Introduction to Computing

Advanced Things - I

Recall Array

```
int states[5], i;                    //declaration
for (i=0; i<5; i++)
        states[i]=i;                 // initialization
for (i=0; i<5; i++)
        printf("%d, ", states[i]);   //access
        // outputs 1, 2, 3, 4, 5,
```

```
int a[5]; int *arr;
printf("%d, %d, %d \n", sizeof(int), sizeof(a[3]), sizeof(a));
        // outputs 4, 4, 20
printf (“%d”, sizeof (arr));              // outputs 8
arr = a;
printf (“%d”, sizeof (arr));              // outputs ??
```

- This is also called a one dimensional array or 1D array
- Sometimes we need to work on multidimensional data, e.g. 2D coordinates, matrix, system of equations
- 1D array is not convenient enough for such problems

2D Array

- int arr[m][n];
  - m is the number of rows and n is the number of columns
  - Array of m*n integers
  - Useful to store multidimensional data
- Accessing element at i^{th} row and j^{th} column using arr[i][j]
- Each arr[i] is an 1D array of size n

2D Array (contd.)

```
int a[2][3], i, j; //declaration

//initialization
for (i=0; i<2; i++)
    for (j=0; j<3; j++)
        a[i][j] = i + j;

//access
for (i=0; i<2; i++)
    for (j=0; j<3; j++)
        printf("%d, ", a[i][j]);

printf("%d, %d, %d", sizeof(a), sizeof(a[1]), sizeof(a[1][2]));
//outputs 24, 12, 4
```

3D Array

- int arr [m][n][p];
  - Array of m*n*p integers
- Each arr[i] is an 2D array of size n*p integers
- Each arr[i][j] is an 1D array of p integers
- Each arr[i][j][k] is a single integer element

→how to calculate the address of any element in the array?

3D Array (contd.)

```
int a[2][3][4], i, j, k; //declaration

//initialization
for (i=0; i<2; i++)
    for (j=0; j<3; j++)
        for (k=0; k<4; k++)
            a[i][j][k] = (i + j)*k;

//access
for (i=0; i<2; i++)
    for (j=0; j<3; j++)
        for (k=0; k<4; k++)
            printf("%d, ", a[i][j][k]);

printf("%d, %d, %d, %d \n", sizeof(a), sizeof(a[1]), sizeof(a[0][2]), sizeof(a[1][0][2]));
//outputs ??
```

Memory layout of Arrays: Row-Major and Column-Major

Row-Major:

- Stores data row by row: All elements of the first row are stored, followed by all elements of the second row, and so on.
- Address of arr[i][j] = Base Address + ((i * Columns) + j) * sizeof(data_type)
- Used by C and C++.

Column-Major:

- Stores data column by column: All elements of the first column are stored, followed by all elements of the second column, and so on.
- Address of arr[i][j] = Base Address + ((j * Rows) + i) * sizeof(data_type)
- Used by Fortran and MATLAB.

Why It Matters?

- Affects how multi-dimensional arrays are stored in memory.
- Impacts performance in terms of cache efficiency and access patterns.

Arrays in C:
Arrays are stored in **contiguous memory** locations.

For multi-dimensional arrays in C: **Row-major order** is used by default.

Key Terminology:

- **Base Address**: Address of the first element.
- **Index Calculation**: Computed based on the dimensions and row-major order.

Memory Layout of 2D Arrays

A 2D array `arr[R][C]` is stored row by row.

Flattened into a 1D representation:
`arr[0][0], arr[0][1], ..., arr[0][C-1], arr[1][0], ..., arr[R-1][C-1]`

Memory Address Calculation:
Address of `arr[i][j] = Base Address + ((i * Columns) + j) * sizeof(data_type)`

```
int arr[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Flattened in memory as:

`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`

Memory Layout of 3D Arrays

A 3D array `arr[X][Y][Z]` is stored as:

- First, all elements of the 1st slice (`arr[0][Y][Z]`),
- Then, all elements of the 2nd slice (`arr[1][Y][Z]`)
- Then 3rd slice, *and so on.*

Memory Address Calculation:

Address of `arr[i][j][k] = Base Address + ((i * Y * Z) + (j * Z) + k) * sizeof(data_type)`

```
int arr[2][3][2] = {
    {{1, 2}, {3, 4}, {5, 6}},
    {{7, 8}, {9, 10}, {11, 12}}
};
```

Flattened in memory as:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Visual Representation of 3D Array Layout

Layers (Slices):

- `arr[0][*][*]`: Contains all elements from the first slice.
- `arr[1][*][*]`: Contains all elements from the second slice.

