MODERN ANSI C PROGRAMMING

A COMPLETE GUIDE TO CORRECT, PORTABLE, AND PROFESSIONAL C CODE

- PORTABLE ACROSS PLATFORMS
- SECURE AND RELIABLE
- EFFICIENT AND PERFORMANCE FOCUSED
- MAINTAINABLE CODE FOR TEAMS

- CONFORMANT WITH ISO STANDARDS
- DATA STRUCTURES AND ALGORITHMS
- CONCURRENCY AND SYSTEMS PROGRAMMING
- PROFESSIONAL ENGINEERING PRACTICES

COMPLETE, COMPILABLE EXAMPLES IN EVERY CHAPTER
EXPLAINING WHAT WORKS, WHAT CAN GO WRONG, AND WHY

BENJAMIN GRAYSON

Modern ANSI C Programming

A Complete Guide to Correct, Portable, and Professional C Code

Steve Publications

This book is available at https://leanpub.com/theanatomyofansic

This version was published on 2026-08-24

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Complete Guide to Correct, Portable, and Professional C Code

This book takes you from fundamental C concepts through advanced, production-quality development. It teaches you how to write code that is portable across platforms, conformant with ISO standards, secure against common vulnerabilities, efficient in its use of resources, and maintainable by teams over years. Every chapter builds on the last, moving from the compilation model and core syntax through memory management, data structures, concurrency, and professional engineering practices. You will find complete, compilable examples throughout; each one is designed to illustrate not just what a construct does but why it works that way, what can go wrong, and how to use it correctly in real projects.

Introduction: Why C Still Demands Mastery

In 2024 the ISO committee published its fifth major revision of the C standard, more than forty years after the first formal specification appeared. The fact that this language continues to evolve at all is remarkable; the fact that it remains essential is not surprising. C runs operating systems, database engines, web servers, embedded controllers, cryptographic libraries, and compilers for other languages. It is the lingua franca of systems programming, the foundation on which much of modern computing rests.

This endurance is not accidental. C gives you direct control over memory layout, representation, and lifetime without imposing a runtime system or garbage collector. It compiles to efficient machine code on virtually every architecture in use today. Its standard library is small enough to understand completely yet powerful enough for serious work. When performance matters, when resource constraints are tight, when you need to speak the same language as the hardware, C remains unmatched.

That power comes with responsibility. C trusts the programmer to reason correctly about memory boundaries, evaluation order, type compatibility, and concurrency. It does not prevent you from writing incorrect code; indeed, it provides very few guardrails at all. When you violate its rules, the standard says only that the behavior is undefined, which means the compiler is free to do anything: produce wrong results silently, crash unpredictably, or generate optimized code that breaks your assumptions in subtle ways. Understanding what those rules are, and why they exist, is the central task of this book.

What This Book Covers

The book is organized into a progression from foundations through professional practice:

Part I establishes the mental model for how C programs are built and executed. You will learn about the compilation pipeline, translation units,

linkage, and the lexical structure of the language. These concepts underpin everything that follows; without them, errors in linking or preprocessing are mysterious rather than explainable.

Part II dives into types, declarations, and expressions. This is where C reveals its depth: integer promotions and conversions follow precise rules that determine whether your arithmetic does what you expect; sequence and evaluation rules govern the order in which side effects occur; undefined behavior lurks behind common mistakes like signed integer overflow and out-of-bounds pointer arithmetic. You will learn these rules systematically, not as trivia but as tools for reasoning about correctness.

Part III covers functions, scope, storage duration, and linkage — the mechanisms that let you organize code into modules. You will see how to design clean APIs with proper encapsulation, error handling, and const-correctness, and how separate compilation actually works under the hood.

Part IV is where pointers and memory take center stage. Pointers are both C's most powerful feature and its most dangerous. You will learn pointer arithmetic from first principles, understand the relationship between arrays and pointers, master function pointers and callbacks, and develop a rigorous mental model for how C views memory: object representation, alignment, padding, effective type, strict aliasing, and lifetime. This part also covers dynamic memory management in depth, including ownership conventions and patterns for avoiding leaks, double-frees, and use-after-free errors.

Part V brings together structures, unions, enumerations, and bit fields into practical data structures and generic programming techniques. You will implement dynamic arrays, linked lists, hash tables, and other fundamental components as complete, reusable modules. You will also master the preprocessor — macros as both tool and hazard — and learn professional header design practices that keep build times reasonable and dependencies under control.

Parts VI and VII cover the standard library comprehensively: formatted I/O, file handling, string operations, time and date facilities, mathematics, character classification, and more. You will see not just how to call these functions but when each one is appropriate, what guarantees they provide, and what pitfalls to avoid.

The final parts address concurrency through C11's standardized threading and atomic facilities, undefined behavior as a systematic topic, and professional engineering practices including security vulnerabilities, portability hazards,

debugging tools, testing strategies, profiling, and performance optimization.

Prerequisites and Approach

This book assumes you have written programs in some language before — that you understand variables, loops, conditionals, and functions at a basic level. It does not assume prior C knowledge. We define terminology before relying on it, explain difficult topics from first principles, and avoid hand-waving where the language requires precision.

