C Intermediate

A Complete 7-Chapter Programming Course

Topics covered:

Structs & unions · Dynamic memory & C's own ownership model

Memory bugs & real tooling (valgrind/ASan) · The preprocessor

Multi-file projects & Makefiles · File I/O · Function pointers & hand-rolled dispatch

Exercises: 21 hands-on exercises with worked solutions

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

Course 2 of 3 · the Advanced course follows

Table of Contents

CHAPTER 1 OF 7

Structs & Unions

COURSE 2 · CH 1

Structs & Unions

Composite data — and the one place C quietly stops tracking what's actually there

Course 1 covered scalars, arrays, and the pointers connecting them. Course 2 opens with C's way of grouping several values into one — and, in the same chapter, its way of letting several values *share* one.

Declaring & Using Structs

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

struct Point p = {3, 4};
printf("%d, %d\n", p.x, p.y);
```

typedef

Referring to `struct Point` everywhere gets verbose fast. `typedef` gives the type a shorter name — an idiom so common in real C that it's effectively the default style.

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

Point p = {3, 4}; // no "struct" keyword needed anymore
```

Rust never needed this ceremony — a `struct`'s own name is always the type, directly, with no separate aliasing step required.

Structs Are Pure Data

A C struct has no methods attached to it — no `impl` block equivalent from Rust. Any function operating on a struct is written separately, typically taking a pointer to it as a parameter. Course 2's own Function Pointers chapter (Ch.7) later shows one way C approximates method-like dispatch without a language feature for it at all.

Unions

A `union` looks like a struct syntactically, but its members **share the same memory** — only one is genuinely "active" at a time. A union's `sizeof` equals its *largest* member's size, not the sum of all members, unlike a struct.

```
union Value {
    int as_int;
    float as_float;
};

union Value v;
v.as_int = 42;
// v.as_float now reads the SAME bytes, reinterpreted as a float — not 42.0
```

The Danger of Type Punning via Unions

Reading a union member that wasn't the one last written is, in most cases, undefined behavior — the union itself carries no record of which member is currently valid. Compare this directly to Rust's own data-carrying enums (`rust1-6`): a Rust enum stores a hidden **discriminant** tracking which variant is actually active, so accessing the wrong one is caught as a compile-time type error through `match`, not silently allowed at runtime. A C union tracks nothing at all — the programmer is expected to remember.

CONCEPT	RUST ENUM	C UNION
Tracks which variant/member is active	**yes — a hidden discriminant, checked by match**	**no — nothing tracks it at all**
Reading the "wrong" one	**not possible — match forces handling every case**	**undefined behavior in most cases**
Size	**largest variant + discriminant tag**	**exactly the largest member — no tag**

> **TYPEDEF STRUCT IS THE IDIOMATIC DEFAULT**
> Combining an anonymous struct definition with a `typedef` in one statement is extremely common in real C code — expect to see it constantly when reading existing codebases.

READING THE WRONG UNION MEMBER ISN'T "READING GARBAGE" — IT'S UB

This is genuinely stronger than "you'll get a nonsense value." Because it's undefined behavior, the compiler is free to assume it never happens at all, and can optimize the surrounding code in ways that produce surprising results well beyond just a wrong number — exactly the same class of danger as Course 1's signed-overflow warning.

Coding Challenges

CHALLENGE 1

Define a Point struct with x and y int fields using typedef struct, create an instance with values 5 and 10, and print both fields.

View solution

CHALLENGE 2

Define a union with an int member and a float member. Write 100 into the int member, then print both the int member and the float member. Explain why they don't show a related value.

View solution

CHALLENGE 3

Explain how Rust's enum discriminant makes reading the "wrong" variant impossible at compile time, while a C union has no equivalent safeguard at all — and why this makes the union version genuinely undefined behavior rather than just an inconvenience.

