C Programming Examples

Today's Agenda

1. Recap on Data types
2. Recap on Basic Structs
3. Recap on pointers
4. Pass by Value & Pass by Reference
5. Memory Management
6. Next Recitation: Assembly Programming

Note: Each code block is a standalone example. In practice, only one `main()` should be compiled at a time.

Basic Embedded Boilerplate

```
#include<mbed.h>

int main(){
  // Setup Code: Code meant to run only once

  while(1){
    // Loop Code: Code meant to run indefinitely (Infinite Loop!)
    thread_sleep_for(1000); // Sleep (Embedded Delay)
  }
}
```

Data Types

Why does using the right type of data types matter? C/C++ provides several primitive data types. In embedded systems, data type choice directly impacts performance, memory usage, and hardware compatibility.

- int: Integer, commonly 16 or 32-bit depending on target.
- char: Character data type, 8 bits.
- float/double: Floating-point numbers, often avoided in embedded systems due to lack of floating-point hardware.
- void: Used to indicate no return value.
- typedef/struct: Allow creation of custom data types.

```
#include "mbed.h"
int main()
{
 // Declare and Initialize a variable
   int my_integer = 1;
   int my_negative_integer = -1;
   bool is_true = true;
   bool is_false = false;
   char my_char = 'a';
```

```
    float my_float = 0.1;
    float my_negative_float = -0.1;

    uint8_t small = 255;              // 1 byte, 0 to 255
    int8_t small_signed = -128;       // 1 byte, -128 to 127
    int16_t medium = -123;            // 2 bytes
    uint16_t medium_u = 65535;        // 2 bytes unsigned
    int32_t large = 2147483647;       // 4 bytes signed
    uint32_t large_u = 123456;        // 4 bytes unsigned
    int64_t very_large = -9223372036854775807LL; // 8 bytes signed
    uint64_t very_large_u = 18446744073709551615ULL; // 8 bytes unsigned

    uint32_t register1 = 21;
    uint32_t register1_hex = 0x21213212;
    uint32_t register1_bin = 0b00000000'00000000'00000000'00000000;

    uint32_t *my_ptr = &register1; // more on pointer in next section
    printf("%d", *my_ptr);

    int x = 6;
    x = 5;

    volatile int v = 6;
    v = 5;

// What is the Difference between int and volatile int? Hint
(optimisation)

// When to use volatile

// Hardware registers: When a variable represents a memory-mapped hardware
register that can
// change independently of your program.

// Interrupt service routines: Variables modified by interrupt handlers
need to be volatile
// so the main program sees changes immediately.

// Multi-threaded programming: Volatile alone is not enough for shared
data.
// Use atomics or mutexes for synchronization. Volatile is for visibility
only.

// Signal handlers: Variables modified in signal handlers.

    while (1)
    {
    thread_sleep_for(500);
    }
}
```

```c
#include "mbed.h"

// Declaring Functions
uint32_t add_two_numbers(uint32_t a, uint32_t b)
{
   uint32_t res = a + b;
   return res;
}

uint32_t add_all_numbers(uint32_t arr[], uint32_t n)
{
   uint32_t res = 0;
   for (uint32_t i = 0; i < n; i++)
   {
       res = res + arr[i];
   }
   return res;
}

int main()
{
   uint32_t num1 = 2, num2 = 3;
   uint32_t sum_two = add_two_numbers(num1, num2);
   printf("value = %lu\n", (unsigned long)sum_two);

   uint32_t nums[] = {1, 2, 3, 4};
   uint32_t sum_all = add_all_numbers(nums, 4);

   printf("value = %lu\n", (unsigned long)sum_all);
   while (1)
   {
       thread_sleep_for(500);
   }
}
```

Sizeof and Alignment (Embedded)

Structs may include padding for alignment, which affects memory usage and peripheral layouts.

```c
#include "mbed.h"

struct Example {
    uint8_t  a;
    uint32_t b;
};

int main() {
    printf("sizeof(uint8_t)  = %lu\n", (unsigned long)sizeof(uint8_t));
    printf("sizeof(uint32_t) = %lu\n", (unsigned long)sizeof(uint32_t));
    printf("sizeof(Example)  = %lu\n", (unsigned long)sizeof(Example));
```

```
    while (true) {
        thread_sleep_for(1000);
    }
}
```

Pointers

Pointers are variables that hold memory addresses. They are essential for referencing hardware registers, efficient array manipulation, and dynamic memory.

