Introduction to Computing

Advanced Things - II

Swapping Variables Without a Temp Variable

Explanation:
XORing two numbers twice effectively undoes the operation, making it possible to swap values without extra memory.

```
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    printf("After swapping: a = %d, b = %d\n", a, b);
    return 0;
}
```

Checking for Power of 2

Explanation:

If a number is a power of 2, its binary representation has exactly one 1 bit. x & (x - 1) clears the lowest 1 bit, so for powers of 2, the result is 0.

```c
#include <stdio.h>

int isPowerOfTwo(int x) {
    return x && !(x & (x - 1));
}

int main() {
    int n = 16;
    printf("%d is %sa power of 2\n", n,
      isPowerOfTwo(n) ? "" : "not ");
    return 0;
}
```

Printing "Hello" Without Using Semicolon

Explanation:
printf returns the number of characters printed, which is non-zero for "Hello, World!". So the if statement evaluates as true, and we don't need a semicolon.

```
#include <stdio.h>

int main() {
    if (printf("Hello, World!\n")) {}
    return 0;
}
```

Reverse an Integer Using Recursion

Explanation:

This function repeatedly divides the number by 10, printing the remainder, effectively reversing the digits.

```
#include <stdio.h>

void reverse(int n) {
    if (n == 0)
        return;
    printf("%d", n % 10);
    reverse(n / 10);
}

int main() {
    int num = 12345;
    reverse(num);
    printf("\n");
    return 0;
}
```

Detecting Endianness

Explanation:

By casting the address of x to a char*, we can check if the lowest byte (first byte in memory) is 1 (little-endian) or 0 (big-endian).

```
#include <stdio.h>

int main() {
    unsigned int x = 1;
    char *c = (char*)&x;
    if (*c)
        printf("Little Endian\n");
    else
        printf("Big Endian\n");
    return 0;
}
```

Using Comma Operator in For Loop

Explanation:
Both a and b are updated in each loop iteration, creating a visually balanced convergence of a and b.

```
#include <stdio.h>

int main() {
    int a = 1, b = 10;
    for (; a <= b; a++, b--) {
        printf("a = %d, b = %d\n", a, b);
    }
    return 0;
}
```

Nested Macros

Explanation:
Here, the SQUARE macro expands into its expression and is used within PRINT_AND_SQUARE to evaluate the square, allowing a combination of expansion and substitution.

```
#include <stdio.h>

#define SQUARE(x) ((x) * (x))
#define PRINT_AND_SQUARE(x) printf("Square of
    %d is %d\n", x, SQUARE(x))

int main() {
    int num = 5;
    PRINT_AND_SQUARE(num);
    return 0;
}
```

Self-Referencing printf

Explanation:
The first `printf` prints "Hello, World!" and returns the count of characters printed (12), which is then printed by the second `printf`.

```
#include <stdio.h>

int main() {
    int count = printf("Hello, World!");
    printf("\nNumber of characters printed: %d\n",
     count);
    return 0;
}
```

Sum of Digits Using Recursion

Explanation:

This function adds the last digit to the result of the recursive call with `n / 10`, eventually summing all digits.

```
#include <stdio.h>

int sumOfDigits(int n) {
    if (n == 0) return 0;
    return (n % 10) + sumOfDigits(n / 10);
}

int main() {
    int num = 1234;
    printf("Sum of digits: %d\n",
      sumOfDigits(num));
    return 0;
}
```

Fibonacci Sequence Using ternary operator

Explanation: This recursive function uses the ternary operator (`? :`) operator to stop the recursion when `n` is 0 or 1, otherwise summing the two previous Fibonacci numbers.

```c
#include <stdio.h>

int fibonacci(int n) {
    return (n <= 1) ? n : fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n = 5;
    printf("Fibonacci of %d is %d\n", n, fibonacci(n));
    return 0;
}
```

Modifying a String Constant

Explanation: Since `"Hello"` is stored in a read-only section, modifying it may cause a segmentation fault. Use `char str[] = "Hello";` to allow modification.

```
#include <stdio.h>

int main() {
    char *str = "Hello";
    str[0] = 'M';  // This may cause a segmentation fault
    printf("%s\n", str);
    return 0;
}
```

Declaring a Variable Inside for Loop

Explanation:

`j` is re-declared with each loop iteration, and its scope is limited to within the `for` block.

