The Little Book of C

Version 0.1.0

Duc-Tam Nguyen

2025-09-06

Table of contents

Part I. First Steps

Chapter 1. Getting started

1.1 What is C?

C is a programming language. But before we go deeper, let's step back: what is a programming language?

A programming language is a way for humans to give precise, step-by-step instructions to a computer. Just as we use English, Vietnamese, or Japanese to talk to each other, we use languages like C, Python, or Java to "talk" to a computer.

C was created in the early 1970s at Bell Labs by Dennis Ritchie. It was designed to write the UNIX operating system - and it became so successful that many modern languages (C++, Java, Rust, Go, even parts of Python) are deeply influenced by it.

Even today, more than fifty years later, C remains everywhere:

- The operating system inside your phone and laptop has thousands of lines of C code.
- Device drivers - the programs that let your computer talk to printers, cameras, or Wi-Fi cards - are usually written in C.
- Embedded systems, like the computer inside your microwave or car, are often powered by C.

C is sometimes called a "low-level high-level language":

- It's -low-level- because it lets you control memory and hardware directly, almost like assembly.
- It's -high-level- because it still uses readable words and structures that humans can understand, unlike raw 1s and 0s.

Think of C as the middle ground: close enough to the machine to be powerful, but high-level enough to be practical for humans.

A First Glimpse

Here's a very small piece of C code. Don't worry if it looks mysterious - we'll explain it step by step in the next sections:

```c
#include <stdio.h>

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

This tiny program does one thing: it prints the words `Hello, world!` on the screen.

Even though this is just a few lines, it already shows you the essence of programming: you write instructions, the computer follows them, and you get a result.

Why Learn C First?

1. C has a small set of features. You can learn the whole language core in this little book.
2. Once you learn C, you'll find Java, C++, Go, and others easier to understand.
3. C gives you direct access to memory, which is rare in modern languages. This helps you understand how computers really work.
4. C has been around for over 50 years, and it isn't going away anytime soon. Learning it is like learning the grammar of computing.

Why It Matters

Learning C is like learning to drive a manual car before moving to an automatic: it teaches you what's happening under the hood. Even if you later use higher-level languages, your knowledge of C will give you confidence and deeper insight into performance, efficiency, and problem-solving.

Exercises

1. Look up (online or in books) one example of a system you use every day that was written in C.
2. Explain in your own words the difference between a programming language and a human language.
3. If C is over 50 years old, why do you think it's still widely used today?

1.2 Hello, World!

Every journey in C programming begins with a simple tradition: writing a program that prints "Hello, world!" to the screen.

This may look small, but it teaches you the essentials of how a C program is structured.

The Full Program

Here is the complete code:

```c
#include <stdio.h>

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

When you run this program, it shows:

```
Hello, world!
```

Breaking It Down

Let's go through this step by step:

1. `#include <stdio.h>`

   - This tells the compiler: *"Before you build my program, please include the Standard Input/Output library."*
   - The file `stdio.h` contains the definition of `printf`, which we'll use to display text.
   - Without this line, the compiler wouldn't know what `printf` means.

2. `int main(void)`

   - Every C program starts execution at a special function called `main`.
   - `int` means that `main` will return an integer number to the operating system when it finishes.
   - `(void)` means *"main takes no arguments."* Later we'll see other forms that can take inputs.

3. `{ ... }`

   - The curly braces mark the start `{` and end `}` of a block of code.
   - Everything inside belongs to the `main` function.

4. `printf("Hello, world!\n");`

   - `printf` is a function that prints text.
   - The text we want to print is inside quotation marks `"..."`.
   - The `\n` is a newline character: it moves the cursor to the next line after printing. Without it, the next output would continue on the same line.
   - The semicolon `;` ends the instruction - in C, every statement must end with `;`.

5. `return 0;`

   - This line tells the operating system: -"The program finished successfully."-
   - A return value of `0` usually means success. Other numbers can indicate errors, which you'll learn about later.

How It Fits Together

- The program starts at `main`.
- It runs each line inside the braces.
- It prints `Hello, world!` with a newline.
- It exits and returns `0` to the system.

That's the life cycle of your very first program.

Why It Matters

This tiny program shows you many key ideas that repeat throughout C:

- Libraries (`#include`) give you tools you didn't write yourself.
- Functions (`main`, `printf`) are the building blocks of programs.
- Statements (ending with `;`) are instructions the computer follows.
- Return values (`return 0`) communicate success or failure.

Once you understand "Hello, world!", you can build more complex programs with confidence.

Exercises

1. Change the message in the program to print your name instead of "Hello, world!".
2. Remove the `\n` and see what happens when you run the program.
3. Try printing two lines by calling `printf` twice:

   ```c
   printf("First line\n");
   printf("Second line\n");
   ```

1.3 Compiling and Running a Program

Writing a program is only the first step. To make the computer understand it, we need to translate our C code into something the machine can run. This translation process is called compilation.

From Source Code to Program

When you write C code in a text file (for example, `hello.c`), you are writing source code - human-readable instructions. But your computer only understands machine code - long sequences of 0s and 1s.

The compiler's job is to take your source code and produce a program that your computer can execute.

The process usually looks like this:

1. Write code → You save your C code in a file like `hello.c`.
2. Compile → The compiler checks your code and translates it into an executable program.
3. Run → You execute the compiled program, and the computer follows your instructions.

Using a Compiler

The most common compiler today is GCC (GNU Compiler Collection). Another popular one is Clang. Both support the modern C23 standard.

Let's say your program is saved in a file called `hello.c`.

To compile it with GCC:

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

- `gcc` is the compiler command.

- `hello.c` is your source code.
- `-o hello` tells the compiler to create an output program named `hello`.

After this command, you will see a new file named `hello`.

Running the Program

On Linux or macOS, you can run it like this:

```
./hello
```

On Windows, you might just type:

```
hello.exe
```

And you should see:

```
Hello, world!
```

Common Mistakes

When compiling, beginners often see errors. Here are some typical ones:

- Missing semicolon:

  ```
  error: expected ';' before 'return'
  ```

  → In C, every statement must end with `;`.
- Misspelled function name:

  ```
  error: implicit declaration of function 'print'
  ```

  → You probably wrote `print` instead of `printf`.
- Missing quotes:

  ```
  error: missing terminating " character
  ```

  → Strings must always be inside `" "`.

Why It Matters

Understanding compilation is crucial because:

- It shows you that C is different from interpreted languages (like Python).
- It teaches you to think in two steps: write → compile → run.
- It prepares you for reading error messages - an essential skill for every programmer.

Exercises

1. Save the "Hello, world!" program in a file called `hello.c`, compile it with GCC or Clang, and run it.
2. Introduce a mistake (like removing a semicolon) and see what error message the compiler gives you.
3. Try compiling with a different output name, e.g.:

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

   Then run `./myprogram`.

1.4 Editing, Saving, and Errors

Now that you know how to compile and run a C program, let's talk about something every programmer does all the time: editing and fixing errors. Writing code isn't just about typing it once perfectly. In fact, most of programming is a cycle of edit → compile → fix errors → run → repeat.

Editing and Saving Code

C programs are just text files. You can edit them with any text editor. Some common choices:

- Linux/macOS: VS Code, Sublime Text, nano, vim
- Windows: VS Code, Notepad++, or even the built-in Notepad (though more advanced editors are recommended)

When you save your file, make sure it has the `.c` extension. For example:

```
hello.c
```

The compiler looks at that file extension to know it's C source code.

The Reality of Errors

It’s normal for your first compilation to show errors. Don’t be discouraged - errors are part of programming. Even professionals see them daily.

For example, if you forget a semicolon:

```c
#include <stdio.h>

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

When compiled, GCC might say:

```
hello.c: In function ‘main’:
hello.c:4:5: error: expected ‘;’ before ‘return’
```

This looks intimidating, but let’s break it down:

- `hello.c:4:5` → File `hello.c`, line 4, character 5
- `error: expected ‘;’ before ‘return’` → The compiler wanted a `;` but found `return` instead

Once you add the missing semicolon, the program compiles.

Warnings vs Errors

Compilers give two kinds of feedback:

- Errors → Your program cannot run until you fix them.
- Warnings → Your program will run, but the compiler is alerting you to something suspicious.

Example warning:

```c
int main(void) {
    int x;
    printf("%d\n", x);
    return 0;
}
```

The compiler may warn:

```
warning: ‘x’ is used uninitialized in this function
```

This means you're trying to print `x` before giving it a value - dangerous behavior. The program may run, but the result will be unpredictable.

Debugging Mindset

When you see an error:

1. Read it carefully - The compiler usually tells you the line number.
2. Don't panic - Errors are normal.
3. Fix one at a time - Often, one small mistake (like a missing semicolon) can cause multiple errors.
4. Recompile after each fix - Don't try to fix everything at once without testing.

Why It Matters

Learning how to read and respond to error messages is a key skill. Beginners who treat errors as enemies get frustrated. Experts see errors as helpful feedback. The compiler is your friend: it points out problems before your program misbehaves.

Exercises

1. Write a simple program with a deliberate mistake (for example, misspell `printf` as `prntf`). Compile it, and read the error message.
2. Remove a closing brace `}` and see what kind of error appears.
3. Create a program with both a warning and an error. Can you fix both?

1.5 Why C Still Matters

At this point, you might be wondering: *C is over fifty years old. Why should I learn it today?* After all, there are newer languages like Python, JavaScript, Rust, and Go. The answer is simple: C is still at the heart of computing.

C Is Everywhere

- The Linux kernel, Windows core, and macOS internals are mostly written in C.
- The tiny chips inside your car, microwave, TV, or washing machine often run code written in C.
- MySQL, SQLite, Git, and even parts of Python itself are implemented in C.
- Many performance-critical parts of game engines rely on C for speed.

Learning C means understanding the language that underpins much of modern software.

Foundation for Other Languages

Many modern languages borrow C's syntax and concepts:

- C++ extends C with classes and object-oriented features.
- Java, C#, Go, Rust all use C-like curly braces, operators, and control flow.
- Python itself is implemented in C, and its extension modules often use C for performance.

When you learn C, you build a foundation that transfers to other languages.

Performance and Control

C gives you direct access to memory. Unlike Python or Java, where memory is managed for you, in C you decide:

- Where data lives
- How much memory is used
- When memory is freed

This makes C extremely fast - and also teaches you how computers actually work under the hood.

Longevity and Stability

Languages come and go, but C endures. The C23 standard is the latest in a long line of updates that keep C modern while preserving compatibility. Code written decades ago in C can often still compile today.

That's rare in the world of technology.

A Mindset Shift

Learning C isn't just about syntax. It changes how you think:

- You learn to be precise.
- You become comfortable with errors and debugging.
- You start thinking about efficiency, not just correctness.

C teaches you programming at its most essential.

Why It Matters

Even if you later move on to Python, JavaScript, or Rust, learning C gives you a mental model that makes you a stronger programmer. It's like learning arithmetic before using a calculator: once you understand the basics, you can appreciate and use advanced tools better.

Exercises

1. Look up three widely used software projects that are written in C.
2. Ask yourself: if you had to write a program for a device with very little memory (like a smartwatch or a sensor), why might C be a good choice?
3. Compare the age of C (1972) with another technology you use every day. Why do you think C has lasted so long?

Chapter 2. Variables and Types

2.1 Numbers and Text

So far, our programs only printed fixed text like `"Hello, world!"`. That's nice, but real programs need to work with data - numbers, words, and symbols that can change each time the program runs. In C, we represent data using variables.

Variables: Boxes for Data

Think of a variable as a box with a label on it:

- The box holds some value (like a number or letter).
- The label (the variable's name) tells us how to refer to it later.

In C, you must tell the computer what kind of data the box holds. This is called the type of the variable.

Numbers

C supports several kinds of numbers:

- Integers: whole numbers like `-5`, `0`, `42`
- Floating-point numbers: numbers with decimal points like `3.14`, `-0.5`, `2.71828`

Here's a simple program that uses both:

```c
#include <stdio.h>

int main(void) {
    int age = 20;              // an integer
    float height = 1.75;       // a floating-point number

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

    return 0;
}
```

Output:

```
Age: 20
Height: 1.75 meters
```

Notice:

- `%d` tells `printf` to print an integer.
- `%f` tells `printf` to print a floating-point number.
- `%.2f` means "print with 2 digits after the decimal point."

Text (Characters)

C represents text as characters, written in single quotes: `'A'`, `'z'`, `'?'`. Characters are stored as small integer codes (ASCII codes).

Example:

```c
#include <stdio.h>

int main(void) {
    char grade = 'A';    // a single character

    printf("Your grade is %c\n", grade);

    return 0;
}
```

Output:

```
Your grade is A
```

The `%c` format specifier prints a single character.

Text (Strings)

A string is a sequence of characters, like `"Hello"` or `"C programming"`. In C, strings are written inside double quotes `"..."` and stored as arrays of characters (we'll learn more in Chapter 6).

For now, you can print them directly:

```c
#include <stdio.h>

int main(void) {
    char name[] = "Alice";

    printf("Hello, %s!\n", name);

    return 0;
}
```

Output:

```
Hello, Alice!
```

The `%s` format specifier prints a string.

Why It Matters

- Variables let your program handle different inputs instead of fixed text.
- Types ensure the computer knows how to store and work with the data.
- By learning numbers and text, you're starting to write programs that can interact with the world, not just display the same message every time.

Exercises

1. Write a program that declares an integer `year` and prints:

   ```
   The year is 2025
   ```

   using `printf`.

2. Modify the program to include your favorite decimal number (like your height or a constant such as). Print it with 3 decimal places.

3. Create a program that stores your first initial as a `char` and prints:

   ```
   My initial is X
   ```

   replacing `X` with your character.

2.2 Declaring Variables

In the previous section, we used variables like `int age = 20;` and `float height = 1.75;`. But what does it mean to declare a variable in C?

The Anatomy of a Declaration

A variable declaration in C has two parts:

```
type name;
```

- type → what kind of data the variable will hold (e.g., `int`, `float`, `char`)
- name → the label you give the variable, so you can use it later

Example:

```c
int score;
```

This means: "Make a box named `score` that can hold an integer." Right now, the contents of the box are undefined (whatever random value happened to be in memory).

Initialization

You can also give a variable an initial value when declaring it:

```c
int score = 100;
```

This both creates the variable and sets its value. This is called initialization.

If you don't initialize, C does not set it to zero automatically. Using an uninitialized variable is a common beginner mistake.

Multiple Declarations

You can declare several variables of the same type on one line:

```c
int x = 1, y = 2, z = 3;
```

This creates three integers: `x`, `y`, and `z`. For readability, beginners are encouraged to declare each variable on its own line.

Naming Rules

Variable names in C:

- Must start with a letter or underscore (`_`)
- Can contain letters, digits, and underscores
- Are case-sensitive (`Age` and `age` are different)
- Cannot be a keyword (e.g., `int`, `return`, `for`)

Good names make your code clearer. Compare:

```c
int x;   // unclear
int age; // clear
```

A Full Example

Here's a small program that shows variable declaration, initialization, and usage:

```c
#include <stdio.h>

int main(void) {
    int year = 2025;           // integer
    float pi = 3.14159;        // floating-point number
    char initial = 'N';        // character
    int a = 5, b = 10;         // multiple integers

    printf("Year: %d\n", year);
    printf("Value of pi: %.2f\n", pi);
    printf("My initial: %c\n", initial);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Output:

```
Year: 2025
Value of pi: 3.14
My initial: N
a = 5, b = 10
```

Why It Matters

- Declaring variables tells the computer what kind of information to expect.
- Initializing ensures your variables start with predictable values.
- Good naming makes your code readable and easier to maintain.

Without proper declarations, your program won't compile, or worse - it will run with unpredictable results.

Exercises

1. Write a program that declares an integer `day`, a float `temperature`, and a char `grade`. Initialize them and print their values.
2. Create two integer variables, `x` and `y`, and print their sum.
3. Try declaring a variable without initializing it. Print it - what happens? Why is this dangerous?

2.3 Basic Types in C23

Every variable in C must have a type. The type tells the computer:

- How much memory to reserve
- How to interpret the bits stored inside
- What operations are allowed

C23 keeps the same familiar types from older standards but also adds refinements for safety and clarity.

Integers

- `int` → most common integer type (size depends on the system, often 32 bits)
- `short`, `long`, `long long` → smaller or larger ranges
- `unsigned` versions → only non-negative values

Examples:

```c
int age = 20;
unsigned int population = 8000000;
long long stars = 10000000000LL;
```

Floating-Point Numbers

For decimals and scientific values:

- `float` → typically 32-bit
- `double` → 64-bit, more precise
- `long double` → even more precision (implementation-dependent)

Examples:

```c
float pi = 3.14f;
double e = 2.718281828;
```

Characters

- `char` → stores a single character, like `'A'` or `'z'`
- Stored as an integer code (usually ASCII or UTF-8 in modern systems)

Example:

```c
char letter = 'C';
```

Booleans (C23 and C99+)

C did not originally have a true boolean type. Now you can use:

```c
#include <stdbool.h>

bool is_happy = true;
bool is_sad = false;
```

Special Types in C23

C23 introduces safer and clearer usage:

- `nullptr`: a special constant to represent "no pointer" (instead of older `NULL`).
- `char8_t`: for explicit UTF-8 characters, making Unicode handling clearer.

Example:

```c
#include <stddef.h>   // for nullptr
#include <uchar.h>    // for char8_t

int -p = nullptr;     // safe null pointer
char8_t smiley = u8' ';  // UTF-8 character literal
```

A Full Example

This program demonstrates several basic types side by side:

```c
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <uchar.h>

int main(void) {
    int year = 2025;
    unsigned int population = 1000000;
    float pi = 3.14159f;
    double e = 2.718281828;
    char grade = 'A';
    bool is_student = true;
    int -ptr = nullptr;
    char8_t smiley = u8' ';

    printf("Year: %d\n", year);
    printf("Population: %u\n", population);
    printf("Pi: %.2f\n", pi);
    printf("Euler's number: %.5f\n", e);
    printf("Grade: %c\n", grade);
    printf("Is student: %s\n", is_student ? "true" : "false");
    printf("Pointer: %p\n", (void-)ptr);
    printf("Smiley (UTF-8 code): U+%04X\n", smiley);

    return 0;
}
```

Sample Output:

```
Year: 2025
Population: 1000000
Pi: 3.14
Euler's number: 2.71828
Grade: A
Is student: true
Pointer: (nil)
Smiley (UTF-8 code): U+263A
```

Why It Matters

Understanding types is crucial because:

- They define the shape of your data.
- Choosing the right type avoids wasted memory or incorrect results.
- Modern C23 types (`bool`, `nullptr`, `char8_t`) make code safer and clearer.

Exercises

1. Declare one variable of each type (`int`, `float`, `double`, `char`, `bool`). Print them using `printf`.
2. Try printing an `unsigned int` with `%d`. What happens? Why is it wrong?
3. Write a program that stores the value of   in both `float` and `double`, then prints both with 10 decimal places. Compare the results.

2.4 Constants (`const` and `constexpr`)

So far, we've seen variables like:

```c
int age = 20;
float pi = 3.14159f;
```

But sometimes, we want values that should never change once set. These are called constants.

Constants make your programs safer, clearer, and easier to maintain.

`const`

The `const` keyword tells the compiler: -"This variable's value cannot be changed."-

Example:

```c
const int days_in_week = 7;
```

If you try to assign a new value later:

```c
days_in_week = 8;   //  Error: assignment of read-only variable
```

`constexpr` (C23 and C11+)

The `constexpr` keyword ensures that a value is calculated at compile time, not at runtime. It's like telling the compiler: -"Work this out in advance."-

Example:

```c
constexpr int minutes_in_day = 24 - 60;
```

This guarantees that `minutes_in_day` is a constant known before the program even runs.

Difference Between `const` and `constexpr`

- `const` → value cannot be changed after the program starts
- `constexpr` → value must be known before the program starts (at compile time)

Every `constexpr` is also `const`, but not every `const` is `constexpr`.

A Full Example

Here's a program that shows both `const` and `constexpr` in action:

```c
#include <stdio.h>

int main(void) {
    const float pi = 3.14159f;                // constant value
    constexpr int seconds_in_hour = 60 - 60; // compile-time constant
    int radius = 5;

    float circumference = 2 - pi - radius;  // uses const
    int hours = 2;
    int seconds = hours - seconds_in_hour;  // uses constexpr

    printf("Radius: %d\n", radius);
    printf("Circumference: %.2f\n", circumference);
    printf("%d hours = %d seconds\n", hours, seconds);

    return 0;
}
```

Output:

```
Radius: 5
Circumference: 31.42
2 hours = 7200 seconds
```

Why It Matters

- Safety → prevents accidental changes to values that should stay fixed.
- Clarity → communicates intent (“this never changes”).
- Performance → `constexpr` lets the compiler compute values ahead of time.

Good C programs rely heavily on constants for configuration and clarity.

Exercises

1. Declare a `const int` for the number of days in a year and print it.
2. Write a program using `constexpr` to calculate the number of minutes in a week.
3. Try declaring `const int x = 10;` and then reassigning `x = 20;`. What happens?

2.5 Input and Output with `scanf` and `printf`

So far, our programs only showed fixed values using `printf`. But real programs need to interact with the user:

- Input → let the user type values
- Output → display results

In C, the two main functions for this are:

- `printf` → prints output to the screen
- `scanf` → reads input from the user

Both are defined in the `stdio.h` library.

Using `printf`

You’ve already seen examples like:

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

The key idea is format specifiers:

- `%d` → integer
- `%f` → floating-point number
- `%c` → single character
- `%s` → string

Example:

```c
int age = 20;
printf("I am %d years old.\n", age);
```

Using `scanf`

The `scanf` function reads user input.

Syntax:

```c
scanf("format", &variable);
```

- `"format"` tells `scanf` what kind of data to expect
- `&variable` gives the address of the variable (so `scanf` can write into it)

Example:

```c
int age;
scanf("%d", &age);
```

This reads an integer from the keyboard and stores it in `age`.

A Full Example

Here's a complete program that combines input and output:

```c
#include <stdio.h>

int main(void) {
    int age;
    float height;
    char initial;
    char name[20];   // enough space for up to 19 characters + '\0'

    printf("Enter your age: ");
    scanf("%d", &age);
    printf("Enter your height in meters: ");
    scanf("%f", &height);

    printf("Enter your first initial: ");
    scanf(" %c", &initial);   // notice the space before %c to skip whitespace

    printf("Enter your name: ");
    scanf("%19s", name);      // read string (up to 19 chars)

    printf("\n--- Profile ---\n");
    printf("Name: %s\n", name);
    printf("Initial: %c\n", initial);
    printf("Age: %d years\n", age);
    printf("Height: %.2f m\n", height);

    return 0;
}
```

Sample Run:

```
Enter your age: 21
Enter your height in meters: 1.72
Enter your first initial: A
Enter your name: Alice

--- Profile ---
Name: Alice
Initial: A
Age: 21 years
Height: 1.72 m
```

Why It Matters

- `printf` makes your program communicate results.
- `scanf` makes your program interactive, letting users supply data.
- Together, they turn static programs into useful tools.