Every substantial code example is complete and compilable. You will see required headers, error handling, cleanup code, and a main function when applicable. Each example identifies the minimum C standard version it requires and provides representative compiler commands using GCC and Clang with strong warning flags. The examples are not isolated fragments pulled from thin air; they build progressively toward realistic programs involving strings, dynamic data structures, parsers, file I/O, modular libraries, callbacks, concurrency, and other systems-programming tasks.

The tone is technical but approachable. Where something is subtle or dangerous, we say so explicitly. Where the standard leaves room for implementation-defined behavior, we tell you how to write portable code that does not depend on a particular compiler's choices. Where expert opinion differs, we present the competing views fairly and explain the reasoning behind each position.

Standards Coverage

We work primarily with ISO/IEC 9899 — the official C standard maintained by WG14 of the International Organization for Standardization. The book covers:

- C89/C90 (ISO/IEC 9899:1990): the first official standard, still relevant for understanding legacy code and maximum portability
- C99 (ISO/IEC 9899:1999): added features like designated initializers, flexible array members, variable-length arrays, inline functions, and new integer types
- C11 (ISO/IEC 9899:2011): introduced threading support, atomics, generic selections, alignment control, and improved Unicode support

- C17 (ISO/IEC 9899:2018): defect fixes only, no new language features
- C23 (ISO/IEC 9899:2024): the current standard, adding auto type inference, nullptr, attributes, bit-precise integers, and mandating two's complement representation

We distinguish carefully between features that are universally portable across all conforming implementations and those that are version-specific or optional. Compiler extensions — particularly from GCC and Clang — are mentioned where they are widely used in practice, but we always make clear when you are leaving the standard behind.

How to Use This Book

Read it sequentially if you are new to C; the chapters build on each other deliberately. If you already know C but want deeper understanding, use it as a reference: jump to the chapter on the topic you care about — memory model, undefined behavior, concurrency, API design — and read that section thoroughly. The code examples serve both purposes: beginners can study them as templates for their own programs; experienced readers can check whether their assumptions about edge cases match what the standard actually says.

Throughout, pay attention to three recurring themes: correctness first (code that works reliably is more valuable than code that runs fast but sometimes produces wrong answers), explicitness over cleverness (clear code with documented assumptions beats obscure tricks every time), and defense in depth (use compiler warnings, static analysis, sanitizers, assertions, and testing together rather than relying on any single tool).

C is a mature language with decades of accumulated wisdom about what works and what does not. This book distills that wisdom into a coherent narrative, grounded in the standard itself and tested against real-world experience. Let us begin.

Chapter 1: The C Language and Its Standards

Why C Still Matters

Consider this fact: if you are reading these words on any modern computer or mobile device, C is almost certainly involved somewhere in the chain that brought them to your screen. The operating system kernel was written in C. The browser rendering engine contains millions of lines of C and C++. The compiler that built those programs may itself be written in C. If you are using a smartphone, its firmware, drivers, and core services run C code. In embedded systems, C dominates almost completely; microcontrollers powering everything from medical devices to industrial controllers execute C-compiled binaries.

This is not nostalgia or inertia. C persists because it occupies a unique position in the programming language landscape: it gives you fine-grained control over memory and representation while remaining portable across architectures and compilers. Higher-level languages like Python, Java, or Rust abstract away details that are often convenient to ignore but sometimes essential to control. Lower-level assembly languages give complete control but sacrifice portability and productivity entirely. C sits between them — close enough to the metal for performance-critical work, high enough to express algorithms clearly and move code between platforms with minimal changes.

Specifically, C remains indispensable in several domains:

Operating systems and kernels. The Linux kernel, FreeBSD, Windows NT derivatives, and macOS are all primarily written in C. The language's ability to map directly to hardware concepts — memory addresses, registers, interrupt handlers — makes it the natural choice for code that manages physical resources.

Embedded and real-time systems. Microcontrollers with kilobytes of RAM and no operating system run C code because it produces small, predictable binaries without runtime overhead. Safety-critical standards like MISRA C and

AUTOSAR C are built on ISO C precisely because aerospace, automotive, and medical industries need deterministic behavior.

Performance-critical libraries. When every nanosecond counts – in cryptographic operations, numerical linear algebra, compression algorithms, or database query execution – C's predictability and efficiency are unmatched. Libraries like OpenSSL, SQLite, zlib, and the GNU C Library itself are written in C.

Language runtimes and toolchains. The garbage collectors, virtual machines, and JIT compilers that power higher-level languages are themselves typically implemented in C. GCC, Clang/LLVM, and many other compilers have substantial C components.

Interoperability. C has become the de facto foreign function interface lingua franca. When Python needs to call C++ code, or Java needs to access native libraries, or Rust needs to integrate with existing C APIs, the boundary is usually expressed in C terms. The language's simple calling conventions and well-defined type system make it a stable bridge between ecosystems.

None of this means C is easy to use correctly. It demands discipline and understanding. But for work where performance, control, or portability cannot be compromised, no alternative offers the same combination of capabilities.

The ISO/ANSI Standardization Story

Before standardization, C existed in multiple incompatible dialects. The original language was documented informally in the 1978 book "The C Programming Language" by Brian Kernighan and Dennis Ritchie – known as K&R C. As C spread across different Unix variants and hardware platforms, compiler vendors added their own extensions and made independent decisions about edge cases. Code that compiled on one system might fail or behave differently on another.

