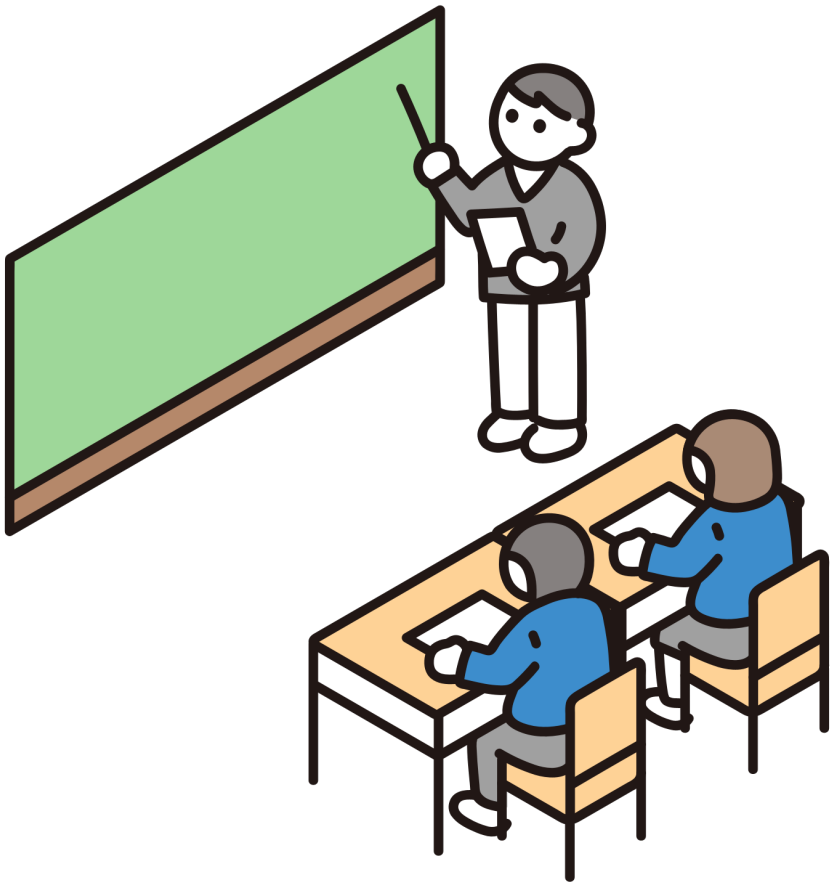
C for Beginners



readbytes

C for Beginners

All you need to know about C

readbytes.github.io

2025-07-17

This page is intentionally left blank.

Contents

Chapter 1.

Getting Started with C

1. What is C and Why Learn It?
2. Installing a C Compiler (GCC, Clang, IDEs)
3. Writing and Compiling Your First Program
4. Understanding the Structure of a C Program

1 Getting Started with C

1.1 What is C and Why Learn It?

The **C programming language** is often referred to as the "mother of all modern programming languages." Developed in the early 1970s by **Dennis Ritchie** at Bell Labs, C was initially created to implement the UNIX operating system. At the time, most systems were written in assembly language, which was both platform-specific and difficult to manage. C provided a way to write efficient, low-level code that could run on multiple platforms, marking a revolution in software development.

A Historical Perspective

C's origin story is deeply intertwined with the evolution of modern computing. UNIX, one of the first portable operating systems, was rewritten in C in 1973. This was groundbreaking because it allowed UNIX to run on different machines with minimal changes to the codebase. At a time when computers were large, expensive, and incompatible with one another, this portability was a game changer.

Over the decades, C became the foundation for many subsequent programming languages, including **C++, Java, C#, Objective-C, Go, Rust, and even modern scripting languages** like Python and JavaScript. Many of these languages borrow C's syntax and concepts, making C an ideal starting point for learning programming. Understanding C gives programmers insight into how higher-level abstractions are built and how the computer manages resources under the hood.

Why Learn C Today?

Despite being over 50 years old, C is still widely taught and used. There are several compelling reasons why C remains relevant and essential in today's software landscape:

1. **Performance**: C provides direct access to hardware and memory through pointers, making it extremely fast and efficient. There is very little abstraction between what you write and what the machine executes. This makes it a preferred choice for performance-critical applications like game engines, real-time systems, and embedded software.
2. **Portability**: Programs written in C can be compiled and run on virtually any type of computer with minimal modification. This is largely due to the existence of standard C libraries and the ANSI C standard, which define a consistent set of behaviors across platforms. As a result, C is often used for cross-platform development, especially in system-level programming.
3. **Low-Level Control**: C provides fine-grained control over system resources, such as memory allocation and CPU usage. While this level of control demands a greater responsibility from the programmer, it also offers deeper understanding and flexibility—valuable skills in many computing disciplines.
4. **Foundation for Other Languages**: Learning C helps you understand how computers

work at a fundamental level. It introduces concepts like pointers, memory management, bitwise operations, and manual resource control. These are critical for understanding how other languages work internally and for troubleshooting complex problems in any language.

5. **Wide Usage in Real-World Applications**: C is everywhere—even if you don't see it. Here are just a few areas where C is heavily used:
    - **Operating Systems**: Windows, Linux, and macOS all contain large portions of code written in C.
    - **Embedded Systems**: Microcontrollers in cars, medical devices, home appliances, and IoT devices often run software written in C due to its efficiency and low-level hardware access.
    - **Game Development**: Game engines like Unreal Engine use C and C++ to achieve high performance.
    - **Databases and Interpreters**: Many database engines (like SQLite and MySQL) and language interpreters (like Python's CPython) are implemented in C.
    - **Compilers and Virtual Machines**: Many compilers and virtual machines (e.g., LLVM, GCC, JVM) are either written in C or use it extensively for performance-critical components.

C in Education and Industry

C is a staple in computer science education worldwide. Most computer science programs introduce students to programming through C or C++, as it forces them to think critically about what their code is doing, especially regarding memory and system resources. Unlike higher-level languages that hide these details, C lays everything bare—giving learners a transparent and honest look at what's happening inside the machine.

In industry, C is a reliable and trusted choice for building infrastructure software. Its long-standing presence and widespread adoption mean that there is a wealth of tools, libraries, and documentation available. Moreover, many job roles—particularly those in systems programming, embedded development, and cybersecurity—explicitly require or prefer proficiency in C.

1.2 Installing a C Compiler (GCC, Clang, IDEs)

Before you can write and run C programs, you need a **C compiler**—a tool that converts your C code into machine code the computer can execute. Two of the most widely used C compilers are **GCC (GNU Compiler Collection)** and **Clang**. Depending on your operating system, the installation steps vary slightly. You may also want to use an **IDE (Integrated Development Environment)** or a **text editor** to make writing C code easier and more productive.

Let's go through the setup process for each major platform.

Installing GCC or Clang

Windows Windows doesn't come with a built-in C compiler, but you have several options:

Option 1: Install GCC via MinGW (Minimalist GNU for Windows)

1. Download the MinGW installer from https://osdn.net/projects/mingw/releases/.
2. Run the installer and select:
   - `mingw32-gcc-g++` (this includes the C/C++ compilers)
3. After installation, add the `bin` folder (e.g., `C:\MinGW\bin`) to your **system PATH**:
   - Open **Start Menu → Environment Variables**
   - Edit the `Path` variable and add:

     `C:\MinGW\bin`
4. Open **Command Prompt** and verify:

```
gcc --version
```

Option 2: Use MSYS2 (GCC and more tools)

1. Download MSYS2 from https://www.msys2.org.
2. Follow the setup instructions, then run:

```
pacman -Syu   # update the system
pacman -S mingw-w64-ucrt-x86_64-gcc
```

Option 3: Use WSL (Windows Subsystem for Linux)

1. Install **WSL** and a Linux distribution (e.g., Ubuntu) from the Microsoft Store.
2. Open the WSL terminal and install GCC:

```
sudo apt update
sudo apt install build-essential
```

macOS macOS users can easily install Clang, which is Apple's version of the LLVM compiler suite.

Step 1: Install Xcode Command Line Tools

Open **Terminal** and run:

```
xcode-select --install
```

This installs Clang, `make`, and other essential tools.

Verify the installation:

```
clang --version
```

Linux (Ubuntu/Debian) Most Linux distributions include GCC or make it easy to install:

```
sudo apt update
sudo apt install build-essential
```

For Clang:

```
sudo apt install clang
```

Verify:

```
gcc --version
# or
clang --version
```

Choosing an IDE or Text Editor

While you can write C programs in any text editor, using an **IDE** or **editor with C support** makes development easier with features like syntax highlighting, auto-completion, and debugging tools.

Here are popular options:

Visual Studio Code (VS Code) *Free, cross-platform*

- Download from: https://code.visualstudio.com
- Install the **C/C++ extension** by Microsoft
- Configure tasks to compile and run using `gcc` or `clang`

Code::Blocks *Free, beginner-friendly*

- Download from: http://www.codeblocks.org
- Choose the version **with MinGW** included (on Windows)
- Built-in compiler integration and debugging support

CLion *Professional IDE by JetBrains (paid)*

- Download: https://www.jetbrains.com/clion
- Works on Windows, macOS, and Linux
- Includes refactoring tools, project navigation, and debugger

Other Editors

- **Notepad++ (Windows)** – Lightweight and configurable
- **Vim / Emacs (Linux/macOS)** – Popular with power users
- **Geany** – A simple IDE available on all platforms

Verifying Installation and Environment Setup

After installing your compiler, test it by creating a simple C program:

Create a file `hello.c`:

Full runnable code:

```c
#include <stdio.h>

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

Compile and run:

```
gcc hello.c -o hello
./hello
```

If you see `Hello, world!` printed on your screen, your development environment is ready!

Summary

Installing a C compiler is your first step toward writing real-world programs. Whether you choose **GCC** or **Clang**, or use an IDE like **VS Code** or **Code::Blocks**, the key is to pick a setup you're comfortable with. Once installed, you're ready to write, compile, and run C code right on your own machine. In the next section, we'll guide you through creating and compiling your very first C program.

1.3 Writing and Compiling Your First Program

Now that your development environment is ready, it's time to write your very first C program. A traditional way to begin learning any programming language is by printing the message: **"Hello, World!"** This simple program demonstrates the basic structure of a C program and introduces essential concepts like functions, headers, and statements.

Step 1: Writing the Code

Open your preferred text editor or IDE, and create a new file called `hello.c`.

Type in the following code:

Full runnable code:

```c
#include <stdio.h>

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

Let's break it down line by line:

Code Explanation

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

- This line tells the compiler to **include the standard input/output library**.
- `stdio.h` contains functions like `printf`, which we use to display text on the screen.

```c
int main() {
```

- This is the **main function**—the entry point of every C program.
- `int` means the function returns an integer (0 means success).
- The `{` marks the beginning of the function's body.

```c
    printf("Hello, World!\n");
```

- This line prints the text **Hello, World!** to the console.
- `\n` is a special character called a **newline**, which moves the cursor to the next line after printing.
- `printf` stands for "print formatted" and is part of the `stdio.h` library.

```c
    return 0;
```

- This tells the operating system that the program has finished successfully.
- `0` is a common convention for success; other numbers can indicate different errors.

```c
}
```

- This closes the `main` function block.

Step 2: Compiling the Program (Command Line)

Make sure your terminal or command prompt is open and you're in the same directory as your `hello.c` file.

To compile with GCC:

```
gcc hello.c -o hello
```

- `hello.c` is the source file.
- `-o hello` tells the compiler to name the output file `hello`.

If there are no errors, it will produce an executable file:

- On Linux/macOS: `./hello`
- On Windows: `hello.exe`

Run the program:

```
./hello     # (Linux/macOS)
hello.exe   # (Windows)
```

Expected Output:

```
Hello, World!
```

Step 3: Compiling and Running in an IDE

Using Code::Blocks:

1. Open Code::Blocks and choose **File → New → Project → Console Application**.
2. Select **C**, name your project, and set the project location.
3. In the `main.c` file that appears, replace the default code with your `hello.c` example.
4. Click **Build and Run (F9)**.

Using Visual Studio Code:

1. Install the **C/C++ extension** by Microsoft.
2. Open a folder with your `hello.c` file.
3. Use **Terminal → Run Build Task** or create a `tasks.json` file to compile with GCC.
4. Run the executable from the integrated terminal.

Common Beginner Errors

1. **Missing Semicolon:**

```
printf("Hello, World!\n")   // Error: missing semicolon
```

   - **Fix:** Always end statements with a `;`

2. **Typos in Keywords:**

```
#includ <stdio.h>     // Error: "includ" should be "include"
```

3. **Mismatched Braces:**

```
int main() {
    printf("Hello");
// Missing closing brace
```

4. **Wrong File Name or Directory:**

   - Make sure you compile the correct file and that you're in the right folder.

Tips for Beginners

- Always **save your file** before compiling.
- If you see an error message, **read it carefully**—compilers often point you to the exact line.
- Don't worry if it doesn't work the first time. Debugging is a normal part of programming!

Conclusion

You've just written and run your first C program! While it's simple, it lays the foundation for more complex topics like variables, control structures, functions, and memory management. In the next section, we'll dive deeper into understanding how a C program is structured—and why things like `main()` and `#include` are essential parts of every C project.

1.4 Understanding the Structure of a C Program

Every C program—no matter how simple or complex—follows a specific structure. Understanding this structure is essential to writing readable, correct, and efficient code. In this section, we'll break down the key components of a typical C program and explain the role each part plays. By the end, you'll be able to recognize and build cleanly structured C programs with confidence.

Basic Structure of a C Program

Let's revisit the classic "Hello, World!" program:

Full runnable code:

```c
#include <stdio.h>

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

This short program illustrates many of the key components that make up the skeleton of a C application.

1.4.1 Preprocessor Directives

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

- Lines that begin with `#` are **preprocessor directives**.
- The `#include` directive tells the compiler to include the contents of another file—in this case, the **standard input/output library** (`stdio.h`) which defines the `printf()` function.
- These directives are processed **before** compilation and are not actual C statements (so they don't end with a semicolon).

1.4.2 The `main()` Function

```c
int main() {
    // Code goes here
    return 0;
}
```

- Every C program **must have a `main()` function**. This is where execution starts.
- The `int` return type means that `main()` returns an **integer value** to the operating system. Conventionally, `return 0;` signals successful completion.

- You can write `main()` with or without parameters (like `int argc, char *argv[]`), but we'll use the simpler version for now.

1.4.3 Statements and Functions

```
    printf("Hello, World!\n");
    return 0;
```

- These lines are **statements**—instructions that perform actions.
- Each statement must end with a **semicolon (`;`)**.
- `printf()` is a **function call**. It prints text to the console. The `\n` at the end is a **newline character**, which moves the cursor to the next line.
- `return 0;` ends the program and signals success to the operating system.

1.4.4 Braces and Code Blocks

```
int main() {
    // statements
}
```

- The **curly braces `{}`** define the **body of the function**, also known as a code block.
- All code that belongs to a function or a control structure (like loops or if-statements) must be enclosed in braces.
- Blocks can contain one or more statements.

1.4.5 Comments

```
// This is a single-line comment

/*
   This is a
   multi-line comment
*/
```

- Comments are ignored by the compiler and are meant to **explain code to humans**.
- Use `//` for single-line comments.
- Use `/* */` for multi-line comments.
- Writing meaningful comments is a good habit—it helps both you and others understand your code later.

1.4.6 Indentation and Formatting

C doesn't require indentation to function correctly (unlike Python), but **consistent indentation improves readability**. For example:

Poorly formatted:

```
int main(){printf("Hello");return 0;}
```

Well-formatted:

```
int main() {
    printf("Hello");
    return 0;
}
```

- Most C programmers follow the convention of **4 spaces or 1 tab** per indent level.
- Braces are usually placed on the same line or the next, depending on style, but consistency is key.

1.4.7 Alternative Minimal C Programs

Let's look at two slightly different, but valid, C programs:

Program 1:

Full runnable code:

```
#include <stdio.h>

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

Program 2 (Compact Style):

```
#include <stdio.h>
int main(){puts("Hi!");return 0;}
```

- Both programs print text to the screen and exit cleanly.
- Program 2 uses `puts()` instead of `printf()`. `puts()` adds a newline automatically.
- Both styles are acceptable, but the first is easier to read and preferred for learning.

1.4.8 Summary

A C program is built from simple building blocks: **preprocessor directives**, the `main()` function, **statements**, **code blocks**, and **comments**. You've also seen how **semicolons**,

braces, and **indentation** play vital roles in defining program structure and clarity. While this structure is simple, mastering it lays the foundation for writing more advanced and professional C code.

In the next chapter, we'll dive into variables, data types, and how to store and manipulate data in your programs.

Chapter 2.

Basic Syntax and Data Types

1. Keywords, Identifiers, and Comments
2. Variables and Constants
3. Basic Data Types (`int`, `char`, `float`, `double`)
4. Input and Output (`scanf`, `printf`)

2 Basic Syntax and Data Types

2.1 Keywords, Identifiers, and Comments

Understanding the basic building blocks of C—**keywords**, **identifiers**, and **comments**—is essential to writing readable and correct programs. These elements help define how a C program is written, what it does, and how others (or you in the future) can understand it.

2.1.1 Keywords in C

Keywords are reserved words in the C language that have special meaning to the compiler. You **cannot** use them as names for variables, functions, or other identifiers. These words are the fundamental vocabulary of C and define the structure and behavior of programs.

Here are some commonly used C keywords:

Keyword	Purpose
`int`	Declares an integer variable
`char`	Declares a character variable
`float`	Declares a floating-point variable
`double`	Declares a double-precision float
`return`	Returns a value from a function
`if`	Starts a conditional block
`else`	Specifies an alternative branch
`for`	Starts a loop with counter
`while`	Starts a loop based on a condition
`void`	Indicates no return value
`break`	Exits a loop or switch statement

Example:

```c
int main() {
    int number = 10;
    if (number > 0) {
        printf("Positive number\n");
    }
    return 0;
}
```

In this example, `int`, `if`, and `return` are all keywords that help define the behavior of the program.

2.1.2 Identifiers in C

Identifiers are the names used for variables, functions, arrays, and other user-defined elements. While you create identifiers, you must follow C's naming rules:

Rules for Naming Identifiers:

- Must begin with a **letter (A–Z, a–z)** or an **underscore (_)**
- Can contain **letters**, **digits (0–9)**, and **underscores**
- Cannot use **C keywords** as identifiers
- C is **case-sensitive**: `score`, `Score`, and `SCORE` are different

Examples:

YES Valid identifiers:

```c
int age;
float total_score;
char name1;
```

NO Invalid identifiers:

```c
int 1value;     // Cannot start with a digit
float float;    // Cannot use a keyword
char full-name; // Hyphen is not allowed
```

Use **descriptive names** to make your code easier to read:

```c
int studentAge;     // Good
int x;              // Too vague
```

2.1.3 Comments in C

Comments are notes you write in your code to explain what it does. They are ignored by the compiler and have no effect on the program's execution.

C supports two types of comments:

Single-line comments

Use `//` to write a comment that ends at the end of the line.

```c
int age = 18; // store the user's age
```

Multi-line comments

Use `/* */` to write comments that span multiple lines.

```c
/* This program calculates
   the area of a rectangle */
int length = 10;
int width = 5;
```

Why use comments?

- To explain **what** the code does and **why**
- To temporarily disable a line of code for testing
- To improve **readability** for yourself and others

2.1.4 Conclusion

In summary:

- **Keywords** are reserved by the language and must not be used as names.
- **Identifiers** are names you create for your variables, functions, and more.
- **Comments** help document your code for better understanding.

Getting comfortable with these elements early on will make writing and reading C code much easier as you move on to more complex programs.

2.2 Variables and Constants

In any programming language, **variables** and **constants** are fundamental concepts that allow you to store and work with data. In C, understanding how to declare, assign, and use variables and constants correctly is essential for writing effective programs.

What Are Variables?

A **variable** is a named storage location in the computer's memory that holds a value you can change during program execution. Variables allow programs to store user inputs, calculate results, and keep track of data.

Declaring Variables

To declare a variable in C, you specify its **data type** followed by its **name (identifier)**:

```
int age;
float price;
char grade;
```

- `int age;` declares a variable named `age` of type `int` (integer).
- `float price;` declares a floating-point variable `price`.
- `char grade;` declares a character variable `grade`.

At this point, the variable is **declared but uninitialized**, meaning it holds garbage (random) data until you assign a value.

Assigning Values

You assign a value to a variable using the **assignment operator =**:

```
age = 25;
price = 19.99;
grade = 'A';
```

You can also **declare and initialize** a variable in one statement:

```
int score = 100;
char letter = 'B';
float temperature = 36.5;
```

Using Variables

Once declared and assigned, variables can be used in expressions, function calls, and more:

```
int a = 5;
int b = 10;
int sum = a + b;          // sum now holds 15
printf("Sum: %d\n", sum);
```

Naming Conventions

Choosing meaningful names helps make your code easier to understand and maintain:

- Use letters, digits, and underscores (_).
- Start names with a letter or underscore.
- Avoid using keywords.
- Use **camelCase** or **snake_case** styles for readability:
  - `totalScore` (camelCase)
  - `total_score` (snake_case)
- Be descriptive: `studentAge` is better than `a`.

What Are Constants?

Sometimes, you need values that **do not change** throughout the program. These are called **constants**. Using constants helps avoid accidental modification of values that should stay fixed, making your code safer and clearer.

Using the `const` Keyword

The `const` keyword declares a variable whose value cannot be changed after initialization:

```
const float PI = 3.14159;
const int MAX_USERS = 100;
```

If you try to assign a new value to a `const` variable later, the compiler will generate an error.

Using `#define` for Constants

Another way to define constants is with the **preprocessor directive** `#define`. This instructs the preprocessor to replace all occurrences of the identifier with the specified value before compilation:

```
#define PI 3.14159
#define MAX_USERS 100
```

Unlike `const`, `#define` does not consume memory, and it doesn't have a data type — it's a simple text substitution.

Differences Between `const` and `#define`

Feature	`const`	`#define`
Type checking	Yes	No
Memory usage	Allocates memory	No memory (text replacement)
Debugging	Variables visible in debugger	No visibility
Scope	Respects C scoping rules	Global across file

When to Use Constants

- Use **`const`** when you want type safety and to create typed constants.
- Use **`#define`** for simple symbolic constants or macros.
- Constants improve **code readability** by giving meaningful names to fixed values.
- They help avoid **magic numbers**—unnamed literal values scattered throughout code.

Example: Variables and Constants Together

Full runnable code:

```
#include <stdio.h>

#define DAYS_IN_WEEK 7

int main() {
    const float TAX_RATE = 0.08;
    int hoursWorked = 40;
    float hourlyWage = 15.50;

    float salary = hoursWorked * hourlyWage;
    float tax = salary * TAX_RATE;

    printf("Weekly salary: %.2f\n", salary);
    printf("Tax amount: %.2f\n", tax);
    printf("Days in a week: %d\n", DAYS_IN_WEEK);

    return 0;
}
```

Here:

- `hoursWorked` and `hourlyWage` are variables because their values might change.
- `TAX_RATE` is a constant that shouldn't change during program execution.
- `DAYS_IN_WEEK` is a constant defined with `#define`.

2.2.1 Summary

- **Variables** hold data that can change.
- They must be declared with a **data type** and can be initialized at declaration.
- **Constants** hold values that do not change and are declared with `const` or `#define`.
- Using constants makes programs safer and easier to read.
- Follow good naming conventions for clarity.

Mastering variables and constants is a key step toward writing useful C programs that manipulate data effectively.

2.3 Basic Data Types (`int`, `char`, `float`, `double`)

In C, data types define the kind of data a variable can hold. Choosing the right data type is important because it affects how much memory your program uses and how accurately values are stored and processed. In this section, we'll explore four fundamental built-in data types: **`int`**, **`char`**, **`float`**, and **`double`**.

2.3.1 `int` Integer Type

- **What is it?** `int` stands for **integer** and is used to store whole numbers without decimals.
- **Typical size:** Usually 4 bytes (32 bits) on modern systems, but this can vary depending on the system and compiler.
- **Range:** Approximately -2,147,483,648 to 2,147,483,647 for a 4-byte signed `int`.
- **Use case:** Used for counting, indexing, flags, and any numeric value without fractions.

Example:

Full runnable code:

```c
#include <stdio.h>

int main() {
    int age = 30;
```

```
    printf("Age: %d\n", age);
    return 0;
}
```

- `%d` is the **format specifier** for printing integers with `printf()`.

2.3.2 `char` Character Type

- **What is it?** `char` stores a **single character**. Internally, characters are represented as small integers using ASCII or Unicode values.
- **Typical size:** 1 byte (8 bits).
- **Range:** -128 to 127 (signed) or 0 to 255 (unsigned), depending on compiler defaults.
- **Use case:** Storing characters like letters, digits, or symbols. Also used for manipulating text strings (arrays of `char`).

Example:

Full runnable code:

```
#include <stdio.h>

int main() {
    char letter = 'A';
    printf("Letter: %c\n", letter);
    return 0;
}
```

- `%c` is the **format specifier** for printing a single character.

2.3.3 `float` Floating-Point Type (Single Precision)

- **What is it?** `float` is used to store **real numbers** (numbers with decimal points) with limited precision.
- **Typical size:** 4 bytes (32 bits).
- **Range:** Approximately 1.2 × 10^-38 to 3.4 × 10^38.
- **Precision:** About 6 decimal digits of accuracy.
- **Use case:** When memory is limited and you need to store decimal numbers, but high precision is not critical—e.g., simple measurements or graphics.

Example:

Full runnable code:

```c
#include <stdio.h>

int main() {
    float temperature = 36.5f;
    printf("Temperature: %.2f\n", temperature);
    return 0;
}
```

- `%f` is the **format specifier** for printing floats.
- `%.2f` limits output to 2 decimal places.
- Note the `f` suffix in `36.5f` indicates a float literal.

2.3.4 `double` Double Precision Floating-Point Type

- **What is it?** `double` stores **real numbers** with double the precision of `float`.
- **Typical size:** 8 bytes (64 bits).
- **Range:** Approximately 2.2 × 10^-308 to 1.8 × 10^308.
- **Precision:** About 15 decimal digits of accuracy.
- **Use case:** When higher precision for calculations is needed, such as scientific computations or financial calculations.

Example:

Full runnable code:

```c
#include <stdio.h>

int main() {
    double pi = 3.141592653589793;
    printf("Value of Pi: %.15lf\n", pi);
    return 0;
}
```

- `%lf` is the **format specifier** for printing doubles.
- `%.15lf` prints 15 decimal places.

2.3.5 Why Choose the Right Data Type?

- **Memory efficiency:** Using the smallest appropriate data type saves memory, especially important in embedded systems or large-scale applications.

- **Accuracy:** Using `float` where precision matters can cause errors; use `double` for better accuracy.
- **Performance:** Some systems handle certain types more efficiently; for example, integer operations are often faster than floating-point.

2.3.6 Summary Table

Data Type	Size (Typical)	Range / Precision	Format Specifier	Use Case
`int`	4 bytes	~ -2 billion to +2 billion	`%d`	Whole numbers
`char`	1 byte	0 to 255 or -128 to 127	`%c`	Single characters
`float`	4 bytes	±1.2E-38 to ±3.4E+38 (6 digits)	`%f`	Decimal numbers (less precise)
`double`	8 bytes	±2.2E-308 to ±1.8E+308 (15 digits)	`%lf`	Decimal numbers (high precision)

2.3.7 Final Notes

Remember that the size and range of these types can vary depending on your system and compiler, but the above are typical values on modern desktop platforms. Always pick the data type that best fits your needs—too large wastes memory, too small causes errors.

With this understanding, you're ready to declare variables, store data, and print output accurately using the correct data types in C!

2.4 Input and Output (`scanf`, `printf`)

One of the most important skills in programming is **communicating with the user** — taking input and displaying output. In C, the standard way to do this is through two functions: `scanf()` for input and `printf()` for output. These functions allow your program to interact dynamically with users rather than just running static code.

2.4.1 Understanding `printf()`

The `printf()` function sends formatted output to the screen (console). Its basic syntax is:

```
printf("format string", arguments);
```

- The **format string** contains plain text and **format specifiers**, which are placeholders for variables.
- The **arguments** are the variables or values you want to print.

Common Format Specifiers in `printf()`

Specifier	Data Type	Description
`%d`	`int`	Prints a decimal (integer) number
`%f`	`float/double`	Prints a floating-point number
`%c`	`char`	Prints a single character
`%s`	`char[]`	Prints a string (character array)

Example:

Full runnable code:

```c
#include <stdio.h>

int main() {
    int age = 25;
    float height = 5.9f;
    char grade = 'A';
    char name[] = "Alice";

    printf("Name: %s\n", name);
    printf("Age: %d years\n", age);
    printf("Height: %.1f feet\n", height);
    printf("Grade: %c\n", grade);

    return 0;
}
```

Output:

```
Name: Alice
Age: 25 years
Height: 5.9 feet
Grade: A
```

- Notice the `%.1f` in `height` — this prints the floating-point number with **1 decimal place**.
- `%s` is used for strings, which are arrays of characters terminated by a null character (`\0`).

2.4.2 Understanding `scanf()`

The `scanf()` function reads formatted input from the user via the keyboard. Its basic syntax is:

```
scanf("format string", &variables);
```

- The format string tells `scanf()` what type of input to expect.
- Variables are preceded by `&` (address-of operator) so that `scanf()` can store the input in the correct memory location.

Common Format Specifiers in `scanf()`

Specifier	Data Type	Description
`%d`	`int`	Reads an integer
`%f`	`float`	Reads a floating-point number
`%lf`	`double`	Reads a double precision number
`%c`	`char`	Reads a single character
`%s`	`char[]`	Reads a string (no spaces)

2.4.3 Basic Input Example

Full runnable code:

```c
#include <stdio.h>

int main() {
    int age;
    printf("Enter your age: ");
    scanf("%d", &age);
    printf("You entered: %d\n", age);

    return 0;
}
```

2.4.4 Reading Multiple Inputs

Full runnable code:

```c
#include <stdio.h>

int main() {
    int day, month, year;
    printf("Enter date (day month year): ");
```

```
    scanf("%d %d %d", &day, &month, &year);
    printf("Date entered: %02d/%02d/%04d\n", day, month, year);

    return 0;
}
```

- `%02d` pads single-digit numbers with a zero (e.g., `05`).
- `%04d` pads the year to four digits.

2.4.5 Common Issues with `scanf()`

Forgetting the `&`

Since `scanf()` needs the **address** of the variable, forgetting to add `&` causes errors:

```
int number;
scanf("%d", number);  // Wrong! Should be &number
```

This usually causes a runtime error or unpredictable behavior.

Input Buffer and Character Input

`scanf("%c", &ch);` can behave unexpectedly if preceded by another input because newline characters (`\n`) remain in the input buffer.

Example:

```
char ch;
scanf("%c", &ch);  // May read leftover '\n'
```

Solution: Add a space before `%c`:

```
scanf(" %c", &ch);  // Skips whitespace characters
```

Reading Strings with Spaces

Using `%s` reads a string only up to the first whitespace:

```
char name[20];
scanf("%s", name);
```

If you enter "John Doe", only "John" is read.

To read full lines with spaces, use `fgets()` instead, which reads an entire line including spaces.

2.4.6 Full Example: Input and Output Together

Full runnable code:

```c
#include <stdio.h>

int main() {
    char name[50];
    int age;
    float height;

    printf("Enter your name: ");
    scanf("%s", name);  // Reads until first space

    printf("Enter your age: ");
    scanf("%d", &age);

    printf("Enter your height in meters (e.g., 1.75): ");
    scanf("%f", &height);

    printf("\nSummary:\n");
    printf("Name: %s\n", name);
    printf("Age: %d\n", age);
    printf("Height: %.2f meters\n", height);

    return 0;
}
```

Sample output:

```
Enter your name: Alice
Enter your age: 28
Enter your height in meters (e.g., 1.75): 1.68

Summary:
Name: Alice
Age: 28
Height: 1.68 meters
```

2.4.7 Summary

- Use `printf()` to **display output** with format specifiers matching your data types.
- Use `scanf()` to **read input** and remember to pass variable **addresses using `&`**.
- Common format specifiers include `%d` (int), `%f` (float), `%c` (char), and `%s` (string).
- Watch out for **input buffer issues**, especially with `%c` and strings containing spaces.
- Practice careful formatting for clean, user-friendly programs.

Mastering input and output is a key step to making interactive programs that can respond dynamically to users' needs.

Chapter 3.

Operators and Expressions

1. Arithmetic Operators
2. Relational and Logical Operators
3. Assignment and Compound Assignment
4. Increment and Decrement
5. Type Conversion and Casting

3 Operators and Expressions

3.1 Arithmetic Operators

Arithmetic operators in C are used to perform basic mathematical operations such as addition, subtraction, multiplication, division, and modulus. These operators work on numeric values, either integers or floating-point numbers, allowing your program to calculate and manipulate data effectively.

3.1.1 The Five Basic Arithmetic Operators in C

Operator	Meaning	Description
`+`	Addition	Adds two operands
`-`	Subtraction	Subtracts the second operand from the first
`*`	Multiplication	Multiplies two operands
`/`	Division	Divides the first operand by the second
`%`	Modulus	Finds the remainder of integer division

3.1.2 Using Arithmetic Operators with Integers

Let's look at some examples using integers:

Full runnable code:

```c
#include <stdio.h>

int main() {
    int a = 15, b = 4;

    printf("Addition: %d + %d = %d\n", a, b, a + b);
    printf("Subtraction: %d - %d = %d\n", a, b, a - b);
    printf("Multiplication: %d * %d = %d\n", a, b, a * b);
    printf("Division: %d / %d = %d\n", a, b, a / b);
    printf("Modulus: %d %% %d = %d\n", a, b, a % b);

    return 0;
}
```

Output:

```
Addition: 15 + 4 = 19
Subtraction: 15 - 4 = 11
Multiplication: 15 * 4 = 60
```

```
Division: 15 / 4 = 3
Modulus: 15 % 4 = 3
```

Note:

- The division of two integers results in an **integer quotient**. The fractional part is truncated (discarded).
- The modulus operator `%` works only with integers and gives the **remainder** after division.

3.1.3 Using Arithmetic Operators with Floating-Point Numbers

Now let's try arithmetic with floating-point numbers:

Full runnable code:

```c
#include <stdio.h>

int main() {
    float x = 15.0f, y = 4.0f;

    printf("Addition: %.2f + %.2f = %.2f\n", x, y, x + y);
    printf("Subtraction: %.2f - %.2f = %.2f\n", x, y, x - y);
    printf("Multiplication: %.2f * %.2f = %.2f\n", x, y, x * y);
    printf("Division: %.2f / %.2f = %.2f\n", x, y, x / y);

    // Note: Modulus operator (%) is NOT valid for floats!

    return 0;
}
```

Output:

```
Addition: 15.00 + 4.00 = 19.00
Subtraction: 15.00 - 4.00 = 11.00
Multiplication: 15.00 * 4.00 = 60.00
Division: 15.00 / 4.00 = 3.75
```

Important:

- Floating-point division returns a **floating-point result** including fractions.
- The modulus operator `%` **cannot** be used with floats or doubles. If you need the remainder for floating-point numbers, use the `fmod()` function from `math.h`.

3.1.4 Integer Division vs. Floating-Point Division

This distinction is crucial when mixing data types:

Full runnable code:

```c
#include <stdio.h>

int main() {
    int a = 7, b = 2;
    float x = 7.0f, y = 2.0f;

    printf("Integer division: %d / %d = %d\n", a, b, a / b);
    printf("Floating division (cast): %f / %f = %f\n", x, y, x / y);
    printf("Mixed division: %d / %d = %.2f\n", a, b, (float)a / b);

    return 0;
}
```

Output:

```
Integer division: 7 / 2 = 3
Floating division (cast): 7.000000 / 2.000000 = 3.500000
Mixed division: 7 / 2 = 3.50
```

- When both operands are integers, division truncates the decimal part.
- Casting an integer to `float` before division ensures floating-point division.

3.1.5 Operator Precedence and Grouping

Arithmetic operators have a natural precedence (order of operations) in C:

Operator	Precedence Level	Associativity
`()` (parentheses)	Highest	Left to right
`*`, `/`, `%`	Higher	Left to right
`+`, `-`	Lower	Left to right

Use **parentheses** to control the order explicitly:

Full runnable code:

```c
#include <stdio.h>

int main() {
    int result;

    result = 10 + 5 * 3;    // Multiplication first: 5*3=15, then +10 = 25
    printf("Without parentheses: %d\n", result);

    result = (10 + 5) * 3;  // Parentheses change order: 15*3=45
    printf("With parentheses: %d\n", result);
```

```
    return 0;
}
```

Output:

```
Without parentheses: 25
With parentheses: 45
```

3.1.6 Summary

- Arithmetic operators `+`, `-`, `*`, `/`, `%` perform addition, subtraction, multiplication, division, and modulus respectively.
- Integer division truncates decimals; float division returns fractional results.
- `%` works only with integers.
- Operator precedence matters; use parentheses to clarify and control computation order.
- Mixing types can lead to unexpected results unless you use **casting**.

Mastering these basics will allow you to perform calculations and build more complex expressions confidently as you progress in C programming.

3.2 Relational and Logical Operators

In programming, making decisions based on conditions is fundamental. C provides **relational operators** to compare values and **logical operators** to combine multiple conditions. These operators are essential when writing **conditional expressions** used in control flow statements like `if`, `while`, and `for`.

3.2.1 Relational Operators

Relational operators compare two values and produce a result that is either **true** (non-zero) or **false** (zero). These operators are primarily used to test conditions.

Operator	Meaning	Description
`==`	Equal to	True if two operands are equal
`!=`	Not equal to	True if operands are different
`>`	Greater than	True if left operand is greater
`<`	Less than	True if left operand is smaller
`>=`	Greater than or equal to	True if left operand is greater or equal
`<=`	Less than or equal to	True if left operand is smaller or equal

Operator	Meaning	Description

Examples of Relational Operators

Full runnable code:

```c
#include <stdio.h>

int main() {
    int a = 10, b = 20;

    if (a == b) {
        printf("a and b are equal.\n");
    } else {
        printf("a and b are not equal.\n");
    }

    if (a < b) {
        printf("a is less than b.\n");
    }

    if (b >= 20) {
        printf("b is greater than or equal to 20.\n");
    }

    return 0;
}
```

Output:

```
a and b are not equal.
a is less than b.
b is greater than or equal to 20.
```

3.2.2 Logical Operators

Logical operators allow you to combine multiple relational expressions into a **single compound condition**. These are particularly useful when decisions depend on more than one test.

Operator	Symbol	Description
AND	`&&`	True if **both** operands are true
OR	`|`	True if **either** operand is true
NOT	`!`	Reverses the truth value (negation)

Using Logical Operators

1. **Logical AND (`&&`)**

Both conditions must be true for the entire expression to be true.

```
int age = 25;
int hasID = 1; // 1 means true

if (age >= 18 && hasID) {
    printf("Allowed to enter.\n");
}
```

2. **Logical OR (||)**

At least one condition must be true for the entire expression to be true.

```
int isStudent = 0;
int isTeacher = 1;

if (isStudent || isTeacher) {
    printf("Discount applies.\n");
}
```

3. **Logical NOT (!)**

Reverses the truth value.

```
int isRaining = 0;

if (!isRaining) {
    printf("Go outside!\n");
}
```

3.2.3 Combining Relational and Logical Operators

You can combine multiple relational operators with logical operators to form complex conditions.

Full runnable code:

```
#include <stdio.h>

int main() {
    int score = 85;
    int attendance = 90;

    if (score >= 80 && attendance >= 75) {
        printf("Passed the course.\n");
    } else {
        printf("Did not pass.\n");
    }

    // Check if score is outside a range
    if (score < 50 || attendance < 60) {
        printf("Fail due to low score or attendance.\n");
    }

    return 0;
```

```
}
```

3.2.4 Edge Cases and Common Pitfalls

- **Avoid using = instead of ==** in conditions. The = is an assignment operator and will assign a value instead of comparing. For example:

```
if (a = 5) {  // WRONG! Assigns 5 to a and always true
    // This block will always execute
}
```

Always use == for comparison:

```
if (a == 5) {  // Correct
    // Executes only if a equals 5
}
```

- **Short-circuit evaluation:** Logical AND (`&&`) and OR (`||`) operators perform **short-circuit evaluation**. This means:
  - In `A && B`, if `A` is false, `B` is **not evaluated** because the whole expression is already false.
  - In `A || B`, if `A` is true, `B` is **not evaluated** because the whole expression is already true.
- This behavior can be used to avoid errors, such as checking a pointer for `NULL` before dereferencing it.

3.2.5 Example: Multiple Conditions

Full runnable code:

```
#include <stdio.h>

int main() {
    int temp = 25;
    int isRaining = 0;

    if ((temp > 20 && temp < 30) && !isRaining) {
        printf("Nice day for a walk!\n");
    } else {
        printf("Better stay inside.\n");
    }

    return 0;
}
```

3.2.6 Summary

- **Relational operators** compare values and return true or false.
- **Logical operators** combine relational expressions to form compound conditions.
- Use `==` for equality check, **not** `=`.
- Use parentheses `()` to group expressions clearly.
- Logical operators perform short-circuit evaluation, improving efficiency.
- Combine conditions carefully to control program flow effectively.

With these operators, you can make your C programs make decisions, enabling dynamic and interactive behavior.

3.3 Assignment and Compound Assignment

In C programming, the **assignment operator** `=` is used to **store** a value into a variable. It is one of the most fundamental operators, as it allows you to give your variables values that your program can then use and manipulate.

3.3.1 The Basic Assignment Operator =

The syntax is straightforward:

```
variable = expression;
```

This means: **evaluate the expression on the right**, then **store the result** in the variable on the left.

Example:

```
int count;
count = 10;  // Assign 10 to count
```

The variable `count` now holds the value 10.

3.3.2 Beware: Assignment = vs Equality ==

A very common mistake beginners make is confusing the assignment operator `=` with the equality operator `==` used in comparisons.

- `=` assigns a value.
- `==` checks if two values are equal.