Any attempt to access `j` outside the loop would result in a compilation error.

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 3; i++) {
        int j = 10;
        j += i;
        printf("j = %d\n", j);
    }
    // printf("%d\n", j);  // Uncommenting this will cause
     an error
    return 0;
}
```

Accessing Array Out-of-Bounds (Undefined Behavior)

Explanation:
Accessing `arr[5]` goes beyond the array's allocated memory.
This behavior is undefined, so it may show garbage data, crash, or produce no visible effect, depending on the compiler and platform.

```
#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    printf("%d\n", arr[5]);  // Out of bounds
     access
    return 0;
}
```

Using ++ and -- Operators in Complex Expressions

Explanation:

Here, `a++` uses a’s original value (10), then increments it. The `++a` then increments a again before adding. So b is calculated as `10 + 12`, with a ending as 12.

```
#include <stdio.h>

int main() {
    int a = 10;
    int b = a++ + ++a;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

Using `goto` for Looping

Explanation: The `goto` keyword allows jumping to labeled sections of code, enabling a loop-like structure.

However, `goto` can make code difficult to read and maintain, so it's generally discouraged.

```
#include <stdio.h>

int main() {
    int i = 0;
start:
    if (i < 5) {
        printf("%d ", i);
        i++;
        goto start;
    }
    printf("\n");
    return 0;
}
```

Understanding Type Promotion

Explanation:
Applying ~ (bitwise NOT) to `char` promotes it to `int`, so b becomes `-61` due to sign extension, which may surprise if expecting `unsigned` behavior.

```
#include <stdio.h>

int main() {
    char a = 60;  // 0011 1100 in binary
    char b = ~a;  // Bitwise NOT of a
    printf("b = %d\n", b);  // Output may be
     unexpected due to promotion
    return 0;
}
```

Printing a String Without Loops or Recursion

Explanation:

The format specifier `%.ns` limits the number of characters printed. Each call to `printf` prints one more character of the string, allowing us to print the entire string one character at a time without loops or recursion.

```
#include <stdio.h>

int main() {
    char str[] = "Hello, World!";
    printf("%.1s\n%.2s\n%.3s\n%.4s\n%.5s\n%.6s\
     n%.7s\n%.8s\n%.9s\n%.10s\n%.11s\n%.12s\n
     %.13s\n",
           str, str, str, str, str, str, str, str, str, str, str,
     str, str);
    return 0;
}
```

Using Ternary Operator for Multi-Line Logic

Explanation:
This nested ternary operator evaluates each condition sequentially, allowing for a clean, inline decision structure similar to a `switch-case`.

```
#include <stdio.h>

int main() {
    int score = 85;
    char *grade = (score >= 90) ? "A" :
                  (score >= 80) ? "B" :
                  (score >= 70) ? "C" :
                  (score >= 60) ? "D" : "F";
    printf("Grade: %s\n", grade);
    return 0;
}
```

Static Variable in Recursive Function

Explanation:
The `static` variable `count` retains its value between function calls, so each time `countCalls` is called, `count` continues incrementing from its previous value.

```
#include <stdio.h>

void countCalls() {
    static int count = 0;  // Static variable retains its value
      across calls
    count++;
    printf("Function called %d times\n", count);
}

int main() {
    countCalls();
    countCalls();
    countCalls();
    return 0;
}
```

Multi-Dimensional Array as Function Parameter

Explanation:

The array dimensions must be specified in the function parameter list for multi-dimensional arrays, as C needs to know the inner dimension to calculate the row’s address offset.

```
#include <stdio.h>

void printMatrix(int matrix[3][3], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
    printMatrix(matrix, 3, 3);
    return 0;
}
```

Calculating Factorial Using Inline Assembly (for GCC)

