C Fundamentals

A Complete 8-Chapter Programming Course

Topics covered:

The compile-then-link model & toolchain · Variables & basic types

Operators & control flow · Loops & functions

Arrays & pointers · Strings & the classic buffer-overflow risk

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed throughout against Rust

Course 1 of 3 · Intermediate and Advanced courses follow

Table of Contents

CHAPTER 1 OF 8

Getting Started

COURSE 1 · CH 1

Getting Started

The language every "manual control" tradeoff on this site has ultimately been measured against

`rust1-1` opened by naming the tradeoff Rust refuses to accept: manual-control performance without manual-control memory bugs. C is the language that tradeoff was named against — no garbage collector, no borrow checker, no compiler enforcing safety at all. Just the programmer, trusted completely. This course starts exactly where that trust begins.

The Compile-Then-Link Model

C has no virtual machine and no interpreter — source code becomes a real, standalone executable in two genuinely separate steps. **Compilation** turns each `.c` file into an object file (machine code, but not yet a runnable program). **Linking** then combines every object file, plus any libraries being used, into one final executable. A single `gcc` command usually runs both steps back to back — but they remain conceptually distinct, and Chapter 5 of Course 2 (Multi-File Projects) is built entirely around treating them as such.

Installing a Compiler: gcc and clang

Unlike Rust's `rustup` — one official installer managing one official toolchain — C has no single canonical compiler. **gcc** (GNU Compiler Collection) and **clang** (LLVM's compiler) are the two most common; both compile standard C, both are genuinely widely used, and neither is "more official" than the other the way `rustc` is for Rust.

```
gcc --version
# gcc (Ubuntu 13.2.0) 13.2.0
```

The Smallest C Program

```c
#include <stdio.h>

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

`#include <stdio.h>` pulls in the standard I/O library's declarations — `printf` isn't a language keyword, it's an ordinary library function, declared in that header. `int main()` is the entry point, same role as Rust's `fn main()` — but notice the explicit `return 0;`: C's `main` reports a real exit status back to the operating system, and leaving it out (pre-C99) was undefined behavior, a theme this course returns to often.

Compiling & Running

```
gcc hello.c -o hello
./hello
# Hello, C!
```

There is no single-command equivalent of `cargo run` or `go run` — compiling and running are always two separate commands. `-o hello` names the output executable explicitly.

> **FORGETTING -O PRODUCES A.OUT**
> `gcc hello.c` with no `-o` flag silently compiles to a file named `a.out` — a genuine, still-common beginner surprise. Always name the output explicitly.

No Built-In Package Manager or Build Tool

This is the first real toolchain difference from both Rust and Go. Go's toolchain is minimal but complete — `go build` handles everything. Rust bundles even more into `cargo` — building, dependencies, testing, all official. C has **none of this as an official standard**: no built-in package registry, no single blessed build tool. Third-party tools like `make` (Course 2's own Chapter 5) and CMake exist and are genuinely widely used — but nothing plays the role `cargo` plays for Rust. This isn't an oversight; C predates the very idea of a language-integrated package manager by decades.

CONCEPT	RUST	C
Compile & run	`cargo run (one step)`	`gcc file.c -o out && ./out (two steps)`
Toolchain manager	`rustup (official, singular)`	`none — gcc or clang, your choice`
Package manager	`cargo + crates.io`	`none official — make/CMake handle builds only`

CONCEPT	RUST	C
Print a line	`println!(...)`	`printf("...\n")`
Entry point	`fn main()`	`int main()`

"MANUAL CONTROL" STARTS AT THE TOOLCHAIN, NOT JUST MEMORY

Rust's whole pitch was refusing the memory-safety tradeoff C accepts. But notice this chapter never even reached memory yet — C already trusts the programmer to choose a compiler, name an output file, and assemble a build process by hand. The same "no safety net, no guardrails" philosophy that governs C's memory model (Course 2's own central chapter) governs its tooling too.

Coding Challenges

CHALLENGE 1

Write a C program that prints your name and a short greeting using two separate printf calls. Compile it with gcc, explicitly naming the output file, and run it.

View solution

CHALLENGE 2

Compile a program without the -o flag. Identify the name of the resulting executable, then run it directly by that name.

View solution

CHALLENGE 3

Explain, in your own words, why C having no official package manager or build tool is a genuinely different situation than Rust or Go — not simply "C is missing a feature," but a real consequence of when and why the language was designed.

View solution

CHAPTER 1 QUICK REFERENCE

- **Compile-then-link:** two conceptually separate steps, usually run together by one `gcc` command
- **gcc** / **clang** — no single official compiler, unlike Rust's `rustc`
- `#include <stdio.h>` — pulls in library declarations; `printf` is an ordinary function, not a keyword

- `int main() { ... return 0; }` — the entry point; the explicit return value matters
- `gcc file.c -o name` — always name the output; omitting `-o` produces `a.out`
- No official package manager or build tool — `make`/CMake fill that gap unofficially, covered in Course 2
- **Next chapter:** variables, basic types, and why the standard doesn't fix their exact sizes

CHAPTER 2 OF 8

Variables & Basic Types

COURSE 1 · CH 2

Variables & Basic Types

Where C's type sizes come from a promise, not a guarantee

Chapter 1 showed C trusting the programmer with the toolchain itself. This chapter shows the same trust applied to something Rust and Go both nail down precisely: how big a number actually is.

Declaring Variables