Example of wrong usage inside an `if` statement:

```c
if (count = 5) {  // Incorrect! This assigns 5 to count, not compare.
    // This block always executes because assignment
    //  returns the assigned value (5), which is true.
}
```

Correct comparison should be:

```c
if (count == 5) {
    // Executes only if count equals 5.
}
```

3.3.3 Compound Assignment Operators

C provides **compound assignment operators** that combine an arithmetic operation with assignment. They make code shorter and clearer by updating a variable **in place**.

Operator	Equivalent To	Meaning
`+=`	`x = x + y`	Add right operand to left operand and assign result to left operand
`-=`	`x = x - y`	Subtract right operand from left operand and assign the result
`*=`	`x = x * y`	Multiply left operand by right operand and assign the result
`/=`	`x = x / y`	Divide left operand by right operand and assign the result
`%=`	`x = x % y`	Assign remainder of left operand divided by right operand

3.3.4 Examples Using Compound Assignment

Instead of writing:

```c
int counter = 0;
counter = counter + 1;  // increment counter by 1
```

You can simply write:

```c
counter += 1;
```

Both do the same thing, but the compound operator is more concise.

3.3.5 Common Uses: Counters and Accumulators

Compound assignment operators are widely used in loops and data processing.

Example: Counting with +=:

```c
int sum = 0;
for (int i = 1; i <= 5; i++) {
    sum += i;  // Add i to sum each loop iteration
}
printf("Sum: %d\n", sum);  // Outputs 15 (1+2+3+4+5)
```

Example: Multiplying with *=:

```c
int product = 1;
for (int i = 1; i <= 4; i++) {
    product *= i;  // Multiply product by i each loop iteration
}
printf("Product: %d\n", product);  // Outputs 24 (1*2*3*4)
```

3.3.6 Compound Assignment with Other Types

Compound assignment operators also work with other data types such as floats:

```c
float balance = 1000.0f;
balance -= 250.0f;  // Withdraw 250
printf("Remaining balance: %.2f\n", balance);  // 750.00
```

3.3.7 Summary

- The assignment operator `=` stores the value of an expression in a variable.
- Compound assignment operators like `+=`, `-=`, `*=`, `/=`, and `%=` update variables by combining an arithmetic operation with assignment.
- They reduce redundancy and make code cleaner.
- Watch out for confusion between `=` (assignment) and `==` (comparison).
- Compound assignments are widely used for counters, accumulators, and in-place updates.

Mastering assignment and compound assignment helps write efficient, readable code and is a step toward more complex programming constructs like loops and functions.

3.4 Increment and Decrement

In C, two special operators — the **increment operator** `++` and the **decrement operator** `--` — provide a shorthand way to increase or decrease a variable's value by 1. These operators are widely used in loops, counters, and arithmetic expressions, making code more concise and readable.

3.4.1 What Are Increment and Decrement Operators?

- **Increment (`++`)** adds 1 to a variable.
- **Decrement (`--`)** subtracts 1 from a variable.

Both operators come in **two forms**:

- **Prefix:** `++variable` or `--variable` — The variable is modified *before* its value is used in the expression.
- **Postfix:** `variable++` or `variable--` — The variable's current value is used *first*, then it is incremented or decremented.

3.4.2 Prefix vs Postfix: How Do They Differ?

The main difference lies in **when the increment or decrement happens relative to the value being used** in an expression.

Example: Increment in Prefix Form

```
int a = 5;
int b = ++a;  // a is incremented first, then b is assigned

printf("a = %d, b = %d\n", a, b);
```

Output:

```
a = 6, b = 6
```

- `++a` increments `a` before its value is used.
- So `a` becomes 6, and then 6 is assigned to `b`.

Example: Increment in Postfix Form

```
int a = 5;
int b = a++;  // b is assigned the current value of a, then a is incremented

printf("a = %d, b = %d\n", a, b);
```

Output:

```
a = 6, b = 5
```

- `a++` uses the current value of `a` (which is 5) for the assignment to `b`.
- Then `a` is incremented to 6.

3.4.3 Practical Examples in Loops

The increment and decrement operators are most commonly seen in **loops** such as `for` loops.

Full runnable code:

```c
#include <stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {   // i++ increments i after each loop iteration
        printf("i = %d\n", i);
    }
    return 0;
}
```

This loop prints numbers from 0 to 4. The `i++` means after each iteration, the value of `i` is increased by 1.

You can also use the decrement operator to count down:

```c
for (int i = 5; i > 0; i--) {   // i-- decrements i after each iteration
    printf("i = %d\n", i);
}
```

This will print numbers from 5 down to 1.

3.4.4 Using Increment and Decrement in Expressions

Because prefix and postfix forms behave differently, understanding their effect inside expressions is important.

```c
int x = 3;
int y = x++ + 2;  // y = 3 + 2, then x becomes 4
printf("x = %d, y = %d\n", x, y);
```

Output:

```
x = 4, y = 5
```

Compare with prefix:

```c
int x = 3;
int y = ++x + 2;  // x becomes 4 first, then y = 4 + 2
printf("x = %d, y = %d\n", x, y);
```

Output:

```
x = 4, y = 6
```

3.4.5 Common Pitfalls

- **Confusing prefix and postfix in complex expressions** can lead to unexpected results.
- Avoid writing expressions where the same variable is modified multiple times without sequence points (e.g., `i = i++ + ++i;`) as this leads to undefined behavior.
- Remember that **in a statement on its own**, such as `i++;` or `++i;`, both forms will simply increment `i` by 1.

3.4.6 Increment/Decrement Without Using the Variable Immediately

If you just want to increase or decrease a variable without using its current value in the expression, the difference between prefix and postfix does not matter:

```
int count = 10;
count++;   // increment by 1
--count;   // decrement by 1
```

Both lines effectively update `count` by 1.

3.4.7 Summary

- `++` and `--` increase or decrease a variable by one, respectively.
- **Prefix (`++i` / `--i`)** changes the variable before the expression is evaluated.
- **Postfix (`i++` / `i--`)** uses the variable's original value first, then changes it.
- Most common use is in loops and counters.
- Be cautious using them in complex expressions to avoid confusing results or undefined behavior.

Mastering increment and decrement operators will help you write concise and efficient loops and expressions in C.

3.5 Type Conversion and Casting

In C programming, data often comes in different types — integers, floats, characters, and more. When you perform operations involving different data types, C automatically converts (or **promotes**) some types to others to make the expressions compatible. This process is called **implicit type conversion** or **type promotion**. Sometimes, however, you need to manually convert a value from one type to another using **explicit type casting**.

Understanding how type conversion and casting work is important to avoid unexpected results

and bugs.

3.5.1 Implicit Type Conversion (Type Promotion)

C automatically promotes smaller or less precise types to larger or more precise types during expressions, following a set of rules called the **usual arithmetic conversions**.

For example:

```
int a = 5;
float b = 2.0f;

float result = a + b;
printf("Result: %.2f\n", result);
```

Here, the integer `a` is **implicitly converted** to a float before addition, so the operation is `5.0 + 2.0`, resulting in `7.0`.

Key Points of Implicit Conversion:

- When an operation involves two different types, C converts the **lower-ranked type** to the higher-ranked type.
- Integer types (`char`, `short`) are promoted to `int` before arithmetic.
- Between `int` and `float`, `int` is promoted to `float`.
- Between `int` and `double`, `int` is promoted to `double`.

3.5.2 Why Does This Matter?

Implicit conversion helps keep expressions consistent and prevents some type errors. But it can also cause unintended results, especially with division.

3.5.3 Example: Integer Division vs. Floating-Point Division

Consider this:

```
int x = 7, y = 2;
float result = x / y;
printf("Result: %.2f\n", result);
```

You might expect `3.5`, but the output will be:

```
Result: 3.00
```

Why?

- Both `x` and `y` are integers, so integer division is performed first (`7 / 2 = 3`).
- The integer result `3` is then converted to float and stored in `result` as `3.0`.

3.5.4 Explicit Type Casting

To get the expected floating-point division result, you must **explicitly convert** one operand to float:

```
float result = (float)x / y;
printf("Result: %.2f\n", result);
```

Output:

```
Result: 3.50
```

Here, `(float)x` casts `x` to a float, so the division becomes floating-point division, and the result retains the decimal.

3.5.5 How to Use Type Casting

The syntax is:

```
(type) expression
```

- `type` is the data type you want to convert to.
- `expression` is the value or variable to be converted.

Example:

```
double pi = 3.14159;
int wholePart = (int)pi;   // wholePart will be 3
```

Casting `pi` to `int` truncates the decimal part.

3.5.6 Examples Showing Effects of Casting

1. **Casting before division:**

```
int a = 9, b = 4;

printf("Without cast: %d\n", a / b);          // Output: 2
printf("With cast: %.2f\n", (float)a / b);    // Output: 2.25
```

2. **Casting float to int (data loss):**

```c
float f = 5.9f;
int i = (int)f;

printf("Float: %.2f, Int after cast: %d\n", f, i);
// Output: Float: 5.90, Int after cast: 5
```

The decimal part is lost during the cast.

3.5.7 Important Warnings About Casting

- **Data Loss:** Casting from a larger to a smaller type (e.g., `double` to `int`) can truncate values or cause overflow.
- **Undefined Behavior:** Casting pointers incorrectly or forcing incompatible types can lead to crashes or unpredictable behavior.
- Always ensure that the cast makes sense logically and semantically.

3.5.8 Summary

- **Implicit conversion** happens automatically in expressions involving different data types.
- Smaller types are promoted to larger or more precise types according to C's rules.
- Integer division truncates decimals unless one operand is cast to float or double.
- **Explicit casting** lets you manually convert a value's type using `(type)`.
- Use casting carefully to avoid data loss and undefined behavior.

Understanding type conversion and casting is crucial for writing reliable, predictable C programs, especially when mixing data types in calculations and assignments.

Chapter 4.

Control Flow

1. `if`, `else`, and Nested Conditions
2. `switch` Statement
3. Loops: `for`, `while`, `do-while`
4. `break`, `continue`, and `goto` (with caution)

4 Control Flow

4.1 `if`, `else`, and Nested Conditions

Controlling the flow of a program is essential to making it **dynamic** and **responsive**. In C, this control is often achieved through **conditional statements**. The primary conditional constructs are `if`, `else if`, and `else`. These allow your program to execute different blocks of code based on certain conditions — decisions like "if this is true, do this; otherwise, do something else."

4.1.1 The Basic `if` Statement

The `if` statement evaluates a condition inside parentheses. If the condition is **true** (non-zero), the code block immediately following it executes.

Syntax:

```
if (condition) {
    // code to execute if condition is true
}
```

Example:

```
int number = 10;

if (number > 0) {
    printf("The number is positive.\n");
}
```

If `number` is greater than 0, the message prints.

4.1.2 The `else` Clause

The `else` clause provides an alternative block of code that runs **only if** the `if` condition is false.

Syntax:

```
if (condition) {
    // executes if condition is true
} else {
    // executes if condition is false
}
```

Example:

```
int number = -5;
```

```c
if (number > 0) {
    printf("Positive number.\n");
} else {
    printf("Non-positive number.\n");
}
```

4.1.3 The `else if` Ladder

When multiple conditions need to be checked sequentially, use `else if`. It allows you to test many alternatives one by one.

Syntax:

```c
if (condition1) {
    // executes if condition1 is true
} else if (condition2) {
    // executes if condition1 is false and condition2 is true
} else {
    // executes if all above conditions are false
}
```

Example: Grading System

```c
int score = 85;

if (score >= 90) {
    printf("Grade: A\n");
} else if (score >= 80) {
    printf("Grade: B\n");
} else if (score >= 70) {
    printf("Grade: C\n");
} else if (score >= 60) {
    printf("Grade: D\n");
} else {
    printf("Grade: F\n");
}
```

This example assigns letter grades based on numeric scores.

4.1.4 Nested Conditions

You can place an `if` or `else` block **inside** another `if` or `else`. This is called **nesting** and is useful for complex decision-making.

Example: Eligibility Check

```c
int age = 20;
int hasID = 1;  // 1 means true, 0 false

if (age >= 18) {
```

```
    if (hasID) {
        printf("Allowed to enter.\n");
    } else {
        printf("ID required for entry.\n");
    }
} else {
    printf("Not allowed to enter due to age.\n");
}
```

Here, the program first checks age, then inside that block, it checks if the person has ID.

4.1.5 Practical Example: Menu Selection

```
int choice = 2;

if (choice == 1) {
    printf("You selected option 1.\n");
} else if (choice == 2) {
    printf("You selected option 2.\n");
} else if (choice == 3) {
    printf("You selected option 3.\n");
} else {
    printf("Invalid selection.\n");
}
```

4.1.6 Common Mistakes to Avoid

1. **Missing Braces {}**

Even though C allows omitting braces if the block has only one statement, it's best practice to always include them. Omitting braces can lead to bugs that are hard to spot.

Incorrect:

```
if (x > 0)
    printf("Positive\n");
    printf("This always prints.\n");  // runs regardless of condition
```

Correct:

```
if (x > 0) {
    printf("Positive\n");
    printf("This prints only if x > 0.\n");
}
```

2. **Misplaced Semicolons**

A semicolon immediately after the `if` condition ends the `if` prematurely:

```c
if (x > 0);  // <- semicolon here ends if statement
{
    printf("This always prints.\n");
}
```

3. **Indentation and Readability**

Proper indentation does not affect how the program runs but is vital for human readers.

Good style:

```c
if (x > 0) {
    printf("Positive\n");
} else {
    printf("Non-positive\n");
}
```

Poor style:

```c
if (x > 0) {
printf("Positive\n");}
else {printf("Non-positive\n");}
```

4.1.7 Summary

- Use `if` to execute code conditionally.
- Use `else if` for multiple conditions.
- Use `else` for a default fallback.
- Nest conditions for complex logic.
- Always use braces `{}` to clearly define blocks.
- Keep code readable with consistent indentation.
- Avoid misplaced semicolons that break conditional logic.

Mastering these conditional statements enables you to write programs that **respond to different situations intelligently**, which is crucial for real-world applications.

4.2 `switch` Statement

When your program needs to select between multiple possible discrete values, writing a long chain of `if-else` statements can become cumbersome and harder to read. C provides a cleaner alternative for such situations: the **`switch` statement**.

4.2.1 What is a `switch` Statement?

A `switch` statement allows you to compare a single expression against a set of constant values and execute the code corresponding to the matching case. It is ideal when you have multiple possible values for one variable and want to take different actions based on those values.

4.2.2 Syntax of `switch`

```
switch (expression) {
    case constant1:
        // code to execute if expression == constant1
        break;

    case constant2:
        // code to execute if expression == constant2
        break;

    // ... more cases ...

    default:
        // code to execute if none of the cases match
}
```

- The `expression` is evaluated once.
- The program compares the result to each `case` label.
- If a match is found, execution starts from that case.
- The `break` statement prevents execution from "falling through" to subsequent cases.
- The `default` case is optional and runs if no other case matches.

4.2.3 How Does `switch` Work?

When the value of the `expression` matches a `case` label, the statements following that label execute until a `break` is encountered or the switch ends.

4.2.4 Practical Example: Menu Selection

Full runnable code:

```
#include <stdio.h>

int main() {
    int choice;
```

```
    printf("Menu:\n");
    printf("1. Add\n2. Subtract\n3. Multiply\n4. Divide\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("You chose Add.\n");
            break;
        case 2:
            printf("You chose Subtract.\n");
            break;
        case 3:
            printf("You chose Multiply.\n");
            break;
        case 4:
            printf("You chose Divide.\n");
            break;
        default:
            printf("Invalid choice.\n");
            break;
    }

    return 0;
}
```

If the user enters `3`, the output will be:

```
You chose Multiply.
```

4.2.5 Importance of `break`

Without `break` statements, C's `switch` exhibits **fall-through** behavior, where execution continues into the next case regardless of matching. Sometimes this is useful, but often it leads to bugs if unintentional.

Example without `break`:

```
int number = 2;

switch (number) {
    case 1:
        printf("One\n");
    case 2:
        printf("Two\n");
    case 3:
        printf("Three\n");
        break;
    default:
        printf("Other\n");
}
```

Output:

```
Two
Three
```

Since `case 2` matched and there was no `break`, execution continued through `case 3`.

4.2.6 Using `default`

The `default` case handles situations where none of the cases match the expression. It is similar to the `else` in an `if-else` chain and ensures your program can handle unexpected inputs gracefully.

4.2.7 When to Use `switch` vs `if-else`

- **Use `switch` when:**
  - You have a single variable that can take multiple discrete integral values.
  - You want clearer, more readable code than long `if-else` chains.
  - You want to leverage fall-through behavior intentionally.
- **Use `if-else` when:**
  - Conditions involve ranges or complex expressions (e.g., `x > 10 && y < 5`).
  - You need to compare floating-point numbers or strings (since `switch` supports only integral types).
  - Your conditions are not constant values.

4.2.8 Limitations of `switch`

- The `switch` expression must evaluate to an **integral type** (`int`, `char`, `enum`).
- Floating-point values and strings **cannot** be used directly.
- Case labels must be **constant expressions** known at compile time.

4.2.9 Another Example: Simple Calculator

Full runnable code:

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

```c
int main() {
    char op;
    int a, b;

    printf("Enter operator (+, -, *, /): ");
    scanf(" %c", &op);

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    switch (op) {
        case '+':
            printf("Result: %d\n", a + b);
            break;
        case '-':
            printf("Result: %d\n", a - b);
            break;
        case '*':
            printf("Result: %d\n", a * b);
            break;
        case '/':
            if (b != 0)
                printf("Result: %d\n", a / b);
            else
                printf("Error: Division by zero.\n");
            break;
        default:
            printf("Invalid operator.\n");
    }

    return 0;
}
```

4.2.10 Summary

- `switch` is a clean alternative to multiple `if-else` statements for discrete values.
- Use `case` labels to define possible values, and `break` to prevent fall-through.
- `default` handles unmatched cases.
- Best for integral types with constant values.
- Avoid for complex or range-based conditions.

Understanding the `switch` statement helps you write more organized and efficient decision-making code, especially when handling menus, commands, or multiple fixed choices.

4.3 Loops: `for`, `while`, `do-while`

Loops are fundamental constructs that allow a program to **repeat a block of code** multiple times. This repetition can be controlled by a condition and helps automate tasks like counting, accumulating values, or repeatedly asking for user input until valid data is entered.

In C, there are three primary loop types:

- `for` loop
- `while` loop
- `do-while` loop

Each has its own syntax and behavior suited for different scenarios.

4.3.1 The `for` Loop

The `for` loop is typically used when the number of iterations is known beforehand, or you want a concise loop control with initialization, condition, and update all in one place.

Syntax:

```
for (initialization; condition; update) {
    // code to repeat
}
```

- **Initialization:** Executed once at the beginning.
- **Condition:** Evaluated before each iteration; if false, the loop ends.
- **Update:** Executed after each iteration.

Example: Counting from 1 to 5

Full runnable code:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        printf("i = %d\n", i);
    }
    return 0;
}
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

4.3.2 The `while` Loop

The `while` loop checks the condition **before** each iteration and repeats the block as long as the condition is true. It's useful when the number of iterations is not known in advance, and the loop should continue until a certain condition changes.

Syntax:

```
while (condition) {
    // code to repeat
}
```

Example: Summing numbers until total exceeds 100

Full runnable code:

```
#include <stdio.h>

int main() {
    int sum = 0, num = 0;

    while (sum <= 100) {
        printf("Enter a number: ");
        scanf("%d", &num);
        sum += num;
        printf("Sum so far: %d\n", sum);
    }

    printf("Total sum exceeded 100.\n");
    return 0;
}
```

4.3.3 The `do-while` Loop

The `do-while` loop is similar to the `while` loop but guarantees that the loop body executes **at least once**, because the condition is checked **after** the body runs.

Syntax:

```
do {
    // code to repeat
} while (condition);
```

Example: Input validation

Full runnable code:

```
#include <stdio.h>

int main() {
    int number;
```

```c
    do {
        printf("Enter a positive number: ");
        scanf("%d", &number);
    } while (number <= 0);

    printf("You entered: %d\n", number);
    return 0;
}
```

In this example, the prompt appears at least once, and repeats until the user enters a positive number.

4.3.4 Key Differences in Control Flow

Loop Type	Condition Checked	When Loop Body Runs at Least Once?	Typical Use Case
`for`	Before each iteration	Only if condition true	Counting loops, known iterations
`while`	Before each iteration	Only if condition true	Unknown number of iterations, sentinel loops
`do-while`	After each iteration	Always	Input validation, menus requiring at least one run

4.3.5 When to Choose Which Loop?

- Use **`for` loops** when you know exactly how many times you want to repeat something.
- Use **`while` loops** when the loop should continue based on a condition that might not be linked to a counter.
- Use **`do-while` loops** when the body must run at least once regardless of the condition.

4.3.6 Infinite Loop Pitfalls

An **infinite loop** occurs when the loop's condition never becomes false. This can happen due to:

- Forgetting to update the loop variable.
- Incorrect loop condition.
- Logic errors inside the loop.

Example of an infinite loop:

```c
int i = 0;

while (i < 5) {
    printf("%d\n", i);
    // Missing i++ causes infinite loop!
}
```

To avoid infinite loops:

- Always ensure the loop condition can become false.
- Update loop counters properly.
- Use debugging prints or tools if a loop behaves unexpectedly.

4.3.7 Summary

- Loops enable repeating code blocks efficiently.
- `for` loops are great for counting and known iterations.
- `while` loops are best for unknown iterations and conditions.
- `do-while` loops guarantee the body runs at least once.
- Careful loop design prevents infinite loops and logic errors.

Mastering these loops will help you automate repetitive tasks and control your program's flow precisely.

4.4 `break`, `continue`, and `goto` (with caution)

When working with loops and conditional branches, you often need finer control over the flow of your program. C provides three statements — `break`, `continue`, and `goto` — that allow you to alter the natural progression of loops or blocks in specific ways. Understanding these statements helps write more flexible programs but also calls for caution, especially with `goto`.

4.4.1 The `break` Statement

The `break` statement is used to **exit a loop or a `switch` statement immediately**, regardless of whether the loop condition is still true or not. When a `break` executes inside a loop, control jumps to the statement immediately following the loop.

`break` in Loops

Suppose you want to search an array and stop once you find a match. Instead of looping through the entire array unnecessarily, you can use `break` to exit early.

Full runnable code:

```c
#include <stdio.h>

int main() {
    int numbers[] = {2, 4, 6, 8, 10};
    int size = 5;
    int target = 6;
    int found = 0;

    for (int i = 0; i < size; i++) {
        if (numbers[i] == target) {
            found = 1;
            break;  // Exit loop immediately after finding target
        }
    }

    if (found) {
        printf("Number %d found!\n", target);
    } else {
        printf("Number %d not found.\n", target);
    }

    return 0;
}
```

`break` in `switch` Statements

The `break` statement prevents **fall-through**, where execution continues from one `case` to the next unintentionally. Without `break`, all subsequent cases run after the matched case.

```c
char grade = 'B';

switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break;
    case 'B':
        printf("Good job.\n");
        break;
    default:
        printf("Keep trying.\n");
}
```

Here, each `case` ends with a `break` to stop further execution.

4.4.2 The `continue` Statement

`continue` tells the loop to **skip the rest of the current iteration** and move directly to the next iteration.

`continue` Example

Suppose you want to print only even numbers from 1 to 10, skipping odd numbers.

Full runnable code:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 != 0) {
            continue;  // Skip odd numbers
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
2 4 6 8 10
```

When `i` is odd, `continue` skips the `printf` and proceeds with the next loop cycle.

4.4.3 The `goto` Statement (Use With Caution)

`goto` is a **jump statement** that transfers control to another labeled part of the code. Unlike structured control flow, `goto` can jump **anywhere** in the function, which can lead to **"spaghetti code"** — tangled, hard-to-read code that's difficult to maintain or debug.

Syntax of `goto`

```c
goto label;

...

label:
// some code here
```

Example of `goto`

Full runnable code:

```c
#include <stdio.h>

int main() {
    int i = 0;

start_loop:
    if (i >= 5) {
        goto end_loop;
    }
```

```
    printf("%d ", i);
    i++;
    goto start_loop;

end_loop:
    printf("\nLoop ended.\n");
    return 0;
}
```

This code prints numbers 0 to 4 by jumping between labels.

4.4.4 Why Should You Avoid or Use `goto` Sparingly?

- **Reduces readability:** Jumps break the normal flow of execution, making the code harder to follow.
- **Complicates maintenance:** Future changes become difficult as you must track all possible jumps.
- **Increases bugs:** Unexpected jumps can skip variable initialization or cleanup.
- **Better alternatives:** Loops, functions, and control statements like `break` and `continue` usually cover all use cases cleanly.

4.4.5 When Is `goto` Sometimes Useful?

- Error handling and cleanup in deeply nested code, especially in C programs that manually manage resources.
- Situations where avoiding duplicated cleanup code is cumbersome without `goto`.

Even in such cases, it's best to use `goto` carefully and document its purpose.

4.4.6 Summary

State-ment	Purpose	Typical Use Case
`break`	Exit a loop or `switch` early	Stop searching once found, prevent fall-through
`continue`	Skip current loop iteration, go to next	Skip unwanted iterations (e.g., filtering)
`goto`	Jump to labeled code anywhere	Rare cases like error handling; use sparingly

Use `break` and `continue` to write clearer, more efficient loops. Avoid `goto` unless absolutely necessary, and even then, use it with care to preserve code clarity and maintainability.

Chapter 5.

Functions in C

1. Defining and Calling Functions
2. Function Arguments and Return Values
3. Scope and Lifetime of Variables
4. Recursion Basics

5 Functions in C

5.1 Defining and Calling Functions

Functions are one of the fundamental building blocks of C programming. They allow you to group a set of instructions into reusable blocks that perform specific tasks. Using functions not only makes your code organized and easier to maintain, but also lets you avoid repetition by calling the same block of code multiple times.

5.1.1 What Is a Function?

A **function** in C is a named block of code that can take inputs, perform actions, and optionally return a value. Functions help break down complex programs into smaller, manageable pieces.

5.1.2 Function Definition Syntax

The general syntax for defining a function is:

```
return_type function_name(parameter_list) {
    // function body (code to execute)
}
```

- **`return_type`**: Specifies the type of value the function will return to the caller (e.g., `int`, `float`, `void`).
- **`function_name`**: The identifier used to call the function.
- **`parameter_list`**: A comma-separated list of input variables (parameters) with their types. Can be empty if the function takes no parameters.
- **Function body**: The block of code inside `{}` that runs when the function is called.

5.1.3 Example 1: A Simple Function That Adds Two Numbers

Let's define a function that takes two integers, adds them, and returns the sum:

```
int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Here:

- `int` before `add` means the function returns an integer.
- `add` is the function name.

- `(int a, int b)` means the function takes two integer parameters named `a` and `b`.
- The function calculates the sum and returns it using the `return` keyword.

5.1.4 Calling a Function

To use a function, you **call** it from another part of the program, such as from the `main()` function.

Full runnable code:

```c
#include <stdio.h>

// Function definition
int add(int a, int b) {
    return a + b;
}

int main() {
    int result;

    // Function call
    result = add(5, 3);

    printf("Sum is %d\n", result);
    return 0;
}
```

What happens when `add(5, 3)` is called?

1. The program pauses executing `main`.
2. Control passes to the `add` function, with parameters `a=5` and `b=3`.
3. The function executes its code, calculates `5 + 3`, and returns `8`.
4. Control goes back to `main`, where `result` receives the returned value `8`.
5. The program prints "Sum is 8" and continues.

5.1.5 `void` Functions: Functions That Do Not Return Values

Sometimes, functions just perform an action without returning data. Such functions use the `void` return type.

Example 2: Function to Print a Message

Full runnable code:

```c
#include <stdio.h>

void greet() {
```

```
    printf("Hello from a function!\n");
}

int main() {
    greet();  // Calling the greet function
    return 0;
}
```

Since `greet` has `void` as the return type and no parameters, it simply prints a message when called.

5.1.6 Function Prototypes

Before you call a function in your program, the compiler must know about its **signature** — the return type, name, and parameters. This is often done using a **function prototype**, which is a declaration of the function placed before `main()` or at the top of your file.

Prototype syntax:

```
return_type function_name(parameter_types);
```

Example:

```
int add(int a, int b);  // Function prototype

int main() {
    int sum = add(10, 20);
    printf("Sum: %d\n", sum);
    return 0;
}

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

If you omit the prototype and call the function before it's defined, the compiler may give an error or warning.

5.1.7 Organizing Code into Reusable Blocks

Functions help you organize your program into **modular pieces**. Instead of writing all code inside `main()`, you divide logic into functions with specific tasks. This makes code:

- Easier to read
- Easier to debug
- Easier to maintain and update
- Reusable in other programs or parts of the program

5.1.8 Example: Multiple Function Calls

Full runnable code:

```c
#include <stdio.h>

void printWelcome() {
    printf("Welcome to the program!\n");
}

int multiply(int x, int y) {
    return x * y;
}

int main() {
    printWelcome();

    int result = multiply(4, 5);
    printf("4 times 5 is %d\n", result);

    result = multiply(7, 3);
    printf("7 times 3 is %d\n", result);

    return 0;
}
```

Here, the same `multiply` function is called twice with different arguments, demonstrating how functions can be reused.

5.1.9 Summary

- A function groups code with a name, parameters, and return type.
- Functions help you write modular, maintainable programs.
- You define a function and call it from other parts of your program.
- The `return` keyword sends a result back to the caller.
- `void` functions perform actions without returning a value.
- Function prototypes declare functions before use, helping the compiler.
- Using functions improves code clarity, reuse, and debugging.

Mastering function definitions and calls is a critical step toward writing effective C programs. Next, you'll learn how to pass arguments and handle return values in more detail.

5.2 Function Arguments and Return Values

Functions become powerful when they accept inputs (arguments) and return outputs (return values). Understanding how arguments are passed and how return values work in C is essential

for writing effective and reusable code.

5.2.1 How Are Arguments Passed in C?

In C, **arguments are passed to functions by value**. This means the function receives a **copy** of the actual data, not the original variable itself.

Example:

Full runnable code:

```c
#include <stdio.h>

void increment(int x) {
    x = x + 1;
    printf("Inside increment: %d\n", x);
}

int main() {
    int num = 5;
    increment(num);
    printf("After increment: %d\n", num);
    return 0;
}
```

Output:

```
Inside increment: 6
After increment: 5
```

Explanation: The `increment` function modifies its local copy of `x`. The original `num` in `main` remains unchanged because only a copy was passed.

5.2.2 Passing Multiple Arguments

Functions can take multiple parameters separated by commas. Each argument is passed by value independently.

Full runnable code:

```c
#include <stdio.h>

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

int main() {
    int sum = add(10, 20);
```

```
    printf("Sum: %d\n", sum);
    return 0;
}
```

5.2.3 Return Values

Functions return values to the caller using the `return` statement. The returned value replaces the function call expression.

- Functions can return one value of any data type (except `void` functions, which return nothing).
- The return type must match the function's declared return type.

```
float multiply(float x, float y) {
    return x * y;
}
```

5.2.4 Limitations of Pass-By-Value

Since C passes arguments by value, the called function cannot modify the caller's variables directly. This is sometimes limiting when you want to update variables within a function.

5.2.5 Modifying Variables via Pointers

To modify a variable's value inside a function, you pass its **address** (a pointer) and use **indirection**.

Example:

Full runnable code:

```
#include <stdio.h>

void increment(int *ptr) {
    (*ptr)++;  // Dereference and increment the value at address ptr
}

int main() {
    int num = 5;
    increment(&num);  // Pass address of num
    printf("After increment: %d\n", num);
    return 0;
}
```

Output:

```
After increment: 6
```

Explanation: The `increment` function receives a pointer to `num`, dereferences it, and modifies the actual variable.

5.2.6 Returning Pointers from Functions

Functions can also return pointers, but you must be careful **not to return pointers to local variables**, which get destroyed once the function ends.

Unsafe example:

```c
int* unsafeFunction() {
    int x = 10;
    return &x;  // x is local; pointer becomes invalid after return
}
```

Safe alternative:

- Return pointers to dynamically allocated memory (`malloc`), or
- Pass pointers as function arguments to store results.

5.2.7 Returning Structs

C allows returning **structs** by value. This is useful when you want to return multiple related values packaged as a single entity.

Full runnable code:

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} Point;

Point createPoint(int x, int y) {
    Point p;
    p.x = x;
    p.y = y;
    return p;
}

int main() {
    Point pt = createPoint(5, 10);
    printf("Point: (%d, %d)\n", pt.x, pt.y);
    return 0;
```

```
}
```

Returning structs by value is straightforward but can be inefficient for very large structs, in which case pointers may be preferred.

5.2.8 Best Practices

- Use pass-by-value for simple data types (e.g., `int`, `float`, `char`) when you do not need to modify the caller's variable.
- Use pointers to pass large data structures or when you want the function to modify the caller's variable.
- Avoid returning pointers to local variables.
- Use `const` with pointer parameters when the function should not modify the data (e.g., `const int *ptr`).
- For multiple related return values, consider returning a `struct` or using pointer parameters.

5.2.9 Summary

Concept	Explanation	Example Use Case
Pass-by-value	Arguments passed as copies	Simple calculations without side effects
Pass-by-pointer	Pass address to modify original data	Update variables inside functions
Return values	Functions return a single value	Return results like sums or averages
Return pointers	Functions return memory addresses (careful!)	Return dynamically allocated data
Return structs	Return multiple values grouped together	Returning complex data like coordinates

5.2.10 Quick Example Combining These Concepts

Full runnable code:

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

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    printf("Before swap: x=%d, y=%d\n", x, y);
    swap(&x, &y);
    printf("After swap: x=%d, y=%d\n", x, y);
    return 0;
}
```

Output:

```
Before swap: x=5, y=10
After swap: x=10, y=5
```

Understanding how to pass arguments and return values properly lets you design flexible, efficient functions that communicate effectively within your program.

5.3 Scope and Lifetime of Variables

Understanding **scope** and **lifetime** of variables is crucial to writing reliable and maintainable C programs. These concepts determine **where a variable can be accessed** in your code and **how long the variable exists in memory** during program execution.

5.3.1 What Is Variable Scope?

Scope defines the region of your program where a variable is visible and accessible. Variables can have:

- **Local scope**
- **Global scope**
- **File scope**

Local Scope (Block Scope)

Variables declared inside a function or a block `{ ... }` are **local variables**. They exist only within that block and are not accessible outside it.

Full runnable code:

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

```c
void example() {
    int localVar = 10;  // local to example()
    printf("Inside example(): %d\n", localVar);
}

int main() {
    // localVar is not visible here - this would cause an error:
    // printf("%d", localVar);

    example();
    return 0;
}
```

Within the function `example()`, `localVar` exists and can be used. Outside, it's unknown and inaccessible.

Local variables have **block scope** — they are confined to the nearest enclosing `{}` braces where they are declared.

Global Scope

Variables declared **outside all functions**, usually at the top of the file, have **global scope**. They are accessible from any function within the same file (and even other files, depending on how they're declared).

Full runnable code:

```c
#include <stdio.h>

int globalVar = 100;  // global variable

void printGlobal() {
    printf("Global variable: %d\n", globalVar);
}

int main() {
    printGlobal();
    printf("Access from main(): %d\n", globalVar);
    return 0;
}
```

Because `globalVar` is global, both `main()` and `printGlobal()` can access it.

File Scope and `static` Globals

By default, global variables have **external linkage**, meaning other files can access them if declared with `extern`. To restrict a global variable's scope to the current file only, use the `static` keyword:

```c
static int fileScopedVar = 50;
```

This **limits the variable's scope to the file**, helping avoid naming conflicts with other files in larger projects.

5.3.2 Variable Lifetime (Storage Duration)

While scope governs **where** variables are accessible, **lifetime** governs **how long** the variable exists in memory during program execution.

Automatic Storage Duration (Local Variables)

Local variables declared inside functions without `static` keyword have **automatic storage duration**:

- They are **created when the function/block is entered**.
- They are **destroyed when the function/block exits**.
- They are **not initialized automatically**, so they contain garbage values unless explicitly initialized.

```
void foo() {
    int count = 0;  // automatic variable
    count++;
    printf("Count: %d\n", count);
}
```

Each time `foo()` runs, `count` is freshly created and reset to 0.

Static Storage Duration

Variables declared with the `static` keyword inside functions or globally have **static storage duration**:

- They exist **for the entire duration of the program**.
- Inside a function, a `static` local variable **retains its value between calls**.
- Globally, `static` limits scope to the file but lifetime remains for the entire program.

Example: Static local variable inside a function

Full runnable code:

```
#include <stdio.h>

void counter() {
    static int count = 0;  // initialized only once
    count++;
    printf("Static count: %d\n", count);
}

int main() {
    counter();  // prints 1
    counter();  // prints 2
    counter();  // prints 3
    return 0;
}
```

Here, the `count` variable is **initialized only once**, and its value persists between function calls.

5.3.3 Summary Table: Scope vs. Lifetime

Variable Type	Scope	Lifetime	Storage Duration	Example Use Case
Local automatic	Block/function	Exists during block execution	Automatic	Temporary counters, loop variables
Local static	Block/function	Entire program execution	Static	Persistent counters inside functions
Global (extern)	Entire program	Entire program execution	Static (by default)	Shared constants, flags, configuration
Global static	Current file only	Entire program execution	Static	File-private globals to avoid conflicts

5.3.4 Best Practices

- **Minimize the use of global variables:** They can lead to unexpected side effects and make debugging harder.
- Use **local variables whenever possible** to keep your functions independent.
- Use `static` for variables that need to **retain state inside a function** but should not be visible outside.
- Use **file-scoped `static` globals** to limit variable visibility to a single file and prevent naming collisions in large projects.
- Always **initialize your variables**, especially static locals (which are initialized once) and globals (initialized to zero by default).
- Keep variable names **meaningful and consistent** to avoid confusion, especially when scope overlaps.

5.3.5 Example Illustrating Scope and Lifetime

Full runnable code:

```c
#include <stdio.h>

int globalVar = 10;       // Global variable (global scope, static lifetime)

void func() {
    // Local automatic variable (local scope, automatic lifetime)
    int localVar = 20;

    // Local static variable (local scope, static lifetime)
    static int staticVar = 30;
```

```c
    printf("Inside func(): globalVar=%d, localVar=%d, staticVar=%d\n",
           globalVar, localVar, staticVar);

    localVar++;
    staticVar++;
}

int main() {
    func();  // Prints 10,20,30
    func();  // Prints 10,20,31 - localVar reset, staticVar incremented
    func();  // Prints 10,20,32
    return 0;
}
```

Output:

```
Inside func(): globalVar=10, localVar=20, staticVar=30
Inside func(): globalVar=10, localVar=20, staticVar=31
Inside func(): globalVar=10, localVar=20, staticVar=32
```

Notice how `localVar` is reset every call, but `staticVar` remembers its value across calls.

5.3.6 Conclusion

- **Scope** defines where variables can be accessed.
- **Lifetime** defines how long variables exist.
- Local variables typically have limited scope and lifetime.
- Global and static variables persist for the program duration but differ in accessibility.
- Proper use of scope and lifetime leads to clean, efficient, and maintainable C programs.

Understanding these concepts helps prevent bugs related to variable visibility and unintended value changes, leading to safer and clearer code.

5.4 Recursion Basics

Recursion is a powerful programming technique where a function **calls itself** to solve smaller, simpler instances of the same problem. It allows complex problems to be broken down into easier subproblems, which can lead to elegant and concise code.

5.4.1 What Is Recursion?

In recursion, a function repeatedly invokes itself with modified arguments until it reaches a condition known as the **base case**, which stops further recursive calls. The function then unwinds, returning results back through the chain of calls.

5.4.2 Anatomy of a Recursive Function

A well-designed recursive function has two key components:

1. **Base Case:** The simplest scenario that can be solved directly without further recursion. This case stops the recursion.
2. **Recursive Case:** The part where the function calls itself with arguments that bring it closer to the base case.

5.4.3 Example 1: Calculating Factorial Using Recursion

The **factorial** of a non-negative integer `n`, written as `n!`, is the product of all positive integers up to `n`. For example:

- `5! = 5 × 4 × 3 × 2 × 1 = 120`
- `0! = 1` by definition

The factorial can be defined recursively as:

- `0! = 1` (base case)
- `n! = n × (n - 1)!` (recursive case)

C implementation:

Full runnable code:

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {          // Base case
        return 1;
    } else {               // Recursive case
        return n * factorial(n - 1);
    }
}

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

```
}
```

Output:

```
Factorial of 5 is 120
```

5.4.4 How the Recursive Calls Work

For `factorial(5)`, the call sequence is:

```
factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
= 5 * (4 * (3 * (2 * (1 * factorial(0)))))
= 5 * (4 * (3 * (2 * (1 * 1))))  // base case reached
= 120
```

Each recursive call waits for the next to return, then multiplies by `n`.

5.4.5 Example 2: Fibonacci Numbers

The Fibonacci sequence is another classic example:

- `Fib(0) = 0` (base case)
- `Fib(1) = 1` (base case)
- For `n > 1`: `Fib(n) = Fib(n - 1) + Fib(n - 2)` (recursive case)

C code:

Full runnable code:

```c
#include <stdio.h>

int fibonacci(int n) {
    if (n == 0) {
        return 0;    // Base case
    } else if (n == 1) {
        return 1;    // Base case
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case
    }
}

int main() {
```

```
    int n = 7;
    printf("Fibonacci number %d is %d\n", n, fibonacci(n));
    return 0;
}
```

Output:

```
Fibonacci number 7 is 13
```

5.4.6 Common Pitfalls of Recursion

While recursion is elegant, it can lead to problems if not implemented carefully:

- **Infinite Recursion:** If the base case is missing or never reached, the function calls itself endlessly, causing a **stack overflow**.
- **Stack Overflow:** Each recursive call uses memory on the call stack. Deep or infinite recursion can exhaust stack space, crashing the program.

5.4.7 Preventing Infinite Recursion

Always ensure:

- Your base case(s) are reachable.
- Recursive calls modify parameters to move toward the base case.

5.4.8 Recursion vs. Iteration

Many problems solved recursively can also be solved using **loops (iteration)**.

- **Clarity:** Recursive solutions often mirror the mathematical definition and are easier to understand conceptually.
- **Performance:** Recursive functions can be less efficient due to overhead of many function calls and use of stack memory. Iterative versions are generally faster and use less memory.

5.4.9 Example: Iterative Factorial

```c
int factorial_iterative(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

This loop avoids the overhead of recursive calls.

