# A Basic Overview of Memory

> Understanding how memory is structured and accessed is essential for understanding pointers, as pointers directly interact with memory addresses.

## Bits, Bytes, and Addresses

1. **Memory Representation:**
    - Memory can be visualised as a collection of boxes, where each box represents a **byte** of storage.
    - A byte consists of **8 bits** (binary digits: `0` or `1`).
    - A group of **4 bits** is known as a **nibble**.
    - Every byte in memory has a **unique address** used to locate and access data.
2. **Addresses:**
    - Memory addresses are represented in **hexadecimal format** (base-16) for readability and convenience.
    - For example, a memory address may be represented as: `0x7ffee6bdc8`.

---

## Storing Data in Memory

When a program is compiled, the **compiler allocates memory** for variables based on their **data type**. This process involves:

- **Reserving a specific number of bytes in memory** depending on the data type. For example, an `int` typically uses `4 bytes`, while a `double` uses `8 bytes`.
- **Assigning a memory address to each variable**, which acts as a unique identifier for accessing the stored value.
- **Ensuring contiguous memory allocation for arrays.** When arrays are declared, the compiler allocates a block of memory large enough to hold all elements, with each element placed consecutively. The memory address of each element is calculated using the starting address and the size of the data type.

Memory allocation occurs in two ways:

- **Compile-time (Stack Memory):** This is used for static variables and arrays, where sizes are known in advance, and memory is reserved from the stack.
- **Runtime (Heap Memory):** This is used for dynamically allocated variables (e.g., using `malloc()` in C), where memory is reserved from the heap as needed during program execution.

The number of bytes allocated for each data type is consistent across arrays, structures, and other memory operations, allowing the compiler to accurately calculate memory addresses and access data efficiently.

## Data Types and Their Sizes

| Data Type | Size (Bytes) |
|---|---|
| `char` | 1 |
| `short` | 2 |
| `int` | 4 |
| `long` | 8 |
| `float` | 4 |
| `double` | 8 |

---

## Memory Offsets for Arrays

When arrays are stored in memory, each element is placed in consecutive memory locations. The address of each element is calculated based on the data type size. This consistency allows efficient access to array elements.

### Example: Integer Array ( `int` )

```
int arr[4] = {10, 20, 30, 40};
```

| Element | Address (Hex) | Memory Offset |
|---|---|---|
| arr[0] | 0x7ffee6bd99c8 | +0 |
| arr[1] | 0x7ffee6bd99cc | +4 |
| arr[2] | 0x7ffee6bd99d0 | +8 |
| arr[3] | 0x7ffee6bd99d4 | +12 |

### Example: Long Array ( `long` )

```
long arr[4] = {1000, 2000, 3000, 4000};
```

| Element | Address (Hex) | Memory Offset |
|---|---|---|
| arr[0] | 0x7ffee6bd99c8 | +0 |
| arr[1] | 0x7ffee6bd99d0 | +8 |
| arr[2] | 0x7ffee6bd99d8 | +16 |
| arr[3] | 0x7ffee6bd99e0 | +24 |