View solution

CHAPTER 1 QUICK REFERENCE

- `struct` groups multiple values; access members with `.`
- `typedef struct { ... } Name;` — the idiomatic default, avoiding repeated `struct` keywords
- C structs carry no methods — functions operating on them are always separate, taking a pointer
- `union` members **share memory** — `sizeof` is the largest member's size, not the sum
- Reading the wrong union member is undefined behavior — no discriminant tracks which one is active, unlike Rust's enums

- **Next chapter:** dynamic memory — malloc/calloc/realloc/free, and C's own version of Rust's ownership model

CHAPTER 2 OF 7

Dynamic Memory

COURSE 2 · CH 2

Dynamic Memory

The second half of this track's central payoff — what Rust's ownership model does automatically, done entirely by hand

`c1-7` named dangling pointers as the exact bug class Rust's borrow checker exists to prevent. This chapter shows where those dangling pointers actually come from — and hands you the same responsibility Rust's ownership system otherwise carries for you.

Stack vs. Heap

The **stack** is automatic: a local variable's memory exists for exactly as long as its function is running, freed the instant that function returns — this is precisely what made returning `&local_var` in `c1-7` a dangling pointer. The **heap** is manual: memory allocated there exists until something explicitly releases it, with no connection to any particular function's lifetime, and effectively unlimited size.

`malloc`

Allocates raw, *uninitialized* memory of a given byte size, returning `void *` — cast or assigned to a typed pointer. On failure, it returns `NULL`, which must always be checked; Rust, by contrast, typically aborts automatically on allocation failure rather than returning a value the caller might forget to verify.

```
int *nums = malloc(5 * sizeof(int));
if (nums == NULL) {
    // allocation failed — must be handled, not assumed away
}
```

`calloc`

Like `malloc`, but **zero-initializes** the memory, and takes element count and element size as two separate arguments. The zero-init distinction genuinely matters — `malloc`'s memory is garbage until written, and reading it before initializing is its own class of bug.

```c
int *nums = calloc(5, sizeof(int)); // all 5 ints start at 0
```

`realloc`

Resizes a previous allocation — and may move it to an entirely new address. The returned pointer can genuinely differ from the one passed in, and the *old* pointer becomes invalid the instant `realloc` succeeds.

```c
int *temp = realloc(nums, 10 * sizeof(int));
if (temp == NULL) {
    // realloc failed — nums is still valid and unchanged; don't overwrite it
} else {
    nums = temp; // only reassign after confirming success
}
```

`free`

Releases heap memory back to the system. Immediately afterward, the pointer that referenced it becomes a genuine **dangling pointer** — this is the exact mechanism `c1-7` warned about, closing the loop directly: freeing is how dangling pointers are most commonly created in real code.

```c
free(nums);
// nums now points at freed memory — using it here is undefined behavior
```

Manual Memory Management vs. Rust's Ownership

This is the chapter's real subject. In C, *you* decide when to call `free`, and nothing enforces it. Forget it — a **leak**. Call it twice on the same pointer — a **double-free**, undefined behavior. Use the pointer after freeing — **use-after-free**, also undefined behavior. Rust's ownership model (`rust1-3`) tracks, at *compile time*, exactly when a value's owner goes out of scope, and automatically inserts the equivalent of `free` at precisely that point — no leaks in safe code, no double-frees, no use-after-free, guaranteed by the compiler rather than the programmer's own discipline. Every rule this chapter just described by hand is what Rust's ownership system enforces silently, every time, without exception.

CONCEPT	RUST	C
When memory is freed	automatically, when the owner goes out of scope	only when you explicitly call free()
Forgetting to free	not possible in safe code	a memory leak

CONCEPT	RUST	C
Freeing twice	`not possible — ownership moves, can't double-drop`	`a double-free — undefined behavior`
Using memory after it's freed	`compile error — the borrow checker rejects it`	`a use-after-free — undefined behavior`

WRITE THE FREE RIGHT AFTER THE ALLOCATION

A genuinely effective habit: write the matching `free` immediately after an allocation, before writing the code that uses the memory in between — then move the `free` to its real, correct location. It's much harder to forget a `free` you already wrote once than one you were planning to add "later."

NEVER REASSIGN REALLOC'S RESULT DIRECTLY INTO THE ORIGINAL POINTER

`ptr = realloc(ptr, new_size);` is a real, common bug: if `realloc` fails and returns `NULL`, that `NULL` overwrites `ptr` — permanently losing the only reference to the still-valid original allocation, which now leaks with no way to free it. Always assign to a temporary variable first.