```c
int age = 25;
float price = 19.99;
char grade = 'A';
double pi = 3.14159265;
```

`type name = value;` — no `let`, no `var`, no inference. Every variable's type is written explicitly, every time.

The Basic Types

- `int` — a whole number
- `float` — single-precision floating point
- `double` — double-precision floating point, the default for real work
- `char` — a single byte, holding a small integer that's interpreted as a character (there is no separate "character" type underneath — `char` genuinely is a small integer)

`sizeof` and Why Type Sizes Aren't Fixed

This is the chapter's real subject. In Rust, `i32` is exactly 32 bits, everywhere, by definition — the type's name states its size as a language guarantee. In C, the standard only guarantees *minimums*: `int` is guaranteed to be *at least* 16 bits, but is commonly 32 bits on today's platforms — "commonly," not "guaranteed."

```
printf("%zu\n", sizeof(int));    // commonly 4, not guaranteed
printf("%zu\n", sizeof(char));   // always 1, by definition
```

Writing genuinely portable C — code that behaves identically across every platform — means never assuming `int` is exactly 4 bytes, something Rust or Go code never has to worry about at all.

Fixed-Width Types via `<stdint.h>`

C99 added exactly the guarantee Rust and Go bake in by default — opt-in, not automatic.

```
#include <stdint.h>

int32_t exact32 = 100;   // always exactly 32 bits, guaranteed
uint64_t exact64 = 100;  // always exactly 64 bits, unsigned
```

Signed vs. Unsigned

Every integer type has a `signed` (default) and `unsigned` variant. This distinction hides a genuine asymmetry worth flagging now, well before Course 3's Undefined Behavior chapter covers it in full: **unsigned** overflow is well-defined — it wraps around (`UINT_MAX` + 1 becomes 0). **Signed** overflow is *undefined behavior* — the standard makes no promise about what happens at all, not even wraparound.

CONCEPT	RUST	C
Integer size guarantee	`i32/i64 — exact, by definition`	`int/long — minimum only, platform-dependent`
Opt-in exact width	`n/a — always exact`	`int32_t/uint64_t via <stdint.h>`
Unsigned overflow	`panics in debug, wraps in release`	`always wraps — well-defined`
Signed overflow	`panics in debug, wraps in release`	`undefined behavior — no guarantee at all`

SIZEOF IS AN OPERATOR, NOT A FUNCTION

`sizeof(int)` looks like a function call, but `sizeof` is genuinely a compile-time operator — `sizeof x` (no parentheses) works too when applied directly to a variable. The parenthesized form is just the conventional style.

SIGNED OVERFLOW DOES NOT RELIABLY WRAP AROUND

Many programmers assume signed integers wrap the same way unsigned ones do. They don't — signed overflow is undefined behavior, meaning the compiler is free to assume it never happens and optimize accordingly, sometimes

producing results stranger than a simple wraparound. Course 3's Undefined Behavior chapter covers exactly why this matters in practice.

Coding Challenges

CHALLENGE 1

Write a program that prints the sizeof int, char, float, double, and long on your system, one per line, using %zu format specifiers.

View solution

CHALLENGE 2

Declare an unsigned char variable, set it to 255, add 1 to it, and print the result. Explain what happened and why it's well-defined behavior rather than a bug.

View solution

CHALLENGE 3

Explain why int32_t from stdint.h is a genuinely different guarantee than plain int, and why Rust's i32 never needed an equivalent "opt-in exact width" type at all.

View solution

CHAPTER 2 QUICK REFERENCE

- **int/float/double/char** — explicit types, no inference
- `char` is genuinely a small integer, not a distinct character type
- The standard guarantees *minimum* sizes only — `int` is commonly 4 bytes, never guaranteed
- `<stdint.h>`'s `int32_t` / `uint64_t` — opt-in, Rust/Go-style exact widths
- **Unsigned overflow** — well-defined wraparound; **signed overflow** — undefined behavior, no guarantee
- `sizeof` is an operator, not a function — parentheses are convention, not a requirement
- **Next chapter:** operators, control flow, and C's historical lack of a true boolean type

CHAPTER 3 OF 8

Operators & Control Flow

COURSE 1 · CH 3

Operators & Control Flow

C didn't have a real boolean type until 1999 — and even then, only sort of

Chapter 2 showed the standard being deliberately vague about integer sizes. This chapter shows the same looseness applied to something even more fundamental — what counts as "true."

Arithmetic, Comparison & Logical Operators

The familiar set — `+ - * / %`, `== != < > <= >=`, `&& || !` — behaves mostly as expected, with one early trap: dividing two integers **truncates**, it doesn't round.