Explanation:

The inline assembly code multiplies n into eax until n is decremented to 1, creating a factorial calculation directly at the machine level.

```
#include <stdio.h>

int factorial(int n) {
    int result;
    __asm__ (
        "movl $1, %%eax;"
        "1:"
        "cmpl $1, %1;"
        "jle 2f;"
        "imull %1, %%eax;"
        "decl %1;"
        "jmp 1b;"
        "2:"
        : "=a" (result)
        : "r" (n)
    );
    return result;
}

int main() {
    int n = 5;
    printf("Factorial of %d is %d\n", n, factorial(n));
    return 0;
}
```

Conditional Compilation with `#ifdef`

Explanation:

When `DEBUG` is defined, the code within the `#ifdef DEBUG` block is compiled.

Removing or undefining `DEBUG` will skip that block during compilation.

```c
#include <stdio.h>

#define DEBUG

int main() {
#ifdef DEBUG
    printf("Debug mode is ON\n");
#else
    printf("Debug mode is OFF\n");
#endif
    return 0;
}
```

Generating a Random Number Using Time Seed

Explanation: `srand` seeds the random number generator with the current time, making each run of the program generate different random numbers.

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

int main() {
    srand(time(0));  // Seed the random number
     generator with the current time
    printf("Random number: %d\n", rand());
    return 0;
}
```

Function Pointers

Explanation:

A function pointer `operation` is used to point to different functions (`add` and `subtract`) at runtime, allowing dynamic function calls.

```c
#include <stdio.h>

void add(int a, int b) {
    printf("Sum: %d\n", a + b);
}

void subtract(int a, int b) {
    printf("Difference: %d\n", a - b);
}

int main() {
    void (*operation)(int, int);
    operation = add;
    operation(5, 3);
    operation = subtract;
    operation(5, 3);
    return 0;
}
```

Variadic Functions Using stdarg.h

Explanation:

`va_list`, `va_start`, and `va_arg` macros are used to handle a variable number of arguments, making `printNumbers` flexible in the number of arguments it can accept.

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

Array of Function Pointers

Explanation:

`options` is an array of function pointers, enabling a menu structure where each option can call a specific function depending on user input.

