c_programming_session_5

اللهمَّ علِّمنا ما ينفعنا، وانفعنا بما علمتنا، وزدنا علمًا. وافتح علينا فتحًا عظيمًا.

Tags: c programming
Status: #Adult

C Programming Session 5 - Enums, Unions, and Advanced Directives

Enumerations (enum)

Basic Enumeration Syntax

```
enum days {
    Saturday,    // 0
    Sunday,      // 1
    Monday,      // 2
    Tuesday,     // 3
    Wednesday,   // 4
    Thursday,    // 5
    Friday       // 6
};

enum days today;
today = Sunday;  // today = 1
```

Custom Enumeration Values

```
enum days {
    Saturday,           // 0
    Sunday,             // 1
    Monday,             // 2
    Tuesday = 5,        // 5 (explicitly set)
    Wednesday,          // 6 (continues from previous)
    Thursday,           // 7
    Friday              // 8
};
```

```
enum days weekend = Friday;  // weekend = 8
```

Typedef with Enumerations

```
typedef enum directions {
    North,
    South,
    East,
    West
} Direction;

Direction current_dir = North;
```

Practical Example: Direction Control

```
#include <stdio.h>

typedef enum directions {
    North = 0,
    South = 1,
    East = 2,
    West = 3
} Direction;

int main() {
    Direction code;

    while (1) {
        printf("Enter direction code (0-3): ");
        scanf("%d", (int*)&code);

        if (code < 0 || code > 3) {
            printf("Invalid direction code, please try again.\n");
            continue;
        }

        switch (code) {
            case North:
                printf("Moving North\n");
                break;
            case South:
```

```
                printf("Moving South\n");
                break;
            case East:
                printf("Moving East\n");
                break;
            case West:
                printf("Moving West\n");
                break;
            default:
                printf("Unknown direction\n");
                break;
        }
    }

    return 0;
}
```

Unions

Union vs Structure Comparison

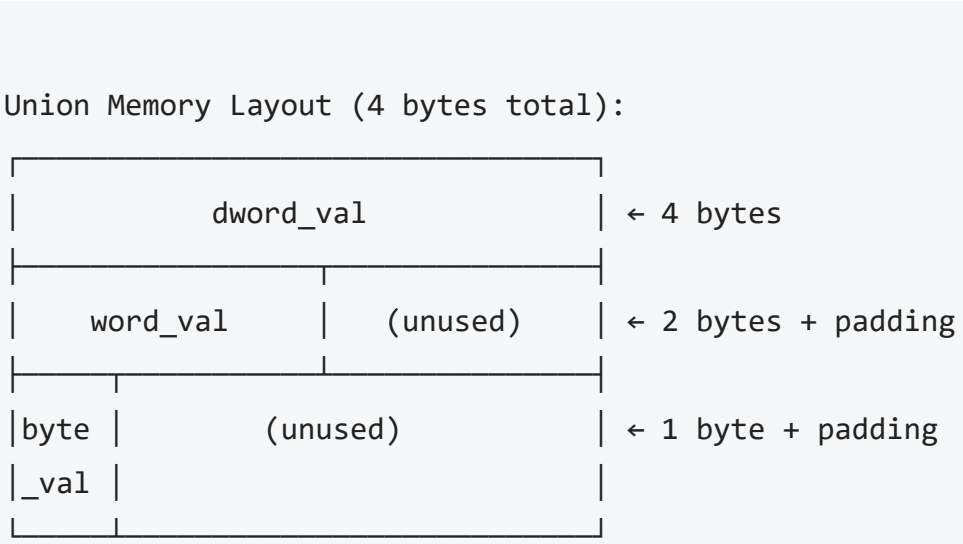
Feature	Structure	Union
Memory Usage	Sum of all members	Size of largest member
Data Access	All members simultaneously	One member at a time
Memory Layout	Sequential allocation	Overlapping allocation
Use Case	Group related data	Alternative data representations

