Section 1 / Alignment Within Structs

Overview

First, it is important to note that this section applies equally to both classes and structs. Ignoring methods (which are just functions connected to classes and structs), there is only a small difference between them.

In a C++ `class`, members default to `private`.

In a C++ `struct`, members default to `public`.

Classes, of course, do not exist in C.

Hereafter, we will use `class` and `struct` interchangeable.

In order to access data members of a `struct` you must be able to locate them relative to the start of the `struct`. If an instance of a `struct` begins at some address X, the first data member is also located at X so its relative offset from X is 0.

In our discussion of alignment, we'll frequently refer to the notion of *offset*.

Simple Rule

Data members exhibit natural alignment.

That is:

- a `long` will be found at addresses which are a multiple of 8.
- an `int` will be found at addresses which are a multiple of 4.
- a `short` will be found at addresses which are even.
- a `char` can be found anywhere.

Impact of the Simple Rule

Let's assume assume an `int` data member is placed properly at address some address, let's say 104. This is OK because 104 is a multiple of 4, the length of an int.

Suppose a `long` comes next. Does it start at location 108 or 112?

The answer is 112 even though this leaves a 4 byte gap between the end of the `int` and the beginning of the `long`. This is because the natural alignment of a `long` is upon addresses that are multiples of 8, the length of a `long`.

Sometimes, there are holes or gaps in a `struct`.

Higher level languages like C and C++ know this and produce the right code. Assembly language programmers have no obligation to stick to no stinkin' rules.

Figure 1: badges

If they don't mind writing code that's buggy, that is.

Example 1

```
struct {
    long a;
    short b;
    int c;
};
```

Wrong:

Offset	Width	Member
0	8	a
8	2	b
10	4	c

Correct:

Offset	Width	Member
0	8	a
8	2	b
10	2	– gap –
12	4	c

Demonstration:

Given this:

```
struct Foo {
    long a;
    short b;
    int c;
};

struct Foo Bar = { 0xaaaaaaaaaaaaaaaa, 0xbbbb, 0xcccccccc };
```

A hex dump will show:

```
aaaa aaaa aaaa aaaa bbbb 0000 cccc cccc
```

Notice the gap filled in which zeros. Note, if this were a local variable, the zeros might be garbage.

Example 2

Given this:

```
struct Foo {
    short a;
    char b;
    int c;
};

struct Foo Bar = { 0xaaaa, 0xbb, 0xcccccccc };
```

A hex dump will show:

```
aaaa 00bb cccc cccc
```

Notice there is only one byte of gap before the `int c` starts.

But, but, but - why are the zeros to the left of the b's?

This ARM processor is running as a *little endian* machine.

Diversion: Little Endian

Little endian means that within each unit of 2 (above a word), the **least** significant bytes come first.

In a little endian machine:

Type	Logical Contents	Actual Contents
`long`	`aabbccddeeff0011`	`0011 eeff ccdd aabb`

This shows that a `long` is 8 bytes:

```
aabbccddeeff0011
```

Transpose the two 4 byte groups:

`eeff0011 aabbccdd`

Transpose the 2 byte groups:

`0011 eeff ccdd aabb`

Type	Logical Contents	Actual Contents
`int`	`44556677`	`6677 4455`

This shows that an `int` is 4 bytes:

`44556677`

Transpose the two 2 byte groups:

`6677 4455`

The discussion on little endian is important if you are looking directly at the contents of memory, like when you are using `gdb`.