```
int result = 5 / 2;   // 2, not 2.5 — integer division truncates toward zero
float exact = 5.0 / 2; // 2.5 — one operand being a float forces float division
```

No True Boolean Before C99

This is the real subject of the chapter. Original C had **no boolean type at all**. Truth was just an `int`: `0` means false, *any nonzero value* means true — including negative numbers. Every `if`, every `while`, every logical expression in classic C evaluates to a plain integer, not a distinct boolean value the way Rust's `bool` always has been.

```
int is_valid = (5 > 3);  // is_valid is an int holding 1, not a bool holding true
```

`<stdbool.h>` in Modern C

C99 added `bool`, `true`, and `false` — but even now, they're not a genuinely new primitive the way Rust's `bool` is. Under the hood, `bool` is a macro for `_Bool` (an integer type that can only hold 0 or 1), and `true`/`false` are just `1` and `0` in disguise.

```c
#include <stdbool.h>

bool is_valid = true;
```

`if` / `else`

Because truth is just "nonzero," C allows *any* expression as a condition — not just a genuine comparison.

```c
if (x) {   // valid: means "if x != 0"
    // ...
}
```

This flexibility hides a classic, still-common bug: writing `=` (assignment) where `==` (comparison) was meant. `if (x = 5)` compiles cleanly, assigns 5 to `x`, and the condition is then "is 5 nonzero" — always true.

`switch`

Cases **fall through** by default — execution continues into the next case unless an explicit `break` stops it. Rust's `match` arms never fall through; each one is fully self-contained.

```c
switch (day) {
    case 1:
        printf("Monday\n");
        // no break — falls through into case 2!
    case 2:
        printf("Tuesday\n");
        break;
}
```

CONCEPT	RUST	C
Boolean type	`bool` — a real primitive, always existed	`_Bool`/`bool` since C99 — really an `int` in disguise
Condition requirement	must be a genuine `bool`	any nonzero expression works
match/switch fallthrough	never — each arm is isolated	falls through by default without `break`

COMPILE WITH WARNINGS ON

The `= vs ==` mistake is common enough that `-Wall` flags it directly. Compiling with warnings enabled from day one catches a real class of bugs the language itself won't stop you from writing.

FORGETTING BREAK IS A GENUINELY DANGEROUS DEFAULT

Unlike Rust's `match`, C's `switch` silently falls through into the next case unless every case ends in `break` — a missing `break` doesn't warn by default and can execute code that was never meant to run for that case.

Coding Challenges

CHALLENGE 1

Write a program computing 7 / 2 as an int and 7.0 / 2 as a double, printing both. Explain the difference in the output.

View solution

CHALLENGE 2

Write a switch statement over an int day (1-3) that prints a different message for each day, deliberately omitting one break to demonstrate fallthrough, then fix it.

View solution

CHALLENGE 3

Explain why if (x = 5) compiles without error in C, what it actually does, and why the equivalent mistake isn't possible in Rust.

View solution

CHAPTER 3 QUICK REFERENCE

- Integer division truncates — `5 / 2` is `2`, not `2.5`
- Classic C has no boolean type — `0` is false, any nonzero value is true
- `<stdbool.h>` (C99+) — `bool` / `true` / `false`, but really `_Bool` (an int) underneath
- Any nonzero expression is a valid `if` condition — not just genuine comparisons
- `= vs ==` in a condition compiles silently — enable `-Wall` to catch it
- `switch` falls through by default — every case needs an explicit `break`

- **Next chapter:** loops — for/while/do-while and break/continue

CHAPTER 4 OF 8

Loops

COURSE 1 · CH 4

Loops

The three-part for loop Rust deliberately doesn't have

Loops are one of the few places C's syntax and Rust's genuinely diverge in shape, not just in guarantees. This chapter covers all three C loop forms and where each one's closest Rust equivalent actually is.

The Three-Part `for` Loop

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

Init, condition, and increment, all in the parentheses. Rust has **no equivalent syntax at all** — Rust's own `for` only ever iterates over a range or an iterator (`for i in 0..5`), never a raw three-part counter. C's version is lower-level: nothing stops the counter, the bound, or the step from being anything at all.

`while` and `do-while`