```
#include <stdio.h>

void option1() {
    printf("Option 1 selected\n");
}

void option2() {
    printf("Option 2 selected\n");
}

int main() {
    void (*options[2])() = { option1, option2 };
    int choice;
    printf("Enter 0 or 1: ");
    scanf("%d", &choice);
    if (choice >= 0 && choice < 2) {
        options[choice]();
    } else {
        printf("Invalid choice\n");
    }
    return 0;
}
```

Useful Libraries in C – I

The `math.h` Library

Purpose: Provides mathematical functions for computations.

Common Functions:

- `pow()` - Calculates the power of a number.
- `sqrt()` - Computes the square root.
- `sin()`, `cos()`, `tan()` - Calculates trigonometric functions.
- `log()`, `log10()` - Computes logarithms.
- `fabs()` - Returns the absolute value of a float.

The `time.h` Library

Purpose: Provides functions for date and time manipulation.

Common Functions:

- `time()` - Returns the current calendar time.
- `clock()` - Returns the processor time consumed.
- `difftime()` - Calculates the difference between two times.
- `localtime()` - Converts time to the local time zone.
- `strftime()` - Formats time as a string.

Useful Libraries in C – II

The `string.h` Library

Purpose: Provides functions for manipulating C-style strings (character arrays).

Common Functions:

- `strlen()` - Returns the length of a string.
- `strcpy()`, `strncpy()` - Copies one string to another.
- `strcat()`, `strncat()` - Concatenates two strings.
- `strcmp()`, `strncmp()` - Compares two strings.
- `strchr()`, `strstr()` - Finds a character or substring within a string.

The `ctype.h` Library

Purpose: Provides character classification and conversion functions.

Common Functions:

- `isalpha()` - Checks if a character is alphabetic.
- `isdigit()` - Checks if a character is a digit.
- `isspace()` - Checks if a character is whitespace.
- `tolower()`, `toupper()` - Converts a character to lowercase or uppercase.
- `isalnum()` - Checks if a character is alphanumeric.

Useful Libraries in C – III

The `errno.h` Library

Purpose: Defines macros for reporting and handling error conditions.

Common Macros:

- `errno` - Stores the last error code.
- `EDOM`, `ERANGE` - Represent domain and range errors in math functions.
- `EIO`, `ENOMEM` - Represent input/output and memory errors.

Usage Example:

  - After a function call fails, check `errno` for the error code.

The `assert.h` Library

Purpose: Provides a way to add diagnostics to programs.

Common Functions:

- `assert()` - Evaluates a condition; if false, the program terminates and outputs an error message.

assert(pointer != NULL); // Ensures the pointer is not null

Useful Libraries in C – IV

The `stdlib.h` Library

Purpose: Provides general utility functions, including memory management, random numbers, and conversions.

Common Functions:

- `malloc()`, `free()` - Allocates and deallocates memory.
- `rand()`, `srand()` - Generates random numbers.
- `atoi()`, `atof()` - Converts strings to integers and floats.
- `exit()` - Terminates the program.
- `system()` - Executes system commands.

The `stdio.h` Library

Purpose: Provides input and output operations.

Common Functions:

- `printf()` - Outputs formatted data to stdout.
- `scanf()` - Reads formatted data from stdin.
- `fopen()`, `fclose()` - Opens and closes files.
- `fgets()` - Reads a line from a file.
- `fputs()` - Writes a line to a file.
- `fprintf()` - Writes formatted data to a file.

More Useful Libraries in C

`limits.h`:
Purpose: Defines constants for various limits of data types (e.g., `INT_MAX`, `CHAR_BIT`).
Useful for: Determining the range and capacity of built-in data types.
`float.h`:
Purpose: Defines constants for floating-point limits and properties (e.g., `FLT_MAX`, `DBL_EPSILON`).
Useful for: Precision control and setting thresholds for floating-point calculations.
`stdarg.h`:
Purpose: Provides macros to handle functions with a variable number of arguments.
Common macros: `va_start()`, `va_arg()`, `va_end()`.
Useful for: Functions like `printf()` that accept variable arguments.
`stdbool.h`:
Purpose: Defines the `bool` type and values `true` and `false` in C99 and later.
Useful for: Adding boolean types to C for better readability.
`signal.h`:
Purpose: Provides signal handling functions and macros (e.g., `SIGINT`, `signal()`).
Useful for: Managing and handling interruptions and exceptions like `Ctrl+C` in the terminal.
`stddef.h`:
Purpose: Defines common macros and types (e.g., `NULL`, `size_t`, `ptrdiff_t`).
Useful for: Standardizing types across platforms and aiding in pointer arithmetic.
`complex.h`:
Purpose: Provides complex number arithmetic functions.
Common functions: `cabs()`, `creal()`, `cimag()`.
Useful for: Mathematical operations involving complex numbers.

`locale.h`:
Purpose: Manages and customizes localization settings.
Common functions: `setlocale()`, `localeconv()`.
Useful for: Adapting programs to different regional settings, such as date formats and currency symbols.
`wchar.h`:
Purpose: Supports wide characters and wide strings, enabling Unicode text processing.
Common functions: `wprintf()`, `wcscpy()`, `wcslen()`.
Useful for: Applications that need to handle internationalization and multi-byte characters.
`wctype.h`:
Purpose: Provides character classification and conversion functions for wide characters.
Common functions: `iswalpha()`, `iswdigit()`, `towupper()`.
Useful for: Performing case conversion and checking properties of wide characters.
`inttypes.h`:
Purpose: Defines macros and types for formatted input/output of fixed-width integers.
Common macros: `PRId32`, `PRIu64` for portable integer formatting.
Useful for: Platform-independent handling of specific integer sizes.
`stdnoreturn.h`:
Purpose: Provides the `noreturn` specifier for functions that do not return (C11).
Common usage: `noreturn void fatal_error(const char* message);`
Useful for: Indicating that a function (e.g., an error handler) does not return, improving readability and optimizations.
`tgmath.h`:
Purpose: Provides generic macros for type-generic mathematical functions (C99).
Useful for: Performing math operations without needing to specify different functions for `float`, `double`, and `long double`.

Common Mistakes - I

Using = Instead of == in Conditions

Mistake:

```
if (a = 5) {
// Always true because assignment returns the value.
}
```

Fix: Use `==` for comparison:

```
if (a == 5) {
// Correct comparison
}
```

Dangling Pointer Access

Mistake: Accessing a pointer after freeing memory.

```
int *p = malloc(sizeof(int));
free(p);
*p = 10; // Undefined behavior
```

Fix: Set the pointer to `NULL` after freeing:

```
free(p);
p = NULL;
```

Off-by-One Errors

Mistake: Accessing beyond array bounds:

```
int arr[5];
for (int i = 0; i <= 5; i++) {
// Incorrect: i <= 5 leads to out-of-bounds access
arr[i] = i;
}
```

Fix: Use `<` instead of `<=` in loops for zero-indexed arrays:

```
for (int i = 0; i < 5; i++) {
 arr[i] = i;
 }
```

Buffer Overflow

Mistake: Writing beyond allocated memory.

```
char buffer[10];
strcpy(buffer, "This is a long string!"); // Overflows buffer
```

Fix: Use safer functions like `strncpy`:

```
strncpy(buffer, "This is a long string!", sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0'; // Ensure null-termination
```

Common Mistakes - II

Misunderstanding Array-Pointer Equivalence:

Mistake: Treating arrays and pointers as interchangeable without considering context.

```
int arr[5];
arr++; // Invalid: Arrays are not modifiable pointers
```

Fix: Understand that arrays decay to pointers in certain contexts but are not pointers themselves.

Ignoring Return Values

Mistake: Not checking return values from functions like `scanf` or `malloc`.

```
scanf("%d", &x); // What if input fails?
```

Fix: Always validate return values:

```
if (scanf("%d", &x) != 1) {
    printf("Input error!\n");
}
```

Integer Division

Mistake: Expecting a float result from integer division.

```
int a = 5, b = 2;
float c = a / b; // Result is 2, not 2.5
```

Fix: Cast to `float` for correct behavior:

```
float c = (float)a / b; // Result is 2.5
```

Confusing `sizeof` for Pointer Arrays

Mistake: Expecting `sizeof` to return the size of the entire array when used on a pointer.

```
int *arr = malloc(10 * sizeof(int));
printf("%zu\n", sizeof(arr)); // Prints size of pointer, not the array
```

Fix: Keep track of the allocated size separately or use a macro:

```
int arrSize = 10;
```

Common Mistakes - III

Implicit Type Conversions

Mistake: Losing precision when assigning a larger type to a smaller type.

```
int a = 300;
char b = a; // b gets truncated
```

Fix: Explicitly handle type conversion:

```
char b = (char)a; // Be aware of truncation
```

Forgotten Semicolon After Macro

Mistake: Omitting a semicolon when using macros.

```
#define MAX 100
int a = MAX // Missing semicolon causes a syntax error
```

Fix: Always follow macros with appropriate syntax:

```
int a = MAX;
```

Undefined Behavior with ++ and --

Mistake: Using ++ or -- multiple times in the same statement.

```
int a = 5;
int b = a++ + ++a; // Undefined behavior
```

Fix: Avoid modifying a variable more than once in a single statement:

```
int b = a++;
b += ++a;
```

String Literal Modification

Mistake: Modifying a string literal leads to undefined behavior.

```
char *str = "Hello";
str[0] = 'h'; // Undefined behavior
```

Fix: Use an array to store strings if modification is needed:

```
char str[] = "Hello";
str[0] = 'h'; // Valid
```

Common Mistakes - IV

Neglecting `const` with Strings

Mistake: Declaring a string as `char *` when it should be `const char *`.

