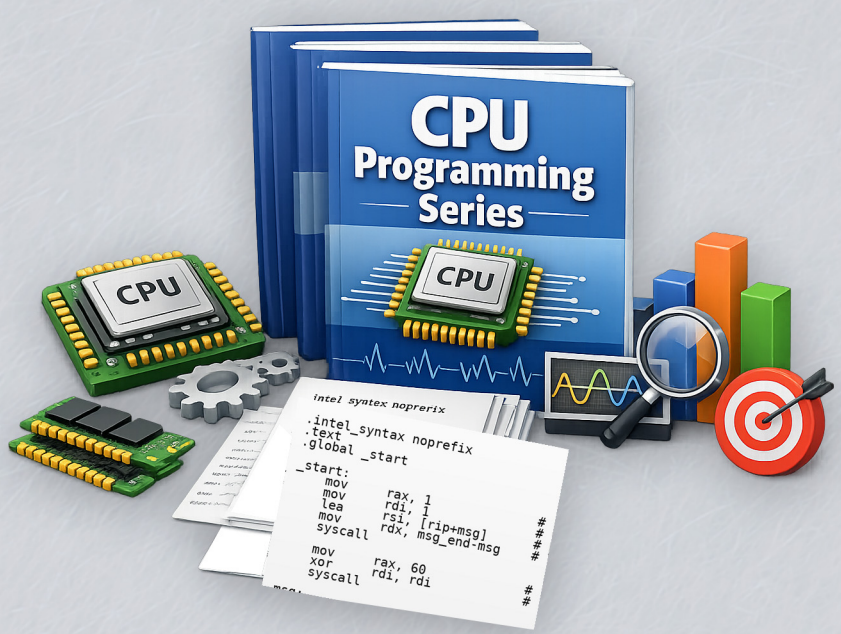
https://simplifycpp.org

CPU Programming Series

The Stack & Calling Conventions

ABI Foundations Across Architectures



3

Prepared by Ayman Alheraki

CPU Programming Series

The Stack & Calling Conventions

ABI Foundations Across Architectures

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Preface — Why This Booklet Exists

The Invisible Machinery Behind Every Function Call

Every function call is more than a jump to a new address. A correct call must establish an agreed-upon execution *contract* between two independently compiled pieces of code: the caller and the callee. That contract defines, at minimum:

- how control returns (where the return address lives and how it is restored),
- where arguments are placed (registers, stack, or a defined mixture),
- where results are delivered (registers, memory, or caller-provided storage),
- which machine state must survive across the call (preserved registers and invariants),
- and what stack invariants are required (layout, alignment, reserved areas, unwind metadata, etc.).

These rules are not "compiler preferences"; they are what makes separately compiled code interoperable. This booklet explains that machinery as a *portable mental model*, so you can reason about execution across architectures without memorizing any one platform's rules.

Why Most Programmers Misunderstand the Stack

Many programmers encounter the stack only as a vague idea: “local variables live there.” That simplification hides the real purpose of the stack in compiled systems:

- The stack is primarily a *call/return structure* that enables nested execution.
- A stack frame is not “just locals”; it is a *structured record* of a call: saved state, bookkeeping, temporaries, outgoing arguments, and space required by the ABI.
- Correctness depends on invariants that are invisible in high-level code (preservation responsibilities, alignment, reserved call areas, and consistent layout for debugging/unwinding).

When these invariants are violated, failures are often non-local: a bug may surface far from its cause, because corruption propagates through later calls.

The Danger of Learning Calling Conventions Too Late

Calling conventions are often treated as “advanced assembly trivia.” In reality, they define the boundary between correctness and undefined behavior for any low-level work:

- Mixed-language projects (C, C++, Rust, Zig, assembly) rely on ABI compliance at every boundary.
- Debugging optimized code requires knowing which values are guaranteed to survive a call and which are not.
- Security-critical failures frequently stem from stack misuse: clobbered return addresses, corrupted frames, and broken unwinding assumptions.

Learning these concepts late usually means learning them *reactively* while chasing hard-to-reproduce bugs. This booklet aims to make the core rules intuitive before you go architecture-specific.

ABI Knowledge as a Prerequisite for Low-Level Mastery

To work confidently near the machine, you must distinguish three layers:

- **Language rules:** what C/C++ guarantees and what it does not.
- **ISA rules:** what an instruction set can do (the instruction semantics).
- **ABI rules:** how binary components agree to *use* the ISA to implement calls, data layout, linkage, and interoperability.

The ABI is the bridge that turns instructions into a stable calling interface. Without ABI awareness, you can read assembly yet still misunderstand what is *guaranteed* versus what is merely an accident of a particular compiler build.

What This Booklet Intentionally Teaches — and What It Does Not

What you will learn here (architecture-neutral)

- The stack as a conceptual structure for nested execution.
- The idea of a stack frame and what it must contain in principle.
- Prologue/epilogue as the mechanism for establishing and restoring call invariants.

- Caller-saved vs callee-saved as a responsibility model, not a register list.
- Argument passing and return values as contract-defined data movement.
- The purpose of an ABI and why interoperability depends on it.
- C/C++ interoperability concepts at the linkage boundary (what makes calls link-compatible).

What you will *not* learn here (by design)

- No ISA-specific calling rules (no platform register lists, no platform-only stack layouts).
- No operating-system syscalls (those are a separate interface with different contracts).
- No "memorize this ABI" tables; those belong in the architecture-specific booklets.

A small illustrative sketch (not a rule set)

The following assembly-shaped sketch exists only to visualize the *roles* of prologue/epilogue and frame structure. Register names and exact sequences differ by architecture and ABI; the *concept* is what matters.

```
# Illustrative-only sketch (Intel-like syntax, conceptual roles).
# Do NOT treat this as a platform rule set.

func:
    # Prologue: establish a stable frame and preserve required state
    push FP              # save old frame pointer (concept)
    mov  FP, SP          # create a new frame base
    sub  SP, N           # reserve frame space (locals/temps/outgoing)
```

```
    # Body: may call other functions (must obey preservation responsibiliti
    # call other_func

    # Epilogue: restore preserved state and tear down the frame
    mov  SP, FP
    pop  FP
    ret                  # return control to the caller
```

C / C++ interoperability at the boundary (concept)

