CS 141 - C Programming Language

CUET 24

June 25, 2025

Contents

1 Introduction to C Programming

C is a powerful, procedural programming language developed at Bell Labs by Dennis Ritchie in the early 1970s. It was initially designed for system programming, particularly for writing the Unix operating system. Its influence on modern programming languages is immense, with many languages (like C++, Java, C#, JavaScript, PHP, Python) borrowing syntax and concepts from C.

1.1 Why Learn C?

- **Performance**: C programs are fast due to low-level memory access and efficient compilation.
- **System Programming**: Ideal for operating systems, embedded systems, drivers, and compilers.
- **Foundation**: Understanding C provides a solid foundation for learning other programming languages and understanding how computers work at a lower level.
- **Memory Management**: Gives programmers direct control over memory, fostering deeper understanding of computer architecture.
- **Portability**: C code can often be compiled and run on different platforms with minimal changes.

1.2 The General Form of a Simple C Program

A basic C program typically includes:

1. **Preprocessor Directives**: Instructions for the preprocessor, like `#include` to include header files.
2. **Functions**: A collection of statements that perform a specific task. Every C program must have a `main` function, which is the entry point.
3. **Statements**: Instructions executed by the program. Each statement ends with a semicolon (`;`).
4. **Comments**: Explanatory notes within the code, ignored by the compiler. C supports single-line (`//`) and multi-line (`/* ... */`) comments.

```c
#include <stdio.h> // Preprocessor directive: include standard input/output library

int main(void) { // main function definition: entry point of the program
    /* This is a multi-line comment.
     * The printf function is used for output. */
    printf("Hello, World!\n"); // Statement: prints string to console, \n is newline
    return 0; // Statement: indicates successful execution to the operating system
}
```

Listing 1: A Simple "Hello, World!" Program

1.3 Compiling and Linking

The process of transforming C source code (`.c` files) into an executable program involves several stages:

1. **Preprocessing**: The preprocessor handles directives (e.g., `#include`, `#define`). It expands macros and includes header file contents into the source file. The output is often an `.i` file.
2. **Compilation**: The compiler translates the preprocessed code into assembly language. The output is often an `.s` file.
3. **Assembly**: The assembler converts assembly code into machine code (object file, e.g., `.o` or `.obj`). These files contain machine code but are not yet executable as they may have unresolved external references.
4. **Linking**: The linker combines the object file(s) with necessary library functions (e.g., `printf` from `stdio.h`'s compiled library) and other object files to create a final executable file. This resolves all external references.

An Integrated Development Environment (IDE) or a build system (like Make) typically automates these steps.

2 Data Types and Variables

Data types specify the type of data a variable can hold, determining the range of values and the operations that can be performed on it. Variables are named memory locations used to store data.

2.1 Identifiers and Keywords

- **Identifiers**: Names used for variables, functions, structures, etc. Rules: must start with a letter or underscore, followed by letters, digits, or underscores. Case-sensitive.
- **Keywords**: Reserved words in C that have special meaning (e.g., `int`, `if`, `while`, `return`). Cannot be used as identifiers.

2.2 Basic Data Types

C provides several fundamental data types, categorized as arithmetic types:

- **Integer Types**: Store whole numbers (positive, negative, or zero). Size can vary by system, but minimum sizes are guaranteed.
  - `char`: Typically 1 byte. Used for characters (ASCII values) or small integers. Can be `signed char` (default on some systems, -128 to 127) or `unsigned char` (0 to 255).
  - `short int` (`short`): At least 2 bytes (e.g., -32,768 to 32,767).
  - `int`: The most commonly used integer type. Typically 4 bytes (e.g., $\approx \pm 2 \times 10^9$).
  - `long int` (`long`): At least 4 bytes, often 8 bytes on modern systems.
  - `long long int` (`long long`): At least 8 bytes (C99 standard). For very large integers.

  Integer Constants: Written as decimal (123), octal (017), or hexadecimal (0xAF). Suffixes 'L' (long), 'LL' (long long), 'U' (unsigned) can specify type, e.g., '100ULL'. **Integer Overflow**: Occurs when an integer calculation produces a value outside the range of its type. For 'signed' types, this is undefined behavior. For 'unsigned' types, it wraps around (e.g., 'UCHAR$_{MAX}$+1'*becomes*'0').**Floating-Point Types** : *Storerealnumbers*(*withfractionalparts*).
-
  - `float`: Single-precision floating-point. Less precise, smaller range.
  - `double`: Double-precision floating-point (more common due to higher precision and range).
  - `long double`: Extended precision floating-point.

Floating Constants: Written with a decimal point (e.g., '3.14', '.5', '1.0e-3'). Suffixes 'f' (float) or 'L' (long double) can specify type, e.g., '3.14f'. Default is 'double'.

Boolean Type (C99): C99 introduced $_{B}$*oolandtypicallyusedvia*`bool`*from*`<stdbool.h>`.*Stores*`true`(1)*or*`false`(0).*InC*89,*i*
zerofortrue)*wereusedforlogicalvalues*.`Void Type` : `void`*indicatestheabsenceofatype.It'susedforfunctionsthatdon'treturn*

2.3 Variable Declaration and Initialization

Variables must be declared before use. Declaration specifies the variable's type and name.

```c
int age;            // Declaration: declares an integer variable named age
double salary = 50000.0; // Declaration and Initialization: declares a double and
    assigns a value
char initial = 'J'; // Declares a character and assigns a value (single quotes for
     char literals)
int count = 0, sum = 0; // Multiple declarations on one line
```

Listing 2: Variable Declaration and Initialization

Variables can be initialized during declaration or assigned a value later using the assignment operator (`=`). Uninitialized local (automatic) variables contain garbage values. Uninitialized global and static variables are implicitly initialized to zero.

2.4 Type Conversion (Casting)

C automatically performs implicit type conversions (known as "usual arithmetic conversions") in certain contexts (e.g., arithmetic operations, assignments to ensure operands have compatible types). For explicit conversion, **casting** is used:

```c
double result = (double) 5 / 2; // Explicit cast: 5 is treated as double, result is 2.5
int x = (int) 3.14;             // Explicit cast: x becomes 3 (truncation)
char c = (char) 97;             // Explicit cast: c becomes 'a'
```

Listing 3: Type Casting

Implicit conversions can sometimes lead to loss of data (e.g., 'double' to 'int' truncates, 'long' to 'int' can lose precision if 'long' is larger).

2.5 The `sizeof` Operator

The `sizeof` operator returns the size (in bytes) of a variable or a data type. Its result type is `size_t`, which is an unsigned integer type.

```c
printf("Size of int: %zu bytes\n", sizeof(int)); // %zu is for size_t
int arr[10];
printf("Size of arr: %zu bytes\n", sizeof(arr)); // Size of the entire array (10 * sizeof(int))
```

Listing 4: Using sizeof

2.6 Type Definitions (`typedef`)

The `typedef` keyword creates an alias (new name) for an existing data type. It improves code readability, makes complex declarations simpler, and aids in portability (e.g., defining types based on platform-specific sizes).

```c
typedef unsigned long long ULL; // ULL is now an alias for unsigned long long
ULL big_number = 1234567890ULL;

typedef char* String; // String is now an alias for char*
String name = "Alice"; // Equivalent to char* name = "Alice";
```

Listing 5: Using typedef

Note: `typedef` does not create a new type; it just creates a synonym.

3 Operators and Expressions

Operators are symbols that perform operations on values and variables. An expression is a combination of operators, operands, and function calls that evaluates to a single value.

3.1 Arithmetic Operators

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division (For integer operands, performs integer division, discarding remainder. For floating-point operands, performs floating-point division.)
- `%`: Modulo (remainder of integer division). Operands must be integers.

3.2 Assignment Operators

- **Simple Assignment (=)**: Assigns the value of the right operand to the left operand. The left operand must be an **lvalue** (something that can appear on the left side of an assignment, typically a modifiable memory location).
- **Compound Assignment Operators**: Combine an arithmetic/bitwise operation with assignment (e.g., `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^`, `<<=`, `>>=`).
- Example: `x += 5;` is equivalent to `x = x + 5;`

3.3 Increment and Decrement Operators

Used to increase or decrease a variable's value by 1. They operate only on lvalues.

- `++`: Increment (e.g., `i++` or `++i`)
- `--`: Decrement (e.g., `i--` or `--i`)

Prefix (`++i`): Increments the variable first, then uses its new value in the expression. **Postfix** (`i++`): Uses the variable's current value in the expression first, then increments it.

```
int a = 5, b = 5;
int x = ++a; // x is 6, a is 6
int y = b++; // y is 5, b is 6
printf("x=%d, a=%d, y=%d, b=%d\n", x, a, y, b);
```

Listing 6: Increment/Decrement Example

3.4 Relational Operators

Used for comparison; return 1 (true) or 0 (false) based on the comparison result.