```
int arr[2][3][2] = {
    {{1, 2}, {3, 4}, {5, 6}},
    {{7, 8}, {9, 10}, {11, 12}}
};
```

Flattened Order:
arr[0][0][0], arr[0][0][1], arr[0][1][0], ..., arr[1][2][1]

Diagram:
Layer 0 (arr[0]):
[[1, 2], [3, 4], [5, 6]]

Layer 1 (arr[1]):
[[7, 8], [9, 10], [11, 12]]

Memory Layout of n-D Arrays

An n-D array `arr[D1][D2]...[Dn]` is stored as:

- First, all elements of the innermost dimension for the first index of the outer dimensions.
- Then proceed to the next index of the outer dimensions.

Memory Address Calculation:

Address of `arr[i1][i2]...[in]` =

Base Address + (((((i1 * D2) + i2) * D3) + ... + in) * sizeof(data_type)

Flattened Representation:

arr[0][0]...[0], arr[0][0]...[1], ..., arr[D1-1][D2-1]...[Dn-1]

Array of Pointers

- It's an array of pointer variables
- Each element in the array can contain address of a variable of the declared type
- So, array of different sized arrays can be done
- Often used with strings, functions, or arrays.

```
double *buf[3];  // Array of 5 double pointers
double d0 = 8, d1[2] = {11, 12}, d2 = 10;
buf[0] = &d0; buf[1] = &d1[0]; buf[2] = &d2;

printf("%ld, %ld \n", sizeof(double*), sizeof(buf));
//prints 8, 24

printf("%p, %p, %p, %lf \n", buf, &buf[0], buf[1],
*buf[1]);
//prints 0x..b200, 0x..b200, 0x..b1f0, 11.000000

 .. what will be buf[1][1] ?
```

Pointer to an Array

- It's a pointer that can point to a whole array
- Declaration: data_type (*ptr)[size];
- It's has subtle difference from a normal array variable

```
int(*buf)[4]; // just another pointer type

printf("%ld, %ld, %ld, %ld \n", sizeof(int), sizeof(buf), sizeof(*buf), sizeof(*(buf+1)));
//prints 4, 8, 16, 16

printf("%p, %p \n", buf, buf+1);
//prints 0x..7e30, 0x..7e40 ← notice that the jump is 16 bytes
```

Accessing Elements using Pointer to an Array

- `ptr` holds the address of `arr`.
- Using `(*ptr)[index]`, the pointer accesses elements of the array.