At the binary boundary, the key idea is: *both sides must agree on a single ABI contract*. One common tool is using C linkage for an exported interface surface:

```
extern "C" int api_add(int a, int b);

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

This booklet explains what this kind of boundary accomplishes conceptually (and what it does *not* guarantee), so that later, when you study an architecture-specific ABI, you will understand *why* each rule exists and what breaks when it is violated.

Chapter 1

Why the Stack Exists

1.1 The Stack as an Execution Necessity

At the machine level, a program is a stream of control transfers. As soon as execution becomes *nested* (a function calls another function, which calls another), the machine needs a reliable way to:

- remember **where to return** after each call,
- preserve enough **temporary state** to resume correctly,
- and provide each activation of a function with **private working space**.

The stack is the simplest general mechanism that supports unbounded nesting with constant-time updates: each call *pushes* a new activation record, and each return *pops* it, restoring the previous context.

1.2 Automatic Storage vs Static and Heap

High-level languages expose multiple storage durations. At runtime they map to different lifetime and ownership guarantees:

- **Automatic storage (stack-based lifetime):** objects are created when a scope or function activation begins and are destroyed when that activation ends. This lifetime is naturally modeled by stack frames.
- **Static storage:** objects exist for (roughly) the entire program lifetime. Their memory is not tied to call nesting, so they do not require per-call allocation.
- **Dynamic storage (heap):** objects outlive the scope that created them and require explicit management (manual or via runtime/allocators). The heap supports arbitrary lifetimes but at higher overhead and with more complex failure modes.

The key distinction is not *where* memory lives, but *who controls lifetime*. The stack is optimized for *structured lifetimes* that follow call nesting.

1.3 Why Function Calls Require Structure

A function call is not only "jump to code." It must establish a **binary contract** between caller and callee so that separately compiled code can interoperate. Conceptually, the call boundary must define:

- **control return:** how the return address is recorded and used,
- **state preservation:** which machine state must remain intact across the call,
- **data transfer:** where arguments are placed and where results are returned,

- **frame invariants:** what stack layout/alignment is required for correctness and tooling.

A stack frame is the physical representation of that structure: it packages the information needed to resume execution correctly after nested calls.

1.4 Stack Discipline and Deterministic Execution

The stack enforces a discipline: **last call returns first**. This LIFO rule is what gives deterministic restoration of contexts:

- Each call creates a new, isolated activation record.
- Returns restore the exact previous activation in constant time.
- Resource lifetimes that follow scope nesting become predictable and easy to reason about.

Determinism here means: if call nesting is the same, then the sequence of frame creation/destruction is the same. Violating stack discipline (mismatched call/return structure, corrupted frames, or broken preservation assumptions) breaks determinism and typically yields undefined behavior, often surfacing far from the actual bug.

Conceptual Execution Sketch (Architecture-Neutral)

The following sketch is *illustrative only*. It shows the **roles** a stack frame plays during a call. Exact registers, instructions, and layout differ by ABI and architecture.

```
# Conceptual-only sketch (Intel-like syntax). Not a platform rule set.

caller:
    # prepare arguments (location depends on ABI)
```

```
    call callee          # transfers control and records return point (conc
    # resumes here after callee returns

callee:
    # Prologue: establish frame + preserve required state
    push FP              # save old frame base (concept)
    mov  FP, SP          # new frame base
    sub  SP, N           # reserve frame space (locals/temps/outgoing)

    # ... function body ...
    # may call other functions, must preserve required state

    # Epilogue: restore state + destroy frame
    mov  SP, FP
    pop  FP
    ret                  # return to caller's recorded point
```

What to Take Away

- The stack exists to support **nested execution** with predictable restoration.
- Automatic storage aligns naturally with stack frames because lifetimes are **structured**.
- Function calls require structure because the boundary is an **ABI contract**.
- Stack discipline provides **deterministic context management**; breaking it is catastrophic.

Chapter 2

Conceptual Model of the Stack

2.1 Abstract Stack Behavior (Push / Pop)

At the conceptual level, a stack is an abstract data structure that supports two fundamental operations:

- **push**: create a new element on top of the stack,
- **pop**: remove the most recently created element.

In execution terms, a *push* corresponds to allocating space for a new activation record and recording state needed to resume execution. A *pop* corresponds to deallocating that record and restoring the previously saved state. These operations are logical actions; their concrete implementation varies across architectures and ABIs, but their semantic meaning remains constant.

2.2 LIFO and Control Flow

The defining property of the stack is **Last-In, First-Out (LIFO)** ordering. This ordering is not arbitrary; it mirrors the structure of nested function calls:

- The most recent call must return before its caller can resume.
- Each return restores the immediately preceding execution context.

LIFO ordering guarantees that control flow unwinds in the exact reverse order of call nesting. This property enables correct restoration of return addresses, preserved registers, and local state without global bookkeeping or search. Any deviation from LIFO behavior breaks the correspondence between calls and returns and results in undefined execution.

2.3 Stack Growth Direction (Conceptual, ISA-Independent)

From a conceptual standpoint, a stack grows by *adding* new elements and shrinks by *removing* them. Whether this corresponds to increasing or decreasing memory addresses is an *implementation detail*, not a semantic requirement:

- Some architectures define growth toward lower addresses.
- Others define growth toward higher addresses.
- The ABI defines which convention applies for a given platform.

For reasoning about correctness, the only invariant that matters is that push and pop operations are inverse and consistently ordered. All higher-level reasoning about frames, lifetimes, and call boundaries must remain independent of address direction.

2.4 Logical vs Physical Representation

It is essential to distinguish between the **logical stack** and its **physical realization**:

- The **logical stack** is the abstract sequence of activation records created by nested execution.
- The **physical stack** is a concrete region of memory managed according to ABI rules.

Multiple physical representations can implement the same logical behavior. Optimizing compilers may omit physical stack usage entirely for some functions (inlining, register allocation, or frame elision), while preserving the logical stack semantics required by the language and ABI.

Conceptual Push/Pop Sketch

The following sketch illustrates the *idea* of stack manipulation. It does not define instruction sequences or ABI rules.

```
# Conceptual illustration only (Intel-like syntax).

# push X  -> reserve space and store X
sub SP, 8         # adjust stack pointer (concept)
mov [SP], X       # store value at top of stack

