Why C, and not another language?

The primary design of C is to produce portable code while maintaining performance and minimizing footprint (CPU time, memory, disk I/O, etc.). This is useful for operating systems, embedded systems or other programs where performance matters a lot ("high-level" interface would affect performance)

One powerful reason is memory allocation. Unlike most programming languages, C allows the programmer to write directly to memory.

C gives control over the memory layout of data structures

Moreover, dynamic memory allocation is under the control of the programmer (which also means that memory deallocation has to be done by the programmer).

We use the C programming language in embedded systems instead of other languages because we deal with microcontrollers. This requires full access to all ranges on the memory bus of the System on Chip (SoC). Transactions on this bus are managed by modules that act as the master, with the most common module being the processor. The processor uses assembly instructions to execute commands like "store" and "load". C provides us with a window through pointers, allowing us to write directly to memory address ranges.

The differences between Python and C in accessing memory on an embedded system are significant, given the nature and design of each language. Here's a detailed comparison:

1. Memory Management

C:

- **Manual Memory Management**: In C, memory management is manual. Developers explicitly allocate and deallocate memory using functions like malloc() and free(). This gives fine-grained control over memory usage but also requires careful management to avoid issues like memory leaks and buffer overflows.
- **Pointers**: C uses pointers extensively for direct memory access and manipulation. This allows for efficient and direct interaction with hardware and memory addresses.

Python:

- **Automatic Memory Management**: Python manages memory automatically through garbage collection. The interpreter handles memory allocation and deallocation, which simplifies development but adds overhead.
- **No Direct Memory Access**: Python abstracts away direct memory access. It does not use pointers in the same way C does, which limits low-level memory manipulation but reduces the risk of memory-related errors.

2. Performance

C:

- **High Performance**: C is a compiled language, and its memory access operations are translated directly into machine code, resulting in fast and efficient execution.
- **Low-level Access**: Direct memory access and manipulation using pointers result in minimal overhead and high performance, which is critical in embedded systems.

Python:

- **Lower Performance**: Python is an interpreted language, which introduces overhead. Memory access in Python is slower compared to C due to the layers of abstraction.
- **High-level Abstraction**: Python's high-level abstraction provides ease of use but at the cost of performance, which can be a significant limitation in resource-constrained embedded systems.