5.4.10 When to Use Recursion?

- When the problem has a **natural recursive structure**, such as:
  - Tree traversals
  - Divide and conquer algorithms (e.g., quicksort, mergesort)
  - Mathematical sequences (factorials, Fibonacci)
  - Backtracking problems (puzzles, permutations)
- When clarity and simplicity are more important than raw performance.

5.4.11 Tail Recursion Optimization (Advanced Note)

Some compilers optimize **tail-recursive** functions (where the recursive call is the last operation), turning them into efficient loops internally. However, this is not guaranteed in C, so iterative solutions are often preferred in performance-critical code.

5.4.12 Summary

Aspect	Recursion	Iteration
Code clarity	Mirrors problem definition, easier to read	May be less intuitive for some problems
Memory use	Uses stack for each call	Uses fixed memory for loop variables
Performance	Slower due to call overhead	Faster, less overhead
Risk	Stack overflow if uncontrolled	Generally safe

5.4.13 Final Thoughts

Recursion is a valuable tool that every C programmer should understand. It offers elegant solutions for many problems but requires care to avoid infinite loops and stack overflows. When used properly, recursion can simplify complex logic and serve as a foundation for advanced programming techniques.

Chapter 6.

Arrays and Strings

1. Single and Multi-Dimensional Arrays
2. String Manipulation with Character Arrays
3. Common String Functions (`strlen`, `strcpy`, `strcmp`, etc.)
4. Array of Strings

6 Arrays and Strings

6.1 Single and Multi-Dimensional Arrays

Arrays are fundamental data structures in C that allow you to store multiple values of the same type in a contiguous block of memory. They enable efficient handling of collections of data, such as lists of numbers, characters, or more complex datasets like tables.

6.1.1 Single-Dimensional Arrays

A **single-dimensional array** (often called a one-dimensional array) is like a list or a sequence of elements indexed by a single integer.

Declaration

To declare an array, you specify the type of elements, the array name, and the number of elements it will hold inside square brackets:

```
int scores[5];
```

This declares an array named `scores` that can hold 5 integers.

Initialization

You can initialize an array at the time of declaration using curly braces `{}` with a list of values:

```
int scores[5] = {90, 85, 78, 92, 88};
```

If you provide fewer initializers than the declared size, the remaining elements are automatically set to zero:

```
int numbers[5] = {1, 2};  // numbers = {1, 2, 0, 0, 0}
```

Alternatively, you can let the compiler count the elements by omitting the size:

```
int scores[] = {90, 85, 78, 92, 88};  // size inferred as 5
```

Accessing Elements

Array elements are accessed using their index, starting at 0:

```
printf("First score: %d\n", scores[0]);  // Outputs 90
printf("Third score: %d\n", scores[2]);  // Outputs 78
```

6.1.2 Iterating Over Arrays

Loops are commonly used to process arrays. Here's an example using a `for` loop to print all elements in the `scores` array:

```
for (int i = 0; i < 5; i++) {
    printf("Score %d: %d\n", i, scores[i]);
}
```

6.1.3 Multi-Dimensional Arrays

Multi-dimensional arrays extend the concept of arrays to multiple dimensions, the most common being two-dimensional arrays (2D arrays), which you can think of as tables or matrices.

Declaration

A 2D array is declared by specifying sizes for two dimensions:

```
int matrix[3][4];  // 3 rows, 4 columns
```

This declares a 3x4 matrix of integers.

Initialization

You can initialize a 2D array with nested curly braces, where each inner brace initializes one row:

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

You can also omit some initializers to have the remaining elements zero-initialized.

Accessing Elements

Elements in a 2D array are accessed using two indices: one for the row and one for the column.

```
printf("Element at row 2, column 3: %d\n", matrix[1][2]);  // Outputs 7
```

6.1.4 Iterating Over Multi-Dimensional Arrays

To process all elements in a 2D array, you typically use **nested loops**:

```
for (int i = 0; i < 3; i++) {          // rows
    for (int j = 0; j < 4; j++) {      // columns
```

```
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

Output:

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

6.1.5 Memory Layout and Arrays as Pointers

In C, arrays are stored in **contiguous memory locations**. For a single-dimensional array, elements are placed one after another in memory.

For multi-dimensional arrays, the layout is **row-major order**, meaning rows are stored consecutively.

For example, the 2D array `matrix[3][4]` is stored as:

```
Row 0: matrix[0][0], matrix[0][1], matrix[0][2], matrix[0][3]
Row 1: matrix[1][0], matrix[1][1], matrix[1][2], matrix[1][3]
Row 2: matrix[2][0], matrix[2][1], matrix[2][2], matrix[2][3]
```

Since arrays decay to pointers when passed to functions, the name of an array is treated as a pointer to its first element.

For a 1D array `arr`, `arr` is equivalent to `&arr[0]`.

For a 2D array, `matrix` points to the first row `matrix[0]`, and you can use pointer arithmetic to access elements, though this is more advanced.

6.1.6 Practical Use Cases

Storing Scores

Imagine you want to store the scores of 5 students in a single test:

```
int testScores[5] = {78, 92, 85, 88, 90};
```

You can calculate the average score by iterating over the array and summing the values.

Representing Tables of Data

Suppose you want to keep track of sales figures for 3 products over 4 quarters:

```c
int sales[3][4] = {
    {5000, 7000, 8000, 6500},
    {3000, 4000, 4500, 4200},
    {7000, 8500, 9000, 7800}
};
```

You can iterate over the rows and columns to calculate totals or find the highest sales.

6.1.7 Summary

Topic	Description
Single-dimensional array	Linear list of elements, indexed by one integer
Multi-dimensional array	Arrays of arrays, e.g., 2D for tables/matrices
Declaration	Specify type, name, and size(s)
Initialization	Use curly braces `{}` to set initial values
Access	Use indices starting at zero
Iteration	Use loops (`for`) to process array elements
Memory layout	Contiguous storage in row-major order for 2D
Arrays and pointers	Array names decay to pointers to first element

6.1.8 Conclusion

Arrays, both single and multi-dimensional, are essential tools for organizing and managing collections of data in C. Understanding how to declare, initialize, and iterate over arrays prepares you to work with more complex data structures and algorithms. Their memory layout and close relationship with pointers provide flexibility but require careful handling to avoid common mistakes such as out-of-bounds access.

6.2 String Manipulation with Character Arrays

In C, **strings** are represented as **arrays of characters** terminated by a special null character `'\0'`. This null terminator signals the end of the string, distinguishing it from arbitrary character data in memory.

6.2.1 How Strings Are Represented in C

Unlike some higher-level languages that treat strings as a distinct type, in C, a string is simply a sequence of characters stored in contiguous memory, ending with the `'\0'` character.

For example, the string `"Hello"` is stored as:

H	e	l	l	o	\0

The null terminator is essential because many C functions rely on it to know where the string ends.

6.2.2 Declaring and Initializing Strings

You can declare a string as a **character array**:

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Or more commonly, initialize it with a string literal, which automatically appends the null terminator:

```
char greeting[] = "Hello";
```

Here, the size of the array is inferred to be 6 to accommodate the 5 characters plus the null terminator.

6.2.3 Accessing and Manipulating Strings

You can access characters in a string using array indexing:

```
printf("%c\n", greeting[1]);  // Outputs 'e'
```

6.2.4 Importance of the Null Terminator

If the null terminator is missing, functions that expect a proper C string (like `printf`, `strlen`, or `strcpy`) will continue reading memory beyond the intended end, leading to undefined behavior or program crashes.

For example:

```
char badString[5] = {'H', 'e', 'l', 'l', 'o'};  // Missing '\0'
printf("%s\n", badString);  // May print garbage or cause errors
```

Always ensure your strings are **null-terminated**.

6.2.5 Common Pitfalls: Buffer Overflows

A common issue when manipulating strings manually is **buffer overflow** — writing beyond the allocated array size, which corrupts memory and causes bugs or security vulnerabilities.

For example:

```
char buffer[5];
strcpy(buffer, "Hello");  // 'Hello' needs 6 bytes including '\0', buffer only 5
```

This overwrites adjacent memory since `"Hello"` requires space for the null terminator.

Always ensure your buffers are large enough to hold the string plus the `'\0'`.

6.2.6 Copying Strings Manually

You can copy one string to another character by character using a loop:

Full runnable code:

```
#include <stdio.h>

void copyString(char *dest, const char *src) {
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';  // Null terminate the destination
}

int main() {
    char source[] = "C Programming";
    char destination[20];

    copyString(destination, source);
    printf("Copied string: %s\n", destination);
    return 0;
}
```

Output:

```
Copied string: C Programming
```

6.2.7 Concatenating Strings Manually

Concatenation means appending one string to the end of another.

Here is a simple example that concatenates two strings:

Full runnable code:

```c
#include <stdio.h>

void concatenate(char *dest, const char *src) {
    int i = 0;
    // Find end of dest string
    while (dest[i] != '\0') {
        i++;
    }

    int j = 0;
    // Append src to dest
    while (src[j] != '\0') {
        dest[i] = src[j];
        i++;
        j++;
    }

    dest[i] = '\0';  // Null terminate
}

int main() {
    char str1[30] = "Hello, ";
    char str2[] = "World!";

    concatenate(str1, str2);
    printf("%s\n", str1);
    return 0;
}
```

Output:

```
Hello, World!
```

Make sure `str1` has enough space to hold the concatenated result to avoid buffer overflow.

6.2.8 Printing Strings Using Loops

Though `printf` can directly print strings with `%s`, you can also print character-by-character using a loop:

```c
char message[] = "Hi there!";
int i = 0;

while (message[i] != '\0') {
    putchar(message[i]);
```

```
    i++;
}
putchar('\n');
```

This prints:

```
Hi there!
```

6.2.9 Summary

- Strings in C are **arrays of characters terminated by `'\0'`**.
- Always ensure strings are properly **null-terminated**.
- Manipulating strings requires careful handling to avoid **buffer overflows**.
- Manual copying and concatenation can be done with loops, but standard library functions (`strcpy`, `strcat`) are safer and more convenient.
- Access individual characters via array indexing.
- Printing strings can be done directly or character-by-character.

Understanding how strings are represented and manipulated as character arrays forms the foundation for effective text processing in C programming.

6.3 Common String Functions (`strlen`, `strcpy`, `strcmp`, etc.)

C provides a standard library header `<string.h>` which contains many useful functions for working with strings — arrays of characters terminated by a null character `'\0'`. These functions simplify common tasks such as measuring string length, copying, concatenating, and comparing strings.

In this section, we'll explore some of the most frequently used string functions:

- `strlen()`
- `strcpy()`
- `strcat()`
- `strcmp()`
- `strncpy()`

Understanding how to use these correctly is essential for safe and effective string manipulation.

6.3.1 `strlen()` Calculate String Length

`strlen()` returns the length of a string, **excluding the null terminator**.

Syntax:

```
size_t strlen(const char *str);
```

- `str` is a pointer to the null-terminated string.
- The return value is the number of characters before the `'\0'`.

Example:

Full runnable code:

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

int main() {
    char str[] = "Hello, World!";
    size_t length = strlen(str);
    printf("Length: %zu\n", length);  // Output: Length: 13
    return 0;
}
```

6.3.2 `strcpy()` Copy One String to Another

`strcpy()` copies a source string into a destination buffer, including the null terminator.

Syntax:

```
char *strcpy(char *dest, const char *src);
```

- `dest` must have enough space to hold the contents of `src` plus the null terminator.
- Returns a pointer to `dest`.

Example:

Full runnable code:

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

int main() {
    char source[] = "C programming";
    char destination[20];
    strcpy(destination, source);
    printf("Copied string: %s\n", destination);
    return 0;
}
```

Important: `strcpy()` does not check buffer sizes, so if `destination` is too small, this causes **buffer overflow**, leading to unpredictable behavior or security vulnerabilities.

6.3.3 `strncpy()` Safer String Copy with Length Limit

To avoid buffer overflows, `strncpy()` copies at most `n` characters and does not write beyond the destination size.

Syntax:

```
char *strncpy(char *dest, const char *src, size_t n);
```

- Copies up to `n` characters.
- If `src` is shorter than `n`, `dest` is padded with `'\0'`.
- Does **not** guarantee null-termination if `src` is longer or equal to `n`, so manual null termination might be necessary.

Example:

Full runnable code:

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

int main() {
    char source[] = "Hello, World!";
    char buffer[6];
    strncpy(buffer, source, sizeof(buffer) - 1);
    buffer[5] = '\0';  // Manually null-terminate
    printf("Copied safely: %s\n", buffer);
    return 0;
}
```

Output:

```
Copied safely: Hello
```

6.3.4 `strcat()` Concatenate Two Strings

`strcat()` appends the contents of one string (`src`) to the end of another (`dest`), overwriting the terminating null byte at the end of `dest`, and then adds a new null terminator.

Syntax:

```
char *strcat(char *dest, const char *src);
```

- `dest` must have enough space to hold the combined string.
- Returns `dest`.

Example:

Full runnable code:

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

int main() {
    char greeting[20] = "Hello, ";
    char name[] = "Alice";
    strcat(greeting, name);
    printf("%s\n", greeting);  // Outputs: Hello, Alice
    return 0;
}
```

6.3.5 Buffer Size Reminder

Always ensure the **destination buffer has enough space** for both the original string and the appended string plus the null terminator. Failure to do so causes buffer overflow.

For safer concatenation, consider using `strncat()`, which limits the number of characters appended.

6.3.6 `strcmp()` Compare Two Strings

`strcmp()` compares two strings lexicographically.

Syntax:

```c
int strcmp(const char *str1, const char *str2);
```

- Returns 0 if strings are equal.
- Returns a negative value if `str1` is lexicographically less than `str2`.
- Returns a positive value if `str1` is greater than `str2`.

Example:

Full runnable code:

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

int main() {
    char str1[] = "apple";
    char str2[] = "banana";

    int result = strcmp(str1, str2);
    if (result == 0) {
        printf("Strings are equal.\n");
    } else if (result < 0) {
        printf("%s comes before %s\n", str1, str2);
    } else {
```

```
        printf("%s comes after %s\n", str1, str2);
    }
    return 0;
}
```

Output:

```
apple comes before banana
```

6.3.7 Summary of Common Functions

Function	Purpose	Important Notes
`strlen`	Get string length (excluding `'\0'`)	Input must be null-terminated
`strcpy`	Copy string (including `'\0'`)	Destination buffer must be large enough
`strncpy`	Copy string with length limit	May not null-terminate; manual termination often needed
`strcat`	Append string to end of another	Destination buffer must be large enough
`strcmp`	Compare two strings lexicographically	Returns 0 if equal, <0 or >0 otherwise

6.3.8 Best Practices

- Always ensure **buffers are large enough** to hold strings plus the null terminator.
- When using `strncpy()` or `strncat()`, manually add `'\0'` to guarantee termination.
- Use `strcmp()` for safe string comparison instead of checking equality with `==`.
- For safety-critical code, consider safer string handling libraries or functions that prevent buffer overflows.

6.3.9 Conclusion

The string functions provided by `<string.h>` are essential tools in C programming for handling text data efficiently and safely. Mastering these functions, along with understanding their limitations, helps avoid common bugs and security issues related to string manipulation.

6.4 Array of Strings

In C, an **array of strings** is typically implemented as an **array of pointers to `char`**. Each element in this array points to the first character of a string. This approach is flexible and memory efficient, allowing you to manage collections of strings like names, commands, or messages.

6.4.1 Difference Between Two-Dimensional `char` Arrays and Arrays of String Pointers

Before diving into arrays of strings, it's important to understand the distinction:

- **Two-dimensional `char` array**: A fixed-size block of memory where all strings have the same maximum length.

```
char names[3][10];  // 3 strings, each can hold up to 9 chars + '\0'
```

- **Array of pointers to `char`**: An array where each element points to a separate string stored elsewhere (string literals or dynamically allocated).

```
char *names[] = {"Alice", "Bob", "Charlie"};
```

The latter is more flexible since each string can be of different lengths, and memory is allocated separately.

6.4.2 Declaring an Array of Strings

You can declare and initialize an array of string pointers like this:

```
char *commands[] = {"start", "stop", "pause", "exit"};
int count = sizeof(commands) / sizeof(commands[0]);
```

Here, `commands` is an array of 4 pointers, each pointing to a null-terminated string literal.

6.4.3 Accessing Elements

You access strings through indexing:

```
printf("First command: %s\n", commands[0]);  // Outputs "start"
```

Each string can be treated like a normal C string — you can print, compare, or copy it.

6.4.4 Iterating Over an Array of Strings

Loops help to process all strings. Here's an example printing all commands:

Full runnable code:

```c
#include <stdio.h>

int main() {
    char *commands[] = {"start", "stop", "pause", "exit"};
    int count = sizeof(commands) / sizeof(commands[0]);

    for (int i = 0; i < count; i++) {
        printf("Command %d: %s\n", i + 1, commands[i]);
    }
    return 0;
}
```

Output:

```
Command 1: start
Command 2: stop
Command 3: pause
Command 4: exit
```

6.4.5 Reading Input into an Array of Strings

To store user input strings, you can use a two-dimensional array because string literals are constant and should not be modified.

Full runnable code:

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

int main() {
    char names[3][20];  // Space for 3 strings, each up to 19 characters + '\0'

    for (int i = 0; i < 3; i++) {
        printf("Enter name %d: ", i + 1);
        fgets(names[i], sizeof(names[i]), stdin);
        // Remove newline character if present
        size_t len = strlen(names[i]);
        if (len > 0 && names[i][len - 1] == '\n') {
            names[i][len - 1] = '\0';
        }
    }

    printf("You entered:\n");
    for (int i = 0; i < 3; i++) {
```

```
        printf("%s\n", names[i]);
    }

    return 0;
}
```

Here, each `names[i]` is a fixed-size array of characters, allowing safe modification and input storage.

6.4.6 Manipulating Arrays of String Pointers

If you need an array of strings where strings can change or be assigned dynamically, you can declare an array of pointers and allocate memory for each string:

Full runnable code:

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

int main() {
    char *words[3];

    for (int i = 0; i < 3; i++) {
        words[i] = malloc(20 * sizeof(char));  // Allocate space for each string
        if (words[i] == NULL) {
            printf("Memory allocation failed\n");
            return 1;
        }
        printf("Enter word %d: ", i + 1);
        fgets(words[i], 20, stdin);
        size_t len = strlen(words[i]);
        if (len > 0 && words[i][len - 1] == '\n') {
            words[i][len - 1] = '\0';
        }
    }

    printf("Words entered:\n");
    for (int i = 0; i < 3; i++) {
        printf("%s\n", words[i]);
        free(words[i]);  // Free allocated memory
    }

    return 0;
}
```

This method allows strings of varying lengths up to the allocated size, but requires careful management of dynamic memory.

6.4.7 Summary

Concept	Description
Two-dimensional `char` array	Fixed-size matrix of characters (strings)
Array of pointers to `char`	Flexible array pointing to individual strings
Initialization	Can use string literals or dynamic allocation
Access	Use array indexing `arr[i]` to get each string
Modification	2D arrays allow in-place editing; literals don't
Memory management	Dynamic allocation needs manual `malloc` and `free`

6.4.8 Practical Use Case: Menu System

Arrays of strings are perfect for building menus:

```c
char *menu[] = {"Add", "Delete", "Search", "Exit"};
int options = sizeof(menu) / sizeof(menu[0]);

for (int i = 0; i < options; i++) {
    printf("%d. %s\n", i + 1, menu[i]);
}
```

6.4.9 Conclusion

Arrays of strings in C offer a powerful way to organize multiple text values. By understanding the difference between fixed-size 2D arrays and arrays of string pointers, you can choose the right approach based on your needs—whether it's storing fixed-length input or managing flexible strings dynamically. Iterating, reading, and manipulating such arrays is a common and essential skill for handling text data in C programs.

Chapter 7.

Pointers and Memory Addressing

1. Introduction to Pointers
2. Pointer Arithmetic
3. Pointers and Functions (Call by Reference)
4. Pointers and Arrays

7 Pointers and Memory Addressing

7.1 Introduction to Pointers

One of the most powerful and distinctive features of the C programming language is its support for **pointers** — variables that store **memory addresses**. Understanding pointers is crucial for mastering C because they allow direct memory manipulation, efficient data handling, and enable many advanced programming techniques.

7.1.1 What is a Pointer?

A **pointer** is a variable that holds the **address** of another variable in memory, rather than holding a direct value. In other words, instead of storing an integer or a character, a pointer stores the location where that integer or character is kept.

Imagine your computer's memory as a huge collection of numbered boxes, each with a unique address. A pointer points to one of these boxes, allowing you to access or modify the value inside that box indirectly.

7.1.2 Why Use Pointers?

Pointers offer several important advantages:

- **Efficiency:** Passing large data structures (like arrays or structs) by pointer avoids copying the entire data, saving memory and time.
- **Direct memory access:** Pointers let you manipulate memory locations directly, which is essential for systems programming, writing device drivers, and low-level tasks.
- **Dynamic memory management:** Using pointers, programs can request and free memory during runtime.
- **Enabling complex data structures:** Pointers are key to implementing linked lists, trees, graphs, and other dynamic data structures.

7.1.3 Declaring Pointers

To declare a pointer, you specify the type of data it points to, followed by an asterisk (*), and then the pointer's name.

```
int *p;     // p is a pointer to an int
char *cptr; // cptr is a pointer to a char
float *fptr; // fptr is a pointer to a float
```

The type before the * indicates the type of variable the pointer points to. This is important because pointer arithmetic and dereferencing depend on the pointed-to type.

7.1.4 Initializing Pointers

A pointer should be initialized to the address of a variable of the matching type using the **address-of operator** `&`:

```
int num = 42;
int *p = &num;  // p now holds the address of num
```

Here, `&num` means "the address of `num`".

7.1.5 Dereferencing Pointers: The * Operator

Dereferencing means accessing the value stored at the memory location the pointer points to. This is done using the dereference operator * (also called the indirection operator):

```
int value = *p;  // Get the value at the address stored in p
```

This reads the integer stored at the memory address held by `p`.

You can also **modify** the value pointed to by the pointer:

```
*p = 100;  // Change the value of num to 100 via the pointer
```

7.1.6 Example: Using a Pointer to Access and Modify a Variable

Full runnable code:

```
#include <stdio.h>

int main() {
    int num = 10;
    int *p = &num;  // p stores the address of num

    printf("Value of num: %d\n", num);      // 10
    printf("Address of num: %p\n", p);      // prints address
    printf("Value via pointer *p: %d\n", *p);  // 10

    *p = 25;  // Modify num through pointer

    printf("New value of num: %d\n", num);  // 25

    return 0;
```

```
}
```

Output:

```
Value of num: 10
Address of num: 0x7ffc9e123abc  // example address
Value via pointer *p: 10
New value of num: 25
```

This example shows how the pointer `p` stores the address of `num` and how `*p` accesses the value at that address. Changing `*p` directly modifies `num`.

7.1.7 Common Confusions: `p` vs. `*p`

- `p` is the pointer variable itself, holding a memory **address**.
- `*p` is the **value** stored at that memory address.

Think of `p` as the "pointer" (the arrow), and `*p` as what the arrow points to.

For example:

```
printf("%p\n", p);  // Prints the memory address stored in p
printf("%d\n", *p); // Prints the value at that address
```

7.1.8 Pointers Can Be Null

A pointer that doesn't point to any valid memory location should be initialized to `NULL` to avoid undefined behavior when dereferenced.

```
int *ptr = NULL;
if (ptr != NULL) {
    printf("%d\n", *ptr);
} else {
    printf("Pointer is NULL, cannot dereference.\n");
}
```

Always check for `NULL` before dereferencing pointers, especially when they point to dynamically allocated memory or function parameters.

7.1.9 Summary

- A pointer stores the **address** of a variable.
- Use `&` to get the address of a variable.

- Use `*` to **dereference** a pointer and access the value at that address.
- Pointers enable **efficient and powerful** memory manipulation.
- Distinguish between the pointer itself (`p`) and the value it points to (`*p`).
- Always initialize pointers and consider `NULL` checks to avoid errors.

7.1.10 Conclusion

Mastering pointers is a key step in becoming proficient in C programming. Pointers open up many possibilities for efficient data management and low-level system access, but they require careful handling to avoid mistakes like dangling pointers or memory corruption.

In the next sections, we will explore **pointer arithmetic**, how pointers interact with arrays, and how to use pointers to pass data to functions by reference.

7.2 Pointer Arithmetic

In C, pointers are not just simple variables holding memory addresses — you can perform **arithmetic operations** on pointers. Understanding pointer arithmetic is essential, especially when working with arrays and dynamic data structures, because it allows you to navigate through memory efficiently and correctly.

7.2.1 How Pointer Arithmetic Works

When you perform arithmetic on pointers, the operations take into account the **size of the data type** the pointer points to.

For example, if you have a pointer `int *p`, and you do `p + 1`, the pointer does not increase by 1 byte. Instead, it moves forward by **`sizeof(int)` bytes**, typically 4 bytes on many systems.

This means pointer arithmetic advances the pointer by increments or decrements of the size of the data type it references.

7.2.2 Common Pointer Arithmetic Operations

- **Increment (`++`)**: Moves the pointer to the next element.
- **Decrement (`--`)**: Moves the pointer to the previous element.

- **Addition (`+ n`)**: Moves the pointer forward by `n` elements.
- **Subtraction (`- n`)**: Moves the pointer backward by `n` elements.
- **Pointer subtraction**: Calculates the number of elements between two pointers.

7.2.3 Example: Pointer Arithmetic with Arrays

Suppose we have an array of integers:

```
int numbers[] = {10, 20, 30, 40, 50};
int *p = numbers;  // Points to the first element
```

Here, `p` points to the first element `numbers[0]`. Using pointer arithmetic:

```
printf("%d\n", *p);       // Output: 10
printf("%d\n", *(p + 1)); // Output: 20
printf("%d\n", *(p + 4)); // Output: 50
```

When we do `p + 1`, the pointer moves forward by `sizeof(int)` bytes (typically 4 bytes), pointing to the next element.

7.2.4 Incrementing and Decrementing Pointers

```
p++;  // Moves p to point to numbers[1]
printf("%d\n", *p);  // Output: 20

p--;  // Moves p back to numbers[0]
printf("%d\n", *p);  // Output: 10
```

7.2.5 Iterating Over an Array Using Pointer Arithmetic

Using pointers, you can iterate over an array without using array indexing:

Full runnable code:

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr;
    int length = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < length; i++) {
        printf("%d ", *(ptr + i));  // Access elements by pointer arithmetic
    }
    printf("\n");
```

```
    return 0;
}
```

Output:

```
1 2 3 4 5
```

7.2.6 Pointer Subtraction

You can subtract two pointers that point to elements of the same array. The result is the number of elements between them:

```
int arr[] = {10, 20, 30, 40, 50};
int *p1 = &arr[1];  // points to 20
int *p2 = &arr[4];  // points to 50

printf("Distance: %ld\n", p2 - p1);  // Output: 3
```

The subtraction tells us there are three elements between `p1` and `p2`.

7.2.7 Pointer Comparisons

Pointers can be compared using relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) if they point within the same array or one element past the end. This is useful in loops or conditional statements.

Example:

```
int arr[] = {1, 2, 3};
int *start = arr;
int *end = arr + 3;  // Points just past the last element

while (start < end) {
    printf("%d ", *start);
    start++;
}
```

Output:

```
1 2 3
```

7.2.8 Important: Avoid Undefined Behavior

Pointer arithmetic must be done carefully:

- **Do not access memory outside the bounds of an array**. Doing so causes

undefined behavior — your program might crash, produce garbage values, or corrupt data.

- It is legal to have a pointer point **one past the last element** of an array, but you **cannot dereference** this pointer.

7.2.9 Pointer Arithmetic with Different Data Types

The size of the data type affects how far a pointer moves when incremented or decremented.

Example with `char` pointers:

```
char str[] = "Hello";
char *cptr = str;

printf("%c\n", *cptr);     // H
cptr++;                    // Moves forward by 1 byte (sizeof(char) == 1)
printf("%c\n", *cptr);     // e
```

With `double` pointers (assuming 8 bytes per `double`):

```
double values[] = {1.1, 2.2, 3.3};
double *dptr = values;

printf("%f\n", *dptr);      // 1.1
dptr++;                    // Moves forward by 8 bytes
printf("%f\n", *dptr);      // 2.2
```

7.2.10 Summary

Operation	Effect on Pointer	Example
`p++`	Moves to next element	`p = p + 1`
`p--`	Moves to previous element	`p = p - 1`
`p + n`	Moves forward by `n` elements	`p = p + n`
`p - n`	Moves backward by `n` elements	`p = p - n`
`p1 - p2`	Number of elements between `p1` and `p2`	Must point within same array
`p1 == p2`, `<`, etc.	Compare positions within array	Used for loop conditions

7.2.11 Conclusion

Pointer arithmetic is a core concept in C programming that allows you to traverse arrays and manage memory addresses intuitively and efficiently. By understanding how pointer arithmetic depends on data type sizes, and how to safely use and compare pointers, you can write flexible and high-performance programs.

Always be cautious to stay within array boundaries to avoid undefined behavior and program crashes. With practice, pointer arithmetic becomes a natural and powerful tool in your C programming skillset.

7.3 Pointers and Functions (Call by Reference)

In C, **functions receive arguments by value** — meaning they get a copy of the variable, not the original. This behavior can make it tricky when you want a function to modify a variable defined outside its own scope. Fortunately, C allows you to achieve **call-by-reference** semantics using pointers. This enables functions to directly modify variables passed from the caller.

7.3.1 Understanding Pass-by-Value vs. Call-by-Reference

When you call a function with normal arguments, C copies the values into the function's parameters. For example:

```c
void increment(int x) {
    x = x + 1;  // Modifies local copy only
}

int main() {
    int num = 5;
    increment(num);
    printf("%d\n", num);  // Output: 5 (unchanged)
    return 0;
}
```

Here, `num` remains unchanged because `increment` only modifies a copy of `num`.

To allow a function to modify the caller's variable, you must pass the **address** of the variable — i.e., a pointer to it.

7.3.2 Passing Pointers to Functions

You can pass pointers as function parameters, then **dereference** the pointers inside the function to access or modify the original variables.

7.3.3 Example: Swapping Two Variables Using Pointers

A classic example that requires modifying variables in the caller is swapping two values.

Full runnable code:

```c
#include <stdio.h>

// Function to swap two integers using pointers
void swap(int *a, int *b) {
    int temp = *a;  // Dereference pointer to get value
    *a = *b;        // Assign value pointed to by b to location pointed by a
    *b = temp;      // Assign temp to location pointed by b
}

int main() {
    int x = 10, y = 20;

    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y);  // Pass addresses of x and y
    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}
```

Output:

```
Before swap: x = 10, y = 20
After swap: x = 20, y = 10
```

- In `main()`, we pass the **addresses** of `x` and `y` using the address-of operator `&`.
- Inside `swap()`, `a` and `b` are pointers to `int`.
- By dereferencing `*a` and `*b`, we modify the actual variables `x` and `y`.

7.3.4 Modifying Array Elements via Pointers

Since arrays decay to pointers when passed to functions, you can modify array elements inside a function.

Full runnable code:

```c
#include <stdio.h>

void updateFirstElement(int *arr) {
    arr[0] = 100;  // Modify first element of array
}

int main() {
    int numbers[] = {1, 2, 3};
    printf("Before: %d\n", numbers[0]);  // Output: 1
    updateFirstElement(numbers);
    printf("After: %d\n", numbers[0]);   // Output: 100
    return 0;
}
```

Passing `numbers` passes the pointer to its first element. Changes to `arr[0]` inside `updateFirstElement` affect the original array.

7.3.5 Returning Multiple Values Indirectly

C functions can return only a single value via `return`, but pointers enable you to return multiple values by modifying variables at the caller's addresses.

Full runnable code:

```c
#include <stdio.h>

void calculate(int a, int b, int *sum, int *diff) {
    *sum = a + b;
    *diff = a - b;
}

int main() {
    int x = 15, y = 10, s, d;
    calculate(x, y, &s, &d);
    printf("Sum = %d, Difference = %d\n", s, d);
    return 0;
}
```

Here, `calculate` updates the values of `s` and `d` through pointers passed from `main`.

7.3.6 Why Does C Use Pass-by-Value?

C was designed with simplicity and efficiency in mind. Pass-by-value is straightforward and avoids many complications of references seen in other languages. However, by passing pointers (addresses), C programmers gain flexibility to:

- Modify variables in the caller.
- Efficiently pass large data structures without copying.

- Implement dynamic memory management.

In essence, pointers **simulate** call-by-reference by giving functions access to the original variables via their memory addresses.

7.3.7 Key Points to Remember

- Always pass the **address** of a variable to a function if you need it to modify that variable.
- Inside the function, use the **dereference operator** `*` to access or change the value at the pointer's address.
- Be careful to ensure the pointer is valid and not `NULL` before dereferencing.
- For arrays, passing the array name passes a pointer to the first element.
- Use pointers to return multiple values or modify data in the caller.

7.3.8 Summary

Concept	Explanation	Example
Pass-by-value	Function receives a copy of argument	`void foo(int x)`
Call-by-reference (via pointer)	Function receives address of variable; modifies original	`void foo(int *x)`
Dereferencing	Access or modify the value the pointer points to	`*x = 10;`
Passing arrays	Arrays decay to pointers; functions modify original array	`void update(int *arr)`

7.3.9 Conclusion

Pointers provide a powerful way to extend the capabilities of C functions beyond pass-by-value limitations. By passing pointers, you enable functions to modify caller variables, work efficiently with arrays, and return multiple results. Mastering pointers and their use in function calls is essential for writing robust and efficient C programs.

In the next section, we will explore how pointers interact with arrays and how you can use them together for even more powerful programming patterns.

7.4 Pointers and Arrays

In C, **pointers and arrays share a very close relationship** — so close, in fact, that array names often behave like pointers. Understanding this connection is crucial because it unlocks powerful ways to access and manipulate data efficiently. In this section, we'll explore how arrays and pointers interact, how to use pointer arithmetic to access array elements, and best practices to safely handle arrays via pointers.

7.4.1 Array Names Decay Into Pointers

When you declare an array in C, such as:

```
int arr[5] = {10, 20, 30, 40, 50};
```

the name `arr` represents the entire array, but when used in most expressions, **it "decays" into a pointer to the first element**. This means `arr` is equivalent to `&arr[0]` — the address of the first element.

For example:

```
printf("%d\n", *arr);      // Output: 10, same as arr[0]
printf("%d\n", *(arr+2));  // Output: 30, same as arr[2]
```

Here, `*arr` dereferences the pointer to get the value of the first element, and `*(arr+2)` moves two `int` elements forward and dereferences to get the third element.

7.4.2 Accessing Array Elements Using Pointer Arithmetic

You can use pointers to access array elements without the usual square bracket notation:

```
int *p = arr;  // Pointer to the first element

for (int i = 0; i < 5; i++) {
    printf("%d ", *(p + i));  // Access element at index i
}
printf("\n");
```

Output:

```
10 20 30 40 50
```

In this loop, `p + i` advances the pointer by `i` elements (taking the size of `int` into account), and `*` dereferences it to access the value.

7.4.3 Modifying Array Contents via Pointers

Since pointers directly refer to memory locations, you can modify array elements by dereferencing pointers:

```
int *p = arr;

*(p + 1) = 25;  // Change second element from 20 to 25
p[3] = 45;      // Another way to change fourth element

for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
printf("\n");
```

Output:

```
10 25 30 45 50
```

Notice that `p[3]` is equivalent to `*(p + 3)`, demonstrating that pointer arithmetic and array indexing are closely linked.

7.4.4 Iterating Over Arrays with Pointers

You can also iterate through arrays by incrementing pointers:

```
int *p = arr;
int *end = arr + 5;  // Pointer to one past the last element

while (p < end) {
    printf("%d ", *p);
    p++;  // Move to the next element
}
printf("\n");
```

Output:

```
10 25 30 45 50
```

This approach is common in C and can make your loops more flexible and expressive.

7.4.5 Pointer Increment vs. Array Indexing

Both pointer increment and array indexing can be used to access array elements, but they have subtle differences:

- **Array indexing (`arr[i]`)** uses the array name and index, which the compiler translates internally to pointer arithmetic.
- **Pointer increment (`p++`)** explicitly moves the pointer to the next element.

Example:

```
int arr[] = {1, 2, 3};
int *p = arr;

printf("%d\n", arr[1]);  // Output: 2
printf("%d\n", *(p + 1)); // Output: 2

p++;
printf("%d\n", *p);      // Output: 2
```

Both methods achieve the same result. Understanding this equivalence helps you read and write flexible C code.

7.4.6 Distinctions Between Arrays and Pointers

Despite their close relationship, arrays and pointers are not the same:

- **Arrays have fixed size and allocate memory for all elements**.
- **Pointers are variables that hold memory addresses**, and can be reassigned.

Example:

```
int arr[3];
int *p;

p = arr;     // Valid: p points to arr[0]
p = p + 1;   // Pointer can move
// arr = arr + 1;  // Invalid: array name is not modifiable
```

You can move pointers freely, but you **cannot change the base address of an array**.

7.4.7 Best Practices to Avoid Pointer Errors with Arrays

Working with pointers and arrays can be tricky, so keep these points in mind:

1. **Avoid accessing memory out of bounds**: Always ensure pointers stay within the allocated array range to prevent undefined behavior.
2. **Remember the null terminator for character arrays**: If dealing with strings (which are arrays of `char`), ensure they are null-terminated (`'\0'`).
3. **Be mindful of pointer types**: Pointer arithmetic is based on the size of the data type it points to. Using the wrong pointer type may lead to unexpected results.
4. **Use clear variable names and comments** to improve code readability when using pointers and arithmetic.

7.4.8 Practical Scenario: Storing and Modifying Scores

Imagine you want to store students' scores and increase each by 5 points:

Full runnable code:

```c
#include <stdio.h>

int main() {
    int scores[] = {70, 85, 90, 75, 88};
    int n = sizeof(scores) / sizeof(scores[0]);
    int *ptr = scores;

    for (int i = 0; i < n; i++) {
        *(ptr + i) += 5;  // Increase each score by 5
    }

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

    return 0;
}
```

Output:

```
75 90 95 80 93
```

Here, pointer arithmetic allows concise modification of the array elements.

7.4.9 Summary

Concept	Explanation	Example
Array name decay	Array name acts like a pointer to first element	`arr` is same as `&arr[0]`
Access via pointer arithmetic	Use `*(arr + i)` instead of `arr[i]`	`*(p + 2)`
Pointer increment	Move pointer to next element	`p++`
Modifying array via pointer	Dereference pointer to change element	`*(p + 1) = 25`
Array vs Pointer	Array base address fixed; pointer can move	`arr` cannot be assigned, but `p` can
Avoid out-of-bounds access	Always ensure pointer stays in array bounds	Use loop limits

7.4.10 Conclusion

Pointers and arrays are inseparable concepts in C programming. Arrays give you a convenient way to store sequences of data, while pointers provide the flexibility to traverse and manipulate this data efficiently. By mastering pointer arithmetic and understanding the subtle distinctions between arrays and pointers, you gain powerful tools to write high-performance and memory-efficient programs.

Always remember to use pointers carefully, respect array boundaries, and use meaningful code structure to avoid common pitfalls and bugs.

Chapter 8.

Dynamic Memory Management

1. `malloc`, `calloc`, `realloc`, `free`
2. Memory Leaks and Best Practices
3. Building Dynamic Arrays

8 Dynamic Memory Management

8.1 `malloc`, `calloc`, `realloc`, `free`

In C, **dynamic memory management** allows you to allocate, resize, and free memory during program execution, rather than relying only on fixed-size, compile-time arrays. This is essential for creating flexible programs that adapt to varying data sizes, such as user input or growing data structures.

The C standard library provides four key functions to manage dynamic memory:

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

Understanding these functions is fundamental to efficient memory handling and avoiding common bugs like memory leaks or segmentation faults.

8.1.1 `malloc()`: Allocate Memory Block

The `malloc()` function (short for **memory allocation**) reserves a block of memory of a specified size (in bytes) and returns a pointer to the beginning of that block.

Syntax:

```
void *malloc(size_t size);
```

- `size` specifies how many bytes to allocate.
- Returns a `void*` pointer to the allocated memory, or `NULL` if allocation fails.
- The allocated memory contains **garbage values** (uninitialized).

Example: Allocating an array of 10 integers using `malloc`:

Full runnable code:

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

int main() {
    int *arr = (int *)malloc(10 * sizeof(int));  // Allocate memory for 10 ints

    if (arr == NULL) {  // Always check if malloc succeeded
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 10; i++) {
        arr[i] = i * 2;  // Initialize array elements
```

```
    }

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

    free(arr);  // Free allocated memory
    return 0;
}
```

8.1.2 `calloc()`: Allocate and Zero-Initialize

`calloc()` (**contiguous allocation**) works like `malloc()` but **initializes the allocated memory to zero**.

Syntax:

```
void *calloc(size_t num, size_t size);
```

- Allocates memory for `num` elements, each of size `size`.
- Returns a pointer to the zero-initialized block, or `NULL` if it fails.

Example: Allocate and zero-initialize an array of 5 doubles:

```
double *arr = (double *)calloc(5, sizeof(double));
if (arr == NULL) {
    // Handle error
}
```

The memory for `arr` contains zeros, so you don't have to manually initialize it.

8.1.3 `realloc()`: Resize Previously Allocated Memory

Sometimes you allocate memory but later realize you need more or less space. `realloc()` allows resizing the allocated memory block without losing existing data (up to the minimum of old and new sizes).