In 1983 the American National Standards Institute formed committee X3J11 to create an official specification. After six years of work, the result was published in December 1989 as ANSI X3.159-1989, commonly called C89 or ANSI C. The same standard was adopted by ISO/IEC with only formatting changes as ISO/IEC 9899:1990, hence the alternate name C90. For practical purposes C89 and C90 are identical; they refer to the same language.

C89/C90 established several foundations that persist today:

- A complete grammar for the language
- A standard library with headers like stdio.h, stdlib.h, string.h, math.h
- Precise rules for type conversions, integer promotions, and expression evaluation
- Formal definitions of undefined behavior, implementation-defined behavior, and unspecified behavior
- The framework for separate compilation via translation units and linkage

The first amendment arrived in 1995 as ISO/IEC 9899/AMD1:1995 (sometimes called C94 or C95), adding wide character support and internationalization facilities. This was a relatively minor update compared to what came next.

C99, published in 1999 as ISO/IEC 9899:1999, was the first major revision. It added features that have become standard practice:

- New integer types (int8_t, int16_t, int32_t, int64_t via stdint.h)
- The long long type for guaranteed 64-bit integers
- Boolean type (_Bool) and complex number support
- Variable-length arrays (VLAs)
- Designated initializers for structs and arrays
- Flexible array members in structures
- Inline functions
- Single-line comments using //
- Compound literals
- Varargs macros
- Restrict qualifier for aliasing hints

C99 was controversial. Some committee members felt it changed too much too fast; Microsoft, in particular, was slow to adopt many C99 features and continues to have incomplete support decades later. The standard was withdrawn in 2011 when superseded by C11 but remains widely used as a baseline for portable code.

C11, published on December 8, 2011 as ISO/IEC 9899:2011, focused on modernization and safety:

- Multi-threading support via threads.h and stdatomic.h
- Type-generic macros using _Generic
- Alignment control with _Alignas and _Alignof

- Static assertions via _Static_assert
- Improved Unicode support (char16_t, char32_t, UTF-8 string literals)
- Anonymous structures and unions
- The noreturn function specifier
- Removal of the unsafe gets() function

Notably, C11 made several features optional rather than mandatory: threading support, atomics, VLAs, and complex numbers are all permitted but not required. This was a pragmatic concession to embedded implementations with severe resource constraints, but it complicated portability since code using these features must now check for their availability.

C17 (ISO/IEC 9899:2018, published June 2018) introduced no new language features. It was purely a defect-fix release, incorporating corrections and clarifications identified since C11. The only practical difference between compiling as C11 versus C17 is the value of the predefined macro **STDC_VERSION**.

C23 (ISO/IEC 9899:2024, published October 31, 2024) is the current standard and represents the most significant revision since C99. Key additions include:

- The auto keyword for type inference in object declarations
- nullptr as a proper null pointer constant with its own type nullptr_t
- Bit-precise integer types via _BitInt(N)
- typeof and typeof_unqual for generic macro programming
- Standardized attributes: [[fallthrough]], [[nodiscard]], [[maybe_unused]], [[deprecated]], [[noreturn]]
- Binary integer literals (0b prefix) and digit separators (apostrophes)
- Empty initializer syntax ={}
- Preprocessor improvements: #elifdef, #elifndef, #embed, #warning
- Library additions: strdup(), strndup(), memset_explicit(), stdbit.h utilities
- Mandatory two's complement representation for signed integers

C23 also removed several obsolete features: trigraph sequences and K&R-style function declarations are no longer part of the language. The standard now reflects forty years of accumulated experience about what actually works in practice.

Standards, Implementations, and Extensions

The ISO C standard defines a specification, not an implementation. A conforming C compiler must implement all mandatory features correctly and may optionally support additional features. In practice, the major compilers – GCC, Clang, and MSVC – all support substantial superset of the standard through extensions.

GCC (GNU Compiler Collection) has historically been the reference implementation for many years. It supports all ISO standards from C89 through C23 via command-line flags like -std=c17 or -std=c23. By default it enables GNU extensions in addition to the selected standard; use -std=c17 without the gnu prefix for strict standard compliance. GCC provides numerous useful extensions: statement expressions, typeof, **attribute**, inline assembly, and many others that have become de facto standards in their own right across Unix-like systems.

Clang/LLVM aims for compatibility with both the ISO standard and GCC extensions. It generally tracks GCC's extension set closely while maintaining its own diagnostic quality and optimization technology. Clang is notable for its sanitizers (ASan, UBSan, TSan, MSan) which have become essential development tools even when using GCC for production builds.

Microsoft Visual C++ (MSVC) has historically lagged behind ISO standards adoption but improved significantly starting with Visual Studio 2019. It now supports most C11 and C17 features but remains incomplete on C23. MSVC uses its own extension syntax (__declspec, __forceinline) that differs from GCC/Clang conventions.

