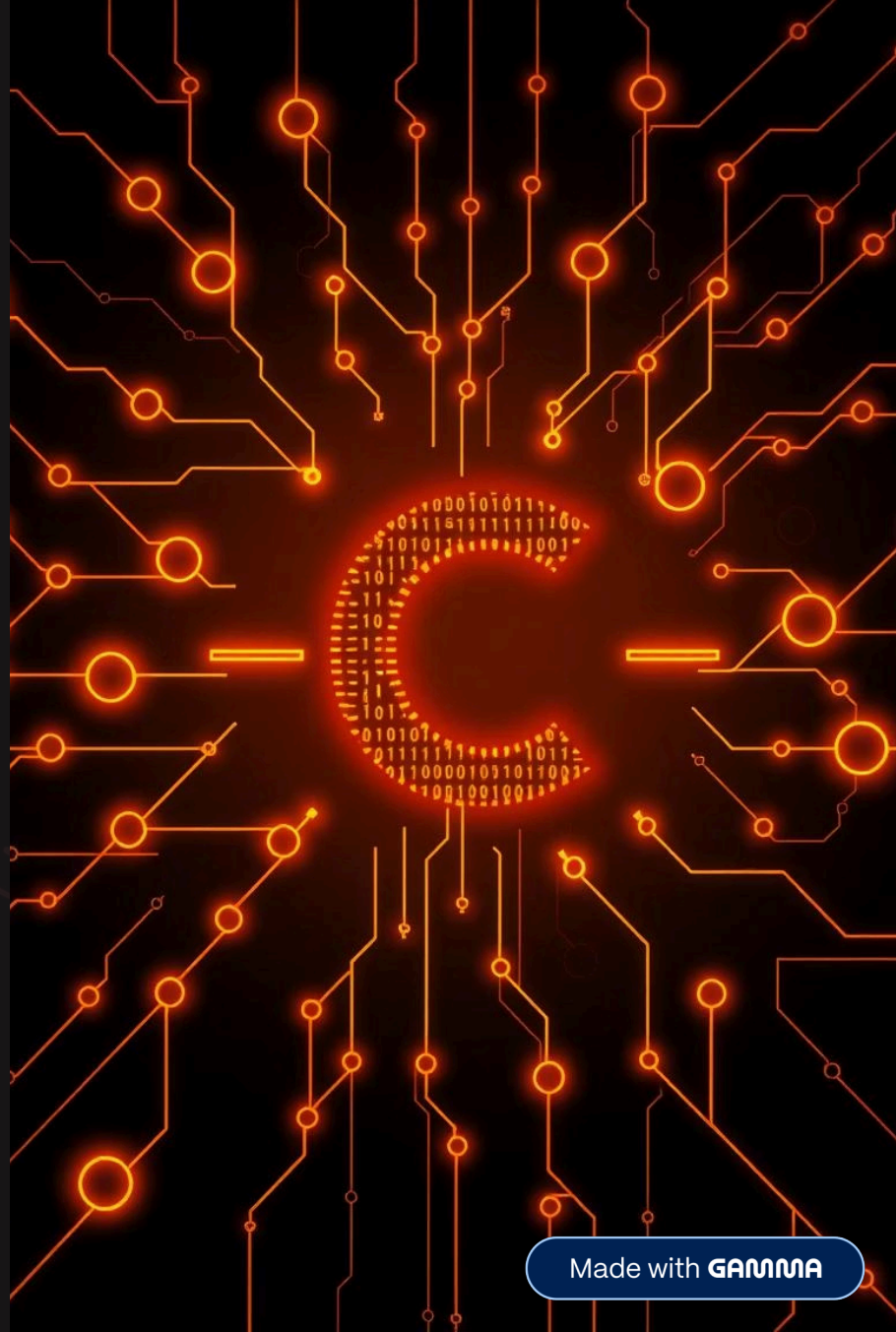
C Structs: A Concise Overview

In C, a **struct** (short for **structure**) is a user-defined data type that allows you to group different data types together. A struct is used to represent a collection of related data items, possibly of different types, as a single entity.