# pop X   -> load value and release space
mov X, [SP]       # load most recent value
add SP, 8         # restore stack pointer
```

Key Invariants

- Push and pop operations are strictly ordered and paired.
- LIFO ordering mirrors call/return structure.
- Growth direction is irrelevant at the conceptual level.
- Logical stack semantics must hold even when no physical stack frame exists.

Understanding this abstraction allows you to reason about execution, debugging, and ABI behavior without tying your mental model to a specific architecture or instruction set.

Chapter 3

Stack Frames

3.1 What a Stack Frame Represents

A **stack frame** (also called an *activation record*) is the per-call execution context created when a function begins and removed when it ends. Conceptually, it is the concrete representation of the call boundary contract between caller and callee. A frame typically provides storage or bookkeeping for:

- **Control linkage:** information needed to return correctly (e.g., a return address and, when used, a link to the caller's frame).
- **State preservation:** saved machine state required by the ABI (preserved registers or equivalent).
- **Automatic objects and temporaries:** storage tied to the function activation's lifetime.
- **Call support:** space for outgoing arguments, spill slots, and ABI-required reserved areas.

- **Tooling metadata (conceptual):** enough structure to support debugging, unwinding, and exception propagation where applicable.

The exact layout and presence of these components is platform- and optimization-dependent, but the *role* of the frame is consistent: it makes nested execution well-defined and mechanically reversible.

3.2 Lifetime of a Stack Frame

A frame's lifetime is precisely the lifetime of one **function activation**:

- **Creation:** conceptually occurs at function entry, when the callee establishes its working context (often via a prologue).
- **Use:** persists across the function body and any nested calls made by the function.
- **Destruction:** conceptually occurs at function exit, when the callee restores invariants and relinquishes its storage (often via an epilogue).

From a language perspective (e.g., C/C++), automatic objects associated with that activation must be treated as valid only within this lifetime. After the frame is destroyed, any reference to its storage is invalid.

3.3 Frame Isolation and Ownership

A frame enforces **isolation** between activations:

- Each call owns its own frame. Its automatic storage and saved state belong to that activation only.

- The caller must not assume it can access or interpret the callee's frame layout; the ABI and compiler control that layout.
- Correctness requires respecting ownership boundaries: data needed after a return must not depend on storage owned by the callee's frame.

Frame isolation is a cornerstone of reliability: it prevents nested calls from trampling each other's working state, and it enables independent compilation. When isolation is violated (stack corruption), failures typically appear non-locally because the damaged state is used later by unrelated code.

3.4 Nested Calls and Frame Stacking

Nested calls naturally form a **stack of frames**:

- When function `A` calls `B`, `B` creates a new frame on top of `A`'s frame.
- If `B` calls `C`, `C`'s frame becomes the top.
- Returns unwind in reverse order: `C` returns to `B`, then `B` returns to `A`.

This LIFO stacking provides constant-time creation and teardown of activation contexts and guarantees deterministic restoration of state assuming ABI invariants are obeyed.

Conceptual Frame Stack Sketch

The following diagram shows the logical picture of nested calls. It is not a platform layout.

```
# Call nesting:
#   A() calls B()
#   B() calls C()
```

```
Top of stack (most recent activation)
+--------------------+
| Frame for C()      |
+--------------------+
| Frame for B()      |
+--------------------+
| Frame for A()      |
+--------------------+
Bottom of stack (older activations)
```

What to Remember

- A frame is the per-activation execution record that makes calls and returns reversible.
- Frame lifetime equals activation lifetime; after return, its storage is invalid.
- Isolation and ownership boundaries prevent state interference and enable independent compilation.
- Nested calls stack frames in LIFO order, producing deterministic unwinding when invariants hold.

Chapter 4

Function Entry and Exit

4.1 Function Boundaries as Execution Contracts

A function boundary is not merely a syntactic construct; it is an **execution contract** enforced at the binary level. This contract defines the obligations of the caller and the callee so that independently compiled code can cooperate correctly. At minimum, the contract specifies:

- how control is transferred to the callee and returned to the caller,
- which parts of machine state must be preserved across the call,
- how arguments are made visible to the callee,
- and how results are delivered back to the caller.

These rules are formalized by the ABI. Neither side is free to deviate: the caller must prepare the call environment correctly, and the callee must restore all required invariants before returning.

4.2 Call as Control Transfer Plus Context Preservation

Conceptually, a call operation performs two logically distinct actions:

- **Control transfer:** execution jumps from the caller to the callee, with a return point recorded so execution can resume.
- **Context preservation:** enough execution state is saved so the caller can continue as if the call were an atomic operation.

The details of how this is achieved (return address handling, register saving, stack adjustment) vary by architecture and ABI, but the semantic goal is invariant: after the call completes, the caller observes a well-defined continuation point with required state intact.

4.3 Return as Controlled State Restoration

A return operation is the inverse of a call. It must restore the execution environment expected by the caller:

- reestablish the caller's stack context,
- restore all state that the callee was responsible for preserving,
- deliver the return value in the ABI-defined location,
- and transfer control back to the recorded return point.

From the caller's perspective, a correctly implemented return makes the callee's internal execution invisible, except for its intended observable effects.

4.4 Symmetry Between Entry and Exit

Correct function execution relies on **symmetry** between entry and exit actions:

- every piece of state saved on entry must be restored on exit,
- every stack adjustment performed in the prologue must be undone in the epilogue,
- invariants established at entry must hold again at exit.

This symmetry ensures that nested calls can be composed arbitrarily without accumulating corruption. As long as each function honors the same contract, the global execution remains consistent and deterministic.

Conceptual Entry/Exit Sketch

The following sketch illustrates the logical symmetry of entry and exit. It is not an ABI specification.

```
# Conceptual illustration only (Intel-like syntax).

func:
    # Entry: establish context
    push FP              # save caller frame base (concept)
    mov  FP, SP          # define new frame
    sub  SP, N           # reserve frame space

    # ... function body ...

    # Exit: restore context
    mov  SP, FP
```

```
pop  FP
ret                    # restore control to caller
```

Key Consequences

- Treating function boundaries as contracts enables independent compilation and linking.
- Calls and returns are structured operations, not free-form jumps.
- Entry/exit symmetry is mandatory; breaking it leads to undefined behavior.
- ABI-defined contracts are what make large systems reliably composable.

Chapter 5

Prologue and Epilogue

5.1 Purpose of the Prologue

The **prologue** is the entry sequence that establishes the callee's execution context so the function can run while still honoring the ABI contract. Conceptually, it exists to:

- **Create a stable frame context:** define where this activation's frame begins and how it is referenced.
- **Reserve stack space:** allocate storage for automatic objects, temporaries, spill slots, and ABI-required areas.
- **Preserve required state:** save any machine state the callee is responsible for preserving across the call (per the calling convention).
- **Establish invariants:** ensure stack alignment and any other ABI invariants needed for nested calls, debugging, and unwinding.

Even when optimizations remove or shrink the visible prologue, the *semantic obligations* remain: the function must behave as if required invariants are satisfied.

5.2 Purpose of the Epilogue

The **epilogue** is the exit sequence that restores the caller-visible state and returns control. Conceptually, it exists to:

- **Undo stack allocation:** release the current frame's storage and reestablish the caller's stack context.
- **Restore preserved state:** reload any state that the callee promised to preserve.
- **Reestablish ABI invariants:** ensure the caller resumes with the expected stack and register conditions.
- **Return control predictably:** transfer execution to the caller's recorded return point.

The epilogue is the prologue's logical inverse. If that symmetry breaks, nested execution becomes unreliable and typically results in undefined behavior.

5.3 Frame Setup and Teardown (Conceptual)

A stack frame is not a fixed template; it is a **structured record** whose contents depend on language needs, ABI rules, and optimization decisions. Conceptually, setup and teardown follow this shape:

- **Setup (entry):** establish a frame reference (explicit or implicit), reserve needed space, save required state, align the stack.
- **Teardown (exit):** restore saved state, deallocate reserved space, restore the prior context, return.

Two important points keep the model correct across architectures:

- The stack may be used without an explicit frame pointer; a compiler can address frame objects relative to the stack pointer when allowed.
- Some functions may use *no* physical frame (e.g., leaf functions that keep everything in registers), but they still obey the calling convention's preservation and alignment rules.

5.4 Why Compilers Must Generate Them

Compilers must generate prologues/epilogues (or equivalent transformations) because they are the mechanism that **enforces the ABI contract** in the emitted machine code. In particular, compilers must ensure:

- **Interoperability:** separately compiled units (and different languages) can call each other reliably.
- **Correctness under optimization:** register allocation, inlining decisions, and stack slot assignment still preserve ABI obligations.
- **Safe nested calls:** a callee can call other functions without breaking alignment or clobbering preserved state.
- **Tooling support:** debugging, stack tracing, and exception/unwind mechanisms require consistent, ABI-compliant frame behavior (even if partially optimized).

In short: prologue/epilogue sequences (or their optimized equivalents) are how a compiler turns the abstract concept of "a function call" into a concrete, verifiable machine-level protocol.

Illustrative Sketch (Concept Only)

The following is a *conceptual* sketch to visualize roles. It is not a platform rule set; exact registers, instructions, and ordering vary by ABI and architecture.