When should you use compiler extensions? The answer depends on your project:

- For maximum portability (code intended to compile everywhere): avoid extensions entirely; stick to the lowest ISO standard your target compilers support
- For open-source libraries targeting multiple platforms: prefer standard C, but conditionally enable extensions where they provide clear benefits using #ifdef guards
- For single-platform projects or internal tools: extensions are often reasonable if they improve safety, performance, or clarity

- For systems programming on Linux/Unix: many GCC extensions have become so ubiquitous that avoiding them entirely is impractical

The key principle is awareness. When you use an extension, document it, guard it with preprocessor conditionals, and understand what you are sacrificing in portability. Never mistake a compiler's willingness to accept non-standard code for correctness.

Portability as a First-Class Concern

Portability in C means something specific: your program behaves identically (or within explicitly documented tolerances) when compiled with different conforming compilers on different platforms. This is not guaranteed automatically; the standard deliberately leaves many details implementation-defined or unspecified to allow optimizations for particular architectures.

The standard distinguishes four categories of behavior:

Defined behavior: The standard fully specifies what must happen. Your code relies only on defined behavior when it is maximally portable. Examples include the value of 2 + 3 being 5, or sizeof(char) being 1.

Implementation-defined behavior: The implementation (compiler plus library) must choose a specific behavior and document it. Different implementations may make different choices. Examples include whether char is signed or unsigned by default, the exact width of int, or the result of right-shifting a negative number. Portable code either avoids depending on these or queries them at compile time.

Unspecified behavior: The standard permits multiple behaviors and does not require the implementation to document which it chooses. Examples include the order in which function arguments are evaluated, or whether an array index out of bounds by one happens to read adjacent memory or crashes. Portable code must never depend on unspecified behavior.

Undefined behavior: The standard imposes no requirements whatsoever. The compiler may generate any code it likes, including code that appears correct during testing but fails catastrophically in production under optimization. Examples include signed integer overflow, null pointer dereference, and use of uninitialized variables. Undefined behavior is not a bug category; it is the absence of any contract between your code and the compiler.

Writing portable C means understanding these categories and designing your code accordingly. Use fixed-width types from stdint.h instead of assuming int is 32 bits. Check sizeof results rather than assuming struct layouts. Never rely on argument evaluation order. Treat undefined behavior not as "probably works" but as a complete breakdown of the language contract.

The abstraction barrier is central to this philosophy. C provides layers of abstraction — types, functions, translation units, standard library interfaces — that let you write code without knowing implementation details. Respect those barriers: do not reach around type systems with pointer casts unless you understand the aliasing rules; do not depend on internal layout of standard library types; do not assume anything about memory addresses beyond what the standard guarantees.

How to Read This Book

Throughout this book, code examples follow a consistent convention. Each substantial example includes:

- Required headers and all necessary declarations
- A complete main() function when applicable, or clearly marked interface/implementation separation
- Error handling for system calls, memory allocation, and I/O operations
- Proper cleanup to avoid resource leaks
- Comments explaining non-obvious decisions

Examples are tagged with their minimum C standard requirement. These appear as plain text markers:

Requires: C99 Requires: C11 Requires: C23

When an example uses a feature available only in a specific standard version or later, the tag makes this explicit. For examples that work in all standards back to C89/C90, no tag appears.

Compiler commands are provided for building examples. The baseline command uses GCC or Clang with strong warning flags:

```
gcc -std=c17 -Wall -Wextra -Wpedantic -Werror example.c -o example
clang -std=c17 -Wall -Wextra -Wpedantic -Werror example.c -o example
```

The -std flag selects the target standard; -Wall enables common warnings; -Wextra adds additional checks; -Wpedantic diagnoses non-standard extensions; -Werror treats all warnings as errors. This combination catches most mistakes at compile time and is recommended for serious development. Some examples may require additional flags like -lm for math library linking or -pthread for threading support, which will be noted where needed.

Technical claims about language rules are supported by citations to the ISO standard sections, WG14 defect reports, compiler documentation, or authoritative references. Inline citations appear as numbered references [n] pointing to the References section at the end of the book. When sources disagree or the standard is ambiguous, we note the conflict explicitly rather than papering over it.

The book occasionally uses callout boxes for best practices and common pitfalls. These are distilled from decades of collective experience in the C community: patterns that have proven reliable across many projects, and mistakes that keep appearing despite being well-documented. Treat them as guidance rather than dogma; understand the reasoning behind each recommendation so you can adapt it to your own context.

Now that we understand what C is and why its standards matter, let us examine how C programs are actually built from source code into executable programs. The compilation model shapes everything about how we organize and write C code.

Chapter 2: The Compilation Model and Program Structure

From Source to Executable

When you compile a C program, the toolchain transforms your text files into an executable binary through a series of conceptual stages. The ISO C standard describes eight translation phases that any conforming implementation must follow, even if it folds multiple phases together internally for efficiency. Understanding these phases is essential because many common bugs — missing includes, macro expansion errors, linker failures — originate from misunderstandings about what happens at each stage.

The eight phases are:

Phase 1: Physical source file multibyte characters are converted to the implementation's source character set. Trigraph sequences (like ??= for #) are replaced with their corresponding single characters. This phase handles the encoding of your source file before any parsing occurs. Note that trigraphs were removed in C23, so modern code no longer needs to worry about them.