- `==`: Equal to
- `!=`: Not equal to
- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal to
- `<=`: Less than or equal to

3.5 Logical Operators

Combine or modify boolean expressions (where 0 is false, non-zero is true); return 1 (true) or 0 (false). These operators short-circuit evaluation.

- `&&`: Logical AND (true if both operands are true). If the left operand is false, the right is not evaluated.
- `||`: Logical OR (true if at least one operand is true). If the left operand is true, the right is not evaluated.
- `!`: Logical NOT (inverts the boolean value). '!0' is 1, '!any$_n on_z ero$'is0.

3.6 Bitwise Operators

Perform operations on individual bits of integer operands.

- `&`: Bitwise AND
- `|`: Bitwise OR
- `^`: Bitwise XOR (Exclusive OR)
- `~`: Bitwise NOT (One's Complement)
- `<<`: Left Shift (shifts bits to the left, filling with zeros; equivalent to multiplying by powers of 2).
- `>>`: Right Shift (shifts bits to the right; equivalent to dividing by powers of 2. Behavior with signed negative numbers can be implementation-defined for the highest bit).

3.7 Conditional (Ternary) Operator

A shorthand for simple `if-else` statements. It's an operator that takes three operands. Syntax: `condition ? expression_if_true : expression_if_false;`

```c
int a = 10, b = 20;
int max = (a > b) ? a : b; // max will be 20
printf("Max is %d\n", max);
```

Listing 7: Conditional Operator Example

3.8 Comma Operator

The comma operator (`,`) evaluates its left operand, discards the result, then evaluates its right operand, and its result is the result of the right operand. It's often used in `for` loops.

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

Listing 8: Comma Operator Example

3.9 Operator Precedence and Associativity

- **Precedence**: Determines the order in which operators are evaluated (e.g., multiplication before addition). Higher precedence operators are evaluated first.
- **Associativity**: Determines the order of evaluation when operators have the same precedence (e.g., left-to-right or right-to-left).

Parentheses `()` can always be used to override default precedence and associativity, making expressions clearer and ensuring desired evaluation order.

3.10 Expression Statements

An expression statement consists of an expression followed by a semicolon (`;`). The expression is evaluated, and its result is discarded. The primary purpose is for side effects (e.g., assignment, function call, increment/decrement).

```c
int x = 5;       // Assignment expression statement
printf("Hello"); // Function call expression statement
x++;             // Increment expression statement
x + 10;          // Valid but useless expression statement (result discarded)
```

Listing 9: Expression Statements

4 Input/Output Operations

C uses a standard library (`<stdio.h>`) for input and output operations, which operate on streams.

4.1 Standard I/O Streams

- `stdin`: Standard input stream (usually keyboard).
- `stdout`: Standard output stream (usually console).
- `stderr`: Standard error stream (usually console, for error messages).

4.2 Formatted Output: `printf`

The `printf` function is used for formatted output to `stdout`. Syntax: `int printf(const char *format, ...);`

- **Conversion Specifications**: Determine how data is formatted and printed. They begin with `%`.
  - `%d`, `%i`: Signed decimal integer.
  - `%u`: Unsigned decimal integer.
  - `%o`: Unsigned octal integer.
  - `%x`, `%X`: Unsigned hexadecimal integer (lowercase/uppercase).
  - `%f`, `%F`: Floating-point number (decimal notation).
  - `%e`, `%E`: Floating-point number (scientific notation).
  - `%g`, `%G`: Floating-point number (uses `%f` or `%e`, whichever is shorter).
  - `%c`: Single character.
  - `%s`: String (null-terminated character array).
  - `%p`: Pointer address (implementation-defined format).
  - `%%`: Prints a literal `%`.
- **Format Modifiers**: Can be used for width, precision, flags (e.g., `%.2f` for 2 decimal places, `%5d` for minimum width 5).
- **Escape Sequences**: Special characters that control output (e.g., `\n` for newline, `\t` for tab, `\\` for backslash, `\"` for double quote).

```c
int num = 123;
double val = 123.4567;
char letter = 'A';
char name[] = "C Language";

printf("Integer: %d, Hex: %X\n", num, num);
printf("Floating-point: %.2f (2 decimal places)\n", val);
printf("Character: %c\n", letter);
printf("String: %s\n", name);
```

Listing 10: Using printf

4.3 Formatted Input: `scanf`

The `scanf` function is used for formatted input from `stdin`. Syntax: `int scanf(const char *format, ...);`

- Similar conversion specifications as `printf` (e.g., `%d`, `%f`, `%c`, `%s`).
- **Crucial**: Pass the **address** of the variable using the **address-of operator** (`&`) so `scanf` can modify its value. For arrays (like character arrays for strings), the array name itself acts as an address.
- `scanf` returns the number of items successfully read and assigned, or `EOF` on input failure/end-of-file.
- **How `scanf` Works**: It reads characters from the input stream, attempts to match them to the format string. Whitespace in the format string matches any amount of whitespace in the input. Non-whitespace characters in the format string must match exactly. Conversion specifications consume input until a mismatch or whitespace.
- **Ordinary Characters in Format Strings**: Non-whitespace characters in the format string (other than `%`) must match the input literally. E.g., `scanf("%d/%d", &month, &day);` expects input like `12/25`.

```c
int num;
char char_val;
double double_val;

```

```c
printf("Enter an integer, a character, and a double (separated by spaces): ");
// Reads an integer, then skips whitespace, then reads a character, then a double
int items_read = scanf("%d %c %lf", &num, &char_val, &double_val);
printf("Items read: %d\n", items_read);
printf("You entered: Num=%d, Char='%c', Double=%.2f\n", num, char_val, double_val)
    ;
```

Listing 11: Using scanf

4.4 Character I/O: `getchar`, `putchar`

These functions are typically faster for single character I/O as they are simpler.

- `int getchar(void)`: Reads a single character from `stdin`. Returns the character as an `int` (to accommodate `EOF`, which is typically -1), or `EOF` on end-of-file or error.
- `int putchar(int c)`: Writes a single character `c` to `stdout`. Returns the character written, or `EOF` on error.

4.5 Line I/O: `fgets`, `puts`

- `char *gets(char *str)`: Reads a line from `stdin` into `str` until newline or EOF. **Dangerous and deprecated**, as it doesn't check buffer size, leading to potential buffer overflow if input is longer than 'str' can hold. **Avoid using it.**
- `char *fgets(char *str, int size, FILE *stream)`: The **safe alternative** to `gets()`. Reads at most `size-1` characters from `stream` into `str` until a newline character is read, or EOF. It stores the newline character if read and always null-terminates the string.
- `int puts(const char *str)`: Writes a null-terminated string `str` to `stdout`, followed by a newline character. Returns a non-negative value on success, `EOF` on error.

5 Control Flow: Branching

Branching statements allow different code blocks to be executed based on conditions, providing decision-making capability.

5.1 The `if-else` Statement

Executes a block of code if a condition is true, optionally executing another block if false.

```c
int num = 10;
if (num > 0) { // Condition (num > 0) is true
    printf("Number is positive.\n");
} else if (num < 0) { // else-if for multiple conditions
    printf("Number is negative.\n");
} else { // No other conditions met
    printf("Number is zero.\n");
}

// Compound Statements: A block of code enclosed in {} acts as a single statement.
// If there's only one statement, {} are optional but recommended for clarity.
if (num % 2 == 0)
    printf("Even.\n"); // Single statement, no {}
else
    printf("Odd.\n");
```

Listing 12: if-else Statement

Cascaded `if` statements: A sequence of `if-else if-else` statements for handling multiple exclusive conditions. **The "Dangling else" Problem**: An `else` clause always associates with the nearest preceding `if` that is not yet matched with an `else`. Use curly braces {} to clarify intent.

```
if (condition1)
    if (condition2)
        statement1;
    else // This else associates with 'if (condition2)'
        statement2;

// To associate with 'if (condition1)':
if (condition1) {
    if (condition2)
        statement1;
} else { // This else associates with 'if (condition1)'
    statement2;
}
```

Listing 13: Dangling Else Example

5.2 The `switch` Statement

Used for multi-way branching based on the value of an integer expression.

- The controlling expression must evaluate to an integer type.
- Its value is compared against `case` labels, which must be integer constants.
- `break` statement is crucial to exit the switch block after a match, preventing "fall-through" to subsequent cases.
- `default` case is optional, executed if no other case matches. It can appear anywhere but typically at the end.

```
char grade = 'B';
switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break; // Exit switch after 'A'
    case 'B':
    case 'C': // Fall-through: if 'B' matches, prints "Well done." and then breaks
        printf("Well done.\n");
        break;
    case 'D':
        printf("You passed.\n");
        break;
    case 'F':
        printf("Better try again.\n");
        break;
    default:
        printf("Invalid grade.\n"); // Executed if no case matches
}
```

Listing 14: switch Statement

6 Control Flow: Loops

Looping statements allow a block of code to be executed repeatedly, based on a condition or a counter.