Coding Challenges

CHALLENGE 1

Allocate an array of 5 ints with malloc, fill it with values 1 through 5, print them, then free the memory.

View solution

CHALLENGE 2

Allocate an array of 3 ints with calloc, print all three values before writing anything to them, then explain what they show and why, contrasted with what malloc's uninitialized memory would show instead.

View solution

CHALLENGE 3

Explain, precisely, why Rust's ownership model makes a double-free structurally impossible in safe code, rather than merely unlikely or discouraged.

View solution

CHAPTER 2 QUICK REFERENCE

- **Stack** — automatic, freed on function return; **heap** — manual, freed only by explicit action
- `malloc` — uninitialized memory, must check for `NULL`
- `calloc` — zero-initialized memory, count and size given separately
- `realloc` — may move the allocation; always assign to a temp variable first, never the original pointer
- `free` — releases memory; the pointer becomes dangling immediately afterward
- **Leaks, double-frees, use-after-free** — all manual failure modes in C; all structurally prevented by Rust's ownership model at compile time
- **Next chapter:** Memory Bugs — the tooling (valgrind) that catches exactly these mistakes

CHAPTER 3 OF 7

Memory Bugs

COURSE 2 · CH 3

Memory Bugs

Catching at runtime, with tools, what Rust's compiler catches for free

Every bug this chapter names has already appeared somewhere in this course — `c1-6`'s buffer overflow, `c1-7`'s dangling pointer, `c2-2`'s leaks and double-frees. This chapter gathers them into one catalog, and introduces the tools real C developers actually use to catch them.

The Bug Catalog, Named Precisely

- **Use-after-free** — dereferencing a pointer after its memory was freed (`c1-7`, `c2-2`)
- **Double-free** — calling `free` twice on the same allocation (`c2-2`)
- **Memory leak** — allocated memory that's never freed, with no remaining pointer to it at all (`c2-2`)
- **Buffer overflow** — reading or writing past an array or allocation's actual bounds, stack or heap (`c1-6`, `c1-8`)
- **Uninitialized read** — reading memory (from `malloc`, or a declared-but-unset variable) before anything was written to it

A Genuine Leak Example

```
void process() {
    int *buf = malloc(100 * sizeof(int));
    // ... uses buf ...
    // no free(buf) — the pointer goes out of scope, the memory doesn't
}

for (int i = 0; i < 1000000; i++) {
    process(); // leaks 100 ints, one million times over
}
```

Each call leaks a small, fixed amount — individually harmless-looking, but accumulating without bound. Rust makes this specific pattern impossible: a value's `Drop` runs automatically the moment its owner goes out of scope, with no equivalent "forgot to call it" failure mode in safe code.

`valgrind`

A dynamic analysis tool — run an already-compiled program under it, no recompilation needed. It reports leaks, invalid reads/writes, and use-after-free, each with a stack trace showing exactly where the bad allocation, access, or free occurred.

```
$ valgrind --leak-check=full ./program

==12345== HEAP SUMMARY:
==12345==   definitely lost: 400 bytes in 1 blocks
==12345==
==12345== 400 bytes in 1 blocks are definitely lost
==12345==    at malloc (in /usr/lib/valgrind/...)
==12345==    by 0x4006B7: process (program.c:3)
```

AddressSanitizer (ASan)

A compile-time instrumentation approach — add `-fsanitize=address` and recompile. It catches much of the same bug class, but faster than `valgrind`, and crashes immediately with a detailed report at the exact moment the bad access happens, rather than only summarizing at program exit.

```
$ gcc -fsanitize=address -g program.c -o program
$ ./program

==12345==ERROR: AddressSanitizer: heap-use-after-free
    READ of size 4 at 0x602000000010 thread T0
    #0 0x... in main program.c:8
```

TOOL	REQUIRES RECOMPILING	SPEED
`valgrind`	no	slower — full emulation
`AddressSanitizer`	yes — a compiler flag	much faster, precise timing

Why This Chapter Exists

This is the honest cost of the tradeoff Chapter 1 named: C's speed and control come with these exact bug classes as a genuine, permanent possibility — not a temporary tooling gap that will eventually be fixed. Rust's compiler catches most of this entire category at *compile time*, before the program ever runs. C requires catching it at *runtime*, with tools like these, ideally before shipping — a real and lasting asymmetry between the two languages, not a difference in maturity or polish.