```c
while (n > 0) {
    n--;
}

do {
    n--;
} while (n > 0);
```

`while` checks its condition *before* the first iteration — the body might never run. `do-while` checks *after* — the body always runs at least once. Rust has no `do-while` construct at all; the closest equivalent is a plain `loop { ... if !condition { break; } }`, spelled out manually rather than given its own keyword.

`break` and `continue`

Same meaning as in Rust — `break` exits the loop immediately, `continue` skips straight to the next iteration.

Loop Variable Scope: C89 vs. C99

Genuinely older C (C89) required loop counters to be declared *before* the loop, since variable declarations had to appear at the top of a block — `for (int i = ...)` wasn't legal syntax at all. C99 relaxed this, allowing the now-familiar in-loop declaration. Legacy C codebases still sometimes show the older style; recognizing it matters when reading real, older C.

```
// C89 style — still valid, sometimes seen in older code
int i;
for (i = 0; i < 5; i++) { /* ... */ }
```

Infinite Loops

C has no dedicated keyword for "loop forever" — the idiom is `for (;;)` or `while (1)`, both relying on an empty or always-true condition. Rust's `loop` keyword states the same intent explicitly and self-documentingly, rather than as a side effect of an empty condition.

```
for (;;) {
    // runs forever, until an explicit break
}
```

CONCEPT	RUST	C
Counted loop	`for i in 0..5`	`for (int i = 0; i < 5; i++)`
Check-after loop	`no dedicated keyword`	`do { ... } while (cond);`
Infinite loop	`loop { ... }`	`for (;;) or while (1)`

PREFER DECLARING THE LOOP VARIABLE INSIDE THE FOR

C99-style `for (int i = 0; ...)` keeps the counter's scope tightly bound to where it's actually used — simply better practice in new code, even though the older C89 style remains valid and appears in legacy codebases.

DO-WHILE NEEDS A TRAILING SEMICOLON

`} while (cond);` — the semicolon after the closing parenthesis is required and easy to forget, since no other block-ending construct in C needs one.

Coding Challenges

CHALLENGE 1

Write a for loop that prints the numbers 1 through 10, then rewrite the same logic using a while loop instead.

View solution

CHALLENGE 2

Write a do-while loop that runs its body at least once even though its condition is false from the start, printing a message that proves it ran. Explain why a plain while loop couldn't do this.

View solution

CHALLENGE 3

Explain why C's for (;;) idiom for an infinite loop is a side effect of the loop's general syntax, while Rust's loop keyword is a dedicated, self-documenting construct — and what that difference reflects about each language's design philosophy.

View solution

CHAPTER 4 QUICK REFERENCE

- `for (init; cond; incr)` — Rust has no equivalent syntax, only range/iterator-based `for`
- `while` — checks before; `do-while` — checks after, always runs at least once
- `break` / `continue` — same meaning as Rust
- C89 required loop variables declared before the loop; C99 allows in-loop declaration
- `for (;;)` / `while (1)` — C's infinite-loop idiom, vs. Rust's dedicated `loop` keyword
- `do { ... } while (cond);` — don't forget the trailing semicolon
- **Next chapter:** functions — declarations, definitions, and pass-by-value by default

CHAPTER 5 OF 8

Functions

COURSE 1 · CH 5

Functions

Why C makes you announce a function before you're allowed to call it

Chapter 1 briefly used `main` without explaining what makes a function "known" to the compiler. This chapter covers that directly — and sets up the exact problem pointers exist to solve next.

Function Syntax

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

Declarations vs. Definitions

A **declaration** (or *prototype*) tells the compiler a function exists and its exact signature — return type, name, parameter types — without providing a body. A **definition** provides the actual body. C compiles top to bottom in a single pass: calling a function before the compiler has seen either its declaration or its definition is an error. Rust, by contrast, allows calling any function defined anywhere in the same module, in any order — no forward declaration needed.