- The `*` (dereference operator) is used to access the value stored at the pointer's address.
- The `&` (address-of operator) is used to get the address of a variable.

Pointer Arithmetic

- Since pointers store addresses, you can move across memory using arithmetic.
- Incrementing (`p++`) moves the pointer to the next element of its type.

Pointers and Arrays

- Arrays and pointers are closely related: in most expressions, the array name decays to a pointer to the first element.

Pointer to Pointer

- You can have multiple levels of pointers (e.g., `int **pp`).
- Common in dynamic memory allocation and working with command-line arguments.

Function Pointers

- Allow storing the address of a function in a pointer.
- Useful for callback mechanisms, interrupt handling in embedded systems, and implementing jump tables.

Void Pointers

- `void *` can point to any type, making it generic.
- Must be typecast before dereferencing. \

Dangling and Null Pointers

- **Dangling pointer**: points to memory that is freed or invalid. \
- **Null pointer**: pointer explicitly assigned to `NULL`, meaning it points nowhere. \
- Always initialize pointers to `NULL` and check before dereferencing. \

Why pointer matter in embedded systems?

- **Memory Access**

    - Example: Accessing hardware registers via memory-mapped addresses.
- **Efficiency**
    - Avoids copying large data structures (pass by reference instead of value).
- **Arrays & Strings**
    - Arrays are closely tied to pointers; pointer arithmetic lets you navigate through them.
- **Dynamic Access**
    - Useful for managing lookup tables, communication buffers, or sensor data streams.

To master pointers, it takes a lot of time and practice. Here, we've just given a small recap and introduction.

```
#include<mbed.h>

int main() {

    int x = 42;
    int *p = &x; // pointer to x

    printf("x = %d\n", x);
    printf("Address of x (&x) = %p\n", &x);
    printf("Pointer p = %p, *p = %d\n", p, *p);

    *p = 100; // modify value through pointer
    printf("After *p = 100, x = %d\n", x);

    int arr[5] = {10, 20, 30, 40, 50};
    int *pa = arr; // points to first element

    printf("\nArray via pointer:\n");
    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d, *(pa+%d) = %d\n",
               i, arr[i], i, *(pa + i));
    }

    const int val = 200;
    const int *ptr_to_const = &val;  // can't modify the value via pointer
    // *ptr_to_const = 300; //not allowed

    int another = 500;
    int *const const_ptr = &another;  // pointer itself can't move
    *const_ptr = 600;                 // but can modify value

    printf("\nval = %d, another = %d\n", val, another);

    // Volatile pointer
    volatile uint32_t fake_register = 0x12345678;
    volatile uint32_t *reg_ptr = &fake_register;

    printf("\nHardware register (before) = 0x%08X\n", *reg_ptr);
    *reg_ptr = 0xDEADBEEF; // here we are just simulating writing to
register
    printf("Hardware register (after)  = 0x%08X\n", *reg_ptr);
```

```
    while (true) {
        thread_sleep_for(1000);
    }
    return 0;
}
```

Structure

- What are Structures?
- A structure is a user-defined data type in C/C++ that groups different variables (members) under one name.
- Key word is struct
- Members can be of different types (int, float, array, etc.).
- You can access members using (.) Dot operator.

```
#include "mbed.h"

struct Point { //Basic struct definition
    int x;
    char y;
};

void demo_basic_struct() {
    Point p1;        // declare struct variable
    p1.x = 10;
    p1.y = 'A';

    printf("\n--- Basic Struct ---\n");
    printf("Point p1: x = %d, y = %c\n", p1.x, p1.y);

    // Initialization at declaration
    Point p2 = {30, 'B'};
    printf("Point p2: x = %d, y = %c\n", p2.x, p2.y);
}

void demo_array_of_structs() {                  // 2. Array of Structs
    Point points[3] = {{1,'A'}, {3,'C'}, {5,'D'}};

    printf("\n--- Array of Structs ---\n");
    for (int i = 0; i < 3; i++) {
        printf("points[%d]: x = %d, y = %c\n", i, points[i].x,
points[i].y);
    }
}

// struct Polygon {    // 3. Struct containing Arrays
//     int sides[5];  // array inside struct
//     int num_sides;
// };
```

```
// void demo_struct_with_arrays() {
//     Polygon poly = {{10, 20, 30, 40, 50}, 5};

//     printf("\n--- Struct Containing Arrays ---\n");
//     printf("Polygon with %d sides: ", poly.num_sides);
//     for (int i = 0; i < poly.num_sides; i++) {
//         printf("%d ", poly.sides[i]);
//     }
//     printf("\n");
// }

int main() {
    demo_basic_struct();
    thread_sleep_for(500);

    demo_array_of_structs();
    thread_sleep_for(500);

    // demo_struct_with_arrays();
    // thread_sleep_for(500);

    while (true) {
        thread_sleep_for(1000);
    }
}
```