```
int arr[5] = {1, 2, 3, 4, 5};
int (*ptr)[5] = &arr;  // 'ptr' is a pointer to an array
    of 5 integers
//Access
printf("%d\n", (*ptr)[2]);  // Output: 3
```

Differences Between Pointer to Array and Array of Pointers

Feature	Pointer to Array	Array of Pointers
Definition	Points to an entire array.	Array where each element is a pointer.
Declaration	`int (*ptr)[size];`	`int *arr[size];`
Use Case	Access a single array.	Access multiple variables or objects.
Memory Layout	Single pointer to a contiguous block.	Multiple pointers stored in the array.
Example Access	`(*ptr)[index]`	`*arr[index]`

Applications and Practice Problems

Applications

Pointer to Array:

- Useful in 2D arrays where you want to pass a row or a block of data to a function.
- Example: Accessing or manipulating matrix rows.

Array of Pointers:

- Commonly used with strings (array of `char*`), function pointers, and dynamic memory.
- Example: Storing multiple strings or function addresses.

Practice Problems

- Write a program that uses a **pointer to an array** to access and modify elements of a 2D array.
- Implement a program that stores and prints 5 strings using an **array of pointers**.
- Create a function that accepts a **pointer to an array** and calculates the sum of all its elements.
- Use an **array of pointers** to create a dynamic menu system where each menu option corresponds to a function pointer.

Introduction to Bitwise Operators

- Bitwise operators operate directly on binary representations of data
- Commonly used for low-level programming, data manipulation, and performance-critical tasks.

- Efficient performance for certain tasks (e.g., toggling settings, encryption).
- Essential for fields like embedded systems, network programming, and game development.

Decimal vs. Binary
Example: `5` in decimal is `0101` in binary, `7` in decimal is `0111`.

Binary Operations

- Bit positions from right to left: `2^0`, `2^1`, `2^2`, etc.
- Understanding binary helps visualize bitwise operations.

Bitwise AND Operator (&), Bitwise OR Operator (|)

Syntax: `result = a | b;`
Description: Performs OR on each pair of bits in two numbers.
Example:
`5 | 3 → 0101 | 0011 → 0111`
(Result: 7)

Use Case: Setting specific bits to 1.

Syntax: `result = a & b;`
Description: Performs AND on each pair of bits in two numbers.
Example:
`5 & 3 → 0101 & 0011 → 0001`
(Result: 1)

Use Case: Masking specific bits.

Bitwise XOR Operator (^), Bitwise NOT Operator (~)

Syntax: `result = a ^ b;`
Description: Performs XOR on each pair of bits, setting the result bit if bits differ.
Example:
`5 ^ 3` → `0101 ^ 0011` → `0110`
(Result: 6)

Use Case: Toggling bits.

Syntax: `result = ~a;`
Description: Flips all bits (0s become 1s and vice versa).
Example:
`~5` for an 8-bit integer → `0000 0101` becomes `1111 1010`
(Result: `-6` in two's complement)

Use Case: Inverting bits for binary operations, generating complements.

Bitwise Shift Operators (<<, >>)

- **Left Shift (<<)**: Moves bits to the left, adding 0s at the right end.
  Example: `5 << 1` → `0101 << 1` → `1010`
  (Result: `10`)

- **Right Shift (>>)**: Moves bits to the right.
  Example: `5 >> 1` → `0101 >> 1` → `0010`
  (Result: 2)

Use Cases:

Left shift: Efficient multiplication by powers of 2.

Right shift: Division by powers of 2.

Practical Applications of Bitwise Operators

Checking Power of 2:
(Only powers of 2 have one 1 bit).
`(n & (n - 1)) == 0`

Swapping Two Numbers
(without a third variable):

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

Setting a Bit at a Position:

```
n |= (1 << position);
```

Clearing a Bit at a Position:

```
n &= ~(1 << position);
```

Masking: Isolating bits in a number using &.

Example: Check if a number is odd using `n & 1`.

Setting/Clearing Bits: Using | and & with a mask.

Toggling Bits: Using XOR (^) to switch specific bits.

Shifts:

Left shifts for multiplying by powers of 2.
Right shifts for quickly dividing by powers of 2.

Bitwise Operator Practice Problems

- Write a function to check if a number is even or odd using bitwise operators.
- Write a function that toggles the 3rd bit of a given integer.
- Create a function to swap two integers without using a temporary variable.
- Implement a left rotation of an integer.

More problems to try:

Count the Number of 1 Bits in an Integer

Problem: Write a function to count how many bits are set to 1 in the binary representation of an integer.
Hint: Use a loop with `n = n & (n - 1);` to reduce the number of set bits in each iteration.

Find the Only Non-Duplicate Number in an Array

Problem: Given an array where every element appears twice except for one, find that single element.
Hint: XOR all elements. Pairs will cancel out, leaving the unique element.

Determine If Bits Are Alternating in a Number

Problem: Write a function to check if a number's bits alternate between 1 and 0 (e.g.,`101010...`).
Hint: XOR the number with itself shifted one bit, then check if the result is a power of 2.

Flip a Specific Bit

Problem: Write a function to flip a bit at a given position in a number.
Hint: Use XOR with a mask: `n ^= (1 << position);`

Clear All Bits from MSB through a Given Position

Problem: Write a function to clear all bits from the most significant bit (MSB) down to a specified bit position.
Hint: Use a mask with `~((1 << (position + 1)) - 1);`

Multiply a Number by 3.5 Using Bitwise Operators

Problem: Without using multiplication, create a function that multiplies a number by 3.5.
Hint: `n * 3.5` can be represented as `(n << 1) + n + (n >> 1);`

Swap Even and Odd Bits in an Integer

Problem: Write a function to swap even and odd bits in an integer.
Hint: Mask and shift: `((n & 0xAAAAAAAA) >> 1) | ((n & 0x55555555) << 1);`

Introduction to Variadic Functions

Variadic Functions

- Functions that accept a variable number of arguments
- Useful for situations where the number of arguments isn't fixed (e.g., `printf, fscanf`).

Common Use Cases

- Logging, formatting, mathematical functions (e.g., `sum` of multiple values).
- Simplifies code where multiple parameters may vary in count.

Syntax of Variadic Functions

Header: `#include<stdarg.h>`

- `va_list`: A type to hold information about variable arguments.
- `va_start`: Initializes a `va_list` object with the last known fixed argument.
- `va_arg`: Accesses each argument in the list.
- `va_end`: Cleans up the `va_list` when done.

Defining a Variadic Function:

```
#include <stdarg.h>
void functionName(int fixedArg, ...);
```

- The ellipsis (`...`) represents additional arguments after any fixed parameters.
- At least one fixed parameter must precede the ellipsis to know where variable arguments begin.

Example:

```
void printNumbers(int count, ...);
```

Basic Example of Variadic Function

How the Code Works

- **va_list**: Declares the list to hold arguments.
- **va_start(args, count)**: Initializes `args`, with `count` as the last known fixed parameter.
- **va_arg(args, type)**: Retrieves each argument in the specified type (e.g., `int`).
- **va_end(args)**: Cleans up after usage.

Note: All variable arguments must be accessed in the correct type (e.g., `int`, `double`).

Function to Print Any Number of Integers:

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

void printNumbers(int count, ...) {
    va_list args;
    va_start(args, count);
    for (int i = 0; i < count; i++) {
        int num = va_arg(args, int);
        printf("%d ", num);
    }
    va_end(args);
    printf("\n");
}

int main() {
    printNumbers(3, 10, 20, 30);
    printNumbers(5, 1, 2, 3, 4, 5);
    return 0;
}
```

Understanding Argument Types

Importance of Correct Types

- The type of each argument must be known at compile-time.
- Mismatch in expected type can lead to undefined behavior.
- Use the correct `type` in `va_arg(args, type)`.

Common Usage Patterns

- Functions with similar types across arguments (e.g., integers, floats).
- Handling strings requires checking for specific delimiters or count limits.

Practical Use Case: Sum of Numbers

```
int sum(int count, ...) {
    va_list args;
    va_start(args, count);
    int total = 0;
    for (int i = 0; i < count; i++) {
        total += va_arg(args, int);
    }
    va_end(args);
    return total;
}

int main() {
    printf("Sum: %d\n", sum(3, 10, 20, 30));  // Output: 60
    return 0;
}
```

Handling Variable Argument Types

Mixed Argument Types Example:

- Pass in a known type order (e.g., alternating `int` and `double` values)
- Use `enum` or a format string to specify types, similar to `printf`

```
void mixedArgs(int n, ...) {
    va_list args;
    va_start(args, n);
    int i = va_arg(args, int);
    double d = va_arg(args, double);
    va_end(args);
}
```

Common Pitfalls and Considerations

Practice Problems

- **Implement a Variadic `max` Function**
  Task: Find the maximum value from any number of integer arguments.
- **Variadic Logging Function**
  Task: Create a function `logMessage(level, ...)` where `level` is a string (e.g., `"INFO"`, `"ERROR"`) followed by a format string and corresponding arguments.
- **Product Calculator**
  Task: Write a function `multiply(count, ...)` that returns the product of `count` integers.
- **Dynamic Formatting**
  Task: Create a variadic function that formats and prints a string, handling `int`, `double`, and `char*`.

Type Safety: C does not verify argument types; incorrect types can lead to runtime errors.
Performance: Varargs functions are slower due to extra processing.
Alternatives: If the argument count or type varies significantly, consider passing an array or struct instead.

Examples to Avoid:
Incorrect type usage (e.g., `va_arg` as `double` for an `int` argument).
Excessive argument use without clear structure

Introduction to Standard I/O Streams in C

What are Standard I/O Streams?

- **stdin**, **stdout**, and **stderr** are predefined streams in C used for input and output.
- They handle data flow between the program and its environment (e.g., terminal, file).