BUG CLASS	RUST	C
Use-after-free	compile-time — borrow checker	runtime — valgrind/ASan, if you run them
Double-free	compile-time — single ownership	runtime — valgrind/ASan, if you run them
Memory leak	not possible in safe, non-cyclic code	runtime — valgrind's leak checker, if you run it
Buffer overflow	runtime panic — always checked	runtime — ASan, if you run it; otherwise silent UB

RUN THESE TOOLS ROUTINELY, NOT JUST WHEN SOMETHING LOOKS WRONG

Many memory bugs produce no visible symptom most of the time. Building the habit of running `valgrind` or an ASan-instrumented build regularly during development — not only when chasing a specific crash — catches problems long before they become a mystery in production.

UNDEFINED BEHAVIOR IS NOT GUARANTEED TO CRASH

A program with a genuine memory bug can appear to work correctly for a long time — sometimes indefinitely, on a given machine and compiler. Undefined behavior means the standard makes no promise at all, not "it will crash reliably." This unpredictability is exactly what makes this bug class so dangerous in practice: the absence of a crash is not evidence of correctness.

Coding Challenges

CHALLENGE 1

Write a small program with a deliberate memory leak (malloc without a matching free), compile it, and run it under valgrind --leak-check=full. Report what the leak summary shows.

View solution

CHALLENGE 2

Write a small program with a deliberate use-after-free (free a pointer, then dereference it), compile it with -fsanitize=address, and run it. Report what AddressSanitizer's error output shows.

View solution

CHALLENGE 3

Explain why "the program didn't crash" is not proof a C program is free of memory bugs, and why the same reasoning doesn't apply to a Rust program that compiles successfully.

View solution

CHAPTER 3 QUICK REFERENCE

- **Bug catalog:** use-after-free, double-free, memory leak, buffer overflow, uninitialized read
- `valgrind --leak-check=full` — no recompile needed, slower, reports leaks/invalid access with stack traces
- `-fsanitize=address` — requires recompiling, much faster, crashes precisely at the moment of the bad access
- Rust catches most of this category at compile time; C requires catching it at runtime, with tools
- Undefined behavior is not guaranteed to crash — a program can "work" for a long time despite a real bug
- **Next chapter:** the preprocessor — #define, macros, and conditional compilation

CHAPTER 4 OF 7

The Preprocessor

COURSE 2 · CH 4

The Preprocessor

A text-substitution pass that runs before the compiler even sees real C

`#include <stdio.h>` has appeared in nearly every example since Chapter 1. This chapter finally explains what it — and every other line starting with `#` — actually does.

The Preprocessor Runs Before Compilation

The preprocessor is a genuinely separate pass: pure text substitution over the source file, completed entirely before parsing or type-checking ever begins. Rust has **no preprocessor at all** — its macro system (`rust3-4`) operates on the actual parsed syntax tree, with real awareness of types and structure, not raw text. This is a fundamental difference in kind, not just syntax.

`#define` — Object-like Macros

```
#define MAX_SIZE 100
```

Every occurrence of `MAX_SIZE` in the file is replaced, textually, with `100` — before the compiler ever sees it as a number. No type checking happens at this stage at all.

`#define` — Function-like Macros

```
#define SQUARE(x) ((x) * (x))
```

Looks like a function; is pure text substitution. Without full parenthesization, this breaks badly:

```
// #define SQUARE(x) x * x   (missing parens)
SQUARE(a + b)   // expands to: a + b * a + b — wrong, due to operator precedence
```

Even with correct parentheses, macros carry a trap no real function has: arguments can be **evaluated more than once**.

```
int i = 5;
SQUARE(i++);   // expands to ((i++) * (i++)) — i is incremented TWICE, not once
```

`#include`

Now the honest explanation: `#include` pastes the *entire contents* of the named file, textually, in place of the `#include` line itself — nothing more sophisticated than that. It's not an "import" in Rust's sense at all; there's no module system underneath, just literal text insertion before compilation.

