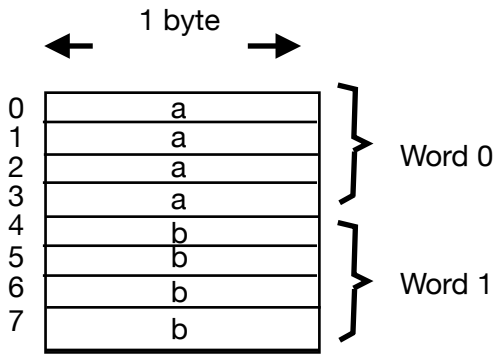
Memory alignment support in C

Radhika Grover
UCSC Silicon Valley Extension

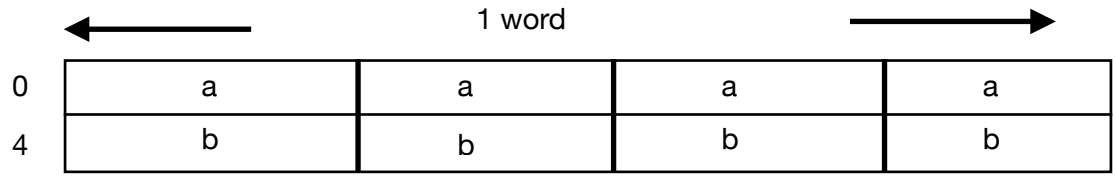
What is memory alignment?

Memory alignment is placing data in memory at a location that is a multiple of a *word* address. For example, assume that 1 word is 32 bits (4 bytes). Assume that two words ("aaaa", "bbbb") are stored in memory. Then each word takes 4 bytes as shown below.



Physical view

The logical view of memory assumes that a word or its multiple is fetched from memory at a time. So the word address should be a multiple of the word size.

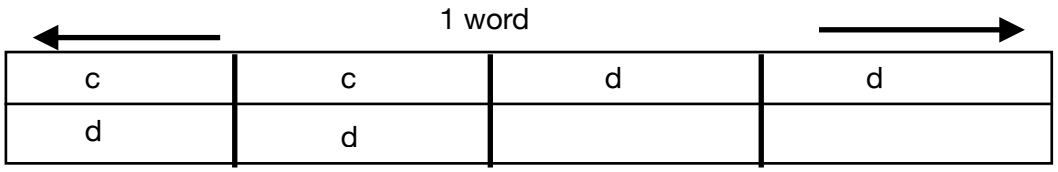


Logical view

The *word* size is different on different processor architectures (32 bits for x86, 32 bit for ARMv6, 64 bit for ARMv8 etc.). In general, the C compiler takes care of placing variables so that they are aligned correctly. See the following reference for more information on word sizes in different processor architectures:

https://en.wikipedia.org/wiki/Word_(computer_architecture)

Suppose we have two data values, “cc” with a size of 2 bytes and “dddd” with a size of 4 bytes, and they are placed as shown below:

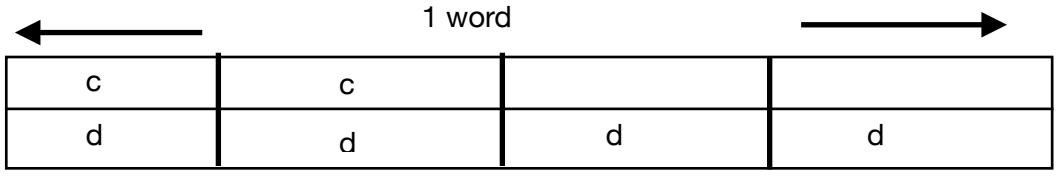


Logical view

The value “dddd” is not memory aligned, so reading this value will take two accesses, at 0 and 4. If the processor on which the program is running does not support this type of memory access and so it can lead to an exception and program failure. If the data is not memory aligned, it could be split across cache lines leading to more cache misses and slower performance. Similarly on a system with virtual memory, a page fault can occur leading to poorer performance:

https://en.wikipedia.org/wiki/Data_structure_alignment

The compiler or the programmer will specify padding to align the data values correctly. For example, the values “cc” and “dddd” can be stored with 2 bytes of padding inserted after “cc”:



Logical view

In general, the compiler adds padding automatically and so the programmer does not have to worry about this, but there may be special requirements for embedded systems. The C standard has added support via the compiler alignment attribute and the aligned_alloc function for dynamic memory allocation.

Structures

Consider the following C program with a structure named *Data*. Suppose that the program is run on a processor with a word size of 8 bytes.

```
typedef struct {
	uint8_t id1;
	uint8_t id2;
	uint64_t id3;
	uint64_t id4;

} Data;

int main(void) {
	Data data;
	printf("Unaligned addresses:\n");
	printf("The address of the struct data is %p \n", &data);
	printf("The address of the field id1 is %p \n", &data.id1);
	printf("The address of the field id2 is %p \n", &data.id2);
	printf("The address of the field id3 is %p \n", &data.id3);
	printf("The address of the field id4 is %p \n", &data.id4);
}
```

Running this program gives the following address:

```
The address of the struct data is 0x7ffeeeb1fe58
The address of the field id1 is 0x7ffeeeb1fe58
The address of the field id2 is 0x7ffeeeb1fe59
The address of the field id3 is 0x7ffeeeb1fe60
The address of the field id4 is 0x7ffeeeb1fe68
```

You can see that the 1-byte values id1 and id2 are placed in consecutive locations at hexadecimal addresses 0x..58 and 0x..59 but then a 6 byte padding is added and member id3 is at 0x..60. This padding is added automatically by the compiler.

Suppose that we would like to use 16 bytes for each member. Then the compiler alignment attribute can be used:

Compiler alignment attribute

This attribute will force the compiler to place align each variable on a 16-byte boundary:

```
typedef struct  {
      uint8_t id1 __attribute__ ((aligned (16)));
      uint8_t id2 __attribute__ ((aligned (16)));
      uint64_t id3 __attribute__ ((aligned (16)));
      uint64_t id4 __attribute__ ((aligned (16)));
} Data_align ;


int main(void) {
      Data_align data_align;
      printf("The address of the struct data_align is %p \n", &data_align);
      printf("The address of the field id1 is %p \n", &data_align.id1);
      printf("The address of the field id2 is %p \n", &data_align.id2);
      printf("The address of the field id3 is %p \n", &data_align.id3);
      printf("The address of the field id4 is %p \n", &data_align.id4);
}
```

When we run this program, it gives the following output on a processor with 8 byte word size:

```
The address of the struct data_align is 0x7ffedfe67e70
The address of the field id1 is 0x7ffedfe67e70
The address of the field id2 is 0x7ffedfe67e80
The address of the field id3 is 0x7ffedfe67e90
The address of the field id4 is 0x7ffedfe67ea0
```

You can see that each member takes 16 bytes, so the addresses are 0x..70, 0x..80, 0..x90 etc.

For more information on these attributes, see this reference:

https://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Type-Attributes.html

Dynamically allocating memory

C11 provides a new function for dynamic memory allocation that gives memory addresses that are aligned called **aligned_alloc**. More information on how to use aligned_alloc is available here:

https://en.cppreference.com/w/c/memory/aligned_alloc