Syntax:

```
void *realloc(void *ptr, size_t new_size);
```

- `ptr`: Pointer previously returned by `malloc()`, `calloc()`, or `realloc()`.
- `new_size`: New size in bytes.
- Returns a pointer to the resized block or `NULL` if allocation fails (original pointer remains valid).

Example: Resize the earlier allocated integer array from 10 to 20 elements:

```c
int *arr = (int *)malloc(10 * sizeof(int));
// Initialize arr...

int *temp = (int *)realloc(arr, 20 * sizeof(int));
if (temp == NULL) {
    printf("Reallocation failed\n");
    free(arr);  // Free original if realloc fails
    return 1;
} else {
    arr = temp;  // Use resized array
}
```

8.1.4 `free()`: Release Allocated Memory

Whenever you allocate memory dynamically, you must release it once you're done using it, to avoid **memory leaks** — situations where memory remains allocated but is no longer accessible.

Syntax:

```c
void free(void *ptr);
```

- `ptr` is a pointer to previously allocated memory.
- After calling `free()`, the memory is returned to the system.
- Do **not** use the pointer after freeing it (dangling pointer).

8.1.5 Important Notes and Best Practices

- **Always check if `malloc()`, `calloc()`, or `realloc()` returns `NULL`.** Failing to check can lead to crashes or undefined behavior.
- When casting the result of allocation functions to another pointer type (e.g., `(int *)`), note that in C, this cast is optional but common practice for clarity.
- **Never forget to `free()` dynamically allocated memory** — especially in long-running programs or those with many allocations.
- After freeing memory, it's a good habit to set the pointer to `NULL` to avoid accidental dereferencing:

```c
free(arr);
arr = NULL;
```

- Use `calloc()` when you want memory initialized to zero, such as for arrays where default zero values are required.
- `realloc()` is very useful for dynamic arrays where the size is unknown upfront, but be mindful that the returned pointer may differ from the original.
- If `realloc()` fails, the original memory remains valid — avoid losing the pointer

without freeing.

8.1.6 Practical Example: Dynamic Array Input

Full runnable code:

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

int main() {
    int *arr = NULL;
    int capacity = 2;
    int size = 0;

    arr = (int *)malloc(capacity * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    int input;
    printf("Enter integers (negative to stop):\n");
    while (1) {
        scanf("%d", &input);
        if (input < 0) break;

        if (size == capacity) {
            capacity *= 2;
            int *temp = (int *)realloc(arr, capacity * sizeof(int));
            if (temp == NULL) {
                printf("Reallocation failed\n");
                free(arr);
                return 1;
            }
            arr = temp;
        }
        arr[size++] = input;
    }

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

    free(arr);
    return 0;
}
```

This program dynamically grows the array as the user enters values, demonstrating `malloc()`, `realloc()`, and `free()` in practice.

8.1.7 Summary

Function	Purpose	Key Points
`malloc()`	Allocate uninitialized memory block	Check for `NULL`, returns `void*`
`calloc()`	Allocate zero-initialized memory block	Useful for arrays/structs
`realloc()`	Resize previously allocated block	Can move memory, handle `NULL`
`free()`	Release allocated memory	Prevent memory leaks

8.1.8 Conclusion

Mastering `malloc()`, `calloc()`, `realloc()`, and `free()` is essential for effective memory management in C. These functions provide the flexibility to build dynamic data structures and manage resources efficiently. Always remember to check for allocation failures and free memory when it's no longer needed. Proper dynamic memory management leads to robust, efficient, and leak-free programs.

8.2 Memory Leaks and Best Practices

When programming in C, managing memory correctly is one of the most important — and challenging — responsibilities. Unlike some modern languages that automatically handle memory cleanup (like Python or Java), C requires the programmer to explicitly allocate and free memory. Failure to do so properly can lead to **memory leaks**, which degrade program performance and may eventually cause crashes or system instability.

8.2.1 What is a Memory Leak?

A **memory leak** occurs when a program allocates memory dynamically (using `malloc()`, `calloc()`, or `realloc()`) but **fails to release it back to the system** using `free()`. The allocated memory remains reserved and inaccessible, reducing the available memory for the program and other processes.

Imagine your program requests memory repeatedly but never returns it — over time, the operating system runs out of free memory, and the program or even the entire system can slow down or crash.

8.2.2 How Memory Leaks Occur

Here are some common scenarios that cause memory leaks:

1. **Forgetting to call `free()`**

```
int *arr = malloc(10 * sizeof(int));
// Use the array
// Missing free(arr);
```

Here, the allocated memory is never freed, causing a leak.

2. **Losing track of allocated memory**

```
int *ptr = malloc(20 * sizeof(int));
ptr = malloc(30 * sizeof(int));  // Previous allocation lost
```

In this case, the first allocated block of 20 integers is not freed before `ptr` is reassigned to a new block of 30 integers. The original memory is "orphaned," causing a leak.

3. **Returning pointers to local (stack) variables instead of heap memory**

This is a related error that causes undefined behavior and not exactly a leak, but improper pointer management can also contribute to leaks.

8.2.3 Impact of Memory Leaks

Memory leaks may seem harmless for small or short-running programs, but they can cause serious problems:

- **Gradual slowdown:** As free memory decreases, your program's performance may degrade.
- **Crashes:** Exhaustion of memory may cause segmentation faults or aborts.
- **System instability:** In systems with limited resources (e.g., embedded devices), leaks may affect the entire system.

Therefore, **detecting and fixing memory leaks is vital** for writing robust, efficient software.

8.2.4 Best Practices to Avoid Memory Leaks

Here are some essential tips and practices to prevent memory leaks:

Always Pair Allocation and Deallocation

Every time you call `malloc()`, `calloc()`, or `realloc()`, ensure there is a matching `free()` when the memory is no longer needed.

Example:

```
int *data = malloc(100 * sizeof(int));
if (data == NULL) {
    // Handle error
}
// Use data...
free(data);  // Free when done
```

It helps to think of memory management as a contract: every allocation implies a responsibility to free.

Initialize Pointers to NULL

Initializing pointers to NULL reduces the chance of freeing invalid or uninitialized pointers:

```
int *ptr = NULL;
ptr = malloc(10 * sizeof(int));
if (ptr != NULL) {
    // Use ptr
    free(ptr);
    ptr = NULL;  // Avoid dangling pointer
}
```

Setting pointers to NULL after freeing them avoids accidental usage of "dangling" pointers — pointers that refer to freed memory.

Avoid Losing Pointers to Allocated Memory

If you need to reassign a pointer that owns dynamically allocated memory, free the old memory first:

```
int *ptr = malloc(20 * sizeof(int));
if (ptr == NULL) exit(1);

// Later...
int *temp = malloc(30 * sizeof(int));
if (temp == NULL) {
    free(ptr);
    exit(1);
}
free(ptr);
ptr = temp;
```

Never overwrite a pointer without freeing its current memory first, or you risk leaking the original block.

Use Tools for Leak Detection

Manual detection of leaks is hard, especially in complex programs. Tools like **Valgrind** (Linux/Mac) or **Dr. Memory** (Windows) can automatically detect memory leaks by monitoring your program's allocation and deallocation.

Example Valgrind usage:

```
valgrind --leak-check=full ./your_program
```

Valgrind will report memory still allocated at program exit, helping you identify where leaks occur.

Write Clear, Maintainable Code

Good code structure reduces errors:

- Use **functions** to allocate and free memory, isolating responsibility.
- Document ownership rules: who allocates, who frees.
- Keep allocations and frees close in logic flow when possible.
- Comment your code to clarify memory management intent.

8.2.5 Example: Memory Leak and Fix

Leaky Code:

Full runnable code:

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

void leaky_function() {
    int *arr = malloc(10 * sizeof(int));
    if (arr == NULL) return;
    // Use arr...
    // Missing free(arr);
}

int main() {
    for (int i = 0; i < 1000; i++) {
        leaky_function();  // Each call leaks 10 ints
    }
    return 0;
}
```

Fixed Code:

```c
void fixed_function() {
    int *arr = malloc(10 * sizeof(int));
    if (arr == NULL) return;
    // Use arr...
    free(arr);  // Properly freed
}
```

8.2.6 Summary Table

Practice	Description
Pair every allocation with free	Always release allocated memory
Initialize pointers to `NULL`	Prevent dangling pointers
Avoid pointer overwrites	Free old memory before pointer reassignment
Use memory-checking tools	Detect leaks with Valgrind, Dr. Memory, or similar
Write maintainable code	Clear ownership and structured memory management

8.2.7 Conclusion

Memory leaks are a common and dangerous problem in C programming. They occur when allocated memory is not freed properly, leading to wasted resources, degraded performance, and potential crashes. By adopting best practices—such as pairing every allocation with a corresponding free, carefully managing pointers, and utilizing memory debugging tools—you can write safer and more efficient C programs.

Consistent attention to memory management not only prevents leaks but also makes your code easier to maintain and debug. As you grow in your C programming journey, mastering these practices will become a key part of your skill set.

8.3 Building Dynamic Arrays

In C, dynamic arrays allow programs to handle datasets that grow or shrink during runtime—unlike static arrays, which require fixed sizes at compile-time. Building a dynamic array manually involves using `malloc()` and `realloc()` to allocate and resize memory as needed. This technique is foundational in implementing higher-level data structures like lists, stacks, and queues.

8.3.1 Key Concepts

When constructing a dynamic array, we must manage:

- **Size**: The current number of elements stored.
- **Capacity**: The total number of elements the allocated memory can hold before resizing is needed.

This pattern allows us to expand the array only when necessary, reducing the overhead of frequent memory operations.

8.3.2 Basic Strategy

1. Start with a small allocated array using `malloc()`.
2. As elements are added, check if the size has reached capacity.
3. If full, use `realloc()` to allocate a larger block—usually doubling the capacity.
4. Add the new element and update the size counter.
5. At the end, `free()` the allocated memory.

8.3.3 Example: Storing User Inputs

Here's a simple program that stores a list of integers entered by the user. The list grows as needed using `realloc()`.

Full runnable code:

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

int main() {
    int *arr = NULL;
    int size = 0;
    int capacity = 2;
    int input;

    // Allocate initial memory
    arr = (int *)malloc(capacity * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter integers (negative number to stop):\n");
    while (1) {
        scanf("%d", &input);
        if (input < 0) break;

        // Resize if needed
        if (size == capacity) {
            capacity *= 2;
            int *temp = realloc(arr, capacity * sizeof(int));
            if (temp == NULL) {
                printf("Reallocation failed.\n");
                free(arr);
                return 1;
            }
            arr = temp;
        }

        arr[size++] = input;
    }

    // Print stored numbers
```

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

    free(arr); // Clean up memory
    return 0;
}
```

8.3.4 Explanation

- **Initial Allocation**: We start with space for 2 integers.
- **Size vs Capacity**: `size` tracks how many elements have been added, and `capacity` tracks the current total capacity of the array.
- **Reallocation**: When `size == capacity`, we double the capacity and use `realloc()` to extend the array.
- **Safety Checks**: After `malloc()` and `realloc()`, we check if the pointer is `NULL` in case of failure.
- **Memory Cleanup**: Before the program exits, we `free()` the allocated memory to avoid leaks.

8.3.5 Resizing Strategy

A common approach for resizing is **doubling the capacity**. This provides **amortized linear time** for insertions: though some insertions are slow (requiring a reallocation and copy), most are fast.

Other strategies include:

- Fixed-size increments (e.g., +10): simple but may result in frequent reallocations.
- Power-of-two growth (e.g., $1 \to 2 \to 4 \to 8 \to \ldots$): balances memory use and performance.
- Custom thresholds based on memory constraints.

You should choose the strategy based on your application's needs.

8.3.6 Preserving Data with `realloc()`

One benefit of `realloc()` is that it copies existing data to the new block if the block is moved. If possible, the system may also expand the memory in place, avoiding a copy. Either way, it's essential to store the return value in a **temporary pointer** and check for `NULL`:

```c
int *temp = realloc(arr, new_capacity * sizeof(int));
if (temp == NULL) {
    // Failed: original arr still valid
    free(arr); // Or handle error gracefully
    return 1;
}
arr = temp; // Use resized block
```

This pattern prevents you from losing access to the original memory if the reallocation fails.

8.3.7 Dynamic Array of Strings

You can also build a dynamic array of `char*` (strings). Here's a simplified version:

Full runnable code:

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

int main() {
    char **words = NULL;
    int size = 0, capacity = 2;
    char buffer[100];

    words = malloc(capacity * sizeof(char *));
    if (words == NULL) return 1;

    printf("Enter words (type 'exit' to stop):\n");
    while (1) {
        scanf("%s", buffer);
        if (strcmp(buffer, "exit") == 0) break;

        if (size == capacity) {
            capacity *= 2;
            char **temp = realloc(words, capacity * sizeof(char *));
            if (temp == NULL) {
                printf("Realloc failed.\n");
                for (int i = 0; i < size; i++) free(words[i]);
                free(words);
                return 1;
            }
            words = temp;
        }

        words[size] = malloc(strlen(buffer) + 1);
        if (words[size] == NULL) return 1;
        strcpy(words[size], buffer);
        size++;
    }

    printf("You entered:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("%s\n", words[i]);
        free(words[i]);
    }
    free(words);

    return 0;
}
```

This example demonstrates handling an array of dynamically allocated strings and carefully freeing all resources.

8.3.8 Tips and Best Practices

Practice	Description
Track size and capacity separately	Avoids buffer overflows
Always check for allocation failures	Prevents crashes
Use `realloc()` safely	Assign result to a temp pointer
Free all memory before exiting	Prevents memory leaks
Initialize unused pointers to `NULL`	Helps catch errors early

8.3.9 Conclusion

Building dynamic arrays in C gives your programs the flexibility to handle varying amounts of data efficiently. By combining `malloc()`, `realloc()`, and `free()`, along with good memory management practices, you can implement scalable and safe data structures tailored to your needs. As your programs grow in complexity, mastering these techniques will allow you to write more dynamic, responsive, and robust software.

Chapter 9.

Structures and Unions

1. Defining and Using `struct`
2. Nested Structures and Arrays of Structures
3. `typedef` for Structs
4. Introduction to `union` and Comparison with `struct`

9 Structures and Unions

9.1 Defining and Using `struct`

9.1.1 Introduction to Structures

In C, a `struct` (short for "structure") is a **user-defined composite data type** that allows you to group variables of different types under one name. This is particularly useful when modeling real-world entities that have multiple attributes. For example, a student might have a name, an ID, and a GPA — each of a different type. A structure helps organize this data as a single unit.

The `struct` keyword provides a way to define new types that group variables (called **members**) of different types.

9.1.2 Defining a Structure

A structure is defined using the `struct` keyword followed by a name and a block of member declarations:

```
struct Student {
    int id;
    char name[50];
    float gpa;
};
```

In this example, we define a structure type named `Student` that has three members:

- `id` of type `int`
- `name` of type character array (string)
- `gpa` of type `float`

This definition **does not** allocate memory. It only tells the compiler what a `Student` looks like.

9.1.3 Declaring Structure Variables

After defining a structure, you can declare variables of that type:

```
struct Student s1;
```

You can also combine the definition and declaration:

```
struct Student {
    int id;
    char name[50];
```

```
    float gpa;
} s1, s2;
```

9.1.4 Initializing Structures

You can initialize a structure at the time of declaration:

```
struct Student s1 = {1001, "Alice", 3.8};
```

C99 and later allow **designated initializers**:

```
struct Student s2 = {.id = 1002, .name = "Bob", .gpa = 3.5};
```

9.1.5 Accessing Structure Members

To access or modify members of a structure, use the **dot (`.`) operator**:

```
s1.id = 1003;
printf("Name: %s\n", s1.name);
s1.gpa = 4.0;
```

You can read and write each member just like regular variables.

9.1.6 Pointers to Structures and the Arrow Operator

You can create a pointer to a structure like this:

```
struct Student *ptr = &s1;
```

To access members using a pointer, you can use either:

```
(*ptr).gpa = 3.9;  // Using dereferencing and dot
```

or the more common and concise **arrow operator (`->`)**:

```
ptr->gpa = 3.9;    // Equivalent to above
```

Both achieve the same result: accessing a member of the structure pointed to by `ptr`.

9.1.7 Practical Example: Student Record

Here's a simple program that demonstrates defining, initializing, and accessing a structure.

Full runnable code:

```c
#include <stdio.h>

struct Student {
    int id;
    char name[50];
    float gpa;
};

int main() {
    struct Student s1 = {1001, "Alice", 3.7};
    struct Student *ptr = &s1;

    printf("Student Details:\n");
    printf("ID: %d\n", ptr->id);
    printf("Name: %s\n", ptr->name);
    printf("GPA: %.2f\n", ptr->gpa);

    ptr->gpa = 3.9;  // Modify GPA
    printf("Updated GPA: %.2f\n", s1.gpa);

    return 0;
}
```

Output:

```
Student Details:
ID: 1001
Name: Alice
GPA: 3.70
Updated GPA: 3.90
```

9.1.8 Another Example: 2D Point

Let's look at another use-case — representing a 2D point:

```c
struct Point {
    int x;
    int y;
};

int main() {
    struct Point p1 = {5, 10};
    printf("Point: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Structures are perfect for grouping related data like this, which improves readability and modularity.

9.1.9 Arrays of Structures

You can also create arrays of structures to hold multiple records:

```
struct Student class[3];

class[0].id = 1;
strcpy(class[0].name, "Tom");
class[0].gpa = 3.4;

// and so on...
```

This is useful for managing collections of objects, such as students, employees, products, etc.

9.1.10 Summary

Concept	Example
Structure definition	`struct Student { ... };`
Variable declaration	`struct Student s1;`
Member access	`s1.name`, `ptr->gpa`
Pointer access	`struct Student *p = &s1;`
Initialization	`{1001, "Alice", 3.8}`

9.1.11 Best Practices

- Use meaningful structure names (`Student`, `Point`, `Employee`) for clarity.
- Prefer the `->` operator when working with structure pointers.
- When designing complex programs, structures help organize and encapsulate data logically.
- Consider using `typedef` to simplify struct usage (covered in the next section).

9.1.12 Conclusion

Structures allow you to create complex data models in C by grouping related variables of different types into a single, manageable unit. They are essential for organizing data, passing grouped information to functions, and forming the foundation of more advanced data structures. Mastering structures is a key step toward writing clear, maintainable, and modular C programs.

9.2 Nested Structures and Arrays of Structures

C structures become even more powerful when you use **nested structures** (structures within structures) and **arrays of structures** (multiple records of the same type). These features allow you to create and manage complex data hierarchies, such as student records in a classroom, shapes in a graphics system, or employees in a company.

9.2.1 Nested Structures

A **nested structure** is a structure that contains another structure as a member. This is useful when logically grouping related sub-information inside a larger entity.

Example: Student with Date of Birth

Full runnable code:

```c
#include <stdio.h>

struct Date {
    int day;
    int month;
    int year;
};

struct Student {
    int id;
    char name[50];
    struct Date dob;  // Nested structure
};

int main() {
    struct Student s1 = {1001, "Alice", {15, 3, 2002}};

    printf("Student Name: %s\n", s1.name);
    printf("Date of Birth: %02d/%02d/%04d\n", s1.dob.day,
                s1.dob.month, s1.dob.year);

    return 0;
}
```

Output:

```
Student Name: Alice
Date of Birth: 15/03/2002
```

In the example above, `struct Student` contains a `struct Date` as a member. You access the nested members using the dot (`.`) operator repeatedly: `s1.dob.day`, `s1.dob.month`, etc.

9.2.2 Accessing Nested Members via Pointers

When using pointers to a structure that contains another structure, access the nested members like this:

```
struct Student *ptr = &s1;
printf("%d\n", ptr->dob.year);  // Equivalent to (*ptr).dob.year
```

This is commonly used when dealing with dynamic data and functions.

9.2.3 Arrays of Structures

An **array of structures** allows you to store multiple instances of a structure in a single collection, making it easier to manage and process lists of related entities.

Example: Classroom of Students

Full runnable code:

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

struct Student {
    int id;
    char name[50];
    float gpa;
};

int main() {
    struct Student class[3];

    // Manually initializing data
    class[0].id = 101;
    strcpy(class[0].name, "Alice");
    class[0].gpa = 3.8;

    class[1].id = 102;
    strcpy(class[1].name, "Bob");
    class[1].gpa = 3.5;

    class[2].id = 103;
    strcpy(class[2].name, "Charlie");
    class[2].gpa = 3.9;

    // Iterating over the array
    printf("Class Roster:\n");
    for (int i = 0; i < 3; i++) {
        printf("ID: %d, Name: %s, GPA: %.2f\n",
               class[i].id, class[i].name, class[i].gpa);
    }

    return 0;
}
```

Output:

```
Class Roster:
ID: 101, Name: Alice, GPA: 3.80
ID: 102, Name: Bob, GPA: 3.50
ID: 103, Name: Charlie, GPA: 3.90
```

This pattern is helpful when dealing with groups of similar items — products, employees, books, etc.

9.2.4 Combining Nested Structures and Arrays

You can also combine **arrays of nested structures**, which is useful when managing multi-layered data.

Example: Multiple Students with Address Info

```c
struct Address {
    char city[30];
    int zip;
};

struct Student {
    int id;
    char name[50];
    struct Address addr;
};
```

You can now declare an array of `Student` and assign values like this:

```c
struct Student class[2] = {
    {101, "Alice", {"New York", 10001}},
    {102, "Bob", {"Los Angeles", 90001}}
};

for (int i = 0; i < 2; i++) {
    printf("Student: %s, City: %s, ZIP: %d\n",
           class[i].name, class[i].addr.city, class[i].addr.zip);
}
```

This makes your code cleaner and more logically organized.

9.2.5 Practical Use Case: Graphical Scene

In a simple graphics application, you may represent a 2D point and a shape using nested structures and arrays.

```c
struct Point {
    int x, y;
};

struct Circle {
    struct Point center;
    float radius;
};

int main() {
    struct Circle circles[2] = {
        {{5, 10}, 3.5},
        {{15, 20}, 7.2}
    };

    for (int i = 0; i < 2; i++) {
        printf("Circle %d - Center: (%d, %d), Radius: %.1f\n",
               i + 1, circles[i].center.x, circles[i].center.y,
                circles[i].radius);
    }

    return 0;
}
```

9.2.6 Summary

Concept	Example Syntax
Nested struct definition	`struct Student { struct Date dob; };`
Access nested member	`student.dob.year` or `ptr->dob.year`
Array of structs	`struct Student class[10];`
Access via index	`class[i].name`, `class[i].dob.day`

9.2.7 Best Practices

- Use meaningful structure names and member names.
- Break down large structures into nested components for clarity.
- Always initialize arrays of structures carefully to avoid garbage values.
- Use loops to iterate over arrays, especially when reading from or printing to the user.

9.2.8 Conclusion

By combining **nested structures** and **arrays of structures**, you can model real-world problems with multiple levels of data in a clean and organized way. These tools are essential for building scalable C programs and serve as a foundation for implementing more advanced data structures.

9.3 `typedef` for Structs

In C programming, writing `struct` repeatedly can be cumbersome, especially when working with large or complex codebases. The `typedef` keyword helps by creating an alias for existing data types, including structures. Using `typedef` with `struct` simplifies code and improves readability, making it easier to maintain and use structures throughout your program.

9.3.1 Why Use `typedef` with Structs?

Without `typedef`, every time you declare a variable of a structure type, you need to include the `struct` keyword:

```
struct Student {
    int id;
    char name[50];
};

struct Student s1;  // Must repeat 'struct'
```

This can get repetitive, particularly when passing structures to functions or working with pointers. By using `typedef`, you can define a new type name that acts as a shorthand for `struct Student`, allowing cleaner, more concise declarations.

9.3.2 Basic Syntax

Here's how you use `typedef` with a structure:

```
typedef struct {
    int id;
    char name[50];
} Student;
```

Now you can declare variables simply as:

```
Student s1;
```

This is functionally equivalent to:

```
struct Student {
    int id;
    char name[50];
};
typedef struct Student Student;
```

Both methods are valid and widely used.

9.3.3 Before and After: A Comparison

Without `typedef`:

```
struct Point {
    int x;
    int y;
};

struct Point p1;
```

With `typedef`:

```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
```

You save keystrokes, and the code looks cleaner and more intuitive.

9.3.4 Creating Pointer Type Aliases

You can also create a pointer alias using `typedef`:

```
typedef struct {
    int id;
    char name[50];
} Student;

typedef Student* StudentPtr;
```

Now, you can declare a pointer like this:

```
StudentPtr ptr;
```

Which is equivalent to:

```
Student *ptr;
```

This approach is especially useful in API design and data structure implementations (e.g., linked lists, trees) where pointers are used frequently.

9.3.5 Practical Example

Let's demonstrate a complete use case with `typedef`:

Full runnable code:

```
#include <stdio.h>

typedef struct {
    int id;
    char name[50];
    float gpa;
} Student;

void printStudent(Student s) {
    printf("ID: %d, Name: %s, GPA: %.2f\n", s.id, s.name, s.gpa);
}

int main() {
    Student s1 = {1001, "Alice", 3.8};
    printStudent(s1);
    return 0;
}
```

The code is cleaner because we no longer have to write `struct Student` everywhere.

9.3.6 Usage in Modular Code

In large projects, `typedef` is frequently used to improve modularity and abstraction. You might declare types in a header file:

```
// student.h
#ifndef STUDENT_H
#define STUDENT_H

typedef struct {
    int id;
    char name[50];
    float gpa;
} Student;

void printStudent(Student s);

#endif
```

Then use them in multiple `.c` files without exposing implementation details.

9.3.7 Common Patterns

Pattern	Example
Structure alias	`typedef struct { ... } Foo;`
Named struct with alias	`typedef struct Foo { ... } Foo;`
Pointer alias	`typedef Foo* FooPtr;`
API-friendly naming	`typedef struct Person Person;`

9.3.8 Summary

- `typedef` allows you to create aliases for `struct` types, reducing verbosity.
- It improves readability, especially when structures are used frequently.
- You can alias both structures and pointers to structures.
- It is a best practice in API design and modular programming.

9.3.9 Conclusion

The `typedef` keyword is a simple but powerful tool in C that significantly enhances code clarity when working with structures. It allows you to abstract away the underlying `struct` keyword and write cleaner, more expressive code. As your programs grow in complexity, using `typedef` becomes an essential practice for writing maintainable and modular code.

9.4 Introduction to `union` and Comparison with `struct`

In C programming, a `union` is a user-defined data type similar to a `struct`, but with a key difference in how it stores data. While `struct` allows you to group variables of different types together (and each member occupies its own memory space), a `union` allows storing different data types in the same memory location. This can lead to significant memory savings when used correctly, but it also requires careful handling.

9.4.1 Syntax of `union`

The syntax for declaring a `union` is nearly identical to that of a `struct`:

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

This `union` named `Data` can store an `int`, a `float`, or a string of up to 19 characters (plus the null terminator). However, it can only store **one of these at a time**—all members share the same memory.

9.4.2 Accessing Union Members

To use a union, you define a variable and access its members just like with structures:

Full runnable code:

```
#include <stdio.h>

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

int main() {
    union Data d;

    d.i = 42;
    printf("d.i = %d\n", d.i);

    d.f = 3.14;
    printf("d.f = %.2f\n", d.f);

    // Note: d.i is now overwritten
    printf("d.i (after d.f) = %d\n", d.i);

    return 0;
}
```

Output:

```
d.i = 42
d.f = 3.14
d.i (after d.f) = [garbage or unexpected value]
```

In this example, assigning a new value to `d.f` **overwrites** the previous value stored in `d.i`, because both share the same memory space.

9.4.3 Memory Layout: `union` vs. `struct`

The key difference between a `union` and a `struct` is in how memory is allocated:

- **`struct`** allocates enough memory to hold **all** its members.
- **`union`** allocates memory equal to the **largest** member, and all members share that space.

Example:

```
struct ExampleStruct {
    int i;
    float f;
    char str[20];
}; // Size: sizeof(int) + sizeof(float) + 20 bytes  28 bytes

union ExampleUnion {
    int i;
    float f;
    char str[20];
}; // Size: max(sizeof(int), sizeof(float), 20 bytes) = 20 bytes
```

Using a union can save memory in cases where only one member is needed at a time.

9.4.4 Use Cases for `union`

Unions are ideal in situations where:

1. You need to **store different types of data** in the **same memory space**, but never more than one type at a time.
2. You are working with **variant records**, such as in parsers or network protocols.
3. You need to **interpret the same memory in multiple ways**, such as in **bit manipulation** or **type punning**.

Example: Tagged Union

Full runnable code:

```
#include <stdio.h>

enum DataType { INT, FLOAT };

struct Value {
    enum DataType type;
    union {
        int i;
        float f;
    } data;
};
```

```c
int main() {
    struct Value val;

    val.type = INT;
    val.data.i = 42;

    if (val.type == INT)
        printf("Integer: %d\n", val.data.i);

    val.type = FLOAT;
    val.data.f = 3.14;

    if (val.type == FLOAT)
        printf("Float: %.2f\n", val.data.f);

    return 0;
}
```

This technique, sometimes called a **tagged union**, combines a `union` with an enum to track the type currently stored.

9.4.5 Risks and Considerations

While unions are powerful, they must be used with care:

- **Type confusion**: Accessing a member that was not most recently written to results in **undefined behavior**.
- **Portability issues**: Not all systems represent floating-point and integer values in compatible ways.
- **Debugging complexity**: It's harder to debug code that uses unions improperly, especially when switching types frequently.

Always ensure you **track the active type** using an external indicator (like an `enum`).

9.4.6 Comparison Summary

Feature	`struct`	`union`
Memory Allocation	Sum of all members	Size of the largest member
Member Access	All at once	One at a time
Use Case	Represent related, full records	Save memory for mutually exclusive data
Safety	Safer and more intuitive	Riskier, needs careful handling

9.4.7 Conclusion

Unions provide a way to manage memory more efficiently when you know that only one type of value will be used at a time. While they are not as commonly used as structures, they are indispensable in low-level programming, embedded systems, and performance-critical code. Always compare your use case to the memory-saving and complexity trade-offs of unions. When used properly, unions can be a powerful addition to your C programming toolkit.

Chapter 10.

File Input/Output

1. Reading and Writing Text Files
2. File Pointers and Modes (`r`, `w`, `a`, `rb`, etc.)
3. `fscanf`, `fprintf`, `fgets`, `fputs`
4. Binary File Operations

10 File Input/Output

10.1 Reading and Writing Text Files

File Input/Output (I/O) is a critical part of many programs. In C, you can read from and write to text files using functions provided in the standard library `<stdio.h>`. Text files store data in human-readable form, making them ideal for logs, configuration files, or saving user data.

This section will guide you through the process of reading from and writing to text files in C, including how to open, manipulate, and close files properly, with practical examples and error handling.

10.1.1 Opening a File

To work with a file in C, you first need to open it using the `fopen()` function:

```
FILE *fopen(const char *filename, const char *mode);
```

- `filename`: Name or path to the file.
- `mode`: The file access mode (e.g., `"r"` for reading, `"w"` for writing).

Example:

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    printf("Failed to open file.\n");
    return 1;
}
```

10.1.2 Writing to a Text File

To write data to a text file, use `"w"` mode (write) or `"a"` mode (append). `"w"` creates a new file or overwrites an existing one; `"a"` adds content to the end of the file.

Full runnable code:

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("log.txt", "w");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
    }
```

```
    fprintf(fp, "This is a log entry.\n");
    fprintf(fp, "Logging another entry.\n");

    fclose(fp);
    return 0;
}
```

- `fprintf()` works like `printf()` but writes to a file.
- Always call `fclose()` to flush buffers and close the file.

10.1.3 Appending to a File

Use `"a"` mode if you need to add text to an existing file:

```
fp = fopen("log.txt", "a");
fprintf(fp, "Appending a new line.\n");
fclose(fp);
```

10.1.4 Reading from a Text File

Use `"r"` mode to read from a file. Common reading functions include:

- `fgetc()` – reads one character.
- `fgets()` – reads a line.
- `fscanf()` – formatted input, like `scanf()`.

Example: Reading a file line-by-line

Full runnable code:

```
#include <stdio.h>

int main() {
    char line[100];
    FILE *fp = fopen("config.txt", "r");

    if (fp == NULL) {
        printf("Cannot open file.\n");
        return 1;
    }

    while (fgets(line, sizeof(line), fp)) {
        printf("Line: %s", line);
    }

    fclose(fp);
    return 0;
}
```

- `fgets()` reads up to `sizeof(line) - 1` characters or until a newline is found.
- Be cautious of buffer size to avoid overflows.

10.1.5 Reading Character-by-Character

```c
char ch;
while ((ch = fgetc(fp)) != EOF) {
    putchar(ch);
}
```

This is useful for low-level parsing tasks or when analyzing file contents at the character level.

10.1.6 Writing User Input to a File

Let's combine input and output: ask the user for data and save it to a file.

Full runnable code:

```c
#include <stdio.h>

int main() {
    char name[50];
    int age;

    FILE *fp = fopen("users.txt", "a");
    if (fp == NULL) {
        printf("Failed to open file.\n");
        return 1;
    }

    printf("Enter your name: ");
    fgets(name, sizeof(name), stdin);

    printf("Enter your age: ");
    scanf("%d", &age);

    fprintf(fp, "Name: %sAge: %d\n", name, age);

    fclose(fp);
    return 0;
}
```

10.1.7 Error Handling in File Operations

Always check if a file pointer returned by `fopen()` is `NULL`. This may happen if:

- The file does not exist (in read mode).
- You don't have permissions.
- Disk or memory issues occur.

Checking for `NULL` ensures your program can fail gracefully.

```
if (fp == NULL) {
    perror("Error opening file");
    return 1;
}
```

Use `perror()` to print a descriptive system error message.

10.1.8 Closing Files

Use `fclose(fp)` to close a file when done. This:

- Releases system resources.
- Ensures that all data is flushed to disk.
- Prevents data loss.

Always close files, especially when writing.

10.1.9 Summary of Key Functions

Function	Purpose
`fopen()`	Opens a file
`fclose()`	Closes a file
`fprintf()`	Writes formatted data to a file
`fscanf()`	Reads formatted data from a file
`fgets()`	Reads a string from a file
`fgetc()`	Reads a character from a file

10.1.10 Conclusion

Reading from and writing to text files in C is straightforward but powerful. By mastering these file operations, you can build programs that store data persistently, log user activity, and read configuration files. Always remember to check for errors, close files, and ensure proper formatting when handling input and output.

10.2 File Pointers and Modes (`r`, `w`, `a`, `rb`, etc.)

In C, working with files requires the use of **file pointers** and understanding different **file modes**. File pointers are used to manage file access, while file modes determine how the file will be opened—whether for reading, writing, or appending, and whether in text or binary form.

This section explains what file pointers are, describes the various file modes available in C, and provides examples to help you choose the right mode for your needs.

10.2.1 What Is a File Pointer?

A **file pointer** is a pointer to a `FILE` object, which is defined in the `<stdio.h>` library. It keeps track of the file being accessed and the current position in that file. You obtain a file pointer using the `fopen()` function:

```
FILE *fp = fopen("data.txt", "r");
```

If `fopen()` succeeds, it returns a pointer to a `FILE` structure. If it fails, it returns `NULL`.

10.2.2 Common File Opening Modes

The second argument to `fopen()` specifies the file access **mode**. Here's a table summarizing the most commonly used modes:

Mode	Description
`"r"`	Open for **reading**. Fails if file doesn't exist.
`"w"`	Open for **writing**. Creates a new file or truncates existing one.
`"a"`	Open for **appending**. Data is added to the end of the file.
`"r+"`	Open for **reading and writing**. File must exist.
`"w+"`	Open for **reading and writing**. Truncates existing or creates new.
`"a+"`	Open for **reading and appending**. File is created if it doesn't exist.
`"rb"`	Open **binary file** for reading.
`"wb"`	Open **binary file** for writing.
`"ab"`	Open **binary file** for appending.

> Text modes (`r`, `w`, `a`) interpret newline characters. Binary modes (`rb`, `wb`, etc.) do not and are used for non-text files like images or audio files.

10.2.3 Examples

Reading from a File

```c
FILE *fp = fopen("input.txt", "r");
if (fp == NULL) {
    printf("File not found!\n");
    return 1;
}
// Read data here
fclose(fp);
```

- Use `"r"` when you only want to **read**.
- File must exist, or `fopen()` will return `NULL`.

Writing to a File

```c
FILE *fp = fopen("output.txt", "w");
fprintf(fp, "Hello, World!\n");
fclose(fp);
```

- `"w"` creates the file if it doesn't exist.
- If it does exist, the file is **cleared (truncated)**.

Appending to a File

```c
FILE *fp = fopen("log.txt", "a");
fprintf(fp, "New log entry\n");
fclose(fp);
```

- `"a"` does **not truncate** existing content.
- All writes go to the end of the file.

10.2.4 Binary Modes: `rb`, `wb`, `ab`

Binary modes are used when dealing with non-text data. For example:

```c
FILE *fp = fopen("image.dat", "rb");
```

Binary modes are essential when working with custom file formats or raw memory data to ensure byte-for-byte accuracy. Unlike text mode, binary mode does **not translate** `\n` or other special characters.

10.2.5 Choosing the Right Mode

Use `"r"`:

- When reading from an existing text file.

Use `"w"`:

- When writing from scratch and **overwriting** is acceptable.

Use `"a"`:

- When you want to **preserve** existing content and add more.

Use `"r"`, `"w"`, or `"a"`:

- When both reading and writing are required.

Use `"rb"`, `"wb"`, `"ab"`:

- When dealing with binary files (e.g., images, executable data).

10.2.6 Error Handling

Always check the file pointer after calling `fopen()`:

```
if (fp == NULL) {
    perror("Error opening file");
    return 1;
}
```

Using `perror()` provides a system error message.

10.2.7 Closing the File

Don't forget to close the file with `fclose(fp);` when you're done. Failing to close files can lead to data corruption and resource leaks.

10.2.8 Summary

- **File pointers** (`FILE *`) are used to manage open files in C.
- **File modes** determine how files are accessed—read, write, append, or combinations thereof.
- Choose **text or binary** modes depending on the content.

- Always check for `NULL` when opening files and close files properly using `fclose()`.

10.3 `fscanf`, `fprintf`, `fgets`, `fputs`

In C, file I/O (input/output) operations can be performed using a rich set of standard functions. While `fopen()` and `fclose()` are used to manage files, reading and writing data to and from files is often done using `fscanf()`, `fprintf()`, `fgets()`, and `fputs()`. These functions provide fine-grained control for handling both formatted and line-based data.

This section introduces these four essential functions, with syntax, examples, and important usage tips.

10.3.1 `fscanf()` Reading Formatted Data

`fscanf()` is used to read formatted input from a file, similar to how `scanf()` works with standard input.

Syntax:

```
int fscanf(FILE *stream, const char *format, ...);
```

- `stream`: File pointer returned by `fopen()`.
- `format`: Format string like in `scanf()`, e.g., `"%d %s"`.
- Additional arguments are pointers to variables where the input will be stored.

Example:

```
FILE *fp = fopen("data.txt", "r");
int id;
char name[50];

if (fp != NULL) {
    fscanf(fp, "%d %s", &id, name);
    printf("ID: %d, Name: %s\n", id, name);
    fclose(fp);
}
```

Pitfall: Always check for successful reads using the return value of `fscanf()`. It returns the number of successfully read items.

10.3.2 `fprintf()` Writing Formatted Data

`fprintf()` works like `printf()`, but it writes to a file instead of the console.

Syntax:

```
int fprintf(FILE *stream, const char *format, ...);
```

Example:

```
FILE *fp = fopen("report.txt", "w");
int score = 95;
char name[] = "Alice";

if (fp != NULL) {
    fprintf(fp, "Student: %s, Score: %d\n", name, score);
    fclose(fp);
}
```

This writes a formatted line to the file. The format string and arguments follow the same rules as `printf()`.

10.3.3 `fgets()` Reading Strings Line by Line

`fgets()` reads a line or a portion of it from a file, including spaces and tabs.

Syntax:

```
char *fgets(char *str, int n, FILE *stream);
```

- `str`: Buffer to store the input.
- `n`: Maximum number of characters to read (including `\0`).
- `stream`: File pointer.

Example:

```
FILE *fp = fopen("notes.txt", "r");
char buffer[100];

if (fp != NULL) {
    while (fgets(buffer, sizeof(buffer), fp)) {
        printf("Line: %s", buffer);
    }
    fclose(fp);
}
```

Pitfall: `fgets()` includes the newline character (`\n`) if there is space in the buffer. You may need to strip it manually.

10.3.4 `fputs()` Writing Strings to Files

`fputs()` writes a string (excluding the null terminator) to a file.

Syntax:

```
int fputs(const char *str, FILE *stream);
```

Example:

```
FILE *fp = fopen("output.txt", "a");

if (fp != NULL) {
    fputs("This is a new line.\n", fp);
    fclose(fp);
}
```

Note: `fputs()` does not add a newline automatically. If you need one, you must include `\n` in the string.

10.3.5 Practical Example: Reading and Writing Structured Data

Suppose we want to read student data from a file and write a formatted report.

Input File (`students.txt`):

```
101 Alice 85
102 Bob 92
103 Charlie 78
```

Code:

Full runnable code:

```
#include <stdio.h>

int main() {
    FILE *in = fopen("students.txt", "r");
    FILE *out = fopen("report.txt", "w");

    int id, score;
    char name[50];

    if (in == NULL || out == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fprintf(out, "STUDENT REPORT\n");
    fprintf(out, "==========================\n");

    while (fscanf(in, "%d %s %d", &id, name, &score) == 3) {
        fprintf(out, "ID: %d, Name: %s, Score: %d\n", id, name, score);
    }

    fclose(in);
    fclose(out);
    return 0;
```

```
}
```

This program reads from one file and creates a nicely formatted report in another.

10.3.6 Common Pitfalls and Tips

- **Check file opening:** Always verify that `fopen()` returned a valid pointer.
- **Beware of buffer sizes:** Use appropriate buffer sizes with `fgets()` to avoid overflows.
- **Don't forget newline characters:** When using `fputs()` or `fprintf()`, include `\n` when needed.
- **Handle EOF properly:** Functions like `fscanf()` return `EOF` or the number of matched items—check these to avoid reading errors.
- **Avoid mixing `fscanf()` and `fgets()`** on the same stream without care, due to newline and buffer issues.

10.3.7 Summary

- `fscanf()` and `fprintf()` allow you to read/write formatted data, similar to `scanf()` and `printf()`.
- `fgets()` and `fputs()` handle strings and are ideal for reading/writing lines.
- These functions are powerful tools for structured file I/O in C.
- Always check return values and manage buffers to avoid errors.

10.4 Binary File Operations

While text files are convenient for human readability and simple data storage, they are not always efficient or suitable for storing structured data such as images, game states, or serialized objects. That's where binary files come in.

Binary files store data exactly as it exists in memory—byte for byte—making them more compact and faster to read/write. However, because of their raw format, they require more careful handling and are not human-readable.

In this section, we'll cover how to perform binary I/O in C using `fread()`, `fwrite()`, and related functions. You'll also learn about file positioning with `fseek()` and `ftell()`, and understand when binary file operations are the right choice.

10.4.1 Binary Files vs. Text Files

In **text files**, data is stored as characters, often using ASCII encoding. Numbers are converted to their string representations, which can take more space and require parsing.

In **binary files**, data is written in its raw, binary form, without conversion. For example, an `int` is stored as 4 bytes directly from memory, not as the string `"123"`.

Advantages of binary files:

- Faster read/write operations.
- More space-efficient.
- Ideal for storing structs or large datasets.

Drawbacks:

- Not human-readable.
- Potential issues with cross-platform compatibility (due to byte order or struct padding).

10.4.2 Writing Binary Data with `fwrite()`

To write binary data to a file, use `fwrite()`.

Syntax:

```
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);
```

- `ptr`: pointer to the data.
- `size`: size of each element.
- `count`: number of elements to write.
- `stream`: file pointer.

Example:

Full runnable code:

```c
#include <stdio.h>

int main() {
    FILE *fp = fopen("data.bin", "wb");
    int scores[] = {90, 85, 78, 92};

    if (fp != NULL) {
        fwrite(scores, sizeof(int), 4, fp);
        fclose(fp);
    }

    return 0;
}
```

This program writes 4 integers to a binary file called `data.bin`.

10.4.3 Reading Binary Data with `fread()`

To read binary data, use `fread()`.

Syntax:

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
```

Example:

Full runnable code:

```c
#include <stdio.h>

int main() {
    FILE *fp = fopen("data.bin", "rb");
    int scores[4];

    if (fp != NULL) {
        fread(scores, sizeof(int), 4, fp);
        fclose(fp);
    }

    for (int i = 0; i < 4; i++) {
        printf("Score %d: %d\n", i + 1, scores[i]);
    }

    return 0;
}
```

This reads 4 integers from the binary file and prints them.