Almost every C program you'll write uses these functions in some way.

Exercises

1. Write a program that asks the user for two integers and prints their sum.
2. Ask the user for their name and favorite number, then print:

   ```
   Hello NAME, your favorite number is N.
   ```

3. Modify the full example to include weight in kilograms and print the BMI (Body Mass Index = weight / (height - height)).

Problems

1. My First Profile

Write a program that asks the user for their name, age, and favorite character, then prints them in a short introduction. Example run:

```
Enter your name: Alice
Enter your age: 21
Enter your favorite character: Z

Hello Alice! You are 21 years old and your favorite character is Z.
```

2. Temperature Converter

Ask the user for a temperature in Celsius and print it in Fahrenheit using the formula:

```
F = C × 9/5 + 32
```

3. Rectangle Calculator

Read two integers from the user: `length` and `width`. Print both the area and the perimeter of the rectangle.

4. Circle Calculator (with const)

Use a `const float pi = 3.14159f;` to calculate the area and circumference of a circle given its radius. Input the radius from the user.

5. Minutes in a Week (with constexpr)

Define a `constexpr int` to represent the number of minutes in a week. Print the result.

6. Swap Two Numbers

Ask the user for two integers and print them before and after swapping their values using a temporary variable.

7. Character Codes

Ask the user to type a character. Print both the character and its ASCII code (integer value). Example:

```
Enter a character: A
You entered: A
ASCII code: 65
```

8. String Greeting

Ask the user for their first name and print:

```
Hello, NAME! Nice to meet you.
```

9. Simple BMI Calculator

Ask the user for height (meters) and weight (kilograms). Calculate and print their Body Mass Index (BMI = weight / (height - height)) with 2 decimal places.

10. Sum of Three Numbers

Ask the user for three integers and print their sum and average.

11. Days to Hours and Minutes

Ask the user for a number of days. Calculate and print how many hours and minutes that equals.

12. Initials Program

Ask the user for their first, middle, and last initials (as `char`). Print them together as one string:

```
Enter your initials: A B C
Your initials are: ABC
```

13. Circle Comparison

Ask the user for two radii, `r1` and `r2`. Use constants for . Print which circle is larger by area.

14. Student Pass/Fail

Ask the user for an integer grade (0–100). Print `Pass` if it is 50 or above, otherwise `Fail`.

15. Profile with Multiple Types

Declare and initialize:

- An `int` for your current year (e.g. 2025)
- A `float` for your height
- A `char` for your grade
- A `char[]` string for your name

Print them all in a formatted way, like:

```
Name: Alice
Year: 2025
Height: 1.72 m
Grade: A
```

16. Age in Seconds

Ask the user for their age in years. Approximate and print how many seconds they have lived, assuming 365 days per year.

17. Favorite Number Game

Ask the user for their favorite integer. Print the number, its square, and its cube.

18. Welcome Banner

Ask the user for their name and print it inside a “banner” made of `-` symbols:

```
Enter your name: Bob

- Bob  -
```

19. Type Limits Exploration (Bonus)

Use `<limits.h>` and `<float.h>` to print the minimum and maximum values of `int`, `unsigned int`, `float`, and `double`. (Not beginner-essential, but a good exploration.)

20. Combine Everything

Write a program that:

1. Uses `const` for .
2. Reads an integer `age`, a float `height`, a char `initial`, and a string `name`.
3. Prints them all back with labels. This “mini-profile” program should combine all concepts from Chapter 2.

Chapter 3. Expressions and Operators

3.1 Arithmetic Operators

Arithmetic operators let your program calculate with numbers. In C, the basic operators are:

- `+` (addition)
- `-` (subtraction / unary negation)
- `-` (multiplication)
- `/` (division)
- `%` (remainder, modulo - integers only)

You’ll use them with both integers (`int`, `long`, …) and floating-point numbers (`float`, `double`).

Quick Examples

```c
int a = 7 + 3;        // 10
int b = 7 - 3;        // 4
int c = 7 - 3;        // 21
int d = 7 / 3;        // 2  (integer division truncates the fraction)
int r = 7 % 3;        // 1  (remainder)

float x = 7.0f / 3;  // 2.333333 (floating division)
double y = 7 / 3.0;  // 2.333333 (promoted to double)
```

Unary Minus

You can flip the sign of a number with unary `-`:

```c
int k = 5;
int neg = -k;    // -5
```

Modulo (%)

The `%` operator gives the remainder after integer division:

```c
int r1 = 11 % 4;    // 3  (11 = 2-4 + 3)
int r2 = 7 % 2;     // 1
```

Modulo is useful for tasks like checking even/odd (`n % 2`).

A Full Example

This program shows all arithmetic operators with two integers and also demonstrates floating division:

```c
#include <stdio.h>

int main(void) {
    int a, b;
    printf("Enter two integers (a b): ");
    scanf("%d %d", &a, &b);
    int sum  = a + b;
    int diff = a - b;
    int prod = a - b;
    int quot = a / b;      // integer division
    int rem  = a % b;      // remainder
    double fquot = (double)a / b; // floating division

    printf("\n--- Results ---\n");
    printf("%d + %d = %d\n", a, b, sum);
    printf("%d - %d = %d\n", a, b, diff);
    printf("%d - %d = %d\n", a, b, prod);
    printf("%d / %d = %d (integer division)\n", a, b, quot);
    printf("%d %% %d = %d (remainder)\n", a, b, rem);
    printf("%d / %d = %.6f (floating division)\n", a, b, fquot);
    printf("Unary minus of %d is %d\n", a, *a);

    return 0;
}
```

Example run:

```
Enter two integers (a b): 11 4

--- Results ---
11 + 4 = 15
11 - 4 = 7
11 - 4 = 44
11 / 4 = 2 (integer division)
11 % 4 = 3 (remainder)
11 / 4 = 2.750000 (floating division)
Unary minus of 11 is -11
```

Why It Matters

Arithmetic operators are the foundation of programming. They let you move beyond printing fixed text and start writing programs that calculate answers. Understanding the difference between integer and floating division is one of the first "aha!" moments for every C beginner.

Exercises

1. Write a program that reads two integers and prints their sum, difference, product, integer division, remainder, and floating division.
2. Read one integer and print its square and cube.
3. Read one integer and print whether it is even or odd using `%`. (Just print the remainder, we'll learn `if` later.)
4. Read three integers and print their total and average (as floating).
5. Read two integers and print the result of applying unary minus to each.

3.2 Assignment and Precedence

So far we've calculated values and stored them in variables like this:

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

This uses the assignment operator `=`. It means: -take the value on the right-hand side and store it into the variable on the left-hand side.-

Basic Assignment

```
int x;     // declare a variable
x = 5;     // assign the value 5 to x
```

After this, `x` holds the number `5`.

You can change it later:

```
x = 10;    // now x holds 10
```

Combined Assignments

C has shorthand operators that combine calculation and assignment:

- `x += y;` $\rightarrow$ same as `x = x + y;`
- `x -= y;` $\rightarrow$ same as `x = x - y;`
- `x -= y;` $\rightarrow$ same as `x = x - y;`
- `x /= y;` $\rightarrow$ same as `x = x / y;`
- `x %= y;` $\rightarrow$ same as `x = x % y;`

These make code shorter and often clearer.

Operator Precedence

When an expression has multiple operators, C needs to decide which to do first. This is called precedence.

General order (from higher to lower, the ones we've seen so far):

1. Parentheses `( )`
2. Unary minus `-x`
3. Multiplication, division, modulo `- / %`
4. Addition, subtraction `+ -`
5. Assignment `=`

So:

```c
int r = 2 + 3 - 4;     // 3-4 happens first → 2 + 12 = 14
int s = (2 + 3) - 4;   // parentheses force addition first → 5-4 = 20
```

Always use parentheses when in doubt. They make code easier to read and prevent mistakes.

A Full Example

This program shows assignment, combined assignments, and precedence:

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int y = 3;

    printf("Initial: x = %d, y = %d\n", x, y);

    // Basic assignment
    x = x + y;
    printf("x = x + y → %d\n", x);

    // Reset x
    x = 10;

    // Combined assignment
    x += y;
    printf("x += y → %d\n", x);
    x -= y;
    printf("x -= y → %d\n", x);

    x -= y;
    printf("x -= y → %d\n", x);

    x /= y;
    printf("x /= y → %d\n", x);

    x = 10;
    x %= y;
    printf("x %%= y → %d\n", x);

    // Precedence
    int a = 2 + 3 - 4;
    int b = (2 + 3) - 4;
    printf("2 + 3 - 4 = %d\n", a);
    printf("(2 + 3) - 4 = %d\n", b);

    return 0;
}
```

Example run:

```
Initial: x = 10, y = 3
x = x + y → 13
x += y → 13
x -= y → 10
x -= y → 30
x /= y → 10
x %= y → 1
2 + 3 - 4 = 14
(2 + 3) - 4 = 20
```

Why It Matters

- Assignment is how you store results into variables.
- Combined assignments make updates simpler (important in loops later).
- Precedence ensures the computer reads your math the same way you do - or lets you override with parentheses.

Exercises

1. Start with `int n = 10;`. Use `+=`, `-=`, `-=`, `/=`, `%=` with another integer and print the result after each step.
2. Calculate `5 + 2 - 3` and `(5 + 2) - 3` and print both results.
3. Write a program that computes `10 - 4 + 2 - 3`. Then add parentheses in different places and print the different results.
4. Declare `int a = 7, b = 2;`. Compute and print `(a + b) - (a - b)`.
5. Show that `x = x + 1;` and `x += 1;` give the same result.

3.3 Relational and Logical Operators

Programs often need to compare values or combine conditions. C provides operators for this. They don't give you numbers like `5` or `7` - instead they produce results that are either:

- 1 → means *true*
- 0 → means *false*

This is how C represents truth values internally.

Relational Operators

These compare two values:

- `==` → equal
- `!=` → not equal
- `<` → less than
- `>` → greater than
- `<=` → less than or equal
- `>=` → greater than or equal

Examples:

```c
int a = 5, b = 3;

int r1 = (a == b);  // 0 (false)
int r2 = (a > b);   // 1 (true)
int r3 = (a <= 5);  // 1 (true)
```

Logical Operators

These combine true/false values:

- `&&` → logical AND (true only if both are true)
- `||` → logical OR (true if at least one is true)
- `!` → logical NOT (flips true/false)

Examples:

```
int x = 1, y = 0;

int r1 = (x && y); // 0 (1 AND 0 is false)
int r2 = (x || y); // 1 (1 OR 0 is true)
int r3 = (!x);     // 0 (NOT 1 is false)
```

A Full Example

Here's a program that shows relational and logical operators in action:

```
#include <stdio.h>

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

    printf("\n--- Relational ---\n");
    printf("%d == %d → %d\n", a, b, a == b);
    printf("%d != %d → %d\n", a, b, a != b);
    printf("%d <  %d → %d\n", a, b, a < b);
    printf("%d >  %d → %d\n", a, b, a > b);
    printf("%d <= %d → %d\n", a, b, a <= b);
    printf("%d >= %d → %d\n", a, b, a >= b);

    printf("\n--- Logical ---\n");
    int x = (a > 0);  // true if a is positive
    int y = (b > 0);  // true if b is positive

    printf("(a > 0) → %d\n", x);
    printf("(b > 0) → %d\n", y);
    printf("(a > 0) && (b > 0) → %d\n", x && y);
    printf("(a > 0) || (b > 0) → %d\n", x || y);
    printf("!(a > 0) → %d\n", !x);

    return 0;
}
```

Example run:

```
Enter two integers (a b): 5 3

--- Relational ---
5 == 3 → 0
5 != 3 → 1
5 <  3 → 0
5 >  3 → 1
5 <= 3 → 0
5 >= 3 → 1

--- Logical ---
(a > 0) → 1
(b > 0) → 1
(a > 0) && (b > 0) → 1
(a > 0) || (b > 0) → 1
!(a > 0) → 0
```

Why It Matter

Relational and logical operators are how C evaluates conditions. They let you compare numbers and combine truth values, producing results as `1` (true) or `0` (false). This gives you a way to test equality, check ranges, and express logical ideas directly in your code.

Even in simple printouts, these operators show how the computer understands relationships and logic in numeric form.

Exercises

1. Read two integers and print the results of all six relational operators.
2. Read one integer and print `n % 2 == 0` to check if it's even (you'll see 1 or 0).
3. Read two integers `a` and `b`, and print `(a > 0) && (b > 0)`. Try positive and negative inputs.
4. Print the result of `!(a == b)` and compare it with `a != b`.
5. Experiment with `(a > 5) || (b < 10)` and record the outputs for different inputs.

3.4 Working with Characters

In C, characters are stored in variables of type `char`. Even though they look like letters, underneath they are just small integers representing codes (usually ASCII). This means you can compare them, add or subtract from them, and print both the character and its numeric code.

Declaring and Printing Characters

A character literal is written in single quotes:

```c
char letter = 'A';
printf("Letter: %c\n", letter);
```

Here `%c` prints the character itself. If you use `%d`, you'll see its integer code:

```c
printf("Code of %c is %d\n", letter, letter);
```

Output:

```
Code of A is 65
```

Character Arithmetic

Because characters are stored as numbers, you can do math with them.

```c
char letter = 'A';
char next = letter + 1;   // 'B'
```

Similarly:

```c
char digit = '3';
char nextdigit = digit + 1;  // '4'
```

Character Comparisons

You can compare characters with relational operators:

```c
char c = 'm';

int is_lower = (c >= 'a' && c <= 'z');  // 1 if lowercase
int is_upper = (c >= 'A' && c <= 'Z');  // 1 if uppercase
```

This works because ASCII letters are stored in order.

A Full Example

This program reads one character and shows its properties:

```c
#include <stdio.h>

int main(void) {
    char c;
    printf("Enter a character: ");
    scanf(" %c", &c);   // space before %c skips whitespace

    printf("\n--- Character Info ---\n");
    printf("Character: %c\n", c);
    printf("ASCII code: %d\n", c);

    printf("Next character: %c\n", c + 1);
    printf("Previous character: %c\n", c - 1);

    printf("Is uppercase? %d\n", (c >= 'A' && c <= 'Z'));
    printf("Is lowercase? %d\n", (c >= 'a' && c <= 'z'));
    printf("Is digit?     %d\n", (c >= '0' && c <= '9'));

    return 0;
}
```

Example run:

```
Enter a character: m

--- Character Info ---
Character: m
ASCII code: 109
Next character: n
Previous character: l
```

```
Is uppercase? 0
Is lowercase? 1
Is digit?     0
```

Why It Matters

Characters are the building blocks of text. Knowing that they are really numbers helps you:

- understand comparisons (`'a' < 'z'`)
- move through letters and digits by simple arithmetic
- start working with strings later, since strings are arrays of characters

Exercises

1. Read one character and print its ASCII code.
2. Read a digit character (e.g. `'5'`) and print the next digit.
3. Read a letter and print both its lowercase and uppercase version by adding or subtracting 32 (`'A' + 32 = 'a'`).
4. Read one character and print whether it is between `'a'` and `'z'`.
5. Read one character and print the three following characters in sequence.

3.5 A First Calculator Program

So far, you've learned how to use arithmetic operators, assignment, precedence, and comparisons. Now let's combine them into a mini calculator program that reads two numbers from the user and prints the results of all the basic operations.

The Idea

- Input: two integers from the user (`a` and `b`)
- Output: sum, difference, product, integer division, remainder, and floating-point division

This lets you practice:

- `+ - - / %`
- assignment to store results
- formatted output with `printf`

A Full Example

```c
#include <stdio.h>

int main(void) {
    int a, b;

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

    int sum = a + b;
    int diff = a - b;
    int prod = a - b;
    int quot = a / b;          // integer division
    int rem  = a % b;          // remainder
    double fquot = (double)a / b;  // floating-point division

    printf("\n--- Calculator ---\n");
    printf("%d + %d = %d\n", a, b, sum);
    printf("%d - %d = %d\n", a, b, diff);
    printf("%d - %d = %d\n", a, b, prod);
    printf("%d / %d = %d (integer division)\n", a, b, quot);
    printf("%d %% %d = %d (remainder)\n", a, b, rem);
    printf("%d / %d = %.6f (floating division)\n", a, b, fquot);

    return 0;
}
```

Example run:

```
Enter two integers (a b): 11 4

--- Calculator ---
11 + 4 = 15
11 - 4 = 7
11 - 4 = 44
11 / 4 = 2 (integer division)
11 % 4 = 3 (remainder)
11 / 4 = 2.750000 (floating division)
```

Why It Matters

This small calculator shows how different operators behave side by side. You see the difference between integer and floating division, how remainders work, and how assignment stores intermediate results. It also demonstrates how to format outputs clearly.

Exercises

1. Modify the program to also calculate the square and cube of each input.
2. Change the program to read two floating-point numbers and show their sum, difference, product, and division.
3. Add output that shows `a + b - 2` and `(a + b) - 2` to illustrate operator precedence.
4. Extend the program to ask for three integers and print their total and average (as floating).
5. Write a version that only prints the floating-point division result, but with 2, 4, and 8 decimal places.

Problems

1. Arithmetic Basics

Write a program that reads two integers and prints their sum, difference, product, integer division, remainder, and floating-point division.

2. Square and Cube

Read one integer and print its square and cube.

3. Average of Two

Read two integers and print both their integer average and their floating average.

4. Even or Odd (with %)

Read one integer and print the result of `n % 2`. Use this to check even/odd (0 means even, 1 means odd).

5. Precedence Practice

Compute and print the results of:

- `5 + 2 - 3`
- `(5 + 2) - 3`
- `10 - 4 + 2 - 3`
- `(10 - 4 + 2) - 3`

6. Combined Assignment

Start with `int x = 10;`. Then use `+=`, `-=`, `-=`, `/=`, `%=` with another integer and print the result after each step.

7. Equality Test

Read two integers and print the result of `a == b` and `a != b`.

8. Greater Than Check

Read two integers and print the result of `a > b` and `a < b`.

9. Range Test

Read one integer and print whether it is between 1 and 100 using `(n >= 1 && n <= 100)`.

10. Logical OR

Read two integers and print `(a > 10 || b > 10)`.

11. Logical NOT

Read one integer and print both `n > 0` and `!(n > 0)`.

12. Character Info

Read one character and print:

- the character
- its ASCII code
- the next character (`c+1`)
- the previous character (`c-1`)

13. Uppercase or Lowercase

Read one character and print whether it is uppercase (`'A'..'Z'`) or lowercase (`'a'..'z'`) using relational operators.

14. Digit Check

Read one character and print whether it is a digit (`'0'..'9'`).

15. Character Math

Read a letter and print the letter 3 positions ahead (e.g., input `A` $\to$ output `D`).

16. Expression Explorer

Read three integers `a, b, c` and print the result of:

- `a + b - c`
- `(a + b) - c`
- `a - b + c`

17. Increment with +=

Read one integer `n`. Print `n`, then `n += 1`, then `n += 5`.

18. Logical Combination

Read two integers `a, b`. Print the results of:

- `(a > 0 && b > 0)`
- `(a > 0 || b > 0)`
- `!(a > 0 && b > 0)`

19. Mini Character Table

Print the characters `'A'` through `'Z'` with their ASCII codes using a loop-free approach (declare them directly and print with `%c` and `%d`).

20. Mini Calculator

Write a program that reads two integers and prints their sum, difference, product, integer division, remainder, and floating division - formatted like a calculator.

Part II. Building blocks

Chapter 4. Control Flow

4.1 `if` and `else`

Until now, all our programs did the same thing every time, no matter the input. But useful programs often need to make choices:

- If the user is old enough → print "Welcome."
- If a number is negative → print "Error."
- Otherwise → do something else.

In C, decisions are made with the `if` statement.

The `if` Statement

```
if (condition) {
    // statements run only if condition is true
}
```

- `condition` is an expression that evaluates to true (1) or false (0).
- If it's true, the block inside `{}` runs.
- If it's false, the block is skipped.

Example:

```
int age = 20;
if (age >= 18) {
    printf("You are an adult.\n");
}
```

Adding `else`

You can provide an alternative with `else`:

```c
if (age >= 18) {
    printf("You are an adult.\n");
} else {
    printf("You are a minor.\n");
}
```

`if … else if … else`

For multiple choices:

```c
if (score >= 90) {
    printf("Grade A\n");
} else if (score >= 75) {
    printf("Grade B\n");
} else if (score >= 50) {
    printf("Grade C\n");
} else {
    printf("Fail\n");
}
```

The computer checks conditions in order. The first one that's true is executed, and the rest are skipped.

A Full Example

This program reads an integer and classifies it:

```c
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    if (n > 0) {
        printf("%d is positive.\n", n);
    } else if (n < 0) {
        printf("%d is negative.\n", n);
    } else {
        printf("The number is zero.\n");
    }

    return 0;
}
```

Example run:

```
Enter a number: -7
-7 is negative.
```

Why It Matters

`if` and `else` give programs the power to branch - to take different paths depending on conditions. This is the foundation for decision-making, error checking, and interactive behavior in all real-world programs.

Exercises

1. Write a program that reads an integer and prints whether it is even or odd.
2. Write a program that reads two integers and prints which one is larger (or if they are equal).
3. Write a program that reads an exam score (0–100) and prints a letter grade (`A`, `B`, `C`, or `F`).
4. Write a program that reads a temperature in Celsius and prints:
   - `"Cold"` if less than 10
   - `"Warm"` if between 10 and 25
   - `"Hot"` if above 25
5. Write a program that reads a character and prints whether it is uppercase, lowercase, or not a letter.

4.2 `switch` Statements

Sometimes you need to compare the same variable against several constant values. Writing many `if … else if … else` lines can become messy. The `switch` statement gives you a clearer way.

The Structure

```
switch (expression) {
    case value1:
        // code if expression == value1
        break;
    case value2:
        // code if expression == value2
        break;
    ...
    default:
        // code if no case matches
}
```

- The `expression` is usually an integer or character.
- Each `case` is compared against the expression.
- `break;` ends that case so execution does not "fall through" to the next one.
- `default` runs if none of the cases match (like an "else").

Example: Day of the Week

```
#include <stdio.h>

int main(void) {
    int day;
    printf("Enter a number (1-7): ");
    scanf("%d", &day);

    switch (day) {
        case 1: printf("Monday\n"); break;
        case 2: printf("Tuesday\n"); break;
        case 3: printf("Wednesday\n"); break;
        case 4: printf("Thursday\n"); break;
```