```c
int add(int a, int b); // declaration — no body

int main() {
    printf("%d\n", add(2, 3)); // legal — the compiler already saw the declaration
    return 0;
}

int add(int a, int b) { // the actual definition, further down
    return a + b;
}
```

Header Files

A `.h` file conventionally holds declarations, shared across multiple `.c` files via `#include` — the mechanism that lets one source file call a function defined in a completely different one. Course 2's Multi-File Projects & Makefiles chapter covers this in full; for now, know that a header is essentially a shared table of "these functions exist, here are their signatures."

Pass-by-Value by Default

Function arguments are **copied** into the function — modifying a parameter inside the function has no effect on the caller's original variable. This is genuinely worth a three-way comparison: Go is *also* pass-by-value by default, exactly like C — a rare point of agreement between the two. Rust is the outlier, requiring an explicit choice: pass by value (moves or copies), or pass a reference (`&T` / `&mut T`) deliberately.

```
void try_to_change(int x) {
    x = 99; // only changes the local copy
}

int main() {
    int n = 5;
    try_to_change(n);
    // n is still 5 here
}
```

Simulating Pass-by-Reference With Pointers

To let a function genuinely modify the caller's variable, C requires passing a **pointer** explicitly — the caller's address, not its value. This is exactly what the very next chapter covers in depth; for now, just recognize that "I need the function to change my variable" is the specific problem pointers solve.

CONCEPT	C	GO	RUST
Default argument passing	by value (copy)	by value (copy)	by value — moves or copies
Passing a reference	explicit pointer (&var)	explicit pointer (&var)	explicit reference (&var / &mut var)
Forward declaration needed?	yes — top-to-bottom compilation	no	no

PROTOTYPE EARLY, OR DEFINE BEFORE USE

Either declare every function used across multiple files in a shared header, or simply order definitions so nothing is called before the compiler has seen it. Both solve the same top-to-bottom compilation constraint.

LEGACY C COULD SILENTLY ASSUME A MISSING FUNCTION RETURNS INT

Pre-C99 compilers, faced with a call to a genuinely undeclared function, would silently assume it returned `int` and accept the code — a real, dangerous default mostly removed in modern C, but worth recognizing if reading older, pre-standard codebases.

Coding Challenges

CHALLENGE 1

Write a function multiply(int a, int b) that returns the product of its two arguments, declared with a prototype above main and defined below it. Call it from main and print the result.

View solution

CHALLENGE 2

Write a function that attempts to double an int parameter by reassigning it inside the function body. Call it from main with a variable set to 10, then print the variable afterward. Explain the output.

View solution

CHALLENGE 3

Explain why Rust doesn't require forward declarations for functions defined later in the same file, while C does — what does this reveal about how each language's compiler actually processes source code?

View solution

CHAPTER 5 QUICK REFERENCE

- **Declaration/prototype** — signature only, no body; **definition** — the actual body
- C compiles top to bottom — a function must be declared or defined before it's called
- **Header files (.h)** — shared declarations across multiple .c files, via #include
- Arguments are passed **by value** by default — a real point of agreement with Go, unlike Rust's explicit reference choice
- Modifying a parameter inside a function never affects the caller's variable — pointers (next chapter) are how C works around this
- **Next chapter:** arrays — fixed size, no bounds checking, and the first real "manual control" moment

CHAPTER 6 OF 8

Arrays

COURSE 1 · CH 6

Arrays

The first genuine payoff of everything Chapter 1 promised about "manual control"

Every chapter so far has shown small toolchain and syntax differences from Rust. This one is different — arrays are where C's total absence of a safety net becomes something you can actually see happen.

Declaring & Initializing Arrays

