# Aligned And Un-Aligned Data Access

# On

# Structures

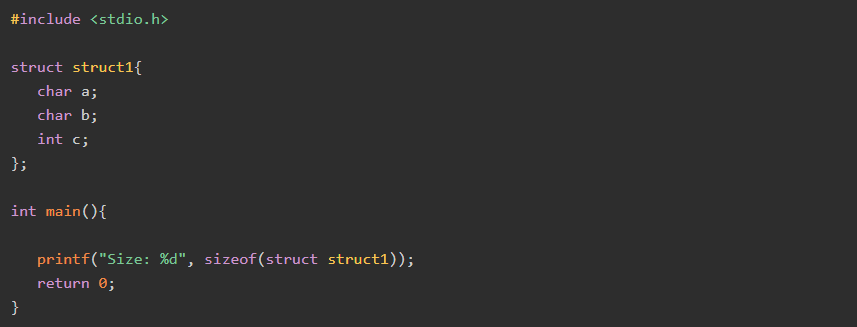
## Structure Padding in C

**Structure padding in C:**

* Managed by the CPU architecture.
* Adds extra bytes within a structure for natural alignment.
* Alignment requirements depend on the processor, not the C language.
* Varies based on data bus size and architectural specifics.

**Structure Padding with CPU Architecture:**

* A 32-bit CPU reads 4 bytes at a time (1 word = 4 bytes).
* Accesses are aligned to word boundaries, requiring padding for efficient reading.
* For example, to access an **int**, the CPU may need two cycles if the data isn’t aligned.
* Padding adds extra bytes before data to align it with word boundaries, optimizing access.







## Is sizeof for a struct equal to the sum of sizeof of each member?

* The sizeof for a struct is not always equal to the sum of sizeof of each individual member.

This is because of the padding added by the compiler to avoid alignment issues. Padding is only added when a structure member is followed by a member with a larger size or at the end of the structure.

// C program to illustrate

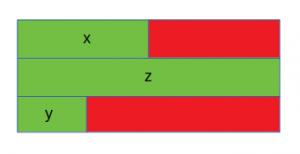
// size of struct

**#include** <stdio.h>

**int** **main**()

{

**struct** **A** {



// sizeof(int) = 4

**int** x;

// Padding of 4 bytes

// sizeof(double) = 8

**double** z;

// sizeof(short int) = 2

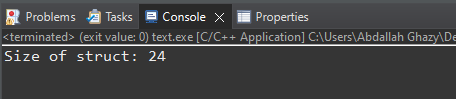
**short** **int** y;

// Padding of 6 bytes

};

**printf**("Size of struct: %ld", **sizeof**(**struct** **A**));

**return** 0;

}

The red portion represents the padding added for data alignment and the green portion represents the struct members.

In this case, x (int) is followed by z (double), which is larger in size as compared to x. Hence padding is added after x. Also, padding is needed at the end for data alignment.

// C program to illustrate

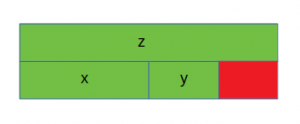
// size of struct

**#include** <stdio.h>

**int** **main**()

{

**struct** **B** {

 // sizeof(double) = 8

**double** z;

// sizeof(int) = 4

**int** x;

// sizeof(short int) = 2

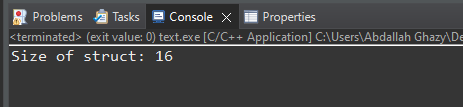
**short** **int** y;

// Padding of 2 bytes

};

**printf**("Size of struct: %ld", **sizeof**(**struct** **B**));

**return** 0;

}

// C program to illustrate

// size of struct

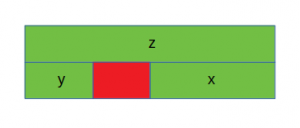
**#include** <stdio.h>

**int** **main**()

{

**struct** **C** {

// sizeof(double) = 8

 **double** z;

// sizeof(short int) = 2

**short** **int** y;

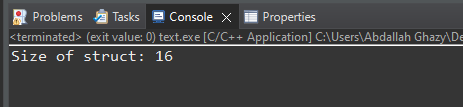
// Padding of 2 bytes

// sizeof(int) = 4

**int** x;

};