Phase 2: Each pair of backslash-newline is removed, splicing physical source lines together into logical lines. Then the resulting character sequence is divided into preprocessing tokens and sequences of white-space characters including comments. A preprocessing token is either a header name, identifier, punctuator, number literal, character constant, string literal, or non-standard preprocessing token. Comments at this stage are recognized but not yet removed.

Phase 3: Preprocessing directives are executed, macros are expanded, and _Pragma operator expressions are evaluated. Directives like #include, #define, #ifdef, and #if are processed here. This is where header files get inserted into your source, macro definitions are substituted throughout the code, and conditional compilation blocks are either included or discarded based on defined macros.

Phase 4: Each character constant and string literal is translated from source encoding to the execution character set. Universal character names (like \u0041) are converted to their corresponding characters. Escape sequences like \n, \t, and \x41 are replaced with their actual character values. This phase determines what bytes actually end up in your string literals at runtime.

Phase 5: Adjacent string literal tokens are concatenated into a single string literal. This means “Hello” “ “ “World” becomes the single string literal “Hello World” before compilation proper begins. This feature is commonly used to break long strings across multiple lines or to conditionally include parts of a string via macros.

Phase 6: White-space characters separating tokens are no longer significant and may be discarded. Comments are replaced with single space characters. The result is a sequence of tokens that form the translation unit proper, ready for compilation.

Phase 7: The translation unit is compiled. Tokens are syntactically and semantically analyzed according to the language grammar, type checking occurs, and the compiler generates an intermediate representation or directly produces assembly code. This is where most compile-time errors are detected: syntax errors, type mismatches, undeclared identifiers, constraint violations.

Phase 8: Translation units and library components are collected and linked into a program image. External references are resolved — each function call must find its definition, each global variable reference must locate its storage. The linker combines object files and libraries into a single executable or shared library.

While the standard describes these as separate phases, modern compilers typically implement them in a more integrated pipeline. GCC and Clang, for example, combine phases 1-6 into their preprocessor component (cpp), perform compilation proper in the main compiler (cc1), optionally run an assembler (as), and invoke a linker (ld). You can observe individual phases by using specific flags:

```
gcc -E source.c         # Phases 1-6 only: output preprocessed code
gcc -S source.c          # Phases 1-7: output assembly language
gcc -c source.c          # Phases 1-7 with assembly: output object file
gcc source.c -o program  # All phases: produce executable
```

Being able to inspect intermediate output is invaluable for debugging preprocessing issues, understanding optimization decisions, or diagnosing linking

problems. When a macro expands unexpectedly, running gcc -E and examining the output reveals exactly what the compiler sees after expansion. When assembly output shows surprising instruction sequences, it may reveal optimization effects of undefined behavior or alignment assumptions.

Translation Units and Compilation

A translation unit is the fundamental unit of compilation in C. It consists of a single source file after all preprocessing has been applied — that is, after #include directives have inserted header contents, macros have been expanded, and conditional compilation has selected which code to include. Each .c file you compile produces one translation unit, regardless of how many headers it includes or how many other files' content ends up in the preprocessed result.

The concept of translation units underpins separate compilation, C's mechanism for managing large programs. Instead of compiling an entire project as one monolithic unit, you compile each source file independently into an object file (typically with a .o extension), then link all object files together. This approach provides several benefits:

Incremental builds: When you modify one source file, only that translation unit needs recompilation. The rest of the project can reuse existing object files, dramatically reducing build times for large projects.

Modular organization: Different source files can implement different components of your system, each compiled independently. This separation supports clean architectural boundaries and enables team collaboration on large codebases.

Library creation: Object files can be archived into static libraries (.a files) or packaged as shared libraries (.so or .dll files), allowing code reuse across multiple programs without recompilation.

The tradeoff is that separate compilation requires careful management of declarations and definitions across translation unit boundaries. Functions and variables used by multiple translation units must be declared in headers that each translation unit includes. The rules governing these cross-translation-unit references — linkage, the one-definition rule for functions, and declaration consistency — are critical to understanding how C programs are structured.

Consider a simple project with two source files:

```c
/* math_ops.h */
#ifndef MATH_OPS_H
#define MATH_OPS_H

int add(int a, int b);
int multiply(int a, int b);

#endif /* MATH_OPS_H */

/* math_ops.c */
#include "math_ops.h"

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

int multiply(int a, int b) {
    return a * b;
}

/* main.c */
#include <stdio.h>
#include "math_ops.h"

int main(void) {
    int x = add(3, 4);
    int y = multiply(x, 2);
    printf("Result: %d\n", y);
    return 0;
}
```

Requires: C90 Compile with: `gcc -std=c17 -Wall -Wextra -Wpedantic main.c math_ops.c -o program`

Here we have two translation units: main.c (which includes math_ops.h during preprocessing) and math_ops.c (which also includes math_ops.h). The header file provides declarations that both translation units need. When compiled separately with gcc -c, each produces an object file containing its own code and data. The linker then resolves the references: main.o refers to add() and multiply(), which are defined in math_ops.o.

The include guard (the #ifndef/#define/#endif pattern) prevents problems when a header is included multiple times – directly or indirectly through other headers. Without it, including math_ops.h twice would produce duplicate declarations that the compiler rejects. Header guards are mandatory best practice for every header file; we will discuss their design in more detail later.