Basic Union Example

```
union data {
    uint8_t  byte_val;    // 1 byte
    uint16_t word_val;    // 2 bytes
    uint32_t dword_val;   // 4 bytes
};

union data my_data;
```

```
// Total union size = 4 bytes (largest member)
// All members share the same memory location
```

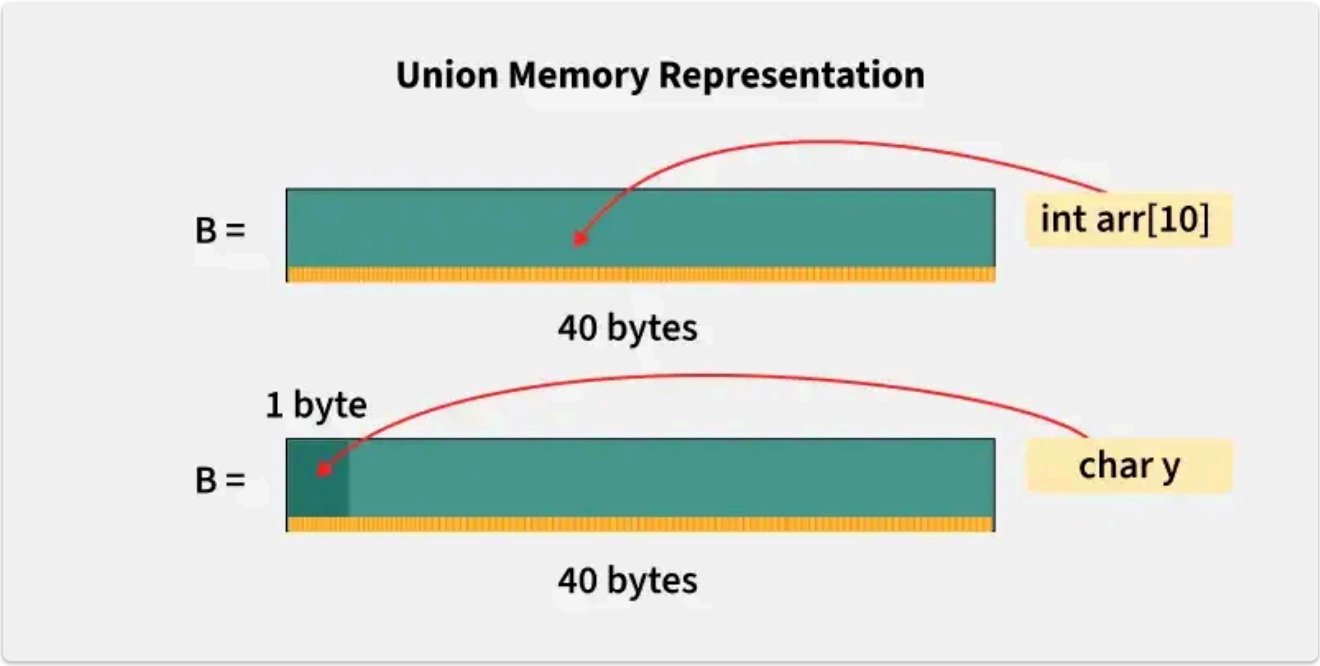
Memory Layout Visualization

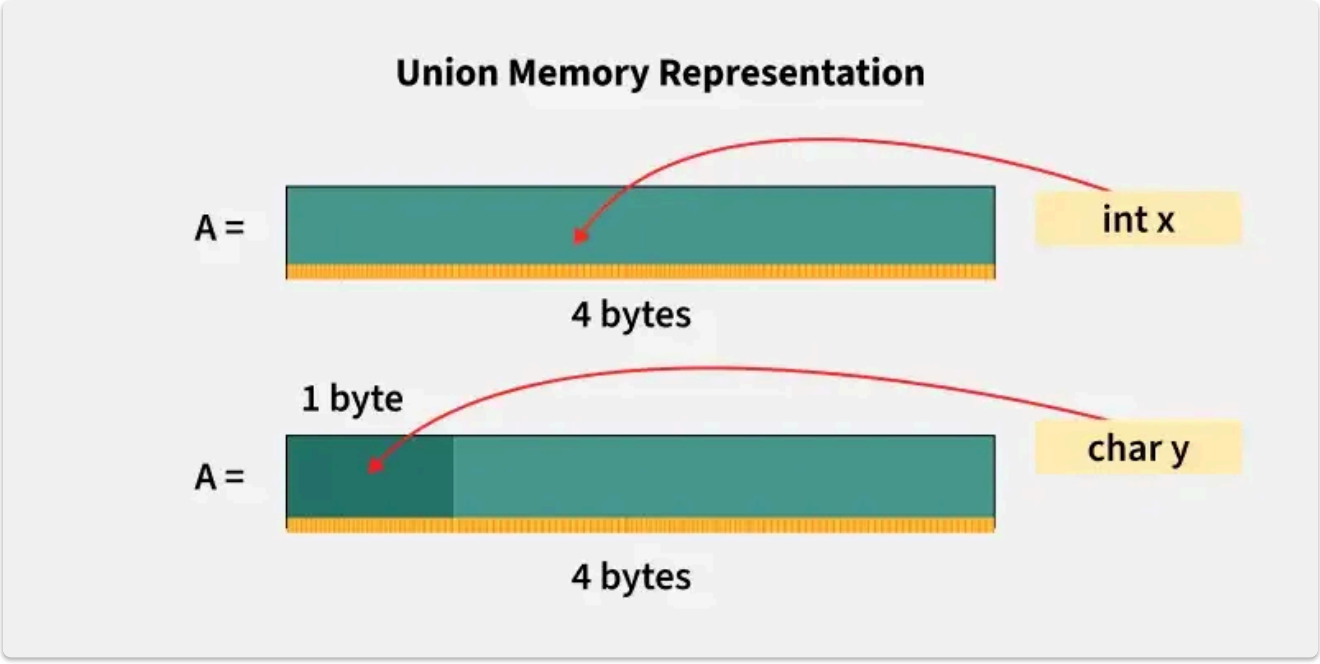


Another Union Example

```c
#include <stdio.h>

union A{
    int x;
    char y;
};

union B{
    int arr[10];
    char y;
};


int main() {

    // Finding size using sizeof() operator
    printf("Sizeof A: %ld\n", sizeof(union A));
    printf("Sizeof B: %ld\n", sizeof(union B));
    return 0;
}
```

```
Sizeof A: 4
Sizeof B: 40
```





Union Overwriting Behavior

```c
#include <stdio.h>

union example {
    uint8_t  x;  // 1 byte
    uint16_t y;  // 2 bytes
    uint32_t z;  // 4 bytes
};

int main() {
```

```
    union example data;

    data.z = 0x12345678;
    printf("z = 0x%08X\n", data.z);         // z = 0x12345678
    printf("y = 0x%04X\n", data.y);         // y = 0x5678 (lower 2 bytes)