```c
int scores[5];
int values[5] = {10, 20, 30, 40, 50};
```

A plain C array has a fixed size, known at compile time — genuinely comparable to Rust's own fixed-size `[T; N]` array (not `Vec`, which can grow). Both languages have this exact concept. What happens when you go past the end is where they stop agreeing.

No Bounds Checking — The Real Difference

Accessing `values[10]` on a 5-element array is not an error in C. It's not a panic. It compiles, it runs, and it reads or writes whatever memory happens to sit past the array — memory that belongs to something else entirely. Rust's `[T; N]`, by contrast, checks every index access at runtime and **panics** on an out-of-bounds index — a controlled, visible failure instead of silent memory corruption.

```c
int values[5] = {10, 20, 30, 40, 50};
printf("%d\n", values[10]); // compiles and runs — prints garbage, or crashes, or "works"
```

A Concrete Demonstration

Two adjacent local variables often sit next to each other in memory. Writing past the end of one array can silently overwrite the other — no error, no warning, just a value that changed for no visible reason.

```c
int small[3] = {1, 2, 3};
int sentinel = 42;

small[5] = 0; // out of bounds — may silently corrupt sentinel
printf("%d\n", sentinel); // might not print 42 anymore
```

Course 2's Memory Bugs chapter covers this class of problem in full, with real tooling (`valgrind`) to catch it. For now, the point is simpler: C genuinely will not stop this.

Array-to-Pointer Decay, Previewed

Chapter 5 established pass-by-value as C's default. Arrays quietly break that rule: passing an array to a function actually passes a **pointer to its first element**, not a copy of the whole array. Chapter 7 covers pointers in full — this is just naming the exception now, so it doesn't feel unexplained later.

Getting the Array's Size

`sizeof(arr) / sizeof(arr[0])` computes the element count — but only where the array was actually declared. Once it decays to a pointer (inside a function it was passed to), `sizeof` on the parameter gives the *pointer's* size, not the array's.

```c
int values[5] = {10, 20, 30, 40, 50};
int count = sizeof(values) / sizeof(values[0]); // 5 — correct, here
```

CONCEPT	RUST [T; N]	C
Fixed size, known at compile time	yes	yes
Out-of-bounds access	panics — a controlled, visible failure	undefined behavior — silent, no check at all
Passed to a function	by value (or reference, explicit) — stays a real array	decays to a pointer to the first element

THIS IS WHAT THE WHOLE COURSE KEEPS COMING BACK TO

Every future chapter's "manual control" comparison against Rust traces back to this one moment: C trusts the programmer to stay in bounds. Nothing in the language checks. Every safety habit from here forward exists because of this fact, not despite it.

SIZEOF GIVES THE WRONG ANSWER INSIDE A FUNCTION

`sizeof(arr) / sizeof(arr[0])` only works in the scope where the array itself was declared. Compute it there, and pass the count as a separate parameter — a genuinely common, real beginner trap otherwise.

Coding Challenges

CHALLENGE 1

Declare an int array of 6 elements, initialize it with values, and print each element using a for loop and the sizeof(arr)/sizeof(arr[0]) trick to determine the loop bound.

View solution

CHALLENGE 2

Declare a 3-element array and a separate int variable directly after it. Write past the end of the array (e.g. arr[4] or arr[5]) and print the separate variable afterward. Explain what you observe and why C allows it.

View solution

CHALLENGE 3

Explain why sizeof(arr)/sizeof(arr[0]) gives the correct element count in the function where an array is declared, but gives a wrong answer if computed inside a different function the array was passed into.

View solution

CHAPTER 6 QUICK REFERENCE

- C arrays are fixed-size, known at compile time — genuinely comparable to Rust's `[T; N]`
- **No bounds checking, ever** — an out-of-range access is undefined behavior, not an error or a panic
- Rust's `[T; N]` checks every access and panics on out-of-bounds — the central contrast this chapter exists to demonstrate
- Arrays passed to a function **decay to a pointer** to the first element — breaking Chapter 5's pass-by-value rule
- `sizeof(arr)/sizeof(arr[0])` only works where the array was declared — not after it decays to a pointer
- **Next chapter:** pointers — the mechanism arrays already quietly depend on

CHAPTER 7 OF 8

Pointers

COURSE 1 · CH 7

Pointers

What Rust's borrow checker exists, specifically, to prevent

Chapter 5 needed a pointer to let a function modify a caller's variable. Chapter 6's arrays turned out to already secretly be pointers. This chapter finally names the mechanism directly — and closes the loop this whole course opened back in Chapter 1.

What a Pointer Actually Is

A pointer is a variable whose value is a **memory address**. `&` (address-of) gets a variable's address; `*` (dereference) accesses the value stored at an address.