Pass by Value

- A **copy of the variable** is passed to the function.
- Changes inside the function **do not affect** the original variable.

Pass by Reference

- Instead of copying, the function gets **direct access to the original variable by passing the address**.
- Can be done with **pointers (*)** in C, or **references (&)** in C++.
- Changes inside the function **will affect** the original variable unlike in pass by value.
- Useful in cases where you have **save memory** (avoid copying large structs/arrays) and for **direct hardware manipulation**.

```
#include "mbed.h"

void increment_by_value(int a) { // Pass by Value
    a = a + 5;
    printf("Inside increment_by_value: a = %d\n", a);
}

void increment_by_pointer(int *a) { // Pass by Pointer
    (*a) = (*a) + 5;
    printf("Inside increment_by_pointer: *a = %d\n", *a);
```

```
}

void increment_by_reference(int &a) { // Pass by Reference
    a = a + 5;
    printf("Inside increment_by_reference: a = %d\n", a);
}

int main() {

    int x = 10;

    increment_by_value(x);
    printf("After increment_by_value, x = %d\n\n", x);

    increment_by_pointer(&x);
    printf("After increment_by_pointer, x = %d\n\n", x);

    increment_by_reference(x);
    printf("After increment_by_reference, x = %d\n\n", x);

        while (true) {
        thread_sleep_for(1000);
    }

}
```

Memory Management

Types of Memory in C

- **Stack**
  - Automatically managed by the compiler.
  - Stores local variables, function parameters, and return addresses.
  - Grows and shrinks as functions are called/returned.
  - Very fast, but limited in size.
- **Heap**
  - Dynamically allocated memory using `malloc`, `calloc`, or `realloc`.
  - Persists until explicitly freed using `free()`.
  - More flexible than stack, but slower and prone to fragmentation.

Why Dynamic Memory Can Be Risky in Embedded Systems

- **Fragmentation**: Over time, the heap can get divided into unusable small chunks.
- **Memory Leaks**: If `free()` is forgotten, memory is permanently lost until reboot.
- **Unpredictability**: Heap allocation takes variable time, which is dangerous in real-time systems.
- **Standards**: Some high-reliability systems (e.g., avionics, automotive) forbid dynamic allocation after initialization.

Best Practices for Embedded Systems

Prefer static or stack allocation:

- char buffer[100]; // stack allocated, automatically managed
- **Use memory pools**: Pre-allocate a large block at startup and manage it manually.
- **Check every allocation**: Always verify if `malloc` returned `NULL`.
- **Free properly**: Every successful `malloc`/`calloc`/`realloc` must have a matching `free`.

Common Pitfalls

- **Double Free**: Calling `free()` twice on the same pointer → undefined behavior.
- **Use After Free**: Using memory after `free()` was called.
- **Dangling Pointers**: Pointers left pointing to freed or invalid memory.
- **Not Freeing**: Forgetting to release memory (leads to leaks).

Example Scenarios

Stack Overflow (literally): Recursive functions without a base case can exhaust stack memory.

```
 void recurse() {
    int x[1000]; // large stack allocation
    recurse();   // eventually crashes
}
```

Heap Misuse:

```
 int *p = (int*) malloc(sizeof(int));
free(p);
*p = 5;  // ERROR: use-after-free
```

Embedded Alternative Approaches

- **Static Buffers**: Allocate all memory upfront before the system starts.
- **Ring Buffers / Circular Queues**: Useful for communication between tasks without dynamic allocation.
- **Compile-Time Allocation**: Fixed memory layout defined at compile time ensures determinism.