Why Use Standard I/O?

- Simplifies input/output handling.
- Provides a consistent interface for reading/writing data across different platforms.

Overview of `stdin`, `stdout`, and `stderr`

stdin: Standard input stream.

Default: Keyboard input.

stdout: Standard output stream.

Default: Screen/console output.

stderr: Standard error stream.

Default: Screen output (separate from stdout).

Purpose: Separates normal output and error messages for easier debugging.

Standard Input (`stdin`)

Definition: Reads input data for the program.

Common Functions:

- `scanf()`: Reads formatted input.
- `getchar()`: Reads a single character.
- `fgets()`: Reads a string (more robust than `gets()`).

```
int num;
printf("Enter a number: ");
scanf("%d", &num);
printf("You entered: %d\n", num);
```

Usage Tip: Use `fgets()` with `sscanf()` to handle multiple inputs safely.

Standard Output (`stdout`)

Definition: Writes normal output data.

Common Functions:

- `printf()`: Prints formatted output.
- `putchar()`: Outputs a single character.
- `puts()`: Prints a string followed by a newline.

```
int num = 10;
printf("The value is %d\n", num);
```

Redirection: `stdout` can be redirected to files or other devices (e.g., `> output.txt`).

Standard Error (`stderr`)

Definition: Writes error messages or diagnostic output.

Why Use `stderr`?

- Keeps error messages separate from standard output.
- Error messages aren't redirected when `stdout` is redirected, making debugging easier.

```
if (error) {
    fprintf(stderr, "An error occurred!\n");
}
```

Tip: Use `stderr` for all critical error messages.

Using `fprintf` and Redirection with Standard Streams

Syntax:
`fprintf(stream, "format", args);`
- Works with any output stream (`stdout`, `stderr`, file pointers).

Examples:
- Normal output: `fprintf(stdout, "This is standard output.\n");`
- Error output: `fprintf(stderr, "This is an error message.\n");`

Benefit: Offers flexibility in specifying the output stream.

Command Line Redirection
- Redirect `stdout`: `program > output.txt`
- Redirect `stderr`: `program 2> error.txt`
- Redirect both: `program > output.txt 2> error.txt`

Why Redirect?
- Useful for logging, debugging, and separating normal output from errors.

Best Practices with Standard Streams

Example: Redirecting stdout and stderr

```
#include <stdio.h>

int main() {
    fprintf(stdout, "This is standard
      output.\n");
    fprintf(stderr, "This is an error
      message.\n");
    return 0;
}
```

Run with Redirection:

```
./program > output.txt 2> error.txt
```

Explanation:
`stdout` goes to `output.txt`, and `stderr` goes to `error.txt`.

Use `stdout` for Normal Program Output: Ensures separation between regular data and errors.

Use `stderr` for Errors and Debugging Information: Keeps error messages visible even if `stdout` is redirected.

Redirect Output for Logging and Analysis: Save outputs to files for later review or automated testing.

Practice Problems

Write a Program with Conditional Error Output: Write a program that asks for a number. Print an error to `stderr` if the input is negative.

Implement a Logging System: Create a function that logs messages to either `stdout` or `stderr` based on the message type.

Redirection Exercise: Run a program that outputs both `stdout` and `stderr` messages, and practice redirecting each to separate files.

More problems to try

Count Words Using `stdin` and `stdout`

Task: Write a program that reads lines of text from `stdin` and counts the number of words. Output the word count to `stdout`.
Hint: Use `fgets()` to read each line and `sscanf()` or `strtok()` to count words.

Error Logging Simulation

Task: Create a program that processes a list of numbers, printing each to `stdout` if it is positive. If a negative number is encountered, log an error message to `stderr` and continue.
Hint: Use `fprintf(stderr, ...)` for logging and skip further processing for negative values.

Redirection Test

Task: Write a program that outputs a series of messages to `stdout` and `stderr`. Run the program from the command line and try redirecting `stdout` to a file while keeping `stderr` in the console.
Hint: Use `./program > output.txt` and `2>` redirection options to separate streams.