```
char *str = "Hello"; // Should be const char *
```

Fix: Use `const` for immutable string literals:

```
const char *str = "Hello";
```

Array and Pointer Confusion in Function Arguments

Mistake: Passing a 2D array incorrectly to a function.

```
void func(int arr[][]) { } // Invalid
```

Fix: Specify all dimensions except the first:

```
void func(int arr[][3]) { }
```

Precedence Confusion

Mistake: Misinterpreting operator precedence in complex expressions.

```
int a = 10, b = 5, c = 2;
int result = a + b << c; // Misinterpreted as (a + b) << c
```

Fix: Use parentheses to clarify precedence:

```
int result = (a + b) << c; // Explicit and correct
```

Using the Wrong Format Specifier in `printf`/`scanf`

Mistake: Using mismatched specifiers leads to undefined behavior.

```
int x = 10;
printf("%f", x); // Undefined behavior
```

Fix: Always use the correct format specifier:

```
printf("%d", x); // Correct for integers
```

Common Mistakes - V

Uninitialized Variables

Mistake: Using variables without initializing them.

```
int x;
printf("%d\n", x); // Undefined value
```

Fix: Always initialize variables:

```
int x = 0;
```

Returning a Local Variable

Mistake: Returning the address of a local variable causes undefined behavior.

```
int* func() {
 int x = 10;
 return &x; // x goes out of scope after the function returns
}
```

Fix: Use dynamically allocated memory or pass the variable as an argument:

```
int* func() {
 int *ptr = malloc(sizeof(int));
 *ptr = 10;
 return ptr;
}
```

Infinite Loops Due to Floating-Point Comparison

Mistake: Floating-point values are imprecise and can cause infinite loops.

```
for (float x = 0.1; x != 1.0; x += 0.1) { // May never terminate
 printf("%f\n", x);
}
```

Fix: Use a threshold or integer-based iteration:

```
for (int i = 1; i <= 10; i++) {
float x = i * 0.1;
printf("%f\n", x);
}
```

Using a Comma Instead of a Semicolon in `for` Loops

Mistake: Writing a comma instead of a semicolon leads to syntax errors or unintended behavior.

```
for (int i = 0, i < 10, i++) { // Incorrect
 printf("%d\n", i);
 }
```

Fix: Use semicolons to separate loop components:

```
for (int i = 0; i < 10; i++) {
 printf("%d\n", i);
 }
```

Common Mistakes - VI

Null Pointer Dereference

Mistake: Dereferencing a `NULL` pointer causes a runtime crash.

```
int *p = NULL;
*p = 10; // Crash
```

Fix: Always check if the pointer is `NULL` before dereferencing:

```
if (p != NULL) {
*p = 10;
}
```

Forgetting `break` in `switch` Cases

Mistake: Omitting `break` causes fall-through.

```
switch (x) {
case 1:
printf("One\n");
case 2:
printf("Two\n"); // Falls through if x == 1
}
```

Fix: Use `break` explicitly unless intentional fall-through is required:

```
switch (x) {
case 1:
printf("One\n");
break;
case 2:
printf("Two\n");
break;
}
```

Implicitly Declaring Functions

Mistake: Calling a function without a prototype assumes it returns `int`.

```
printf("%d\n", myFunc()); // myFunc implicitly assumed to return int
```

Fix: Declare or include the function prototype before calling:

```
int myFunc();
printf("%d\n", myFunc());
```

Overwriting `malloc` Pointer Without Freeing

Mistake: Losing access to allocated memory causes a memory leak.

```
int *p = malloc(10 * sizeof(int));
p = malloc(20 * sizeof(int)); // Memory leak: original memory not freed
```

Fix: Free the old memory before reallocating:

```
free(p);
p = malloc(20 * sizeof(int));
```

Common Mistakes - VII

Incorrect Use of `sizeof` with Dynamically Allocated Arrays

Mistake: Using `sizeof` to determine the size of a dynamically allocated array.