10.4.4 Using Binary I/O with Structs

Binary files are especially useful for saving and restoring complex structures.

Example:

Full runnable code:

```c
#include <stdio.h>

struct Player {
    char name[20];
    int score;
};

int main() {
    struct Player p1 = {"Alice", 1200};
    FILE *fp = fopen("player.dat", "wb");

    if (fp != NULL) {
        fwrite(&p1, sizeof(struct Player), 1, fp);
```

```
        fclose(fp);
    }

    return 0;
}
```

Later, you can read the same structure back:

Full runnable code:

```c
#include <stdio.h>

struct Player {
    char name[20];
    int score;
};

int main() {
    struct Player p2;
    FILE *fp = fopen("player.dat", "rb");

    if (fp != NULL) {
        fread(&p2, sizeof(struct Player), 1, fp);
        fclose(fp);
    }

    printf("Name: %s, Score: %d\n", p2.name, p2.score);
    return 0;
}
```

10.4.5 File Positioning: `fseek()` and `ftell()`

Sometimes you need to jump to a specific location in a binary file. This is done using `fseek()` and `ftell()`.

`fseek()` Syntax:

```c
int fseek(FILE *stream, long offset, int origin);
```

- `offset`: number of bytes to move.
- `origin`: starting point (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).

`ftell()` Syntax:

```c
long ftell(FILE *stream);
```

Returns the current position in the file (in bytes).

Example:

```c
fseek(fp, sizeof(struct Player) * 2, SEEK_SET);  // Jump to 3rd record
```

Use case: Quickly access a specific record in a large file without reading everything.

10.4.6 Real-World Scenarios for Binary Files

- **Game development**: Save player state, maps, and assets efficiently.
- **Embedded systems**: Store configuration in compact binary format.
- **Image/audio files**: Store raw pixel or sample data for performance.

10.4.7 Cautions and Portability Issues

- **Endianness**: The byte order of multibyte types (e.g., `int`, `float`) may vary between systems.
- **Struct padding**: Compilers may insert extra bytes between struct members for alignment.
- **Avoid writing raw pointers**: They are meaningless when read back from disk.

If portability is critical, consider using standardized serialization formats or handling endian conversion manually.

10.4.8 Summary

- Binary files store data in raw memory format and are useful for performance and space efficiency.
- Use `fwrite()` to write binary data and `fread()` to read it.
- Structures can be stored directly but watch for portability issues.
- Use `fseek()` and `ftell()` to navigate binary files.
- Be mindful of memory layout and alignment when writing complex data.

Chapter 11.

Command-Line Arguments

1. `argc` and `argv` Explained
2. Building CLI Utilities
3. Parsing and Validating User Input

11 Command-Line Arguments

11.1 `argc` and `argv` Explained

When you run a program from the command line, you can pass arguments to it. These arguments can be file names, configuration flags, or any other input your program needs to perform its task. In C, command-line arguments are handled using two special parameters in the `main()` function: `argc` and `argv`.

Understanding how `argc` and `argv` work is key to building flexible and user-friendly command-line utilities.

11.1.1 The `main()` Function Signature

To receive command-line arguments in a C program, you must define the `main()` function with two parameters:

```
int main(int argc, char *argv[])
```

- `argc` stands for **argument count**. It is an integer representing the number of arguments passed to the program, including the program name.
- `argv` stands for **argument vector**. It is an array of character pointers (strings), each pointing to one of the command-line arguments.

You might also see the function declared as:

```
int main(int argc, char **argv)
```

This is equivalent and equally valid.

11.1.2 Breaking Down `argc` and `argv`

Let's say you run the following command in a terminal:

```
./greet Alice Bob
```

Here's how `argc` and `argv` will look inside the program:

- `argc` = 3
- `argv[0]` = `"./greet"` (the name of the program)
- `argv[1]` = `"Alice"` (first argument)
- `argv[2]` = `"Bob"` (second argument)

So, `argc` tells you how many elements are in the `argv` array, and `argv` contains the actual strings entered.

11.1.3 Example: Printing Command-Line Arguments

Full runnable code:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

Sample output if run as `./program file1.txt -v`:

```
Number of arguments: 3
Argument 0: ./program
Argument 1: file1.txt
Argument 2: -v
```

This example loops through the `argv` array using `argc` and prints each argument.

11.1.4 Using Arguments in Your Program

You can use command-line arguments to control your program's behavior.

Example: A Simple Greeting Tool

Full runnable code:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    printf("Hello, %s!\n", argv[1]);
    return 0;
}
```

Run it like this:

```
./greet Alice
```

Output:

```
Hello, Alice!
```

If no name is passed, it prompts the user with usage instructions.

11.1.5 Optional Flags and Positional Parameters

A common pattern in CLI programs is to support flags and options.

Example: Verbose Mode

Full runnable code:

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

int main(int argc, char *argv[]) {
    int verbose = 0;

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-v") == 0) {
            verbose = 1;
        }
    }

    if (verbose) {
        printf("Verbose mode enabled\n");
    } else {
        printf("Running in normal mode\n");
    }

    return 0;
}
```

This program checks whether `-v` was passed and adjusts its behavior.

11.1.6 Important Notes

- **Always check `argc`** before accessing elements of `argv` to avoid out-of-bounds errors.
- **Arguments are strings**: Even if the user enters a number, it is stored as a string (`char*`). Use functions like `atoi()` or `strtol()` to convert them to integers.
- **The program name is always `argv[0]`**. It's often the path or name of the executable.

11.1.7 Summary

Command-line arguments allow your program to take input directly from the user when launched. The `argc` and `argv` parameters to `main()` give you access to those inputs:

- `argc` tells you how many arguments were passed.
- `argv` is an array of strings containing each argument.

With just a few lines of code, you can make your C programs interactive and flexible for real-world use cases like file processing, configuration management, and tool automation.

11.2 Building CLI Utilities

Command-line utilities are programs designed to perform specific tasks, taking input directly from the user through command-line arguments. These utilities are powerful because they can be combined with other programs or automated through scripts. In this section, you will learn how to create simple command-line utilities in C that accept arguments, parse them, and perform different actions.

11.2.1 Basic Principles for CLI Utility Design

When building command-line tools, it's important to:

- **Parse inputs clearly:** Check and interpret user-provided arguments.
- **Validate inputs:** Make sure the inputs are correct and meaningful.
- **Handle errors gracefully:** Provide helpful error messages and usage instructions.
- **Offer help or usage messages:** Explain how to use your utility.

11.2.2 Example 1: A Simple Calculator

Let's build a command-line calculator that accepts two numbers and an operation.

Full runnable code:

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

int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("Usage: %s <num1> <operator> <num2>\n", argv[0]);
        printf("Operators supported: +, -, *, /\n");
        return 1;
    }

    double num1 = atof(argv[1]);
    double num2 = atof(argv[3]);
    char op = argv[2][0];
```

```
    double result;

    switch (op) {
        case '+':
            result = num1 + num2;
            break;
        case '-':
            result = num1 - num2;
            break;
        case '*':
            result = num1 * num2;
            break;
        case '/':
            if (num2 == 0) {
                printf("Error: Division by zero is undefined.\n");
                return 1;
            }
            result = num1 / num2;
            break;
        default:
            printf("Invalid operator '%c'. Use +, -, *, or /.\n", op);
            return 1;
    }

    printf("Result: %.2f\n", result);
    return 0;
}
```

How this works:

- The program expects exactly 3 arguments after the program name (`argc == 4`).
- It converts the first and third arguments to numbers using `atof()`.
- It reads the operator from the second argument.
- Using a `switch` statement, it performs the operation.
- It includes checks for invalid operators and division by zero.
- If inputs are invalid or missing, it prints usage instructions.

Sample run:

```
./calc 12.5 + 7.3
Result: 19.80
```

11.2.3 Example 2: Searching for a Word in a File

Here's a simple tool to search for a word inside a text file.

Full runnable code:

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

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <filename> <search_word>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    char line[256];
    int line_number = 0;
    int found = 0;

    while (fgets(line, sizeof(line), file)) {
        line_number++;
        if (strstr(line, argv[2]) != NULL) {
            printf("Found '%s' on line %d: %s", argv[2], line_number, line);
            found = 1;
        }
    }

    fclose(file);

    if (!found) {
        printf("'%s' not found in %s\n", argv[2], argv[1]);
    }

    return 0;
}
```

How this works:

- The program expects two arguments: a filename and a search word.
- It opens the file and reads it line by line.
- For each line, it checks if the search word is present using `strstr()`.
- If found, it prints the line number and line contents.
- If the file cannot be opened, it uses `perror()` to show the system error.
- If the word isn't found, it informs the user.

Sample run:

```
./search notes.txt C
Found 'C' on line 3: I love programming in C language.
```

11.2.4 Handling Missing or Invalid Input

Robust programs always check inputs carefully.

- Use `argc` to verify the number of arguments.

- Validate data types where possible.
- Provide clear error messages to guide users.
- Print usage instructions if inputs are missing or incorrect.

11.2.5 Providing Help Messages

You can add support for a `-h` or `--help` flag to display usage:

```
if (argc == 2 && (strcmp(argv[1], "-h") == 0 ||
    strcmp(argv[1], "--help") == 0)) {
    printf("Usage: %s [options]\n", argv[0]);
    printf("Options:\n");
    printf("  -h, --help    Show this help message\n");
    // more info
    return 0;
}
```

This makes your utility more user-friendly.

11.2.6 Tips for Building More Complex CLI Utilities

- Consider using libraries like `getopt()` for parsing options.
- Always check return values of system calls (e.g., file opening).
- Keep user communication clear and concise.
- Document your program's usage inside the code or external manuals.

11.2.7 Summary

Creating command-line utilities in C involves:

- Parsing and validating input arguments.
- Performing actions based on input.
- Handling errors and providing help messages.

By following these principles, your programs become more reliable and easier to use, opening the door to automation and integration with other tools.

11.3 Parsing and Validating User Input

When building command-line programs in C, properly parsing and validating user input is critical. This ensures your program behaves correctly, handles errors gracefully, and prevents unexpected crashes or security issues.

11.3.1 Why Parse and Validate?

Command-line arguments come as strings (`char *argv[]`), so any numeric or structured data must be converted. Users may provide wrong, incomplete, or malicious input, so your program must:

- Check the number of arguments (`argc`) matches expectations.
- Convert strings to the correct data types safely.
- Validate values fall within acceptable ranges.
- Provide meaningful error messages and guidance.

11.3.2 Checking Argument Count

The first step is always to verify if the number of arguments is correct.

```
if (argc != 3) {
    printf("Usage: %s <integer> <float>\n", argv[0]);
    return 1;
}
```

Failing to do this may cause your program to access invalid memory or behave unexpectedly.

11.3.3 Converting Strings to Numbers

C offers multiple functions to convert strings to numeric types, but some are safer and more flexible than others.

Using `atoi()`

`atoi()` converts a string to an integer but offers **no error detection**:

```
int x = atoi(argv[1]);
```

If `argv[1]` is not a valid number, `atoi()` returns 0, but 0 could also be a valid input. This ambiguity can cause errors.

Better: Using `strtol()` for Integers

`strtol()` (string to long) is preferred because it provides error detection:

```
#include <errno.h>
#include <stdlib.h>

char *endptr;
errno = 0; // reset errno before call
long val = strtol(argv[1], &endptr, 10);

if (errno != 0 || *endptr != '\0') {
    printf("Invalid integer input: %s\n", argv[1]);
    return 1;
}
```

- `endptr` points to the first invalid character after the parsed number.
- If it is not `'\0'`, extra invalid characters exist.
- `errno` signals conversion errors like overflow.

Converting Floating-Point Numbers: `strtof()` and `strtod()`

For floats and doubles, use `strtof()` or `strtod()` which work similarly:

```
char *endptr;
float fval = strtof(argv[2], &endptr);

if (*endptr != '\0') {
    printf("Invalid float input: %s\n", argv[2]);
    return 1;
}
```

11.3.4 Validating Allowed Values

After conversion, check if the value falls within expected bounds.

Example: For a percentage input between 0 and 100:

```
if (val < 0 || val > 100) {
    printf("Value out of range: %ld (expected 0-100)\n", val);
    return 1;
}
```

11.3.5 Example: Parsing and Validating Command-Line Input

Here is a complete example combining the above concepts:

Full runnable code:

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <age> <height>\n", argv[0]);
        return 1;
    }

    char *endptr;

    // Parse age as integer
    errno = 0;
    long age = strtol(argv[1], &endptr, 10);
    if (errno != 0 || *endptr != '\0' || age < 0 || age > 130) {
        printf("Invalid age: %s\n", argv[1]);
        return 1;
    }

    // Parse height as float
    float height = strtof(argv[2], &endptr);
    if (*endptr != '\0' || height <= 0 || height > 3) {
        printf("Invalid height: %s\n", argv[2]);
        return 1;
    }

    printf("Age: %ld years, Height: %.2f meters\n", age, height);
    return 0;
}
```

This program:

- Checks argument count.
- Uses `strtol()` and `strtof()` for safe parsing.
- Validates age and height are within reasonable ranges.
- Prints helpful error messages.

11.3.6 Input Sanitization and Graceful Failure

Input sanitization means filtering inputs to remove or reject unexpected or harmful data. For command-line programs, this often involves:

- Rejecting invalid or suspicious strings.
- Checking for buffer overflows or excessively long inputs (when reading files or user input).
- Providing fallback or usage messages on error.

Example: If your program expects only a set of commands like `"start"`, `"stop"`, or `"status"`, explicitly check for those strings:

```c
if (strcmp(argv[1], "start") != 0 &&
    strcmp(argv[1], "stop") != 0 &&
    strcmp(argv[1], "status") != 0) {
    printf("Invalid command '%s'. Use start, stop, or status.\n", argv[1]);
    return 1;
}
```

11.3.7 Handling Optional Arguments and Flags

Many utilities accept optional flags (e.g., `-h` for help). You can scan `argv[]` for these options before processing positional arguments.

Example:

```c
for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
        printf("Usage: %s [options] <args>\n", argv[0]);
        return 0;
    }
}
```

11.3.8 Third-Party Libraries for Advanced Parsing

For complex CLI tools, manual argument parsing can get cumbersome. Libraries like:

- **`getopt()` / `getopt_long()`**: Available in POSIX, helps parse flags and options.
- **argp**: GNU extension for more sophisticated parsing.
- **Others**: External libraries like `argtable`, `cxxopts` (C++), or `CLI11`.

These tools support features like:

- Named options with values.
- Optional and required arguments.
- Automatic help message generation.

For beginner projects, manual parsing as shown above is sufficient and instructive.

11.3.9 Summary

Parsing and validating command-line arguments involves:

- Checking argument counts with `argc`.
- Converting strings safely to integers and floats (`strtol`, `strtof`).
- Validating that inputs are within allowed ranges.

- Providing clear error and usage messages.
- Sanitizing inputs to avoid invalid or malicious data.

By following these practices, your C programs will be robust, user-friendly, and reliable.

Chapter 12.

Bitwise Operations

1. Bitwise AND, OR, XOR, NOT, Shifts
2. Bit Masking and Flag Operations
3. Practical Bitwise Applications

12 Bitwise Operations

12.1 Bitwise AND, OR, XOR, NOT, Shifts

In C programming, bitwise operators allow you to manipulate data at the bit level, which can be extremely efficient for tasks like setting flags, encoding information, or optimizing performance. Unlike arithmetic operators that work on whole numbers, bitwise operators operate directly on the individual bits of an integer value.

Let's explore the basic bitwise operators in C: **AND (`&`)**, **OR (`|`)**, **XOR (`^`)**, **NOT (`~`)**, and the shift operators **left shift (`<<`)** and **right shift (`>>`)**.

12.1.1 Bitwise AND (`&`)

The bitwise AND operator compares each bit of two integers and returns a new integer where each bit is set to `1` only if **both** corresponding bits are `1`.

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```c
unsigned char a = 12;   // binary: 00001100
unsigned char b = 10;   // binary: 00001010
unsigned char c = a & b; // result: 00001000 (decimal 8)
printf("%d\n", c);      // Outputs: 8
```

Here, only the bits that are `1` in both `a` and `b` remain set in `c`.

12.1.2 Bitwise OR (`|`)

The bitwise OR operator returns a new integer where each bit is `1` if **either** corresponding bit is `1`.

A	B	A | B
0	0	0
0	1	1
1	0	1

A	B	A | B
1	1	1

Example:

```
unsigned char a = 12;   // 00001100
unsigned char b = 10;   // 00001010
unsigned char c = a | b; // 00001110 (decimal 14)
printf("%d\n", c);     // Outputs: 14
```

Here, any bit set in either operand is set in the result.

12.1.3 Bitwise XOR (^)

The XOR (exclusive OR) operator returns `1` if **only one** of the corresponding bits is `1`, but `0` if both are the same.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```
unsigned char a = 12;   // 00001100
unsigned char b = 10;   // 00001010
unsigned char c = a ^ b; // 00000110 (decimal 6)
printf("%d\n", c);     // Outputs: 6
```

XOR is often used to toggle bits: bits that differ become `1`, bits that are the same become `0`.

12.1.4 Bitwise NOT (~)

The NOT operator is a unary operator that flips every bit of its operand — changing `1` bits to `0` and vice versa.

Example:

```
unsigned char a = 12;    // 00001100
unsigned char c = ~a;    // 11110011 (decimal 243 if unsigned)
printf("%u\n", c);       // Outputs: 243
```

Note: Because `unsigned char` is 8 bits, the result flips all 8 bits. For signed types, the result depends on the system's representation and can be negative.

12.1.5 Left Shift (<<)

The left shift operator shifts bits to the left by a specified number of positions, filling the vacant bits on the right with zeros. This effectively multiplies the number by 2 for each shift.

Example:

```
unsigned char a = 3;    // 00000011
unsigned char c = a << 2; // 00001100 (decimal 12)
printf("%d\n", c);     // Outputs: 12
```

Shifting left by 2 moves bits two positions left. The bits shifted out on the left are discarded.

12.1.6 Right Shift (>>)

The right shift operator shifts bits to the right by a specified number of positions. The behavior for filling the left bits depends on the type:

- For **unsigned types**, zeros are filled in (logical shift).
- For **signed types**, the sign bit may be replicated (arithmetic shift), but this is implementation-defined.

Example:

```
unsigned char a = 12;   // 00001100
unsigned char c = a >> 2; // 00000011 (decimal 3)
printf("%d\n", c);     // Outputs: 3
```

Shifting right by 2 moves bits two positions right, discarding the bits on the right.

12.1.7 Practical Bit Manipulation Examples

Bitwise operators enable you to manipulate specific bits for purposes like setting, clearing, toggling, or testing bits.

Assuming we work with an 8-bit flag variable:

```
unsigned char flags = 0; // 00000000
```

- **Set a bit (turn bit 3 ON):**

```
flags |= (1 << 3);  // Set bit 3 to 1
```

- **Clear a bit (turn bit 3 OFF):**

```
flags &= ~(1 << 3); // Clear bit 3 to 0
```

- **Toggle a bit (flip bit 3):**

```
flags ^= (1 << 3);  // Toggle bit 3
```

- **Test if a bit is set:**

```
if (flags & (1 << 3)) {
    printf("Bit 3 is set\n");
} else {
    printf("Bit 3 is not set\n");
}
```

12.1.8 Summary Table of Bitwise Operators

Operator	Description	Example	Result Example (decimal)
`&`	AND	`12 & 10`	8
`|`	OR	`12 | 10`	14
`^`	XOR	`12 ^ 10`	6
`~`	NOT (bitwise complement)	`~12`	243 (unsigned char)
`<<`	Left shift	`3 << 2`	12
`>>`	Right shift	`12 >> 2`	3

12.1.9 Important Notes

- Bitwise operations are only valid for integral types (`int`, `char`, `long`, etc.).
- Be cautious with signed types and shifts — results may be implementation-defined.
- Bitwise operators are powerful for low-level programming like hardware control, encryption, compression, and performance optimization.

Bitwise operators unlock the power to manipulate data at its most fundamental level, giving you control over individual bits for efficient and precise programming.

12.2 Bit Masking and Flag Operations

Bit masking is a powerful technique in C programming that uses bitwise operations to selectively modify, set, clear, or check specific bits within an integer variable. This technique is essential when you want to manage multiple Boolean flags or options compactly and efficiently, especially in low-level programming, configuration management, or controlling hardware features.

12.2.1 What Is Bit Masking?

A **bit mask** is a binary pattern (usually represented as a constant) used to isolate or manipulate particular bits within a variable. By combining masks with bitwise operators such as AND (`&`), OR (`|`), XOR (`^`), and NOT (`~`), you can control individual bits without affecting the rest.

Imagine you have a single 8-bit variable, each bit representing a different feature or permission:

```
Bit positions: 7 6 5 4 3 2 1 0
Flags:         F7 F6 F5 F4 F3 F2 F1 F0
```

You can define constants representing each flag as bit masks:

```c
#define FLAG_0  (1 << 0)  // 00000001
#define FLAG_1  (1 << 1)  // 00000010
#define FLAG_2  (1 << 2)  // 00000100
#define FLAG_3  (1 << 3)  // 00001000
// ... and so on
```

These masks help you isolate or modify specific bits by applying bitwise operators.

12.2.2 Setting Flags (Turning Bits ON)

To **set** (turn ON) one or more bits, use the bitwise OR operator (`|`). ORing a variable with a mask ensures that the bits corresponding to the mask are set to 1, leaving other bits unchanged.

```c
unsigned char config = 0; // All flags initially off (00000000)

// Set FLAG_1 and FLAG_3
config = config | FLAG_1 | FLAG_3;
// Now config is 00001010 (decimal 10)
```

This operation is very common when enabling specific features or permissions.

12.2.3 Clearing Flags (Turning Bits OFF)

To **clear** (turn OFF) specific bits, use the bitwise AND operator (`&`) combined with the NOT operator (`~`) on the mask.

```
// Clear FLAG_1 (turn bit 1 off)
config = config & ~FLAG_1;
// If config was 00001010, now it becomes 00001000 (decimal 8)
```

By negating the mask (`~FLAG_1`), you create a mask with all bits set except the one you want to clear, and ANDing with it clears that bit without affecting others.

12.2.4 Toggling Flags (Flipping Bits)

Toggling a bit means switching its state: if it was ON, turn it OFF, and vice versa. Use the XOR operator (`^`) with the mask to toggle bits.

```
// Toggle FLAG_3
config = config ^ FLAG_3;
// If config was 00001000, toggling bit 3 results in 00000000
```

This operation is useful for switches or features that need to alternate states.

12.2.5 Testing Flags (Checking Bits)

To **test** whether a bit is set, use the bitwise AND operator (`&`) with the mask and compare the result with zero.

```
if (config & FLAG_3) {
    printf("Flag 3 is set\n");
} else {
    printf("Flag 3 is not set\n");
}
```

If the AND result is non-zero, the bit is ON; otherwise, it's OFF.

12.2.6 Example Scenario: Managing Permissions

Suppose you are creating a user permission system where different bits represent different permissions:

```
#define PERM_READ   (1 << 0)  // 00000001
#define PERM_WRITE  (1 << 1)  // 00000010
#define PERM_EXEC   (1 << 2)  // 00000100
```

```c
unsigned char permissions = 0;

// Grant read and write permissions
permissions |= PERM_READ | PERM_WRITE;

// Check if execute permission is granted
if (permissions & PERM_EXEC) {
    printf("Execute permission granted\n");
} else {
    printf("Execute permission denied\n");
}

// Revoke write permission
permissions &= ~PERM_WRITE;
```

This compact representation uses a single variable to hold multiple permissions, enabling efficient storage and quick operations.

12.2.7 Why Use Bit Masking?

- **Efficiency**: Storing multiple Boolean flags in a single integer saves memory.
- **Performance**: Bitwise operations are very fast at the hardware level.
- **Portability**: Works consistently across systems.
- **Simplicity**: Enables clean, concise code for flag management.

12.2.8 Tips for Using Bit Masks Effectively

- Always define masks using left-shift notation `(1 << n)` to avoid mistakes and improve clarity.
- Group related flags together to improve readability.
- Use descriptive constant names to clarify purpose.
- Document which bit positions are used for what flags to avoid conflicts.
- Consider using `enum` combined with bit shifts for more structured code.

12.2.9 Summary: Bit Masking Operations

Operation	Syntax	Effect	
Set bit(s)	‘var	= MASK;‘	Turns bit(s) ON
Clear bit(s)	`var &= ~MASK;`	Turns bit(s) OFF	
Toggle bit(s)	`var ^= MASK;`	Flips bit(s)	

Operation	Syntax	Effect
Test bit(s)	`(var & MASK) != 0`	Checks if bit(s) are ON

12.2.10 Conclusion

Bit masking and flag operations provide a powerful toolset to manipulate individual bits within variables efficiently. This approach is fundamental in systems programming, device control, and anywhere compact, performant flag management is needed. Mastering bit masks lets you write code that is both expressive and optimized.

12.3 Practical Bitwise Applications

Bitwise operations in C are not just academic curiosities — they have many practical uses in real-world programming. Leveraging bitwise techniques can improve both memory efficiency and performance, especially in systems programming, embedded devices, networking, and low-level data manipulation. In this section, we explore several common applications with illustrative examples.

12.3.1 Packing Multiple Boolean Values into a Single Integer

Suppose you need to track multiple true/false states, such as feature toggles or sensor statuses. Instead of using separate `bool` or `int` variables for each flag, you can **pack multiple Boolean values into the bits of a single integer**.

This reduces memory usage and allows fast checks and updates using bitwise operations.

Full runnable code:

```c
#include <stdio.h>

// Define bit positions for each flag
#define FLAG_A (1 << 0)  // bit 0
#define FLAG_B (1 << 1)  // bit 1
#define FLAG_C (1 << 2)  // bit 2

int main() {
    unsigned char flags = 0;  // All flags initially off

    // Set FLAG_A and FLAG_C
    flags |= FLAG_A | FLAG_C;
```

```
    // Check if FLAG_B is set
    if (flags & FLAG_B) {
        printf("FLAG_B is ON\n");
    } else {
        printf("FLAG_B is OFF\n");
    }

    // Toggle FLAG_A
    flags ^= FLAG_A;

    // Print current flags as binary
    printf("Flags: %u\n", flags);

    return 0;
}
```

Output:

```
FLAG_B is OFF
Flags: 4
```

Here, `flags` is an 8-bit variable holding three Boolean states, saving memory and enabling efficient operations.

12.3.2 Efficient Arithmetic Using Bit Shifts

Bitwise shifts (`<<` and `>>`) offer **faster alternatives to multiplication and division by powers of two**.

```
int x = 8;

// Multiply by 2 using shift left
int y = x << 1;  // Equivalent to 8 * 2 = 16

// Divide by 4 using shift right
int z = x >> 2;  // Equivalent to 8 / 4 = 2

printf("y = %d, z = %d\n", y, z);
```

Output:

```
y = 16, z = 2
```

Shifts are often optimized by compilers into single CPU instructions, making them faster than arithmetic operations, especially in performance-critical code.

Note: Use shifts only with powers of two and unsigned integers to avoid undefined behavior.

12.3.3 Checksum and Parity Calculations

Bitwise operators are ideal for calculating **checksums** or **parity bits** for error detection.

Example: Simple parity bit calculation

A parity bit is a bit that indicates whether the number of set bits in data is even or odd.

Full runnable code:

```c
#include <stdio.h>

// Function to calculate even parity bit for an 8-bit value
int parity(unsigned char value) {
    int count = 0;
    for (int i = 0; i < 8; i++) {
        count += (value >> i) & 1;  // Extract each bit
    }
    return count % 2 == 0;  // Returns 1 if even parity
}

int main() {
    unsigned char data = 0b10110101;
    printf("Parity bit (even parity): %d\n", parity(data));
    return 0;
}
```

Output:

```
Parity bit (even parity): 1
```

This approach uses shifts and bitwise AND to count bits efficiently.

12.3.4 Encoding and Decoding Data Fields

Bitwise operations enable **packing multiple smaller values into a single integer** by assigning different bit ranges for each value.

For example, consider a system where you want to store a color in a single 32-bit integer using 8 bits each for red, green, blue, and alpha (transparency):

Full runnable code:

```c
#include <stdio.h>

unsigned int packColor(unsigned char r, unsigned char g, unsigned char b, unsigned char a) {
    return (a << 24) | (r << 16) | (g << 8) | b;
}

void unpackColor(unsigned int color, unsigned char *r, unsigned char *g, unsigned char *b, unsigned cha
    *a = (color >> 24) & 0xFF;
    *r = (color >> 16) & 0xFF;
```

```
    *g = (color >> 8) & 0xFF;
    *b = color & 0xFF;
}

int main() {
    unsigned char red = 200, green = 100, blue = 50, alpha = 255;
    unsigned int color = packColor(red, green, blue, alpha);

    printf("Packed color: 0x%X\n", color);

    unsigned char r, g, b, a;
    unpackColor(color, &r, &g, &b, &a);
    printf("Unpacked RGBA: %d, %d, %d, %d\n", r, g, b, a);

    return 0;
}
```

Output:

```
Packed color: 0xFFC86432
Unpacked RGBA: 200, 100, 50, 255
```

By shifting bits into their correct position and masking, you efficiently encode and decode complex data structures.

12.3.5 Flag Management in Configuration Settings

Bitwise flags can store multiple configuration options compactly and check them efficiently:

```
#define CONFIG_ENABLE_LOGGING   (1 << 0)
#define CONFIG_ENABLE_DEBUG     (1 << 1)
#define CONFIG_USE_ENCRYPTION   (1 << 2)

unsigned int config = 0;

// Enable logging and encryption
config |= CONFIG_ENABLE_LOGGING | CONFIG_USE_ENCRYPTION;

// Check if debugging is enabled
if (config & CONFIG_ENABLE_DEBUG) {
    printf("Debugging enabled\n");
} else {
    printf("Debugging disabled\n");
}
```

Using bit flags avoids the need for many separate Boolean variables, making your code scalable.

12.3.6 Benefits of Bitwise Operations

- **Memory savings:** Pack multiple flags or small values in a single variable.
- **Speed:** Bitwise operations are extremely fast and often map to single CPU instructions.
- **Compact data representation:** Ideal for file formats, network protocols, and embedded systems.
- **Clear semantics:** Bit flags clearly indicate presence/absence of options.

12.3.7 Conclusion

Bitwise operations are a foundational skill for C programmers who want to write efficient, compact, and high-performance code. From packing multiple Boolean values, performing fast arithmetic, implementing error detection, to encoding complex data, bitwise techniques unlock many powerful solutions in both systems and application programming.

By mastering bitwise operations and understanding their practical applications, you gain the ability to solve problems elegantly and with optimal resource use.

Chapter 13.

Preprocessor Directives

1. `#define`, `#include`, `#ifdef`, `#ifndef`
2. Macros and Constants
3. Conditional Compilation

13 Preprocessor Directives

13.1 `#define`, `#include`, `#ifdef`, `#ifndef`

Before your C program is compiled, it passes through a special phase called **preprocessing**. The **C preprocessor** handles directives that begin with the `#` symbol. These directives instruct the compiler to perform text substitution, include files, and conditionally compile code — all before the actual compilation begins.

In this section, we'll explore four important preprocessor directives:

- `#define`
- `#include`
- `#ifdef`
- `#ifndef`

Understanding these directives is essential for writing modular, readable, and maintainable C code.

13.1.1 `#define` Defining Constants and Macros

The `#define` directive allows you to create symbolic names or macros that the preprocessor replaces throughout your source code.

Syntax:

```
#define NAME replacement_text
```

Example:

```
#define PI 3.14159
#define MAX_SIZE 100
```

Here, every occurrence of `PI` in your code will be replaced with `3.14159` by the preprocessor before compilation.

You can also define **macros with parameters** to create simple inline functions:

```
#define SQUARE(x) ((x) * (x))
```

Usage:

```
int result = SQUARE(5);  // expands to ((5) * (5))
```

Important notes:

- Macros are replaced literally; parentheses around parameters and the whole macro body help avoid bugs related to operator precedence.
- Macros do not have a type and do not perform type checking.
- Avoid side effects in macro arguments, e.g., `SQUARE(i++)` can behave unexpectedly.

13.1.2 `#include` Including Header Files

The `#include` directive inserts the contents of another file into your source code at the point where the directive appears. This is commonly used to include header files (`.h`) that contain declarations and macros.

Syntax:

```
#include <filename>  // For system headers
#include "filename"  // For user headers
```

Example:

```
#include <stdio.h>  // Standard library header
#include "myheader.h"  // User-defined header
```

- When you use angle brackets `< >`, the compiler searches in the system include paths.
- When you use double quotes `" "`, the compiler searches in the current directory first, then system paths.

Including headers lets you reuse code and share function prototypes, type definitions, and constants across multiple source files.

13.1.3 `#ifdef` and `#ifndef` Conditional Compilation

Sometimes, you want to **compile code only if a certain macro is defined or undefined**. The preprocessor provides conditional directives for this:

- `#ifdef` — "if defined"
- `#ifndef` — "if not defined"

These allow selective compilation, useful for portability, debugging, or enabling/disabling features.

Syntax:

```
#ifdef MACRO_NAME
    // code here is compiled only if MACRO_NAME is defined
#endif

#ifndef MACRO_NAME
    // code here is compiled only if MACRO_NAME is not defined
#endif
```

Example:

```
#define DEBUG

#ifdef DEBUG
    printf("Debug mode is ON\n");
#endif
```

If `DEBUG` is defined, the print statement will be included in the compiled program; otherwise, it will be omitted.

13.1.4 Header Guards Preventing Multiple Inclusions

One very common pattern using `#ifndef` is the **header guard**, which prevents a header file from being included multiple times in a project. Multiple inclusions can cause errors like redefinition of types or functions.

Typical header guard structure:

```
#ifndef MYHEADER_H
#define MYHEADER_H

// Header file contents (declarations, macros, etc.)

#endif  // MYHEADER_H
```

How it works:

1. The first time the header is included, `MYHEADER_H` is not defined, so the content between `#ifndef` and `#endif` is processed, and `MYHEADER_H` is defined.
2. On subsequent inclusions, `MYHEADER_H` is already defined, so the preprocessor skips the contents, preventing duplicate definitions.

Example header file `myheader.h`:

```
#ifndef MYHEADER_H
#define MYHEADER_H

void printMessage();

#endif
```

13.1.5 Summary and Best Practices

- Use `#define` to create meaningful names for constants or simple macros.
- Use `#include` to organize code into multiple files and include libraries.
- Use `#ifdef` and `#ifndef` for conditional compilation to support different environments or debugging.
- Always use header guards in your header files to avoid multiple inclusion errors.
- Avoid complex macros when possible; prefer functions for better type safety and debugging.

13.1.6 Example Putting It All Together

```c
// myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H

#define VERSION 1.0

void greet(void);

#endif

// main.c
#include <stdio.h>
#include "myheader.h"

void greet(void) {
#ifdef VERSION
    printf("Program version: %.1f\n", VERSION);
#else
    printf("Version unknown\n");
#endif
}

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

This program prints the program version using macros and conditional compilation, demonstrating how these preprocessor directives control compilation and program behavior.

By mastering these preprocessor directives, you gain powerful tools to write flexible, portable, and maintainable C programs.

13.2 Macros and Constants

In C programming, both **macros** and **constants** serve the purpose of representing fixed values that do not change during program execution. However, they work very differently and have their own benefits and pitfalls. Understanding the distinction is crucial for writing clean, safe, and maintainable code.

13.2.1 What Are Macros?

Macros are **preprocessor directives** defined with `#define`. During the preprocessing phase, the preprocessor replaces every occurrence of the macro name with its defined value or expression before compilation begins.

Example of a simple macro:

```
#define PI 3.14159
```

Whenever you write `PI` in your code, the preprocessor replaces it with `3.14159`.

Macros are not variables; they don't have types, and they exist only as text substitutions.

13.2.2 Function-Like Macros

Macros can also behave like functions by accepting parameters. These **function-like macros** are a powerful tool for inline code substitution.

Example:

```
#define SQUARE(x) ((x) * (x))
```

If you call `SQUARE(5)`, the preprocessor replaces it with `((5) * (5))`, which the compiler then processes as regular code.

Important: Parentheses are critical around parameters and the entire expression to ensure correct operator precedence.

13.2.3 Benefits of Macros

- **No runtime overhead:** Macros are expanded inline, so there's no function call overhead.
- **Flexible:** Can be used for both constants and code snippets.
- **Works with any type:** Since they're just text substitutions, macros can be used without worrying about types.

13.2.4 Pitfalls and Dangers of Macros

Despite their power, macros can introduce subtle bugs and maintenance issues:

Lack of Type Checking

Macros do not perform type checking, which can lead to unexpected results.

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

If you call `MAX(3, 4.5)`, it works, but if you call `MAX("hello", "world")`, the result is undefined behavior, and the compiler won't warn you.

Side Effects

Arguments in macros can be evaluated multiple times, causing unintended side effects.

```
int i = 3;
int result = SQUARE(i++);  // Expands to ((i++) * (i++))
```

Here, `i++` is incremented twice, which can lead to unpredictable behavior.

13.2.5 When to Use `const` Variables

Modern C supports `const` variables, which are safer alternatives to simple value macros for constants.

Example:

```
const double pi = 3.14159;
```

Advantages of `const` variables over macros:

- **Type safety:** `const` variables have types, so the compiler can check usage.
- **Scope control:** Unlike macros, `const` variables obey scope rules.
- **Debugger friendly:** You can inspect `const` variables in a debugger, but macros are replaced before compilation and don't exist in debug information.

However, `const` variables may occupy storage, whereas macros typically do not (though modern compilers optimize this). For small constants, the difference is usually negligible.

13.2.6 Using `enum` for Integral Constants

Another alternative for defining related integral constants is the `enum` type.

Example:

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

Benefits:

- Grouping related constants improves code readability.
- `enum` constants have type safety and are scoped within the enum.
- Can be used in switch statements for better clarity.

13.2.7 Comparing Macros and Constants: Summary

Feature	Macros (`#define`)	Constants (`const`, `enum`)
Type checking	None	Yes
Debugging	Not visible (replaced before compile)	Visible and inspectable
Scope	Global (no scope)	Scoped according to variable or enum
Side effects	Possible (especially in function-like macros)	None
Storage	None (text substitution)	May use storage (optimized by compiler)
Use for functions	Can define inline code (no type checking)	Use `inline` functions instead

13.2.8 Practical Examples

Simple constant macro:

```
#define BUFFER_SIZE 256
char buffer[BUFFER_SIZE];
```

Safer constant variable:

```
const int buffer_size = 256;
char buffer[buffer_size];
```

Function-like macro:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int max_value = MAX(10, 20);
```

Safer alternative with inline function (C99):

```
static inline int max(int a, int b) {
    return (a > b) ? a : b;
}
int max_value = max(10, 20);
```

13.2.9 Best Practices

- Use **macros for conditional compilation** or where textual replacement is necessary.
- Avoid complex macros that behave like functions; prefer **inline functions** for type safety and clarity.
- Use `const` variables or `enum` constants for fixed values to leverage compiler checks and

improve maintainability.
- Always add parentheses around macro parameters and the whole macro body to avoid operator precedence issues.
- Be careful with macro arguments that have side effects.

13.2.10 Conclusion

Macros are a powerful feature of C that enable textual substitution and simple code generation, but they come with risks related to type safety and unexpected side effects. Constants declared with `const` or `enum` provide safer, more maintainable alternatives for defining fixed values. Balancing the use of macros and constants is key to writing clear, robust C code.

13.3 Conditional Compilation

In C programming, **conditional compilation** is a powerful feature that allows you to include or exclude parts of your code during the compilation process based on certain conditions. This capability is controlled through preprocessor directives and enables writing flexible, configurable, and portable programs.

13.3.1 Why Use Conditional Compilation?

There are several practical scenarios where conditional compilation is invaluable:

- **Debugging:** Enable or disable debug logging without changing source code logic.
- **Platform-Specific Code:** Compile different code for Windows, Linux, or embedded systems.
- **Feature Flags:** Include or exclude features depending on compile-time options.
- **Performance Tuning:** Omit expensive checks or instrumentation in release builds.

By controlling which code is compiled, you avoid cluttering your runtime logic with unnecessary checks or complicated configurations.

13.3.2 Key Directives: `#if`, `#elif`, `#else`, and `#endif`

These directives guide the preprocessor to conditionally compile code based on expressions or macro definitions.

- `#if` checks if a condition evaluates to true (non-zero).
- `#elif` ("else if") checks an alternative condition if previous `#if` or `#elif` was false.
- `#else` includes code if all prior conditions are false.
- `#endif` ends the conditional block.

13.3.3 Example 1: Debugging Toggle

You might want to add debug messages that only appear in debug builds.