6.1 The `while` Statement

Executes a block of code repeatedly as long as a condition is true. The condition is checked **before** each iteration. If the condition is initially false, the loop body never executes.

```c
int i = 0;
while (i < 5) { // Condition: i < 5
    printf("%d ", i);
    i++;        // Update: increment i
} // Output: 0 1 2 3 4
printf("\n");

// Infinite loop example: if i++ was missing or condition always true
// while (1) { printf("Looping forever!\n"); }
```

Listing 15: while Loop

6.2 The `do-while` Statement

Similar to `while`, but the block of code is executed **at least once** because the condition is checked **after** the first iteration.

```c
int i = 0;
do {
    printf("%d ", i);
    i++;
} while (i < 5); // Condition checked after first iteration
// Output: 0 1 2 3 4
printf("\n");

int j = 10;
do {
    printf("This will print at least once: %d\n", j);
    j++;
} while (j < 5); // Condition is false, but loop body ran once
```

Listing 16: do-while Loop

6.3 The `for` Statement

A compact loop structure suitable when the number of iterations is known or can be easily determined, or when there's a clear initialization, condition, and update. Syntax: `for (initialization; condition; update) { // code }`

- `initialization`: Executed once at the beginning.
- `condition`: Evaluated before each iteration. If true, loop continues.
- `update`: Executed after each iteration of the loop body.

```c
for (int i = 0; i < 5; i++) { // C99 allows declaration in init part
    printf("%d ", i);
} // Output: 0 1 2 3 4
printf("\n");

// Omitting expressions:
// for (;;) { /* infinite loop */ }
// int k = 0;
// for (; k < 5; k++) { /* ... */ } // init omitted
```

Listing 17: for Loop

For Statement Idioms: Common patterns for ‘for‘ loops, such as counting up/down, iterating through arrays, etc. The flexibility allows for complex loop control.

6.4 Loop Control Statements

- `break`: Terminates the innermost loop (or `switch` statement) immediately and transfers control to the statement following the loop/switch.
- `continue`: Skips the rest of the current iteration of the innermost loop and proceeds to the next iteration. For `while/do-while`, the condition is re-evaluated. For `for`, the update expression is executed, then the condition is re-evaluated.
- `goto`: Transfers control unconditionally to a labeled statement within the same function. Generally discouraged as it can lead to ”spaghetti code” that is difficult to read, debug, and maintain. Use it sparingly, if at all (e.g., for breaking out of nested loops, or unified error handling in functions).

```c
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        continue; // Skip 3, go to next iteration
    }
    if (i == 7) {
        break;    // Exit loop when i is 7
    }
    printf("%d ", i);
} // Output: 0 1 2 4 5 6
printf("\n");

// goto example (minimal use case for error handling)
int fd = open("file.txt", O_RDONLY);
if (fd == -1) {
    goto error_open;
}
// ... do something with fd
close(fd);
return 0;

error_open:
    perror("Failed to open file");
    return 1;
```

Listing 18: Loop Control Statements

6.5 The Null Statement

A null statement is just a semicolon `;`. It does nothing. It’s often used when the loop’s work is done within the loop’s condition or update expressions, or as a placeholder.

```c
// Finds the first non-space character in a string
char *str = "  Hello";
int i = 0;
while (str[i] == ' ')
    i++; // Null statement: loop body is empty, all work in condition/update
printf("First non-space char: %c\n", str[i]);
```

Listing 19: Null Statement Example

7 Arrays

An array is a collection of elements of the same data type, stored in contiguous memory locations, accessed using an integer index.

7.1 One-Dimensional Arrays

- **Declaration**: `type arrayName[size];` where 'size' must be a compile-time constant for traditional arrays (or a variable for VLAs in C99).
- Elements are accessed using zero-based indexing: `arrayName[index]`.

```c
int numbers[5]; // Declares an array of 5 integers, indices 0-4
numbers[0] = 10; // Assigns value to the first element
numbers[4] = 50; // Assigns value to the last element

// Accessing out of bounds (e.g., numbers[5]) leads to undefined behavior.
for (int i = 0; i < 5; i++) {
    printf("numbers[%d] = %d\n", i, numbers[i]);
}
```

Listing 20: One-Dimensional Array Example

7.2 Array Initialization

Arrays can be initialized during declaration using an initializer list {}.

```c
int scores[3] = {90, 85, 95}; // Initialize all elements
int data[] = {1, 2, 3};       // Size is automatically determined by the initializer (3)
int partial[5] = {1, 2};      // Remaining elements (indices 2,3,4) are initialized to 0
int all_zeros[5] = {0};       // All elements initialized to 0
```

Listing 21: Array Initialization

(C99) **Designated Initializers**: Allow specifying elements by their index, useful for sparse arrays.

```c
int a[10] = {[2] = 5, [9] = 12}; // a[2] is 5, a[9] is 12, all other elements are 0
```

Listing 22: Designated Array Initializers (C99)

Constant Arrays: Arrays declared with 'const' cannot have their elements modified after initialization.

7.3 Multidimensional Arrays

Arrays of arrays, often used to represent tables or matrices. Declaration: `type arrayName[rowSize][colSize];`

```c
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}}; // 2 rows, 3 columns

// Accessing element at row 0, column 1
printf("Element at (0,1): %d\n", matrix[0][1]); // Output: 2

// Iterating through a 2D array
for (int i = 0; i < 2; i++) { // Loop through rows
    for (int j = 0; j < 3; j++) { // Loop through columns
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
```

```
}
```

Listing 23: Two-Dimensional Array Example

For a multi-dimensional array parameter in a function, all dimensions except the first must be specified for pointer arithmetic to work correctly. E.g., ‘void printMatrix(int arr[][3], int rows);‘.

7.4 Variable-Length Arrays (VLAs) (C99)

Allows array size to be specified by a variable whose value is determined at runtime, not just compile time. VLAs are allocated on the stack.

```c
int n;
printf("Enter array size: ");
scanf("%d", &n);
int vla_array[n]; // Size 'n' is determined at runtime, allocated on stack

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

Listing 24: Variable-Length Array (C99)

VLAs are deallocated automatically when their scope is exited. They can consume large amounts of stack memory and have security considerations (e.g., stack overflow if ‘n‘ is too large). The C11 standard made VLAs an optional feature.

8 Strings

In C, a string is a sequence of characters terminated by a null character (`'\0'`). Strings are typically stored in character arrays.

8.1 String Literals

A sequence of characters enclosed in double quotes (e.g., `"Hello World"`). The compiler automatically appends a null terminator. String literals are stored in read-only memory.

- **How String Literals Are Stored**: Typically in the read-only data segment of the program's memory. Multiple identical string literals may share the same memory location.
- **Operations on String Literals**: You cannot modify a string literal. Attempting to do so results in undefined behavior (often a segmentation fault).
- **String Literals vs Character Constants**: ‘char c = 'A';‘ stores a single character. ‘char* s = "A";‘ stores a pointer to a string of two characters ('A', '').

8.2 String Variables (Character Arrays and Pointers)

- **Character Array**: The most common way to store mutable strings. Make sure the array is large enough to hold the string plus the null terminator.

```c
    char greeting[20] = "Hello"; // Array of size 20, initialized. Holds up to
        19 characters + '\0'.
    // Can modify:
    greeting[0] = 'J'; // greeting is now "Jello"
```

Listing 25: String as Character Array

- **Character Pointer**: Can point to a string literal (read-only data).