```c
        case 5: printf("Friday\n"); break;
        case 6: printf("Saturday\n"); break;
        case 7: printf("Sunday\n"); break;
        default: printf("Invalid day number.\n");
    }

    return 0;
}
```

Example run:

```
Enter a number (1-7): 3
Wednesday
```

Example: Character Menu

Because characters are really small integers, you can use them in `switch`:

```c
#include <stdio.h>

int main(void) {
    char choice;
    printf("Choose (a/b/c): ");
    scanf(" %c", &choice);

    switch (choice) {
        case 'a': printf("You chose A.\n"); break;
        case 'b': printf("You chose B.\n"); break;
        case 'c': printf("You chose C.\n"); break;
        default:  printf("Unknown choice.\n");
    }

    return 0;
}
```

Why It Matters

- `switch` is easier to read when one variable has many fixed options.
- It avoids repetitive `if (x == 1) … else if (x == 2) …`.
- Useful for menus, command selection, and state handling in larger programs.

Exercises

1. Write a program that reads a digit (0–9) and prints its English word (`"zero"`, `"one"`, …).
2. Write a program that reads a grade character (`'A'`, `'B'`, `'C'`, `'D'`, `'F'`) and prints a message (e.g., `"Excellent"`, `"Good"`, `"Pass"`, `"Fail"`).
3. Extend the day-of-week program to also print `"Weekend"` if the day is 6 or 7.
4. Write a simple calculator that reads an operator character (`+`, `-`, `-`, `/`) and two integers, then prints the result. Use `switch` to handle the operator.
5. Write a program that reads a character and prints whether it is a vowel (`a, e, i, o, u`) or consonant.

4.3 `while` Loops

Sometimes you want to do something over and over while a condition is true. For example:

- Keep asking for input until the user types `0`.
- Count from 1 to 10.
- Process characters in a string one by one.

This is what the `while` loop is for.

The Structure

```
while (condition) {
    // statements run repeatedly
}
```

- The `condition` is checked before each iteration.
- If it's true, the loop body runs.
- If it's false, the loop stops.

Example: Counting to 5

```
#include <stdio.h>

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

Output:

```
1
2
3
4
5
```

Example: Summing Numbers

```
#include <stdio.h>

int main(void) {
    int n, sum = 0;
    printf("Enter positive numbers (0 to stop):\n");
    scanf("%d", &n);

    while (n != 0) {
        sum += n;
        scanf("%d", &n);
    }

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

Run:

```
Enter positive numbers (0 to stop):
3
5
7
0
Sum = 15
```

Infinite Loops

If the condition never becomes false, the loop runs forever. Example:

```c
while (1) {
    printf("Looping forever!\n");
}
```

You should almost always include an update step (like `i = i + 1;`) so the loop eventually stops.

Why It Matters

The `while` loop introduces repetition - programs can keep working until a condition changes. This is essential for tasks like input validation, iterative calculations, and data processing.

Exercises

1. Write a program that prints the numbers from 1 to 10 using a `while` loop.
2. Write a program that reads integers until the user enters `0`, then prints their total.
3. Write a program that prints the first 10 even numbers.
4. Write a program that asks for a password (string) and keeps asking until the correct one is entered.
5. Write a program that reads a number `n` and uses a `while` loop to print its multiplication table (from `n - 1` to `n - 10`).

4.4 `for` Loops

The `while` loop is flexible, but sometimes you want to count through a range in a very compact form. For this, C provides the `for` loop.

The Structure

```c
for (initialization; condition; update) {
    // loop body
}
```

- initialization → set the starting value
- condition → loop continues while true

- update → run after each iteration (e.g., increment)

It's just a compact way of writing the pattern:

```
initialization;
while (condition) {
    // body
    update;
}
```

Example: Counting to 5

```
#include <stdio.h>

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

Output:

```
1
2
3
4
5
```

Example: Sum of 1 to N

```
#include <stdio.h>

int main(void) {
    int n, sum = 0;
    printf("Enter n: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        sum += i;
    }

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

Run:

```
Enter n: 5
Sum = 15
```

Example: Multiplication Table

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

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

Nested Loops

A loop inside another loop is called nested. Example: printing a rectangle of - characters:

```
#include <stdio.h>

int main(void) {
    for (int row = 1; row <= 3; row++) {
        for (int col = 1; col <= 5; col++) {
            printf("-");
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
-
-
-
```

Why It Matters

- `for` loops are the standard tool for counting tasks.
- They combine initialization, condition, and update neatly in one line.
- Nested loops let you handle two dimensions (rows and columns).

Exercises

1. Write a program that prints the numbers from 1 to 20 using a `for` loop.
2. Write a program that prints all odd numbers from 1 to 19.
3. Write a program that computes the factorial of `n` (product of 1 × 2 × … × n) using a `for` loop.
4. Write a program that prints a multiplication table from 1 to 10 (all rows and columns).
5. Write a program that prints a right triangle of `-` with `n` rows, where `n` is read from input. Example for `n=4`:

4.5 Breaking and Continuing

Sometimes you don't want to finish a loop normally:

- You may want to stop early when a condition is met.
- Or you may want to skip one iteration and continue with the next.

C gives two keywords for this: `break` and `continue`.

`break`

`break` immediately exits the nearest loop.

Example: stop when `i` reaches 5:

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

```
1
2
3
4
```

The loop ends as soon as `i == 5`.

`continue`

`continue` skips the rest of the current iteration and jumps to the next one.

Example: skip even numbers:

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue;  // skip printing even numbers
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

```
1
3
5
7
9
```

Combining with `while`

`break` and `continue` also work with `while` loops.

Example: reading until user enters `0`:

```c
#include <stdio.h>

int main(void) {
    int n;
    while (1) {          // infinite loop
        scanf("%d", &n);
        if (n == 0) {
            break;       // exit when n is 0
        }
        if (n < 0) {
            continue;    // skip negatives
        }
        printf("You entered: %d\n", n);
    }
    return 0;
}
```

Why It Matters

- `break` lets you exit loops early, useful for searches or stopping when a goal is reached.
- `continue` lets you skip specific cases without leaving the loop.
- They give you finer control inside loops, making programs more efficient and easier to read.

Exercises

1. Write a program that prints numbers from 1 to 20 but stops at 13 using `break`.
2. Write a program that prints numbers from 1 to 20 but skips multiples of 3 using `continue`.
3. Write a program that reads integers until `0` is entered; skip negative numbers, and print only positives.
4. Write a program that searches for the first number divisible by 17 between 1 and 100, then stops.
5. Write a program that prints all letters `A` to `Z` but skips vowels (`A, E, I, O, U`) using `continue`.

Problems

1. Positive, Negative, or Zero

Read an integer and print whether it is positive, negative, or zero.

2. Maximum of Two

Read two integers and print the larger one (or print "Equal" if they are the same).

3. Grading System

Read a score (0–100) and print the grade:

- `A` for 90–100
- `B` for 75–89
- `C` for 50–74
- `F` for below 50

4. Temperature Classifier

Read a Celsius temperature and print:

- "Cold" if < 10
- "Warm" if 10–25
- "Hot" if > 25

5. Calculator with Switch

Read two integers and an operator (`+`, `-`, `-`, `/`) and use a `switch` to perform the calculation.

6. Digit to Word

Read a single digit (0–9) and print its English word (“zero”, “one”, …). Use a `switch`.

7. Vowel or Consonant

Read a character and print whether it is a vowel (`a, e, i, o, u`) or consonant.

8. Count from 1 to 10

Use a `while` loop to print the numbers 1 through 10.

9. Sum Until Zero

Keep reading integers until the user enters `0`. Print their sum.

10. Multiplication Table

Read a number `n` and use a `for` loop to print its multiplication table (from `n × 1` to `n × 10`).

11. Factorial

Read a number `n` and compute its factorial using a `for` loop.

12. Print Odd Numbers

Use a `for` loop with `continue` to print only the odd numbers between 1 and 20.

13. Stop at Thirteen

Use a `for` loop with `break` to print numbers from 1 to 20 but stop at 13.

14. Skip Negatives

Keep reading integers until `0` is entered. Skip negative numbers with `continue` and print only positives.

15. First Divisible by 17

Use a loop to find the first number between 1 and 100 that is divisible by 17, then stop with `break`.

16. Print a Right Triangle

Read `n` and use nested `for` loops to print a right triangle of `-` with `n` rows. Example for `n=4`:

17. Rectangle of Stars

Read `rows` and `cols` and print a rectangle of `-`.

18. Guessing Game

Pick a secret number (hardcode it, e.g., `42`). Use a `while` loop to keep asking the user until they guess it correctly. Print “Too low” or “Too high” for wrong guesses.

19. Countdown

Read an integer `n` and use a `while` loop to count down from `n` to 1, then print “Blast off!”.

20. Prime Check

Read an integer `n` and check if it is prime by testing divisibility in a loop. Print “Prime” or “Not prime.”

Chapter 5. Functions

5.1 Why Functions Matter

As programs grow, putting everything into `main` becomes messy. You end up with one giant block of code: hard to read, hard to change, easy to break.

Functions are how we divide programs into small, clear pieces. Each function has a name and does one job.

Everyday Analogy

Think of a program like a kitchen. Instead of one person doing everything, you split tasks:

- one function washes vegetables,
- another cuts them,
- another boils water,
- another prepares sauce.

The recipe is easier to follow, and each part can be reused whenever needed.

What Functions Give You

- Clarity → code is broken into named steps
- Reuse → write once, use many times
- Testing → check each part independently
- Flexibility → update one function, all callers benefit

A First Example

Here's a tiny function that squares an integer:

```c
#include <stdio.h>

int square(int n) {
    return n - n;
}

int main(void) {
    int x = 7;
    printf("square(%d) = %d\n", x, square(x));
    return 0;
}
```

Notice how `square` gives a clear name to the operation. If you see `square(7)`, you immediately know what it means.

More Than One Function

You can define several functions in the same program. Each does one job, and together they make the program easier to follow:

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
void print_line(void) { printf("----------\n"); }

int main(void) {
    int a = 12, b = 5;
    print_line();
    printf("add = %d\n", add(a, b));
    printf("sub = %d\n", sub(a, b));
    print_line();
    return 0;
}
```

This reads like a story: draw a line, add numbers, subtract numbers, draw a line.

Why It Matters

Functions are the building blocks of bigger programs. They let you:

- tell the computer -what to do- in small, named steps,
- avoid repeating code,
- and make your programs easier for humans to read and understand.

The rest of this chapter will show how to define, call, and organize functions in C.

Exercises

1. Write a function `hello(void)` that prints “Hello, world!” and call it from `main`.
2. Write `int double_it(int n)` that returns twice the value of `n`.
3. Write two functions `line(void)` and `stars(void)` where `line` prints dashes and `stars` prints stars. Call them from `main` to decorate output.
4. Write a function `square(int n)` and use it to print the squares of numbers 1 through 5.
5. Write two functions `add(int a,int b)` and `mul(int a,int b)`. Call them with different inputs and print the results.

5.2 Defining and Calling Functions

Now that you know why functions matter, let’s look at how to write and use them in C.

Function Definition

A function has three parts:

1. Return type - the kind of value it gives back (`int`, `double`, `void`, …)
2. Name - what you call it
3. Parameters - inputs inside parentheses

```
return_type name(parameters) {
    // body
    return value;   // if not void
}
```

Example:

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

Calling a Function

You call a function by writing its name followed by arguments:

```
int result = add(3, 4);
printf("%d\n", result);
```

The values `3` and `4` are arguments; inside the function they are received as parameters (`a`, `b`).

Functions Defined Before `main`

If a function is defined before `main`, the compiler already knows it, so no extra declaration is needed:

```c
#include <stdio.h>

int square(int n) {
    return n - n;
}

int main(void) {
    printf("%d\n", square(7));
    return 0;
}
```

Prototypes and Defining After `main`

If you prefer to put `main` first and functions later, you must give the compiler a prototype before `main`.

A prototype tells the compiler the function's name, return type, and parameter types.

```c
#include <stdio.h>

int square(int n);  // prototype

int main(void) {
    printf("%d\n", square(7));
    return 0;
}

int square(int n) {
    return n - n;
}
```

Without this, modern C (C99 and later) will not compile: every function must be declared or defined before it is used.

Matching Prototypes

The prototype must match the definition:

```c
int add(int a, int b);          // OK
int add(int a, int b) { return a + b; }
```

If the return type or parameters don't match, the compiler warns or errors.

Multiple Functions in One Program

You can organize many functions. Either define them all before `main`, or put prototypes above `main` and definitions after.

```c
#include <stdio.h>

/- Prototypes -/
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);

int main(void) {
    int x = 10, y = 4;
    printf("add: %d\n", add(x, y));
    printf("sub: %d\n", sub(x, y));
    printf("mul: %d\n", mul(x, y));
    return 0;
}

/- Definitions -/
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a - b; }
```

Why It Matters

- The compiler must know a function before you call it.
- You can achieve this by defining the function first, or by writing a prototype.
- This rule becomes more important when we split code into multiple files - prototypes live in headers (`.h`), definitions in source files (`.c`).

Exercises

1. Write `int triple(int n)` that returns $3 \times$ `n`. Define it before `main` and call it with different numbers.
2. Write `int max2(int a, int b)` but put its definition after `main`. Add a prototype before `main`.
3. Write `double area_rectangle(double w, double h)` and `double perimeter_rectangle(double w, double h)`. Call them from `main`.
4. Create a program with functions `add`, `sub`, `mul`, and `div_int` (assume divisor not zero). Place prototypes at the top, `main` next, definitions at the bottom.
5. Modify the previous program: move the prototypes into a separate file called `mathlib.h`, include it with `#include "mathlib.h"`, and keep definitions in your `.c` file.

5.3 Arguments and Return Values

A function is like a machine:

- You feed it inputs (arguments).
- It processes them.
- It may give you an output (return value).

Parameters and Arguments

- Parameters are the variable names inside the function definition.
- Arguments are the actual values you pass in when calling the function.

Example:

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

int main(void) {
    int x = 5, y = 7;
    int result = add(x, y);  // x, y are arguments
    printf("%d\n", result);
    return 0;
}
```

Functions with No Parameters

If a function doesn't need input, you declare it with `void`:

```c
void greet(void) {
    printf("Hello!\n");
}

int main(void) {
    greet();
    return 0;
}
```

Functions with Multiple Parameters

You can pass as many inputs as you like, separated by commas:

```c
int max3(int a, int b, int c) {
    int m = a;
    if (b > m) m = b;
    if (c > m) m = c;
    return m;
}
```

Return Values

The `return` statement sends a value back to the caller:

```c
int square(int n) {
    return n - n;
}
```

- The type of the return value must match the function's declared return type.
- If a function is declared `void`, it should not return a value.

Example: Average Function

```c
#include <stdio.h>

double average3(int a, int b, int c) {
    return (a + b + c) / 3.0;
}

int main(void) {
    printf("Average = %.2f\n", average3(4, 7, 10));
    return 0;
}
```

Output:

```
Average = 7.00
```

Ignoring Return Values

You don't have to store the return value - you can call a function and ignore it:

```c
printf("Result: %d\n", square(6));  // no variable needed
```

Why It Matters

- Arguments let functions take in data.
- Return values let functions produce results.
- Together, they make functions reusable, flexible, and powerful - the building blocks of modular programs.

Exercises

1. Write `int is_even(int n)` that returns `1` if `n` is even, `0` otherwise. Test it with numbers 1–10.
2. Write `double area_circle(double r)` that returns $\cdot r^2$. Call it with several radii.
3. Write `int min2(int a, int b)` that returns the smaller of two integers.
4. Write `double convert_c_to_f(double c)` that converts Celsius to Fahrenheit: `F = C - 9/5 + 32`.
5. Write `void line(int n)` that prints `n` dashes in a row. Call it multiple times with different values of `n`.

5.4 Scope of Variables

When you write a program, you can declare variables in different places. Where a variable is declared determines where it can be used - this is called its scope.

Local Variables

A variable declared inside a function exists only in that function.

```c
#include <stdio.h>

void demo(void) {
    int x = 10;   // local to demo
    printf("x in demo = %d\n", x);
}

int main(void) {
    demo();
    // printf("%d", x);  //  error: x not visible here
    return 0;
}
```

Local variables are created when the function is called, and destroyed when it ends.

Function Parameters Are Local Too

Parameters behave like local variables.

```c
int square(int n) {   // n is local
    return n - n;
}
```

Here `n` exists only while `square` runs.

Global Variables

A variable declared outside all functions is called a global variable. It can be used by all functions in the file.

```c
#include <stdio.h>

int counter = 0;   // global

void increment(void) {
    counter++;
}

int main(void) {
    increment();
    increment();
    printf("counter = %d\n", counter);  // prints 2
    return 0;
}
```

Globals are created when the program starts and live until it ends.

Shadowing

A local variable can have the same name as a global. In that case, the local one hides the global in its scope.

```c
#include <stdio.h>

int value = 100;   // global

void test(void) {
    int value = 50;   // shadows global
    printf("local value = %d\n", value);
}

int main(void) {
    test();
    printf("global value = %d\n", value);
    return 0;
}
```

Output:

```
local value = 50
global value = 100
```

Block Scope

Variables declared inside a block `{ ... }` are visible only inside that block.

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    {
        int y = 20;   // visible only here
        printf("%d %d\n", x, y);
    }
    // printf("%d", y);  //   error: y not visible here
    return 0;
}
```

Why It Matters

- Local variables keep functions independent and safe.
- Globals are shared, but overusing them makes code hard to manage.
- Scope rules prevent name clashes and keep data where it belongs.
- Understanding scope helps avoid bugs where variables "mysteriously" change.

Exercises

1. Write a program with a global counter and a function `tick()` that increments it. Call `tick()` five times and print the result.
2. Write a program where a local variable shadows a global variable with the same name. Print both values.
3. Write a function `int cube(int n)` that uses only local variables. Show that the parameter is local.
4. Declare a variable inside a block `{ ... }` in `main` and print it. Then try printing it outside the block to see the error.
5. Write a program with two functions, each with a local variable of the same name. Show that they do not interfere with each other.

5.5 Writing a Reusable Math Library

By now you know how to define functions, pass arguments, return values, and handle scope. The next step is to organize functions into a library that you can reuse across multiple programs.

Splitting Code into Files

A common C pattern is:

- Header file (`.h`) → contains function prototypes (the "interface")
- Source file (`.c`) → contains function definitions (the "implementation")
- Main program → uses the library by including the header

Example: `mathlib.h`

This file contains only prototypes:

```c
#ifndef MATHLIB_H
#define MATHLIB_H

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div_int(int a, int b);
int square(int n);

#endif
```

The `#ifndef ... #define ... #endif` block is called an -include guard-. It prevents multiple inclusion errors.

Example: `mathlib.c`

This file contains the function definitions:

```c
#include "mathlib.h"

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a - b; }
int div_int(int a, int b) { return a / b; }   // assumes b != 0
int square(int n) { return n - n; }
```

Example: `main.c`

The main program includes the header and uses the library:

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

int main(void) {
    int a = 12, b = 5;

    printf("a = %d, b = %d\n", a, b);
    printf("add = %d\n", add(a, b));
    printf("sub = %d\n", sub(a, b));
    printf("mul = %d\n", mul(a, b));
    printf("div = %d\n", div_int(a, b));
    printf("square(a) = %d\n", square(a));

    return 0;
}
```

Compiling Together

You compile both files and link them:

```
gcc main.c mathlib.c -o program
```

Extending the Library

You can add more functions later, just by:

1. Adding the prototype to `mathlib.h`.
2. Adding the definition to `mathlib.c`.

Any program that includes `mathlib.h` can then use the new function.

Why It Matters

- Organizing functions into headers and source files is the first step toward modular programming.
- It separates -what functions do- (interface) from -how they are implemented- (details).
- This pattern scales from small projects to huge systems.

Exercises

1. Write a small library `shapes.h` and `shapes.c` with functions:

   - `area_rectangle(w,h)`
   - `perimeter_rectangle(w,h)`
   - `area_circle(r)` (use `3.14159`) Use it in `main.c`.

2. Add a function `int max2(int a, int b)` to `mathlib`. Update both the header and the source. Test it in `main.c`.

3. Add a function `int factorial(int n)` to `mathlib`. Demonstrate it in a program that prints factorials of numbers 1–10.

4. Move the `printf` line separator function `print_line(void)` into its own library `util.h` / `util.c`. Include it in `main.c` alongside `mathlib.h`.

5. Combine `mathlib.c` and `util.c` into one program with `main.c`. Compile with:

   ```
   gcc main.c mathlib.c util.c -o program
   ```

Problems

1. Hello Function

Write a function `void hello(void)` that prints `"Hello, world!"`. Call it from `main`.

2. Double Function

Write `int double_it(int n)` that returns twice the input. Test it with several values.

3. Maximum of Two

Write `int max2(int a, int b)` that returns the larger of two integers. Demonstrate it in `main`.

4. Minimum of Three

Write `int min3(int a, int b, int c)` that returns the smallest of three integers.

5. Even Check

Write `int is_even(int n)` that returns `1` if the number is even, `0` otherwise. Print results for numbers 1–10.

6. Average of Three

Write `double average3(int a, int b, int c)` that returns the average. Call it with several sets of numbers.

7. Square Function

Write `int square(int n)` and use it to print squares of numbers from 1 to 10 in a loop.

8. Circle Area

Write `double area_circle(double r)` that returns $\cdot r^2$ (use `3.14159`). Test it with different radii.

9. Celsius to Fahrenheit

Write `double c_to_f(double c)` that converts Celsius to Fahrenheit. Demonstrate with 0, 25, 100.

10. Print Line

Write `void line(int n)` that prints a row of `n` dashes. Call it multiple times with different lengths.

11. Scope Demonstration

Write a program with a global variable `counter` and a function `tick()` that increments it. Call `tick()` five times from `main` and print the result.

12. Shadowing

Write a program with a global variable `value = 100` and a function that declares a local variable `value = 50`. Print both the local and global values.

13. Block Scope

Write a program that declares a variable `y` inside a block `{ ... }`. Print it inside the block, then try printing it outside (observe the compile error).

14. Separate Functions

Write three functions: `add`, `sub`, `mul`. Place their definitions before `main`. Call each with two integers.

15. Prototypes and Later Definitions

Rewrite the previous program so that `main` comes first. Add prototypes above `main` and put definitions after.

16. Rectangle Functions

Write functions `double area_rectangle(double w, double h)` and `double perimeter_rectangle(double w, double h)`. Test them in `main`.

17. Factorial Function

Write `int factorial(int n)` that returns `n!`. Use it to print factorials of numbers 1–10.

18. Function Reuse

Write `int sum_range(int a, int b)` that returns the sum of all integers between `a` and `b` inclusive. Use it in `main` to compute the sum of 1–100.

19. Tiny Math Library

Create two files:

- `mathlib.h` → prototypes for `add`, `sub`, `mul`, `div_int`, `square`
- `mathlib.c` → definitions Write `main.c` that includes `mathlib.h` and uses the functions.

20. Shapes Library

Create a header `shapes.h` and source `shapes.c` with:

- `area_rectangle(w,h)`
- `perimeter_rectangle(w,h)`
- `area_circle(r)` Write `main.c` that uses these functions to print areas and perimeters for different shapes.

Chapter 6. Arrays and Strings

6.1 Introduction to Arrays

So far, we've stored one value per variable: one `int`, one `double`, one `char`. But many tasks need a collection of values.

- A list of exam scores
- The names of players on a team
- The pixels in an image

In C, the simplest way to store such a collection is with an array.

What Is an Array?

An array is a block of memory that holds multiple values of the same type, arranged one after another.

- Each value is called an element.
- You access elements using an index (position number).
- Indexes start at 0 in C.

Declaring an Array

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

This creates space for 5 integers:

- `scores[0]`
- `scores[1]`
- `scores[2]`

- `scores[3]`
- `scores[4]`

Initializing an Array

You can set initial values at once:

```c
int numbers[4] = {10, 20, 30, 40};
```

Or partially:

```c
int numbers[4] = {10, 20};  // others become 0
```

You can also let the compiler count:

```c
int primes[] = {2, 3, 5, 7, 11};
```

Accessing Elements

You use the index in square brackets:

```c
#include <stdio.h>

int main(void) {
    int nums[3] = {4, 7, 9};
    printf("First = %d\n", nums[0]);
    printf("Second = %d\n", nums[1]);
    printf("Third = %d\n", nums[2]);
    return 0;
}
```

Output:

```
First = 4
Second = 7
Third = 9
```

Changing Elements

```c
nums[1] = 42;  // changes the second element
```

Now `nums` is `{4, 42, 9}`.

A Full Example

```c
#include <stdio.h>

int main(void) {
    int scores[5];

    // read 5 scores
    for (int i = 0; i < 5; i++) {
        printf("Enter score %d: ", i + 1);
        scanf("%d", &scores[i]);
    }

    // print them back
    printf("You entered:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", scores[i]);
    }
    printf("\n");

    return 0;
}
```

Why It Matters

Arrays let you store and process many values efficiently without creating dozens of separate variables. They are the foundation for working with strings, data tables, and more complex structures.

Exercises

1. Declare an array of 10 integers and set each element to its index (0, 1, 2, …, 9). Print them.
2. Read 5 integers into an array, then print their sum.

3. Read 5 integers and print the largest.
4. Initialize an array of the first 6 even numbers and print them.
5. Read 10 integers and print them in reverse order.

6.2 Iterating Over Arrays

An array is most powerful when combined with loops. Instead of writing code for each element, you let a loop handle all of them.

Accessing Elements with a Loop

```c
#include <stdio.h>

int main(void) {
    int nums[5] = {10, 20, 30, 40, 50};

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

    return 0;
}
```

Output:

```
nums[0] = 10
nums[1] = 20
nums[2] = 30
nums[3] = 40
nums[4] = 50
```

Reading Into an Array

```c
#include <stdio.h>

int main(void) {
    int arr[5];
    for (int i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);  // note the &
    }

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

    return 0;
}
```

Calculating a Sum

```
#include <stdio.h>

int main(void) {
    int scores[5] = {80, 90, 70, 60, 85};
    int sum = 0;

    for (int i = 0; i < 5; i++) {
        sum += scores[i];
    }

    printf("Total = %d\n", sum);
    return 0;
}
```

Finding a Maximum

```
#include <stdio.h>

int main(void) {
    int scores[5] = {12, 45, 67, 23, 89};
    int max = scores[0];

    for (int i = 1; i < 5; i++) {
        if (scores[i] > max) {
            max = scores[i];
        }
    }

    printf("Max = %d\n", max);
    return 0;
}
```

Using `sizeof` to Get Array Length

Hardcoding the array size (`5` above) works, but we can calculate it:

```
int arr[] = {2, 4, 6, 8, 10};
int len = sizeof(arr) / sizeof(arr[0]);
```

This way, if the array changes size, the loop still works correctly.

Why It Matters

- Loops and arrays go hand in hand.
- You can process data of any length with the same logic.
- This is the foundation for algorithms like searching, sorting, and aggregation.

Exercises

1. Read 10 integers into an array and print their average.
2. Find the minimum value in an array of 8 integers.
3. Count how many numbers in an array of 10 are even.
4. Read 5 integers and print them in reverse order using a loop.
5. Given `int arr[] = {1,2,3,4,5}`, write a program to compute the sum of squares of all elements.

6.3 Strings as Character Arrays

In C, a string is not a special type. It's just an array of characters ending with a special character:

- `'\0'` (null terminator)

This null character marks the end of the string, so functions know where to stop.

Declaring Strings

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

Easier with double quotes:

```c
char word[] = "Hello";
```

The compiler adds the `'\0'` automatically.

Printing Strings

Use `%s` in `printf`:

```c
#include <stdio.h>

int main(void) {
    char msg[] = "C is fun!";
    printf("%s\n", msg);
    return 0;
}
```

Output:

```
C is fun!
```

Reading Strings

```c
#include <stdio.h>

int main(void) {
    char name[20];
    printf("Enter your name: ");
    scanf("%19s", name);   // limit to 19 chars + '\0'
    printf("Hello, %s!\n", name);
    return 0;
}
```

`scanf("%s", ...)` stops at the first space. For reading full lines, safer functions like `fgets` are better (later section).

Character by Character

Strings are just arrays:

```c
#include <stdio.h>

int main(void) {
    char word[] = "Hi";
    printf("%c %c %c\n", word[0], word[1], word[2]);
    return 0;
}
```

Output:

```
H i \0
```

Notice the third element is `'\0'`.

Changing Characters

You can change individual letters:

```c
char greet[] = "Cat";
greet[0] = 'H';  // now "Hat"
```

A Full Example

```c
#include <stdio.h>

int main(void) {
    char city[20];
    printf("Enter a city: ");
    scanf("%19s", city);

    printf("First letter: %c\n", city[0]);

    int i = 0;
    while (city[i] != '\0') {
        i++;
    }
    printf("Length = %d\n", i);

    return 0;
}
```

Why It Matters

- Strings are essential for text processing.
- Understanding that they are arrays helps explain how input, output, and libraries like `<string.h>` work.
- Remember: always leave room for the null terminator!

Exercises

1. Declare a string `"Hello"` and print its characters one by one in a loop.
2. Read a name into a char array and print a greeting.
3. Write a program that counts the number of characters in a string (without using `strlen`).
4. Modify a string `"dog"` into `"fog"` by changing its first character.
5. Read two strings and print them in reverse order (first the second, then the first).

6.4 Standard String Functions

C provides many useful functions for handling strings in the header `<string.h>`. These functions work with arrays of characters that end with `'\0'`.

`strlen` - String Length

Counts the number of characters before the null terminator.

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

int main(void) {
    char word[] = "Hello";
    printf("Length = %zu\n", strlen(word));
    return 0;
}
```

Output:

```
Length = 5
```

`strcpy` - Copy a String

Copies characters from one string into another.

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

int main(void) {
    char src[] = "world";
    char dst[20];
    strcpy(dst, src);
    printf("%s\n", dst);
    return 0;
}
```

Make sure the destination is large enough.

`strcat` - Concatenate Strings

Appends one string to another.

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

int main(void) {
    char a[20] = "Good";
    char b[] = " morning";
    strcat(a, b);
    printf("%s\n", a);
    return 0;
}
```

Output:

```
Good morning
```

`strcmp` - Compare Strings

Compares two strings lexicographically (like dictionary order).

- Returns `0` if equal
- Negative if first < second
- Positive if first > second

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

int main(void) {
    char s1[] = "apple";
    char s2[] = "banana";

    if (strcmp(s1, s2) < 0) {
        printf("%s comes before %s\n", s1, s2);
    }
    return 0;
}
```

Safer Variants

Many libraries provide safer versions:

- `strncpy` (copy with size limit)
- `strncat` (concatenate with size limit)
- `strncmp` (compare up to n chars)

Example:

```c
char src[] = "hello";
char dst[10];
strncpy(dst, src, sizeof(dst)-1);
dst[sizeof(dst)-1] = '\0';   // ensure termination
```

A Full Example

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

int main(void) {
    char a[30] = "Hello";
    char b[] = "World";

    printf("a length = %zu\n", strlen(a));
    strcpy(a, "Hi");
    strcat(a, " there");
    printf("a now = %s\n", a);

    if (strcmp(a, b) == 0)
        printf("a equals b\n");
    else
        printf("a != b\n");

    return 0;
}
```

Why It Matters

- Without these helpers, you'd write long loops to process strings.
- `<string.h>` functions are efficient and standard across all C compilers.
- Learning them prepares you for real-world text processing.

Exercises

1. Read a string and print its length using `strlen`.
2. Copy one string into another and print both.
3. Concatenate `"Hello"` and `"World"` into a buffer and print the result.
4. Read two strings and print which comes first in dictionary order.
5. Use `strncpy` to safely copy `"C programming"` into a buffer of size 8, and print the result.

6.5 Building a Word Counter

Now that we know how to use arrays and string functions, let's build a simple word counter. This program will:

1. Read a line of text.

2. Split it into words.
3. Count how many words there are.

Step 1: Reading a Line

Use `fgets` to safely read a line of input (it includes spaces).

```c
#include <stdio.h>

int main(void) {
    char line[200];
    printf("Enter a sentence:\n");
    fgets(line, sizeof(line), stdin);
    printf("You entered: %s", line);
    return 0;
}
```

Step 2: Splitting Into Words

The `<string.h>` function `strtok` splits a string into tokens (pieces) separated by delimiters.

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

int main(void) {
    char line[200];
    printf("Enter a sentence:\n");
    fgets(line, sizeof(line), stdin);

    int count = 0;
    char -word = strtok(line, " \t\n");  // split by space, tab, newline
    while (word != NULL) {
        printf("Word: %s\n", word);
        count++;
        word = strtok(NULL, " \t\n");
    }

    printf("Total words = %d\n", count);
    return 0;
}
```

Example run:

```
Enter a sentence:
C makes low-level programming fun
Word: C
Word: makes
Word: low-level
Word: programming
Word: fun
Total words = 5
```

Step 3: Counting Word Frequencies (Optional)

We can go further: keep an array of words and their counts.

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

#define MAX_WORDS 50
#define MAX_LEN   20

int main(void) {
    char line[200];
    char words[MAX_WORDS][MAX_LEN];
    int counts[MAX_WORDS] = {0};
    int total = 0;

    printf("Enter a sentence:\n");
    fgets(line, sizeof(line), stdin);

    char -w = strtok(line, " \t\n");
    while (w != NULL && total < MAX_WORDS) {
        int found = 0;
        for (int i = 0; i < total; i++) {
            if (strcmp(words[i], w) == 0) {
                counts[i]++;
                found = 1;
                break;
            }
        }
        if (!found && strlen(w) < MAX_LEN) {
            strcpy(words[total], w);
```

```
                counts[total] = 1;
                total++;
            }
            w = strtok(NULL, " \t\n");
        }

        printf("\nWord frequencies:\n");
        for (int i = 0; i < total; i++) {
            printf("%s : %d\n", words[i], counts[i]);
        }

        return 0;
}
```

Why It Matters

- Combines arrays, strings, and library functions into one real project.
- Shows how to process input text, split it, and analyze it.
- This is a small taste of how text editors, search engines, and compilers start their work.

Exercises

1. Modify the word counter so it ignores case (treat `"C"` and `"c"` as the same).
2. Extend the program to print the longest word in the input.
3. Extend the program to print the average word length.
4. Modify it so it counts only unique words, and prints the total number of distinct words.
5. Write a version that reads from a file instead of user input, and counts words in the whole file.

Problems

1. Index Fill

Declare an array of 10 integers. Use a loop to fill it so that `arr[i] = i`. Print all elements.

2. Array Sum

Read 5 integers into an array and print their sum.

3. Array Maximum

Read 8 integers into an array and print the largest.

4. Reverse Print

Read 10 integers and print them in reverse order using a loop.

5. Even Counter

Read 10 integers into an array and count how many are even.

6. Average of Numbers

Read 10 integers into an array and print their average as a `double`.

7. Sum of Squares

Given `int arr[] = {1,2,3,4,5}`, compute and print the sum of their squares.

8. Print Characters of a String

Declare a string `"Hello"` and print each character on a separate line.

9. Greeting with String

Read a string (name) into a char array and print `"Hello, <name>!"`.

10. Count Characters (Manual)

Read a string and count its length manually (without `strlen`).

11. Modify a String

Declare `char word[] = "dog"` and change it to `"fog"` by modifying one character. Print the result.

12. Two Strings Reverse Order

Read two strings and print them in reverse order (second, then first).

13. Length with `strlen`

Read a string and print its length using `strlen`.

14. String Copy

Read a string into one buffer, copy it into another with `strcpy`, and print both.

15. String Concatenation

Concatenate `"Hello"` and `"World"` into `"HelloWorld"` using `strcat`. Print the result.

16. String Comparison

Read two strings and print which one comes first in dictionary order (use `strcmp`).

17. Safe Copy with `strncpy`

Copy `"C programming"` into a buffer of size 8 using `strncpy`. Ensure the result is null-terminated, then print it.

18. Word Counter (Basic)

Read a sentence with `fgets` and count how many words it has (split by spaces). Print the count.

19. Longest Word

Extend the word counter: print the longest word in the sentence.

20. Word Frequency

Read a line of text and count the frequency of each unique word. Print the results.

Part III. Deeper into C

Chapter 7. Pointers

7.1 What is a Pointer?

In C, a pointer is a variable that stores the *address- of another variable.

Think of it like:

- A normal variable stores a value.
- A pointer stores where that value lives in memory.

Addresses in Memory

Every variable in a program is stored in memory at some location. You can get that location using the address-of operator `&`.

```c
#include <stdio.h>

int main(void) {
    int x = 42;
    printf("x = %d\n", x);
    printf("address of x = %p\n", (void-)&x);
    return 0;
}
```

Example output:

```
x = 42
address of x = 0x7ffee8c48a7c
```

(The exact address will differ.)

Declaring a Pointer

A pointer variable is declared with -:

```
int -p;   // p can hold the address of an int
```

To assign it:

```
int x = 42;
int -p = &x;   // p points to x
```

Dereferencing a Pointer

To get the value stored at the address, use - again (dereference):

```
#include <stdio.h>

int main(void) {
    int x = 42;
    int -p = &x;

    printf("x = %d\n", x);
    printf("p points to %p\n", (void-)p);
    printf("-p = %d\n", -p);   // value at that address
    return 0;
}
```

Output:

```
x = 42
p points to 0x7ffee8c48a7c
-p = 42
```

Changing Through a Pointer

If you change -p, it changes the original variable:

```
int x = 42;
int -p = &x;
-p = 99;        // modifies x
printf("%d\n", x);   // prints 99
```

Pointers and Types

The type of pointer must match the type it points to:

- `int -` $\rightarrow$ points to `int`
- `double -` $\rightarrow$ points to `double`
- `char -` $\rightarrow$ points to `char`

This tells the compiler how to interpret the memory at that address.

Why It Matters

- Pointers let you work directly with memory.
- They are essential for arrays, strings, dynamic memory, and data structures.
- Understanding pointers is the key to mastering C.

Exercises

1. Declare an `int x = 5` and a pointer `p` that points to it. Print both `x` and `-p`.
2. Change `x` by assigning to `-p` instead of `x`. Print the result.
3. Declare two integers `a=10`, `b=20` and pointers `pa`, `pb`. Print their addresses.
4. Write a program that reads an integer into `x` and prints its address.
5. Experiment: declare `double y = 3.14` and `double -py = &y`. Print `y`, its address, and `-py`.

7.2 Pointers and Addresses

In the last section, we saw that a pointer stores the address of another variable. Now let's explore that connection more carefully.

Variables and Their Addresses

Every variable in C has:

1. A value - what you store in it.
2. An address - where it lives in memory.

```c
#include <stdio.h>

int main(void) {
    int a = 123;
    printf("value = %d\n", a);
    printf("address = %p\n", (void-)&a);
    return 0;
}
```

Storing the Address in a Pointer

You can save that address inside a pointer:

```c
int a = 123;
int *pa = &a;   // pa points to a
```

Now:

- `pa` holds the address of `a`.
- `*pa` is another way to refer to the value of `a`.

Visualizing It

Memory (simplified):

```
 a:   123
pa:   &a
```

When you write `*pa`, you are saying "follow pa to where it points" → `123`.

Example: Changing Through a Pointer

```c
#include <stdio.h>

int main(void) {
    int x = 5;
    int *p = &x;
    printf("x = %d\n", x);
    *p = 42;         // modifies x
    printf("x = %d\n", x);
    return 0;
}
```

Output:

```
x = 5
x = 42
```

Multiple Pointers to the Same Variable

More than one pointer can point to the same place:

```
int n = 7;
int *p1 = &n;
int *p2 = &n;

*p1 = 99;   // changes n
printf("%d\n", *p2);  // prints 99
```

Pointer Assignment

Pointers can be reassigned to point to different variables:

```
int a = 10, b = 20;
int *p = &a;
printf("*p = %d\n", *p);  // 10

p = &b;
printf("*p = %d\n", *p);  // 20
```

Why It Matters

- Pointers are *names for addresses-.
- They let functions and data structures share and modify the same memory.
- Understanding addresses is essential for arrays, strings, and dynamic memory.

Exercises

1. Declare an integer `n = 100`, and a pointer `pn`. Print `n`, `&n`, `pn`, and `-pn`.
2. Write a program where two pointers point to the same integer. Modify the value through one pointer and print it through the other.
3. Declare two integers `a` and `b`. Make one pointer point first to `a`, then to `b`, printing values each time.
4. Write a program with three integers and an array of pointers (`int -ptrs[3]`). Make each pointer point to one integer and print their values.
5. Experiment: print the size of an `int -`, `double -`, and `char -` using `sizeof`. Compare the results.

7.3 Arrays and Pointers

In C, an array name often behaves like a pointer. Understanding this connection is key to working with strings, loops, and dynamic memory.

Array Name as an Address

```
#include <stdio.h>

int main(void) {
    int arr[3] = {10, 20, 30};
    printf("arr = %p\n", (void*)arr);
    printf("&arr[0] = %p\n", (void*)&arr[0]);
    return 0;
}
```

Output (addresses match):

```
arr = 0x7ffee2c38930
&arr[0] = 0x7ffee2c38930
```

The name `arr` is treated as the address of its first element.

Accessing with Pointers

You can access array elements with either array indexing (`arr[i]`) or pointer arithmetic (`-(arr + i)`).

```
int arr[3] = {10, 20, 30};
printf("%d\n", arr[1]);       // array indexing
printf("%d\n", -(arr + 1));   // pointer arithmetic
```

Both print 20.

Using a Pointer Variable

```
int arr[3] = {10, 20, 30};
int -p = arr;     // same as &arr[0]

printf("%d\n", -p);     // 10
printf("%d\n", -(p+1)); // 20
printf("%d\n", -(p+2)); // 30
```

Iterating with a Pointer

```
#include <stdio.h>

int main(void) {
    int arr[5] = {2, 4, 6, 8, 10};
    int -p = arr;

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

Arrays Are Not Pointers

Although array names -decay- to pointers in most expressions, they are not the same thing:

- You cannot reassign an array name.
- `sizeof(arr)` gives the full array size, while `sizeof(p)` (a pointer) gives only the size of the pointer type.

```c
int arr[10];
int -p = arr;

printf("%zu\n", sizeof(arr)); // e.g., 40 (10 ints on 64-bit system)
printf("%zu\n", sizeof(p));   // e.g., 8 (pointer size)
```

Strings and Pointers

Strings are arrays of `char`. You can use pointer arithmetic to walk through characters until the null terminator:

```c
#include <stdio.h>

int main(void) {
    char word[] = "Hello";
    char -p = word;

    while (-p != '\0') {
        printf("%c ", -p);
        p++;
    }
    printf("\n");
    return 0;
}
```

Output:

```
H e l l o
```

Why It Matters

- Arrays and pointers are two sides of the same coin in C.
- Pointers give you flexibility in traversing and manipulating arrays.
- This connection is the basis for string handling, dynamic memory, and data structures.

Exercises

1. Declare `int arr[5] = {1,2,3,4,5}`. Print all elements using pointer arithmetic (`-(arr+i)`).
2. Write a function `print_array(int -p, int n)` that prints all elements of an integer array.
3. Read 5 integers into an array, then use a pointer to calculate their sum.
4. Create a string `"C language"` and use a pointer to print each character until `'\0'`.
5. Compare `sizeof(arr)` and `sizeof(p)` where `p` is a pointer to the array. Print both results.

7.4 Pointers to Functions

Just as you can have a pointer to a variable, you can also have a pointer to a function. This lets you:

- Call functions dynamically,
- Pass functions as arguments,
- Build flexible libraries (like sort with a custom comparison).

Function Names as Addresses

The name of a function is its address in memory. So you can assign it to a pointer:

```
#include <stdio.h>

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

int main(void) {
    int (-fp)(int, int) = add;   // fp points to add
    printf("%d\n", fp(3, 4));    // call through pointer
    return 0;
}
```

Output:

```
7
```

Declaring a Function Pointer

Syntax:

```
return_type (-pointer_name)(parameter_types);
```

Example:

```
int (-f)(int, int);   // f is a pointer to a function taking (int,int) and returning int
```

Example: Multiple Functions

```
#include <stdio.h>

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

int main(void) {
    int (-op)(int, int);

    op = add;
    printf("add: %d\n", op(10, 5));

    op = sub;
    printf("sub: %d\n", op(10, 5));

    return 0;
}
```

Passing Function Pointers to Other Functions

```
#include <stdio.h>

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

void compute(int (-f)(int,int), int x, int y) {
    printf("Result = %d\n", f(x,y));
}

int main(void) {
    compute(add, 3, 4);   // pass add
    compute(mul, 3, 4);   // pass mul
    return 0;
}
```

Output:

```
Result = 7
Result = 12
```

Real-World Example: `qsort`

The C standard library function `qsort` uses a function pointer for custom comparison:

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

int cmp_int(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

int main(void) {
    int arr[5] = {4, 2, 5, 1, 3};
    qsort(arr, 5, sizeof(int), cmp_int);

    for (int i = 0; i < 5; i++) printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Output:

```
1 2 3 4 5
```

Why It Matters

- Function pointers enable flexibility - functions become data.
- They are widely used in callbacks, event handlers, sorting, GUIs, and system programming.
- Understanding them opens the door to advanced C patterns.

Exercises

1. Write two functions: `square(int)` and `cube(int)`. Use a function pointer to call each.
2. Write a function `apply(int (-f)(int), int x)` that applies `f` to `x` and prints the result.
3. Write an array of function pointers to basic operations (`add`, `sub`, `mul`, `div_int`) and call each in a loop.
4. Implement a program that takes two numbers and an operator (`+`, `-`, `-`, `/`), then uses function pointers to choose the correct operation.
5. Use `qsort` to sort an array of strings alphabetically. Write a comparison function for strings.

7.5 Safe Pointers in Modern C

Pointers are powerful, but also dangerous. Misusing them can lead to bugs, crashes, or security issues. Modern C (C11–C23) encourages safe pointer practices to reduce risks.

Null Pointers

A pointer that doesn't point anywhere should be set to a null pointer.