```
# Conceptual prologue/epilogue sketch (Intel-like syntax).
# Not an ABI specification.

func:
    # Prologue: establish frame + reserve space + preserve required state
    push FP              # save caller frame base (concept)
    mov  FP, SP          # establish this frame base
    sub  SP, N           # reserve frame storage (locals/temps/spills)
    # push R_saved        # save callee-preserved registers if required (co

    # ... function body ...
    # may perform nested calls; must keep ABI invariants

    # Epilogue: restore required state + tear down frame
    # pop R_saved         # restore callee-preserved registers if saved (co
    mov  SP, FP
    pop  FP
    ret
```

Practical Invariants to Keep in Mind (Architecture-Neutral)

- **Symmetry:** what is saved/allocated on entry must be restored/deallocated on exit.
- **Preservation rules are contractual:** caller-saved vs callee-saved is not optional.
- **Alignment matters:** the callee must maintain ABI-required stack alignment, especially before making calls.
- **Optimization changes shape, not meaning:** frames may be minimized or elided, but obligations remain.

Chapter 6

Caller-Saved vs Callee-Saved

6.1 The Register Preservation Problem

Registers are the fastest storage available to the CPU, and optimizing compilers try to keep as much live state in registers as possible. The problem is that a function call is a boundary where **two independently compiled regions** interact. When the caller transfers control to the callee, the callee will use registers for its own work. Unless there is an agreement, the callee may overwrite values that the caller still needs after the call.
Therefore, every ABI must answer a precise question:

> Which registers are allowed to be overwritten by the callee, and which registers must survive a call?

Without a rule, correct compilation and separate linking would be impossible, because the caller could not safely keep values in registers across calls.

6.2 Division of Responsibility

ABIs solve this by dividing registers into two conceptual classes (names vary, meaning is stable):

- **Caller-saved (volatile) registers:** the callee may freely clobber them. If the caller needs a value to survive the call, the caller must save it before the call and restore it after.
- **Callee-saved (non-volatile) registers:** the callee must preserve their values. If the callee wants to use them, it must save the incoming values (typically in its prologue) and restore them before returning (typically in its epilogue).

This split creates a stable contract. The caller can keep long-lived values in callee-saved registers across many calls, while the callee can use caller-saved registers for temporary work without paying save/restore overhead on behalf of every caller.

6.3 Why Both Strategies Coexist

A single strategy cannot serve all code shapes efficiently:

- If *everything* were callee-saved, then every function would pay a constant overhead saving/restoring many registers even when callers do not need them preserved.
- If *everything* were caller-saved, then callers would pay repeated save/restore costs around each call, even when values could have been cheaply preserved by the callee once.

Real programs contain both patterns:

- **Many short-lived temporaries:** best served by caller-saved registers (cheap to clobber).

- **Long-lived values across calls:** best served by callee-saved registers (stable across many calls).

Thus, ABIs intentionally include both classes to provide efficient defaults for common compiler allocation strategies.

6.4 Design Trade-offs and Consequences

The caller-saved / callee-saved split is a performance and composability design choice with concrete consequences:

Trade-offs

- **Call-site cost vs function-entry cost:** caller-saved pushes work to call sites; callee-saved pushes work to function prologues/epilogues.
- **Leaf vs non-leaf functions:** leaf functions (no nested calls) often benefit from using caller-saved registers because nothing needs to survive further calls.
- **Register allocation flexibility:** compilers prefer caller-saved registers for short lifetimes and callee-saved registers for values that remain live across calls.
- **Interoperability stability:** the split is part of the ABI contract; violating it breaks mixed-language calls and separate compilation.

Consequences

- **Correctness depends on respecting the contract:** if a callee fails to restore callee-saved registers it used, the caller resumes with corrupted state.

- **Performance depends on code shape:** changing which side saves can shift overhead between call density and function complexity.
- **Debugging optimized code requires this model:** when stepping through assembly, you must know which registers are allowed to change across a call.

Conceptual Sketch (Not a Platform Register List)

This sketch shows the *idea* of each responsibility. It is not an ABI specification and does not define which registers belong to which class on any architecture.

```
# Conceptual illustration only (Intel-like syntax).

caller:
    # If caller needs V (in a caller-saved register) after the call:
    push V              # save before call (concept)
    call callee
    pop  V              # restore after call (concept)