```
#define DEBUG 1  // Change to 0 to disable debug code

int main() {
    int x = 10;

#if DEBUG
    printf("Debug: x = %d\n", x);
#endif

    // Rest of program
    return 0;
}
```

If `DEBUG` is defined as `1`, the debug `printf` is compiled. If it's `0` or undefined, the debug message is omitted altogether.

13.3.4 Example 2: Platform-Specific Code

When writing code for multiple platforms, conditional compilation helps maintain a single source file.

```
#ifdef _WIN32
    // Windows-specific code
    printf("Running on Windows\n");
#elif defined(__linux__)
    // Linux-specific code
    printf("Running on Linux\n");
#else
    // Other platforms
    printf("Running on an unknown platform\n");
#endif
```

Here, the program includes only the code relevant to the platform it's being compiled on, identified by predefined macros like `_WIN32` and `__linux__`.

13.3.5 Checking for Macro Definitions: `#ifdef` and `#ifndef`

The `#ifdef` (if defined) and `#ifndef` (if not defined) directives are convenient for checking whether a macro exists.

Example:

```
#ifndef MAX_BUFFER_SIZE
#define MAX_BUFFER_SIZE 1024
#endif
```

This pattern prevents redefinition errors by defining `MAX_BUFFER_SIZE` only if it hasn't been defined before, commonly used in **header guards**.

13.3.6 Combining Conditions with `#if`

The `#if` directive can evaluate expressions involving macros, enabling more complex conditions.

```
#define VERSION 2

#if VERSION >= 2
    printf("Version 2 or higher features enabled.\n");
#else
    printf("Legacy version.\n");
#endif
```

13.3.7 Practical Use: Feature Flags

Suppose you want to compile with or without certain features:

```
#define FEATURE_X 1

#if FEATURE_X
    void featureX() {
        printf("Feature X is enabled\n");
    }
#else
    void featureX() {
        printf("Feature X is disabled\n");
    }
#endif
```

By changing the `FEATURE_X` macro, you control which implementation is included.

13.3.8 Supporting Portable and Configurable Codebases

Using conditional compilation throughout your code allows you to:

- **Maintain one codebase** for different environments.
- **Easily enable or disable features** at compile time.
- **Reduce runtime overhead** by excluding unused code.
- **Create debug or release builds** without modifying source logic.

This approach is widely used in libraries, operating systems, and large-scale projects to keep code maintainable and adaptable.

13.3.9 Tips for Using Conditional Compilation Effectively

- **Keep conditions clear and simple:** Complex nested conditions can make code hard to read.
- **Use meaningful macro names:** Names like `DEBUG`, `FEATURE_X`, or `PLATFORM_WINDOWS` clarify intent.
- **Avoid scattering conditionals:** Group related conditional code to improve readability.
- **Combine with header guards:** Prevent multiple inclusions and redefinitions using `#ifndef` guards.
- **Document flags:** Clearly document the purpose of macros used for conditional compilation.

13.3.10 Summary

Conditional compilation using `#if`, `#elif`, `#else`, and `#endif` directives allows C programmers to write flexible, configurable, and portable code. By selectively including or excluding code at compile time, you can tailor your programs for debugging, multiple platforms, feature toggles, and more. When used wisely, this technique greatly improves maintainability and performance.

Chapter 14.

Enumerations and Type Aliases

1. Defining and Using `enum`
2. Practical Applications of Enums
3. `typedef` for Readable Code

14 Enumerations and Type Aliases

14.1 Defining and Using `enum`

In C programming, **enumerations** (`enum`) provide a convenient way to define a set of named integral constants. Instead of using plain integer literals scattered throughout your code, enums group related constants under meaningful names, improving code clarity, readability, and maintainability.

14.1.1 What Is an `enum`?

An `enum` (short for enumeration) is a user-defined data type that consists of a collection of named integer constants. The `enum` keyword declares this type, allowing you to refer to these constants by names instead of raw numbers.

The basic syntax is:

```
enum EnumName {
    Constant1,
    Constant2,
    Constant3,
    // ...
};
```

Here, `EnumName` is the name of the enumeration type, and each `Constant` is an identifier representing an integral value.

14.1.2 Why Use Enums?

Using enums offers several advantages:

- **Improved readability:** Names like `MONDAY`, `ERROR`, or `STATUS_OK` are easier to understand than raw numbers like `0`, `1`, or `200`.
- **Reduced errors:** It's easier to avoid mistakes when you use descriptive names rather than magic numbers.
- **Code maintainability:** If values need to change, you only update the enum definition rather than hunting down every usage.
- **Type checking:** Some compilers provide warnings or errors if you misuse enums, helping catch bugs early.

14.1.3 Simple Enum Example: Days of the Week

Let's define an enum to represent the days of the week:

```
enum Day {
    SUNDAY,    // 0
    MONDAY,    // 1
    TUESDAY,   // 2
    WEDNESDAY, // 3
    THURSDAY,  // 4
    FRIDAY,    // 5
    SATURDAY   // 6
};
```

By default, the first constant `SUNDAY` is assigned the integer value `0`, and subsequent constants increment by 1 automatically. So, `MONDAY` is `1`, `TUESDAY` is `2`, and so on.

14.1.4 Using Enums in Code

You can declare variables of the enum type and assign enum constants to them:

```
enum Day today;

today = WEDNESDAY;

if (today == WEDNESDAY) {
    printf("It's midweek!\n");
}
```

This code compares the `today` variable with the named constant `WEDNESDAY`, making the intention clear.

14.1.5 Assigning Custom Values

You are not limited to the default numbering. You can assign custom integer values explicitly:

```
enum StatusCode {
    SUCCESS = 0,
    WARNING = 100,
    ERROR = 200,
    CRITICAL = 300
};
```

In this example, `SUCCESS` is 0, `WARNING` is 100, etc. Subsequent constants without explicit values will continue incrementing from the previous one:

```
enum Example {
    A = 5,
    B,  // 6
    C = 10,
```

```
    D   // 11
};
```

Here, `B` is 6 because it follows 5, and `D` is 11 because it follows 10.

14.1.6 Enum and Underlying Type

In C, the constants in an enum are essentially integers (`int` by default), but they improve code expressiveness by grouping related values together.

You can use enum constants anywhere an integer is valid, for example in switch statements or arithmetic (though typically you use them for comparisons).

14.1.7 Example: Status Codes in a Program

Here is a practical example using enums to represent operation statuses:

Full runnable code:

```c
#include <stdio.h>

enum Status {
    OK = 0,
    ERROR = 1,
    TIMEOUT = 2
};

void checkStatus(enum Status s) {
    if (s == OK) {
        printf("Operation successful.\n");
    } else if (s == ERROR) {
        printf("An error occurred.\n");
    } else if (s == TIMEOUT) {
        printf("Operation timed out.\n");
    }
}

int main() {
    enum Status currentStatus = ERROR;
    checkStatus(currentStatus);
    return 0;
}
```

Using enum here makes the status codes self-documenting and the code easier to read.

14.1.8 Summary

- The `enum` keyword defines named integer constants.
- Enums enhance code readability and maintainability by replacing magic numbers with meaningful names.
- Constants start at 0 by default and increment by 1 unless assigned custom values.
- You can use enums in variables, comparisons, and control structures.
- Assigning explicit values allows for flexible mapping to meaningful numbers like error codes or flags.

Enums are a foundational tool in C programming to write clear, maintainable, and bug-resistant code by giving names to sets of related integral constants.

14.2 Practical Applications of Enums

Enums are a powerful tool in C that help you organize and manage related sets of constants with meaningful names. Beyond just improving readability, enums play a key role in structuring real-world programs, such as state machines, error handling, menu systems, and flag management. This section explores these practical applications and shows how enums simplify code while making it easier to debug and maintain.

14.2.1 Managing State Machines

A common use case for enums is to represent the states of a program or device in a **state machine**. Each state is given a descriptive name, which makes the control flow clear.

Example: Traffic Light Controller

```
enum TrafficLight {
    RED,
    YELLOW,
    GREEN
};

void handleTrafficLight(enum TrafficLight state) {
    switch(state) {
        case RED:
            printf("Stop!\n");
            break;
        case YELLOW:
            printf("Prepare to stop.\n");
            break;
        case GREEN:
            printf("Go!\n");
            break;
        default:
```

```
            printf("Invalid state.\n");
    }
}

int main() {
    enum TrafficLight current = RED;
    handleTrafficLight(current);
    return 0;
}
```

Using enums here avoids confusing numeric values like `0`, `1`, or `2` and instead clearly expresses the program's logic.

14.2.2 Error Codes and Status Reporting

When functions need to return status information, enums provide a neat way to define all possible outcomes.

Example: File Operation Status Codes

```
enum FileStatus {
    FILE_OK = 0,
    FILE_NOT_FOUND,
    FILE_PERMISSION_DENIED,
    FILE_READ_ERROR
};

enum FileStatus openFile(const char* filename) {
    // Simulated file opening logic
    if (filename == NULL) return FILE_NOT_FOUND;
    // Other checks...
    return FILE_OK;
}

int main() {
    enum FileStatus status = openFile("data.txt");
    if (status == FILE_OK) {
        printf("File opened successfully.\n");
    } else {
        printf("Error opening file: %d\n", status);
    }
    return 0;
}
```

Here, enums give names to error codes, which makes error handling more intuitive and less error-prone.

14.2.3 Menu Options and User Commands

Enums can represent user choices in menus or commands in CLI utilities, replacing magic numbers with descriptive names.

Example: Simple Calculator Operations

```
enum Operation {
    ADD = 1,
    SUBTRACT,
    MULTIPLY,
    DIVIDE
};

void performOperation(enum Operation op, int a, int b) {
    switch (op) {
        case ADD:
            printf("%d + %d = %d\n", a, b, a + b);
            break;
        case SUBTRACT:
            printf("%d - %d = %d\n", a, b, a - b);
            break;
        case MULTIPLY:
            printf("%d * %d = %d\n", a, b, a * b);
            break;
        case DIVIDE:
            if (b != 0)
                printf("%d / %d = %d\n", a, b, a / b);
            else
                printf("Cannot divide by zero!\n");
            break;
        default:
            printf("Invalid operation.\n");
    }
}

int main() {
    enum Operation op = MULTIPLY;
    performOperation(op, 6, 3);
    return 0;
}
```

Using enums here improves readability, especially in switch statements, compared to using raw numeric codes.

14.2.4 Managing Flags and Options

Enums can also represent **flags** or options, often combined with bitwise operations for compact storage.

Example: File Permissions

```
enum FilePermissions {
    READ = 1 << 0,    // 1
```

```
    WRITE = 1 << 1,   // 2
    EXECUTE = 1 << 2  // 4
};

int main() {
    int permissions = READ | EXECUTE;  // User can read and execute

    if (permissions & READ) {
        printf("Read permission granted.\n");
    }
    if (permissions & WRITE) {
        printf("Write permission granted.\n");
    } else {
        printf("Write permission denied.\n");
    }
    if (permissions & EXECUTE) {
        printf("Execute permission granted.\n");
    }
    return 0;
}
```

Here, enums give meaningful names to bit flags, improving the clarity of bitwise logic.

14.2.5 Advantages of Using Enums in Real-World Code

- **Improved Debugging:** When debugging, seeing symbolic names (via enum constants) instead of numeric values makes the program state easier to understand.
- **Self-Documenting Code:** Enums convey intent clearly. Reading `if (status == ERROR)` is much easier to comprehend than `if (status == 1)`.
- **Reduced Magic Numbers:** Avoiding hardcoded numbers throughout code reduces bugs and makes refactoring simpler.
- **Facilitates Switch-Case Constructs:** Enums work seamlessly with `switch` statements, making control flows explicit and manageable.
- **Compiler Checks:** Some compilers warn when switch cases don't cover all enum constants, helping catch logic errors early.

14.2.6 Summary

Enums are essential tools for organizing related constants and states in your programs. Whether managing states, error codes, menu options, or bitwise flags, enums improve code clarity and robustness. Using enums leads to code that is easier to read, maintain, and debug — a clear win for any C programmer.

14.3 `typedef` for Readable Code

In C programming, `typedef` is a powerful keyword that allows you to create aliases—or alternative names—for existing data types. By using `typedef`, you can make your code shorter, cleaner, and easier to understand. This is especially useful when working with complex types such as structures, pointers, and enums.

14.3.1 What Is `typedef`?

The `typedef` keyword creates a new type name that represents an existing data type. It does **not** create a new data type, but simply gives a new name to an existing one.

Syntax:

```
typedef existing_type new_type_name;
```

14.3.2 Simplifying Primitive Types

You can use `typedef` to make primitive types more meaningful in context. For example:

Without typedef:

```
unsigned int distance;
```

With typedef:

```
typedef unsigned int Distance;
Distance d1, d2;
```

This improves readability, especially in code where the same type appears repeatedly in a specific context.

14.3.3 Typedef with Structures

Structures often require verbose syntax. `typedef` simplifies this significantly.

Without typedef:

```
struct Point {
    int x;
    int y;
};

struct Point p1;
```

With typedef:

```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
```

Now you can declare variables of type `Point` directly, without repeating `struct`.

14.3.4 Typedef with Pointers

`typedef` can be used to simplify pointer declarations, which can often be confusing.

Without typedef:

```
int* (*funcPtr)(float);
```

With typedef:

```
typedef int* (*FunctionPointer)(float);
FunctionPointer myFunc;
```

This makes complex pointer declarations more readable and maintainable.

14.3.5 Typedef with Enums

You can also use `typedef` to alias enum types, especially when they are used frequently.

Without typedef:

```
enum Status {
    OK,
    ERROR
};

enum Status s;
```

With typedef:

```
typedef enum {
    OK,
    ERROR
} Status;

Status s;
```

This reduces verbosity and allows cleaner code when enums are used as function return types or variables.

14.3.6 Naming Conventions

When using `typedef`, follow consistent and descriptive naming conventions to avoid confusion.

- Use **CamelCase** or **PascalCase** for type names (`Point`, `FileStatus`, `Distance`).
- Avoid all-uppercase names (which are usually used for macros).
- Use `Ptr` suffix for pointer types (e.g., `typedef char* StringPtr;`).

14.3.7 `typedef` in APIs and Libraries

In API design or when writing header files for libraries, `typedef` is commonly used to abstract internal data structures. This makes it easier to change the implementation without affecting the users of the API.

Example:

```
typedef struct {
    int id;
    char name[50];
} Student;

void printStudent(Student s);
```

The user doesn't need to know how `Student` is defined internally—they just use it as a type.

14.3.8 Best Practices

- Use `typedef` to reduce complexity, but don't overuse it to the point where it hides important details.
- Avoid using `typedef` for simple types like `int` or `char` unless it adds semantic meaning.
- When naming aliases, make them intuitive to the domain (e.g., `Distance`, `Angle`, `Temperature`).

14.3.9 Summary

The `typedef` keyword enhances code clarity by allowing you to define meaningful aliases for complex or verbose types. Whether you're simplifying a `struct`, clarifying pointer syntax, or designing a clean API, `typedef` helps make your C programs easier to read, understand, and maintain.

By using `typedef` wisely, you reduce boilerplate code and help future readers—yourself included—quickly grasp the intent behind your type definitions.

Chapter 15.

Modular Programming in C

1. Splitting Code Across Multiple Files
2. Header Files and `extern` Keyword
3. Makefiles and `gcc` Compilation Flags

15 Modular Programming in C

15.1 Splitting Code Across Multiple Files

As C programs grow in size and complexity, managing everything in a single `.c` file becomes inefficient and error-prone. Modular programming offers a solution by dividing a program into separate files based on functionality. This approach enhances code readability, reusability, maintainability, and collaboration among multiple developers.

In this section, we'll explore how to split C programs across multiple source (`.c`) and header (`.h`) files, and how to organize your project structure effectively.

15.1.1 Why Modularize?

Before diving into how to split code, let's understand **why** you should:

- **Separation of concerns**: Each module can focus on one task, like file handling, math operations, or data structures.
- **Code reuse**: Common utilities can be reused in multiple projects without rewriting.
- **Improved compilation**: Only changed source files need to be recompiled, saving time.
- **Easier debugging**: Bugs are easier to isolate in smaller files.
- **Team collaboration**: Developers can work on different modules without conflicts.

15.1.2 Basic Structure of a Modular Program

A modular C program typically uses:

- **Header files** (`.h`) for declarations (function prototypes, constants, types).
- **Source files** (`.c`) for definitions (function implementations, global variables).
- A **main file** (`main.c`) that uses functions from other modules.

15.1.3 A Simple Example: Calculator Module

Let's break a small calculator program into multiple files:

`calc.h` Header File

This file contains function prototypes and any needed constants:

```c
// calc.h
#ifndef CALC_H
#define CALC_H

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
float divide(int a, int b);

#endif
```

The `#ifndef`, `#define`, and `#endif` lines are called a *header guard*, which prevents multiple inclusions.

`calc.c` Function Definitions

This file contains the implementation of the functions declared in `calc.h`.

```c
// calc.c
#include "calc.h"

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

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

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

float divide(int a, int b) {
    if (b == 0) return 0.0f;
    return (float)a / b;
}
```

Note: We include the header with quotes `"calc.h"` because it's a user-defined header.

`main.c` The Driver Program

This file contains the `main()` function that uses the calculator functions.

```c
// main.c
#include <stdio.h>
#include "calc.h"

int main() {
    int a = 10, b = 5;

    printf("Add: %d\n", add(a, b));
    printf("Subtract: %d\n", subtract(a, b));
    printf("Multiply: %d\n", multiply(a, b));
    printf("Divide: %.2f\n", divide(a, b));

    return 0;
}
```

15.1.4 Compiling Multiple Files

To compile these files together using `gcc`, use:

```
gcc main.c calc.c -o calculator
```

This tells the compiler to compile both `main.c` and `calc.c` and link them into an executable named `calculator`.

15.1.5 Guidelines for Splitting Code

1. **Group related functionality**: Functions and data that serve a common purpose should be placed together in one `.c` and `.h` file pair.
2. **Use meaningful names**: File names like `math_utils.c`, `io_utils.h`, or `config.c` help clarify their purpose.
3. **Avoid circular dependencies**: Do not include headers in each other unless absolutely necessary. If you must, use forward declarations.
4. **Keep global variables private**: Define global variables in `.c` files and use the `extern` keyword in `.h` files when they need to be accessed elsewhere.
5. **Minimal headers**: Header files should include only what is necessary for other files to use the module (function prototypes, typedefs, constants).

15.1.6 Example Use Case: Modular Student Record System

Imagine you're writing a student record management system. You might organize your project like this:

- `main.c` – Contains `main()` and user interface logic.
- `student.h`/`student.c` – Functions for managing student data.
- `fileio.h`/`fileio.c` – Functions for reading/writing to files.

Each module contains the logic for a specific part of the application, making the project cleaner and easier to manage.

15.1.7 Summary

Splitting your C code across multiple files is a best practice in professional software development. It brings structure to your projects, fosters reuse, and makes maintenance easier. By organizing your code into modules with headers and source files, you can create cleaner, more scalable, and collaborative programs.

15.2 Header Files and `extern` Keyword

In modular programming with C, **header files** (`.h`) play a critical role in organizing and sharing code between multiple source files. Along with that, the `extern` keyword allows variables and functions defined in one file to be used in others without causing multiple definition errors. This section explores how these tools help in building well-structured, maintainable C programs.

15.2.1 What Is a Header File?

A **header file** contains declarations of functions, macros, types, and global variables. It does **not** usually contain actual implementations. Header files are used to allow one source file to "know about" functions or variables defined elsewhere.

For example, consider a file `math_utils.h`:

```
// math_utils.h
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int subtract(int a, int b);

#endif
```

This header declares two functions but does not define them. The definitions would appear in a corresponding `.c` file:

```
// math_utils.c
#include "math_utils.h"

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

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

Then, another file such as `main.c` can use these functions by including the header:

```c
// main.c
#include <stdio.h>
#include "math_utils.h"

int main() {
    printf("%d\n", add(10, 5));
    return 0;
}
```

15.2.2 Header Guards: Preventing Multiple Inclusions

One common problem in C is **multiple inclusions** of the same header file, which can lead to duplicate declarations and compiler errors. To prevent this, **header guards** are used.

Here's the pattern:

```c
#ifndef HEADER_NAME
#define HEADER_NAME
// declarations go here
#endif
```

For example:

```c
#ifndef STUDENT_H
#define STUDENT_H

typedef struct {
    char name[50];
    int age;
} Student;

void print_student(Student s);

#endif
```

Header guards ensure that the contents of `student.h` are only included **once** in any given translation unit, no matter how many times the file is `#include`d.

15.2.3 Using the `extern` Keyword

By default, global variables and functions declared in one `.c` file are not visible to others. The `extern` keyword lets you declare such variables or functions in a header file so they can be **used** in multiple files, without re-defining them.

Sharing Global Variables

Suppose we define a global variable in `config.c`:

```c
// config.c
int max_users = 100;
```

To use `max_users` in other files, we declare it with `extern` in a header file:

```c
// config.h
#ifndef CONFIG_H
#define CONFIG_H

extern int max_users;

#endif
```

Then, in another source file:

```c
// main.c
#include <stdio.h>
#include "config.h"

int main() {
    printf("Max users allowed: %d\n", max_users);
    return 0;
}
```

Here, `extern` tells the compiler that `max_users` is defined elsewhere and not to allocate new storage for it.

Declaring Functions

Functions are `extern` by default, but adding the keyword explicitly can improve clarity:

```c
extern int add(int, int); // function declared but defined elsewhere
```

This is equivalent to:

```c
int add(int, int);
```

But when used consistently with variables, it helps indicate which symbols are defined outside the current file.

15.2.4 Typical Contents of a Header File

A well-structured header file usually contains:

- Function prototypes
- `#define` macros and symbolic constants
- `typedef` type definitions (like `struct`)
- `extern` variable declarations (sparingly)
- Header guards

Example:

```c
// timer.h
#ifndef TIMER_H
#define TIMER_H

#define TIMEOUT 5000

typedef struct {
    int minutes;
    int seconds;
} Timer;

extern int timer_enabled;

void start_timer(Timer *t);
void stop_timer();

#endif
```

In the corresponding `timer.c`, you would define:

```c
#include "timer.h"

int timer_enabled = 0;

void start_timer(Timer *t) {
    timer_enabled = 1;
    // logic...
}

void stop_timer() {
    timer_enabled = 0;
}
```

15.2.5 Best Practices for Header Files

- **Keep them clean**: Do not include implementation details (i.e., function bodies).
- **Avoid unnecessary includes**: Include only what is needed; excessive inclusion slows down compilation.
- **Minimal global variables**: Use `extern` only when necessary; prefer passing data explicitly through functions.
- **Use consistent header guards**: Use `#ifndef`/`#define` with meaningful names like `FILE_NAME_H`.

15.2.6 Summary

Header files are a fundamental part of modular C programming. They provide declarations that allow multiple source files to share functions, types, and variables. With the help of

`extern`, you can safely share global data across files without risking multiple definitions. Properly using header guards and keeping header files clean and focused makes your codebase more organized, reusable, and easier to maintain.

In the next section, we'll explore **Makefiles and `gcc` Compilation Flags** to automate building modular projects.

15.3 Makefiles and `gcc` Compilation Flags

As your C programs grow in size and are split across multiple `.c` and `.h` files, compiling them manually becomes tedious and error-prone. To address this, the C community uses **Makefiles**, a tool for automating the build process. Makefiles help track dependencies and recompile only what's necessary, improving efficiency and maintainability.

This section introduces Makefiles and the `gcc` (GNU Compiler Collection) compilation flags that help you control how C code is compiled.

15.3.1 Why Use Makefiles?

When building large C programs, you often have to compile several source files and link them into a single executable. Imagine a project with the following structure:

```
project/
+-- main.c
+-- utils.c
+-- utils.h
```

Compiling this manually would require:

```
gcc -o myprogram main.c utils.c
```

But what if you modify only `utils.c`? You don't want to recompile everything. A **Makefile** allows you to automate and optimize this process by:

- Defining build rules
- Managing dependencies
- Automating cleaning and recompiling

15.3.2 Basic Structure of a Makefile

A Makefile is a plain text file named `Makefile` (or `makefile`) and consists of **rules**. Each rule has the form:

```
target: dependencies
    <TAB> command
```

- **target**: usually a file to be generated (like an executable).
- **dependencies**: files that the target depends on (like `.c` or `.h` files).
- **command**: the shell command to build the target.

Important: The command line must start with a **TAB** character, not spaces.

15.3.3 Example: Simple Makefile

```
CC = gcc
CFLAGS = -Wall -g

all: myprogram

myprogram: main.o utils.o
    $(CC) $(CFLAGS) -o myprogram main.o utils.o

main.o: main.c utils.h
    $(CC) $(CFLAGS) -c main.c

utils.o: utils.c utils.h
    $(CC) $(CFLAGS) -c utils.c

clean:
    rm -f *.o myprogram
```

Explanation:

- `CC = gcc` sets the compiler.
- `CFLAGS` holds compiler flags: `-Wall` enables warnings, `-g` includes debug info.
- `myprogram` depends on two object files.
- The `clean` rule deletes intermediate files so you can rebuild from scratch.

To compile your project, simply run:

```
make
```

To clean it:

```
make clean
```

15.3.4 Common gcc Compilation Flags

The `gcc` compiler accepts many flags that influence how your code is compiled. Here are some of the most useful:

`-Wall`

Enables most warning messages to help detect potential bugs.

```
gcc -Wall main.c -o main
```

`-Wextra`

Enables additional warning messages.

`-Werror`

Treats warnings as errors, forcing you to fix them.

`-g`

Includes debugging information for tools like `gdb`.

`-O` (Optimization)

Controls optimization level:

- `-O0` = no optimization (default)
- `-O1`, `-O2`, `-O3` = increasing levels of optimization
- `-Os` = optimize for size

```
gcc -O2 main.c -o main
```

`-c`

Compiles a `.c` file to a `.o` (object file) without linking.

```
gcc -c utils.c
```

`-o`

Specifies the output file.

```
gcc -o program main.o utils.o
```

15.3.5 Practical Makefile Enhancements

Using Variables

Define reusable variables for compiler and flags:

```
CC = gcc
CFLAGS = -Wall -g
```

Use them like:

```
$(CC) $(CFLAGS) -c file.c
```

Pattern Rules

Instead of repeating rules for each `.c` file, you can use pattern rules:

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

- `$<` is the first dependency (source file)
- `$@` is the target (object file)

15.3.6 Example Project Walkthrough

Files:

- `main.c`:

```c
#include <stdio.h>
#include "utils.h"

int main() {
    greet("World");
    return 0;
}
```

- `utils.c`:

```c
#include <stdio.h>
#include "utils.h"

void greet(const char *name) {
    printf("Hello, %s!\n", name);
}
```

- `utils.h`:

```c
#ifndef UTILS_H
#define UTILS_H

void greet(const char *name);

#endif
```

Makefile:

```
CC = gcc
CFLAGS = -Wall -g

all: app

app: main.o utils.o
    $(CC) $(CFLAGS) -o app main.o utils.o

%.o: %.c
```

```
    $(CC) $(CFLAGS) -c $<

clean:
    rm -f *.o app
```

To build:

```
make
```

To clean:

```
make clean
```

15.3.7 Conclusion

Makefiles are powerful tools that simplify and automate the compilation process in C projects. Combined with useful `gcc` flags, they help enforce good practices like enabling warnings, generating debug info, and optimizing code. Learning to write and use Makefiles effectively will save you time, reduce errors, and make your projects more manageable.

In the next chapter, we'll dive into debugging techniques and tools to help you catch and fix errors in your C code efficiently.

Chapter 16.

Data Structures in C

1. Linked Lists (Singly, Doubly)
2. Stacks and Queues
3. Trees (Binary Trees, Traversals)
4. Hash Tables (Basic Implementation)

16 Data Structures in C

16.1 Linked Lists (Singly, Doubly)

A **linked list** is a fundamental dynamic data structure in C used to store collections of elements called **nodes**. Unlike arrays, linked lists do not require a fixed size at compile time. Instead, nodes are dynamically allocated in memory, and each node contains data along with pointers that link it to other nodes in the sequence.

There are two main types of linked lists:

- **Singly Linked List**: Each node points only to the next node.
- **Doubly Linked List**: Each node points both to the next and the previous node.

16.1.1 Singly Linked Lists

Structure Definition

A singly linked list node is typically defined like this:

```
struct Node {
    int data;
    struct Node* next;
};
```

Each node contains an integer value (`data`) and a pointer to the next node (`next`).

Creating and Inserting Nodes

You can dynamically allocate a new node using `malloc` and link it to other nodes:

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

struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

To insert a node at the beginning:

```
void insertAtHead(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *head;
```

```
    *head = newNode;
}
```

To insert at the end:

```
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

Traversing the List

Traversal involves walking through each node:

```
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
```

Deleting a Node

To delete the first occurrence of a value:

```
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}
```

Full runnable code:

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

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Insert at the head
void insertAtHead(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
}

// Insert at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Print the list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Delete first occurrence of a value
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
```

```
        free(temp);
        return;
    }

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

// Demo usage
int main() {
    struct Node* head = NULL;

    // Insert values
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtHead(&head, 5);
    insertAtEnd(&head, 30);

    printf("List after insertions:\n");
    printList(head);

    // Delete a node
    deleteNode(&head, 20);

    printf("List after deleting 20:\n");
    printList(head);

    // Cleanup
    while (head != NULL) {
        deleteNode(&head, head->data);
    }

    return 0;
}
```

16.1.2 Doubly Linked Lists

A **doubly linked list** provides more flexibility, as each node contains pointers to both the next and previous nodes.

Structure Definition

```
struct DNode {
    int data;
    struct DNode* prev;
    struct DNode* next;
```

```
};
```

Inserting at the Beginning

```
void insertAtHead(struct DNode** head, int value) {
    struct DNode* newNode = (struct DNode*) malloc(sizeof(struct DNode));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = *head;

    if (*head != NULL)
        (*head)->prev = newNode;

    *head = newNode;
}
```

Inserting at the End

```
void insertAtEnd(struct DNode** head, int value) {
    struct DNode* newNode = (struct DNode*) malloc(sizeof(struct DNode));
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }

    struct DNode* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;
    newNode->prev = temp;
}
```

Traversing the Doubly Linked List

Forward traversal:

```
void printForward(struct DNode* head) {
    while (head != NULL) {
        printf("%d <-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
```

Backward traversal:

```
void printBackward(struct DNode* tail) {
    while (tail != NULL) {
        printf("%d <-> ", tail->data);
        tail = tail->prev;
    }
    printf("NULL\n");
```

```
}
```

Deleting a Node

```
void deleteNode(struct DNode** head, int key) {
    struct DNode* temp = *head;

    while (temp != NULL && temp->data != key)
        temp = temp->next;

    if (temp == NULL) return;

    if (temp->prev != NULL)
        temp->prev->next = temp->next;
    else
        *head = temp->next;

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    free(temp);
}
```

Full runnable code:

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

struct DNode {
    int data;
    struct DNode* prev;
    struct DNode* next;
};

// Function declarations
void printBackward(struct DNode* tail);

// Insert at head
void insertAtHead(struct DNode** head, int value) {
    struct DNode* newNode = (struct DNode*) malloc(sizeof(struct DNode));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = *head;

    if (*head != NULL)
        (*head)->prev = newNode;

    *head = newNode;
}

// Insert at end
void insertAtEnd(struct DNode** head, int value) {
    struct DNode* newNode = (struct DNode*) malloc(sizeof(struct DNode));
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
```

```
        newNode->prev = NULL;
        *head = newNode;
        return;
    }

    struct DNode* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;
    newNode->prev = temp;
}

// Print forward
void printForward(struct DNode* head) {
    printf("Forward: ");
    struct DNode* last = NULL;
    while (head != NULL) {
        printf("%d <-> ", head->data);
        last = head;
        head = head->next;
    }
    printf("NULL\n");

    printf("Backward: ");
    printBackward(last);
}

// Print backward
void printBackward(struct DNode* tail) {
    while (tail != NULL) {
        printf("%d <-> ", tail->data);
        tail = tail->prev;
    }
    printf("NULL\n");
}

// Delete node by value
void deleteNode(struct DNode** head, int key) {
    struct DNode* temp = *head;

    while (temp != NULL && temp->data != key)
        temp = temp->next;

    if (temp == NULL) return;

    if (temp->prev != NULL)
        temp->prev->next = temp->next;
    else
        *head = temp->next;

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    free(temp);
}

// Free entire list
```

```c
void freeList(struct DNode* head) {
    while (head != NULL) {
        struct DNode* next = head->next;
        free(head);
        head = next;
    }
}

int main() {
    struct DNode* head = NULL;

    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtHead(&head, 5);
    insertAtEnd(&head, 30);

    printForward(head);

    deleteNode(&head, 20);
    printf("\nAfter deleting 20:\n");
    printForward(head);

    freeList(head);

    return 0;
}
```

16.1.3 Comparing Linked Lists and Arrays

Feature	Arrays	Linked Lists
Size	Fixed	Dynamic
Memory	Contiguous	Non-contiguous
Insertion	Costly (O(n))	Efficient at ends (O(1))
Deletion	Costly (O(n))	Efficient at ends (O(1))
Access	Fast (O(1)) by index	Sequential (O(n))
Overhead	Minimal	Extra memory for pointers

Linked lists are useful when:

- You need a dynamic collection with frequent insertions/deletions.
- You want to avoid resizing issues with arrays.

However, arrays are preferred when:

- You need fast random access.
- Memory is tight and predictable.

16.1.4 Conclusion

Linked lists are an essential data structure that provide dynamic memory management and efficient insertion and deletion operations. Mastering both singly and doubly linked lists helps you understand how memory and pointers work in C. While arrays are simpler and faster for access, linked lists shine when flexibility and scalability are key.

In the next section, you'll learn about **stacks and queues**, two powerful abstract data types often implemented using linked lists or arrays.

16.2 Stacks and Queues

Stacks and **queues** are two essential abstract data types (ADTs) used in computer science. Though they differ in behavior, both are linear data structures that can be implemented using arrays or linked lists in C. Understanding them helps you build logic for applications like compilers, task schedulers, operating systems, and more.

16.2.1 Stack: Last-In, First-Out (LIFO)

A **stack** is a linear data structure that follows the **Last-In, First-Out** principle. The last element added (pushed) is the first one to be removed (popped). Imagine a stack of plates—new plates are added on top and removed from the top.

Stack Operations

- **Push**: Add an element to the top.
- **Pop**: Remove the top element.
- **Peek/Top**: View the top element without removing it.
- **isEmpty**: Check if the stack is empty.

16.2.2 Stack Implementation Using Arrays

Full runnable code:

```c
#include <stdio.h>
#define MAX 100

int stack[MAX];
int top = -1;

void push(int value) {
```

```
    if (top >= MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = value;
}

int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}

int peek() {
    if (top < 0) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack[top];
}

int main() {
    push(10);
    push(20);
    printf("Top: %d\n", peek());
    printf("Popped: %d\n", pop());
    printf("Top after pop: %d\n", peek());
    return 0;
}
```

16.2.3 Stack Implementation Using Linked List

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL;

void push(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}

int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
```

```
        return -1;
    }
    int val = top->data;
    struct Node* temp = top;
    top = top->next;
    free(temp);
    return val;
}
```

16.2.4 Applications of Stacks

- Expression evaluation (postfix/infix)
- Function call management (recursion)
- Undo functionality in editors
- Syntax parsing

16.2.5 Queue: First-In, First-Out (FIFO)

A **queue** follows the **First-In, First-Out** principle. Think of a line at a bank: the first person to arrive is the first to be served.

Queue Operations

- **Enqueue**: Add an element to the rear.
- **Dequeue**: Remove the element at the front.
- **Peek/Front**: View the front element.
- **isEmpty**: Check if the queue is empty.

16.2.6 Queue Implementation Using Arrays

Full runnable code:

```
#include <stdio.h>
#define SIZE 100

int queue[SIZE];
int front = -1, rear = -1;

void enqueue(int value) {
    if (rear == SIZE - 1) {
        printf("Queue Overflow\n");
        return;
```

```
    }
    if (front == -1) front = 0;
    queue[++rear] = value;
}

int dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return -1;
    }
    return queue[front++];
}

int main() {
    enqueue(5);
    enqueue(10);
    printf("Dequeued: %d\n", dequeue());
    enqueue(15);
    printf("Dequeued: %d\n", dequeue());
    return 0;
}
```

16.2.7 Queue Implementation Using Linked List

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

struct Node {
    int data;
    struct Node* next;
};

struct Node *front = NULL, *rear = NULL;

void enqueue(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        return -1;
    }
    int val = front->data;
    struct Node* temp = front;
    front = front->next;
```

```
    if (front == NULL) rear = NULL;
    free(temp);
    return val;
}
```

16.2.8 Applications of Queues

- Task scheduling (OS job queues)
- Breadth-first search (BFS) in graphs
- Print queues and IO buffers
- Data stream management

16.2.9 Comparison: Stack vs Queue

Feature	Stack	Queue
Order	LIFO	FIFO
Insertion	At the top	At the rear
Deletion	From the top	From the front
Common Use	Function calls	Scheduling, BFS

16.2.10 Conclusion

Stacks and queues are vital tools in your data structure toolkit. Stacks work well for backtracking and nested problems, while queues are ideal for managing ordered tasks and sequential processing. By learning to implement them using arrays and linked lists, you gain flexibility in how you manage data and build more efficient C programs.

In the next section, we'll dive into **trees**, another foundational data structure with powerful hierarchical applications.

16.3 Trees (Binary Trees, Traversals)

A **binary tree** is a hierarchical data structure in which each node has at most two children: a **left** child and a **right** child. This structure is widely used in programming for organizing, storing, and retrieving hierarchical or sorted data efficiently.

Binary trees are the foundation for many advanced data structures, such as binary search trees (BSTs), heaps, and syntax trees used in compilers. Understanding their structure and how to traverse them is crucial for building efficient algorithms.

16.3.1 Structure of a Binary Tree Node

In C, a binary tree node is commonly represented using a `struct` with three members: the data stored in the node, a pointer to the left child, and a pointer to the right child.

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

To create a new node:

```c
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

16.3.2 Binary Tree Traversals

Traversal refers to the process of visiting all the nodes in a tree in a particular order. The three most common types of depth-first traversal are:

1. **Inorder** (Left, Root, Right)
2. **Preorder** (Root, Left, Right)
3. **Postorder** (Left, Right, Root)

These can be implemented recursively or iteratively.

Inorder Traversal (Recursive)

In an inorder traversal, we visit the left subtree, then the root node, and finally the right subtree.

```c
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
```

```
}
```

Example Output: For a tree with root `10`, left child `5`, and right child `15`, the output is: `5 10 15`

This traversal is particularly useful in **binary search trees (BSTs)**, as it visits the nodes in ascending order.

Preorder Traversal (Recursive)

In preorder traversal, we visit the root node first, then the left subtree, and then the right subtree.

```
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

Use case: Preorder is useful for copying a tree or expressing tree structures (e.g., for printing).

Postorder Traversal (Recursive)

In postorder traversal, we visit the left and right subtrees before visiting the root node.

```
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
```

Use case: Postorder is often used to **delete or free memory** in a tree from the leaves up.

16.3.3 Iterative Traversals (Inorder Example)

Recursive methods are elegant but rely on the system's call stack. For large trees, it may be better to use iterative solutions with an explicit stack.