```c
#include <stdio.h>

int main(void) {
    int -p = NULL;   // points nowhere
    if (p == NULL) {
        printf("Pointer is null\n");
    }
    return 0;
}
```

`NULL` vs `nullptr` in C23

- In older C, `NULL` is used (defined in `<stddef.h>`).
- C23 introduces `nullptr` (similar to C++), making null checks safer and clearer:

```c
int -p = nullptr;
if (p == nullptr) { /- safe -/ }
```

Dangling Pointers

A pointer becomes dangling if the variable it points to goes out of scope.

```
int *bad_pointer(void) {
    int x = 10;
    return &x;   //   ERROR: x no longer exists after function ends
}
```

Rule: never return a pointer to a local variable.

Wild Pointers

A pointer that is uninitialized may point to random memory:

```
int *p;  //   uninitialized
*p = 42; // undefined behavior
```

Always initialize pointers: either to a valid address or `NULL`/`nullptr`.

Double Free

If you `free` the same memory twice, the program may crash.

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

Solution: after freeing, set `p = NULL` (or `nullptr`).

Pointer Bounds

Pointers don't carry length information. Accessing outside an array is undefined:

```
int arr[3] = {1,2,3};
int *p = arr;
printf("%d\n", *(p+3)); //   out of bounds
```

Always check bounds when iterating.

Safer Practices

1. Initialize pointers - set to `nullptr` or valid address.
2. Check before use - don't dereference a null pointer.
3. Don't return locals - never return a pointer to a local variable.
4. Set freed pointers to null - avoid double free.
5. Use `const` pointers if data should not change:

   ```c
   const char *msg = "Hello"; // prevents accidental modification
   ```

A Full Example

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

int main(void) {
    int *p = nullptr;       // safe initialization
    p = malloc(sizeof(int));
    if (p == nullptr) {     // check allocation
        printf("Memory allocation failed\n");
        return 1;
    }

    *p = 42;
    printf("Value = %d\n", *p);

    free(p);
    p = nullptr;            // avoid dangling pointer

    return 0;
}
```

Why It Matters

- Pointers are the sharpest tool in C: essential but risky.
- Safe usage prevents segmentation faults, memory leaks, and security vulnerabilities.
- C23's `nullptr` makes code clearer and harder to misuse.

Exercises

1. Declare an uninitialized pointer, then fix it by setting it to `nullptr`. Print a check to confirm.
2. Write a function that safely allocates an integer, sets it to 10, prints it, and frees it.
3. Demonstrate a dangling pointer bug by returning the address of a local variable. Then fix it.
4. Allocate an array of 5 integers with `malloc`, set them to 1–5, print them, then free.
5. Experiment: free a pointer twice without resetting it (observe crash/UB). Then fix it by setting to `nullptr`.

Problems

1. Basic Pointer Access

Declare `int x = 10` and a pointer `p` to it. Print `x`, the address of `x`, the value of `p`, and `-p`.

2. Modify Through a Pointer

Start with `int x = 5`. Use a pointer to change its value to 42. Print both `x` and `-p`.

3. Two Pointers, One Variable

Make two pointers point to the same integer. Change the integer through the first pointer, and print it through the second.

4. Pointer Reassignment

Declare two integers `a=10, b=20` and one pointer `p`. First point `p` to `a`, then to `b`, printing values each time.

5. Array with Pointer Arithmetic

Declare `int arr[5] = {1,2,3,4,5}`. Use pointer arithmetic (`-(arr+i)`) to print all elements.

6. Sum with Pointer Traversal

Read 5 integers into an array. Use a pointer to calculate their sum.

7. Print String via Pointer

Declare `char word[] = "Pointers"`. Use a pointer to walk through the string character by character until `'\0'`.

8. Compare Array vs Pointer Sizes

Declare `int arr[10]` and `int -p = arr`. Print `sizeof(arr)` and `sizeof(p)` to see the difference.

9. Function Pointer Basics

Write two functions: `int square(int)` and `int cube(int)`. Declare a function pointer and use it to call each function.

10. Function Pointer as Argument

Write a function `apply(int (-f)(int), int x)` that applies `f` to `x` and prints the result. Test with both `square` and `cube`.

11. Array of Function Pointers

Create an array of function pointers to four functions: `add`, `sub`, `mul`, `div_int`. Call each in a loop with two numbers.

12. Operator with Function Pointer

Write a calculator program: read two integers and an operator (`+ - - /`), then use function pointers to call the correct operation.

13. Safe Null Pointer

Declare an `int -p = nullptr` (or `NULL`). Print a message if the pointer is null.

14. Dangling Pointer Demo

Write a function that returns the address of a local variable (dangling pointer). Call it and print the result to observe the bug. Then fix it by using `malloc`.

15. Wild Pointer Example

Declare an uninitialized pointer and try dereferencing it (expect crash/UB). Then fix it by proper initialization.

16. Double Free Experiment

Allocate an integer with `malloc`, free it twice (expect UB). Then fix it by setting the pointer to `nullptr` after `free`.

17. Safe Dynamic Array

Allocate an array of 5 integers with `malloc`, fill with values 1–5, print them, then `free` safely.

18. Struct with Pointer

Define a `struct Point { int x,y; }`. Create one instance and a pointer to it. Use the pointer with `->` to print and modify values.

19. Passing Pointers to Functions

Write a function `void swap(int *a, int -b)` that swaps two integers using pointers. Demonstrate it in `main`.

20. Function Pointer with `qsort`

Use `qsort` to sort an array of integers. Write a comparison function and pass it as a function pointer.

Chapter 8. Structs and Enums

Here's Chapter 8.1: Grouping Data with `struct`, the first step into C's way of building custom data types.

8.1 Grouping Data with `struct`

So far, we've worked with simple types: `int`, `double`, `char`, arrays, and pointers. But real-world programs often deal with things that have multiple properties.

Example:

- A point has both an -x- and -y- coordinate.
- A student has a *name-,* age-, and *grade-.
- A book has a -title-, *author-, and -year-.

In C, we can bundle related values into one structure using the keyword `struct`.

Declaring a `struct`

```c
struct Point {
    int x;
    int y;
};
```

This defines a new type `struct Point` with two fields: `x` and `y`.

Creating Variables

```c
struct Point p1;
p1.x = 3;
p1.y = 4;
```

You can also initialize at once:

```c
struct Point p2 = {10, 20};
```

Accessing Fields

Use the dot operator (`.`):

```c
#include <stdio.h>

struct Point {
    int x;
    int y;
};

int main(void) {
    struct Point p = {5, 7};
    printf("x=%d y=%d\n", p.x, p.y);
    return 0;
}
```

Output:

```
x=5 y=7
```

Example: Student Record

```c
#include <stdio.h>

struct Student {
    char name[50];
    int age;
    double grade;
};

int main(void) {
    struct Student s = {"Alice", 20, 3.8};
    printf("Name: %s\n", s.name);
    printf("Age: %d\n", s.age);
    printf("Grade: %.2f\n", s.grade);
    return 0;
}
```

Output:

```
Name: Alice
Age: 20
Grade: 3.80
```

Arrays of Structs

You can create an array of structs, just like an array of ints:

```c
struct Point points[3] = {
    {1, 2},
    {3, 4},
    {5, 6}
};
```

Loop through them:

```c
for (int i = 0; i < 3; i++) {
    printf("(%d,%d)\n", points[i].x, points[i].y);
}
```

Why It Matters

- Structures let you represent real-world entities in code.
- They keep related values together, making programs more organized and readable.
- Almost every serious C program uses `struct` for data modeling.

Exercises

1. Define a `struct Point` with fields `x` and `y`. Create one, set values, and print them.
2. Define a `struct Student` with `name`, `age`, and `gpa`. Initialize and print it.
3. Create an array of 3 `struct Points` and print their coordinates.
4. Write a program that reads a student's name, age, and grade into a `struct` and prints them back.
5. Define a `struct Rectangle` with `width` and `height`. Write a function `area` that takes a rectangle and returns its area.

8.2 Using `typedef`

When you declare a `struct`, you normally have to write the keyword `struct` every time:

```c
struct Point {
    int x;
    int y;
};

struct Point p1 = {3, 4};
```

This can get repetitive, especially in large programs. C provides `typedef` to create a shorter alias for a type.

Basic `typedef`

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

Now you can write:

```c
Point p1 = {3, 4};
```

instead of `struct Point p1`.

Example: Cleaner Code

```c
#include <stdio.h>

typedef struct {
    char name[50];
    int age;
    double gpa;
} Student;

int main(void) {
    Student s = {"Alice", 20, 3.8};
    printf("%s, %d years, GPA=%.2f\n", s.name, s.age, s.gpa);
    return 0;
}
```

Notice: no need to say `struct Student`, just `Student`.

Typedef with Pointers

`typedef` is also useful for pointer types:

```c
typedef struct Point {
    int x, y;
} Point;

typedef Point- PointPtr;

int main(void) {
    Point a = {1, 2};
    PointPtr p = &a;
    printf("(%d,%d)\n", p->x, p->y);
    return 0;
}
```

Typedef vs. `struct`

- Without typedef:

  ```c
  struct Point a;
  ```

- With typedef:

  ```c
  Point a;
  ```

Both create the same kind of variable. Typedef just saves typing.

Why It Matters

- Makes code shorter and cleaner.
- Common in APIs and libraries to simplify complex type names.
- Improves readability by giving meaningful names (e.g., `Point`, `Student`, `Matrix`).

Exercises

1. Rewrite the `struct Point` example from 8.1 using `typedef`.
2. Define a `typedef` for `struct Rectangle { int w,h; }` as `Rectangle`. Write a function that takes a `Rectangle` and returns its area.
3. Define a `typedef` for a pointer to `Student`. Create a `Student` and print values using the pointer.

4. Create a `typedef` for `unsigned long long` called `BigInt`. Use it to compute factorial of 10.
5. Experiment: write a `typedef` for a function pointer `int (-Op)(int,int)` and use it for add/subtract functions.

8.3 Enumerations

Sometimes you need a variable that can take one of a small set of related values:

- Days of the week
- Traffic light colors
- Error codes

In C, you can define these with enumerations, using the keyword `enum`.

Declaring an `enum`

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

This defines `RED = 0`, `GREEN = 1`, `BLUE = 2`.

Using an `enum`

```
#include <stdio.h>

enum Color { RED, GREEN, BLUE };

int main(void) {
    enum Color c = GREEN;
    if (c == GREEN) {
        printf("Go!\n");
    }
    return 0;
}
```

Output:

```
Go!
```

Custom Values

You can assign specific integer values:

```c
enum Status {
    OK = 200,
    NOT_FOUND = 404,
    SERVER_ERROR = 500
};
```

Now `OK` = 200, `NOT_FOUND` = 404, etc.

Sequential Values

Unspecified values continue from the last one:

```c
enum Weekday {
    MON = 1,  // start from 1
    TUE,      // 2
    WED,      // 3
    THU, FRI, SAT, SUN
};
```

`typedef` with `enum`

You can combine `typedef` with `enum` to simplify usage:

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

int main(void) {
    Color c = BLUE;
    printf("Color = %d\n", c);
    return 0;
}
```

Example: Traffic Light

```c
#include <stdio.h>

typedef enum {
    RED, YELLOW, GREEN
} TrafficLight;

int main(void) {
    TrafficLight light = RED;

    if (light == RED) {
        printf("Stop\n");
    } else if (light == GREEN) {
        printf("Go\n");
    } else {
        printf("Wait\n");
    }
    return 0;
}
```

Why It Matters

- Enums make code clearer and safer than using raw integers.
- Useful for fixed sets of options.
- Often combined with `switch` statements for clean logic.

Exercises

1. Define an `enum Weekday` with values for Monday–Sunday. Write a program that prints the number for Wednesday.
2. Define an `enum TrafficLight { RED=1, YELLOW=2, GREEN=3 }`. Write a program that prints `"Stop"` for RED, `"Wait"` for YELLOW, `"Go"` for GREEN.
3. Use `typedef enum` to define `Color` with values {RED, GREEN, BLUE}. Declare a variable and print its integer value.
4. Create an `enum ErrorCode { OK=0, FAIL=1, TIMEOUT=2 }` and use a `switch` to print an error message for each code.
5. Extend the `Weekday` enum so that MON=1, and write a loop to print all days with their numbers.

8.4 `union` (and when to use it)

A `union` is like a `struct`, but instead of giving each member its own storage, all members share the same memory location.

- At any moment, only one member holds a valid value.
- The size of a union is the size of its largest member.

Declaring a Union

```c
union Number {
    int i;
    float f;
};
```

This defines a type `union Number` with two possible views of the same memory.

Example: One at a Time

```c
#include <stdio.h>

union Number {
    int i;
    float f;
};

int main(void) {
    union Number n;
    n.i = 42;
    printf("i = %d\n", n.i);

    n.f = 3.14f;          // overwrites same memory
    printf("f = %.2f\n", n.f);
    printf("i (garbled) = %d\n", n.i); // old value no longer valid
    return 0;
}
```

Output:

```
i = 42
f = 3.14
i (garbled) = some unpredictable value
```

Memory Layout

```
union Number:
+-----------------+
| same memory     |  <-- interpreted as int or float
+-----------------+
```

Practical Use: Variant Data

Unions are useful when a value could be one of several types.

Example: storing a number that might be `int`, `float`, or `double`.

```c
#include <stdio.h>

typedef union {
    int i;
    float f;
    double d;
} Value;

int main(void) {
    Value v;
    v.d = 12.34;
    printf("double = %.2f\n", v.d);
    return 0;
}
```

With `enum` + `struct`

To track which type is active, combine `union` with an `enum`:

```c
#include <stdio.h>

typedef enum { INT, FLOAT } Tag;

typedef struct {
    Tag type;
    union {
        int i;
        float f;
    } data;
} Variant;

int main(void) {
    Variant v;
    v.type = INT;
    v.data.i = 42;

    if (v.type == INT) {
        printf("int=%d\n", v.data.i);
    } else {
        printf("float=%.2f\n", v.data.f);
    }
    return 0;
}
```

Why It Matters

- `union` saves memory by reusing space.
- It's the foundation of variant types and low-level data handling.
- Useful when interfacing with hardware, binary files, or protocols where the same data may mean different things.

Exercises

1. Define a `union Data` with `int i` and `float f`. Assign and print each in turn.
2. Write a program that stores a value in a union as `int`, then as `float`, and prints both. Observe how the second overwrites the first.
3. Combine `enum` and `union` to represent a number that may be either `int` or `float`. Print it safely depending on the tag.
4. Create a union with `char c[4]` and `int n`. Assign to `n` and then print each `c[i]` (observe byte-level representation).
5. Define a `union Value { int i; double d; }`. Write a program that sets the double value and prints the size of the union using `sizeof`.

8.5 A Simple Contact Book

We’ve learned how to group data with `struct`, simplify with `typedef`, and handle arrays. Now let’s put it all together in a small project: a contact book program.

Defining the Contact Structure

```
#include <stdio.h>

typedef struct {
    char name[50];
    char phone[20];
    int age;
} Contact;
```

Each `Contact` stores a name, phone number, and age.

Creating an Array of Contacts

We’ll store multiple contacts in an array:

```
#define MAX_CONTACTS 5
Contact contacts[MAX_CONTACTS];
int count = 0;  // how many we've added
```

Adding a Contact

```
void add_contact(Contact list[], int *count, const char *name, const char -phone, int age) {
    if (*count < MAX_CONTACTS) {
        snprintf(list[*count].name, sizeof(list[*count].name), "%s", name);
        snprintf(list[*count].phone, sizeof(list[*count].phone), "%s", phone);
        list[*count].age = age;
        (*count)++;
    }
}
```

Printing Contacts

```c
void print_contacts(const Contact list[], int count) {
    for (int i = 0; i < count; i++) {
        printf("%s, %s, %d years\n", list[i].name, list[i].phone, list[i].age);
    }
}
```

Full Example

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

typedef struct {
    char name[50];
    char phone[20];
    int age;
} Contact;

#define MAX_CONTACTS 5

void add_contact(Contact list[], int *count, const char *name, const char -phone, int age) {
    if (*count < MAX_CONTACTS) {
        snprintf(list[*count].name, sizeof(list[*count].name), "%s", name);
        snprintf(list[*count].phone, sizeof(list[*count].phone), "%s", phone);
        list[*count].age = age;
        (*count)++;
    }
}

void print_contacts(const Contact list[], int count) {
    for (int i = 0; i < count; i++) {
        printf("%s, %s, %d years\n", list[i].name, list[i].phone, list[i].age);
    }
}

int main(void) {
    Contact contacts[MAX_CONTACTS];
    int count = 0;
    add_contact(contacts, &count, "Alice", "123-4567", 20);
    add_contact(contacts, &count, "Bob", "555-9876", 25);

    print_contacts(contacts, count);

    return 0;
}
```

Output:

```
Alice, 123-4567, 20 years
Bob, 555-9876, 25 years
```

Why It Matters

- Shows how structs can model real-world objects.
- Demonstrates `typedef`, arrays, and function reuse.
- This pattern (define a type → store in array → process with functions) is the basis of larger applications like databases and address books.

Exercises

1. Add a function `find_contact(Contact list[], int count, const char *name)` that searches for a contact by name.
2. Add a function `delete_contact(Contact list[], int *count, const char *name)` that removes a contact.
3. Extend `Contact` with an `email` field. Update `add_contact` and `print_contacts`.
4. Increase `MAX_CONTACTS` to 100 and let the program read contacts from user input instead of hardcoding.
5. Save the contacts to a text file using `fprintf` and reload them with `fscanf`.

Problems

1. Basic Point Struct

Define a `struct Point` with `x` and `y` fields. Create one, set values, and print them.

2. Student Record

Define a `struct Student` with `name`, `age`, and `gpa`. Initialize one student and print their details.

3. Array of Structs

Create an array of 3 `struct Point`s and print their coordinates.

4. Input into Struct

Write a program that reads a student's `name`, `age`, and `gpa` from the user into a struct and prints them back.

5. Rectangle Area

Define a `struct Rectangle` with `width` and `height`. Write a function that takes a `Rectangle` and returns its area.

6. Using `typedef` for Point

Rewrite the `struct Point` example using `typedef` so you can declare variables as `Point p` instead of `struct Point p`.

7. Typedef with Pointers

Define a `typedef` for `struct Student` as `Student`, and another typedef `StudentPtr` for a pointer to `Student`. Create a student and print values through the pointer.

8. Typedef Alias for Primitive

Create a `typedef` called `BigInt` for `unsigned long long`. Use it to compute factorial of 10.

9. Enum for Weekdays

Define an `enum Weekday` with values for Monday–Sunday (starting at 1). Print the integer value for Wednesday.

10. Enum for Traffic Light

Define an `enum TrafficLight { RED=1, YELLOW=2, GREEN=3 }`. Write a program that prints `"Stop"`, `"Wait"`, or `"Go"` depending on the enum value.

11. Enum with Switch

Define an `enum ErrorCode { OK=0, FAIL=1, TIMEOUT=2 }` and use a `switch` to print a message for each code.

12. Enum Loop

Use the `Weekday` enum from problem 9. Write a loop to print all days with their numeric values.

13. Basic Union

Define a `union Data` with `int i` and `float f`. Assign and print each in turn. Observe how one overwrites the other.

14. Union with Char Array

Create a union with `char c[4]` and `int n`. Assign to `n` and then print each `c[i]` (observe byte-level representation).

15. Tagged Union

Combine `enum` and `union` into a `Variant` type that may hold either an `int` or a `float`. Write a program that sets and prints both kinds safely depending on the tag.

16. Contact Struct Array

Define a `struct Contact { char name[50]; char phone[20]; int age; }`. Create an array of 3 contacts, initialize them, and print all.

17. Contact Functions

Write a function `add_contact` that inserts a new contact into an array, and `print_contacts` that prints them. Test with 2–3 contacts.

18. Find Contact by Name

Extend problem 17 with a function `find_contact` that searches for a contact by name and prints their details.

19. Delete Contact

Extend problem 18 with a function `delete_contact` that removes a contact by shifting later ones down. Print the updated list.

20. Save and Load Contacts

Write a program that saves contacts from an array to a text file with `fprintf` and reloads them with `fscanf`.

Chapter 9. Memory Management

9.1 Automatic vs. Dynamic Memory

Every program needs memory to store variables, arrays, and data. In C, there are two main categories of memory you'll work with:

1. Automatic memory (stack)
2. Dynamic memory (heap)

Understanding the difference is essential for writing reliable and efficient C programs.

Automatic Memory (Stack)

- Variables declared inside a function (without `malloc`) are automatic.
- Their lifetime is tied to the function call.
- They are created when the function begins, and destroyed when it ends.

Example:

```c
#include <stdio.h>

void hello(void) {
    int x = 42;  // automatic variable
    printf("x = %d\n", x);
} // x no longer exists here

int main(void) {
    hello();
    // printf("%d", x);  //   error: x is out of scope
    return 0;
}
```

Automatic variables are usually stored on the stack, a region of memory managed by the compiler.

Dynamic Memory (Heap)

- Allocated at runtime with functions like `malloc` and `free`.
- Lifetime is controlled by the programmer (until explicitly freed).
- Useful when you don't know the needed size at compile time.