callee:
    # If callee wants to use a callee-saved register S:
    push S              # save incoming value (concept)
    # ... use S freely ...
    pop  S              # restore before return (concept)
    ret
```

Key Rule (Architecture-Neutral)

- **Caller-saved:** caller preserves if needed.
- **Callee-saved:** callee preserves if used.

This single rule is the backbone of calling convention reasoning. The architecture-specific booklets will later map this model to concrete register sets, but the logic never changes.

Chapter 7

Argument Passing (Conceptual View)

7.1 What It Means to Pass an Argument

To **pass an argument** is to make a value (or a reference to a value) available to the callee in a way that both caller and callee understand identically at the binary boundary. Conceptually, argument passing defines:

- **Representation:** how the argument is encoded (bits, width, signedness interpretation, pointer vs value).
- **Location:** where the callee will find it at entry (register(s), stack slot(s), or a defined combination).
- **Binding:** whether the callee receives a copy of the value or receives an address/handle to caller-owned storage.

At the ABI level, arguments are not “magically visible”; they must be placed according to a contract so the callee can reliably decode them.

7.2 Order, Ownership, and Lifetime

Argument passing is fundamentally about **who owns what** and **how long it stays valid**:

- **Order:** the ABI specifies the mapping from source-level parameter list to concrete locations. This mapping must be deterministic so separately compiled units agree.
- **Ownership:**
  - **By-value:** the callee receives a value copy; the callee owns its local copy.
  - **By-reference (pointer/reference semantics):** the callee receives an address/handle; the caller retains ownership of the referenced storage unless a separate contract transfers it.
- **Lifetime:** if the callee receives an address, the pointed-to object must remain valid for the duration implied by the interface contract. Passing pointers to temporary or expired storage creates immediate correctness hazards.

In practice, "argument passing" is inseparable from lifetime discipline: the ABI ensures location and representation, while the language and interface design determine validity.

7.3 Registers vs Stack (Abstract Model)

Modern ABIs typically use both registers and the stack, because they serve different needs:

- **Registers:** used for the most common, small, fixed-size arguments to minimize memory traffic and reduce call overhead.
- **Stack:** used when there are more arguments than available argument registers, when an argument requires stack-based placement by rule, or when the ABI requires a contiguous area for certain call patterns.

Conceptually, you can treat argument placement as a two-stage model:

1. Fill a defined set of *argument locations* (often registers) in a deterministic order.

2. Place remaining arguments into ABI-defined stack slots in a deterministic order.

The exact number of registers and the exact layout are architecture-specific; the *model* remains stable across platforms.

7.4 ABI Enforcement and Predictability

Argument passing rules exist to guarantee:

- **Binary interoperability:** independently compiled modules can call each other correctly.

- **Language interoperability:** C, C++, and other languages can share callable interfaces when they target the same ABI contract.

- **Optimization safety:** compilers can optimize aggressively while still producing call boundaries that obey the same stable rules.

- **Tooling consistency:** debuggers, profilers, and unwind mechanisms can reason about calls when conventions are consistent.

Predictability is the goal: given a function signature under an ABI, the placement and interpretation of arguments must be fully determined. If code violates those rules (wrong prototype, mismatched calling convention, incorrect varargs usage), behavior becomes undefined because the callee reads the wrong locations with the wrong interpretation.

Conceptual Placement Sketch (Not an ABI Specification)

This sketch illustrates the abstract idea: some arguments map to registers, and the rest map to stack slots. It does not define any real register set.

```
# Conceptual-only illustration (Intel-like syntax).
# Not a platform rule set.

# Suppose a call with arguments: a0, a1, a2, a3, a4 ...
# ABI concept:
#   first K arguments -> ARG_REG0..ARG_REG(K-1)
#   remaining         -> stack slots

mov ARG_REG0, a0
mov ARG_REG1, a1
mov ARG_REG2, a2
mov ARG_REG3, a3

# spill extra args to stack (concept)
sub SP, 16
mov [SP+0],  a4
mov [SP+8],  a5

call callee
add SP, 16
```

Key Rules to Carry Forward

- Argument passing is a **binary contract**: representation + location must match.
- Correctness depends on **ownership and lifetime**, especially for address-based

arguments.

- Registers optimize common cases; the stack handles overflow and ABI-required placement.
- ABI rules provide predictability; mismatches typically produce undefined behavior.

Chapter 8

Return Values

8.1 Returning Control vs Returning Data

A function return performs two logically separate actions:

- **Return control:** transfer execution back to the caller at the recorded continuation point, restoring the caller-visible machine state required by the ABI.
- **Return data:** deliver a result (if any) in ABI-defined form and location, so the caller can observe the function's output.

These actions are coupled but distinct. Control return is mandatory for every call; data return is conditional on the function's interface. The ABI defines how both are made mechanically reliable across separately compiled units.

8.2 Scalar and Aggregate Return Concepts

Return values fall into two broad conceptual categories:

Scalar returns

A **scalar** return is a small fixed-size value that can be represented in a single machine-sized unit or a small number of such units (examples: integer, pointer, floating scalar). Conceptually, ABIs typically return scalars via a designated **return location**, most commonly a register (or a small, fixed set of registers).
Key properties:

- scalar returns minimize memory traffic,
- they are fast and predictable,
- and their location is fixed by the calling convention.

Aggregate returns

An **aggregate** return is a composite object (struct/class, large vector-like value, or multiple fields) whose size or shape may not fit a single scalar location. Conceptually, ABIs handle aggregates in one of two ways:

- **Return-in-registers (when small enough):** the ABI may define that certain small aggregates are returned via multiple fixed return locations (multiple registers), following precise packing rules.
- **Return-via-caller-provided storage (indirect return):** the caller provides an address of writable storage, and the callee constructs/writes the result there. This address is an implicit argument governed by the ABI contract.

Which strategy applies is ABI-specific, but the *conceptual distinction* is stable: if the result cannot be reliably and efficiently represented in fixed return locations, a memory-based protocol is used.

8.3 Ownership Rules for Returned Values

Returning a value is inseparable from **lifetime and ownership** rules:

- **By-value return (copy/move semantics at the language level):** the caller receives a value that is independent of the callee's stack frame after return. The result must remain valid after the callee exits.
- **Returning an address (pointer/reference-like return):** the returned address is only valid if it points to storage whose lifetime outlives the call. Returning an address into the callee's stack frame is invalid because that storage ceases to exist when the frame is destroyed.
- **Indirect returns (caller-provided storage):** the caller owns the destination storage; the callee writes/constructs the result into it. Validity is ensured because the storage is not part of the callee's frame.

In short: a correct return protocol must ensure that the caller never depends on storage that is reclaimed when the callee returns.

8.4 ABI Implications

Return value rules exist to guarantee **binary predictability**:

- **Fixed return locations:** the ABI defines where scalars (and some small aggregates) appear at the moment of return.
- **Indirect return protocol:** the ABI defines how a hidden pointer is passed when the result must be produced in memory.

- **Interoperability constraints:** caller and callee must agree on the function's signature and calling convention; otherwise the caller may read the wrong return location or mishandle an indirect return, producing undefined behavior.
- **Optimization constraints:** compilers must preserve the ABI-visible return behavior even when they optimize away temporaries, elide copies, or transform the function body.

For cross-language boundaries (notably C/C++), return type agreement is critical: mismatched declarations change how the ABI expects results to be produced and consumed.

Conceptual Return Sketch (Not an ABI Specification)

The following sketches illustrate the two common conceptual shapes: scalar return via a return register, and aggregate return via caller-provided storage. These are not platform rule sets.