```
#include <stdbool.h>

#define MAX 100
struct Node* stack[MAX];
int top = -1;

void push(struct Node* node) {
    stack[++top] = node;
}
```

```
struct Node* pop() {
    return stack[top--];
}

bool isEmpty() {
    return top == -1;
}

void inorderIterative(struct Node* root) {
    struct Node* current = root;
    while (current != NULL || !isEmpty()) {
        while (current != NULL) {
            push(current);
            current = current->left;
        }
        current = pop();
        printf("%d ", current->data);
        current = current->right;
    }
}
```

16.3.4 Building a Simple Binary Tree

```
int main() {
    struct Node* root = createNode(10);
    root->left = createNode(5);
    root->right = createNode(15);
    root->left->left = createNode(2);
    root->left->right = createNode(7);

    printf("Inorder: ");
    inorder(root);
    printf("\nPreorder: ");
    preorder(root);
    printf("\nPostorder: ");
    postorder(root);
    return 0;
}
```

Output:

```
Inorder: 2 5 7 10 15
Preorder: 10 5 2 7 15
Postorder: 2 7 5 15 10
```

16.3.5 Practical Applications of Trees

- **Binary Search Trees (BSTs)**: Allow efficient searching, insertion, and deletion in O(log n) time (on average).
- **Expression Trees**: Used to evaluate mathematical expressions.
- **Hierarchical Databases**: Represent file systems or organizational charts.
- **Huffman Trees**: Used in data compression algorithms.

16.3.6 Benefits and Drawbacks

Advantages:

- Efficient for representing hierarchical relationships.
- BSTs offer fast lookups for sorted data.
- Dynamic structure that grows or shrinks as needed.

Disadvantages:

- Not cache-friendly (nodes are dynamically allocated).
- Can become unbalanced without additional logic (e.g., AVL or Red-Black trees).
- Harder to implement compared to arrays or linked lists.

16.3.7 Summary

Binary trees provide a flexible and powerful way to organize hierarchical data. By understanding node structure and traversal techniques (inorder, preorder, postorder), you can effectively use trees in a variety of programming scenarios—from searching and sorting to parsing expressions and managing complex data.

Next, we'll explore **hash tables**, a key-value data structure that enables fast access and storage based on computed indices.

16.4 Hash Tables (Basic Implementation)

A **hash table** is a powerful data structure that allows for fast storage and retrieval of key-value pairs. Hash tables are commonly used for implementing dictionaries, symbol tables, caches, and indexing systems. The main idea is to use a **hash function** to map keys to specific locations in a **table (array)**, allowing for **average-case constant-time** (`O(1)`) lookups, insertions, and deletions.

16.4.1 How a Hash Table Works

At the core of a hash table is a **hash function**, which takes a key (usually a string or integer) and converts it into an index within the bounds of an array. Each element of the array is called a **bucket**. If multiple keys hash to the same index (called a **collision**), the table needs a strategy to handle it. The most common strategy is **separate chaining**, where each bucket holds a **linked list** of entries.

16.4.2 Hash Function

A good hash function distributes keys uniformly across the buckets. For integers, the modulo operation is often used:

```
int hash(int key, int tableSize) {
    return key % tableSize;
}
```

For strings, a more complex function is needed to convert the character data into an integer hash code.

16.4.3 Defining the Hash Table

Let's build a simple hash table in C that stores integer keys using **separate chaining** to resolve collisions.

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

#define TABLE_SIZE 10

// Node structure for chaining
struct Node {
    int key;
    struct Node* next;
};

// Hash table: array of pointers to Node
struct Node* hashTable[TABLE_SIZE];
```

16.4.4 Hash Function

```
int hash(int key) {
    return key % TABLE_SIZE;
}
```

16.4.5 Insert Operation

To insert a key, compute the hash index and add the key to the front of the linked list at that bucket.

```
void insert(int key) {
    int index = hash(key);
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
```

16.4.6 Search Operation

To search for a key, compute the index and traverse the linked list at that bucket.

```
int search(int key) {
    int index = hash(key);
    struct Node* current = hashTable[index];
    while (current != NULL) {
        if (current->key == key) return 1;
        current = current->next;
    }
    return 0;
}
```

16.4.7 Delete Operation

To delete a key, traverse the linked list at the bucket and remove the matching node.

```
void delete(int key) {
    int index = hash(key);
    struct Node* current = hashTable[index];
    struct Node* prev = NULL;

    while (current != NULL) {
        if (current->key == key) {
            if (prev == NULL) {
                hashTable[index] = current->next;
            } else {
                prev->next = current->next;
            }
            free(current);
            printf("Key %d deleted.\n", key);
            return;
        }
        prev = current;
        current = current->next;
    }
    printf("Key %d not found.\n", key);
```

```
}
```

16.4.8 Display the Hash Table

```
void display() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("[%d]:", i);
        struct Node* current = hashTable[i];
        while (current != NULL) {
            printf(" %d ->", current->key);
            current = current->next;
        }
        printf(" NULL\n");
    }
}
```

16.4.9 Example Usage

```
int main() {
    insert(15);
    insert(25);
    insert(35);
    insert(5);
    display();

    printf("Search 25: %s\n", search(25) ? "Found" : "Not Found");
    delete(25);
    display();
    return 0;
}
```

Output:

```
[0]: NULL
[1]: 1 -> NULL
[2]: 2 -> NULL
[3]: 3 -> NULL
[4]: 4 -> NULL
[5]: 35 -> 25 -> 15 -> 5 -> NULL
...
Search 25: Found
Key 25 deleted.
...
```

16.4.10 Limitations and Considerations

1. **Fixed Size**: In our example, the table size is fixed. If too many elements are inserted, collisions become more frequent, reducing performance.
2. **Load Factor**: The **load factor** is the ratio of the number of elements to the number of buckets. A high load factor increases the number of collisions. Typical practice is to keep the load factor below `0.75`.
3. **Resizing**: To handle more data efficiently, hash tables should support **dynamic resizing**, which involves creating a larger table and **rehashing** existing keys into the new table.
4. **Choice of Hash Function**: A poor hash function can lead to clustering or uneven distribution, degrading performance.
5. **Memory Overhead**: Chaining adds memory overhead due to linked list nodes. An alternative is **open addressing**, which stores entries directly in the table and probes for empty slots.

16.4.11 Summary

Hash tables provide **fast and efficient** access to data using keys. In this section, you learned how to implement a basic hash table using separate chaining in C, including functions for insertion, search, and deletion. While this is a simplified version, it demonstrates the core mechanics that underpin more sophisticated hash table implementations.

By understanding hashing and collision resolution strategies, you are well-equipped to build and use hash tables in practical applications such as symbol tables, caches, and indexing systems.

Chapter 17.

Working with the C Standard Library

1. Common Header Files Overview
2. Math Functions, Time, and Character Handling
3. `stdlib.h`, `ctype.h`, `string.h`, `time.h`

17 Working with the C Standard Library

17.1 Common Header Files Overview

C provides a powerful set of built-in libraries that offer functions for input/output, memory management, string operations, mathematics, and more. These libraries are accessed through *header files*, which are included at the beginning of a program using the `#include` directive. Understanding the most commonly used headers helps you leverage the full potential of the C Standard Library and write more efficient, maintainable code.

Let's explore some of the most essential header files and see how they are used in practice.

17.1.1 `stdio.h` Standard Input and Output

The **Standard I/O library** is one of the most fundamental headers in C programming. It provides functions for reading from and writing to input/output streams.

Common functions:

- `printf()` – Prints formatted output to the screen.
- `scanf()` – Reads formatted input from the keyboard.
- `fgets()`, `fputs()` – Read/write strings to/from files or standard I/O.
- `fopen()`, `fclose()`, `fread()`, `fwrite()` – File operations.

Example:

```c
#include <stdio.h>

int main() {
    printf("Enter your name: ");
    char name[50];
    fgets(name, sizeof(name), stdin);
    printf("Hello, %s", name);
    return 0;
}
```

17.1.2 `stdlib.h` Standard Library Utilities

This header contains utility functions for memory allocation, process control, conversions, and more.

Common functions:

- `malloc()`, `calloc()`, `free()` – Dynamic memory management.
- `atoi()`, `atof()` – Convert strings to integers or floats.
- `exit()` – Terminates the program with a status code.

- `rand()`, `srand()` – Pseudo-random number generation.

Example:

```c
#include <stdlib.h>

int* createArray(int size) {
    return (int*) malloc(size * sizeof(int));
}
```

17.1.3 `string.h` String Handling

This header provides functions to manipulate C-style strings (null-terminated character arrays).

Common functions:

- `strlen()` – Gets the length of a string.
- `strcpy()`, `strncpy()` – Copy strings.
- `strcat()`, `strncat()` – Concatenate strings.
- `strcmp()`, `strncmp()` – Compare strings.
- `memcpy()`, `memset()` – Memory operations.

Example:

```c
#include <string.h>

char str1[20] = "Hello, ";
char str2[] = "World!";
strcat(str1, str2); // str1 becomes "Hello, World!"
```

17.1.4 `ctype.h` Character Classification

This header contains functions to test and convert characters.

Common functions:

- `isalpha()`, `isdigit()` – Check if a character is alphabetic or numeric.
- `toupper()`, `tolower()` – Convert character case.
- `isspace()`, `isalnum()` – Check for whitespace or alphanumeric characters.

Example:

```c
#include <ctype.h>

if (isalpha('A')) {
    printf("It's a letter.\n");
}
```

17.1.5 `math.h` Mathematical Functions

This header provides advanced math functions for floating-point calculations. You must link with the math library (`-lm`) when compiling.

Common functions:

- `sqrt()`, `pow()` – Square root and power.
- `sin()`, `cos()`, `tan()` – Trigonometric functions.
- `log()`, `exp()` – Logarithmic and exponential functions.
- `fabs()` – Absolute value for floating-point numbers.

Example:

Full runnable code:

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

int main() {
    double x = 2.0;
    printf("sqrt(2.0) = %lf\n", sqrt(x));
    return 0;
}
```

17.1.6 `time.h` Time and Date Utilities

This header allows access to time-related functions and structures.

Common functions:

- `time()`, `ctime()` – Get the current time.
- `difftime()` – Compute difference between two times.
- `clock()` – Processor time used by the program.
- `struct tm` – A structure to hold calendar time.

Example:

Full runnable code:

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

int main() {
    time_t now;
    time(&now);
    printf("Current time: %s", ctime(&now));
    return 0;
}
```

17.1.7 Summary

These standard header files form the foundation of most C programs:

Header	Purpose
`<stdio.h>`	Input/output (e.g., `printf`, `scanf`)
`<stdlib.h>`	Memory allocation, conversion, exit
`<string.h>`	String manipulation
`<ctype.h>`	Character checks and transformations
`<math.h>`	Mathematical functions
`<time.h>`	Time and date handling

Including and using these headers appropriately lets you write cleaner, more effective programs and reduces the need to reinvent basic functionality. In the next sections, we'll explore some of these libraries in more detail.

17.2 Math Functions, Time, and Character Handling

C provides a powerful set of standard library functions for performing mathematical calculations, handling time, and working with characters. These functions are spread across three important headers: `<math.h>`, `<time.h>`, and `<ctype.h>`. This section will introduce these libraries and show how to use their functions effectively in practical C programs.

17.2.1 Mathematical Functions `math.h`

The `<math.h>` header offers a rich collection of mathematical functions for handling common computations. These functions are essential for applications in science, engineering, graphics, and more. When compiling a program that uses `<math.h>`, you may need to link the math library by adding `-lm` to your `gcc` command.

Common Functions:

- `sqrt(double x)`: Returns the square root of `x`.
- `pow(double base, double exp)`: Returns `base` raised to the power of `exp`.
- `fabs(double x)`: Returns the absolute value of `x`.
- `sin(double x)`, `cos(double x)`, `tan(double x)`: Trigonometric functions.
- `log(double x)`: Natural logarithm.
- `exp(double x)`: Exponential function `e^x`.

Example:

Full runnable code:

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

int main() {
    double x = 4.0;
    printf("Square root of %.2f is %.2f\n", x, sqrt(x));
    printf("2 to the power of 3 is %.2f\n", pow(2, 3));
    printf("Absolute value of -7.5 is %.2f\n", fabs(-7.5));
    return 0;
}
```

These functions operate on `double` types. If you're working with `float` or `long double`, use the versions with suffixes like `sqrtf()` or `sqrtl()`.

17.2.2 Time and Date Handling `time.h`

The `<time.h>` header allows programs to track, measure, and manipulate time. It is useful for implementing timers, measuring performance, or displaying current dates.

Key Types and Functions:

- `time_t`: Represents calendar time.
- `struct tm`: Holds broken-down time (year, month, day, etc.).
- `time(time_t *t)`: Gets the current time.
- `localtime(const time_t *t)`: Converts `time_t` to `struct tm`.
- `ctime(const time_t *t)`: Converts `time_t` to a string.
- `difftime(time_t end, time_t start)`: Calculates the difference in seconds.
- `clock()`: Returns processor time used by the program.
- `sleep(seconds)` (POSIX): Pauses the program for a number of seconds (may require `#include <unistd.h>` on UNIX-like systems).

Example Displaying the Current Time:

Full runnable code:

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

int main() {
    time_t current_time;
    time(&current_time);

    printf("Current time: %s", ctime(&current_time));
    return 0;
}
```

Example Measuring Execution Time:

Full runnable code:

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

int main() {
    clock_t start = clock();

    // Simulate some work
    for (long i = 0; i < 100000000; i++);

    clock_t end = clock();
    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken: %.2f seconds\n", elapsed);
    return 0;
}
```

The `<time.h>` functions are essential for any application that needs to interact with the system clock, schedule tasks, or benchmark performance.

17.2.3 Character Handling `ctype.h`

The `<ctype.h>` header provides functions for testing and manipulating character values. These are especially helpful when writing parsers, validating user input, or processing text.

Common Functions:

- `isalpha(char c)`: Checks if `c` is a letter (A–Z or a–z).
- `isdigit(char c)`: Checks if `c` is a digit (0–9).
- `isalnum(char c)`: Checks if `c` is alphanumeric (letter or digit).
- `isspace(char c)`: Checks if `c` is whitespace (`' '`, `'\t'`, `'\n'`, etc.).
- `toupper(char c)`: Converts a lowercase letter to uppercase.
- `tolower(char c)`: Converts an uppercase letter to lowercase.

Each function takes an `int` as input, typically a `char` promoted to `int`.

Example:

Full runnable code:

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

int main() {
    char ch = 'a';

    if (isalpha(ch)) {
        printf("%c is a letter\n", ch);
```

```
    }

    if (isdigit('5')) {
        printf("'5' is a digit\n");
    }

    printf("Uppercase of %c is %c\n", ch, toupper(ch));
    return 0;
}
```

Example Counting Digits and Letters in a String:

Full runnable code:

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

int main() {
    char text[] = "Hello123!";
    int letters = 0, digits = 0;

    for (int i = 0; text[i] != '\0'; i++) {
        if (isalpha(text[i])) letters++;
        else if (isdigit(text[i])) digits++;
    }

    printf("Letters: %d, Digits: %d\n", letters, digits);
    return 0;
}
```

These character-handling functions make it easy to validate input or format text in ways that would otherwise be tedious and error-prone.

17.2.4 Conclusion

The C Standard Library provides a wide array of utility functions through headers like `<math.h>`, `<time.h>`, and `<ctype.h>`. Whether you're calculating math operations, working with timestamps, or processing characters, these functions simplify many common programming tasks. By learning how to use these headers effectively, you can write more powerful and reliable C programs.

In the next section, we'll take a deeper dive into libraries like `<stdlib.h>`, `<string.h>`, and revisit some functions from `<time.h>` and `<ctype.h>` for more advanced usage.

17.3 `stdlib.h`, `ctype.h`, `string.h`, `time.h`

In the C Standard Library, several headers provide essential functions that help you perform common programming tasks efficiently and safely. In this section, we delve deeper into four important headers: `<stdlib.h>`, `<ctype.h>`, `<string.h>`, and `<time.h>`. We'll explore their most commonly used functions and show how they can be combined in practical, real-world C programs.

17.3.1 Memory Management with `stdlib.h`

The `<stdlib.h>` header contains general utilities, but one of its most crucial roles is dynamic memory management.

- **`malloc(size_t size)`**: Allocates a block of memory of `size` bytes and returns a pointer to it.
- **`free(void *ptr)`**: Frees previously allocated memory pointed to by `ptr`.
- **`calloc(size_t nitems, size_t size)`**: Allocates memory for an array of `nitems`, each of size `size`, and initializes all bytes to zero.
- **`realloc(void *ptr, size_t size)`**: Resizes previously allocated memory block to a new size.

Example: Allocating and freeing a dynamic array of integers

Full runnable code:

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

int main() {
    int n = 5;
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

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

    free(arr); // Always free allocated memory
    return 0;
}
```

17.3.2 Character Classification with `ctype.h`

We briefly covered `<ctype.h>` in the previous section, but here we'll look at how to use its functions for parsing and validating input.

Common functions include:

- `isdigit(int c)`: Checks if the character is a digit (`'0'`–`'9'`).
- `isalpha(int c)`: Checks if the character is a letter.
- `isspace(int c)`: Checks if the character is whitespace.
- `toupper(int c)` / `tolower(int c)`: Convert characters to uppercase/lowercase.

Example: Parsing and validating numeric input

Full runnable code:

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

int main() {
    char input[100];
    printf("Enter a positive integer: ");
    fgets(input, sizeof(input), stdin);

    // Validate input contains only digits
    int valid = 1;
    for (int i = 0; input[i] != '\0' && input[i] != '\n'; i++) {
        if (!isdigit(input[i])) {
            valid = 0;
            break;
        }
    }

    if (valid) {
        printf("You entered a valid positive integer: %s", input);
    } else {
        printf("Invalid input: contains non-digit characters.\n");
    }
    return 0;
}
```

17.3.3 String Manipulation with `string.h`

String handling is essential in most C programs. The `<string.h>` header offers many functions for manipulating C-style null-terminated strings.

Some of the most frequently used functions include:

- `strcpy(char *dest, const char *src)`: Copies the string `src` into `dest`.
- `strcat(char *dest, const char *src)`: Concatenates `src` onto the end of `dest`.
- `strlen(const char *str)`: Returns the length of the string (excluding the null ter-

minator).

- `strcmp(const char *s1, const char *s2)`: Compares two strings lexicographically.
- `strncpy(char *dest, const char *src, size_t n)`: Copies at most `n` characters, safer alternative to `strcpy`.

Example: Combining strings safely and printing length

Full runnable code:

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

int main() {
    char greeting[50] = "Hello, ";
    char name[] = "World";

    // Concatenate name to greeting
    strcat(greeting, name);

    printf("%s\n", greeting);
    printf("Length of greeting: %lu\n", strlen(greeting));

    return 0;
}
```

In real programs, be careful about buffer sizes when using `strcpy` or `strcat` to avoid buffer overflows. Prefer `strncpy` or `strncat` with explicit size limits where possible.

17.3.4 Time Management with `time.h`

Beyond getting the current time with `time()` and formatting it with `ctime()`, `<time.h>` offers functions to measure elapsed time, convert between time formats, and manipulate time values.

Important functions:

- `time_t time(time_t *t)`: Gets the current calendar time.
- `struct tm *localtime(const time_t *t)`: Converts `time_t` to a broken-down local time.
- `double difftime(time_t end, time_t start)`: Computes difference in seconds between two times.
- `clock_t clock(void)`: Returns CPU time used by the program.

Example: Measuring runtime duration

Full runnable code:

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

```c
int main() {
    clock_t start = clock();

    // Simulate workload
    for (long i = 0; i < 100000000; i++);

    clock_t end = clock();
    double duration = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Program runtime: %.3f seconds\n", duration);
    return 0;
}
```

This is especially useful for benchmarking or profiling parts of your code.

17.3.5 Combining These Functions: A Practical Example

Let's write a small program that reads a line of input, validates it as a positive integer, stores it dynamically, and measures the time taken to process it.

Full runnable code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>

int main() {
    char input[100];
    printf("Enter a positive integer: ");
    fgets(input, sizeof(input), stdin);

    // Remove trailing newline if present
    input[strcspn(input, "\n")] = '\0';

    // Validate input
    for (int i = 0; input[i] != '\0'; i++) {
        if (!isdigit(input[i])) {
            printf("Invalid input: only digits are allowed.\n");
            return 1;
        }
    }

    // Convert to integer
    int num = atoi(input);

    // Allocate memory dynamically
    int *arr = (int *)malloc(num * sizeof(int));
    if (!arr) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
```

```
    }

    clock_t start = clock();

    // Initialize array
    for (int i = 0; i < num; i++) {
        arr[i] = i + 1;
    }

    clock_t end = clock();

    printf("Array initialized with numbers 1 to %d\n", num);
    printf("Time taken: %.6f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    free(arr);
    return 0;
}
```

This program demonstrates:

- Using `<ctype.h>` to validate input.
- Using `<stdlib.h>` for conversion and dynamic memory allocation.
- Using `<time.h>` to measure elapsed time.
- Using `<string.h>` to manipulate strings (removing newline).

17.3.6 Summary

- `<stdlib.h>` provides memory management and utility functions.
- `<ctype.h>` allows easy character validation and conversion.
- `<string.h>` supports safe and efficient string handling.
- `<time.h>` offers versatile time and date functions.

By combining these libraries, you can write robust C programs that handle input parsing, memory, strings, and performance measurement elegantly and safely.

Chapter 18.

Error Handling and Debugging

1. Return Codes and `errno`
2. Using `assert` and Debugging with GDB
3. Defensive Programming Practices

18 Error Handling and Debugging

18.1 Return Codes and `errno`

In C programming, handling errors effectively is crucial to creating robust and reliable applications. Since C does not have exceptions like some modern languages, it relies on return codes and global error indicators to signal when something has gone wrong. In this section, we will explore how return codes and the global variable `errno` work, and how to use them properly for error detection and handling.

18.1.1 Return Codes: Signaling Success or Failure

Most C functions that perform operations with the potential to fail return an integer or pointer indicating the success or failure of the operation. It is a convention that:

- A **return value of zero or a non-null pointer** usually indicates success.
- A **non-zero value or `NULL` pointer** indicates an error.

For example:

- The `fopen()` function returns a `FILE*`. If it fails to open a file, it returns `NULL`.
- The `malloc()` function returns a pointer to allocated memory, or `NULL` if allocation fails.
- System functions often return `-1` to indicate failure and set `errno`.

It is important to **always check these return values** to avoid undefined behavior or crashes later in your program.

18.1.2 Introducing `errno`: The Global Error Indicator

`errno` is a global integer variable declared in `<errno.h>` that is set by many system and library functions when an error occurs. It holds an error code representing the specific reason for failure.

Commonly used error codes include:

- `EINVAL`: Invalid argument
- `ENOMEM`: Out of memory
- `EACCES`: Permission denied
- `ENOENT`: No such file or directory

You should never assign to `errno` yourself; instead, check its value **only after a function indicates an error**.

18.1.3 Using errno in Practice

When a function signals failure (e.g., returns `NULL` or `-1`), you can examine `errno` to understand what went wrong.

To get a **human-readable description** of the error, use the function:

```
#include <string.h>
char *strerror(int errnum);
```

which returns a string describing the error code.

18.1.4 Example 1: Handling File I/O Errors

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

int main() {
    FILE *file = fopen("nonexistent.txt", "r");
    if (file == NULL) {
        fprintf(stderr, "Error opening file: %s\n", strerror(errno));
        return 1;
    }

    // Use file...

    fclose(file);
    return 0;
}
```

In this example:

- We attempt to open a file that does not exist.
- `fopen()` returns `NULL` on failure.
- We print a clear error message by using `strerror(errno)`.

18.1.5 Example 2: Checking Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main() {
    int *arr = malloc(1000000000 * sizeof(int)); // Attempt to allocate a large block

    if (arr == NULL) {
        fprintf(stderr, "Memory allocation failed: %s\n", strerror(errno));
```

```
        return 1;
    }

    // Use arr...

    free(arr);
    return 0;
}
```

If memory allocation fails, `malloc()` returns `NULL`, and we report the reason via `errno`.

18.1.6 Example 3: System Call Failure and `errno`

System calls such as `open()`, `read()`, and `write()` also set `errno` on failure. Here's an example using `open()` (requires `<fcntl.h>`):

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>

int main() {
    int fd = open("file.txt", O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "Failed to open file: %s\n", strerror(errno));
        return 1;
    }

    // Read or write using fd...

    close(fd);
    return 0;
}
```

18.1.7 Best Practices for Using Return Codes and `errno`

1. **Check return values immediately** after function calls that may fail.
2. **Use `errno` only after failure is indicated**, not unconditionally.
3. **Reset `errno` to 0** before calling functions if you need to detect new errors precisely.
4. **Print clear, user-friendly error messages**, especially in command-line programs.
5. **Avoid ignoring errors** — unhandled errors often cause bugs or security vulnerabilities.
6. **Document error handling behavior** in your own functions, so callers know how to respond.

18.1.8 A More Complete Example: Reading From a File with Error Handling

Full runnable code:

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main() {
    FILE *file = fopen("data.txt", "r");
    if (!file) {
        fprintf(stderr, "Cannot open file: %s\n", strerror(errno));
        return 1;
    }

    char buffer[256];
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }

    if (ferror(file)) {
        fprintf(stderr, "Error reading file: %s\n", strerror(errno));
        fclose(file);
        return 1;
    }

    fclose(file);
    return 0;
}
```

This example reads a file line-by-line and checks for reading errors with `ferror()`. If an error occurs during reading, we use `errno` to explain the problem.

18.1.9 Summary

- Return codes signal success or failure of functions and should always be checked.
- `errno` provides detailed error information after failure.
- Use `strerror(errno)` to print descriptive error messages.
- Handling errors gracefully improves program reliability and user experience.

By mastering return codes and `errno`, you can write C programs that not only work correctly but also respond intelligently to unexpected situations.

18.2 Using `assert` and Debugging with GDB

When developing C programs, detecting and fixing bugs is an essential part of the process. Two powerful tools to help catch errors and debug your code are the `assert()` macro and

the GNU Debugger (GDB). This section will introduce these tools and guide you through their practical use.

18.2.1 Using `assert()` to Catch Programming Errors Early

`assert()` is a macro defined in `<assert.h>` that helps you verify assumptions your program makes during execution. When the condition passed to `assert()` evaluates to false (zero), the program prints an error message and aborts execution.

This makes `assert()` useful for catching **logical errors and invalid states** during development, such as:

- Checking that pointers are not `NULL` before dereferencing.
- Verifying that function parameters are within expected ranges.
- Ensuring invariants in your code hold true.

Syntax

```c
#include <assert.h>

void someFunction(int x) {
    assert(x > 0);  // Program aborts if x <= 0
    // function body...
}
```

If the condition fails, you'll see output like:

```
a.out: example.c:5: someFunction: Assertion `x > 0` failed.
Aborted (core dumped)
```

Important Notes About `assert()`

- Assertions are **only active in debug builds** by default.
- If you compile your program with the `NDEBUG` macro defined (usually via `-DNDEBUG` flag), all `assert()` calls are disabled and removed by the preprocessor. This means `assert()` should **never have side effects** because they might not execute in production code.
- Use `assert()` to check conditions that should never happen if your code is correct, not to handle runtime errors like invalid user input.

18.2.2 Debugging with GDB: The GNU Debugger

While `assert()` helps detect incorrect assumptions, **GDB** allows you to interactively explore your program at runtime, step through lines of code, inspect variables, and find where things go wrong.

Installing GDB

GDB is widely available on Linux, macOS, and Windows (via MinGW or WSL). On Linux, you can install it using your package manager, e.g.:

```
sudo apt install gdb
```

Compiling with Debug Symbols

To use GDB effectively, you need to compile your program with debugging information enabled. This is done by adding the `-g` flag to your `gcc` command:

```
gcc -g -o myprogram myprogram.c
```

The `-g` flag tells the compiler to include debugging symbols, allowing GDB to map machine instructions back to your source code lines and variable names.

Starting GDB

Run GDB by invoking:

```
gdb ./myprogram
```

You'll enter the GDB command prompt `(gdb)` where you can type commands.

Basic GDB Commands

Here is a quick walkthrough of some essential GDB commands:

- **`run` (or `r`)**: Starts executing your program.
- **`break <line|function>` (or `b`)**: Sets a breakpoint to pause execution. Example: `break main` or `break example.c:25`
- **`next` (or `n`)**: Executes the next line of code but steps **over** function calls.
- **`step` (or `s`)**: Executes the next line and **steps into** function calls.
- **`continue` (or `c`)**: Resumes running until the next breakpoint or program end.
- **`print <variable>` (or `p`)**: Displays the value of a variable. Example: `print x`
- **`list` (or `l`)**: Shows source code around the current line.
- **`backtrace` (or `bt`)**: Shows the call stack at the current point, useful for tracing function calls leading to an error.
- **`quit` (or `q`)**: Exits GDB.

Step-by-Step Example

Imagine you have the following program in `example.c`:

Full runnable code:

```c
#include <stdio.h>

void divide(int a, int b) {
    printf("Result: %d\n", a / b);
}

int main() {
    int x = 10;
    int y = 0;
    divide(x, y);
    return 0;
}
```

This program will crash because of division by zero. Let's debug it.

1. **Compile with debug info:**

```
gcc -g -o example example.c
```

2. **Start GDB:**

```
gdb ./example
```

3. **Set a breakpoint at `divide`:**

```
(gdb) break divide
Breakpoint 1 at 0x400526: file example.c, line 5.
```

4. **Run the program:**

```
(gdb) run
Starting program: ./example

Breakpoint 1, divide (a=10, b=0) at example.c:5
5       printf("Result: %d\n", a / b);
```

5. **Inspect variables:**

```
(gdb) print a
$1 = 10
(gdb) print b
$2 = 0
```

6. **Step through code (optional):**

```
(gdb) step
```

7. **Fix the bug:** You realize `b` is zero, causing a crash.

8. **Quit GDB:**

```
(gdb) quit
```

18.2.3 Advanced Tips

- Use **conditional breakpoints** to stop only when a certain condition is true:

```
break divide if b == 0
```

- Inspect memory with commands like `x` (examine):

```
x/4x &x  # View 4 hex values starting at address of x
```

- Use `watch <var>` to stop execution when a variable changes.

18.2.4 Summary

- Use **`assert()`** during development to catch incorrect assumptions and logic errors early.
- Compile with `-g` and use **GDB** to step through your code, inspect variables, and identify bugs interactively.
- Mastering these tools will make debugging more systematic and less frustrating.

Mastering `assert()` and GDB equips you with powerful ways to find and fix bugs in your C programs, making your code more reliable and easier to maintain.

18.3 Defensive Programming Practices

Writing robust and maintainable C programs is essential to producing reliable software that behaves correctly even under unexpected conditions. Defensive programming is a methodology that encourages anticipating potential problems and coding proactively to prevent bugs, crashes, and security vulnerabilities. This section discusses key defensive programming strategies, including input validation, boundary checks, resource management, and clear error reporting, with practical examples to illustrate these concepts.

18.3.1 Validate Input Early and Often

One of the most common sources of bugs and security issues in C programs is **invalid or malicious input**. Since C offers minimal built-in protection against invalid memory access or out-of-bounds operations, validating inputs before processing them is crucial.

Example: Validating Command Line Arguments

Full runnable code:

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <positive-integer>\n", argv[0]);
        return 1;
    }

    char *endptr;
    long val = strtol(argv[1], &endptr, 10);

    if (*endptr != '\0' || val <= 0) {
        fprintf(stderr, "Invalid input: please provide a positive integer.\n");
        return 1;
    }

    printf("Input is a valid positive integer: %ld\n", val);
    return 0;
}
```

Here, `strtol` parses the input, and checking `endptr` ensures the entire string is numeric. This prevents processing unexpected or malformed input.

18.3.2 Always Perform Boundary Checks

Buffer overflows and out-of-bounds memory access are notorious causes of crashes and security vulnerabilities. Defensive code ensures arrays, buffers, and pointers are accessed only within their valid limits.

Example: Safe String Copying

Full runnable code:

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

void safe_copy(char *dest, size_t dest_size, const char *src) {
    if (strlen(src) >= dest_size) {
        fprintf(stderr, "Error: source string too long to copy safely.\n");
        return;
    }
    strcpy(dest, src);
}

int main() {
    char buffer[10];
    safe_copy(buffer, sizeof(buffer), "Hello");  // Safe
```

```
    safe_copy(buffer, sizeof(buffer), "This string is definitely too long");  // Prevented
    return 0;
}
```

Using explicit checks like `strlen(src) < dest_size` prevents writing beyond the allocated buffer, avoiding undefined behavior and potential security flaws.

18.3.3 Manage Resources Carefully

C programmers must manually manage dynamic memory, file handles, and other resources. Failure to do so causes resource leaks, crashes, or corrupted data.

Best practices include:

- Always pairing every allocation (`malloc`, `fopen`, etc.) with a corresponding release (`free`, `fclose`).
- Checking return values before using allocated resources.
- Setting pointers to `NULL` after freeing to avoid dangling references.
- Using `goto` cleanup patterns for error handling in functions with multiple resource allocations.

Example: Safe File Handling

```
#include <stdio.h>

int process_file(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Failed to open file");
        return -1;
    }

    // Read and process the file...

    if (fclose(file) != 0) {
        perror("Failed to close file");
        return -1;
    }
    return 0;
}
```

Here, `fopen` return is checked, and `fclose` errors are handled. This prevents using invalid file pointers or leaking open files.

18.3.4 Report Errors Clearly and Consistently

Clear, informative error messages improve maintainability and user experience. Instead of silently failing or crashing, programs should communicate what went wrong and possibly how to fix it.

Use `stderr` for error messages and return non-zero codes to signal failure. Avoid generic messages; include relevant context like variable values or function names.

Example: Meaningful Error Reporting

```c
#include <stdio.h>

void divide(int a, int b) {
    if (b == 0) {
        fprintf(stderr, "Error in divide(): division by zero is undefined (a=%d, b=%d)\n", a, b);
        return;
    }
    printf("Result: %d\n", a / b);
}
```

This explicit error message helps diagnose the problem faster than a program crash or silent incorrect output.

18.3.5 Use Assertions to Catch Programmer Errors

While defensive programming handles anticipated runtime issues, **`assert()`** helps detect logic errors during development (covered earlier in this chapter). Combine `assert()` with other defensive techniques to create layers of protection.

18.3.6 Minimize Global State and Use Encapsulation

Global variables increase complexity and risk of unexpected side effects. Encapsulating data within functions or structures helps localize errors and simplifies reasoning about code.

18.3.7 Avoid Undefined Behavior

Undefined behavior (UB) in C—such as dereferencing null pointers or integer overflow—can cause unpredictable results. Defensive programmers:

- Initialize variables before use.
- Avoid dangerous casts.

- Check pointers before dereferencing.
- Use tools like Valgrind or sanitizers to detect UB.

18.3.8 Summary Example: Combining Defensive Practices

Full runnable code:

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

int read_line(char *buffer, size_t size) {
    if (!buffer || size == 0) return -1;

    if (fgets(buffer, size, stdin) == NULL) {
        fprintf(stderr, "Error reading input.\n");
        return -1;
    }

    size_t len = strlen(buffer);
    if (len > 0 && buffer[len - 1] == '\n') {
        buffer[len - 1] = '\0';  // Remove trailing newline
    }
    return 0;
}

int main() {
    char input[100];

    if (read_line(input, sizeof(input)) != 0) {
        fprintf(stderr, "Failed to get user input.\n");
        return 1;
    }

    printf("You entered: %s\n", input);
    return 0;
}
```

This program:

- Checks pointers and sizes.
- Handles input errors gracefully.
- Avoids buffer overflow by using a size-limited input function.

18.3.9 Conclusion

Defensive programming in C is about anticipating problems before they happen. By validating input, performing boundary checks, managing resources carefully, and reporting errors clearly, you build more reliable and secure programs. Adopting these strategies reduces bugs and makes your code easier to maintain and extend.

If you practice these habits consistently, your C programs will be stronger and less prone to common pitfalls, preparing you for writing professional-quality code.

Chapter 19.

Memory Management Deep Dive

1. Stack vs Heap Memory
2. Common Pitfalls and Buffer Overflows
3. Writing Memory-Safe Code

19 Memory Management Deep Dive

19.1 Stack vs Heap Memory

Understanding how memory is managed in a C program is crucial to writing efficient, correct, and safe code. Two primary types of memory used during program execution are **stack memory** and **heap memory**. Each has distinct characteristics, uses, and constraints. This section explains the fundamental differences between stack and heap memory, how they are used in C programs, and practical considerations like lifetime, size limitations, and performance. We also cover common issues like stack overflow and heap fragmentation with examples.

19.1.1 What is Stack Memory?

The **stack** is a special region of memory that stores **automatic variables**, function call information (such as return addresses), and control data. It operates in a Last-In, First-Out (LIFO) manner.

When a function is called, a **stack frame** (or activation record) is created on the stack. This frame holds:

- The function's local variables.
- Function parameters.
- The return address (where the program should continue after the function finishes).
- Saved registers.

Once the function completes, its stack frame is popped off the stack, and all local variables stored in that frame are discarded.

19.1.2 Characteristics of Stack Memory

- **Automatic storage:** Variables declared inside functions are automatically allocated on the stack.
- **Fast allocation and deallocation:** Allocating stack memory is as simple as moving the stack pointer. This speed makes stack allocation very efficient compared to heap allocation.
- **Size limitation:** Stack size is limited and much smaller than the heap. Typical stack sizes range from hundreds of kilobytes to a few megabytes, depending on the system and compiler settings.
- **Lifetime tied to scope:** Variables exist only as long as the function is executing.

Once the function returns, the variables become invalid.

- **Contiguous allocation:** The stack grows and shrinks in one contiguous block, which helps with memory locality and performance.

19.1.3 Example of Stack Memory Usage

Full runnable code:

```c
#include <stdio.h>

void printSquares(int n) {
    int squares[5];  // local array allocated on the stack

    for (int i = 0; i < 5; i++) {
        squares[i] = i * i;
        printf("Square of %d is %d\n", i, squares[i]);
    }
}

int main() {
    printSquares(5);
    return 0;
}
```

In this example, the array `squares` is stored on the stack inside `printSquares`. When `printSquares` returns, `squares` ceases to exist.

19.1.4 What is Heap Memory?

The **heap** is a large pool of memory used for **dynamic allocation**, where blocks of memory can be requested and released in any order at runtime. Unlike stack memory, heap allocation persists until explicitly freed.

Functions like `malloc()`, `calloc()`, `realloc()`, and `free()` allow programs to request and manage heap memory dynamically.

19.1.5 Characteristics of Heap Memory

- **Manual management:** The programmer is responsible for allocating and freeing heap memory. Failure to free allocated memory leads to memory leaks.
- **Larger size:** The heap is typically much larger than the stack, often limited only by

system RAM and OS constraints.

- **Slower allocation:** Allocating memory on the heap is more expensive than on the stack because it involves searching for free blocks, fragmentation management, and bookkeeping.
- **Non-contiguous allocation:** Heap memory can be scattered throughout RAM, which may affect cache performance.
- **Variable lifetime:** Heap allocations can outlive the function that created them, allowing data sharing between functions or dynamic data structures like linked lists and trees.

19.1.6 Example of Heap Memory Usage

Full runnable code:

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

int main() {
    int *array = malloc(5 * sizeof(int));  // allocate array on heap
    if (array == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        array[i] = i * i;
        printf("Square of %d is %d\n", i, array[i]);
    }

    free(array);  // important to free heap memory when done
    return 0;
}
```

Here, `malloc` dynamically allocates memory on the heap. The array exists until `free` is called, independent of any function's scope.

19.1.7 Stack Overflow

A **stack overflow** occurs when the program uses more stack space than is allocated by the operating system or runtime environment. This can happen due to:

- Deep or infinite recursion.
- Allocating large local arrays or variables.

Example causing stack overflow:

```
void recursive(int n) {
    int largeArray[100000];  // very large local array
    if (n > 0) {
        recursive(n - 1);
    }
}
```

Each recursive call pushes a large array onto the stack, quickly exceeding stack limits and causing a crash.

Avoiding stack overflow:

- Use heap allocation for large data.
- Limit recursion depth.
- Increase stack size if necessary (platform dependent).

19.1.8 Heap Fragmentation

Heap fragmentation refers to the situation where free memory is split into small, non-contiguous blocks scattered across the heap. It happens after many allocations and deallocations of varying sizes.

Fragmentation can lead to:

- Inability to allocate large contiguous blocks, even if enough total free memory exists.
- Slower allocations due to the search for suitable blocks.

Mitigation strategies:

- Use memory pools for fixed-size allocations.
- Reuse memory blocks carefully.
- Minimize frequent allocations and deallocations.

19.1.9 Comparing Stack and Heap Memory

Feature	Stack	Heap
Allocation/Deallocation	Automatic, fast	Manual, slower
Size	Limited (small)	Larger, limited by system
Lifetime	Scope-limited	Until explicitly freed
Memory Layout	Contiguous, LIFO	Fragmented, flexible
Use Cases	Local variables, function calls	Dynamic data structures, large buffers

Feature	Stack	Heap
Safety	Safer due to automatic cleanup	Risk of leaks and dangling pointers
Performance	Generally faster due to cache locality	Can be slower due to management overhead

19.1.10 Summary

- **Stack memory** is ideal for local variables and function call management due to its speed and automatic cleanup, but limited in size and lifetime.
- **Heap memory** supports flexible, dynamic allocation required for variable-sized data and complex data structures but requires careful management to avoid leaks and fragmentation.
- Awareness of these differences helps in choosing appropriate storage strategies and avoiding common pitfalls like stack overflow and memory leaks.

Mastering the use of stack and heap memory is foundational for effective C programming and building efficient, reliable applications.

19.2 Common Pitfalls and Buffer Overflows

When programming in C, managing memory correctly is both powerful and challenging. Because C provides direct access to memory and requires manual management, a variety of common bugs can easily creep into your code if you're not careful. This section covers some of the most typical memory-related pitfalls, including **buffer overflows**, **dangling pointers**, **double frees**, and **memory leaks**. Understanding these errors, their causes, and how to prevent them is crucial to writing robust, secure C programs.

19.2.1 Buffer Overflows

A **buffer overflow** happens when a program writes more data to a buffer (usually an array or string) than it can hold, causing data to overwrite adjacent memory. This is one of the most common and dangerous programming errors in C, as it can corrupt memory, crash programs, or open security vulnerabilities.

How Buffer Overflows Occur

Buffer overflows typically happen due to:

- Lack of boundary checks on arrays.

- Using unsafe string functions like `strcpy()` or `gets()` that don't verify the destination buffer size.
- Incorrect loop bounds.

Example: Buffer overflow using `strcpy()`

Full runnable code:

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

int main() {
    char buffer[10];
    // Copy a string longer than buffer size - overflow occurs
    strcpy(buffer, "This string is way too long for buffer!");
    printf("Buffer content: %s\n", buffer);
    return 0;
}
```

In this example, `buffer` can hold only 10 characters, but `strcpy()` blindly copies the entire long string, overflowing the buffer and overwriting adjacent memory. This can cause undefined behavior including program crashes or exploitable vulnerabilities.

Security Implications

Buffer overflows are a primary cause of many security vulnerabilities such as **stack smashing attacks**, where malicious users inject code or alter control flow to execute arbitrary commands. This has led to a large number of security exploits historically.