```
int *arr = malloc(10 * sizeof(int));
printf("%zu\n", sizeof(arr)); // Prints size of pointer, not the array
```

Fix: Keep track of the array size manually:

```
int size = 10;
int *arr = malloc(size * sizeof(int));
```

Signed Integer Overflow

Mistake: Overflow of signed integers causes undefined behavior.

```
int x = INT_MAX;
x = x + 1; // Undefined behavior
```

Fix: Use unsigned integers if overflow is expected, or add checks:

```
unsigned int x = UINT_MAX;
x = x + 1; // Wraps around safely
```

Hardcoding Array Size

Mistake: Hardcoding array size makes the code brittle and error-prone.

```
int arr[10];
for (int i = 0; i < 10; i++) { // Fragile if array size changes
arr[i] = i;
}
```

Fix: Use `sizeof` to determine array size dynamically:

```
for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
arr[i] = i;
}
```

Forgetting to Include Required Header Files

Mistake: Using functions without including their header files leads to compilation errors.

```
printf("Hello"); // Missing #include <stdio.h>
```

Fix: Include the appropriate headers:

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

Common Mistakes - VIII

Modifying Loop Variable in the Loop Body

Mistake: Altering the loop variable can cause unexpected behavior.

```
for (int i = 0; i < 10; i++) {
i += 2; // Skips iterations
}
```

Fix: Avoid modifying the loop variable:

```
for (int i = 0; i < 10; i++) {
printf("%d\n", i);
}
```

Ignoring Warnings

Mistake: Overlooking compiler warnings can lead to bugs.

```
int x;
printf("%d\n", x); // Warning: Variable ‘x’ is used uninitialized
```

Fix: Treat warnings as errors and address them:

```
int x = 0;
printf("%d\n", x);
```

Using `gets()` Instead of `fgets()`

Mistake:

Using the unsafe `gets()` function, which has been removed from modern C standards because it doesn’t prevent buffer overflows.

```
char buffer[10];
gets(buffer); // Dangerous: can overflow the buffer
```

Fix:

Use `fgets()` to limit input size and prevent overflow:

```
char buffer[10];
fgets(buffer, sizeof(buffer), stdin);
```

Mixing Signed and Unsigned Integers

Mistake:

Comparing signed and unsigned integers may produce unexpected results due to implicit type conversion.

```
int x = -1;
unsigned int y = 1;
if (x < y) {
printf("x is less than y\n"); // Never prints because x is converted to unsigned
}
```

Fix:

Explicitly cast to avoid unintended behavior:

```
if ((unsigned int)x < y) { // Explicit conversion
printf("x is less than y\n");
}
```

Common Mistakes - IX

Forgetting `const` When Passing String Literals

Mistake:

Modifying a string literal, which is undefined behavior because string literals are stored in read-only memory.

```
void modify(char *str) {
str[0] = 'X'; // Undefined behavior
}
modify("Hello");
```

Fix:

Use `const` for string literals to enforce immutability:

```
void modify(const char *str) {
// str[0] = 'X'; // Error if uncommented
}
```

Array Index Out of Bounds

Mistake:

Accessing an array element outside its bounds leads to undefined behavior.

```
int arr[5] = {1, 2, 3, 4, 5};
printf("%d\n", arr[5]); // Out-of-bounds access
```

Fix:

Always ensure the index is within bounds:

```
for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
printf("%d\n", arr[i]);
}
```

Misusing the Comma Operator

Mistake:

Misinterpreting the behavior of the comma operator in expressions.

```
int x = (1, 2, 3); // Only assigns 3 to x
printf("%d\n", x); // Prints 3
```

Fix:

Understand that the comma operator evaluates all operands but returns only the last value:

```
int x = (1, 2, 3); // Intentional
```

Dangling Pointers

Mistake:

Using a pointer after the memory it points to has been freed.

```
int *ptr = malloc(sizeof(int));
*ptr = 42;
free(ptr);
printf("%d\n", *ptr); // Undefined behavior
```

Fix:

Set the pointer to `NULL` after freeing to avoid accidental access:

```
free(ptr);
ptr = NULL; // Prevents dangling pointer access
```