**printf**("Size of struct: %ld", **sizeof**(**struct** **C**));

 **return** 0;

}

# Alignment

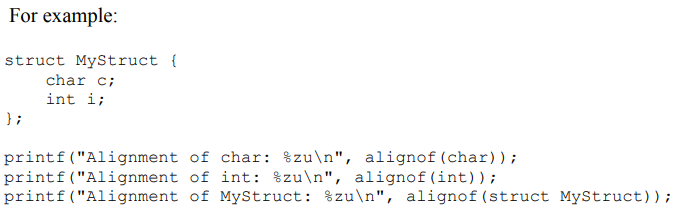
**Data Alignment in C:**

**Definition:** Refers to how data is stored in memory, often aligned to specific addresses.

**Impact:** Proper alignment improves performance and prevents crashes, especially with structures.

**Structures:** Contain members of different types, requiring proper alignment.

**Alignment Check:** Use the **alignof** operator to determine the required alignment for data types.



**Proper Data Alignment in C:**

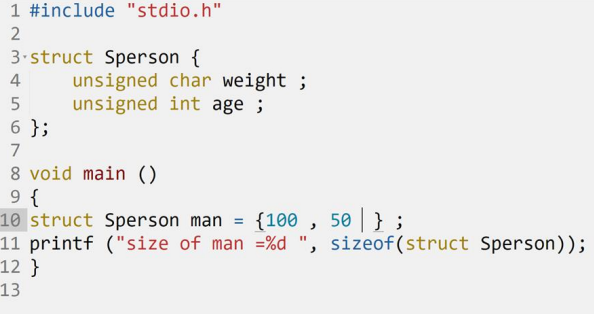
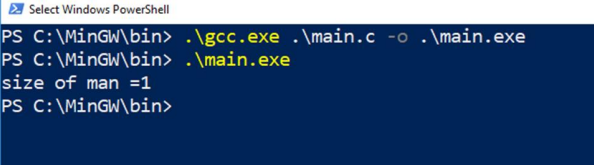
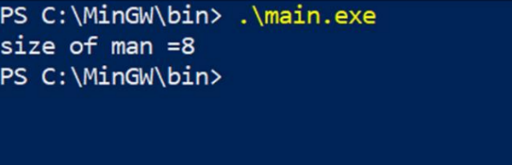
**Example:** Define a structure (MyStruct) with char and int members. Use the alignof operator to check alignment requirements.

**Compiler Behavior:** Most compilers align structures to the largest member for proper alignment.

**Custom Alignment:** Use compiler-specific directives/functions to control alignment, especially for hardware-specific data formats.