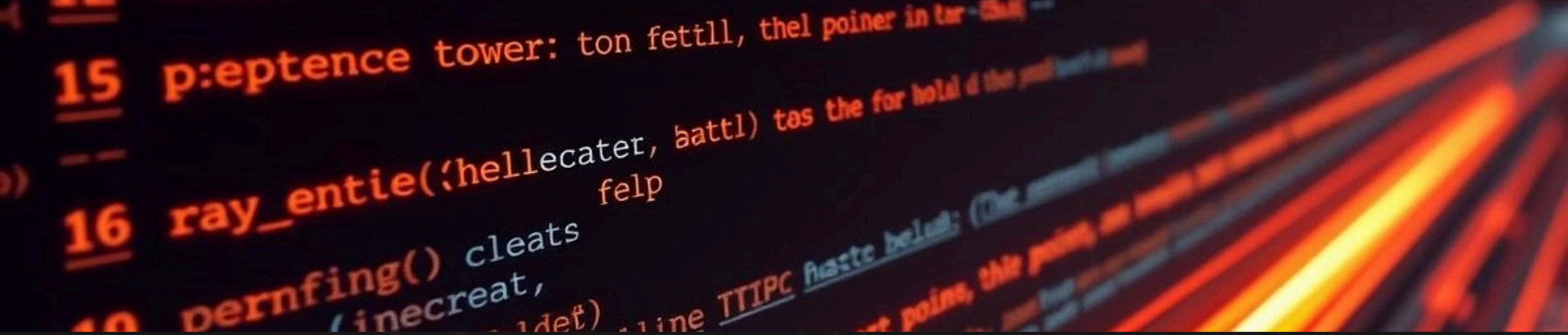
by Hamdi Emad



Structs: Definition and Usage

Usage

- **Grouping data**: A struct groups variables of different types under one name. These variables are referred to as **members** or **fields** of the struct.
- **Custom data types**: A struct is a way to define a custom data type, which can be used to store complex information.
- **Organized storage**: It helps in organizing and storing related data in a more manageable and logical way, especially when working with complex systems.



Pointers and Arrays within Structs

Pointers

Members can be pointers for dynamic memory use.

Example: *char *name;* with *malloc()* allocation.

Arrays

Structs can contain arrays for fixed-size data.

Common for lists of elements inside structs.

Use Cases

Linked lists and dynamic data structures rely on pointers.

Passing Structs to Functions

Pass by Value

Creates a copy of the struct, safe but slower.

Example: *void printPoint(struct Point p);*

Pass by Reference

Uses a pointer to the struct for efficiency.

Example: *void updatePoint(struct Point *p);*

Choose based on performance and data integrity needs.