Header Guards

Including the same header twice — common once a project has several source files each including several headers — pastes its contents twice, causing duplicate definitions and a real compile error. The classic fix:

```
#ifndef MYHEADER_H
#define MYHEADER_H

// header contents

#endif
```

The modern, widely-supported alternative is a single line:

```
#pragma once
```

Rust's module system has no equivalent problem at all — `use` never textually pastes anything, so there's nothing to accidentally duplicate.

Conditional Compilation

`#ifdef` / `#ifndef` / `#if` / `#else` / `#endif` compile different code depending on which macros are defined — commonly used for platform-specific code, or separating debug and release builds.

```
#ifdef DEBUG
    printf("Debug: value = %d\n", value);
#endif
```

CONCEPT	RUST	C
Preprocessor	none — macros operate on the syntax tree	a genuine separate text-substitution pass
Module inclusion	use — a real module reference, no text pasting	#include — literal text pasting
Double-inclusion risk	not possible — no text pasting involved	real — requires header guards or #pragma once

PREFER #PRAGMA ONCE IN NEW CODE

Simpler than the `#ifndef` / `#define` / `#endif` pattern, and supported by every major compiler — though technically a compiler extension, not part of the C standard itself.

MACRO ARGUMENTS CAN BE EVALUATED MORE THAN ONCE

Always fully parenthesize both the parameters and the entire macro body — but even then, remember that a function-like macro is not a real function: an argument with a side effect (like `i++`) can be silently evaluated multiple times, a class of bug that simply cannot occur with a real function call.

Coding Challenges

CHALLENGE 1

Define a macro SQUARE(x) without any parentheses around x or the whole expression. Call it as SQUARE(2 + 3) and show what it actually expands to and evaluates as, compared to the mathematically correct answer (25).

View solution

CHALLENGE 2

Write a header file with a struct definition, protected by a header guard using #ifndef/#define/#endif. Explain what compile error would occur if the guard were removed and the header were included twice from the same source file.

View solution

CHALLENGE 3

Explain why SQUARE(i++), even with SQUARE fully and correctly parenthesized as ((x) * (x)), is still a bug — and why the equivalent call to a real function square(i++) would never have this problem.

📄 View solution

CHAPTER 4 QUICK REFERENCE

- The preprocessor is a text-substitution pass, complete before compilation begins — unlike Rust, which has none
- `#define NAME value` — object-like macro, pure text substitution
- Function-like macros need full parenthesization — and can still evaluate arguments multiple times
- `#include` literally pastes a file's contents — not a real module import
- Header guards (`#ifndef` / `#define` / `#endif` or `#pragma once`) prevent duplicate-inclusion errors
- `#ifdef` / `#if` / `#else` / `#endif` — conditional compilation for platform/debug-vs-release code
- **Next chapter:** Multi-File Projects & Makefiles — separate compilation and linking, for real

CHAPTER 5 OF 7

Multi-File Projects & Makefiles

COURSE 2 · CH 5

Multi-File Projects & Makefiles

Resolving Chapter 1's own open question: what fills the gap where cargo would be

`c1-1` named the gap directly — no official build tool. `c1-5` introduced header files in passing. `c2-4` explained exactly what `#include` does. This chapter puts all three together, for real, on a genuine multi-file project.

Why Split a Project Across Files

Organization at real scale, and something more practical: **faster rebuilds**. Recompiling one changed file is far cheaper than recompiling an entire project from scratch every time — but only if the build process actually takes advantage of that.

Header Files as Contracts

A header declares *what* a source file provides, without exposing *how*. Per `c2-4`'s literal explanation, those declarations get pasted, verbatim, into every file that includes the header — giving each one enough information to call the functions without ever seeing their implementation.

```
// math_utils.h
int add(int a, int b);
```

Separate Compilation

This is `c1-1`'s compile-then-link model, made concrete across multiple files. Each `.c` file compiles *independently* into its own object file (`.o`); only afterward are all the object files linked into one executable.