```
# Conceptual scalar return (Intel-like syntax).
# Callee places result in a designated return location (RET_REG).

callee_scalar:
    # ... compute result into RET_REG ...
    mov RET_REG, VALUE
    ret

# Conceptual indirect return for an aggregate.
# Caller provides OUT_PTR (hidden first argument conceptually).
# Callee writes result into [OUT_PTR], then returns normally.

callee_aggregate:
    # OUT_PTR points to caller-owned storage
    mov [OUT_PTR+0], FIELD0
    mov [OUT_PTR+8], FIELD1
    ret
```

Key Rules to Carry Forward

- Returning control restores the caller's execution; returning data delivers the function's result.
- Scalars usually use fixed return locations; aggregates may require multiple locations or indirect return.
- Ownership/lifetime must ensure the caller does not depend on callee-frame storage after return.
- ABI rules make return behavior predictable; signature mismatches typically cause undefined behavior.

Chapter 9

What Is an ABI?

9.1 ABI vs ISA vs Language Specification

To reason correctly about compiled software, you must separate three layers that are often confused:

- **ISA (Instruction Set Architecture):** defines the programmer-visible machine: instructions, registers, memory model at the instruction level, privilege levels, and architectural behavior. It answers: *what the CPU can execute*.
- **Language specification (e.g., C/C++):** defines the meaning of source programs: types, expressions, object lifetimes, undefined behavior, and abstract machine rules. It answers: *what the source code means*, not how it is laid out in memory or how calls are performed in binary form.
- **ABI (Application Binary Interface):** defines how compiled code components interact in *binary* form. It answers: *how independently compiled units agree to call each other, share data representations, and link/run together*.

A language standard does *not* fully define stack layout, calling convention, name mangling rules, or object binary layout across compilers. Those are ABI territory.

9.2 Why Binaries Must Agree

At the binary level, components are just sequences of machine instructions and data. If two components disagree on the rules, they will still link and run, but they will communicate incorrectly. Agreement is mandatory for correctness because the caller and callee must interpret bits the same way.
Binaries must agree on at least:

- **Calling convention:** where arguments are placed, where results are returned, which registers must be preserved.
- **Data layout:** size/alignment of types, struct/class field layout, padding rules, endianness assumptions.
- **Linkage conventions:** symbol naming, visibility, relocation model, object file conventions.
- **Runtime conventions:** stack alignment rules, exception/unwind conventions (if used), thread-local storage rules (if used).

If any of these differ, typical failure modes include reading arguments from the wrong location, corrupting preserved state, misinterpreting memory layout, or failing to unwind correctly.

9.3 ABI as a Contract Between Compiler, OS, and Code

An ABI is best understood as a **multi-party contract**:

- **Compiler ↔ Code:** the compiler must emit binaries that obey the calling and layout rules so functions and data are interoperable across translation units and libraries.
- **Compiler ↔ OS/Loader:** the OS loader and runtime environment expect binaries to follow specific executable and dynamic linking conventions (binary format, relocations, startup entry points, TLS layout, etc.).
- **Code ↔ Code (separate compilation):** object files and libraries compiled at different times (or by different toolchains that target the same ABI) must still call each other correctly.

In practice, the ABI is what turns "a function" into a concrete protocol that machine code can implement and the runtime can support.

9.4 Stability and Compatibility Guarantees

ABI stability is about **what changes can be made without breaking existing binaries**. The key compatibility notions are:

Binary compatibility

Two binary components are compatible if they can be linked and executed together correctly without recompilation. This typically requires:

- matching calling convention rules,
- matching data layout rules for shared types,
- matching symbol naming and linkage conventions,
- matching runtime expectations (loader, unwind, TLS conventions where relevant).

Source compatibility vs binary compatibility

- **Source compatibility:** source code still compiles after a change.
- **Binary compatibility:** already-compiled code still runs correctly after a change.

A change can preserve source compatibility while breaking binary compatibility (for example: altering the layout of a struct/class used across a library boundary).

What ABIs try to keep stable

Most mature platforms aim to keep stable:

- external function call rules,
- fundamental type sizes/alignments (within a platform model),
- executable and dynamic linking conventions.

What commonly breaks ABIs

Common ABI-breaking changes include:

- changing calling conventions or preserved register rules,
- changing the binary layout of exported types,
- changing symbol names or linkage conventions for exported interfaces,
- changing exception/unwind conventions across boundaries.

Practical rule for systems work

If two components communicate across a compiled boundary (library/API boundary, language boundary, plugin boundary), treat the ABI as part of the interface. Design those boundaries to be explicit and stable.

Conceptual Summary

- **ISA:** what the CPU executes.
- **Language spec:** what source code means.
- **ABI:** how binaries agree to interoperate.

Chapter 10

Calling Conventions

10.1 What a Calling Convention Defines

A **calling convention** is the ABI-defined protocol that makes a function call a well-defined binary operation between independently compiled code. It specifies the concrete rules for:

- **Argument passing:** where each argument is placed at call time (registers, stack slots, or both) and how it is represented.
- **Return values:** where results appear at return (fixed locations or indirect return via caller-provided storage).
- **Register preservation:** which registers are caller-saved vs callee-saved and what must survive across the call boundary.
- **Stack discipline:** how the stack pointer is adjusted, required alignment, and any reserved call areas required by the ABI.
- **Call/return mechanics:** how control transfer and return address handling are performed in a way compatible with the platform's binary model.

- **Boundary rules:** how varargs, tail calls (when applicable), and language interop boundaries behave under the same contract.

The convention is not optional. If caller and callee do not use the same convention, the call may still execute but the interpretation of arguments/returns/state becomes incorrect, leading to undefined behavior.

10.2 Why Multiple Conventions Exist

Multiple conventions exist because a single "best" protocol does not serve all historical, technical, and compatibility constraints. Common reasons include:

- **Historical evolution:** older conventions persist because large ecosystems depend on them, and breaking them would break existing binaries.
- **Different platform goals:** some environments prioritize minimal call overhead, others prioritize simpler tooling or compatibility across many languages.
- **Different interface shapes:** conventions may differ in how they treat varargs, large returns, vector/floating arguments, or special calling contexts.
- **Operating system and toolchain ecosystems:** OS loaders, debuggers, and library ecosystems standardize on conventions that become long-lived.

As a result, a platform may have one dominant "system" convention plus additional conventions used for specific toolchains, legacy APIs, interop layers, or special contexts.

10.3 Performance and Interoperability Impact

Calling conventions directly impact both **performance** and **interoperability**:

Performance

- **Register usage:** returning and passing common arguments in registers reduces memory traffic and lowers call overhead.
- **Save/restore cost:** the choice of caller-saved vs callee-saved distribution changes where overhead occurs (call sites vs function bodies).
- **Stack traffic and alignment:** conventions that require stack arguments or strict alignment can increase memory operations but may enable better vectorization and predictable behavior.
- **Leaf and small functions:** conventions that allow efficient leaf functions reduce overhead in heavily modular code.

Interoperability

- **Binary linkage:** libraries compiled separately must agree on the convention to exchange data correctly.
- **Cross-language calls:** languages can interoperate reliably only when they target the same ABI and calling convention for shared boundaries.
- **Varargs sensitivity:** variadic functions are particularly convention-dependent; mismatches commonly fail catastrophically because argument decoding depends on agreed placement rules.

10.4 Conceptual Comparison Without ISA Bias

Without tying to any specific architecture, calling conventions can be compared by their **contract shape**:

Where arguments go

- **Register-first conventions:** place the first K arguments in designated argument locations; spill remaining arguments to stack slots.
- **Stack-centric conventions:** place most arguments on the stack; use registers primarily for temporaries and returns.
- **Hybrid conventions:** use registers for common scalar arguments and stack for overflow, aggregates, and ABI-required cases.

Who pays preservation cost

- **Caller-heavy conventions:** more registers are caller-saved; call sites pay more save/restore around calls.
- **Callee-heavy conventions:** more registers are callee-saved; function bodies pay more prologue/epilogue overhead.
- **Balanced conventions:** split responsibilities to match typical compiler allocation patterns.

How large results are returned

- **Fixed-location returns:** scalars (and sometimes small aggregates) use designated return locations.
- **Indirect returns:** large or complex aggregates are returned by writing into caller-provided storage (hidden pointer argument conceptually).

What must remain invariant

All conventions, regardless of details, must guarantee:

- deterministic call/return pairing,
- stable argument and return interpretation,
- preservation of ABI-required state,
- and stack invariants sufficient for nested execution and tooling.

Conceptual Call Sketch (Not an ABI Specification)

```
# Conceptual-only illustration (Intel-like syntax).
# Not a platform rule set.

