Memory alignment

By: Yehia M. Abu Eita

Outlines

- Introduction
- Align data into memory
- Alignment and padding into structures
- Minimize padding in structures

Introduction

- **Alignment** means putting the data in memory at **address** equal to some **multiple of the word size**.
- **Word size** depends on the size of **data bus**.
- This **increases** the **performance** of system due to the way the CPU handles memory.
- On the other hand, **alignment may consumes** unused **memory**, **padding**.

Align data into memory

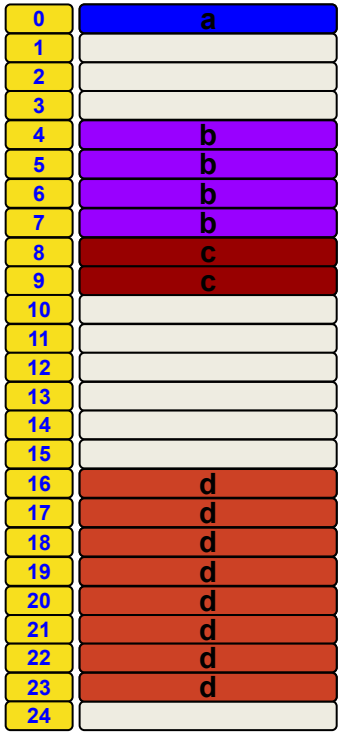
- **Byte data (char):**
  - Can be **stored** in **any** memory **location** **(0, 1, 2, 3, ...etc.)**.
- **Half-Word data (short):**
  - Must be "**half-word aligned**"
  - Must be **stored** in **even** number **addresses** **(0, 2, 4, ...etc.)**.
- **Word data (int):**
  - Must be "**word aligned**".
  - Must be **stored** in **addresses** that are **divisible by 4** **(0, 4, 8, 16, ...etc.)**.
- **Double-word data (long long):**
  - Must be "**double-word aligned**".
  - Must be **stored** in **addresses** that are **divisible by 8** **(0, 8, 16, 32, ...etc.)**.

Alignment and padding into structures

- Alignment appears clearly in structures.

```
typedef struct data
{
        char a;                 // 1 Byte
        int b;                  // 4 Bytes
        short int c;            // 2 Bytes
        long long int d;        // 8 Bytes
} data;
```

24 Bytes

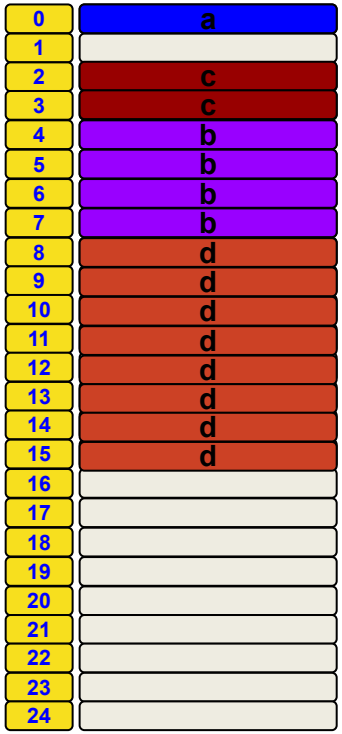


Minimize padding in structures

- Rearranging membery will reduce memory padding.
- Ascending or descending according to the type size.

```
typedef struct data
{
        char a;                 // 1 Byte
        short int c;            // 2 Bytes
        int b;                  // 4 Bytes
        long long int d;        // 8 Bytes
} data;
```

16 Bytes



Summary

- Now you are familiar with memory alignment and padding.
- Take care of structure’s creation, because it may cause unnecessary memory allocations.
- Remember, rearranging structure’s members will reduce padding within the memory.