Example:

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

int main(void) {
    int -p = malloc(sizeof(int));  // dynamic allocation
    if (p == NULL) return 1;       // always check
    -p = 42;
    printf("-p = %d\n", -p);
    free(p);                       // must free after use
    return 0;
}
```

Comparing the Two

Aspect	Automatic (stack)	Dynamic (heap)
Lifetime	Ends when function ends	Until `free` is called
Allocation	Compiler-managed	Programmer-managed (`malloc`)
Speed	Very fast	Slower
Size	Usually small, fixed	Potentially large, flexible

Example: Fixed vs. Flexible Arrays

Automatic array:

```c
int arr[100];  // size fixed at compile time
```

Dynamic array:

```c
int n;
scanf("%d", &n);
int *arr = malloc(n - sizeof(int));  // size chosen at runtime
```

Why It Matters

- Automatic memory is simple, but limited to function lifetimes and fixed sizes.
- Dynamic memory gives flexibility, but requires discipline: every `malloc` should have a matching `free`.
- Choosing the right kind of memory is a fundamental design decision in C programs.

Exercises

1. Write a function that declares an automatic variable, assigns it a value, and prints it. Call the function twice - what do you observe?
2. Allocate an `int` dynamically, assign it `99`, print it, then free it.
3. Write a program that asks the user for `n`, allocates an array of `n` integers dynamically, fills it with 1..n, and prints them.
4. Experiment: try to use a pointer to an automatic variable after the function returns. Why is this unsafe?
5. Compare the sizes of a stack array (`int a[1000];`) and a dynamically allocated array (`malloc(1000-sizeof(int))`). Use `sizeof` and print results.

9.2 `malloc`, `calloc`, `free`

C gives you explicit control over memory allocation. The three most important functions are:

- `malloc` - allocate memory
- `calloc` - allocate and clear memory
- `free` - release memory

All are declared in `<stdlib.h>`.

`malloc` - Allocate Memory

```
#include <stdlib.h>
void -malloc(size_t size);
```

- Allocates a block of memory of given size (in bytes).
- Returns a pointer to the beginning of the block, or `NULL` if it fails.
- Contents of the memory are uninitialized (garbage).

Example:

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

int main(void) {
    int -p = malloc(5 - sizeof(int));
    if (!p) return 1;  // check allocation
    for (int i = 0; i < 5; i++) p[i] = i+1;
    for (int i = 0; i < 5; i++) printf("%d ", p[i]);
    printf("\n");
    free(p);
    return 0;
}
```

`calloc` - Allocate and Clear

```
#include <stdlib.h>
void -calloc(size_t n, size_t size);
```

- Allocates memory for an array of `n` elements, each of `size` bytes.

- Initializes all bits to zero.

Example:

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

int main(void) {
    int -p = calloc(5, sizeof(int));
    if (!p) return 1;
    for (int i = 0; i < 5; i++) printf("%d ", p[i]); // all zero
    printf("\n");
    free(p);
    return 0;
}
```

Output:

```
0 0 0 0 0
```

`free` - Release Memory

```c
#include <stdlib.h>
void free(void -ptr);
```

- Releases a block of memory previously allocated by `malloc` or `calloc`.
- Does nothing if `ptr == NULL`.
- Accessing memory after `free` is undefined behavior.

Example:

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

int main(void) {
    int -p = malloc(3 - sizeof(int));
    if (!p) return 1;
    p[0]=1; p[1]=2; p[2]=3;
    free(p);
    // p is now dangling - should set to NULL
    p = NULL;
    return 0;
}
```

Common Mistakes

Forgetting to call `free` → memory leak Using memory after `free` → dangling pointer Freeing twice → double free error

Always follow the pattern:

```
p = malloc(...);
if (!p) { /- handle error -/ }
...
free(p);
p = NULL;
```

Why It Matters

- `malloc` and `calloc` let programs create data structures that grow and shrink at runtime.
- `free` ensures memory is returned to the system.
- Correct memory management is crucial in C - there is no garbage collector.

Exercises

1. Use `malloc` to allocate space for an array of 10 integers, fill with squares of 1..10, and print them.
2. Repeat using `calloc` and observe the difference in initialization.
3. Allocate an array of `double` of size entered by the user, set all to `3.14`, and print.
4. Write a program that allocates memory, frees it, then tries to use it again (observe error). Fix it by setting pointer to `NULL`.
5. Create a dynamic 2D array (array of `int-`) with 3 rows and 4 columns using `malloc`. Fill it with numbers and print as a matrix.

9.3 Pointer Pitfalls (and how to avoid them)

Pointers give you power over memory - but with great power comes great responsibility. Misusing pointers often leads to undefined behavior, crashes, or security bugs. Here are the most common pitfalls and how to avoid them.

1. Dangling Pointers

A dangling pointer points to memory that is no longer valid.

Example: returning a pointer to a local variable:

```
int- bad(void) {
    int x = 42;
    return &x;   //   invalid after function ends
}
```

Fix: Allocate dynamically or pass results back by value.

```
int- good(void) {
    int -p = malloc(sizeof(int));
    if (p) -p = 42;
    return p;   // caller must free
}
```

2. Memory Leaks

A memory leak happens when you lose all references to allocated memory without freeing it.

```
int -p = malloc(100 - sizeof(int));
p = NULL;  //   leak: cannot free the memory anymore
```

Fix: Always `free` before overwriting pointers.

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

3. Double Free

Freeing the same memory twice can corrupt the heap.

```
int -p = malloc(sizeof(int));
free(p);
free(p);   //   undefined behavior
```

Fix: After `free`, set pointer to `NULL`. Freeing a null pointer is safe.

4. Using After Free

Accessing memory after it's been freed:

```c
int -p = malloc(sizeof(int));
-p = 7;
free(p);
printf("%d\n", -p); //   use*after-free
```

Fix: Never dereference freed pointers. Set them to `NULL` to prevent accidental use.

5. Out-of-Bounds Access

Arrays don't track their size. Accessing outside is undefined.

```c
int a[5] = {1,2,3,4,5};
printf("%d\n", a[5]);  //   out of bounds
```

Fix: Always check indices against array bounds.

6. Uninitialized Pointers

A pointer not assigned an address may point anywhere.

```c
int -p;   // uninitialized
-p = 5;   //   random crash
```

Fix: Initialize pointers to `NULL` or valid memory.

Checklist for Safe Pointers

- ☒ Initialize all pointers.
- ☒ Check return value of `malloc`/`calloc`.
- ☒ `free` every allocation, exactly once.
- ☒ Set pointers to `NULL` after `free`.
- ☒ Always check bounds when indexing arrays.

Example: Safe Allocation Pattern

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

int main(void) {
    int n = 5;
    int *a = malloc(n - sizeof(int));
    if (!a) return 1;   // check for failure

    for (int i = 0; i < n; i++) a[i] = i - i;
    for (int i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");

    free(a);
    a = NULL;           // prevent dangling
    return 0;
}
```

Why It Matters

- Pointer mistakes are a leading cause of bugs in C programs.
- Undefined behavior may look harmless in small tests, but break programs later.
- Safe habits (null checks, freeing, bounds checking) make C code much more robust.

Exercises

1. Write a function that returns a pointer to a local variable. Run it and observe. Then fix it with `malloc`.
2. Create a program that allocates memory but forgets to `free`. Use a loop to make the leak visible. Then fix it.
3. Demonstrate a double free and then fix it with `p = NULL`.
4. Write a program that reads `n`, allocates an array of size `n`, and prints it. Add a check for out-of-bounds access.
5. Experiment: declare an uninitialized pointer and dereference it. Then fix by initializing with `NULL` and checking before use.

9.4 Safer Allocations in C23 (`aligned_alloc`, `nullptr`)

C23 introduced improvements that make memory management less error-prone. Two key features are:

1. `aligned_alloc` - request memory with a specific alignment
2. `nullptr` - a new null pointer constant

`aligned_alloc`

Normal `malloc` returns memory suitable for any type, but sometimes you need aligned memory (e.g., for SIMD instructions, hardware buffers).

Prototype (in `<stdlib.h>`):

```c
void *aligned_alloc(size_t alignment, size_t size);
```

- `alignment` must be a power of two (e.g., 16, 32).
- `size` must be a multiple of `alignment`.
- Returns `NULL` on failure.

Example: Allocate 32-byte aligned array

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

int main(void) {
    size_t n = 8;
    int *a = aligned_alloc(32, n - sizeof(int));
    if (!a) return 1;
    for (size_t i = 0; i < n; i++) a[i] = i;
    for (size_t i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

`nullptr`

Traditionally, C used `NULL` (defined as `((void-)0)` or `0`) to represent a null pointer. This could be confusing because `0` is also an integer.

C23 introduces `nullptr` as a dedicated null pointer constant.

```c
#include <stdio.h>

int main(void) {
    int -p = nullptr;  // safe null initialization
    if (p == nullptr) {
        printf("Pointer is null\n");
    }
    return 0;
}
```

Why `nullptr` is safer:

- No confusion with integers.
- Makes code more readable.
- Aligns C with modern C++ style.

Safer Practices in C23

- Prefer `nullptr` instead of `NULL`.
- Use `aligned_alloc` when alignment matters (e.g., vectorized math, GPU buffers).
- Always check for allocation failure:

  ```c
  if (p == nullptr) { /- handle error -/ }
  ```

Why It Matters

- `aligned_alloc` enables performance optimizations with aligned memory.
- `nullptr` avoids common bugs with `NULL` being treated as integer `0`.
- Together, they represent a step toward safer, clearer C code.

Exercises

1. Allocate a dynamic array of 16 `ints` using `aligned_alloc(16, …)`. Fill and print it.
2. Write a program that initializes a pointer with `nullptr` and checks it before use.
3. Compare `p = NULL;` and `p = nullptr;` in a simple program. Print the result of `(p == 0)`.
4. Allocate a block of 64 bytes with alignment 32. Print its address to confirm divisibility by 32.
5. Write a program that combines both: initialize a pointer with `nullptr`, then allocate aligned memory and use it.

9.5 A Mini Dynamic Array Implementation

Dynamic arrays are one of the most common data structures. In C, there is no built-in "vector" type like in C++ - but we can build one using pointers, `malloc`, `realloc`, and `free`.

Step 1: Define a Struct for the Array

We need to track:

- A pointer to the data
- The number of elements (`size`)
- The capacity (allocated space)

```
#include <stddef.h>  // for size_t

typedef struct {
    int -data;
    size_t size;
    size_t capacity;
} DynArray;
```

Step 2: Initialize

```
#include <stdlib.h>

void init_array(DynArray *a, size_t initial_capacity) {
    a->data = malloc(initial_capacity - sizeof(int));
    a->size = 0;
```

```c
    a->capacity = (a->data ? initial_capacity : 0);
}
```

Step 3: Add Elements (Resize if Full)

```c
#include <string.h>

int push_back(DynArray *a, int value) {
    if (a->size == a->capacity) {
        size_t new_capacity = (a->capacity == 0) ? 1 : a->capacity - 2;
        int -new_data = realloc(a->data, new_capacity - sizeof(int));
        if (!new_data) return 0;  // fail
        a->data = new_data;
        a->capacity = new_capacity;
    }
    a->data[a->size++] = value;
    return 1; // success
}
```

Step 4: Free the Array

```c
void free_array(DynArray *a) {
    free(a->data);
    a->data = NULL;
    a->size = 0;
    a->capacity = 0;
}
```

Step 5: Example Program

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

typedef struct {
    int -data;
    size_t size;
    size_t capacity;
```

```
} DynArray;

void init_array(DynArray *a, size_t initial_capacity) {
    a->data = malloc(initial_capacity - sizeof(int));
    a->size = 0;
    a->capacity = (a->data ? initial_capacity : 0);
}

int push_back(DynArray *a, int value) {
    if (a->size == a->capacity) {
        size_t new_capacity = (a->capacity == 0) ? 1 : a->capacity - 2;
        int -new_data = realloc(a->data, new_capacity - sizeof(int));
        if (!new_data) return 0;
        a->data = new_data;
        a->capacity = new_capacity;
    }
    a->data[a->size++] = value;
    return 1;
}

void free_array(DynArray *a) {
    free(a->data);
    a->data = NULL;
    a->size = 0;
    a->capacity = 0;
}

int main(void) {
    DynArray arr;
    init_array(&arr, 2);

    for (int i = 1; i <= 10; i++) {
        push_back(&arr, i - i);
    }

    for (size_t i = 0; i < arr.size; i++) {
        printf("%d ", arr.data[i]);
    }
    printf("\n");

    free_array(&arr);
    return 0;
```

```
}
```

Output:

```
1 4 9 16 25 36 49 64 81 100
```

Why It Matters

- Shows how to build a resizable container in C.
- Demonstrates safe use of `malloc`, `realloc`, and `free`.
- This pattern underlies real libraries like glib dynamic arrays or C++ vectors.

Exercises

1. Extend the `DynArray` with a function `get(DynArray *a, size_t index)` that safely returns an element (or error).
2. Write a function `pop_back` that removes the last element.
3. Modify `push_back` so that it shrinks the array when too empty (optional).
4. Store `double` instead of `int`. What changes?
5. Write a small program that reads numbers from the user until EOF and stores them in a `DynArray`. Print them back.

Problems

1. Automatic Variable Lifetime

Write a function that declares an automatic variable, assigns it a value, and prints it. Call the function twice. What do you observe?

2. Pointer to Automatic Variable

Write a function that returns a pointer to a local variable. Use it in `main`. What happens? Fix it with `malloc`.

3. Simple `malloc` Allocation

Use `malloc` to allocate space for an integer, assign it `42`, print it, and then free the memory.

4. Array with `malloc`

Ask the user for `n`, allocate an array of `n` integers with `malloc`, fill with 1..n, and print them. Free the memory afterward.

5. Array with `calloc`

Repeat problem 4, but use `calloc`. Print the values right after allocation to show they are initialized to zero.

6. Compare `malloc` vs. `calloc`

Allocate the same array with `malloc` and `calloc`. Print the contents before writing anything. What's the difference?

7. Freeing and Reusing Memory

Allocate memory for an integer, free it, then try to use it again. What happens? Fix it by setting the pointer to `NULL`.

8. Double Free Error

Write a program that frees the same pointer twice. Observe the behavior (may crash). Then fix it.

9. Memory Leak Demo

Write a loop that calls `malloc` repeatedly without calling `free`. Watch memory usage grow. Then fix it by freeing properly.

10. Out-of-Bounds Access

Allocate an array of size 5 with `malloc`. Try to access index 5 (6th element). Observe the result. Then fix with proper bounds checking.

11. Safe Allocation Pattern

Write a function that allocates an array with `malloc`, checks for `NULL`, fills it, and returns it to the caller. Free it in `main`.

12. Aligned Allocation

Use `aligned_alloc` to allocate an array of 16 integers aligned to 32 bytes. Print the address and confirm divisibility by 32.

13. `nullptr` vs. `NULL`

Write a program that declares one pointer with `NULL` and another with `nullptr`. Print both and compare.

14. Safe Null Checks

Initialize a pointer with `nullptr`, check it before dereferencing, then assign memory with `malloc` and use it safely.

15. Shrinking with `realloc`

Allocate an array of 10 integers with `malloc`. Fill with 1..10. Then shrink it to size 5 with `realloc`. Print results.

16. Growing with `realloc`

Allocate an array of 5 integers. Fill with 1..5. Then grow it to size 10 with `realloc`. Fill the new elements with squares of 6..10 and print all.

17. Memory Fragmentation

Allocate and free blocks of different sizes in a loop. Observe if allocations succeed. Discuss fragmentation risk.

18. Mini Dynamic Array - Push Back

Implement a dynamic array with a `push_back` function that doubles capacity when full. Test with 10 numbers.

19. Mini Dynamic Array - Pop Back

Extend problem 18 with a `pop_back` function that removes the last element. Print results after each operation.

20. Mini Dynamic Array - Generalize

Modify the dynamic array to store `double` instead of `int`. Test by pushing 10 floating-point numbers.

Part IV. Working with the Real World

Chapter 10. Files

10.1 Reading and Writing Files

Programs often need to save data (to a file) and load it later. C does this through the standard I/O library `<stdio.h>` using a `FILE -` handle.

Opening a File

Use `fopen(path, mode)` to get a `FILE-`. Check for `NULL` (open failed).

Common text modes:

- `"r"` → read (file must exist)
- `"w"` → write (creates/truncates)
- `"a"` → append (creates if missing)
- Add `+` for read/write (e.g., `"r+"`, `"w+"`, `"a+"`)

```
FILE -f = fopen("data.txt", "r");
if (f == NULL) { /- handle error -/ }
```

Close with:

```
fclose(f);
```

Writing Text

`fprintf` works like `printf`, but to a file. `fputs` writes a string, `fputc` a character.

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("scores.txt", "w");
    if (!f) { perror("open"); return 1; }

    fprintf(f, "Alice %d\n", 95);
    fprintf(f, "Bob %d\n",   82);
    fputs("Carol 88\n", f);

    if (fclose(f) == EOF) { perror("close"); }
    return 0;
}
```

Reading Text

`fscanf` parses formatted text. `fgets` reads a whole line (including spaces) into a buffer.

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("scores.txt", "r");
    if (!f) { perror("open"); return 1; }

    char name[64];
    int score;

    while (fscanf(f, "%63s %d", name, &score) == 2) {
        printf("%s -> %d\n", name, score);
    }

    fclose(f);
    return 0;
}
```

Reading full lines with `fgets`:

```c
char line[128];
while (fgets(line, sizeof line, f)) {
    printf("line: %s", line);  // line already has '\n' if present
}
```

Checking Errors and EOF

Most file functions signal problems:

- Return `NULL`, `EOF`, or a short count.
- Use `feof(f)` (end-of-file) and `ferror(f)` (error happened).
- `perror("msg")` prints a human-readable error based on `errno`.

```
if (ferror(f)) { perror("read error"); }
```

A Full Example: Copy Lines With Numbers

This program:

1. Writes a few lines to a file.
2. Reopens it to read and prepend line numbers, writing to a second file.

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

int main(void) {
    // 1) Write sample file
    {
        FILE -out = fopen("poem.txt", "w");
        if (!out) { perror("open poem.txt for write"); return 1; }
        fputs("Roses are red\n", out);
        fputs("Violets are blue\n", out);
        fputs("C is powerful\n", out);
        fputs("And speedy too\n", out);
        if (fclose(out) == EOF) { perror("close poem.txt"); return 1; }
    }

    // 2) Read poem.txt, write numbered_poem.txt
    FILE -in  = fopen("poem.txt", "r");
    if (!in) { perror("open poem.txt for read"); return 1; }

    FILE -out = fopen("numbered_poem.txt", "w");
    if (!out) { perror("open numbered_poem.txt"); fclose(in); return 1; }

    char line[256];
    int n = 1;
```

```c
    while (fgets(line, sizeof line, in)) {
        fprintf(out, "%02d: %s", n++, line); // line already ends with '\n' (usually)
    }

    if (ferror(in))  { perror("read poem.txt"); }
    if (fclose(in) == EOF)  { perror("close poem.txt"); }
    if (fclose(out) == EOF) { perror("close numbered_poem.txt"); }

    puts("Wrote numbered_poem.txt");
    return 0;
}
```

Tips & Gotchas

- Always check `fopen` for `NULL`.
- Always close files with `fclose`.
- When parsing with `fscanf`, check the return value (how many items read).
- Prefer `fgets` + manual parsing if lines may contain spaces or complicated formats.
- Text vs. binary differences (line endings, encoding) are covered in 10.2.

Why It Matters

File I/O lets your programs persist data, process logs, import/export information, and communicate with other tools. It's a core skill in C.

Exercises

1. Create `numbers.txt` with the integers 1–10, each on its own line (use `fprintf`).
2. Read `numbers.txt`, compute the sum, and print it to stdout.
3. Read lines from `stdin` with `fgets` and write only lines longer than 10 characters to `long.txt`.
4. Make a program that copies a text file to another file, preserving content exactly (use `fgets`/`fputs`).
5. Read a file containing `name score` pairs and print the highest score and the name that achieved it.

10.2 Text vs. Binary Files

Files in C can be opened in text mode or binary mode. The difference is how the data is interpreted and possibly transformed by the operating system.

Text Mode

- Default mode if you use `"r"`, `"w"`, or `"a"`.
- Data is stored as human-readable characters.
- On Windows, `\n` is converted to `\r\n` when writing, and reversed when reading.
- Portable for structured text: logs, CSV, configuration files.

Example:

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("hello.txt", "w");
    if (!f) return 1;
    fprintf(f, "Line1\nLine2\n");
    fclose(f);
    return 0;
}
```

Binary Mode

- Use `"rb"`, `"wb"`, `"ab"` (append `b` for binary).
- Data is stored exactly as bytes in memory.
- No translation of line endings.
- Useful for images, audio, compiled data, or any non-text format.

Example:

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("data.bin", "wb");
    if (!f) return 1;

    int nums[3] = {10, 20, 30};
    fwrite(nums, sizeof(int), 3, f); // write raw bytes
    fclose(f);
    return 0;
}
```

Reading back:

```c
#include <stdio.h>

int main(void) {
    FILE -f = fopen("data.bin", "rb");
    if (!f) return 1;

    int nums[3];
    fread(nums, sizeof(int), 3, f);
    fclose(f);

    for (int i = 0; i < 3; i++) printf("%d ", nums[i]);
    printf("\n");
    return 0;
}
```

When to Use Text vs. Binary

- Text mode: when humans will read or edit the file.
- Binary mode: when performance matters, or you need to store raw data compactly.
- Same data may take more space in text mode (numbers as characters) than binary.

Exercises

1. Write numbers 1–10 to a file in text mode using `fprintf`. Open the file in a text editor and inspect.
2. Write the same numbers to a binary file using `fwrite`. Open the file in a text editor and compare.
3. Create a struct with two fields (`id`, `score`) and write an array of 3 structs to a binary file.
4. Read the binary file from exercise 3 back and print the values.
5. Measure the size of a text file storing 1000 integers and compare it to the binary version.

10.3 Error Handling in File Operations

When working with files, many things can go wrong:

- The file might not exist.
- The disk might be full.
- A read or write may fail.

C provides mechanisms in `<stdio.h>` to detect and handle these conditions.

Return Values

Most file functions signal errors through return values:

- `fopen` → returns `NULL` if the file cannot be opened.
- `fclose` → returns `EOF` on failure.
- `fread` / `fwrite` → return the number of items actually read or written (less than requested on error or end-of-file).
- `fprintf` / `fputs` / `fputc` → return a negative value on failure.

Always check these return values.

Checking Error State

Two functions let you query the file stream:

- `feof(FILE *f)` → nonzero if end-of-file was reached.
- `ferror(FILE *f)` → nonzero if an error occurred.

Resetting errors:

```
clearerr(f);
```

Using `perror` and `errno`

Many library calls set the global variable `errno` on error. Use `perror("msg")` to print a descriptive error message.

Example:

```
#include <stdio.h>

int main(void) {
    FILE *f = fopen("nofile.txt", "r");
    if (!f) {
        perror("open failed");
        return 1;
    }
    fclose(f);
    return 0;
}
```

Output might look like:

```
open failed: No such file or directory
```

Example: Robust File Copy

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

int main(void) {
    FILE -in = fopen("source.txt", "r");
    if (!in) { perror("open source.txt"); return 1; }

    FILE -out = fopen("dest.txt", "w");
    if (!out) { perror("open dest.txt"); fclose(in); return 1; }

    char buf[256];
    size_t n;
    while ((n = fread(buf, 1, sizeof buf, in)) > 0) {
        if (fwrite(buf, 1, n, out) != n) {
            perror("write error");
            fclose(in);
            fclose(out);
            return 1;
        }
    }

    if (ferror(in)) perror("read error");

    if (fclose(in) == EOF) perror("close in");
    if (fclose(out) == EOF) perror("close out");

    return 0;
}
```

Why It Matters

Ignoring return values can lead to silent data loss. Robust programs check errors and handle them gracefully, printing helpful messages for debugging or recovery.