Implement a Simple Logger

Task: Write a logging function `log_message` that accepts a log level (`INFO`, `WARNING`, `ERROR`) and a message. Direct `ERROR` logs to `stderr` and all others to `stdout`.
Hint: Use `fprintf(stdout, ...)` or `fprintf(stderr, ...)` based on the log level.

Dual Output Challenge

Task: Write a program that reads integer input from `stdin`, calculates the square of each integer, and outputs results to `stdout`. If a non-integer is entered, print an error to `stderr`.
Hint: Use `scanf()` to validate integers and `fprintf(stderr, ...)` for error reporting.

Introduction to enum in C

What is an enum?
A user-defined data type in C used to assign names to integral constants, making code more readable.

Syntax:

```
enum Day { SUNDAY, MONDAY,
TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY };
```

Default Values:
By default, enum values start from 0 and increment by 1 for each subsequent item.

Declaration:

```
enum Color { RED, GREEN, BLUE };
```

Usage:

```
enum Color color;
color = RED;
if (color == GREEN) {
    printf("The color is green!\n");
}
```

Accessing Values:

- Access by name, making the code more descriptive.

Assigning Custom Values in enum

Custom Values:
You can assign specific values to enum constants.

```
enum Status { SUCCESS = 0,
FAILURE = -1, PENDING = 2 };
```

Example:
`enum Status result = FAILURE;`
assigns `result` a value of -1.

Skipping Values:
Values can be skipped, creating gaps within the enum.

Why typedef with enum?
Simplifies the code by allowing the omission of enum keyword when declaring variables.

Example:

```
typedef enum { JAN, FEB, MAR } Month;
Month current_month = JAN;
```

Benefit:
Creates more readable and reusable code.

`enum` Usage and Applications

Example: Using `enum` in Switch Statements

```
enum Direction { UP, DOWN, LEFT, RIGHT };
enum Direction dir = LEFT;

switch (dir) {
    case UP:
        printf("Moving up\n");
        break;
    case DOWN:
        printf("Moving down\n");
        break;
    case LEFT:
        printf("Moving left\n");
        break;
    case RIGHT:
        printf("Moving right\n");
        break;
}
```

Purpose: Enhances readability and maintainability by using descriptive names instead of numbers.

Using `enum` as Bit Flags: Assign powers of two to `enum` values for bitwise operations.

```
enum Permissions { READ = 1, WRITE = 2,
    EXECUTE = 4 };
```

Example usage:

```
int perm = READ | EXECUTE;
if (perm & EXECUTE) {
    printf("Execute permission
granted.\n");
}
```

State Representation: Ideal for representing states in state machines or status codes.

Best Practices with `enum`

Name Your `enum` Values Clearly: Use descriptive names for each constant to improve code readability.

Avoid Overlapping Values: Avoid duplicate values within the same `enum` unless intentional (e.g., as bit flags).

Limit Scope: Use `typedef` to limit the scope of `enum`, so it doesn't conflict with other code.

Practice Problems:

Enum Month Example:
- Define an `enum` for months (January to December) with custom values for each.
- Print the numeric representation of a specific month.

Permissions Bit Flags:
- Create an `enum` with bit flags for file permissions (`READ`, `WRITE`, `EXECUTE`).
- Implement a function that checks if a specific permission is granted.

Error Codes with `enum`:
- Define an `enum` for error codes (`SUCCESS`, `ERROR_NOT_FOUND`, `ERROR_ACCESS_DENIED`).
- Write a function that returns these codes based on different conditions.

Introduction to Unions in C

What is a Union?
A data structure in C that allows different variables to share the same memory location.

*Defined with the **`union`** keyword.*

Memory Sharing: Only one member can hold a value at any time; all members occupy the same memory space.

Syntax and Example of a Union

Union Declaration:

```
union Data {
    int i;
    float f;
    char str[20];
};
```

Accessing Union Members:

```
union Data data;
data.i = 10; // Sets integer
data.f = 3.14; // Sets float, overwrites
integer
```

Memory Efficiency: Size of a union is the size of its largest member.

Applications of Unions

Memory Optimization:

- Save memory by sharing space among multiple data types.
- Useful in embedded systems where memory is limited.

Data Interpretation: Access the same memory as different types for byte-level manipulations.

Variant Data Types: Represent multiple types in data communication protocols.

Difference Between Struct and Union

Struct:

- Each member has its own memory.
- Total size is the sum of all members.