```c
char *message = "World"; // Points to a string literal (read-only memory)
// Cannot modify the content pointed to by 'message' directly:
// message[0] = 'X'; // ERROR: Undefined behavior, usually crash.
// 'message' itself can be reassigned to point to another string literal.
```

Listing 26: String as Character Pointer

Character Arrays vs Character Pointers:

- Array: A block of memory allocated, often on the stack. The string content is mutable. The array name is a constant pointer to its first element.
- Pointer: A variable that holds an address. Can point to mutable memory (e.g., 'malloc''d memory) or immutable memory (string literals). The pointer itself is mutable (can be reassigned).

8.3 Reading and Writing Strings

- **Writing Strings**:
  - `printf("%s", str)`: Prints the string.
  - `puts(str)`: Prints the string followed by a newline.
- **Reading Strings**:
  - `scanf("%s", str)`: Reads a sequence of non-whitespace characters into `str`. Stops at the first whitespace. **Dangerous**, no buffer overflow protection.
  - `fgets(str, size, stdin)`: **Recommended** for safe string input from standard input.
  - **Reading Character by Character**: Using a loop with `getchar()` (or `fgetc` for files) offers fine-grained control and can handle arbitrary input lengths safely with dynamic allocation.

8.4 Using the C String Library (<string.h>)

Essential functions for string manipulation:

- `size_t strlen(const char *s)`: Returns the length of string `s` (number of characters before the null terminator).
- `char *strcpy(char *dest, const char *src)`: Copies `src` string to `dest`. **Dangerous**, does not check buffer size.
- `char *strncpy(char *dest, const char *src, size_t n)` : *Copies at most* `n` *characters from* `src` *to* `dest`. *Safer, but if 's terminated.* `char *strcat(char *dest, const char *src)` : *Concatenates* `src` *string to the end of* `dest`. Dangerous, *do*
- `char *strncat(char *dest, const char *src, size_t n)` : *Appends at most 'n' characters from 'src' to 'dest'.* `int strc` *Compares two strings lexicographically. Returns 0 if equal, negative if* `s1 < s2`, *positive if* `s1 > s2`.
- `int strncmp(const char *s1, const char *s2, size_t n)` : *Compares at most 'n' characters.* `char *strstr(const c` *Finds the first occurrence of 'needle' substring in 'haystack'. Returns pointer to found substring or* `NULL`.
- `char *strchr(const char *s, int c)`: Finds the first occurrence of character 'c' in string 's'. Returns pointer to found char or `NULL`.
- `char *strrchr(const char *s, int c)`: Finds the last occurrence of character 'c' in string 's'.

8.5 String Idioms

Common patterns for string manipulation without library functions, often used to understand underlying mechanics:

- **Searching for the End of a String**: 'while (*ptr != '') ptr++; '
- **Copying a String**: 'while ((*dest++ = *src++) != '');' (Efficient but compact).

9 Functions and Program Structure

Functions are reusable blocks of code designed to perform a specific task. They promote modularity, readability, and reusability.

9.1 Defining and Calling Functions

- **Definition**: Contains the function's code, including its return type, name, parameters, and body.
- **Call**: Executes the function. Arguments are passed to the parameters.

```c
// Function definition
int add(int a, int b) { // int a, int b are parameters
    return a + b;
}

int main() {
    int x = 10, y = 20;
    int sum = add(x, y); // Function call, x and y are arguments
    printf("Sum: %d\n", sum);
    return 0;
}
```

Listing 27: Function Definition and Call

9.2 Function Prototypes (Declarations)

A function prototype (or declaration) informs the compiler about a function's return type, name, and parameters (**signature**) *before* its definition. This allows functions to be called before they are defined, which is crucial for organization (e.g., 'main' calling other functions defined later) and for multi-file projects (prototypes in header files).

```c
// Function Prototype (usually placed at the top of a .c file or in a .h file)
int multiply(int num1, int num2); // Parameter names are optional in prototype

int main() {
    int p = multiply(5, 4); // Call before definition
    printf("Product: %d\n", p);
    return 0;
}

// Function Definition (can be after main, or in another .c file)
int multiply(int num1, int num2) {
    return num1 * num2;
}
```

Listing 28: Function Prototype

9.3 Function Arguments (Parameters)

- **Pass by Value**: The default mechanism. A copy of the argument's value is passed to the function's parameter. Changes made to the parameter inside the function do not affect the original variable in the calling function.
- **Pass by Reference**: Achieved by passing the **address** of a variable (using pointers). This allows the function to access and modify the original variable's value indirectly through its address. Used for:
  - Modifying multiple values in the calling function.
  - Passing large data structures efficiently (avoiding full copies).

- **Array Arguments**: When an array is passed to a function, it "decays" into a pointer to its first element. The function does not receive a copy of the entire array, but rather a pointer to its beginning. Changes made via this pointer *will* affect the original array. The size of the array is lost, so it's common to pass the array size as a separate argument.

```c
// arr[] is equivalent to *arr for parameters
void printArray(const int arr[], int size) { // 'const' protects array
    from modification
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
        // arr[i] = 0; // Error: cannot modify a const array
    }
    printf("\n");
}

// Variable-Length Array Parameters (C99):
void processMatrix(int rows, int cols, int matrix[rows][cols]) {
    // ...
}
```

Listing 29: Array Arguments and 'const'

- **Argument Conversions**: C performs automatic type promotion/conversion for arguments to match parameter types. E.g., 'float' arguments are promoted to 'double'.

9.4 The `return` Statement

A function uses the `return` statement to send a value back to the calling function and terminate its execution.

- If the function's return type is `void`, `return;` can be used without a value.
- If the function has a non-`void` return type, it must return a value of that type (or a type convertible to it).
- Execution resumes at the point immediately after the function call.

9.5 Program Termination

- `return 0;` in `main` indicates successful program execution. A non-zero value typically signals an error or abnormal termination. These values are returned to the operating system.
- `void exit(int status)`: From `<stdlib.h>`, terminates the program immediately from anywhere within the code. It performs clean-up (flushes buffers, closes files) before terminating. `EXIT_SUCCESS` (typically 0) and `EXIT_FAILURE` (typically 1) are macros defined in `<stdlib.h>` for portability.

```c
#include <stdlib.h>

void critical_error() {
    fprintf(stderr, "A critical error occurred!\n");
    exit(EXIT_FAILURE); // Terminate with error status
}

int main() {
    // ... some code
    if (/* some error condition */) {
        critical_error();
    }
    // ... more code (will not be reached if critical_error is called)
    return EXIT_SUCCESS;
}
```

Listing 30: Using exit()

10 Scope Rules and Storage Classes

10.1 Scope Rules

Scope determines the visibility and accessibility of an identifier (variable, function, etc.) within a program.

- **Block Scope (Local Variables)**: An identifier declared inside a block (code enclosed in {}) is visible only from its declaration point to the end of that block. This includes function parameters.
- **Function Scope**: Only labels (targets of `goto` statements) have function scope. They are visible throughout the entire function in which they are declared.
- **File Scope (Global Variables/Functions)**: An identifier declared outside any function is visible from its declaration point to the end of the source file. These are often called global variables.
- **Program Scope**: An identifier with file scope has program scope if it has external linkage (default for non-'static' global variables and functions). This means it can be accessed from other source files in the same program.

10.2 Storage Classes

Storage classes define the scope, lifetime, and linkage of variables and functions.

- `auto`:
  - Default for local variables.
  - **Lifetime**: Automatic storage duration (created on entry to the block where declared, destroyed on exit from that block). Memory is typically on the stack.
  - **Scope**: Block scope.
  - **Linkage**: None.
- `static`:
  - **For local variables**:
    - **Lifetime**: Static storage duration (allocated once at program startup, persists throughout program execution). Retains its value between function calls.
    - **Scope**: Block scope (still only visible within its declared block).
    - **Linkage**: None.
  - **For global variables/functions**:
    - **Lifetime**: Static storage duration.
    - **Scope**: File scope.
    - **Linkage**: Internal linkage (visible only within the current source file). Prevents name collisions in multi-file projects.
- `extern`:
  - Used to declare a variable or function that is defined elsewhere (in another source file or later in the current file).
  - **Lifetime**: Determined by its actual definition.
  - **Scope**: File or block scope (depending on where `extern` is used).
  - **Linkage**: External linkage (can be accessed from any source file in the program).
- `register`:
  - A hint to the compiler to store the variable in a CPU register for faster access. The compiler may ignore this hint if registers are unavailable or it deems it unnecessary.
  - **Lifetime**: Automatic storage duration.
  - **Scope**: Block scope.
  - **Linkage**: None.
  - Cannot take the address (`&`) of a 'register' variable.

Summary of Storage Classes and Properties:

Storage Class	Scope	Lifetime	Linkage
`auto` (default local)	Block	Automatic	None
`static` (local)	Block	Static	None
`static` (global/function)	File	Static	Internal
`extern`	File/Block	Static	External
`register`	Block	Automatic	None

10.3 Type Qualifiers

Modifiers that specify additional properties of a variable.

- `const`: Declares a variable as constant, meaning its value cannot be changed after initialization.