```
int x = 42;
int *p = &x;          // p holds x's address
printf("%d\n", *p);   // *p dereferences p — prints 42
*p = 100;             // writes through the pointer — x is now 100
```

Why C Needs Pointers

- **Simulating pass-by-reference** — Chapter 5's own unsolved problem: pass `&x` instead of `x`, and the function can modify the caller's variable through the pointer
- **Working with arrays** — Chapter 6's decay *is* pointer usage; every array access secretly goes through this exact mechanism
- **Dynamic memory** — Course 2's central chapter; memory allocated at runtime is only ever reachable through a pointer

```
void increment(int *p) {
    *p = *p + 1;  // modifies the caller's actual variable
}

int n = 5;
increment(&n);
// n is now 6 — Chapter 5's limitation, solved
```

Pointer Arithmetic

`p + 1` doesn't advance by one byte — it advances by `sizeof` the pointed-to type. This is exactly what makes array indexing work: `arr[i]` is literally defined as `*(arr + i)`, the same operation spelled two different ways.

```
int arr[3] = {10, 20, 30};
int *p = arr;
printf("%d\n", *(p + 1)); // 20 — identical to arr[1]
```

The Null Pointer & Dereferencing It

`NULL` represents "points at nothing." Dereferencing a null (or uninitialized) pointer is undefined behavior — typically a crash, but never guaranteed. Rust's safe code has **no null pointers at all** — `Option<T>` replaces the entire concept, forcing an explicit check before any value can be used, eliminating this bug category by design rather than by convention.

What Rust's Borrow Checker Exists to Prevent

Here's the real payoff. A **dangling pointer** is a pointer to memory that's already been freed, or has gone out of scope — the pointer still looks perfectly valid, but what it points to is gone. C lets you create and use one freely; it's simply undefined behavior. Rust's borrow checker makes this a **compile error** — the program literally won't build if a reference could ever outlive the data it points to. This is the exact mechanism Rust's own course named as its founding goal back in `rust1-1`, and it's this specific bug class it exists to eliminate.

CONCEPT	RUST	C
Null values	`no null — Option<T>, checked explicitly`	`NULL — dereferencing it is undefined behavior`
Dangling references	`compile error — the borrow checker refuses to build`	`undefined behavior — compiles and runs, may crash later or corrupt memory`
Pointer arithmetic	`not available on safe references`	`freely available — scales by the pointed-to type's size`

ALWAYS INITIALIZE A POINTER

An uninitialized pointer holds garbage — some leftover address from whatever was in that memory before. Dereferencing garbage is one of the most common real sources of crashes. Initialize every pointer, even to `NULL`, the moment it's declared.

NEVER RETURN THE ADDRESS OF A LOCAL VARIABLE

A local variable's memory is only valid while its function is running. Returning `&local_var` hands the caller a pointer to memory that's already gone the instant the function returns — a classic dangling-pointer bug, and precisely the class of error Rust's borrow checker refuses to compile.

Coding Challenges

CHALLENGE 1

Declare an int variable, a pointer to it, print the value through the pointer, then modify the value through the pointer and print the original variable directly to confirm it changed.

View solution

CHALLENGE 2

Write a function swap(int *a, int *b) that swaps the values two pointers point to. Call it from main with two variables and print both before and after to prove the swap worked.

View solution

CHALLENGE 3

Explain, precisely, what a dangling pointer is, why C allows creating and using one, and why Rust's borrow checker refuses to compile a program that could ever produce one.

View solution

CHAPTER 7 QUICK REFERENCE

- `&` — address-of; `*` — dereference (also used in a declaration to mean "this is a pointer")
- Pointers solve pass-by-reference, array access, and (Course 2) dynamic memory
- Pointer arithmetic scales by the pointed-to type's size — `arr[i]` is literally `*(arr + i)`
- `NULL` — dereferencing it is undefined behavior; Rust's `Option<T>` removes null entirely
- **Dangling pointer** — points to freed/out-of-scope memory; UB in C, a compile error in Rust
- Never return the address of a local variable
- **Next chapter:** strings — char arrays, null termination, and the classic buffer-overflow risk

CHAPTER 8 OF 8

Strings

COURSE 1 · CH 8

Strings

Arrays and pointers, applied to text — including the bug that shaped decades of security history

C has no dedicated string type. Everything Chapters 6 and 7 covered — fixed-size arrays, no bounds checking, pointer arithmetic — applies directly to text, exactly as-is. This closing chapter of Course 1 is where those two chapters' consequences become the most historically significant.

Strings Are `char` Arrays

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

A string is just an array of `char`, nothing more.

Null Termination

The `'\0'` byte marks the end. There is no separate length field anywhere — C strings carry no length of their own. Rust's `String` stores its length directly, alongside the data; finding a C string's length means *scanning forward until `'\0'` is found*, an O(n) operation every single time, unlike Rust's O(1) length lookup.

String Literals vs. Char Arrays

