Chapter 45: Structure Padding and Packing

By default, C compilers lay out structures so that each member can be accessed fast, without incurring penalties for 'unaligned access, a problem with RISC machines such as the DEC Alpha, and some ARM CPUs.

Depending on the CPU architecture and the compiler, a structure may occupy more space in memory than the sum of the sizes of its component members. The compiler can add padding between members or at the end of the structure, but not at the beginning.

Packing overrides the default padding.

Section 45.1: Packing structures

By default structures are padded in C. If you want to avoid this behaviour, you have to explicitly request it. Under GCC it's `__attribute__((__packed__))`. Consider this example on a 64-bit machine:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

The structure will be automatically padded to have`8-byte` alignment and will look like this:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

So `sizeof(struct foo)` will give us `24` instead of `17`. This happened because of a 64 bit compiler read/write from/to Memory in 8 bytes of word in each step and obvious when try to write `char c;` a one byte in memory a complete 8 bytes (i.e. word) fetched and consumes only first byte of it and its seven successive of bytes remains empty and not accessible for any read and write operation for structure padding.

Structure packing

But if you add the attribute `packed`, the compiler will not add padding:

```
struct __attribute__((__packed__)) foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

Now `sizeof(struct foo)` will return `17`.

Generally packed structures are used:

- To save space.

- To format a data structure to transmit over network without depending on each architecture alignment of each node of the network.

It must be taken in consideration that some processors such as the ARM Cortex-M0 do not allow unaligned memory access; in such cases, structure packing can lead to *undefined behaviour* and can crash the CPU.

Section 45.2: Structure padding

Suppose this `struct` is defined and compiled with a 32 bit compiler:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

We might expect this `struct` to occupy only 10 bytes of memory, but by printing `sizeof(str_32)` we see it uses 12 bytes.

This happened because the compiler aligns variables for fast access. A common pattern is that when the base type occupies N bytes (where N is a power of 2 such as 1, 2, 4, 8, 16 — and seldom any bigger), the variable should be aligned on an N-byte boundary (a multiple of N bytes).

For the structure shown with `sizeof(int) == 4` and `sizeof(short) == 2`, a common layout is:

- `int a;` stored at offset 0; size 4.
- `short b;` stored at offset 4; size 2.
- unnamed padding at offset 6; size 2.
- `int c;` stored at offset 8; size 4.

Thus `struct test_32` occupies 12 bytes of memory. In this example, there is no trailing padding.

The compiler will ensure that any `struct test_32` variables are stored starting on a 4-byte boundary, so that the members within the structure will be properly aligned for fast access. Memory allocation functions such as `malloc()`, `calloc()` and `realloc()` are required to ensure that the pointer returned is sufficiently well aligned for use with any data type, so dynamically allocated structures will be properly aligned too.

You can end up with odd situations such as on a 64-bit Intel x86_64 processor (e.g. Intel Core i7 — a Mac running macOS Sierra or Mac OS X), where when compiling in 32-bit mode, the compilers place `double` aligned on a 4-byte boundary; but, on the same hardware, when compiling in 64-bit mode, the compilers place `double` aligned on an 8-byte boundary.