# Call site prepares call environment according to the convention:
mov ARG0, a0
mov ARG1, a1
# ... possibly spill extra args to stack slots ...
call callee

# Callee obeys the same convention:
callee:
    # preserve required state if needed (callee-saved)
    # establish stack invariants as required
    # compute result into return location or into caller-provided storage
    ret
```

Key Rule

A calling convention is the **binary protocol** for calls. Multiple conventions exist because compatibility and optimization goals vary. Performance depends on where arguments live and who pays preservation cost; interoperability depends on strict agreement at every boundary.

Chapter 11

Stack Alignment and Discipline

11.1 Why Alignment Matters

Alignment is the rule that certain objects must start at addresses that are multiples of a specific power-of-two boundary. Alignment matters for four practical reasons:

- **Correctness for some instructions:** certain hardware operations (especially vector/SIMD loads/stores and atomics on some platforms) require specific alignment or have stricter behavior/performance when aligned.
- **Performance:** aligned accesses typically map more efficiently to cache lines and memory buses. Misalignment can force extra micro-operations, split loads/stores, or slower access paths.
- **Calling convention predictability:** the ABI must guarantee that the callee can safely assume a known stack alignment at entry (and before making further calls).
- **Tooling and runtime mechanisms:** unwinding, exception propagation, and stack walking rely on stable, ABI-compliant frame assumptions, which include alignment

invariants.

In short: alignment is not stylistic. It is a machine-level constraint that enables safe code generation and efficient execution.

11.2 Alignment as an ABI Requirement

The ABI typically mandates a specific **stack alignment invariant** at key points, most importantly:

- **at function entry:** the callee may assume the stack pointer meets a defined alignment,
- **before a call instruction:** the caller must ensure the stack is aligned as required so the callee's entry assumptions hold,
- **for allocating objects on the stack:** the compiler must place stack objects at offsets that satisfy their alignment requirements.

This is a contract: if the caller fails to maintain alignment, the callee may execute code that assumes alignment and thus misbehave. The ABI requirement exists precisely so compilers can generate fast code without inserting defensive alignment checks at every call boundary.

11.3 Consequences of Misalignment

Misalignment consequences range from subtle slowdowns to immediate failure, depending on architecture, instruction selection, and runtime environment:

- **Performance degradation:** split memory operations, extra micro-ops, pipeline penalties, and cache inefficiencies.

- **Instruction faults or traps (platform-dependent):** some instructions may fault if alignment preconditions are violated.
- **Data corruption symptoms:** if misalignment breaks ABI expectations across calls, the callee may interpret stack-resident data incorrectly.
- **Unwinding/debugging failures:** stack walkers and unwinders may fail to reconstruct call chains correctly if invariants are violated, especially under optimized code.
- **Undefined behavior at ABI boundaries:** calling into foreign code (libraries, system runtime, mixed-language boundaries) with a misaligned stack can produce unpredictable failures far from the call site.

The dangerous aspect is that misalignment bugs often appear *non-local*: the caller breaks an invariant, the callee fails later in a seemingly unrelated place.

11.4 Compiler Responsibility vs Programmer Responsibility

In normal high-level C/C++ code, **the compiler is responsible** for maintaining stack alignment and placing stack objects with correct alignment, because it controls frame layout and call sequences.
However, **the programmer becomes responsible** whenever they step outside the language/compiler-managed calling path, such as:

- writing **hand assembly** that makes calls,
- using **inline assembly** that adjusts the stack pointer,
- implementing **context switching** (coroutines, fibers, green threads) at a low level,
- writing **function trampolines**, hooking, or custom call stubs,

- interfacing with **foreign ABIs** where the compiler cannot verify the boundary.

Rule of discipline:

> If you manually change the stack pointer or manually perform a call boundary, you own the ABI alignment contract.

Conceptual Alignment Sketch (Not a Platform Rule Set)

This sketch shows the idea: before calling, ensure `SP` is aligned to the ABI-required boundary. The specific boundary value is platform-specific; the invariant concept is universal.

```
# Conceptual-only illustration (Intel-like syntax).
# Not an ABI specification.

# Ensure SP is aligned before calling (boundary is ABI-defined).
# If you adjust SP manually, you must preserve the invariant.

and SP, -ALIGN        # concept: align SP down to multiple of ALIGN
call callee
```

Key Takeaways

- Alignment enables correct and efficient machine execution.
- ABIs mandate stack alignment to make call boundaries reliable and optimizable.
- Misalignment can cause slowdowns, faults, corrupt behavior, and broken unwinding.
- Compilers manage alignment in normal code; low-level manual stack work transfers responsibility to the programmer.

Chapter 12

C and C++ Interoperability

12.1 Why C and C++ Must Share an ABI

C and C++ coexist in the same systems software ecosystem: operating systems, language runtimes, standard libraries, device drivers, and third-party libraries frequently expose C interfaces while being implemented in a mixture of C and C++. For this to work, both languages must be able to agree on a common binary calling contract for a shared boundary. At a practical level, interoperability requires agreement on:

- **Calling convention:** how arguments are passed, how returns are produced, and which registers are preserved.
- **Fundamental type model:** size and alignment of primitive types used at the boundary.
- **Data layout rules for shared structs:** field order, padding, and alignment for C-compatible aggregates.
- **Linkage and symbol naming:** how function names are represented in object files so linkers can match declarations to definitions.

C is commonly used as the lowest-common-denominator boundary language because its ABI surface is comparatively stable and widely supported across toolchains and platforms.

12.2 Name Mangling (Conceptual)

C++ supports **overloading**: multiple functions can share the same source name if their parameter types differ. At the binary level, however, symbol names must be unique. To achieve uniqueness, C++ toolchains encode additional information (such as parameter types, namespaces, and class membership) into the exported symbol name. This encoding is called **name mangling**.
Conceptually:

- **C linkage:** exported symbol name is (typically) the plain function name.
- **C++ linkage:** exported symbol name is an encoded form that includes type/signature context.

Because different C++ ABIs/toolchains can use different mangling schemes, relying on raw C++ symbol names across toolchain boundaries is fragile unless the environment standardizes a specific C++ ABI.

12.3 Meaning of `extern "C"`

`extern "C"` is a C++ linkage specification that requests **C language linkage** for the declared function(s) or object(s). Conceptually, it does two core things:

- **Disables C++ name mangling** for the declared entity, so the linker-visible name follows the C-style convention for that platform/toolchain.

- **Selects the C ABI calling interface** expected for C interoperability on that platform (i.e., the calling convention and boundary rules associated with C linkage in that environment).

Important scope clarification:

- `extern "C"` affects **linkage** (how names are exported and matched) and the **ABI boundary contract**.

- It does **not** make arbitrary C++ types "C-compatible." Type layout, exceptions, constructors/destructors, and templates remain C++ semantics.

Minimal Interop Pattern