**Importance:** Proper alignment prevents performance issues and crashes on some architectures.

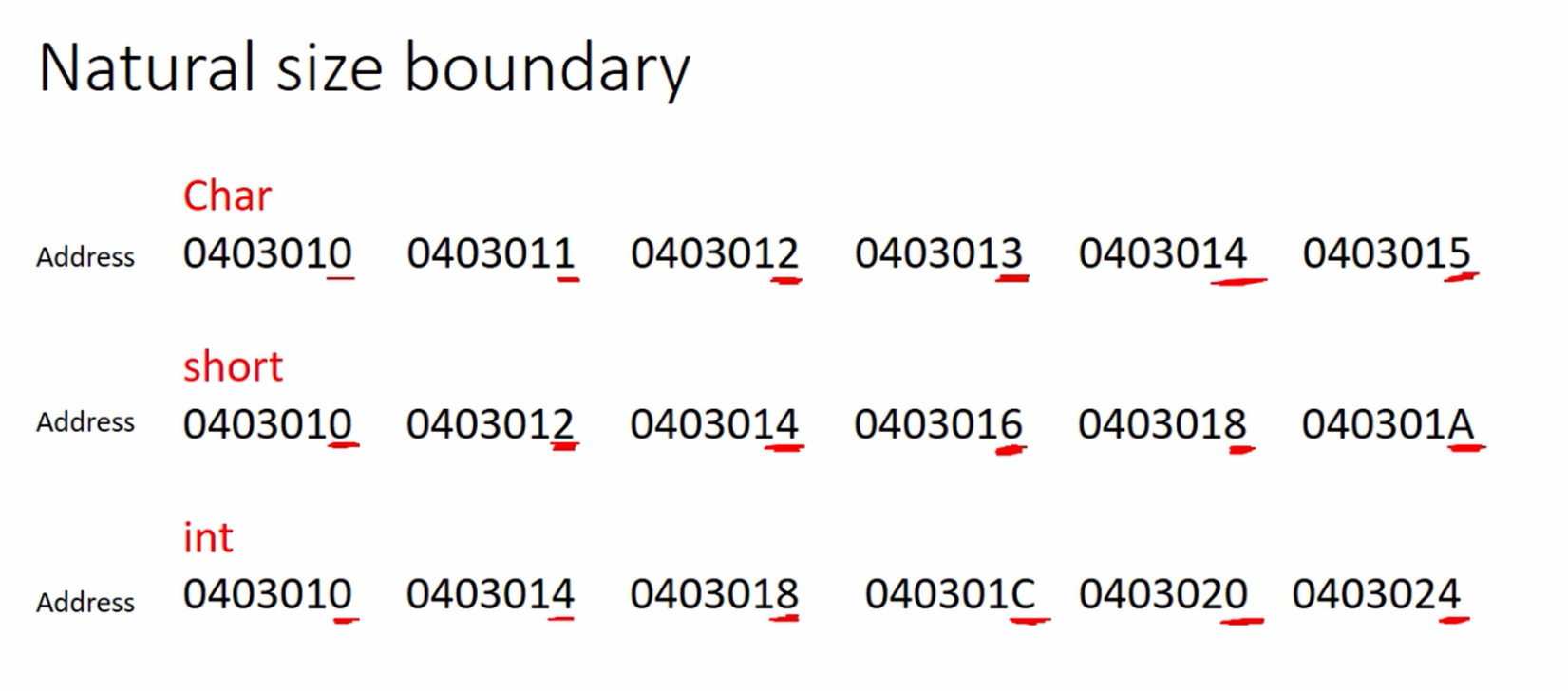
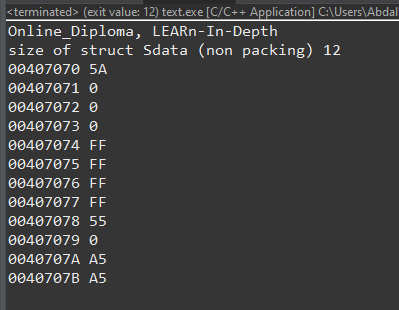
**Best Practice:** Ensure structures are properly aligned to optimize code performance and avoid potential issues.



**Aligned Data Storage:** When data is stored in aligned memory, it means that the memory address at which the data starts is a multiple of the data size.

For example, if you have a 4-byte integer, it should be stored at an address that is divisible evenly by 4.

Aligned data storage is generally more efficient because it allows the processor to fetch data in a single memory access, rather than requiring multiple accesses for unaligned data.



**#include** <stdio.h>

**struct** **Sdata** {

**unsigned** **char** datal;

**unsigned** **int** data2;

**unsigned** **char** data3;

**unsigned** **short** data4;

}gdata;

**void** **main**() {

**printf**("Online\_Diploma, LEARn-In-Depth\n");

gdata.datal = 0x5A5A;

gdata.data2 = 0xFFFFFFFF;

gdata.data3 = 0x55;

gdata.data4 = 0xA5A5;

**int** total\_size = **sizeof**(**struct** **Sdata**);

**printf**("size of struct Sdata (non packing) %d\n", **sizeof**(**struct** **Sdata**));

**dump\_memory**(&gdata, total\_size);

}

**void** **dump\_memory**(**char**\* ptr, **int** size) {

**int** i;

**for** (i = 0; i < size; i++) {

**printf**("%p %X\n", ptr, (**unsigned** **char**)\*ptr);

ptr++;