Union:

- All members share the same memory space.
- Total size is the size of the largest member.
- **Usage**: Unions are used when only one of the members will be used at any given time.

Usage examples of Unions

Simulating a Tagged Union:

```
enum Type { INT, FLOAT };
struct Variant {
    enum Type type;
    union {
        int i;
        float f;
    } data;
};
```

Usage:

Set `type` to indicate the active data type.
Access based on `type` to prevent misuse.

Interpreting Data in Different Formats:

```
union IntFloat {
    int i;
    float f;
};
union IntFloat data;
data.i = 1065353216;
printf("Interpreted as float: %f\n",
   data.f); // May output: 1.0
```

Usage:

Useful for low-level data interpretation, especially in networking and data serialization.

Best Practices with Unions

Use Only One Member at a Time: Avoid accessing multiple members simultaneously.

Use `enum` Tags for Safety: Use an `enum` to track the active member for safer data handling.

Avoid Pointers Inside Unions: Use fixed-size data types to avoid memory issues with pointers.

Practice Problems:

Simple Union with Multiple Data Types: Define a union `Data` that can store an `int`, `float`, and a `char[20]` string.
Write a program to: Assign a value to each member and print it, noting how values overwrite each other.
Demonstrate the shared memory behavior of union members.

Data Interpretation Union: Create a union `Interpret` with two members: `int num` and `float fnum`.
Assign an integer value to `num` and print the corresponding value of `fnum`.
Observe and analyze the behavior. (Hint: This demonstrates interpreting memory as different data types.)

Union for Byte-Level Access: Define a union `ByteAccess` with: An `int` and a `char[4]` array.
Write a program that: Assigns a value to the `int`. Prints each byte of the integer using the `char` array.
Application: This can be used to understand system endianness (byte order).

Tagged Union for Multiple Types: Create a `struct` with: An `enum Type` (e.g., `INT`, `FLOAT`). A union with `int` and `float` members.
Implement a function that: Takes the `enum` to determine the active union member. Assigns and prints the corresponding value based on the type.
Hint: This mimics a “tagged union” for handling different types in the same memory location.

Bitwise Manipulation Using Unions: Define a union `BitwiseData` containing: An `unsigned int` and a `struct` with four `unsigned char` fields.
Write a program to: Assign a value to the `unsigned int`. Access and print individual bytes using the `unsigned char` fields.
Objective: Understand how to manipulate individual bytes for bitwise operations.

Union for Embedded System Data Packing: Design a union `SensorData` to store: Different sensor data types, such as temperature (`float`), humidity (`int`), and status flags (`char`).
Implement a program that: Uses a union to save memory, only holding one sensor data type at a time.
Application: This is useful in embedded systems with limited memory.

What is Endianness?

Definition:
Endianness refers to the byte order used to represent multi-byte data types in memory.

Two Main Types:

- **Big-Endian**: Most significant byte is stored at the lowest memory address.
- **Little-Endian**: Least significant byte is stored at the lowest memory address.
- **Importance**: Determines how data is interpreted and affects data exchange between systems.

Understanding Big-Endian and Little-Endian

Big-Endian:

Bytes are stored from the most significant to the least significant.

Example:

For `0x12345678`:

```
Address:   0x00  0x01  0x02  0x03
Value:     0x12  0x34  0x56  0x78
```

Little-Endian:

Bytes are stored from the least significant to the most significant.

Example:

For `0x12345678`:

```
Address:   0x00  0x01  0x02  0x03
Value:     0x78  0x56  0x34  0x12
```

Why Endianness Matters

Data Exchange Between Systems:
Different systems use different byte orders, so endianness can cause misinterpretation of data if not handled properly.
Example: Sending binary data between a big-endian server and a little-endian client.

Network Protocols:
Protocols like IP standardize on **big-endian** (network byte order).
Data often needs to be converted to big-endian format for network communication.

Example:
Consider a 4-byte integer
`int x = 0x12345678;`.

- In a little-endian system:
  Stored as `[0x78][0x56][0x34][0x12]`.
- In a big-endian system:
  Stored as `[0x12][0x34][0x56][0x78]`.

Endianness and Structs:
When working with structs and unions in C, remember that individual fields might be accessed differently due to the system's endianness.