A Worked Example

```c
// math_utils.c
#include "math_utils.h"

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

// main.c
#include <stdio.h>
#include "math_utils.h"

int main() {
    printf("%d\n", add(2, 3));
    return 0;
}
```

```
$ gcc -c math_utils.c -o math_utils.o
$ gcc -c main.c -o main.o
$ gcc math_utils.o main.o -o program
$ ./program
5
```

The Linking Step, In Detail

`main.o` contains a call to `add()` with no body — an unresolved external symbol. `math_utils.o` contains `add()`'s actual definition. The **linker**'s job is matching them up. Forget to link `math_utils.o` in at all, and the result is one of C's most recognizable real-world errors:

```
$ gcc main.o -o program
undefined reference to `add'
collect2: error: ld returned 1 exit status
```

A Real Makefile

`make` is the unofficial-but-standard answer to `c1-1`'s own open question — targets, dependencies, and rules, rebuilding only what actually changed.

```
program: main.o math_utils.o
	gcc main.o math_utils.o -o program

main.o: main.c math_utils.h
	gcc -c main.c -o main.o

math_utils.o: math_utils.c math_utils.h
	gcc -c math_utils.c -o math_utils.o

clean:
	rm -f *.o program
```

Run `make`, and it rebuilds only the files whose dependencies (listed after the colon) are newer than their target — editing only `math_utils.c` recompiles just that file and re-links, leaving `main.o` untouched.

CONCEPT	RUST / CARGO	C / MAKE
Build tool	**official, built in**	**unofficial — make/CMake fill the gap**
Incremental rebuilds	**automatic**	**explicit — dependency rules must be written correctly**
Unresolved symbol	**compile error — caught before linking exists as a concept**	**a distinct linker error — "undefined reference"**

RECOGNIZE "UNDEFINED REFERENCE" AS A LINKER ERROR

It means the *declaration* was found (the header compiled fine) but the *definition* was never linked in — a genuinely distinct failure mode from a compile error, worth recognizing immediately rather than re-reading the source for a typo that isn't there.

AN INCOMPLETE DEPENDENCY LIST SILENTLY BUILDS STALE FILES

If a Makefile rule doesn't list a header as a dependency, changing that header won't trigger a rebuild of files that include it — `make` will report success while quietly building against outdated object files. Always list every header a `.c` file includes as part of its rule's dependencies.

Coding Challenges

CHALLENGE 1

Create the three files from this chapter's worked example (math_utils.h, math_utils.c, main.c), compile and link them manually with separate gcc -c and gcc linking commands, and run the result.

View solution

CHALLENGE 2

Deliberately compile only main.c into an executable, omitting math_utils.o from the final link command. Report the exact error and explain what it means.

View solution

CHALLENGE 3

Write a Makefile for the three-file project from Challenge 1, following the chapter's own pattern. Explain what happens, in terms of which files actually get recompiled, if you run make twice in a row with no changes in between.

View solution

CHAPTER 5 QUICK REFERENCE

- A header declares *what* a source file provides — a contract, not an implementation
- `gcc -c file.c -o file.o` — compiles independently into an object file
- Linking matches unresolved calls (in one `.o`) against real definitions (in another)
- **"undefined reference"** — a linker error, distinct from a compile error; the definition was never linked in
- `make` — the unofficial standard build tool, rebuilding only what its dependency rules say changed
- Every header a source file includes must be listed as a dependency, or changes to it won't trigger a rebuild
- **Next chapter:** File I/O — fopen/fread/fwrite/fclose, text vs. binary mode

CHAPTER 6 OF 7

File I/O

COURSE 2 · CH 6

File I/O

Another manually-acquired, manually-released resource — and the function C's own standard body removed entirely

Every program so far has lived entirely in memory. This chapter opens the door to the filesystem — and introduces a resource-management discipline that should already feel familiar from `c2-2`'s `malloc`/`free` pairing.