}

}

## Structure Packing in C

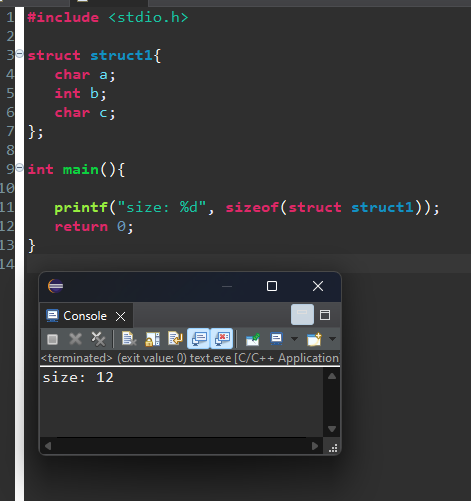
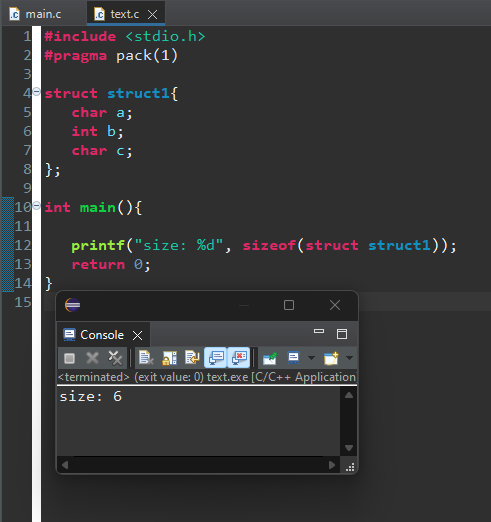
**Structure packing,** on the other hand, is a mechanism for minimizing the effect of padding, thereby trying and reduce wasted memory space. We can use certain pragma directives and attributes to achieve packing.

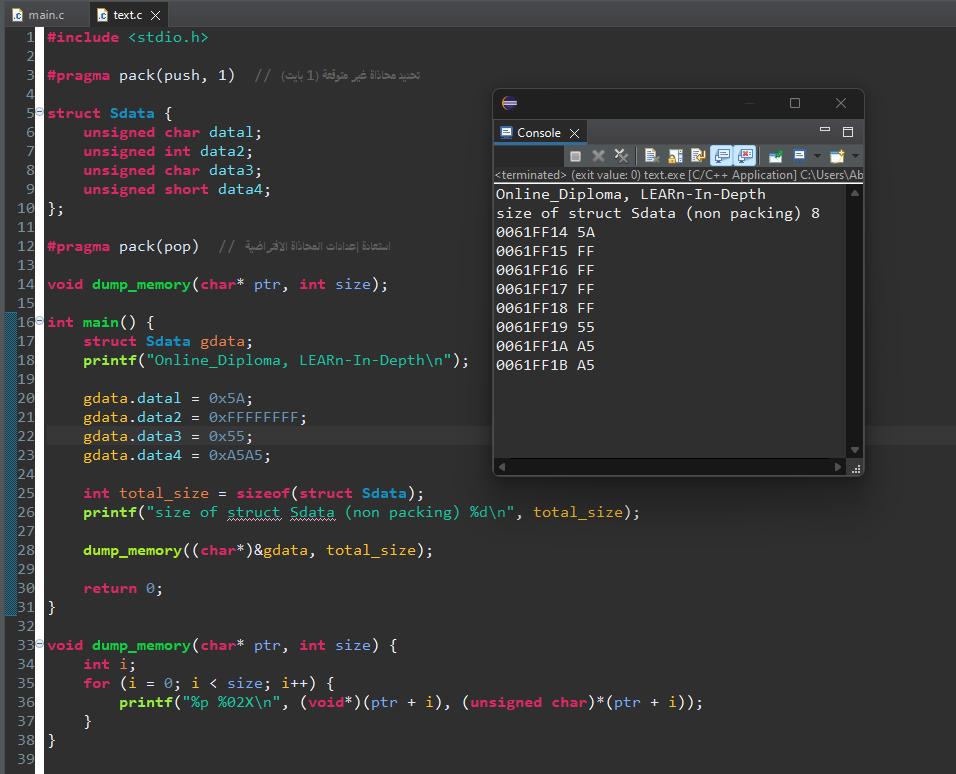
**Unaligned Data Storage:** Unaligned data storage means that the data is stored at memory addresses that do not match the data size.