```c
const int MAX_VALUE = 100; // Cannot change MAX_VALUE
const int *ptr_to_const;   // Pointer to a constant integer (value pointed to is const)
int *const const_ptr = &MAX_VALUE; // Constant pointer to an integer (pointer itself is const)
const int *const all_const; // Constant pointer to a constant integer
```

Listing 31: Using const

- `volatile`: Hint to the compiler that a variable's value may be changed by external factors (e.g., hardware, another thread) outside the program's explicit control. Prevents the compiler from optimizing away reads/writes to this variable.
- `restrict` (C99): Used with pointers, it's a promise to the compiler that the pointer is the sole means of accessing the data it points to within its scope. Allows for more aggressive compiler optimizations.

11 Recursion

Recursion is a programming technique where a function calls itself, directly or indirectly, to solve a problem. It's often used for problems that can be broken down into smaller, similar subproblems.

11.1 Concept of Recursion

A recursive function must have:

- **Base Case**: A condition that stops the recursion, providing a direct solution for the simplest subproblem and preventing infinite calls. Without a base case, recursion leads to stack overflow.
- **Recursive Step**: The part where the function calls itself with a modified input, usually moving towards the base case (e.g., solving a smaller instance of the same problem).

11.2 Examples

```c
long long factorial(int n) {
    // Base case: factorial of 0 or 1 is 1
    if (n == 0 || n == 1) {
        return 1;
    } else {
        // Recursive step: n! = n * (n-1)!
        return n * factorial(n - 1);
    }
}

int main() {
    printf("Factorial of 5: %lld\n", factorial(5)); // Output: 120
    return 0;
```

```
}
```

Listing 32: Factorial using Recursion

Other classic recursive problems include Fibonacci sequence, Tower of Hanoi, tree traversals (e.g., in data structures like binary search trees), and algorithms like QuickSort and MergeSort.

12 Memory Management and Pointers

C provides mechanisms for both automatic and manual memory management, giving programmers fine-grained control over memory. Pointers are integral to this control.

12.1 Memory Areas

A typical program's memory layout includes:

- **Text/Code Segment**: Stores the executable instructions of the program. Read-only.
- **Data Segment**: Stores global and static variables.
  - Initialized data segment: For initialized global/static variables.
  - Uninitialized data segment (BSS): For uninitialized global/static variables (initialized to zero by OS).
- **Heap (Dynamic Memory)**: A large pool of free memory available for dynamic allocation during program execution. Memory requested from the heap must be explicitly allocated and deallocated by the programmer.
- **Stack**: Used for local variables (automatic variables), function parameters, and return addresses during function calls. Memory is allocated and deallocated automatically in a Last-In, First-Out (LIFO) manner as functions are called and return.

12.2 Memory Address and Pointers

A pointer is a variable that stores the memory address of another variable. Pointers are fundamental to C programming, enabling dynamic memory management, efficient array manipulation, and advanced data structures.

12.2.1 Introduction to Pointers

Every variable occupies a memory location, which has a unique address. Pointers store these addresses.

- Declaration: `type *pointerName;` (e.g., `int *ptr;`). The '*' indicates that 'ptr' is a pointer to an 'int'.

12.2.2 Address-of Operator (&)

The & operator returns the memory address of its operand.

```c
int x = 10;
int *ptr_x; // Declare ptr_x as a pointer to an integer
ptr_x = &x; // ptr_x now stores the memory address of variable x
printf("Value of x: %d\n", x);
printf("Address of x: %p\n", (void*)ptr_x); // %p for printing addresses
```

Listing 33: Address-of Operator

12.2.3 Dereference Operator (*)

The * operator (when used with a pointer variable, not in declaration) accesses the value stored at the memory address pointed to by the pointer.

```c
int x = 10;
int *ptr_x = &x;
printf("Value at address %p: %d\n", (void*)ptr_x, *ptr_x); // Output: Value at address ...: 10
*ptr_x = 20; // Changes the value of x to 20 through the pointer
printf("New value of x: %d\n", x); // Output: New value of x: 20
```

Listing 34: Dereference Operator

12.2.4 Pointer Assignment

A pointer variable can be assigned the address of another variable (of compatible type), or the value of another pointer variable (of compatible type). Assigning a `NULL` pointer indicates it points to nothing.

```c
int a = 10, b = 20;
int *p = &a; // p points to a
int *q;
q = p;       // q now also points to a
printf("*q: %d\n", *q); // Output: 10
p = &b;      // p now points to b
printf("*p: %d, *q: %d\n", *p, *q); // Output: *p: 20, *q: 10
```

Listing 35: Pointer Assignment

12.2.5 Pointers and Arrays

- In C, an array's name often behaves as a pointer to its first element. 'int arr[5];' 'arr' is equivalent to 'arr[0]'.
- Pointer arithmetic works naturally with arrays because elements are stored contiguously.
- 'arr[i]' is equivalent to '*(arr + i)'.

```c
int arr[] = {10, 20, 30};
int *ptr = arr; // ptr points to arr[0]

printf("First element: %d\n", *ptr); // Output: 10
printf("Second element: %d\n", *(ptr + 1)); // Output: 20 (moves sizeof(int) bytes)
printf("Third element: %d\n", ptr[2]);      // Using array-like subscripting with pointer

// Reversing an array using pointers
void reverse_array(int *a, int n) {
    int *left = a;
    int *right = a + n - 1;
    while (left < right) {
        int temp = *left;
        *left = *right;
        *right = temp;
        left++;
        right--;
    }
}
```

Listing 36: Pointers and Arrays

12.2.6 Pointers and Multidimensional Arrays

A 2D array like ‘int matrix[2][3]‘ is an array of 2 elements, where each element is an array of 3 integers.

- ‘matrix‘ itself is a pointer to the first row (‘matrix[0]‘).
- ‘matrix + i‘ points to the ‘i‘-th row (‘matrix[i]‘).
- ‘*(matrix + i)‘ or ‘matrix[i]‘ refers to the ‘i‘-th row array.
- ‘*(*(matrix + i) + j)‘ or ‘matrix[i][j]‘ accesses the element at ‘[i][j]‘.

12.2.7 Pointers as Function Arguments and Return Values

- **Pointers as Arguments**: Enables ”pass by reference” to modify variables in the calling function, or to pass large data structures efficiently. (**See** 9.3)
- **Pointers as Return Values**: A function can return a pointer. This is often used to return dynamically allocated memory or a pointer to a specific part of a data structure. **Caution**: Never return a pointer to a local (automatic) variable, as it will be deallocated upon function exit, leading to a dangling pointer.

```c
#include <stdlib.h> // For malloc

int* createDynamicInt() {
    int* newInt = (int*) malloc(sizeof(int));
    if (newInt == NULL) {
        return NULL; // Handle allocation failure
    }
    *newInt = 100;
    return newInt; // Return pointer to heap-allocated memory
}

int main() {
    int *val_ptr = createDynamicInt();
    if (val_ptr != NULL) {
        printf("Dynamically created value: %d\n", *val_ptr);
        free(val_ptr); // Crucial to free
    }
    return 0;
}
```

Listing 37: Pointers as Return Values (Safe Example)

12.2.8 Pointers to Pointers

A pointer variable can store the address of another pointer variable. This is common for working with arrays of pointers or modifying pointer variables themselves within functions. Declaration: `type **pointerToPointerName;`

```c
int x = 10;
int *ptr_x = &x;      // ptr_x stores address of x
int **pptr_x = &ptr_x; // pptr_x stores address of ptr_x

printf("Value of x: %d\n", x);           // Direct access
printf("Value of x via ptr_x: %d\n", *ptr_x); // One dereference
printf("Value of x via pptr_x: %d\n", **pptr_x); // Two dereferences
```

Listing 38: Pointers to Pointers

12.2.9 Pointers to Functions

A pointer can store the memory address of a function. This allows functions to be passed as arguments to other functions (callback functions) or stored in data structures. Declaration: `returnType (*pointerName)(parameterList)`

```c
int sum(int a, int b) { return a + b; }
int product(int a, int b) { return a * b; }

int main() {
    int (*op)(int, int); // Declare a function pointer 'op'
    op = sum; // Assign address of sum function to op
    printf("Sum: %d\n", op(5, 3)); // Calls sum(5,3) -> 8
    op = product; // Assign address of product function to op
    printf("Product: %d\n", op(5, 3)); // Calls product(5,3) -> 15
    return 0;
}
```

Listing 39: Pointers to Functions

The standard library's 'qsort' function (19.2.4) is a prime example of using function pointers.

12.3 Dynamic Memory Management

Functions from `<stdlib.h>` are used for manual memory management on the heap. This memory persists until explicitly 'free'd or the program terminates.

- `void *malloc(size_t size)`: Allocates a block of 'size' bytes. Returns a 'void*' pointer to the beginning of the allocated block, or 'NULL' if allocation fails. The allocated memory is **uninitialized** (contains garbage values).
- `void *calloc(size_t num, size_t size)`: Allocates memory for an array of 'num' elements, each of 'size' bytes. All bytes in the allocated block are **initialized to zero**.

- `void *realloc(void *ptr, size_t new_size)`: Changes the size of the memory block pointed to by 'ptr' to 'new$_s$*ize*'. *It may move the block to a new location. Returns a pointer to the new block, or 'NULL' if reallocation fails (in which case* *Deallocates the memory block pointed to by 'ptr', making it available for reuse. It's crucial to free allocated memory to prevent memo* *NULL;'.*

```c
#include <stdio.h>
#include <stdlib.h> // For malloc, calloc, realloc, free

int main() {
    int *arr = NULL; // Initialize pointer to NULL
    int n = 5;

    // Allocate memory for 5 integers using malloc
    arr = (int *) malloc(n * sizeof(int)); // Cast is good practice, though not
        strictly required in C

    if (arr == NULL) {
        printf("Memory allocation failed (malloc)!\n");
        return 1;
    }
    printf("Malloc'd array (uninitialized):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]); // Contains garbage values
    }
    printf("\n");
    free(arr); // Deallocate
    arr = NULL;