Opening a File

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    // failed to open — must be checked, same discipline as malloc
}
```

File Modes

The mode string controls both direction and format: `"r"` read, `"w"` write (truncating), `"a"` append — each with a `b` variant (`"rb"`, `"wb"`, `"ab"`) for binary.

Text Mode vs. Binary Mode

Text mode may **translate line endings** depending on the platform — on Windows, `\n` can be translated to/from `\r\n` on read/write. Binary mode transfers bytes exactly as-is, with no translation at all. This is genuinely platform-dependent behavior, worth knowing explicitly rather than discovering by accident. Rust's `std::fs` has no equivalent mode distinction at the language level — it's byte-oriented by default, with no automatic translation to opt out of.

Reading & Writing Text

```
fprintf(fp, "%d\n", 42);

char line[256];
fgets(line, sizeof(line), fp); // bounded — safe
```

`fgets` takes a maximum buffer size, exactly the bounds-checking discipline `c1-8` covered. Its predecessor, `gets()`, took no size argument at all — it was so fundamentally unfixable that C11 **removed it from the standard entirely**.

Reading & Writing Binary Data

```
int nums[5] = {1, 2, 3, 4, 5};
size_t written = fwrite(nums, sizeof(int), 5, fp);
// written may be less than 5 — a short write is possible and must be checked
```

`fread` / `fwrite` operate on raw bytes: a pointer, the size of each element, and how many elements. Both return the *actual* number of items transferred — which can genuinely be less than requested, and isn't automatically an error condition to ignore.

Closing a File

```
fclose(fp);
```

Flushes any buffered writes and releases the underlying OS file handle. Forgetting it is the same class of mistake as `c2-2`'s forgotten `free` — a resource acquired manually that must be released manually, just a file descriptor instead of heap memory.

CONCEPT	RUST	C
Text/binary mode distinction	none — byte-oriented by default	explicit — "r" vs "rb", genuinely platform-dependent
Closing a file	automatic — Drop closes it when the handle goes out of scope	manual — fclose must be called explicitly
Unbounded line reading	not available — read_line always takes a growable buffer	gets() existed, then was removed entirely in C11

> **CHECK FOPEN EXACTLY LIKE MALLOC**

A failed `fopen` returns `NULL` — the same discipline `c2-2` established for `malloc`. A missing file, wrong permissions, or a full disk are all real, common causes; never assume the file opened successfully.

GETS() WAS REMOVED FROM THE C STANDARD — THE STRONGEST STATEMENT C MAKES ABOUT A FUNCTION

`gets()` read a line with no way to specify a maximum length at all — genuinely, unfixably unsafe, no matter how carefully called. C11 didn't just discourage it; it removed it from the standard outright. Use `fgets`, always.

Coding Challenges

CHALLENGE 1

Write a program that opens a file for writing, writes three lines of text to it with fprintf, closes it, then reopens the same file for reading and prints each line back out with fgets.

View solution

CHALLENGE 2

Write a program that writes an array of 5 ints to a file in binary mode with fwrite, then reads them back into a different array with fread, printing both arrays to confirm they match.

View solution

CHALLENGE 3

Explain specifically why gets() could never be made safe with better documentation or careful usage alone, and why fgets's extra parameter is what actually fixes the underlying problem.

View solution

CHAPTER 6 QUICK REFERENCE

- `fopen(name, mode)` — returns `NULL` on failure, must always be checked
- `"r"` / `"w"` / `"a"` plus a `b` variant for binary — text mode may translate line endings, platform-dependently
- `fgets` — bounded, safe; `gets()` — unbounded, removed entirely from C11
- `fread` / `fwrite` — return the actual items transferred, which can be less than requested
- `fclose` — manual, exactly like `free`; forgetting it leaks a file descriptor

- **Next chapter:** Function Pointers — syntax, callbacks, and a vtable-style dispatch pattern

CHAPTER 7 OF 7

Function Pointers

COURSE 2 · CH 7

Function Pointers

What C++ classes and Rust's dyn Trait do automatically, built here entirely by hand

`c2-1` named this chapter directly: "one way C approximates method-like dispatch without a language feature for it at all." Course 2 closes by actually building that mechanism — the same one every object-oriented language's runtime dispatch is quietly built from underneath.

Function Pointer Syntax

```
int (*operation)(int, int);
```

A pointer to a function taking two `int`s and returning `int`. The parentheses around `*operation` are mandatory — without them, this parses as something else entirely, a genuine syntax trap worth memorizing rather than re-deriving each time.

Assigning & Calling Through a Function Pointer

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

int (*operation)(int, int) = add; // a function name decays to a pointer to itself
int result = operation(2, 3); // calling through the pointer looks identical
```

A function name decaying to a pointer to itself is genuinely the same phenomenon as array-to-pointer decay from `c1-6`, just applied to functions instead of arrays.