    printf("x = 0x%02X\n", data.x);         // x = 0x78 (lowest byte)

    data.y = 0xABCD;
    printf("After setting y:\n");
    printf("z = 0x%08X\n", data.z);         // z = 0x1234ABCD (upper
bytes unchanged)
    printf("y = 0x%04X\n", data.y);         // y = 0xABCD
    printf("x = 0x%02X\n", data.x);         // x = 0xCD (lowest byte of
y)

    return 0;
}
```

Practical Union Application: Register Manipulation

```
typedef union {
    struct {
        uint8_t B0 : 1;  // Bit 0
        uint8_t B1 : 1;  // Bit 1
        uint8_t B2 : 1;  // Bit 2
        uint8_t B3 : 1;  // Bit 3
        uint8_t B4 : 1;  // Bit 4
        uint8_t B5 : 1;  // Bit 5
        uint8_t B6 : 1;  // Bit 6
        uint8_t B7 : 1;  // Bit 7
    } Bit;
    uint8_t Byte;
} Register;

Register control_reg;

// Access whole byte
control_reg.Byte = 0xA5;  // Sets all bits: 10100101

// Access individual bits
control_reg.Bit.B0 = 1;   // Set bit 0
control_reg.Bit.B7 = 0;   // Clear bit 7
```

```
printf("Register value: 0x%02X\n", control_reg.Byte);
```