The one-definition rule in C states that each function and each object with external linkage must have exactly one definition across all translation units combined. You can declare a function many times (in headers included by multiple files) but define it only once. Violating this rule produces linker errors about multiple definitions. Objects (variables) follow similar rules, though the details differ slightly between C and C++.

Linkage and Symbol Resolution

Linkage determines whether an identifier refers to the same entity across different translation units. The C standard defines three categories:

External linkage: Identifiers with external linkage refer to the same entity everywhere they appear in the program. A function defined at file scope (not inside another function) has external linkage by default, as does a variable defined at file scope without the static keyword. This is what allows main.c to call add() defined in math_ops.c – both translation units' references to add resolve to the same function.

Internal linkage: Identifiers with internal linkage are visible only within their own translation unit. Adding the static keyword to a file-scope function or variable gives it internal linkage. Each translation unit can have its own static function named helper() without conflict, because the linker treats them as separate entities. Internal linkage is essential for encapsulation: functions that are implementation details of a module should not be visible outside that module's translation unit.

No linkage: Identifiers declared inside a block (within curly braces) have no linkage – they are local to that block and invisible elsewhere. Function parameters also have no linkage. Each declaration of a local variable creates a distinct entity, even if multiple blocks use the same name.

Consider this example illustrating all three categories:

```c
/* module_a.c */
#include <stdio.h>

int global_var = 10;              /* external linkage */
static int internal_var = 20;     /* internal linkage */

void shared_function(void) {      /* external linkage */
    printf("shared_function\n");
}

static void private_helper(void) { /* internal linkage */
    printf("private_helper in module_a\n");
}

void test_module_a(void) {
    int local = 30;               /* no linkage */
    printf("global=%d internal=%d local=%d\n", global_var, internal_var,
    ↪  local);
    private_helper();
}

/* module_b.c */
#include <stdio.h>

extern int global_var;            /* refers to same entity as in module_a.c */

/* This is a DIFFERENT internal_var — no conflict with module_a's version */
static int internal_var = 99;

void test_module_b(void) {
    printf("global=%d internal=%d\n", global_var, internal_var);
    shared_function();            /* calls module_a's function */
    /* private_helper();          ERROR: not visible here — internal linkage */
}

/* main.c */
#include <stdio.h>

extern void test_module_a(void);
extern void test_module_b(void);

int main(void) {
    test_module_a();
    test_module_b();
    return 0;
}
```

Requires: C90 Compile with: `gcc -std=c17 -Wall -Wextra -Wpedantic module_a.c module_b.c main.c -o linkage_demo`

Output:

```
global=10 internal=20 local=30
private_helper in module_a
global=10 internal=99
shared_function
```

This demonstrates several key points. First, global_var has external linkage so both modules access the same variable. Second, each module's internal_var is independent because static gives them internal linkage. Third, shared_function is callable from both modules while private_helper is confined to module_a.c. Fourth, local variables in functions have no linkage and are completely independent of anything at file scope.

The extern keyword declares that an identifier with external linkage is defined elsewhere. In module_b.c, the line extern int global_var tells the compiler "this variable exists, but its storage is allocated in another translation unit." The linker resolves this reference when combining object files. Without such a declaration, using global_var in module_b would generate an implicit declaration warning (or error with -Werror) because the compiler would not know its type.

Strong and weak symbols are concepts from linkers rather than the C standard itself, but they matter in practice. By default, every function definition and initialized data definition is a strong symbol – the linker expects exactly one definition. Multiple strong symbols with the same name produce an error. Some compilers support weak symbols via **attribute**((weak)) (GCC/Clang) or #pragma comment(linker, ...) (MSVC), which allow multiple definitions where one "wins" if conflicts arise. Weak symbols are useful for providing default implementations that can be overridden, but they are non-standard and should be used deliberately with clear documentation.

The Preprocessing Pipeline

The C preprocessor operates during translation phase 3 and is responsible for text transformations before the compiler proper sees your code. Understanding preprocessing is critical because macros are powerful but dangerous: they perform blind textual substitution without understanding types or semantics, which can produce subtle bugs that are hard to diagnose.

Preprocessing directives begin with # at the start of a logical line (after whitespace). The major categories are:

File inclusion: #include inserts the contents of another file into your source at that point. There are two forms: #include “file.h” searches for file.h first in the directory of the current file, then in standard include directories; #include <file.h> searches only standard system include directories. Use quotes for your own headers and angle brackets for system or third-party headers – this convention helps tools distinguish project files from external dependencies.

Macro definition: #define creates a macro that will be substituted throughout subsequent code. Object-like macros behave like constants: #define MAX_SIZE 1024 causes every occurrence of MAX_SIZE to be replaced with 1024. Function-like macros take arguments: #define SQUARE(x) ((x) * (x)) substitutes the argument wherever x appears in the body. We will examine macro pitfalls extensively later; for now note the careful parenthesization – without it, SQUARE(a + b) would expand to a + b * a + b, which is not what you want.

Macro undefinition: #undef removes a previous macro definition, preventing further substitution.

Conditional compilation: #if, #ifdef, #ifndef, #elif, #else, and #endif allow selective inclusion of code based on whether macros are defined or their values. C23 adds #elifdef and #elifndef as shorthand for common patterns. Common uses include: guarding platform-specific code (#ifdef _WIN32), enabling debug builds (#ifdef DEBUG), and checking for feature availability (#if **STDC_VERSION** >= 201112L).