19.2.2 Preventing Buffer Overflows

- **Use safer functions:** Replace `strcpy()` with `strncpy()`, which limits the number of characters copied.
- **Always check bounds:** When copying or reading data, ensure the destination buffer has enough space.
- **Use modern, safer functions:** Many systems provide safer alternatives like `strlcpy()` or `snprintf()`.
- **Validate input length:** Always validate user input or data length before processing.

Safe example using `strncpy()`

Full runnable code:

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

int main() {
    char buffer[10];
    strncpy(buffer, "Hello, world!", sizeof(buffer) - 1);
```

```
    buffer[sizeof(buffer) - 1] = '\0';  // Ensure null-termination
    printf("Buffer content: %s\n", buffer);
    return 0;
}
```

Here, `strncpy()` copies only up to 9 characters, and the last position is explicitly set to the null terminator, preventing overflow.

19.2.3 Dangling Pointers

A **dangling pointer** arises when a pointer points to memory that has already been freed or is otherwise invalid. Using such pointers leads to undefined behavior and hard-to-debug errors.

How Dangling Pointers Occur

- Freeing memory but continuing to use the pointer.
- Returning addresses of local variables (which live on the stack and vanish after function returns).

Example of dangling pointer

Full runnable code:

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

int *createNumber() {
    int num = 42;       // Local variable on stack
    return &num;        // Returning address of local variable - bad!
}

int main() {
    int *p = createNumber();
    printf("Value: %d\n", *p);  // Undefined behavior
    return 0;
}
```

Here, `p` points to a local variable `num` that no longer exists after `createNumber()` returns. Accessing `*p` is undefined behavior.

19.2.4 Avoiding Dangling Pointers

- Never return pointers to local variables.
- After `free()`, set the pointer to `NULL` to avoid accidental use.
- Be careful about pointer aliasing and ownership.

19.2.5 Double Free Errors

A **double free** error occurs when `free()` is called more than once on the same pointer, causing undefined behavior. This can lead to program crashes or corruption of the heap.

Example of double free

```c
#include <stdlib.h>

int main() {
    int *p = malloc(sizeof(int));
    free(p);
    free(p);  // Double free - undefined behavior
    return 0;
}
```

19.2.6 Preventing Double Free

- After freeing memory, immediately set the pointer to `NULL`.
- Check pointers before freeing.

19.2.7 Memory Leaks

A **memory leak** happens when allocated memory is never freed, causing the program's memory usage to grow unnecessarily, eventually exhausting system resources.

Example of memory leak

```c
#include <stdlib.h>

int main() {
    int *p = malloc(sizeof(int) * 100);
    // Forgot to call free(p);
    return 0;
}
```

This program allocates memory but never frees it, leading to a leak.

19.2.8 Detecting and Preventing Memory Leaks

- Always pair each `malloc()` or `calloc()` with a corresponding `free()`.
- Use tools like **Valgrind** to detect leaks.
- Design code with clear ownership semantics.
- Avoid unnecessary allocations.

- Use smart resource management patterns where possible.

19.2.9 Summary of Best Practices

Problem	Cause	Prevention
Buffer overflow	Unchecked array/string writes	Use bounded functions, validate input
Dangling pointer	Using freed or invalid pointers	Nullify pointers after free, careful ownership
Double free	Freeing the same pointer twice	Set pointer to NULL after free
Memory leak	Forgetting to free memory	Pair allocation and free, use tools

19.2.10 Conclusion

Memory-related bugs in C can cause crashes, data corruption, and security vulnerabilities. By understanding how these issues arise and adopting safe coding practices—such as boundary checking, careful pointer use, and disciplined memory management—you can write more reliable, maintainable, and secure programs. Tools like static analyzers and runtime checkers further assist in detecting these problems early.

19.3 Writing Memory-Safe Code

Memory safety is a critical aspect of writing reliable C programs. Since C gives you direct access to memory without automatic safety nets, it's easy to introduce bugs that cause crashes, data corruption, or security vulnerabilities. However, by following certain coding practices and using available tools, you can greatly reduce risks and write robust, memory-safe code.

This section outlines practical strategies for safer memory handling in C, covering input validation, bounds checking, careful use of memory functions, and proper management of allocation and deallocation. We'll also touch on useful tools for static and dynamic analysis that help catch errors early.

19.3.1 Validate Input Rigorously

Most memory issues originate from invalid or unexpected input. Whether input comes from a user, file, or network, **validating input length and format** is your first line of defense.

Example: Before copying a string into a fixed-size buffer, check its length.

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

void safe_copy(char *dest, size_t dest_size, const char *src) {
    if (strlen(src) >= dest_size) {
        fprintf(stderr, "Error: input too long for buffer\n");
        return;
    }
    strcpy(dest, src);
}
```

Alternatively, use `strncpy()` or `snprintf()` which limit the number of characters copied and help prevent overflows.

19.3.2 Always Check Buffer Boundaries

When working with arrays or buffers, never assume the data will fit. Carefully check loop indices, offsets, and sizes to prevent writing or reading beyond the allocated memory.

Unsafe loop example:

```c
char buffer[10];
for (int i = 0; i <= 10; i++) {  // off-by-one error, writes 11th element
    buffer[i] = 'a';
}
```

Safe loop example:

```c
char buffer[10];
for (int i = 0; i < sizeof(buffer); i++) {
    buffer[i] = 'a';
}
```

Use `sizeof()` or track buffer sizes explicitly, and avoid magic numbers.

19.3.3 Prefer Safer Functions Over Unsafe Ones

Many classic C functions lack built-in bounds checking. Replace unsafe functions with safer alternatives:

Unsafe Function	Safer Alternative
`strcpy`	`strncpy`, `strlcpy`
`strcat`	`strncat`, `strlcat`
`gets`	`fgets`
`sprintf`	`snprintf`

Example using `fgets()` safely:

```
char buffer[100];
printf("Enter your name: ");
if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
    // Remove trailing newline if present
    buffer[strcspn(buffer, "\n")] = '\0';
    printf("Hello, %s!\n", buffer);
}
```

`fgets()` prevents buffer overflow by limiting input size.

19.3.4 Proper Memory Allocation and Deallocation

When using dynamic memory, always:

- Check the return value of `malloc()`, `calloc()`, or `realloc()` for `NULL`.
- Avoid memory leaks by pairing every allocation with a `free()` when the memory is no longer needed.
- Set pointers to `NULL` after freeing to prevent dangling pointer usage.

Example:

```
int *arr = malloc(10 * sizeof(int));
if (arr == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return 1;
}

// Use arr...

free(arr);
arr = NULL;
```

For `realloc()`, assign to a temporary pointer first to avoid losing your original pointer if reallocation fails:

```
int *temp = realloc(arr, 20 * sizeof(int));
if (temp == NULL) {
    fprintf(stderr, "Reallocation failed\n");
    free(arr);
    return 1;
}
arr = temp;
```

19.3.5 Use Static and Dynamic Analysis Tools

Modern tools can automate detection of common memory errors:

- **Static analyzers** (e.g., Clang Static Analyzer, Coverity) analyze code without running it, detecting potential bugs.
- **Dynamic analysis tools** like **Valgrind** detect memory leaks, invalid reads/writes, and use-after-free errors at runtime.

Example Valgrind usage:

```
valgrind --leak-check=full ./your_program
```

These tools help find bugs early before they cause failures or security breaches.

19.3.6 Employ Defensive Programming Techniques

Writing code defensively means assuming things can go wrong and coding to handle those situations gracefully.

- Validate all inputs and outputs.
- Use assertions (`assert()`) to verify assumptions during development.
- Check all function return values, especially system calls and memory functions.
- Initialize pointers and variables before use.
- Document ownership and lifecycle of dynamically allocated memory clearly.

19.3.7 Example: Safe String and Array Handling

Here's a small example combining many of these principles:

Full runnable code:

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

#define MAX_NAME_LEN 50

int main() {
    char *name = malloc(MAX_NAME_LEN);
    if (!name) {
        fprintf(stderr, "Allocation failed\n");
        return 1;
    }

    printf("Enter your name (max %d chars): ", MAX_NAME_LEN - 1);
    if (fgets(name, MAX_NAME_LEN, stdin) == NULL) {
```

```
        fprintf(stderr, "Failed to read input\n");
        free(name);
        return 1;
    }

    // Strip newline
    name[strcspn(name, "\n")] = '\0';

    // Use the name safely
    printf("Hello, %s!\n", name);

    free(name);
    name = NULL;
    return 0;
}
```

This program prevents buffer overflows, checks for memory allocation errors, and cleans up properly.

19.3.8 Summary

Writing memory-safe C code requires vigilance and discipline:

- Validate all inputs and data sizes.
- Prefer bounded and safer standard library functions.
- Check allocation success and manage memory lifecycles carefully.
- Use modern tools to detect problems before they manifest.
- Code defensively with clear error handling and documentation.

These practices not only reduce bugs but also improve code readability and maintainability. With experience, safe memory management becomes second nature, unlocking the full power and performance of C with confidence.

Chapter 20.

Project: Building a Command-Line Calculator

1. Parsing Expressions
2. Stack-Based Evaluation
3. Modular Design

20 Project: Building a Command-Line Calculator

20.1 Parsing Expressions

Building a command-line calculator requires the ability to understand and process mathematical expressions entered by the user. This process is called **parsing**, and it involves reading an input string, breaking it down into meaningful components (tokens), and organizing those tokens so the program can evaluate the expression correctly.

In this section, we will explore how to parse simple arithmetic expressions from command-line input. We'll discuss tokenization, handling operators and numbers, and respecting operator precedence. Finally, we'll design a basic parser that breaks expressions into tokens such as numbers, operators, and parentheses, which is the first step toward evaluation.

20.1.1 What Is Parsing?

Parsing is the process of analyzing a string of symbols (characters) to determine its grammatical structure according to defined rules. For a calculator, the input expression like:

```
3 + 4 * (2 - 1)
```

needs to be decomposed into tokens:

- Numbers: `3`, `4`, `2`, `1`
- Operators: `+`, `*`, `-`
- Parentheses: `(`, `)`

After tokenization, these tokens can be processed respecting mathematical rules, especially **operator precedence** (multiplication before addition) and **associativity**.

20.1.2 Step 1: Tokenization

Tokenization means scanning the input string and separating it into atomic units called *tokens*. Each token has a type, such as:

- **NUMBER**: Represents numeric values (integers or floating-point).
- **OPERATOR**: Symbols like `+`, `-`, `*`, `/`.
- **PARENTHESIS**: Left `(` or right `)` parentheses.
- **END**: Special token indicating the end of input.

Why tokenize? Tokenization simplifies parsing because instead of dealing with raw characters, the program works with meaningful components.

20.1.3 Basic Tokenizer Design

A tokenizer reads characters from the input string one by one and groups them as follows:

- **Skip whitespace:** Spaces and tabs separate tokens but don't have meaning themselves.
- **Numbers:** One or more digits possibly including a decimal point.
- **Operators and parentheses:** Single characters representing math operations or grouping.

20.1.4 Sample Code: Tokenizer Implementation

Here's a simplified tokenizer in C that reads an input string and extracts tokens.

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

typedef enum {
    TOKEN_NUMBER,
    TOKEN_OPERATOR,
    TOKEN_PAREN_OPEN,
    TOKEN_PAREN_CLOSE,
    TOKEN_END,
    TOKEN_INVALID
} TokenType;

typedef struct {
    TokenType type;
    double value;       // valid if type == TOKEN_NUMBER
    char op;            // valid if type == TOKEN_OPERATOR
} Token;

typedef struct {
    const char *input;
    size_t pos;
    Token current_token;
} Tokenizer;

// Function to advance tokenizer to the next token
void next_token(Tokenizer *tz) {
    const char *str = tz->input;
    size_t i = tz->pos;

    // Skip whitespace
    while (isspace(str[i])) i++;

    char c = str[i];
    if (c == '\0') {
        tz->current_token.type = TOKEN_END;
        tz->pos = i;
        return;
    }
```

```
    if (isdigit(c) || c == '.') {
        // Parse number
        char buffer[64];
        size_t j = 0;
        while (isdigit(str[i]) || str[i] == '.') {
            if (j < sizeof(buffer) - 1) {
                buffer[j++] = str[i];
            }
            i++;
        }
        buffer[j] = '\0';
        tz->current_token.type = TOKEN_NUMBER;
        tz->current_token.value = atof(buffer);
        tz->pos = i;
        return;
    }

    if (c == '+' || c == '-' || c == '*' || c == '/') {
        tz->current_token.type = TOKEN_OPERATOR;
        tz->current_token.op = c;
        tz->pos = i + 1;
        return;
    }

    if (c == '(') {
        tz->current_token.type = TOKEN_PAREN_OPEN;
        tz->pos = i + 1;
        return;
    }

    if (c == ')') {
        tz->current_token.type = TOKEN_PAREN_CLOSE;
        tz->pos = i + 1;
        return;
    }

    tz->current_token.type = TOKEN_INVALID;
    tz->pos = i + 1;
}
```

Explanation:

- The `Tokenizer` struct holds the input string and current position.
- `next_token()` updates `current_token` by reading from `input`.
- Numbers are parsed using `atof()` after collecting all digit and decimal characters.
- Operators and parentheses are recognized as single characters.
- Whitespace is skipped.

20.1.5 Step 2: Handling Operators and Precedence

Once tokenized, your parser needs to respect **operator precedence**:

- Multiplication (`*`) and division (`/`) have higher precedence than addition (`+`) and

subtraction (`-`).
- Parentheses override precedence and force evaluation order.

Handling precedence usually involves parsing techniques such as **recursive descent parsing**, or converting expressions to **Reverse Polish Notation (RPN)** using the **Shunting Yard algorithm**.

For a simple calculator, you can:

- Parse expressions by recursive functions where:
  - One function handles addition and subtraction.
  - Another handles multiplication and division.
  - Another handles parentheses and numbers.

This method respects precedence naturally.

20.1.6 Example: Token Stream Walkthrough

For the input:

```
3 + 4 * (2 - 1)
```

The tokenizer produces:

Token Type	Value/Operator
NUMBER	3
OPERATOR	+
NUMBER	4
OPERATOR	*
PAREN_OPEN	(
NUMBER	2
OPERATOR	-
NUMBER	1
PAREN_CLOSE	)
END	

The parser will recognize that multiplication occurs before addition, and the parentheses indicate that subtraction happens before multiplication.

20.1.7 Summary

Parsing mathematical expressions from command-line input involves:

- **Tokenization**: Splitting the input into numbers, operators, and parentheses.
- **Operator Handling**: Recognizing the role and precedence of operators.
- **Parsing Design**: Structuring code (often with recursion) to process tokens respecting precedence and grouping.

The tokenizer example shown here is a solid foundation. From here, you can extend your parser to build an **abstract syntax tree (AST)** or use a stack-based algorithm for expression evaluation, which we will cover in the next chapter.

20.2 Stack-Based Evaluation

Once you have parsed an expression into tokens, the next step is **evaluating** it to produce a numeric result. One of the most powerful and elegant ways to evaluate arithmetic expressions is by using **stack-based algorithms**. In this section, we introduce key techniques such as the **Shunting Yard algorithm** and **Reverse Polish Notation (RPN)** evaluation, explain how stacks are used to handle operators and operands, and provide code examples.

20.2.1 Why Use Stack-Based Evaluation?

Mathematical expressions typically contain:

- **Operands** (numbers)
- **Operators** (like +, -, *, /)
- **Parentheses** to dictate grouping and precedence

Handling operator precedence and parentheses correctly can be tricky with simple left-to-right evaluation. Stack-based evaluation provides a systematic way to:

- Convert infix expressions (like `3 + 4 * 2`) to a form that's easy to compute.
- Evaluate expressions without recursion.
- Detect invalid input or errors during evaluation.

20.2.2 The Shunting Yard Algorithm: Converting to Reverse Polish Notation

Invented by Edsger Dijkstra, the **Shunting Yard algorithm** converts infix expressions (human-readable) into **Reverse Polish Notation (RPN)** — a postfix form where operators follow their operands.

Example:

Infix:

```
3 + 4 * 2 / (1 - 5)
```

RPN:

```
3 4 2 * 1 5 - / +
```

RPN expressions are straightforward to evaluate with a simple stack.

20.2.3 How Does the Shunting Yard Algorithm Work?

The algorithm uses two stacks:

- **Operator stack:** Holds operators and parentheses.
- **Output queue (or list):** Stores tokens in RPN order.

Basic steps:

1. Read tokens left to right.
2. When reading a **number**, add it directly to the output queue.
3. When reading an **operator**:
    - While there is an operator on the top of the operator stack with higher or equal precedence, pop it to the output queue.
    - Push the current operator on the operator stack.
4. When reading a **left parenthesis** `(`, push it onto the operator stack.
5. When reading a **right parenthesis** `)`, pop operators to the output queue until a left parenthesis is found on the stack (discard both parentheses).
6. After processing all tokens, pop any remaining operators to the output queue.

20.2.4 Evaluating the RPN Expression

Evaluating RPN is simpler:

- Use a stack to store operands.
- Read tokens from the RPN expression left to right.
- When a **number** is encountered, push it onto the stack.
- When an **operator** is encountered, pop the required number of operands (usually two), apply the operator, and push the result back onto the stack.
- After processing all tokens, the stack contains the final result.

20.2.5 Example: RPN Evaluation in C

Below is a simplified example of evaluating an RPN expression stored as tokens.

Full runnable code:

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

typedef enum { NUMBER, OPERATOR } TokenType;

typedef struct {
    TokenType type;
    double value;   // valid if NUMBER
    char op;        // valid if OPERATOR
} Token;

#define MAX_STACK_SIZE 100

typedef struct {
    double data[MAX_STACK_SIZE];
    int top;
} Stack;

void push(Stack *s, double val) {
    if (s->top < MAX_STACK_SIZE - 1) {
        s->data[++(s->top)] = val;
    } else {
        printf("Stack overflow!\n");
        exit(EXIT_FAILURE);
    }
}

double pop(Stack *s) {
    if (s->top >= 0) {
        return s->data[(s->top)--];
    } else {
        printf("Stack underflow!\n");
        exit(EXIT_FAILURE);
    }
}

double evaluate_rpn(Token tokens[], int length) {
    Stack stack = {.top = -1};

    for (int i = 0; i < length; i++) {
        Token t = tokens[i];
        if (t.type == NUMBER) {
            push(&stack, t.value);
        } else if (t.type == OPERATOR) {
            double b = pop(&stack);
            double a = pop(&stack);
            double res;

            switch (t.op) {
                case '+': res = a + b; break;
                case '-': res = a - b; break;
                case '*': res = a * b; break;
```

```
                case '/':
                    if (b == 0) {
                        printf("Error: Division by zero\n");
                        exit(EXIT_FAILURE);
                    }
                    res = a / b; break;
                default:
                    printf("Unknown operator: %c\n", t.op);
                    exit(EXIT_FAILURE);
            }
            push(&stack, res);
        }
    }
    if (stack.top != 0) {
        printf("Invalid expression: stack contains extra values\n");
        exit(EXIT_FAILURE);
    }
    return pop(&stack);
}

int main() {
    // Expression: 3 4 2 * 1 5 - / +
    Token expr[] = {
        {NUMBER, 3, 0},
        {NUMBER, 4, 0},
        {NUMBER, 2, 0},
        {OPERATOR, 0, '*'},
        {NUMBER, 1, 0},
        {NUMBER, 5, 0},
        {OPERATOR, 0, '-'},
        {OPERATOR, 0, '/'},
        {OPERATOR, 0, '+'},
    };
    int length = sizeof(expr) / sizeof(expr[0]);

    double result = evaluate_rpn(expr, length);
    printf("Result: %lf\n", result);  // Output: 3 + (4*2)/(1-5) = 3 + 8 / -4 = 3 - 2 = 1

    return 0;
}
```

20.2.6 Error Handling

Stack-based evaluation can detect errors like:

- **Stack underflow**: When an operator needs more operands than are available.
- **Division by zero**: When dividing by zero, report and terminate safely.
- **Extra operands**: After evaluation, if the stack has more than one value, the expression was invalid.

Proper error messages and early exits help debug user input issues.

20.2.7 Benefits of Stack-Based Evaluation

- **Simplicity**: Evaluating RPN is straightforward with a stack.
- **Precedence Handling**: The Shunting Yard algorithm cleanly converts infix to postfix while respecting precedence.
- **Extensibility**: New operators can be added easily by extending operator handling code.
- **Efficiency**: No recursion or complicated tree structures needed for basic arithmetic.

20.2.8 Summary

Stack-based evaluation breaks down expression computation into two phases:

1. **Parsing** the infix expression into RPN using the Shunting Yard algorithm.
2. **Evaluating** the RPN expression using a simple operand stack.

This approach ensures correct handling of operator precedence and parentheses, while providing clear, manageable code for expression evaluation. You can combine the tokenizer from the previous section with these stack algorithms to build a full-featured command-line calculator.

In the next section, we will explore how to organize this functionality into clean, modular code for easy maintenance and expansion.

20.3 Modular Design

Building a command-line calculator that handles expression parsing, evaluation, input/output, and error handling can quickly become complex if all the code is lumped together in a single file. To keep your code manageable, maintainable, and reusable, it is essential to organize the project into **modular components**. In this section, we discuss how to break down your calculator into logical modules, the benefits of modular design, and offer guidelines on structuring the codebase with clear interfaces and responsibilities.

20.3.1 Why Modular Design?

Modular design is a software engineering practice that involves splitting a program into distinct sections or modules, each responsible for a specific aspect of the overall functionality. The benefits include:

- **Maintainability:** Smaller, focused modules are easier to understand, fix, and update

without impacting unrelated code.
- **Testability:** Modules can be tested independently, making it easier to isolate bugs and verify correctness.
- **Reusability:** Modules can be reused in other projects or contexts with minimal changes.
- **Collaboration:** Modular code allows multiple developers to work simultaneously on different parts without conflicts.

For a calculator, typical concerns such as parsing expressions, evaluating them, managing input/output, and handling errors can be encapsulated in separate modules.

20.3.2 Suggested Module Breakdown

1. **Parser Module** Responsibility: Tokenize input strings and convert them into a structured form (e.g., tokens or an Abstract Syntax Tree). Key functions:
   - `tokenize()` — split input into numbers, operators, parentheses
   - `parse_expression()` — implement algorithms like Shunting Yard to order tokens correctly Example file: `parser.c` / `parser.h`
2. **Evaluator Module** Responsibility: Evaluate the parsed expression, usually in RPN form, producing a numeric result. Key functions:
   - `evaluate_rpn()` — process postfix tokens with a stack
   - `evaluate_expression()` — interface that takes parsed input and returns result or error Example file: `evaluator.c` / `evaluator.h`
3. **Input/Output Module** Responsibility: Handle reading user input and displaying output or error messages. Key functions:
   - `read_input()` — get input from the user or command line arguments
   - `print_result()` — display the final result or error info Example file: `io.c` / `io.h`
4. **Error Handling Module** Responsibility: Define error codes, messages, and handling mechanisms used throughout the calculator. Key functions:
   - `set_error()` / `get_error()`
   - `print_error_message()` Example file: `error.c` / `error.h`

20.3.3 Example File Structure

Here's how your project directory might look:

```
calculator/
```

```
+-- parser.c
+-- parser.h
+-- evaluator.c
+-- evaluator.h
+-- io.c
+-- io.h
+-- error.c
+-- error.h
+-- main.c
+-- Makefile
```

- **main.c** serves as the entry point that ties together all modules.
- Each .h file declares the interface (function prototypes, constants).
- Each .c file contains the implementation details.

20.3.4 Module Interfaces and Separation of Concerns

By defining clear interfaces in header files, you ensure that each module's internal workings are hidden from others, promoting encapsulation.

For example, **parser.h** might look like this:

```c
#ifndef PARSER_H
#define PARSER_H

#include "token.h"  // Defines Token and token types

int tokenize(const char *input, Token tokens[], int max_tokens);
int parse_expression(Token tokens[], int token_count, Token output[], int max_output);

#endif
```

The **evaluator.h** interface could be:

```c
#ifndef EVALUATOR_H
#define EVALUATOR_H

#include "token.h"

int evaluate_rpn(const Token tokens[], int length, double *result);

#endif
```

The **io.h** interface for input/output:

```c
#ifndef IO_H
#define IO_H

int read_input(char *buffer, int max_length);
void print_result(double result);
void print_error(const char *message);
```

```
#endif
```

And **error.h** could define error codes:

```
#ifndef ERROR_H
#define ERROR_H

typedef enum {
    ERR_NONE,
    ERR_SYNTAX,
    ERR_DIV_ZERO,
    ERR_MEMORY,
    // Add more as needed
} ErrorCode;

const char* get_error_message(ErrorCode code);

#endif
```

20.3.5 How the Modules Interact

- `main.c` calls `read_input()` to get user input.
- Input is passed to `tokenize()`, which breaks the string into tokens.
- Tokens are processed by `parse_expression()` to reorder them respecting operator precedence.
- The ordered tokens are then evaluated by `evaluate_rpn()` to produce a result.
- Errors at any step are captured via error codes and reported using `print_error()`.
- Finally, `print_result()` displays the output to the user.

This flow separates concerns clearly, so each module focuses on one task and can be modified or enhanced independently.

20.3.6 Best Practices for Modular Design

- **Single Responsibility:** Each module should do one thing well.
- **Minimal Dependencies:** Avoid unnecessary inter-module coupling.
- **Consistent Naming:** Use clear, descriptive function and variable names.
- **Documentation:** Comment interfaces and modules to explain usage.
- **Error Propagation:** Use error codes or mechanisms to communicate problems cleanly across modules.
- **Unit Testing:** Write tests for each module independently.

20.3.7 Summary

Modular design is a cornerstone of good software development. For your calculator project, dividing the program into parsing, evaluation, I/O, and error handling modules improves readability, debugging, and future enhancements. You get cleaner code, easier testing, and a scalable architecture that can grow — for example, by adding support for variables or more operators — without becoming unwieldy.

By planning your project structure and interfaces upfront, you set a strong foundation for robust and maintainable code.

Chapter 21.

Project: Simple File Compression Utility

1. Reading File Bytes
2. Using Bitwise Operations
3. Writing Compressed Output

21 Project: Simple File Compression Utility

21.1 Reading File Bytes

When building a file compression utility, one of the fundamental tasks is to read the raw data bytes from a file. Unlike text files where you might read line by line, compression requires working with the file in **binary mode**, reading the exact bytes as they exist without any translation or alteration. In this section, we'll explore how to open files for binary reading, efficiently read chunks of bytes, handle the end-of-file (EOF) condition, and implement robust error checking to ensure your program behaves correctly and reliably.

21.1.1 Opening a File in Binary Mode

In C, the standard library provides the `fopen()` function to open files. To read a file in binary mode, you pass the mode string `"rb"` (read binary). This mode ensures that bytes are read exactly as they are on disk, without any newline translation or character encoding conversions that might occur in text mode.

Here's a simple example of opening a file for binary reading:

```c
#include <stdio.h>

FILE *file = fopen("input.dat", "rb");
if (file == NULL) {
    perror("Error opening file");
    // Handle error (e.g., exit or retry)
}
```

If `fopen()` returns `NULL`, it means the file could not be opened, possibly due to it not existing, permission issues, or other file system errors. Always check the return value before proceeding.

21.1.2 Reading Raw Bytes Efficiently

Once the file is open, you can read raw bytes using `fread()`. This function reads a specified number of elements, each of a certain size, from the file into a buffer.

The syntax of `fread()` is:

```c
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

- `buffer` is the pointer to the memory where data will be stored.
- `size` is the size of each element (usually 1 byte when reading raw data).
- `count` is the number of elements to read.
- `stream` is the file pointer.

The function returns the number of elements successfully read, which can be less than `count` if the end of file is reached or an error occurs.

21.1.3 Reading in Chunks

Reading the entire file at once may not be practical for large files due to memory constraints. Instead, read in fixed-size chunks (buffers). This approach balances memory usage and performance.

Example reading chunks of 1024 bytes:

```
#define BUFFER_SIZE 1024

unsigned char buffer[BUFFER_SIZE];
size_t bytesRead;

while ((bytesRead = fread(buffer, 1, BUFFER_SIZE, file)) > 0) {
    // Process bytesRead bytes from buffer here
    // For example, compress or analyze the data
}
```

The loop continues until `fread()` returns 0, indicating no more data.

21.1.4 Handling End-of-File and Errors

Simply checking if `fread()` returns zero is not enough to distinguish between end-of-file and an error condition. Use `feof()` and `ferror()` to identify what happened.

Example:

```
if (ferror(file)) {
    perror("Error reading file");
    // Handle the error, possibly exit
} else if (feof(file)) {
    // End of file reached successfully
}
```

Always perform these checks after your read loop finishes.

21.1.5 Buffering Strategies

File I/O is buffered by default, meaning the system reads data in blocks behind the scenes to improve performance. When reading large files, you can tune the buffer size to optimize throughput.

Choosing a larger buffer size (e.g., 4KB or 8KB) can reduce the number of system calls and speed up reading. However, too large a buffer wastes memory, so balance according to your application needs.

You can also disable buffering for specific scenarios using `setbuf()` or `setvbuf()` but typically the default buffering is sufficient.

21.1.6 Complete Example: Reading a File Byte-by-Byte vs. Chunk

Here's a comparison of two approaches:

Byte-by-byte reading (inefficient):

```c
FILE *file = fopen("input.dat", "rb");
if (!file) {
    perror("Failed to open file");
    return 1;
}

int byte;
while ((byte = fgetc(file)) != EOF) {
    unsigned char c = (unsigned char)byte;
    // Process single byte c
}

fclose(file);
```

Chunk reading (efficient):

```c
#define BUFFER_SIZE 4096
unsigned char buffer[BUFFER_SIZE];
size_t bytesRead;

FILE *file = fopen("input.dat", "rb");
if (!file) {
    perror("Failed to open file");
    return 1;
}

while ((bytesRead = fread(buffer, 1, BUFFER_SIZE, file)) > 0) {
    // Process buffer[0..bytesRead-1]
}

if (ferror(file)) {
    perror("Error during file read");
}

fclose(file);
```

As you can see, chunk reading reduces function call overhead and generally runs faster, especially on larger files.

21.1.7 Best Practices for File Reading in Compression Utilities

- **Always check for errors:** After opening files and reading data, validate the success of these operations.
- **Read in reasonably sized buffers:** Avoid very small reads that cause performance hits and huge buffers that waste memory.
- **Clean up resources:** Always close files after use to free system resources.
- **Handle partial reads:** Sometimes `fread()` returns less than requested due to EOF or interruptions. Always process the actual bytes read.
- **Use binary mode:** For compression, you want exact bytes, so always open files with `"rb"` (read) and `"wb"` (write) modes for binary safety.

21.1.8 Summary

Reading raw bytes from files is foundational in creating a compression utility. By opening files in binary mode and reading data in chunks using `fread()`, you ensure efficient, accurate, and safe access to file contents. Proper error handling and resource management safeguard your program against common file I/O pitfalls. With these techniques, you are ready to build the next steps of your compression project, working directly on the raw byte data.

21.2 Using Bitwise Operations

In file compression, one of the keys to achieving compact storage is working directly with bits, the smallest units of data. Bitwise operations in C allow you to manipulate individual bits within bytes or larger data types, enabling techniques such as bit packing, masking, and shifting. These operations are essential for encoding data more efficiently by reducing the amount of space needed to store information like flags, repetitive values, or small ranges of numbers.

This section explains how bitwise operators can be used to compress data, along with practical examples that demonstrate packing multiple pieces of information into single bytes and manipulating bits effectively.

21.2.1 Why Use Bitwise Operations for Compression?

Typically, computers store data in bytes (8 bits), and many applications waste space by storing small pieces of information in full bytes or larger units. Bitwise operations help you:

- **Pack multiple boolean flags into one byte** (e.g., storing 8 on/off settings in a

single byte).

- **Represent small ranges of values compactly** using fewer bits than the standard types.
- **Manipulate data at the bit level** to implement custom compression schemes, such as run-length encoding or Huffman coding.

By working directly with bits, your program can dramatically reduce file sizes, an important goal for compression utilities.

21.2.2 Essential Bitwise Operators

Before diving into examples, let's review the core bitwise operators in C:

Operator	Description	Example
`&`	Bitwise AND: sets bits to 1 only if both bits are 1	`x & 0x0F` masks lower 4 bits
|	Bitwise OR: sets bits to 1 if either bit is 1	x | 0x80 sets the highest bit
`^`	Bitwise XOR: sets bits to 1 if bits differ	`x ^ 0xFF` inverts bits
`~`	Bitwise NOT: flips all bits	`~x` inverts every bit
`<<`	Left shift: shifts bits left, adding zeros on the right	`x << 2` shifts bits left by 2
`>>`	Right shift: shifts bits right	`x >> 3` shifts bits right by 3

21.2.3 Bit Masking: Extracting and Modifying Bits

Masking allows you to isolate or modify specific bits using `&` and `|`.

Extracting bits

Suppose you want to extract the lower 3 bits of a byte:

```
unsigned char byte = 0b10101100;  // binary literal
unsigned char lower3 = byte & 0x07; // 0x07 = 00000111 in binary
// lower3 is 0b00000100 (decimal 4)
```

Here, the mask `0x07` keeps only the lowest 3 bits, setting the rest to zero.

Setting bits

To set specific bits without affecting others, use bitwise OR (`|`):

```
unsigned char byte = 0b10100000;
byte = byte | 0x0F;  // Set the lower 4 bits to 1
// byte becomes 0b10101111
```

Clearing bits

To clear (set to zero) certain bits, use bitwise AND with a negated mask:

```
unsigned char byte = 0b11111111;
byte = byte & ~0x0F;  // Clear lower 4 bits
// byte becomes 0b11110000
```

21.2.4 Bit Shifting: Packing and Unpacking Data

Bit shifting moves bits left or right, enabling you to pack multiple small values into one larger value or extract them later.

Packing two 4-bit values into one byte

Imagine you have two values, each fitting in 4 bits (0-15):

```
unsigned char high = 9;  // 0b1001
unsigned char low = 5;   // 0b0101

unsigned char packed = (high << 4) | (low & 0x0F);
// packed = 0b10010101 = 149 decimal
```

The high value is shifted left 4 bits to occupy the upper nibble (half-byte), and the low value stays in the lower nibble. Masking `low` with `0x0F` ensures only the lower 4 bits are used.

To unpack:

```
unsigned char unpackedHigh = (packed >> 4) & 0x0F;
unsigned char unpackedLow = packed & 0x0F;
```

21.2.5 Example: Using Bits to Represent Flags

Suppose you want to store multiple boolean flags compactly, such as user permissions:

Permission	Bit Position
Read	0
Write	1
Execute	2
Delete	3
Share	4

You can use a single `unsigned char` to store these:

```c
unsigned char permissions = 0;  // All flags off

// Set Read and Execute permissions
permissions |= (1 << 0);  // Read
permissions |= (1 << 2);  // Execute

// Check if Write permission is set
if (permissions & (1 << 1)) {
    printf("Write permission granted\n");
} else {
    printf("Write permission denied\n");
}

// Clear Execute permission
permissions &= ~(1 << 2);
```

Using bits saves space and simplifies logic when handling multiple flags.

21.2.6 Compressing Repetitive Data with Bit Manipulation

Compression algorithms often encode repeated data compactly by packing counts and values together.

For example, in a simple run-length encoding (RLE) scheme, you could store a run length (up to 15) and a repeated value in a single byte:

```c
unsigned char runLength = 10; // max 15 (4 bits)
unsigned char value = 3;      // max 15 (4 bits)

unsigned char encoded = (runLength << 4) | (value & 0x0F);
```

This packing reduces two pieces of information into one byte, saving memory.

21.2.7 Practical Tips

- Always **mask bits** before combining to avoid overflow or unintended bits.
- Be mindful of **signed vs unsigned** data types when shifting bits.
- Use **parentheses** to ensure correct order of operations in expressions with shifts and bitwise operators.
- For multi-byte packing, consider **endianness** and portability.
- Combine bitwise operations with buffers for reading/writing compressed bit streams.

21.2.8 Summary

Bitwise operations empower your compression utility to represent data compactly and efficiently. Mastering masking, shifting, and bit packing techniques lets you squeeze more information into fewer bytes, which is the essence of compression. Whether managing flags, encoding repeated patterns, or designing custom formats, bitwise manipulation forms the foundation of many advanced compression algorithms.

With these tools, you can start crafting clever schemes to reduce file sizes and enhance your compression project's effectiveness.

21.3 Writing Compressed Output

After reading raw data from a file and applying bitwise operations to compress it, the next crucial step is to write this compressed data back to a file. Writing compressed output correctly ensures that the compressed file is compact, accurate, and can be decompressed without errors later. This section explores techniques for writing compressed data efficiently in binary mode, managing buffers, and maintaining file integrity.

21.3.1 Opening Files in Binary Mode for Writing

When writing compressed data, it's essential to open your output file in **binary mode** using `"wb"` mode with `fopen()`:

```
FILE *outFile = fopen("compressed.bin", "wb");
if (outFile == NULL) {
    perror("Failed to open output file");
    return 1;
}
```

Binary mode prevents unwanted transformations of bytes (like newline conversion on Windows), preserving exact bit patterns critical for compression.

21.3.2 Writing Bytes vs Bits

Since file systems generally store data in bytes (8 bits), but your compression algorithm might produce data in bits or partial bytes, you need a strategy to write these bits correctly.

- **Writing full bytes** is straightforward with `fwrite()` or `fputc()`.
- **Writing individual bits** requires accumulating bits in a buffer until you have a full byte to write.

21.3.3 Managing an Output Bit Buffer

To handle bits, maintain a byte-sized buffer and a counter tracking how many bits have been written so far:

```
unsigned char bitBuffer = 0;
int bitCount = 0;
FILE *outFile;  // Assume already opened

void writeBit(int bit) {
    bitBuffer = (bitBuffer << 1) | (bit & 1);
    bitCount++;

    if (bitCount == 8) {
        fputc(bitBuffer, outFile);
        bitBuffer = 0;
        bitCount = 0;
    }
}
```

Each call to `writeBit()` adds a single bit to the buffer. When the buffer reaches 8 bits, it writes the byte to the file and resets.

21.3.4 Flushing Remaining Bits

After processing all bits, you often have some bits left in the buffer (less than 8). To preserve data integrity, flush these bits by padding the remaining bits (usually with zeros) and writing the final byte:

```
void flushBits() {
    if (bitCount > 0) {
        bitBuffer <<= (8 - bitCount);  // Shift left to fill the rest with 0s
        fputc(bitBuffer, outFile);
        bitBuffer = 0;
        bitCount = 0;
    }
}
```

Without this step, the last bits might be lost or incorrectly interpreted during decompression.

21.3.5 Writing Full Bytes

Sometimes, the compressed data naturally aligns with bytes (e.g., after bit packing). Writing these bytes is simple:

```
unsigned char data = 0xAB;
fputc(data, outFile);

// Or writing multiple bytes:
```

```c
unsigned char buffer[] = {0xAB, 0xCD, 0xEF};
fwrite(buffer, sizeof(unsigned char), 3, outFile);
```

Use `fwrite()` for performance when writing large blocks at once.

21.3.6 Buffering Strategies

Efficient file I/O often uses buffering to minimize the number of system calls:

- The C standard library provides buffering automatically for file streams.
- You can use your own buffers to accumulate compressed data before writing larger chunks at once.

Example of buffering compressed bytes before writing:

```c
#define BUFFER_SIZE 1024
unsigned char outBuffer[BUFFER_SIZE];
int bufferIndex = 0;

void bufferWrite(unsigned char byte) {
    outBuffer[bufferIndex++] = byte;
    if (bufferIndex == BUFFER_SIZE) {
        fwrite(outBuffer, 1, BUFFER_SIZE, outFile);
        bufferIndex = 0;
    }
}

void flushBuffer() {
    if (bufferIndex > 0) {
        fwrite(outBuffer, 1, bufferIndex, outFile);
        bufferIndex = 0;
    }
}
```

Buffering reduces overhead and improves write performance, especially on large files.

21.3.7 Preserving File Integrity and Format

When writing compressed data, it's important to:

- **Write any metadata or headers** needed for decompression (e.g., original file size, compression method).
- **Maintain a consistent file format** so the decompressor knows how to interpret the bits.
- **Handle errors during writing** (e.g., disk full, permission issues) by checking the return values of `fwrite()` and `fputc()` and reacting appropriately.

Example:

```c
if (fwrite(outBuffer, 1, bufferIndex, outFile) != bufferIndex) {
    perror("Error writing output file");
    // Handle error (e.g., clean up and exit)
}
```

21.3.8 Example: Writing Bits to a File

Here's a small example combining the above concepts:

Full runnable code:

```c
#include <stdio.h>

FILE *outFile;
unsigned char bitBuffer = 0;
int bitCount = 0;

void writeBit(int bit) {
    bitBuffer = (bitBuffer << 1) | (bit & 1);
    bitCount++;
    if (bitCount == 8) {
        fputc(bitBuffer, outFile);
        bitBuffer = 0;
        bitCount = 0;
    }
}

void flushBits() {
    if (bitCount > 0) {
        bitBuffer <<= (8 - bitCount);
        fputc(bitBuffer, outFile);
        bitBuffer = 0;
        bitCount = 0;
    }
}

int main() {
    outFile = fopen("compressed.bin", "wb");
    if (!outFile) {
        perror("Failed to open file");
        return 1;
    }

    // Write bits: 1101 (4 bits)
    writeBit(1);
    writeBit(1);
    writeBit(0);
    writeBit(1);

    // Flush remaining bits to write partial byte
    flushBits();

    fclose(outFile);
```

```
    return 0;
}
```

This code writes four bits `1101` padded with zeros to form a byte.

21.3.9 Summary

Writing compressed output requires careful bit-level management to ensure data is stored compactly and correctly:

- Open files in binary mode (`"wb"`) to avoid data corruption.
- Use an output bit buffer to accumulate bits and write full bytes.
- Flush remaining bits by padding to maintain file integrity.
- Consider buffering strategies to optimize I/O performance.
- Always check for errors during writing and close files properly.

By following these techniques, your compression utility can produce reliable, space-saving files ready for efficient decompression.