Advanced Preprocessor Directives

Include Directives

Syntax	Search Path	Use Case
`#include <file.h>`	System directories	Standard library headers
`#include "file.h"`	Current directory, then system	User-defined headers

Cross-Directory Includes

```
// Including from different directories
#include <stdio.h>
#include "D:\project\session5\math_utils.h"
#include "D:\project\session5\gpio_driver.h"

int main() {
    printf("Cross-directory include example\n");
    return 0;
}
```

Macro Definitions

Object-like Macros

```
#define MAX_TEMPERATURE 35
#define PI 3.14159
#define MAX_SIZE 100

int temp = MAX_TEMPERATURE;  // Replaced with: int temp = 35;
```

Function-like Macros

```
#define AREA(l, w) ((l) * (w))
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

```
int rectangle_area = AREA(5, 10);     // Expands to: ((5) * (10))
int square_area = SQUARE(7);          // Expands to: ((7) * (7))
int maximum = MAX(a, b);              // Expands to: ((a) > (b) ? (a) :
(b))
```

Bit Manipulation Macros

```
// math_utils.h
#define SET_BIT(var, bit)    ((var) |= (1 << (bit)))
#define CLEAR_BIT(var, bit)  ((var) &= ~(1 << (bit)))
#define TOGGLE_BIT(var, bit) ((var) ^= (1 << (bit)))
#define GET_BIT(var, bit)    (((var) >> (bit)) & 1)

// Usage example
int main() {
    uint8_t value = 0x00;  // 00000000

    SET_BIT(value, 2);     // 00000100 (set bit 2)
    SET_BIT(value, 5);     // 00100100 (set bit 5)
    CLEAR_BIT(value, 2);   // 00100000 (clear bit 2)
    TOGGLE_BIT(value, 1);  // 00100010 (toggle bit 1)

    printf("Value: 0x%02X\n", value);  // Value: 0x22
    return 0;
}
```

Bit Manipulation Practice Program

```
#include <stdio.h>
#include "math_utils.h"  // Contains bit manipulation macros

int main() {
    int x, bit, operation, result;

    while (1) {
        printf("Enter number: ");
        scanf("%d", &x);
        printf("Enter bit position: ");
        scanf("%d", &bit);
        printf("Choose operation (1:set, 2:clear, 3:toggle): ");
        scanf("%d", &operation);
```

```
        switch (operation) {
            case 1:
                SET_BIT(x, bit);
                printf("After SET bit %d: %d\n", bit, x);
                break;
            case 2:
                CLEAR_BIT(x, bit);
                printf("After CLEAR bit %d: %d\n", bit, x);
                break;
            case 3:
                TOGGLE_BIT(x, bit);
                printf("After TOGGLE bit %d: %d\n", bit, x);
                break;
            default:
                printf("Invalid operation\n");
                break;
        }
    }

    return 0;
}
```

Conditional Compilation

#if Directives

```
#define VERSION 2

#if VERSION > 1
    printf("Advanced features enabled");
#elif VERSION == 1
    printf("Basic features enabled");
#else
    printf("Legacy mode");
#endif
```

Header Guards

```
// Prevent multiple inclusions
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

// Header content
```

```c
int add(int a, int b);
float multiply(float a, float b);

#endif // MATH_UTILS_H
```

Multiple Inclusion Problem

```
file1.c includes:
  ├── file2.h
  └── file3.h
      └── includes file2.h again!
```

Solution: Header Guards

```c
// file2.h
#ifndef FILE2_H
#define FILE2_H

// Content of file2.h
void some_function();

#endif // FILE2_H
```

Multi-line Macros

```c
// Method 1: Line continuation
#define PRINT_INFO() \
    printf("Name: Fares\n"); \
    printf("Course: NTI\n"); \
    printf("Language: C\n");

// Method 2: do-while(0) idiom (safer)
#define PRINT_INFO() do { \
    printf("Name: Fares\n"); \
    printf("Course: NTI\n"); \
    printf("Language: C\n"); \
} while(0)
```

Compiler Diagnostics