Line control: #line changes the apparent filename and line number for subsequent compiler diagnostics, useful for code generators.

Pragma directives: #pragma provides implementation-specific instructions to the compiler. The standard defines very few pragmas; most are compiler extensions. Examples include #pragma once as an alternative to traditional include guards (supported by most but not all compilers), and optimization hints like #pragma GCC optimize.

Predefined macros provide information about the compilation environment. Important ones include:

- **STDC**: Defined as 1 in standard-conforming modes

- **STDC_VERSION**: Numeric macro indicating the C standard version (199409L for C94, 199901L for C99, 201112L for C11, 201710L for C17, 202311L for C23)
- **GNUC**, **clang**, _MSC_VER: Identify the compiler
- **LINE**, **FILE**, **func**: Current line number, filename, and function name
- **DATE**, **TIME**: Compilation date and time

Consider a practical example using conditional compilation for cross-platform code:

```c
/* platform_utils.h */
#ifndef PLATFORM_UTILS_H
#define PLATFORM_UTILS_H

#if defined(_WIN32) || defined(_WIN64)
    #include <windows.h>
    #define PATH_SEPARATOR '\\'
    #define LINE_ENDING "\r\n"
#elif defined(__unix__) || defined(__APPLE__)
    #include <unistd.h>
    #include <dirent.h>
    #define PATH_SEPARATOR '/'
    #define LINE_ENDING "\n"
#else
    #error "Unsupported platform"
#endif

#if __STDC_VERSION__ >= 201112L
    #include <stdatomic.h>
    #define HAS_ATOMICS 1
#else
    #define HAS_ATOMICS 0
#endif

#endif /* PLATFORM_UTILS_H */
```

Requires: C99 (for // comments and **STDC_VERSION** usage pattern)

This header adapts to the target platform at compile time, selecting appropriate system headers and defining platform-specific macros. It also checks for C11 atomics support, allowing code that uses them to be conditionally compiled. The #error directive causes compilation to fail on unsupported platforms rather than silently producing broken code — a defensive practice worth adopting.

Building with Modern Toolchains

Compiling C programs involves more than invoking gcc or clang on individual files. Real projects use build systems to manage dependencies, configuration, and the sequence of operations needed to produce working executables and libraries. Understanding the basic commands and tools is essential even when using higher-level build systems.

The fundamental GCC/Clang compilation command follows this pattern:

```
compiler [options] source_files [libraries] -o output_file
```

Key option categories:

Standard selection:

```
-std=c17    Select C17 standard (recommended baseline for new projects)
-std=c23    Select C23 standard (use when targeting modern compilers)
-std=gnu17  C17 plus GNU extensions (GCC default behavior)
```

Warning flags (always use these in development):

```
-Wall       Enable common warnings
-Wextra     Enable additional warnings beyond -Wall
-Wpedantic  Diagnose non-standard extensions
-Werror     Treat warnings as errors
-Wconversion Warn about implicit type conversions that may lose data
-Wshadow    Warn when a local variable shadows an outer declaration
```

Optimization:

```
-O0         No optimization (fastest compilation, easiest debugging)
-O1         Basic optimization
-O2         Good optimization without aggressive transformations (recommended
↪  for release)
-O3         Aggressive optimization (may increase code size)
-Ofast      Aggressive optimization including some non-standard FP behavior
```

Debugging:

```
-g          Include debug information for gdb
-ggdb       Include extra debug information optimized for gdb
```

Security hardening:

```
-fstack-protector-strong  Stack buffer overflow detection
-D_FORTIFY_SOURCE=2       Runtime checks on common vulnerable functions
-fPIE -pie                Position-independent executable (ASLR support)
```

Sanitizers (development only, significant runtime overhead):

```
-fsanitize=address         Detect buffer overflows and use-after-free (ASan)
-fsanitize=undefined       Detect undefined behavior (UBSan)
-fsanitize=thread          Detect data races (TSan)
-fsanitize=memory          Detect uninitialized memory reads (MSan, Clang only)
```

A recommended development build command:

```
gcc -std=c17 -Wall -Wextra -Wpedantic -Werror -g -O0 \
    -fsanitize=address,undefined \
    source.c -o program
```

And a release build:

```
gcc -std=c17 -Wall -Wextra -Wpedantic -Werror -O2 \
    -fstack-protector-strong -D_FORTIFY_SOURCE=2 \
    -fPIE -pie \
    source.c -o program
```

For projects with multiple source files, compiling each separately and then linking is more efficient:

```
gcc -std=c17 -Wall -Wextra -c module_a.c -o module_a.o
gcc -std=c17 -Wall -Wextra -c module_b.c -o module_b.o
gcc module_a.o module_b.o -o program
```

The -c flag tells the compiler to stop after producing an object file without linking. The final command links all object files into the executable. This pattern scales to hundreds or thousands of source files and is the foundation that build systems automate.

