Introduction to Pointers

What are pointers?

- Pointers are variables that store memory addresses of other variables, rather than actual values.
- They are used to **reference and manipulate data stored elsewhere in memory**.
- A pointer is said to **point to** the variable whose memory address it holds.

Why Use Pointers?

1. **Direct Memory Access:** Pointers provide direct access to memory locations, enabling faster read and write operations compared to working with copies of data.
2. **Efficient Function Arguments:** Instead of passing entire data structures to functions, which creates a copy of the data, pointers allow passing only the memory address. This minimises memory usage and improves performance. The memory address is always a fixed size: `4 bytes` on a `32-bit system` and `8 bytes` on a `64-bit system`.

Understanding Pointers Through a House Analogy

Imagine you want to show someone your house. You have three options:

1. **Bring them to your house yourself.** (Equivalent to directly providing the data)
2. **Create a copy of your house and give it to them.** (Making a duplicate of the data - Inefficient!)
3. **Just give them your house address.** (Using pointers - Efficient and direct)

- In programming, pointers are like providing the house address.
- They allow parts of your program to access data directly without making unnecessary copies.

Defining the `House` Structure

```
typedef struct {
    int SquareFeet;
    int NumBedrooms;
    int NumBathrooms;
} House;

House myHouse = { 1000, 3, 1 };
```

Defining Pointers

- To use pointers with the `House` structure, we need to **declare a pointer** and **assign it the address of an existing** `House` **object**.
- Use `*` to declare a pointer:

```
House *pMyHouse;
```

- Use `&` (address-of operator) to obtain the memory address of a variable:

```
pMyHouse = &myHouse;
```

- This can be done with a single line of code:

```
House *pMyHouse = &myHouse;
```

- The pointer type must match the type of the variable it points to (e.g., `House*` for `House`).

Dereferencing

- **Dereferencing:** This allows us to access or modify the value stored at the memory address a pointer holds.
- To dereference a pointer, we use the `*` operator.
- When working with objects, we can **alternatively** use the `→` operator which **automatically dereferences** the pointer and accesses the specified member value.

Accessing Values

- To dereference a pointer to access a value, use `*` before the pointer variable:

```
printf("Square Feet: %d", (*pMyHouse).SquareFeet);
// Outputs: 1000
```

- For objects, you can also use the `→` operator, which is equivalent to `(*pMyHouse).SquareFeet` but is simpler to read and use:

```
printf("Square Feet: %d", pMyHouse→SquareFeet);
// Outputs: 1000
```

- If you do not dereference the pointer, only the memory address is accessed, not the actual value.

Modifying Values

- When pointers are used to modify values, changes are applied directly to the original data, since you are working with the memory address itself.
- Example of modifying an object via a pointer:

```
void AddExtension(House *pHouse, int addSqFt, int addRooms, int addBaths) {
    pHouse->SquareFeet += addSqFt;
    pHouse->NumBedrooms += addRooms;
    pHouse->NumBathrooms += addBaths;
}

AddExtension(pMyHouse, 500, 1, 1);  // Directly modifies 'myHouse'
```

Pointers and Functions

Pass by Value: The function works with a copy of the data. Changes do not affect the original data.

Pass by Reference (Pointer): The function works with the original data by using its memory address. Changes are permanent. Using pointers in functions is particularly useful for modifying complex data structures or large arrays.

```
void InvitePeople(House *pHouse) {
    printf("House Details: %d SqFt, %d Bedrooms, %d Bathrooms\n",
           pHouse->SquareFeet, pHouse->NumBedrooms, pHouse->NumBathrooms);
}

InvitePeople(pMyHouse);  // Outputs details of the actual 'myHouse'
```

Pointers aren't just for structs and objects. They can also point to single-value data types like integers, floats, and characters. Access and modification follow the same dereferencing rules.

```
int number = 50;
int * pNumber = &number;
printf("%d\n", *pNumber);  // Access
*pNumber = 100; // Modification
```