```c
#warning "This feature is deprecated"
#error "This code requires C99 or later"

// Conditional error
#if !defined(__STDC_VERSION__) || (__STDC_VERSION__ < 199901L)
    #error "This code requires C99 standard"
#endif
```

Practical Examples

qsort Second Largest Element

```c
#include <stdio.h>
#include <stdlib.h>

// Comparison function for qsort
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
    // Returns: negative if a < b, 0 if a == b, positive if a > b
}

int get_second_largest(int arr[], int n) {
    // Sort array in ascending order
    qsort(arr, n, sizeof(int), compare);

    // Find first element different from largest
    for (int i = n - 2; i >= 0; i--) {
        if (arr[i] != arr[n - 1]) {
            return arr[i];
        }
    }

    return -1;  // No second largest found
}

int main() {
    int arr[] = {12, 35, 1, 10, 34, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    int second_largest = get_second_largest(arr, n);
    printf("Second largest: %d\n", second_largest);
```

```
    return 0;
}
```

Understanding qsort Compare Function

```
// Step-by-step explanation:
// 1. (int *)a  → Cast void pointer to int pointer
// 2. *(int *)a → Dereference to get actual int value
// 3. Same for b
// 4. Return difference for sorting order

// Example: if a points to 5, b points to 8
// *(int *)a = 5
// *(int *)b = 8
// return 5 - 8 = -3 (negative means a comes before b)
```

File Organization Best Practices

Project Structure

```
project/
├── src/
│   ├── main.c
│   ├── utilities.c
│   └── drivers.c
├── include/
│   ├── utilities.h
│   ├── drivers.h
│   └── config.h
├── build/
│   └── (compiled objects)
└── docs/
    └── README.md
```

Header File Template

```
// utilities.h
#ifndef UTILITIES_H
#define UTILITIES_H
```

```c
#include <stdint.h>

// Function prototypes
uint32_t calculate_checksum(uint8_t *data, uint16_t length);
void delay_ms(uint32_t milliseconds);
int32_t map_range(int32_t value, int32_t from_low, int32_t from_high,
                  int32_t to_low, int32_t to_high);

// Global constants
#define MAX_BUFFER_SIZE 256
#define DEFAULT_TIMEOUT 1000

// Global variables (extern declarations)
extern uint8_t system_status;
extern uint32_t error_count;

#endif // UTILITIES_H
```

Common Pitfalls and Best Practices

Macro Safety

```c
// WRONG: Unprotected macro
#define SQUARE(x) x * x
int result = SQUARE(3 + 2);  // Expands to: 3 + 2 * 3 + 2 = 11 (not 25!)

// CORRECT: Protected macro
#define SQUARE(x) ((x) * (x))
int result = SQUARE(3 + 2);  // Expands to: ((3 + 2) * (3 + 2)) = 25
```

Side Effects in Macros

```c
// DANGEROUS: Multiple evaluation
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int x = 5;
int result = MAX(++x, 10);  // x is incremented twice if ++x > 10!

// SAFER: Use inline functions for complex operations
static inline int safe_max(int a, int b) {
    return (a > b) ? a : b;
}
```

Advanced Topics Preview

Tomorrow's session will cover:

- **malloc(), calloc(), free(), realloc():** Dynamic memory management
- **Linked Lists:** Dynamic data structures
- **Memory management:** Best practices and common pitfalls
- **Data structure implementation:** Practical examples

Key Takeaways

1. **Enums:** Provide readable constants and type safety
2. **Unions:** Enable memory-efficient alternative data representations
3. **Macros:** Powerful but require careful implementation
4. **Header Guards:** Essential for preventing multiple inclusions
5. **File Organization:** Proper structure improves maintainability

Practice Exercises

1. Create an enum for HTTP status codes (200, 404, 500, etc.)
2. Design a union for handling different data packet types
3. Write header guards for a custom math library
4. Implement a bit manipulation library using macros
5. Create a multi-file project with proper organization