    // Allocate memory using calloc (initialized to zero)
    arr = (int *) calloc(n, sizeof(int));
    if (arr == NULL) { /* handle error */ return 1; }
    printf("Calloc'd array (initialized to 0):\n");
```

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

    // Reallocate to a larger size
    arr = (int *) realloc(arr, (n + 5) * sizeof(int));
    if (arr == NULL) { /* handle error */ return 1; }
    printf("Realloc'd array:\n");
    for (int i = 0; i < n + 5; i++) {
        printf("%d ", arr[i]); // Original 5 elements, then new uninitialized ones
    }
    printf("\n");

    free(arr); // Deallocate the final block
    arr = NULL; // Prevent dangling pointer
    return 0;
}
```

Listing 40: Dynamic Memory Allocation Example

12.4 Common Memory Errors

- **Memory Leak**: Failing to `free` dynamically allocated memory when it's no longer needed, leading to gradual depletion of available memory, potentially crashing long-running programs.
- **Dangling Pointer**: A pointer that still holds the address of a memory location that has been freed or deallocated. Dereferencing a dangling pointer leads to undefined behavior. Always set pointers to `NULL` after freeing the memory they point to.
- **Double Free**: Attempting to `free` the same memory block more than once. Leads to undefined behavior (e.g., heap corruption, crashes).
- **Use-After-Free**: Accessing memory after it has been freed. Similar to a dangling pointer issue.
- **Buffer Overflow**: Writing past the allocated size of a buffer (array), corrupting adjacent memory. This is a common security vulnerability.
- **Uninitialized Read**: Reading from memory that has been allocated but not initialized. Leads to undefined values.

12.5 Linked Lists

A fundamental dynamic data structure where elements (nodes) are not stored contiguously. Each node contains data and a pointer to the next node in the sequence. Useful for collections where elements are frequently added or removed.

```c
struct Node {
    int data;
    struct Node *next; // Pointer to the next node
};

// Creating a new node (example)
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    if (newNode == NULL) return NULL;
    newNode->data = value; // Access member of struct via pointer using '->'
    newNode->next = NULL;
    return newNode;
}

// Inserting a node at the beginning of a list
```

```c
struct Node* insertAtBeginning(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    if (newNode == NULL) return head;
    newNode->next = head;
    return newNode; // New head of the list
}
```

Listing 41: Basic Linked List Node

13 User-Defined Data Types: Structures, Unions, and Enumerations

C allows programmers to define their own complex data types using structures, unions, and enumerations.

13.1 Structures (`struct`)

A structure is a collection of variables (members) of different data types, grouped together under a single name. Members are stored contiguously in memory.

- **Declaration of a Structure Tag**: Defines the blueprint for the structure.

```c
    struct Person { // 'Person' is the tag
        char name[50];
        int age;
        float height;
    }; // Semicolon is crucial
```

Listing 42: Structure Tag Declaration

- **Declaring and Initializing Variables**:

```c
    struct Person p1 = {"Alice", 30, 1.65}; // Initialized in order
    struct Person p2; // Declared, members are uninitialized
    // (C99) Designated Initializers: Can initialize members by name, order
        doesn't matter.
    struct Person p3 = {.age = 25, .name = "Bob", .height = 1.80};
```

Listing 43: Structure Variable Declaration and Init

- **Accessing Members**:
  - **Dot operator (`.`)**: Used for structure variables. E.g., `p1.age`.
  - **Arrow operator (`->`)**: Used for pointers to structures. E.g., `ptr_p1->name` (equivalent to `(*ptr_p1).name`).

```c
    p2.age = 25;
    struct Person *ptr_p1 = &p1;
    printf("Name: %s, Age: %d\n", ptr_p1->name, p1.age);
```

Listing 44: Accessing Structure Members

- **Operations on Structures**: Structures can be assigned, passed to functions by value or pointer, and returned by value.

```c
    struct Person p4 = p1; // p4 is a copy of p1
```

Listing 45: Structure Assignment

- **Structures as Arguments and Return Values**:
  - By Value: A copy of the entire structure is passed. Safe, but inefficient for large structures.

  - By Pointer: The address of the structure is passed. Efficient, allows modification of the original.
- **Compound Literals (C99)**: Create unnamed, temporary objects of a specified type. Useful for passing structure values directly to functions.

```
    void printPoint(struct Point p) {
        printf("Point: (%d, %d)\n", p.x, p.y);
    }
    struct Point { int x, y; }; // Assuming Point is defined
    // In main:
    printPoint((struct Point){.x = 10, .y = 20}); // Creates an anonymous
        struct
```

Listing 46: Structure Compound Literal

13.2 Nested Structures and Arrays of Structures

- **Nested Structures**: A structure can contain another structure as a member.

```
    struct Date { int day, month, year; };
    struct Student {
        char name[50];
        struct Date dob; // Nested structure member
        int student_id;
    };
    struct Student s1;
    s1.dob.year = 2000; // Accessing nested member
```

Listing 47: Nested Structure Example

- **Arrays of Structures**: An array where each element is a structure.

```
    struct Person people[10]; // Array of 10 Person structures
    // Initializing an array of structures:
    struct Person students[] = {
        {"Alice", 20, 1.60},
        {"Bob", 22, 1.75}
    };
    printf("First student's age: %d\n", students[0].age);
```

Listing 48: Array of Structures Example

13.3 Unions (`union`)

A union is similar to a structure, but all its members share the **same memory location**. Only one member can hold a value at any given time. Unions are used to save memory when only one of several possible data types is needed for a single variable, or for "type punning" (interpreting the same bits in different ways).

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

int main() {
    union Data data;
    printf("Size of union Data: %zu bytes (size of its largest member: str[20])\n"
        , sizeof(union Data));

    data.i = 10;
```

```c
    printf("data.i: %d\n", data.i); // Output: 10
    // After next assignment, data.i is invalid
    data.f = 220.5;
    printf("data.f: %f\n", data.f); // Output: 220.500000
    // Reading data.i here would result in garbage/undefined behavior.

    // A common pattern is to use a "tag field" in a structure to indicate which
        union member is active.
    return 0;
}
```

Listing 49: Union Example

Difference with Structures: Structures allocate memory for *all* their members simultaneously (sum of sizes + padding). Unions allocate memory only for their *largest* member.

13.4 Enumerations (`enum`)

An enumeration is a user-defined data type that consists of a set of named integer constants (enumerators). It improves readability by allowing symbolic names instead of magic numbers.

- **Default Values**: By default, enumerators start from 0 and each subsequent enumerator increments by 1.
- **Explicit Values**: Values can be explicitly assigned. If a value is assigned, subsequent unassigned enumerators continue incrementing from that value.
- **Enumerations as Integers**: Enumeration constants are compatible with ‘int‘ type.

```c
enum Weekday {
    MONDAY = 1, // MONDAY is 1
    TUESDAY,    // TUESDAY is 2
    WEDNESDAY = 5,  // WEDNESDAY is 5
    THURSDAY,   // THURSDAY is 6
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int main() {
    enum Weekday today = WEDNESDAY;
    if (today == WEDNESDAY) {
        printf("It's hump day!\n");
    }
    printf("Today's number: %d\n", today); // Output: 5

    enum Color { RED, GREEN, BLUE }; // RED=0, GREEN=1, BLUE=2
    enum Color myColor = GREEN;
    printf("My color index: %d\n", myColor); // Output: 1
    return 0;
}
```

Listing 50: Enumeration Example

14 File Input/Output

C provides a robust set of functions in `<stdio.h>` for file handling, managing data persistence beyond program execution.

14.1 Streams and File Pointers

File I/O in C uses the concept of streams. A stream is an abstraction for a sequence of data, flowing between the program and an I/O device (like a disk file, keyboard, or screen).

- A `FILE*` pointer (file handle) is used to manage communication with a specific file. It points to a structure (defined in `stdio.h`) that holds information about the file (buffer, current position, error status).
- **Text Files vs Binary Files**:
  - **Text files**: Interpret characters based on encoding (e.g., ASCII). Line endings (“) might be translated by the system (e.g., to ‘ ° ‘ on Windows).
  - **Binary files**: No character translation. Data is read/written exactly as it is in memory. Suitable for non-textual data (images, executables).

14.2 File Operations: Opening and Closing

- `FILE *fopen(const char *filename, const char *mode)`: Opens a file. Returns a `FILE*` pointer on success, `NULL` on failure. Always check for ‘NULL‘.
- **Modes**: Specify how the file will be accessed. Suffix ’b’ for binary mode.
  - `"r"`: Read mode (file must exist).
  - `"w"`: Write mode (creates new file or truncates existing to zero length).
  - `"a"`: Append mode (creates new file or appends data to end of existing).
  - `"r+"`: Read and update (file must exist).
  - `"w+"`: Write and update (creates new or truncates existing).
  - `"a+"`: Append and update (creates new or appends to existing).
  - Example binary modes: `"rb"`, `"wb"`, `"ab"`.
- `int fclose(FILE *stream)`: Closes an open file stream, flushing any buffered output. Returns 0 on success, `EOF` on error. It’s crucial to close files to prevent data loss and free resources.