Exercises

1. Attempt to open a non-existent file with `fopen("missing.txt", "r")`. Detect the error with `perror`.
2. Write a program that reads integers from a file with `fscanf`. Stop correctly on end-of-file, not on error.
3. Modify the file copy program to detect and report both read and write errors.
4. Use `clearerr` to reset an error state on a file and continue reading.
5. Experiment: open a file in read-only mode and try to write to it. Capture the error with `ferror`.

10.4 Building a Simple Log Writer

A common task in programs is writing logs: recording events, errors, or data points to a file. This section shows how to build a minimal log writer in C.

Opening a Log File

A log usually grows over time, so use append mode (`"a"`) to add new entries without overwriting old ones.

```c
#include <stdio.h>

int main(void) {
    FILE -log = fopen("app.log", "a");
    if (!log) { perror("open log"); return 1; }
    fprintf(log, "Program started\n");
    fclose(log);
    return 0;
}
```

Adding Timestamps

Logs are more useful with timestamps. Use `<time.h>` to get the current time.

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

void write_log(const char -msg) {
    FILE -log = fopen("app.log", "a");
    if (!log) return;

    time_t now = time(NULL);
    char -ts = ctime(&now);      // returns string with '\n'
    ts[24] = '\0';               // remove newline

    fprintf(log, "[%s] %s\n", ts, msg);
    fclose(log);
}
```

Usage:

```
int main(void) {
    write_log("Application started");
    write_log("An event occurred");
    return 0;
}
```

Flushing the Log

If the program might crash or be interrupted, flush data immediately after writing.

```
fprintf(log, "Critical error\n");
fflush(log);
```

`fflush` forces buffered output to be written to the file.

Log Levels

Use levels (`INFO`, `WARN`, `ERROR`) for clarity.

```
void log_msg(const char -level, const char -msg) {
    FILE -log = fopen("app.log", "a");
    if (!log) return;

    time_t now = time(NULL);
    char -ts = ctime(&now); ts[24] = '\0';

    fprintf(log, "[%s] %s: %s\n", ts, level, msg);
    fclose(log);
}
```

Example:

```
log_msg("INFO", "Application started");
log_msg("ERROR", "Could not open file");
```

Why It Matters

- Logging helps diagnose problems after a program runs.
- Appending ensures past events are preserved.
- Timestamps and levels make logs readable and actionable.

Exercises

1. Write a `log_info(const char *msg)` function that logs with `"INFO"`.
2. Extend with `log_warn` and `log_error`.
3. Write a loop that logs 5 numbered messages. Open the log afterward in a text editor to inspect.
4. Modify the logging function to also print to `stdout` while writing to the file.
5. Add a command-line argument: if `--clear` is passed, start by truncating the log file (`"w"` mode), otherwise append.

10.5 File APIs in C23

C23 did not radically change file I/O, but it brought some cleanups and clarifications that make file handling safer and clearer.

`nullptr` with File Pointers

Traditionally, `fopen` returned `NULL` on failure. In C23, you can also use the new null constant `nullptr` for comparisons, which avoids confusion with integers.

```
#include <stdio.h>

int main(void) {
    FILE *f = fopen("missing.txt", "r");
    if (f == nullptr) {   // C23 style
        perror("open failed");
        return 1;
    }
```

```c
    fclose(f);
    return 0;
}
```

`fopen_s` (Optional, Annex K)

Some platforms provide safer alternatives like `fopen_s`, which require explicit buffer sizes or return codes. These are optional in the C standard library. When available:

```c
FILE *f;
if (fopen_s(&f, "data.txt", "r") != 0) {
    perror("fopen_s failed");
}
```

But note: Annex K functions are not universally supported and may not be portable.

`tmpfile` and `tmpnam`

- `tmpfile()` creates a temporary file that is automatically removed when closed.
- Useful for scratch space where the file does not need to persist.

```c
#include <stdio.h>

int main(void) {
    FILE *t = tmpfile();
    if (!t) { perror("tmpfile"); return 1; }
    fputs("temporary content\n", t);
    rewind(t);  // move back to start
    char buf[64];
    fgets(buf, sizeof buf, t);
    printf("read: %s", buf);
    fclose(t);
    return 0;
}
```

`freopen` for Redirection

`freopen` can reassign `stdin`, `stdout`, or `stderr` to a file.

```c
#include <stdio.h>

int main(void) {
    freopen("out.txt", "w", stdout);
    printf("This goes into out.txt\n");
    return 0;
}
```

Wide Character File I/O

C23 clarifies wide character I/O functions (`fwprintf`, `fwscanf`, etc.) for Unicode text handling. These are useful for programs that must support international text.

Why It Matters

- `nullptr` makes code cleaner and less error-prone.
- `tmpfile` and `freopen` are powerful but often overlooked parts of `<stdio.h>`.
- Awareness of Annex K APIs helps, but portability should guide choices.

Exercises

1. Write a program that opens a file using `fopen`, checks with `nullptr`, and prints `"ok"` if successful.
2. Use `tmpfile` to create a scratch file, write `"hello"`, rewind, and read it back.
3. Redirect `stdout` to a file using `freopen` and confirm output goes there.
4. Try writing and reading wide characters with `fwprintf` and `fwscanf`.
5. Research whether your compiler supports `fopen_s`. Try it if available.

Here's a unified problem set for Chapter 10 (Files). These exercises cover basic file I/O, text vs. binary handling, error detection, logging, and modern C23 file APIs.

Problems

1. Hello File

Write a program that creates `hello.txt` and writes `"Hello, File!"` into it.

2. Write Numbers

Open `numbers.txt` for writing. Write integers 1–10, one per line.

3. Read Numbers and Sum

Read the numbers back from `numbers.txt`, compute their sum, and print it.

4. Copy Text File

Write a program that copies the contents of `source.txt` into `dest.txt` line by line using `fgets` and `fputs`.

5. Long Line Filter

Read lines from `input.txt` and write only those longer than 10 characters to `long.txt`.

6. Text vs. Binary (Write)

Write integers 1–5 to `ints.txt` using `fprintf`. Then write them to `ints.bin` using `fwrite`.

7. Text vs. Binary (Read)

Read the numbers back from both `ints.txt` and `ints.bin`, print them, and confirm they match.

8. Struct to Binary

Define a struct `{ int id; double score; }`. Write an array of 3 structs to `scores.bin` with `fwrite`.

9. Read Structs Back

Read `scores.bin` with `fread` and print the values.

10. File Error Check

Attempt to open a missing file with `fopen("nofile.txt","r")`. Detect the failure with `perror`.

11. Robust Copy with Error Handling

Write a file copy program that reads from `source.txt` and writes to `dest.txt`, checking for read/write errors and reporting them with `perror`.

12. Log Writer

Write a `log_msg(const char -level, const char -msg)` function that appends entries to `app.log` with timestamps.

13. Multiple Log Levels

Add helpers `log_info`, `log_warn`, and `log_error` that call `log_msg` with the appropriate level.

14. Logging Loop

Write a loop that logs 5 numbered `"INFO"` messages to `app.log`.

15. Log Flush

Modify `log_msg` to `fflush` after each write, ensuring messages appear in the log immediately.

16. Clear vs. Append

Modify the logging program so that if run with `--clear`, it truncates `app.log` (open in `"w"` mode). Otherwise, it appends.

17. Use `tmpfile`

Write a program that creates a temporary file with `tmpfile()`, writes `"scratch data"`, rewinds, reads, and prints it.

18. Use `freopen`

Redirect `stdout` to `out.txt` with `freopen` and write `"redirected output"`.

19. Wide Character I/O

Write wide characters to `wide.txt` using `fwprintf`, then read them back with `fwscanf`.

20. `nullptr` in File Handling (C23)

Write a program that tries to open a non-existent file. Compare the file pointer against `nullptr` instead of `NULL`, and report the error with `perror`.

Chapter 11. Modular Programming

11.1 Splitting Code into Multiple Files

Large programs are easier to manage when you separate interfaces (what is provided) from implementations (how it works). In C, this usually means:

- Header files (`.h`) with -declarations- (function prototypes, type definitions, constants).
- Source files (`.c`) with -definitions- (function bodies, private helpers).
- A main file that uses the interface.

Translation Units and Declarations vs. Definitions

- A translation unit is one `.c` file after preprocessing (`#include` expanded).
- Declaration tells the compiler a symbol exists (type, parameters). Example (prototype): `int add(int a, int b);`
- Definition provides the actual storage or function body. Example (function):

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

You declare in headers so -multiple- `.c` files can see the interface, and define in exactly one `.c` file.

Include Guards

Headers must prevent double inclusion:

```
#ifndef MATHX_H
#define MATHX_H

int add(int a, int b);
int mul(int a, int b);

#endif // MATHX_H
```

(Alternatively: `#pragma once` on compilers that support it.)

A Minimal 3-File Example

Goal: split a tiny math library from `main.c`.

`mathx.h` - the interface (declarations)

```
#ifndef MATHX_H
#define MATHX_H

// Function prototypes (declarations)
int add(int a, int b);
int mul(int a, int b);

#endif
```

`mathx.c` - the implementation (definitions)

```
#include "mathx.h"

// Definitions (bodies)
int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a - b; }
```

`main.c` - user of the library

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

int main(void) {
    printf("add(3,4) = %d\n", add(3,4));
    printf("mul(3,4) = %d\n", mul(3,4));
    return 0;
}
```

Compile & link:

```
gcc -std=c23 -Wall -Wextra -c mathx.c    # compile to mathx.o
gcc -std=c23 -Wall -Wextra -c main.c     # compile to main.o
gcc main.o mathx.o -o app                # link to executable
# or one-liner:
gcc -std=c23 -Wall -Wextra main.c mathx.c -o app
```

Run:

```
./app
add(3,4) = 7
mul(3,4) = 12
```

Visibility and "Private" Helpers

Functions only used inside one `.c` file can be marked `static` to give them internal linkage (not visible to other translation units):

```c
// inside mathx.c
static int clamp16(int x) {
    if (x < -32768) return -32768;
    if (x >  32767) return  32767;
    return x;
}
```

Do not place such helpers in the header unless they're intended for public use.

Common Pitfalls

- Multiple definitions: putting a function body in a header and including it in multiple `.c` files causes duplicate symbol errors. Keep -definitions- in exactly one `.c`.
- Missing prototypes: calling a function without a visible prototype can cause wrong calling conventions or implicit declarations*always include the header.
- Mismatched signatures: prototype and definition must match exactly (return type, parameters).

Folder Layout and Includes

A tidy layout helps:

```
/project
  src/
    main.c
    mathx.c
  include/
    mathx.h
```

Compile with an include path:

```
gcc -Iinclude -std=c23 -Wall -Wextra src/main.c src/mathx.c -o app
```

(Optional) Tiny Makefile

```
CC=gcc
CFLAGS=-std=c23 -Wall -Wextra -Iinclude
OBJ=build/main.o build/mathx.o

app: $(OBJ)
    $(CC) $(OBJ) -o app

build/main.o: src/main.c include/mathx.h
    mkdir -p build
    $(CC) $(CFLAGS) -c src/main.c -o $@

build/mathx.o: src/mathx.c include/mathx.h
    mkdir -p build
    $(CC) $(CFLAGS) -c src/mathx.c -o $@

clean:
    rm -rf build app
```

Why This Matters

- Encourages modularity and reuse.
- Speeds up builds (only changed `.c` recompiles).
- Makes interfaces clear, reduces coupling, and prevents multiple-definition errors.

Exercises

1. Split a small math library (`add`, `sub`, `mul`, `div_int`) into `calc.h`, `calc.c`, and `main.c`.
2. Add a `static` helper in `calc.c` that is not exposed publicly.
3. Introduce a signature mismatch intentionally, observe the compiler error, then fix it.
4. Move headers to `include/` and sources to `src/`, compile with `-Iinclude`.
5. Convert your shell commands into a minimal Makefile with separate compile/link steps.

11.2 Header Files (`.h`) and Declarations

In C, header files (`.h`) provide a way to share declarations across multiple source files. They act as the "contract" between modules: what functions, constants, and types are available.

What Goes in a Header?

A header typically contains:

- Function prototypes (declarations, not definitions).
- Type definitions (`typedef`, `struct`, `enum`).
- Constants and macros (`#define` or `const`).
- Extern variables (rare, used sparingly).

Example `mathx.h`:

```c
#ifndef MATHX_H
#define MATHX_H

// Function declarations
int add(int a, int b);
int mul(int a, int b);

// Type definition
typedef struct {
    int x, y;
} Point;

// Constant
#define PI 3.14159

#endif
```

Why Use Headers?

- Prevents copy-pasting declarations into multiple files.
- Ensures consistency: prototype in `.h` must match definition in `.c`.
- Allows separation of interface (header) from implementation (`.c`).
- Makes large projects manageable by defining clear module boundaries.

Include Guards

Always protect headers against double inclusion:

```c
#ifndef HEADER_NAME_H
#define HEADER_NAME_H
// ... contents ...
#endif
```

Or modern alternative (supported by most compilers):

```c
#pragma once
```

How to Use a Header

1. Place declarations in `foo.h`.
2. Place definitions in `foo.c`.
3. In `main.c` (or another module), include the header:

```c
#include "foo.h"
```

This tells the compiler what functions/types exist, even though the code lives in another file.

Example: Library with Header

`mathx.h`:

```c
#ifndef MATHX_H
#define MATHX_H

int add(int a, int b);
int mul(int a, int b);

#endif
```

`mathx.c`:

```c
#include "mathx.h"

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

`main.c`:

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

int main(void) {
    printf("%d\n", add(2,3));
    printf("%d\n", mul(2,3));
    return 0;
}
```

Compile:

```
gcc -std=c23 main.c mathx.c -o app
```

Common Mistakes

- Placing function definitions in headers: If the header is included in multiple `.c` files, this causes multiple-definition errors. Keep -definitions- in `.c`, -declarations- in `.h`.
- Forgetting include guards: Leads to redefinition errors.
- Using `#include` on `.c` files: Bad practice - include headers, compile `.c` separately.

Why It Matters

Headers are the backbone of modular programming in C. They let teams share interfaces without exposing implementation details, leading to clearer, more maintainable code.

Exercises

1. Write `shapes.h` declaring functions `area_rect(int w, int h)` and `area_circle(int r)`.
2. Implement `shapes.c` with the definitions.
3. Write `main.c` that uses `shapes.h` and prints areas.
4. Add include guards to `shapes.h` and test by including it twice in `main.c`.
5. Create a macro `#define SQUARE(x) ((x)-(x))` in `shapes.h` and use it in `main.c`.

Do you want me to keep going with 11.3 The Preprocessor (`#include`, `#define`) next?

11.3 The Preprocessor (`#include`, `#define`)

Before the compiler processes your C source code, the preprocessor runs. It handles directives that start with `#`, such as `#include` and `#define`. Understanding the preprocessor is essential for managing modular code and constants.

`#include`

Tells the preprocessor to insert the contents of another file.

- System headers (use angle brackets):

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

  The compiler searches system include paths.

- User headers (use quotes):

  ```
  #include "mathx.h"
  ```

  The compiler searches the current directory first, then system paths.

This is how you bring in function prototypes, macros, and type definitions from headers.

`#define`

Defines symbolic constants or macros. The preprocessor replaces occurrences of the symbol before compilation.

Constants:

```
#define PI 3.14159
#define MAX_SIZE 100
```

Macros with parameters:

```
#define SQUARE(x) ((x)-(x))
```

Expands inline, no function call overhead. Be careful with parentheses to avoid precedence bugs.

Example:

```
printf("%d\n", SQUARE(3+1));  // expands to ((3+1)-(3+1)) = 16
```

Conditional Compilation

Directives can include or exclude code:

```
#ifdef DEBUG
    printf("Debug info\n");
#endif
```

```
#if defined(WIN32)
    printf("Running on Windows\n");
#elif defined(__linux__)
    printf("Running on Linux\n");
#else
    printf("Unknown OS\n");
#endif
```

#undef

Removes a macro definition:

```
#undef PI
```

Example: Using Preprocessor Features

`config.h`:

```
#ifndef CONFIG_H
#define CONFIG_H

#define VERSION "1.0"
#define DEBUG 1

#endif
```

`main.c`:

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

#define SQUARE(x) ((x)-(x))

int main(void) {
    printf("Version: %s\n", VERSION);

#if DEBUG
    printf("Debug mode on\n");
#endif

    printf("Square of 5: %d\n", SQUARE(5));
    return 0;
}
```

Compile and run:

```
Version: 1.0
Debug mode on
Square of 5: 25
```

Why It Matters

- `#include` keeps code modular by reusing declarations from headers.
- `#define` creates symbolic names and macros for readability and performance.
- Conditional compilation allows platform-specific or debug code without duplicating files.

Exercises

1. Create a header `config.h` with `#define APP_NAME "MyApp"` and print it in `main.c`.
2. Define a macro `CUBE(x)` that computes the cube of `x`. Test with `CUBE(3+1)`.
3. Use `#ifdef` to include extra debugging output only when `DEBUG` is defined.
4. Write a program that prints `"Windows"` if compiled on Windows (`_WIN32` defined) and `"Linux"` if compiled on Linux (`__linux__` defined).
5. Experiment with `#undef` by defining a constant, undefining it, and checking compilation behavior.

11.4 Macros vs. Inline Functions

Both macros (`#define`) and inline functions (introduced in C99, refined in later standards) can be used to avoid repeated code and eliminate function call overhead. But they differ in safety, type-checking, and readability.

Macros

A macro is expanded by the preprocessor before compilation.

```c
#define SQUARE(x) ((x)-(x))
```

Usage:

```c
printf("%d\n", SQUARE(3));    // expands to ((3)-(3)) = 9
printf("%d\n", SQUARE(3+1));  // expands to ((3+1)-(3+1)) = 16
```

Pros:

- Very fast, just text substitution.
- Can be used for constants, inline code, conditional compilation.

Cons:

- No type checking.

- Can cause subtle bugs if arguments have side effects:

```
int i = 2;
printf("%d\n", SQUARE(i++));  // expands to ((i++)-(i++)), increments twice!
```

Inline Functions

An `inline` function is a real function, but the compiler may expand it directly into code at the call site.

```
inline int square(int x) {
    return x - x;
}
```

Usage:

```
printf("%d\n", square(3));
printf("%d\n", square(3+1));
```

Pros:

- Type checking is enforced.
- Safe from multiple evaluation bugs.
- Debuggable, works like a normal function.

Cons:

- Slightly more verbose.
- The compiler may decide not to inline it (but usually does for small functions).

Constants: `#define` vs `const`

Old style:

```
#define PI 3.14159
```

Modern style:

```
const double PI = 3.14159;
```

- `const` has a type, checked by the compiler.
- `#define` is just text substitution.

Example: Macro vs Inline

```c
#include <stdio.h>

#define CUBE_MACRO(x) ((x)-(x)-(x))

inline int cube_func(int x) {
    return x - x - x;
}

int main(void) {
    int a = 3;
    printf("macro: %d\n", CUBE_MACRO(a));
    printf("func:  %d\n", cube_func(a));

    int i = 2;
    printf("macro side effect: %d\n", CUBE_MACRO(i++)); // unsafe
    // printf("func side effect: %d\n", cube_func(i++)); // safe, only one increment
    return 0;
}
```

Output:

```
macro: 27
func:  27
macro side effect: 60
```

Guidelines

- Prefer `const` variables over `#define` for constants.
- Prefer `inline functions` over macros for computations.
- Use macros only for conditional compilation or cases where inline functions are not possible (e.g., include guards).

Exercises

1. Write a macro `ABS(x)` that computes the absolute value. Test it with negative and positive numbers.
2. Write an inline function `abs_inline(int x)` that does the same. Compare outputs.
3. Experiment with `ABS(i++)` vs `abs_inline(i++)`. What difference do you see?

4. Replace `#define PI 3.14159` with `const double PI = 3.14159;`. Try using it in `printf`.
5. Benchmark (simple loop) calling a macro cube and an inline cube 1 million times - do you see performance differences? ### 11.5 Building a Small Project

By this point, you've learned how to split code into files, use headers, and control compilation with the preprocessor. Let's put everything together into a small modular project.

Project: A Simple Calculator Library

We'll build a calculator with four operations (`add`, `sub`, `mul`, `div_int`), place it into a library (`calc.c` + `calc.h`), and use it from a `main.c`.

`calc.h` - header file

```
#ifndef CALC_H
#define CALC_H

// Function prototypes
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div_int(int a, int b); // integer division, b must not be 0

#endif
```

`calc.c` - implementation

```
#include "calc.h"

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a - b; }
int div_int(int a, int b) { return (b != 0) ? a / b : 0; }
```

`main.c` - program entry point

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

int main(void) {
    int x = 12, y = 4;
    printf("%d + %d = %d\n", x, y, add(x, y));
    printf("%d - %d = %d\n", x, y, sub(x, y));
    printf("%d - %d = %d\n", x, y, mul(x, y));
    printf("%d / %d = %d\n", x, y, div_int(x, y));
    return 0;
}
```

Compiling and Linking

Compile each `.c` separately and then link:

```bash
gcc -std=c23 -Wall -Wextra -c calc.c   # produce calc.o
gcc -std=c23 -Wall -Wextra -c main.c   # produce main.o
gcc main.o calc.o -o app               # link to executable
```

Run:

```
12 + 4 = 16
12 - 4 = 8
12 - 4 = 48
12 / 4 = 3
```

Organizing the Project

A good folder structure:

```
/project
  include/
    calc.h
  src/
    calc.c
    main.c
```

Compile with an include path:

```
gcc -Iinclude -std=c23 src/main.c src/calc.c -o app
```

Example Makefile

```
CC=gcc
CFLAGS=-std=c23 -Wall -Wextra -Iinclude
OBJ=build/main.o build/calc.o

app: $(OBJ)
    $(CC) $(OBJ) -o app

build/main.o: src/main.c include/calc.h
    mkdir -p build
    $(CC) $(CFLAGS) -c src/main.c -o $@

build/calc.o: src/calc.c include/calc.h
    mkdir -p build
    $(CC) $(CFLAGS) -c src/calc.c -o $@

clean:
    rm -rf build app
```

Why It Matters

- Shows how real C projects are organized.
- Demonstrates the role of headers as contracts and source files as implementations.
- Teaches separation of concerns: math code is reusable, `main` is just orchestration.