Bitwise Operations

This code is a demonstration of bitwise operations, which allow us to manipulate individual bits within a number. Think of bits as a series of on/off switches, and these operations let us control them directly. The program performs a series of bitwise tests and then uses macros to show how to set, clear, and toggle specific bits.

The Four Bitwise Operations

- **AND (&):** Checks if two bits are both 1. If they are, the result is 1. Otherwise, it's 0. The code tests x & y. Since x (0000 1111) and y (1111 0000) have no matching 1s, the result is 0x00.
- **OR (|):** Checks if at least one of two bits is 1. If either bit is 1, the result is 1. The code tests x | y. Since x has 1s in the lower four positions and y has 1s in the upper four, the result is 1111 1111, which is 0xFF.
- **XOR (^):** Checks if two bits are different. If they are different (1 and 0), the result is 1. The code tests x ^ y. Just like the OR operation, the bits are different at every position, so the result is also 0xFF.
- **Shift (<<):** This operation moves all bits to the left, which is equivalent to multiplying the number by 2 for each position shifted. The code tests x << 2, shifting 0000 1111 two positions to the left, resulting in 0011 1100, which is 0x3C.

Bit Manipulation Macros

The code also uses macros to perform specific bit-level tasks on a variable z.

- **SET_BIT_MACRO:** Uses the | operator to force a specific bit to 1.
- **CLEAR_BIT_MACRO**: Uses & and ~ to force a specific bit to 0.
- **TOGGLE_BIT_MACRO**: Uses ^ to flip a specific bit from 0 to 1 or 1 to 0.
- **CHECK_BIT_MACRO**: Uses a right shift (>>) and a bitwise AND to see if a specific bit is 1 or 0.

The final if statement checks a success variable that tracks the results of each test. The program will report if all tests passed or if any of them failed.

Example of a memory-mapped register access pattern (address is illustrative):

```
#define REG_CONTROL   (*(volatile uint32_t *)0x40000000U)
#define CTRL_ENABLE   (1U << 0)
#define CTRL_RESET    (1U << 1)

void enable_peripheral(void) {
    REG_CONTROL |= CTRL_ENABLE;   // set bit
    REG_CONTROL &= ~CTRL_RESET;   // clear bit
}
```

```
#include <stdio.h>
#include <stdbool.h>

// Macros for bitwise operations
#define SET_BIT_MACRO(value, bit)    ((value) |= (1U << (bit)))
#define CLEAR_BIT_MACRO(value, bit)  ((value) &= ~(1U << (bit)))
```

```c
#define TOGGLE_BIT_MACRO(value, bit) ((value) ^= (1U << (bit)))
#define CHECK_BIT_MACRO(value, bit)  (((value) >> (bit)) & 1U)

void bitwiseDemo(void)
{
    unsigned int x = 0x0F; // 0000 1111
    unsigned int y = 0xF0; // 1111 0000
    unsigned int result;
    bool success = true;

    printf("\n=== Bitwise Demo ===\n");

    result = x & y;
    printf("x & y = 0x%X\n", result);
    success = success && (result == 0x00);

    result = x | y;
    printf("x | y = 0x%X\n", result);
    success = success && (result == 0xFF);

    result = x ^ y;
    printf("x ^ y = 0x%X\n", result);
    success = success && (result == 0xFF);

    result = x << 2;
    printf("x << 2 = 0x%X\n", result);
    success = success && (result == 0x3C);

    unsigned int z = 0x0A; // 1010 in binary
    printf("\nInitial z = 0x%X\n", z);

    SET_BIT_MACRO(z, 0); // set bit 0
    printf("After setting bit 0 : z = 0x%X\n", z);

    CLEAR_BIT_MACRO(z, 3); // clear bit 3
    printf("After clearing bit 3: z = 0x%X\n", z);

    TOGGLE_BIT_MACRO(z, 1); // toggle bit 1
    printf("After toggling bit 1: z = 0x%X\n", z);

    printf("Bit 2 is currently: %d\n", CHECK_BIT_MACRO(z, 2));

    if (success)
        printf("\nAll bitwise tests PASSED!\n");
    else
        printf("\nSome bitwise tests FAILED!\n");
}

int main(void)
{
    bitwiseDemo();
    return 0;
}
```