```
char *literal = "hello";   // pointer to read-only memory — modifying it is UB
char mutable_str[] = "hello"; // a genuine, mutable copy — safe to modify
```

A real, important distinction: `char *s = "hello"` points at a string literal, which may live in read-only memory — writing through `s` is undefined behavior. `char s[] = "hello"` copies the characters into a genuinely mutable local array.

`string.h` Functions

```c
#include <string.h>

size_t len = strlen(s);       // scans for '\0', returns the length
strcat(dest, src);        // appends src onto the end of dest
int same = strcmp(a, b);   // 0 if equal, nonzero otherwise
```

`strcpy` vs. `strncpy` — The Classic Buffer Overflow

`strcpy` copies a string with **zero bounds checking** — exactly Chapter 6's array-overflow problem, applied specifically to text. Copying a longer string into a smaller buffer silently writes past its end. This single mistake is one of the most historically significant vulnerability classes in software security.

```c
char small[5];
strcpy(small, "This is way too long"); // buffer overflow — UB
```

`strncpy` takes a maximum length and is safer — but has its own genuine gotcha: if the source is *at least* `n` characters long, `strncpy` does not guarantee the result is null-terminated at all.

```c
char buf[5];
strncpy(buf, "Hello!", 5); // copies "Hello" — no room left for '\0'
```

CONCEPT	RUST STRING	C
Length tracking	**stored directly — O(1) lookup**	**none — scan for '\0', O(n) every time**
Bounds-checked copy	**always, by design**	**strcpy — never; strncpy — partially, with its own gotcha**
Mutability of a literal	**n/a — owned data is always mutable if declared so**	**a string literal is read-only; modifying it is UB**

ALWAYS LEAVE ROOM FOR THE NULL TERMINATOR

A buffer meant to hold `"hello"` (5 characters) needs **6** bytes, not 5 — one extra for `'\0'`. Forgetting this is one of the most common sizing mistakes in C.

STRNCPY ISN'T FULLY SAFE EITHER

If the source string's length is `>= n`, `strncpy` copies exactly `n` bytes and stops — with no guarantee the result is null-terminated. Treating `strncpy` as a fully "safe" drop-in replacement for `strcpy` is itself a real, common mistake.

Coding Challenges

CHALLENGE 1

Declare a char array using the char s[] = "..." form, print it, modify one character, and print it again to confirm the modification worked.

View solution

CHALLENGE 2

Write a function that computes a string's length by manually scanning for the null terminator (without calling strlen), and compare its result against the real strlen() for a test string.

View solution

CHALLENGE 3

Explain the specific gotcha with strncpy that makes it not a fully safe replacement for strcpy, and describe what a caller must do afterward to guarantee the result is properly null-terminated.

View solution

CHAPTER 8 QUICK REFERENCE — COURSE 1 COMPLETE

- A C string is just a `char` array — no dedicated string type exists
- `'\0'` marks the end — no separate length is ever stored, unlike Rust's `String`
- `char *s = "..."` is a read-only literal; `char s[] = "..."` is a genuine mutable copy
- `strcpy` — zero bounds checking, the classic buffer-overflow source
- `strncpy` — bounded, but doesn't guarantee null-termination if the source is `>= n` characters
- Always size a buffer for the content *plus one* byte for `'\0'`
- **Course 1 complete.** Course 2 picks up with structs, unions, and dynamic memory — C's own central safety-tradeoff chapter