Exercises

1. Extend the calculator with a `mod(int a, int b)` function.
2. Add `pow_int(int base, int exp)` that computes powers with a loop.
3. Update the Makefile so it automatically recompiles when `calc.h` changes.
4. Move calculator code into a static library (`libcalc.a`) and link it.
5. Try compiling the project with `clang` or MSVC - confirm portability. Here's a comprehensive problem set for Chapter 11 (Modular Programming). These exercises cover splitting code into multiple files, using headers, preprocessor usage, macros vs inline, and building small projects.

Problems

1. Split a Math Library

Write a program that computes `add`, `sub`, `mul`, and `div_int`. Place declarations in `mathlib.h`, definitions in `mathlib.c`, and usage in `main.c`.

2. Include Guards

Modify `mathlib.h` to use an include guard. Test by including `#include "mathlib.h"` twice in `main.c`. Confirm no errors.

3. Static Helper

Add a `static` helper function in `mathlib.c` (e.g., `clamp16(int x)`) that is not visible outside the file. Use it inside `div_int` to prevent division overflow.

4. Conditional Debug Output

Use `#ifdef DEBUG` in `mathlib.c` so that when compiled with `-DDEBUG`, the library prints `"debug: add called"`. Otherwise, it stays silent.

5. Constant with `#define`

In `mathlib.h`, define `#define PI 3.14159` and use it in `main.c` to compute the area of a circle. Then replace it with `const double PI`.

6. Macro vs Inline Function

Write a macro `SQUARE(x)` and an inline function `square(int x)`. In `main.c`, test them with `3`, `3+1`, and `i++`. Compare results.

7. Platform-Specific Code

Use `#if defined(_WIN32)` and `#elif defined(__linux__)` in `main.c` to print `"Windows"` or `"Linux"` depending on platform.

8. Redefining with `#undef`

Define a macro `LIMIT 100`, print it, then `#undef LIMIT` and try printing again. Observe compilation behavior.

9. Simple Header + Implementation

Create `shapes.h` with `area_rect(int w, int h)` and `shapes.c` with its definition. Use it in `main.c` to print `area_rect(3,4)`.

10. Enum in Header

Declare an `enum Color { RED, GREEN, BLUE };` in `shapes.h`. In `main.c`, declare a variable `Color c = RED;` and print its value.

11. Struct in Header

Declare a struct `Point { int x; int y; };` in `shapes.h`. In `main.c`, create a point, assign values, and print coordinates.

12. Preprocessor Version Macro

In `config.h`, define `#define VERSION "1.0"`. In `main.c`, include `config.h` and print the version string.

13. Multiple Translation Units

Write `file1.c` with `void f1(void) { puts("f1"); }` and `file2.c` with `void f2(void) { puts("f2"); }`. Declare them in `file.h`. Call both from `main.c`.

14. Build with Separate Compilation

Compile problem 13 with three commands (`gcc -c file1.c`, `gcc -c file2.c`, `gcc -c main.c`) and link into `prog`.

15. Mini Project with Makefile

Write a `Makefile` that builds `prog` from `main.c`, `file1.c`, `file2.c`, and `file.h`. Include `clean` target.

16. Inline vs Non-Inline Performance

Write a loop that calls a `cube_inline(int)` inline function 1 million times. Then try a `#define CUBE(x)` macro. Time the difference with `clock()`.

17. Header in `include/` Directory

Move headers to `include/` and sources to `src/`. Compile with `gcc -Iinclude src/-.c -o app`.

18. Nested Includes

Make `shapes.h` include `mathlib.h` and confirm that `main.c` can use both libraries with just one include.

19. Header Misuse Example

Put a function definition directly in a header file (`bad.h`) and include it twice in different `.c` files. Observe the "multiple definition" error. Fix it by moving the definition into `.c`.

20. Expand Calculator Project

Expand the calculator project:

- Add `mod(int,int)` and `pow_int(int,int)` in `calc.c`.
- Declare them in `calc.h`.
- Use them in `main.c` to print results.

Chapter 12. Standard Library Essentials

12.1 Input/Output (`stdio.h`)

`<stdio.h>` is C's standard input/output library. It provides:

- Streams: `stdin`, `stdout`, `stderr`
- Printing: `printf`, `fprintf`, `puts`, `putchar`
- Reading: `scanf`, `fgets`, `getchar`
- Formatted I/O with -conversion specifiers-

Streams at a glance

- `stdin` → default input (keyboard or redirected file)
- `stdout` → normal output (console or redirected file)
- `stderr` → error output (usually unbuffered; keeps errors separate)

You can redirect on the command line:

```
./app < input.txt > output.txt 2> errors.txt
```

Printing with `printf` / `puts` / `putchar`

```c
#include <stdio.h>

int main(void) {
    int n = 42;
    double x = 3.14159;

    printf("n=%d x=%.2f\n", n, x);   // formatted
    puts("hello");                   // string + newline
    putchar('A'); putchar('\n');     // single chars
    return 0;
}
```

Common format specifiers:

- `%d` (int), `%ld` (long), `%lld` (long long)
- `%u` (unsigned), `%x` (hex), `%o` (octal)
- `%f` / `%e` / `%g` (double)

- `%c` (char), `%s` (string)
- `%p` (pointer), `%zu` (size_t)
- Width/precision: `%.2f`, `%8d`, `%-10s`, `%08x`

Reading with `scanf` (quick, but picky)

```c
#include <stdio.h>

int main(void) {
    int a; double b;
    if (scanf("%d %lf", &a, &b) == 2) {
        printf("a=%d b=%.3f\n", a, b);
    } else {
        fprintf(stderr, "input error\n");
    }
    return 0;
}
```

Tips:

- Always check the return value (number of items read).
- Beware of leftover newlines when mixing `%d`/`%lf` with `%c`/`%s`.

Safer line-based input with `fgets`

Use `fgets` to read a whole line, then parse it.

```c
#include <stdio.h>

int main(void) {
    char buf[128];
    if (fgets(buf, sizeof buf, stdin)) {
        printf("line: %s", buf);  // includes '\n' if it fit
    }
    return 0;
}
```

Parsing a line (e.g., with `sscanf`):

```c
#include <stdio.h>

int main(void) {
    char buf[128];
    if (!fgets(buf, sizeof buf, stdin)) return 0;

    int a; double b;
    if (sscanf(buf, "%d %lf", &a, &b) == 2) {
        printf("ok: a=%d b=%.2f\n", a, b);
    } else {
        puts("parse error");
    }
    return 0;
}
```

Buffered I/O and `fflush`

- `stdout` is usually line-buffered on terminals; fully buffered when redirected.
- Force output now: `fflush(stdout);`
- `stderr` is typically unbuffered (appears immediately).

```c
printf("Working..."); fflush(stdout); // ensure user sees this now
```

Example: Echo with numbering (mixed I/O)

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

int main(void) {
    char line[256];
    int n = 1;
    while (fgets(line, sizeof line, stdin)) {
        // strip trailing newline (if present)
        line[strcspn(line, "\n")] = '\0';
        printf("%03d: %s\n", n++, line);
    }
    if (ferror(stdin)) { perror("stdin error"); }
    return 0;
}
```

Common pitfalls

- Forgetting `&` in `scanf` arguments (except for `%s`, which already decays to pointer).
- Using `scanf("%s", buf)` without a width: use `"%127s"` to avoid overflow.
- Mixing `scanf` and `fgets` without handling leftover newlines.
- Not checking return values of input functions.

Why this matters

Mastering `stdio.h` lets you talk to the outside world: read user input, print results, log errors, and write tools that compose with shells via redirection and pipes.

Exercises

1. Read two integers and a double from one line using `fgets` + `sscanf`. Print them back with labels and formatting.
2. Write a tiny REPL: read a line; if it's `"quit"` (case-sensitive), exit; otherwise print the length of the line (excluding newline).
3. Print a table of `i`, `i*i`, `i*i*i` for `i=1..12`, aligned in columns using width specifiers.
4. Read a word with `scanf("%127s", buf)`, then read a full line with `fgets`. Demonstrate how to handle the leftover newline so the `fgets` works as intended.
5. Print a pointer value with `%p`, and a `size_t` with `%zu`. Explain why `%d` would be incorrect for `size_t`.

12.2 Math Functions (`math.h`)

The `<math.h>` header provides standard mathematical functions for real numbers. They operate primarily on `double` (with variants for `float` and `long double`) and return results in the same type.

Common Functions

- Powers and roots

  ```
  double pow(double x, double y);    // x^y
  double sqrt(double x);             // √x
  double cbrt(double x);             //  x
  ```

- Trigonometry (radians)

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
double atan2(double y, double x); // angle from (x,y)
```

- Exponential and logarithms

```
double exp(double x);     // e^x
double log(double x);     // natural log
double log10(double x);   // base-10 log
```

- Rounding and absolute values

```
double fabs(double x);     // absolute value
double ceil(double x);     // round up
double floor(double x);    // round down
double round(double x);    // round to nearest
double trunc(double x);    // drop fractional part
```

- Hypotenuse and distance

```
double hypot(double x, double y); // sqrt(x^2 + y^2) safely
```

Type-Specific Variants

C provides suffixes to match argument types:

- `sinf`, `cosf`, `sqrtf` → work with `float`
- `sinl`, `cosl`, `sqrtl` → work with `long double`

```
float xf = 0.5f;
printf("%f\n", sinf(xf));
```

Constants

`<math.h>` (since C99) provides:

- `M_PI` (not standard everywhere, but common: 3.14159)
- In C23, constants are part of `<math.h>` under `#define` when available.

You can also define your own:

```
const double PI = 3.141592653589793;
```

Example: Right Triangle

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

int main(void) {
    double a = 3.0, b = 4.0;
    double c = hypot(a, b);   // safer than sqrt(a*a + b-b)
    double angle = atan2(b, a) - 180.0 / M_PI; // in degrees
    printf("c = %.2f, angle = %.1f°\n", c, angle);
    return 0;
}
```

Output:

```
c = 5.00, angle = 53.1°
```

Error Handling

- Many functions return NaN (not*a-number) or `HUGE_VAL` on errors.
- Use `<math.h>` macros:

  ```
  isnan(x), isinf(x), isfinite(x)
  ```

Example:

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

int main(void) {
    double x = sqrt(-1.0);
    if (isnan(x)) puts("x is NaN");
    return 0;
}
```

Why It Matters

Mathematics is at the core of many programs: graphics, simulations, engineering, finance. Using `<math.h>` gives you reliable, efficient implementations across platforms.

Exercises

1. Compute the area of a circle given radius `r` using `PI-r-r` and `M_PI`.
2. Convert degrees to radians and compute `sin`, `cos`, and `tan` of 30°, 45°, and 60°.
3. Read two sides of a right triangle, compute hypotenuse with `hypot` and the angle with `atan2`.
4. Demonstrate `ceil`, `floor`, `round`, and `trunc` on values `2.7` and `-2.7`.
5. Compute compound interest: given `P`, annual rate `r`, and years `n`, compute `P - pow(1+r, n)`.

12.3 Time and Date (`time.h`)

The `<time.h>` header provides facilities for working with time, dates, and durations. It's useful for logging, measuring execution, and scheduling.

Core Types

- `time_t` Represents a point in time (seconds since the epoch, usually Jan 1, 1970 UTC).
- `struct tm` Broken-down calendar time. Fields:

```c
struct tm {
    int tm_sec;   // 0-60
    int tm_min;   // 0-59
    int tm_hour;  // 0-23
    int tm_mday;  // 1-31 (day of month)
    int tm_mon;   // 0-11 (months since January)
    int tm_year;  // years since 1900
    int tm_wday;  // 0-6 (Sunday = 0)
    int tm_yday;  // 0-365 (days since Jan 1)
    int tm_isdst; // daylight saving time flag
};
```

Getting the Current Time

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

int main(void) {
    time_t now = time(NULL);
    printf("Epoch time: %lld\n", (long long)now);

    struct tm -local = localtime(&now);
    printf("Local: %d-%02d-%02d %02d:%02d:%02d\n",
        local->tm_year + 1900,
        local->tm_mon + 1,
        local->tm_mday,
        local->tm_hour,
        local->tm_min,
        local->tm_sec
    );
    return 0;
}
```

Formatting Time (`strftime`)

Convert `struct tm` into a formatted string:

```c
char buf[64];
strftime(buf, sizeof buf, "%Y-%m-%d %H:%M:%S", local);
puts(buf);
```

Common format specifiers:

- `%Y` = year (2025), `%m` = month, `%d` = day
- `%H` = hour, `%M` = minute, `%S` = second
- `%a` = weekday name, `%b` = month name

Measuring Durations

Use `clock()` to measure CPU time:

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

int main(void) {
    clock_t start = clock();
    for (volatile long i=0; i<100000000; i++); // busy loop
    clock_t end = clock();
    double secs = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Elapsed CPU time: %.3f seconds\n", secs);
    return 0;
}
```

For wall-clock time differences:

```c
time_t t1 = time(NULL);
// ... work ...
time_t t2 = time(NULL);
printf("Elapsed wall time: %ld seconds\n", (long)(t2 - t1));
```

Converting Between Representations

- `localtime(&t)` → `struct tm` in local time
- `gmtime(&t)` → `struct tm` in UTC
- `mktime(&tm)` → convert `struct tm` back to `time_t`

```c
struct tm t = {0};
t.tm_year = 2025 - 1900;
t.tm_mon = 11; // December
t.tm_mday = 25;
time_t xmas = mktime(&t);
printf("Christmas 2025 epoch = %lld\n", (long long)xmas);
```

Why It Matters

- Time stamps are needed in logs, files, transactions.
- Duration measurement is essential for profiling and benchmarks.
- Portable date/time handling avoids platform-specific hacks.

Exercises

1. Print the current time in both UTC and local time.
2. Use `strftime` to format today's date as `Saturday, September 6, 2025`.
3. Measure how long it takes to compute the sum of numbers 1–1e7.
4. Ask the user for a year, month, and day, build a `struct tm`, convert with `mktime`, and print the day of the week.
5. Write a function that prints a timestamp `[YYYY-MM-DD HH:MM:SS]` for use in log messages.

12.4 Random Numbers (`stdlib.h`)

C provides basic random number generation via `<stdlib.h>`. While not cryptographically secure, it's useful for simulations, games, and simple randomized behavior.

`rand()` and `srand()`

- `int rand(void);` Returns a pseudo-random integer between `0` and `RAND_MAX` (at least 32767).
- `void srand(unsigned int seed);` Seeds the random number generator. Same seed → same sequence.

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

int main(void) {
    srand((unsigned)time(NULL));  // seed with current time
    for (int i=0; i<5; i++) {
        printf("%d\n", rand());
    }
    return 0;
}
```

Scaling to a Range

Generate numbers in `[0, n)`:

```c
int r = rand() % n;   // biased if n doesn't divide RAND_MAX+1
```

Better scaling (avoids bias):

```c
int rand_range(int min, int max) {
    return min + rand() % (max - min + 1);
}
```

Example:

```c
printf("dice: %d\n", rand_range(1,6));
```

Random Floating-Point Values

Scale to `[0,1)`:

```c
double r = (double)rand() / (RAND_MAX + 1.0);
```

To `[a,b)`:

```c
double rand_double(double a, double b) {
    return a + (b - a) * ((double)rand() / (RAND_MAX + 1.0));
}
```

Deterministic Sequences

```c
srand(1234);  // fixed seed
printf("%d %d %d\n", rand(), rand(), rand());
```

This always produces the same sequence - useful for debugging.

Example: Coin Toss

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

int main(void) {
    srand((unsigned)time(NULL));
    for (int i=0; i<10; i++) {
        puts(rand() % 2 ? "Heads" : "Tails");
    }
    return 0;
}
```

Limitations

- Not secure: predictable with seed.
- Period may be short depending on implementation.
- For secure randomness (e.g., crypto), use platform APIs (`arc4random`, `/dev/urandom`, `getrandom`) instead.

Why It Matters

Randomness is key for:

- Games (dice rolls, shuffles)
- Simulations (Monte Carlo)
- Testing (randomized inputs)

Understanding limitations helps avoid misuse in security-critical code.

Exercises

1. Write a `rand_range(int min, int max)` function and test it by generating 20 random integers between 5 and 15.
2. Generate 100 random doubles in `[0,1)` and compute their average.
3. Simulate rolling two dice 1000 times. Count how often the sum is 7.
4. Seed the generator twice with the same value. Verify the outputs match.
5. Modify the coin toss program to run until you get 3 heads in a row. Print the number of tosses required.

12.5 Strings Revisited (`string.h`)

Strings in C are just arrays of `char` ending with a null terminator (`'\0'`). The `<string.h>` library provides many functions to manipulate them safely and efficiently.

Measuring and Copying

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

int main(void) {
    char s[] = "hello";
    printf("len=%zu\n", strlen(s));   // 5

    char buf[20];
    strcpy(buf, s);                   // copies entire string
    printf("%s\n", buf);

    strncpy(buf, "worldwide", sizeof buf); // copy with limit
    buf[sizeof buf - 1] = '\0';            // ensure termination
    printf("%s\n", buf);
    return 0;
}
```

- `strlen(s)` → number of characters before `'\0'`
- `strcpy(dest, src)` → copy until `'\0'` (unsafe if dest is too small)
- `strncpy(dest, src, n)` → safer copy, but may leave string unterminated

Concatenation

```
char buf[32] = "Hello, ";
strcat(buf, "World!");
printf("%s\n", buf);  // "Hello, World!"
```

Safer: `strncat(buf, "World!", sizeof buf - strlen(buf) - 1);`

Comparison

```
strcmp("abc","abc")   // 0 (equal)
strcmp("abc","abd")   // <0
strcmp("abd","abc")   // >0
strncmp("abc","abd",2) // 0 (first 2 chars equal)
```

Use `==` only for pointers, not contents.

Searching

```c
char -s = "the quick brown fox";
printf("%s\n", strchr(s,'q'));   // "quick brown fox"
printf("%s\n", strstr(s,"brown")); // "brown fox"
```

Functions:

- `strchr(s,c)` → first occurrence of `c`
- `strrchr(s,c)` → last occurrence of `c`
- `strstr(s,sub)` → substring search

Tokenizing

Break a string into tokens:

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

int main(void) {
    char text[] = "red,green,blue";
    char -tok = strtok(text, ",");
    while (tok) {
        puts(tok);
        tok = strtok(NULL, ",");
    }
    return 0;
}
```

`strtok` modifies the input string and is not thread-safe. Modern alternatives: `strtok_r` (POSIX) or manual parsing.

Memory Operations

- `memcpy(dest, src, n)` → copy raw bytes
- `memmove(dest, src, n)` → like `memcpy`, but safe for overlapping regions
- `memset(ptr, val, n)` → fill memory with value

Example:

```c
int a[5] = {1,2,3,4,5};
int b[5];
memcpy(b, a, sizeof a);  // copy entire array
```

Example: Reversing a String

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

void reverse(char *s) {
    size_t n = strlen(s);
    for (size_t i=0; i<n/2; i++) {
        char tmp = s[i];
        s[i] = s[n-1-i];
        s[n-1-i] = tmp;
    }
}

int main(void) {
    char str[] = "abcdef";
    reverse(str);
    puts(str);  // "fedcba"
    return 0;
}
```

Why It Matters

Strings are everywhere: input, output, file names, protocols. Mastering `<string.h>` is essential to handle them safely and correctly in C.

Exercises

1. Write a function `safe_copy(char *dst, size_t n, const char *src)` that copies with `strncpy` and ensures null termination.
2. Concatenate `"Hello"` and `"World"` with a space into a buffer and print it.
3. Write a program that counts how many times `"cat"` appears in `"catapult scatter catalog"`.
4. Tokenize a string `"one two three"` on spaces and print each token.

5. Implement a function `is_palindrome(const char -s)` that returns 1 if a string reads the same backward and forward, ignoring case.

Here's a comprehensive problem set for Chapter 12 (Standard Library Essentials). This covers `stdio.h`, `math.h`, `time.h`, `stdlib.h`, and `string.h` with hands-on exercises for beginners.

Problems

Input/Output (`stdio.h`)

1. Write a program that reads two integers from the user using `scanf` and prints their sum.
2. Use `fgets` to read a full line of text, then print the line and its length.
3. Print a table of numbers 1–10 alongside their squares and cubes, aligned in neat columns.
4. Redirect program output to a file (`./prog > out.txt`). Modify the program to write errors to `stderr` instead of `stdout`.
5. Write a program that reads words until EOF and prints them numbered (`1: word`, `2: word`, …).

Math Functions (`math.h`)

6. Compute the area and circumference of a circle given its radius. Use `M_PI` if available.
7. Convert 45 degrees into radians and print its sine, cosine, and tangent.
8. Write a program that asks for two sides of a right triangle and prints its hypotenuse using `hypot`.
9. Demonstrate `ceil`, `floor`, `round`, and `trunc` on `-2.7` and `2.7`.
10. Implement compound interest: given `P`, annual rate `r`, and years `n`, compute `P - pow(1+r, n)`.

Time and Date (`time.h`)

11. Print the current time in local time and UTC.
12. Format today's date as `Saturday, September 6, 2025` using `strftime`.
13. Measure how long it takes to sum integers 1–100 million.
14. Ask the user for a year, month, and day, then print what day of the week it falls on.
15. Write a function `timestamp()` that prints `[YYYY-MM-DD HH:MM:SS]` for logging.

Random Numbers (`stdlib.h`)

16. Write a `rand_range(int min, int max)` function. Use it to simulate rolling a dice 10 times.
17. Generate 100 random doubles in `[0,1)` and compute their average.
18. Simulate rolling two dice 1000 times. Count how many times the sum is 7.
19. Demonstrate that seeding with the same value gives the same random sequence.
20. Simulate a coin toss until you get 3 heads in a row. Print the number of tosses needed.

Strings (`string.h`)

21. Write a `safe_copy(char -dst, size_t n, const char -src)` function using `strncpy` and test it.
22. Concatenate `"Hello"` and `"World"` with a space in between.
23. Count how many times `"cat"` appears in `"catapult scatter catalog"`.
24. Tokenize the string `"red,green,blue"` with `strtok` and print each token.
25. Implement `is_palindrome(const char -s)` that ignores case. Test with `"Radar"` and `"level"`.