3_t pos (24 bytes)
 - o double x (8 bytes)
 - o double y (8 bytes)
 - o double z (8 bytes)

$4+4+4+4+24 = 40$ bytes. The math checks out! And of course, `sizeof(Vec3_t)` is 24.

Or in ReClass:



```
▼ 00D0A148 (1) Class Player_t [40] //
  0000 00D0A148 Int32 id = 123 //
  0004 00D0A14C Int32 hp = 100 //
  0008 00D0A150 Int32 onGround = 1 //
  000C 00D0A154 PCHAR username = 'you expected george p. burdell, but it was me, Dio!' //
  0010 00D0A158 Instance pos<Vec3_t> //
    ▼ 400000 (2) Class Vec3_t [24] //
      0000 00D0A158 double x = 53 //
      0008 00D0A160 double y = 1001 //
      0010 00D0A168 double z = 3.14 //
```

As you can imagine, things can get pretty crazy with struct nesting, but at the end of the day, it's always perfectly fine to interpret things as a flat structure with all the members flattened.

Lies (Padding)

Sometimes, the compiler will pack a struct differently than you declare in your code. For example, if you declare a single 'char', which is one byte, the compiler may put it there, alongside 3 padding bytes, making it occupy 4 bytes in total. This is done for performance reasons, because aligned memory accesses are faster than unaligned one. It also may place padding at the end of your structure to make it an aligned size. Thankfully, `sizeof` will take into account padding properly.

It's not really important for you to know much about this right now, apart from the fact that the compiler might not layout structures exactly as you specify.