Size of a Struct

Size Calculation

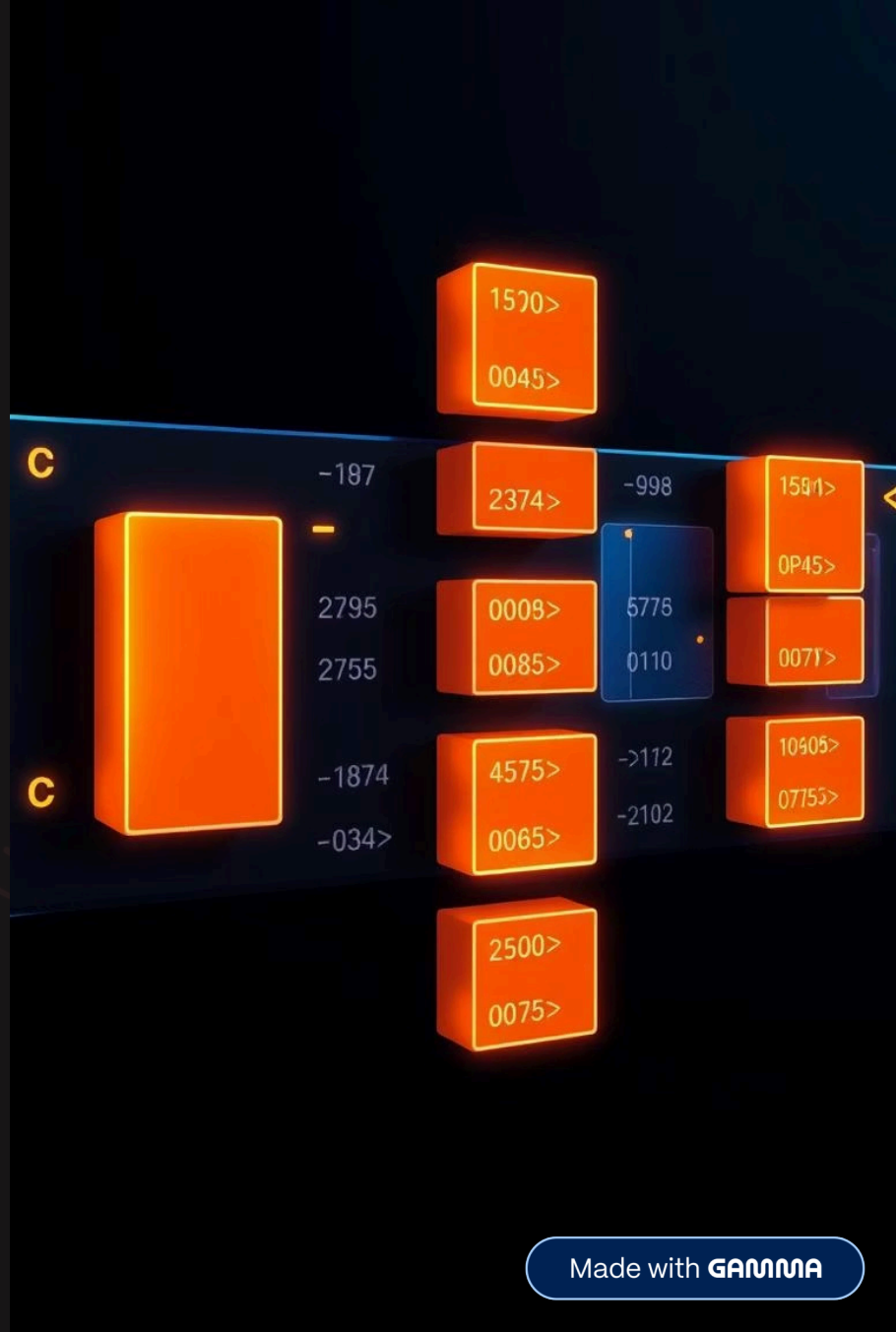
sizeof() gives total bytes used by a struct.

Impact of Padding

Size may exceed sum of member sizes due to padding.

Factors

Data types and alignment requirements influence size.



Memory Padding: Why it Matters

Definition

Unused bytes added to align data properly in memory.

Alignment Rules

Data types must align to specific address multiples.

Impact

Improves access speed but increases memory usage.

Optimization

Reorder members by size to reduce padding.

Aligned vs. Unaligned Memory Access

Aligned Memory

1. Accesses occur at addresses multiple of data size.

2. Enables fast and efficient data retrieval.

Unaligned Memory

1. Accesses at non-matching addresses, slower or faulty.

2. May cause hardware exceptions on some CPUs.

Compilers often insert padding or use directives to ensure alignment.

Aligned vs. Unaligned Memory: Example

Original Layout

Struct with char, int, char has internal padding.

Reordered Layout

Rearranged struct reduces padding significantly.

Performance

Aligned layout improves access speed and cache use.

Structs vs. Objects: Conceptual Differences

Structs (C)

- Contain only data (passive holders)
- No methods or behavior
- No inheritance or polymorphism

Objects (OOP)

- Encapsulate data and behavior
- Support inheritance and polymorphism
- Active entities with methods

C++ structs blur this line by supporting methods.

Conclusion: Structs as Building Blocks

Organized Data

Structs organize related variables into coherent types.

Memory & Performance

Understanding alignment boosts efficiency in software.

Function Interface

Efficient passing preserves speed and data integrity.

Distinct Paradigms

Structs differ fundamentally from object-oriented designs.