Accessing unaligned data may require multiple memory accesses, potentially impacting performance. In some architectures, unaligned memory access can also cause alignment faults or exceptions.

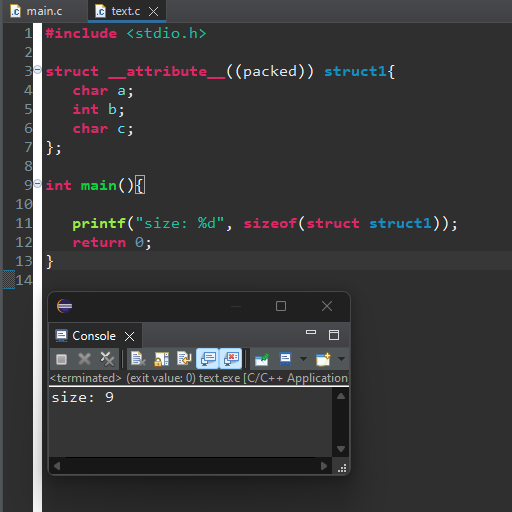
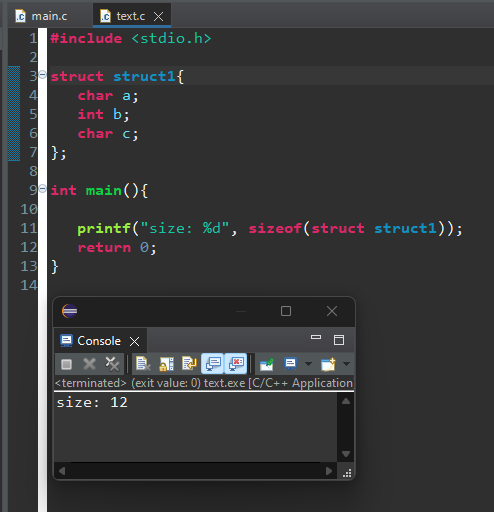
**Using #pragma pack(1) Directive**

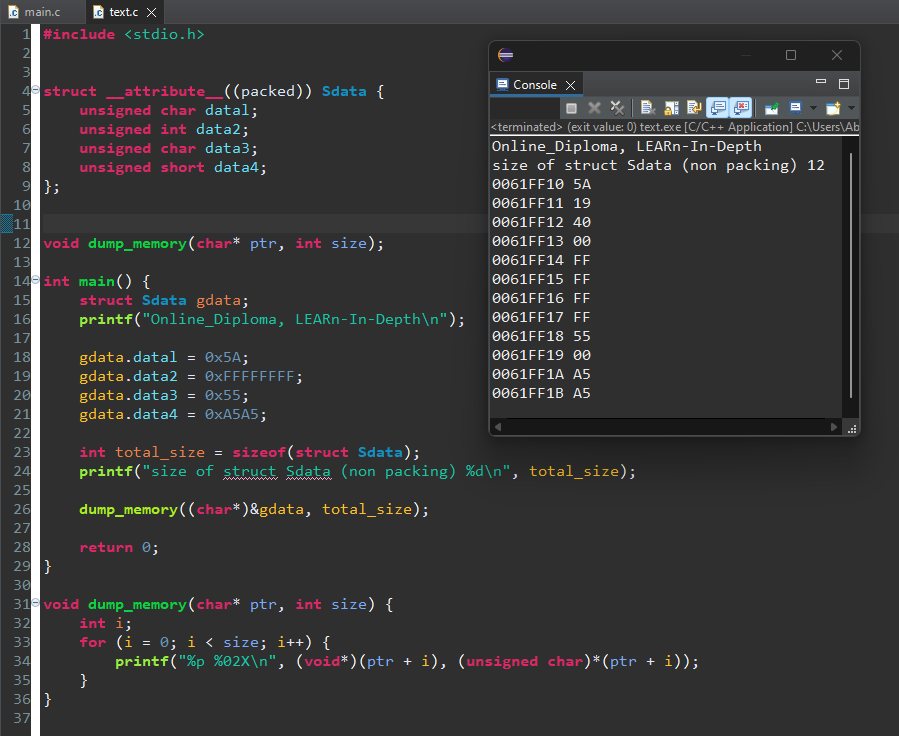
The #pragma pack(1) preprocessor directive forces the compiler to disregard the padding, and align the structure members end to end during the memory allocation process.