Make is the traditional Unix build tool. A Makefile specifies targets (output files), prerequisites (input files they depend on), and commands to produce each target from its prerequisites:

```
# Makefile for our example project
CC = gcc
CFLAGS = -std=c17 -Wall -Wextra -Wpedantic -Werror -O2

all: program

program: main.o math_ops.o
    $(CC) $(CFLAGS) $^ -o $@

main.o: main.c math_ops.h
    $(CC) $(CFLAGS) -c $< -o $@

math_ops.o: math_ops.c math_ops.h
    $(CC) $(CFLAGS) -c $< -o $@

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

Note the tab character before each command — Make requires tabs, not spaces. The automatic variables $^ (all prerequisites), $@ (target name), and $< (first prerequisite) reduce repetition. Running make builds the program, recompiling only files whose dependencies have changed. Running make clean removes generated files.

For larger or cross-platform projects, CMake has become the de facto standard build system generator. It produces native build files (Makefiles, Ninja files, Visual Studio solutions) from a platform-independent CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.20)
project(MyProject C)

set(CMAKE_C_STANDARD 17)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_C_EXTENSIONS OFF)

add_executable(program main.c math_ops.c)
```

After running cmake to configure the build, you use the generated build system (make, ninja, etc.) to compile. CMake handles dependency tracking, installation targets, testing integration, and cross-compilation configuration — all while remaining portable across platforms and IDEs.

Meson is a newer alternative gaining adoption for its speed and clean syntax. It uses Python-based description files and the Ninja build backend:

```
project('myproject', 'c')
cc = meson.get_compiler('c')
executable('program', 'main.c', 'math_ops.c',
    c_args: ['-std=c17', '-Wall', '-Wextra'])
```

Choose your build system based on project needs: Make for small Unix-centric projects, CMake for cross-platform or complex builds, Meson for modern Python-friendly workflows. Regardless of choice, understand the underlying compilation commands — when something goes wrong, you will need to reason about what the compiler actually did.

Cross-compilation — building code for a different platform than your development machine — is common in embedded systems and toolchain development. It requires a target-specific compiler (like arm-none-eabi-gcc) and often custom configuration of include paths, library paths, and architecture flags. The principles remain the same: translation units are compiled separately, object files are linked together, and the linker produces a binary for the target platform rather than the host.

With the compilation model understood, we can now examine the actual vocabulary of C — the tokens, identifiers, literals, and operators that form every program.

Chapter 3: Lexical Elements and the Basics of Syntax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Tokens and Whitespace

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Identifiers and Naming Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Integer Literals and Numeric Constants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Character and String Literals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Operators and Precedence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 4: Types and Declarations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

The Type System Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Signed and Unsigned Integer Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Floating-Point Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

The void Type and _Bool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Declaration Syntax Decoded

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 5: Expressions, Operators, and Evaluation Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Expression Categories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Sequence Points and Evaluation Order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Integer Conversions and Promotions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Signed Arithmetic, Overflow, and Wraparound

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Pointer Expressions and Comparisons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 6: Control Flow and Statements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Blocks, Scope, and Compound Statements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Conditional Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Iteration and Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Jump Statements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Compound Literals and Designated Initializers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 7: Functions and Modular Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Function Declarations and Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Parameter Passing and Return Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Error Handling Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

API Design Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Building Reusable Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 8: Scope, Storage Duration, and Linkage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Scopes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Storage Durations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Linkage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

The static Keyword Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Thread-Local Storage and _Thread_local

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 9: Pointers and Pointer Arithmetic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

What a Pointer Is

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

The Address-of and Indirection Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Pointers and Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Pointer Arithmetic Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Function Pointers and Callbacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 10: The Memory Model and Object Representation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Objects, Values, and Representation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Alignment and Padding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Strict Aliasing and Type-Based Alias Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Object Lifetime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Pointer Provenance and Validity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 11: Dynamic Memory Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

The Allocation Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Ownership and Responsibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Common Mistakes and Vulnerabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Custom Allocators and Pools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Memory Leak Detection and Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 12: Data Structures and Generic Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Dynamic Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Linked Lists and Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Stacks and Queues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Hash Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Generic Programming Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 13: Structures, Unions, Enumerations, and Bit Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Structure Types and Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Unions and Type Punning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Enumerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Bit Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Design Patterns with Aggregates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 14: The Preprocessor, Macros, and Header Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Macro Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Dangerous Macro Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Header File Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

X-Macros and Metaprogramming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 15: The Standard Library — Core Facilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Diagnostics and Assertions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Integer Types and Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Character Handling and Classification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

String Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Memory Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 16: The Standard Library — I/O, Time, and Mathematics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Formatted Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Formatted Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

File Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Time and Date Facilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Mathematics and Utilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 17: Concurrency — Threads, Atomics, and Synchronization

The C11 Thread Library

Atomic Operations and Memory Ordering

The C Memory Model for Concurrency

Portable Concurrency Alternatives

Chapter 18: Undefined Behavior and Defensive Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

What Undefined Behavior Really Means

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Catalog of Common UB Sources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Unspecified and Implementation-Defined Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Static Analysis and Sanitizers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Defensive Programming Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Chapter 19: Security, Portability, and Professional Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Common Security Vulnerabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Portability Hazards and Solutions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Debugging Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Testing and Quality Assurance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Performance Profiling and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Conclusion: The Craft of Modern C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

A Language That Earns Its Power

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

Principles for Writing Excellent C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

The Future of C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

A Final Word

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theanatomyofansic.