```c
#include <stdio.h> // For FILE, fopen, fclose, fprintf, perror

int main() {
    FILE *fp;
    fp = fopen("output.txt", "w"); // Open for writing (creates or truncates)

    if (fp == NULL) { // Always check if fopen was successful
        perror("Error opening file output.txt"); // Prints a system error message
        return 1; // Indicate error
    }

    fprintf(fp, "This is a line written to the file.\n"); // Write formatted
        string
    fputs("Another line.\n", fp); // Write a string

    fclose(fp); // Close the file
    printf("Data written to output.txt\n");

    // Reopen for reading
    fp = fopen("output.txt", "r");
    if (fp == NULL) {
        perror("Error opening file output.txt for reading");
        return 1;
    }

    char buffer[100];
    if (fgets(buffer, sizeof(buffer), fp) != NULL) { // Read a line safely
```

```
        printf("Read from file: %s", buffer);
    }
    fclose(fp);
    return 0;
}
```

Listing 51: Opening and Closing a File

14.3 File Positioning

Functions to control the read/write position within a file:

- `long ftell(FILE *stream)`: Returns the current value of the file position indicator for the stream. Useful for determining current position or file size.
- `int fseek(FILE *stream, long offset, int origin)`: Sets the file position indicator.
  - `offset`: Number of bytes to move.
  - `origin`: Starting point for offset:
    - `SEEK_SET` (0): From the beginning of the file.
    - `SEEK_CUR` (1): From the current position.
    - `SEEK_END` (2): From the end of the file.
- `void rewind(FILE *stream)`: Sets the file position indicator to the beginning of the file (equivalent to 'fseek(stream, 0L, SEEK$_SET$);'*andclearserrorindicators*).

15 Error Handling

Robust C programs include error handling to gracefully manage unexpected situations during runtime, rather than crashing.

15.1 Using `errno` and `perror`

- `<errno.h>`: Defines the integer variable `errno`. Many standard library functions (especially I/O and low-level system calls) set `errno` to an error code when an error occurs. You should check `errno` immediately after a function call that might set it.
- `void perror(const char *s)`: From `<stdio.h>`, prints a descriptive error message to `stderr` based on the current value of `errno`, prefixed by the string `s`.
- `char *strerror(int errnum)`: From `<string.h>`, returns a pointer to a string describing the error code `errnum`.

```
#include <stdio.h>
#include <errno.h>   // For errno
#include <string.h>  // For strerror

int main() {
    FILE *fp = fopen("nonexistent_file.txt", "r");
    if (fp == NULL) {
        perror("File Open Error"); // Prints "File Open Error: No such file or
            directory" (or similar)
        printf("Error number: %d\n", errno); // Prints the numeric error code
        printf("Error description: %s\n", strerror(errno)); // Prints the
            description string
        return 1;
    }
    // ... file operations
    fclose(fp);
```

```
    return 0;
}
```

Listing 52: Error Handling with errno and perror

15.2 Assertions (<assert.h>)

- `void assert(scalar expression)`: Checks if an expression is true. If false (0), it prints an error message to `stderr` (including filename, line number, and expression) and terminates the program via ‘abort()‘.
- Used for debugging and validating assumptions programmers make about their code’s state.
- Assertions are typically disabled in production builds by defining the ‘NDEBUG‘ macro (e.g., by adding ‘define NDEBUG‘ before ‘include ¡assert.h¿‘, or by compiling with ‘-DNDEBUG‘). When ‘NDEBUG‘ is defined, ‘assert‘ statements compile to nothing.

```c
#include <stdio.h>
#include <assert.h> // For assert

double divide(int a, int b) {
    assert(b != 0 && "Denominator cannot be zero"); // Assertion
    return (double)a / b;
}

int main() {
    printf("Result: %.2f\n", divide(10, 2));
    printf("Result: %.2f\n", divide(10, 0)); // This will trigger the assertion
        and terminate
    return 0;
}
```

Listing 53: Using Assertions

16 Command-Line Arguments

C programs can accept arguments from the command line when executed. This allows for flexible program behavior without recompilation. The `main` function can be declared with parameters to receive these arguments: `int main(int argc, char *argv[])` or `int main(int argc, char **argv)`

- `argc`: (Argument Count) An integer representing the number of command-line arguments. It’s always at least 1 (the program name itself).
- `argv`: (Argument Vector) An array of character pointers (strings).
  - `argv[0]` is the program name (or path to executable).
  - `argv[1]` is the first command-line argument.
  - `argv[argc-1]` is the last command-line argument.
  - `argv[argc]` is a `NULL` pointer, marking the end of the array.

```c
#include <stdio.h>
#include <stdlib.h> // For atoi (ASCII to integer)

int main(int argc, char *argv[]) {
    printf("Program name: %s\n", argv[0]);
    printf("Number of arguments (including program name): %d\n", argc);

    if (argc > 1) {
        printf("Arguments provided:\n");
```

```c
        for (int i = 1; i < argc; i++) { // Start from index 1 to skip program
            name
            printf("  Argument %d: %s\n", i, argv[i]);
        }
        // Example: converting an argument to an integer
        if (argc > 1) {
            int num = atoi(argv[1]); // Convert string argument to integer
            printf("First argument as integer: %d\n", num);
        }
    } else {
        printf("No additional command-line arguments provided.\n");
    }
    return 0;
}
```

Listing 54: Command-Line Arguments Example

If compiled as 'myprog' and run as 'myprog 123 hello world': `Program name: myprog Number of arguments (including program name): 4 Arguments provided: Argument 1: 123 Argument 2: hello Argument 3: world First argument as integer: 123`

17 Preprocessor and Header Files

The preprocessor is the first phase of compilation. It processes directives (commands starting with `#`) before the actual compiler sees the code.

17.1 Preprocessor Directives

- `#include`: Inserts the entire content of another file into the current source file at the point of the directive.
  - `#include <filename>`: Used for standard library header files. The preprocessor searches in standard system directories.
  - `#include "filename"`: Used for user-defined header files. The preprocessor typically searches the current directory first, then standard directories.
- `#define`: Defines a macro, which is a text substitution.
  - **Object-like macros**: Simple text replacement.

```c
        #define PI 3.14159
        #define MAX_SIZE 100
        #define MESSAGE "Hello from macro!"
```

Listing 55: Object-like Macro

  - **Function-like macros**: Take arguments and perform substitutions. **Caution**: Use parentheses around arguments and the entire macro body to avoid common pitfalls with operator precedence.

```c
        #define SQUARE(x) ((x)*(x)) // Parentheses crucial!
        // int result = SQUARE(a + b); // Expands to ((a+b)*(a+b))
        // If #define SQUARE(x) x*x, then SQUARE(a+b) expands to a+b*a+b (
            incorrect)
```

Listing 56: Function-like Macro (with caution)

14

- **The # Operator (Stringification)**: Converts a macro argument into a string literal.

```c
#define PRINT_VAR(var) printf(#var " = %d\n", var)
// PRINT_VAR(my_variable); // Expands to printf("my_variable" " = %d\n", my_variable);
```

Listing 57: Stringification Operator

- **The ## Operator (Token Pasting)**: Concatenates two tokens into a single token.

```c
#define CONCAT(a, b) a ## b
// int my_var_1 = CONCAT(my_var_, 1); // Expands to int my_var_1 = my_var_1;
```

Listing 58: Token Pasting Operator

- **Predefined Macros**: Compiler-supplied macros providing useful information (e.g., `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__STDC__`). C99 added '$_{func. for the name of the current function.}$

`#undef`: Undefines a macro.

`#error`: Forces the compiler to issue a fatal error message and stop compilation. Useful for enforcing constraints (e.g., if a certain macro is not defined).

`#line`: Used to change the '$_{LINE, and '}$$_{FILE, predefined macros, typically by code generators.}$`#pragma`$_{: Provides implementation-defined instructions to the compile}$

17.2 Conditional Compilation

Directives that allow parts of the code to be compiled only if certain conditions are met. This is essential for platform-specific code, debugging, and configuring features.

- `#ifdef MACRO`: Compiles the block if `MACRO` is defined.
- `#ifndef MACRO`: Compiles the block if `MACRO` is NOT defined.
- `#if expression`: Compiles the block if `expression` evaluates to non-zero (true). 'expression' can include integer constants, macros, and the 'defined' operator.
- `#else`: Provides an alternative block for `#if`, `#ifdef`, `#ifndef`.
- `#elif expression`: (Else-if) A combination of `#else` and `#if`.
- `#endif`: Marks the end of an `#if/#ifdef/#ifndef` block.