Callbacks

Passing a function pointer so another function can call it later — a real, common C idiom.

```c
void apply_to_all(int *arr, int n, void (*fn)(int)) {
    for (int i = 0; i < n; i++) {
        fn(arr[i]);
    }
}
```

A Simple vtable-Style Dispatch Pattern

The real payoff: store a function pointer *alongside data*, and let different instances point at different implementations.

```c
typedef struct {
    float radius;
    float (*area)(void *self);
} Circle;

float circle_area(void *self) {
    Circle *c = (Circle *)self;
    return 3.14159f * c->radius * c->radius;
}

Circle c = {5.0f, circle_area};
float a = c.area(&c); // dispatches to circle_area at runtime
```

A second shape type populates its own struct's function pointer with a different implementation entirely — calling `shape->area(shape)` dispatches to the correct one, purely because the pointer was set to point there.

What C++ and Rust Formalize

This is not just an analogy — it's the literal mechanism. A C++ class with `virtual` methods, and Rust's own `dyn Trait`, both work by storing a table of function pointers alongside the data, dispatched through one level of indirection — exactly what was just built by hand above. The difference is who manages the table: here, the programmer wires it up manually and must get it right; in C++ and Rust, the *compiler* generates the table automatically and guarantees it's always fully and correctly populated.

CONCEPT	RUST DYN TRAIT	C FUNCTION POINTER STRUCT
Who builds the dispatch table	**the compiler, automatically**	**the programmer, by hand**
Guaranteed fully populated?	**yes — enforced at compile time**	**no — nothing checks it at all**
Calling an unset entry	**not possible — compile error**	**undefined behavior — a NULL or garbage call**

A GENUINELY REAL TECHNIQUE, NOT JUST A TEACHING EXERCISE

This hand-rolled pattern is exactly how real, large C codebases implement polymorphism — the Linux kernel's own driver interfaces are built from structs of function pointers just like this one.

NOTHING CHECKS THAT THE FUNCTION POINTER WAS SET CORRECTLY

Forgetting to populate a struct's function pointer, or assigning the wrong function to it, produces undefined behavior the moment it's called — a `NULL` call, or a call through a mismatched signature, with no compiler check whatsoever. This is precisely what Rust's `dyn Trait` and C++'s `virtual` mechanism guarantee away entirely.

Coding Challenges

CHALLENGE 1

Write two functions, add(int, int) and subtract(int, int), and a single function pointer variable that can be pointed at either one. Call it once through each, printing both results.

View solution

CHALLENGE 2

Write a function apply_to_all(int *arr, int n, void (*fn)(int)) that calls fn on every element of an array, and a print_doubled(int) function to pass as the callback that prints each value doubled.

View solution

CHALLENGE 3

Extend the chapter's Circle vtable-style example with a second shape (e.g. a Rectangle) that has its own area function, both sharing a common calling pattern. Then explain what specifically would happen if a third shape's function pointer were left unset (NULL) and its area were called.

View solution

CHAPTER 7 QUICK REFERENCE — COURSE 2 COMPLETE

- `return_type (*name)(param_types);` — the parentheses around `*name` are mandatory
- A function name decays to a pointer to itself — the same phenomenon as `c1-6`'s array decay
- **Callbacks** — passing a function pointer so another function can call it later

- A struct of data plus a function pointer is a hand-rolled vtable — real dispatch, written manually
- C++'s `virtual` and Rust's `dyn Trait` are the exact same mechanism, compiler-managed and guaranteed correct — this chapter's version is neither
- **Course 2 complete.** Course 3 covers bit manipulation, hand-built data structures, concurrency, undefined behavior in full depth, debugging tooling, and a real capstone CLI project