Practical Applications of Endianness

Data Serialization:
Convert data to a standard byte order before saving or sending it across networks.

Networking:
- Convert to **network byte order** (big-endian) for compatibility.
- Functions in `<arpa/inet.h>` (e.g., `htonl`, `ntohl`) are used to handle this conversion.

Binary File Handling:
Read and write binary files considering endianness to ensure consistency across platforms.

Practice Problems on Endianness

Endianness Detection:
Modify the detection code to store the result in a variable, then use it to decide data serialization.

Byte-Swap Function:
Write a function that swaps bytes in an integer (e.g., `0x12345678` to `0x78563412`).
Hint: Use bitwise operators.

Network Byte Order Conversion:
Use functions like `htonl` and `ntohl` to convert a `uint32_t` to network byte order and back, then print the result.

Memory Layout:
Given a `float` variable, use a `union` with a `char[4]` to print each byte individually on both big- and little-endian systems.

Introduction to Function Pointers

What Are Function Pointers?

- Function pointers are pointers that point to the address of a function in memory.
- They allow functions to be passed as arguments, stored in arrays, and returned from other functions.
- Enable dynamic function calls, which can make code more flexible and modular.

Why Use Function Pointers?

- Useful in callback functions, sorting, event-driven programming, and implementing state machines.
- Improve modularity by allowing you to switch between different functions at runtime.

Declaring Function Pointers

Syntax:

```
return_type (*pointer_name)(parameter_type(s)_list);
```

Example:

```
int (*func_ptr)(int, int);
// Pointer to a function taking two int arguments and returning int
```

Assigning a Function to a Pointer:

```
int my_function(int a, int b)
{ return 0;}

func_ptr = &my_function; //fashion 1
// assigning a value to the variable
 //You  can omit the '&' as shown below
func_ptr = my_function; //same effect as fashion 1
```

Calling a Function via a Pointer:

```
(*func_ptr)(arg1, arg2);
// Call using the function pointer
//  also works without *
func_ptr(arg1, arg2);
```

Using Function Pointers in Code

Explanation:
`operation` can point to either `add` or `subtract` and execute different functions at runtime.

```c
#include <stdio.h>

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int main() {
    int (*operation)(int, int);  // Function pointer declaration
    operation = add;
    printf("Add: %d\n", operation(5, 3));  // Output: 8

    operation = subtract;
    printf("Subtract: %d\n", operation(5, 3));  // Output: 2

    return 0;
}
```

Function Pointers as Function Parameters

Explanation:

`execute` can take any function that matches the `(int, int)` signature.

```
#include <stdio.h>

void execute(int (*operation)(int, int), int x, int y) {
    printf("Result: %d\n", operation(x, y));
}

int multiply(int a, int b) { return a * b; }

int main() {
    // Pass 'multiply' as a callback function
    execute(multiply, 4, 5);
    return 0;
}
```

Function Pointer Arrays

Using Arrays of Function Pointers:

- Enables selecting different functions at runtime using an index.
- Common in implementing simple menu systems or state-based logic.

Explanation: `operations` is an array of function pointers, allowing access to different functions using an index.

```c
#include <stdio.h>

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }

int main() {
    int (*operations[])(int, int) = {add, subtract, multiply};

    for (int i = 0; i < 3; i++) {
        printf("Result: %d\n", operations[i](6, 2));
    }

    return 0;
}
```

Applications of Function Pointers

Callback Functions:
Often used in libraries or APIs to perform custom actions when an event occurs. E.g., a sorting function might take a custom comparison function.

Event-Driven Programming:
Allows functions to be registered for specific events (e.g., GUI programming, signal handling).

State Machines:
Enables switching between different states in embedded systems, games, etc.

Custom Sorting:
Function pointers allow sorting based on different criteria by passing in custom comparator functions.

Practice Problems with Function Pointers

Custom Calculator:
Implement a calculator using function pointers, allowing the user to choose operations (add, subtract, multiply, divide) at runtime.

Callback Sorting:
Write a sorting function that takes a function pointer as a comparator to sort an array of integers in ascending or descending order.

Menu System:
Create a simple menu-based program where each menu option corresponds to a function pointer. Implement options such as "print," "calculate," and "exit."