**Using \_\_attribute\_\_((packed))**



# Aligned and Unaligned Memory Access

**Unaligned** memory access is the access of data with a size of N number of bytes from an address that is not evenly divisible by the number of bytes N. We have aligned memory access if the address is evenly divisible by N.

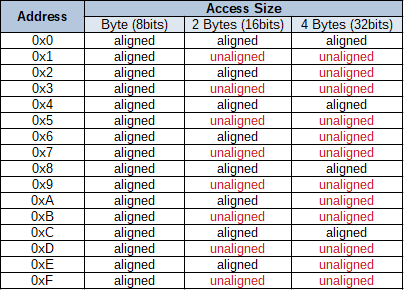
We can express this as **Address/N**, where **Address** is the memory address, and **N** is the number of bytes that are accessed. Here are some examples:

Two byte access from address 4: Address/N = 4/2 = 2 (aligned access)

Two byte access from address 3: Address/N = 3/2 = 1.5 (unaligned access)

Four byte access from address 24: Address/N = 24/4 = 6 (aligned access)

As a practical note, If the rightmost digit of the address (represented in a hexadecimal format) is divisible by the number of bytes, we have aligned memory access.



There are microprocessors that allow unaligned memory access and those that don’t. Unaligned access usually negatively impacts performance, as more operations (instructions) are required to perform it. If the microprocessor does not support unaligned access, an exception can be triggered (e.g., a bus error exception) when such access is attempted.

# Software Point of View

From the software's point of view, memory access is just instructions for reading or writing bytes of data to or from memory.

Structure Alignment:



**Memory Layout:**

* Starting address: 0x00001000
* data\_1 (2 bytes) occupies 0x00001000 and 0x00001001 (aligned).
* data\_2 (4 bytes) occupies 0x00001002 to 0x00001005.
* data\_3 (1 byte) occupies 0x00001006.

**Alignment Considerations:**

* data\_1 is aligned at 0x00001000 (multiple of 2 bytes).
* data\_2 starts at 0x00001002, which is not aligned to a 4-byte boundary (aligned would be 0x00001004).
* data\_3 is aligned at 0x00001006 (single-byte variables are always aligned).

**Compiler’s Role:**

* To align data\_2 properly, the compiler can insert padding bytes after data\_1. For instance, it might place 2 bytes of padding after data\_1 to align data\_2 at 0x00001004.

**With Padding:**

* data\_1 (2 bytes) at 0x00001000 to 0x00001001
* Padding (2 bytes) at 0x00001002 to 0x00001003
* data\_2 (4 bytes) at 0x00001004 to 0x00001007
* data\_3 (1 byte) at 0x00001008

### **Summary:**

* **Alignment in Structures:** Compilers align structure members to optimize memory access and may add padding to meet alignment requirements.
* **Unaligned Access Issues:** Casting pointers to different types and accessing unaligned memory can lead to undefined behavior and performance penalties.
* **Compiler Limitations:** The compiler does not handle runtime checks for pointer alignment, so developers must ensure proper alignment manually when dealing with unaligned memory access.

**Compiler Specifics**

The C programming language classifies unaligned memory access as undefined behavior.

The default behavior of the compiler when it comes to unaligned access is dependent on the target CPU architecture.

If the architecture does not allow unaligned accesses, then the compiler will place all variables, functions, etc., in an aligned manner.

If the CPU architecture allows unaligned access, then the compiler should have options where we can select whether it should take advantage of this or not. For example, **gcc** compiler has the following options for ARM processors that can be used: -munaligned-access -mno-unaligned-access.

[](https://github.com/Abdallah-Ghazy)

[](https://www.linkedin.com/in/abdallah-ghazy/)

[](https://www.facebook.com/profile.php?id=100009485341470)

[](https://www.youtube.com/channel/UCRh59pwh7KTLgfftifu_zrQ)

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"