```c
#define DEBUG_MODE

#ifdef DEBUG_MODE
    printf("Debugging enabled.\n");
#else
    printf("Debugging disabled.\n");
#endif

#if __STDC_VERSION__ >= 199901L
    printf("Compiling with C99 or later.\n");
#endif
```

Listing 59: Conditional Compilation Example

17.3 Header Files

- Header files (`.h` extension) contain declarations (function prototypes, macro definitions, type definitions, external variable declarations). They typically do *not* contain function definitions or global variable definitions.
- They are included in source files (`.c` files) to provide necessary declarations for the compiler to correctly translate function calls and variable accesses.

- **Include Guards**: Prevent multiple inclusions of the same header file within a single compilation unit, which can lead to redefinition errors.

```c
// myheader.h
#ifndef MY_HEADER_H // Check if MY_HEADER_H is not defined
#define MY_HEADER_H // If not defined, define it

// Declarations for this header go here
void myFunction();
extern int globalVar;

#endif // End of MY_HEADER_H guard
```

Listing 60: Include Guard

- **Sharing Variable Declarations**: 'extern' declarations for global variables in headers.
- **`#error` Directives in Header Files**: Can be used to check for compiler features or required macro definitions, causing a compile-time error if conditions are not met.

18 Writing Large Programs: Linking and Libraries

As programs grow, they are typically split into multiple source files for better organization, modularity, and easier management.

18.1 Source Files (`.c`)

Each `.c` file (often called a "compilation unit") typically contains the definitions of functions and global variables related to a specific logical module of the program.

18.2 Building a Multiple-File Program

- Each `.c` file is compiled independently into an object file (`.o` or `.obj`).
- The linker then combines these object files with required library files to create the final executable.
- Example using GCC:

```
# Compile each .c file into an object file (-c flag)
gcc -c main.c -o main.o
gcc -c utility.c -o utility.o

# Link all object files to create the executable
gcc main.o utility.o -o myprogram
```

Listing 61: Compiling Multiple Files

- **Makefiles**: For larger projects, build automation tools like 'make' are used. A Makefile specifies dependencies and rules for building the project, automating the compilation and linking steps.
- **Errors During Linking**: Common linking errors include "undefined reference to" (a function or global variable was declared but never defined) or "multiple definition of" (a function or global variable was defined in more than one source file).

18.3 Library Functions

C programs rely heavily on standard library functions.

- **Standard C Library**: A collection of pre-compiled functions defined by the C standard, available for all C programs. They are organized into headers.
  - `<stdio.h>`: Standard Input/Output (`printf`, `scanf`, `fopen`, etc.)
  - `<stdlib.h>`: Standard Utilities (`malloc`, `free`, `exit`, `atoi`, `qsort`, `rand`, etc.)

  - `<string.h>`: String Manipulation (`strlen`, `strcpy`, `strcmp`, etc.)
  - `<math.h>`: Mathematical Functions (`sqrt`, `sin`, `cos`, `pow`, etc.)
  - `<ctype.h>`: Character Classification and Conversion (`isdigit`, `isalpha`, `tolower`, `toupper`, etc.)
  - `<time.h>`: Date and Time Utilities.
  - `<assert.h>`: Diagnostic functions (`assert`).
  - `<errno.h>`: Error codes (`errno`, `perror`).
  - `<stdarg.h>`: Variable argument lists (for functions like 'printf').
  - `<stdbool.h>` (C99): Boolean type ('bool').
- `<stdint.h>` (C99): Integer types with specific widths (e.g., '$int32_t$', '$uint64_t$').
- **Using the Library**: To use a library function, include its corresponding header file with 'include'. The linker automatically links common standard libraries. For some specialized libraries (e.g., 'math.h'), you might need to specify a linker flag (e.g., '-lm' for GCC).

19 Important Algorithms

C's performance and control make it ideal for implementing and studying algorithms. Understanding these fundamental algorithms is crucial.

19.1 Searching Algorithms

19.1.1 Linear Search

- **Concept**: Sequentially checks each element of a list/array until a match is found or the end of the list is reached.
- **Time Complexity**:
  - Best Case: O(1) (element is at the first position).
  - Average Case: O(n/2) $\rightarrow$ O(n).
  - Worst Case: O(n) (element is at the last position or not present).
- **Applicability**: Works on unsorted arrays and linked lists. Simple to implement.

```c
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i; // Key found at index i
        }
    }
    return -1; // Key not found
}
```

Listing 62: Linear Search

19.1.2 Binary Search

- **Concept**: An efficient search algorithm for a **sorted** array. It repeatedly divides the search interval in half. It compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated, and the search continues on the remaining half.
- **Time Complexity**: O(log n) in all cases (best, average, worst).
- **Prerequisite**: The array must be **sorted** in ascending order.

```c
int binarySearch(int arr[], int n, int key) {
    int low = 0;
    int high = n - 1; // n is the number of elements

    while (low <= high) {
```

```
        int mid = low + (high - low) / 2; // Prevents potential overflow with (low + high) / 2

        if (arr[mid] == key) {
            return mid; // Key found at middle index
        }
        if (arr[mid] < key) {
            low = mid + 1; // Key is in the right half
        } else {
            high = mid - 1; // Key is in the left half
        }
    }
    return -1; // Key not found
}
```

Listing 63: Binary Search (Iterative Implementation)

19.2 Sorting Algorithms

19.2.1 Bubble Sort

- **Concept**: Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. Passes through the list are repeated until no swaps are needed, indicating that the list is sorted. Larger elements "bubble up" to their correct positions.
- **Time Complexity**:
  - Best Case: O(n) (if array is already sorted, optimization can detect no swaps).
  - Average/Worst Case: $O(n^2)$.
- **Space Complexity**: O(1) (in-place sort).
- **Stability**: Stable (maintains the relative order of equal elements).

```
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    int swapped; // Flag to optimize for already sorted arrays

    for (i = 0; i < n - 1; i++) {
        swapped = 0; // Assume no swaps in this pass
        // Last i elements are already in place, so no need to check them
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = 1; // A swap occurred
            }
        }
        // If no two elements were swapped by inner loop, then array is sorted
        if (swapped == 0)
            break;
    }
}
```

Listing 64: Bubble Sort

19.2.2 Selection Sort

- **Concept**: Divides the array into a sorted and an unsorted part. Finds the minimum element from the unsorted part and puts it at the beginning of the unsorted part (which means it's appended to the sorted part). Repeats until the entire array is sorted.
- **Time Complexity**: $O(n^2)$ in all cases (best, average, worst) due to nested loops. Number of swaps is minimal.
- **Space Complexity**: O(1) (in-place sort).
- **Stability**: Unstable.

```c
void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Swap the found minimum element with the first element of the unsorted
            part
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Listing 65: Selection Sort

19.2.3 Insertion Sort

- **Concept**: Builds the final sorted array (or list) one item at a time. It iterates through the input elements and consumes one input element at each repetition, and builds a sorted output list. Each iteration removes one element from the input data and inserts it into the correct position among the already sorted elements.
- **Time Complexity**:
  - Best Case: O(n) (if array is already sorted).
  - Average/Worst Case: $O(n^2)$.
- **Space Complexity**: O(1) (in-place sort).
- **Stability**: Stable.
- **Applicability**: Efficient for small data sets or data sets that are already substantially sorted.

```c
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) { // Start from the second element
        key = arr[i]; // Element to be inserted
        j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j = j - 1;
```

```
        }
        arr[j+1] = key; // Place key at its correct position
    }
}
```

Listing 66: Insertion Sort

19.2.4 Quick Sort (`qsort`)

- **Concept**: A highly efficient, divide-and-conquer sorting algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. The sub-arrays are then recursively sorted.
- **Time Complexity**: O(n log n) on average. $O(n^2)$ in worst case (though rare with good pivot selection).
- **Space Complexity**: O(log n) on average (for recursion stack), O(n) in worst case.
- **Stability**: Unstable.
- **Standard Library Function**: C's standard library provides a general-purpose sorting function `qsort` in `<stdlib.h>`, which often uses an optimized Quick Sort implementation. It's generic and can sort any array of elements by taking a pointer to the array, the number of elements, the size of each element, and a comparison function.

```
#include <stdlib.h> // For qsort
#include <stdio.h>

// Comparison function for integers (required by qsort)
int compareIntegers(const void *a, const void *b) {
    // Cast void pointers to int pointers, then dereference to get values
    int val1 = *(const int *)a;
    int val2 = *(const int *)b;
    if (val1 < val2) return -1;
    if (val1 > val2) return 1;
    return 0; // Equal
    // A shorter way: return (val1 - val2); (but can overflow if values are very
        large/small)
}

int main() {
    int arr[] = {5, 2, 8, 1, 9, 4, 7, 3, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Call qsort
    qsort(arr, n, sizeof(int), compareIntegers);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Listing 67: Using ‘qsort‘