```
#ifdef __cplusplus
extern "C" {
#endif

int api_add(int a, int b);

#ifdef __cplusplus
}
#endif
```

```
extern "C" int api_add(int a, int b) {
    return a + b;
}
```

12.4 Binary Compatibility Boundaries

Interoperability is easiest when the boundary is intentionally narrow and uses C-compatible shapes. Practical compatibility boundaries include:

Stable boundary choices

- **C functions with primitive arguments/returns** (integers, pointers, fixed-size scalars).
- **Plain C structs** used as data packets, with well-defined layout rules and no C++-only features.
- **Opaque pointers (handles)** to hide C++ internals behind a stable C API.

Fragile or ABI-sensitive boundary choices

- **C++ classes across boundaries:** vtables, object layout, and exception models are ABI-governed and can vary by toolchain/standard library.
- **Templates and inline functions as ABI:** they are compiled into each consumer and are not a stable binary boundary.
- **Exceptions across boundaries:** unwind conventions and runtime types must match; crossing language or runtime boundaries is often unsafe unless explicitly standardized.

Design rule for systems interfaces

Use a C ABI surface for long-lived binary interfaces, and treat C++ as an implementation detail behind that surface. This yields predictable linkage, predictable calling behavior, and strong compatibility across compilers and languages targeting the same platform ABI.

Key Takeaways

- C and C++ must agree on a common ABI at the boundary to enable separate compilation and linking.
- C++ name mangling exists to support overloading; it complicates cross-toolchain linking.
- `extern "C"` requests C linkage: stable symbol naming and the platform's C ABI boundary rules.
- Binary compatibility is strongest with narrow, C-shaped interfaces and weakest when exporting C++ object models.

Chapter 13

Undefined Behavior Around the Stack

13.1 Stack Corruption Concepts

Stack corruption is any violation of the invariants that make stack-based execution reversible and ABI-compliant. Conceptually, corruption occurs when the program writes to stack memory that it does not own, restores the wrong saved state, or breaks required stack pointer/frame invariants.
Common corruption patterns (conceptual, ISA-independent):

- **Out-of-bounds writes** into a stack frame (overwriting saved state, return metadata, or neighboring locals).
- **Use-after-lifetime** of automatic storage (using pointers/references to objects whose frame is gone).
- **Incorrect manual stack adjustment** (hand-written assembly or inline asm that changes `SP` incorrectly).

- **Incorrect save/restore logic** for preserved registers (callee-saved not restored, wrong restore order, wrong offsets).

The core danger is that corrupted stack state is often *consumed later* by control-flow operations (returns) and by subsequent calls, making failures appear far from the original bug.

13.2 Mismatched Calls and Returns

Calls and returns must form a strict, structured pair: the return must restore execution to the correct continuation point with the correct state.
Mismatches include:

- **Returning with an invalid stack pointer:** the return address is read from the wrong place, or the caller's frame is not correctly reestablished.
- **Corrupting the return address or linkage:** overwriting the stored return point (directly or indirectly) causes execution to jump unpredictably.
- **Multiple returns without restoring invariants:** early exits that skip required restoration steps (conceptually) break the caller's expectations.

In a well-formed ABI model, *every* exit path must restore the same contractual invariants as the normal epilogue.

13.3 Violating Calling Conventions

Calling conventions are binary contracts. Violations often compile and even "seem to work" until an optimization, a different build, or a different call path makes the mismatch visible.
Typical violation categories:

- **Wrong function signature across a boundary:** caller and callee disagree about argument types/sizes/count, so the callee reads wrong locations or wrong widths.
- **Wrong calling convention selection:** caller and callee disagree about where arguments/returns live and which registers must be preserved.
- **Varargs misuse:** variadic calls depend on precise ABI rules; mismatches in the fixed-parameter prefix or in how arguments are interpreted commonly fail catastrophically.
- **Not preserving callee-saved state:** the callee clobbers state the caller assumes survives, leading to corrupted computation after return.
- **Breaking stack alignment at a call boundary:** the callee may execute code that assumes alignment and thus misbehaves or faults.

The key point: the ABI does not merely optimize; it defines what correctness means for a call boundary.

13.4 Why Stack-Related UB Is Catastrophic

Undefined behavior around the stack is particularly catastrophic because it attacks the **mechanism that controls execution itself**. When stack invariants fail, the program may lose:

- **Control-flow integrity:** a corrupted return path can redirect execution to unintended code locations.
- **State integrity:** corrupted preserved registers or frame data can silently poison computations long before any visible crash.
- **Diagnosability:** stack walkers, debuggers, and unwinders may fail because the call chain can no longer be reconstructed reliably.

- **Reproducibility:** small changes (optimization level, inlining decisions, different compiler version) can move frame layouts and change symptoms.

Stack UB is also highly **non-local**: the bug may occur in one function, but the failure may appear many calls later, in unrelated code, because the corrupted state is consumed only when a later return or later load uses it.

Conceptual "Failure Shapes" (Not Exploit Guidance)

The following sketches show *how* things go wrong conceptually, without providing attack procedures.

```
1) OOB write in a stack frame
   -> overwrites saved state (return linkage / preserved registers)
   -> later return restores wrong state or wrong return target

2) Mismatched prototype across boundary
   -> callee reads arguments from wrong locations / wrong widths
   -> computation corrupt or stack pointer restored incorrectly

3) Stack alignment broken before call
   -> callee assumes alignment
   -> misaligned access / fault / corrupted ABI expectation
```

Discipline Rules (Architecture-Neutral)

- Treat the stack pointer and preserved state as **ABI-owned invariants**.
- Ensure **every exit path** restores the same contractual invariants.
- Do not cross binary boundaries with ambiguous or mismatched signatures.

- If you write assembly/inline asm that touches `SP`, you own the ABI contract: alignment, save/restore, and call/return symmetry.

Chapter 14

What This Booklet Deliberately Excludes

14.1 Why No ISA-Specific Rules

This booklet is intentionally **architecture-neutral**. Instruction sets differ in register sets, instruction encodings, addressing modes, and low-level calling mechanics. If we embed ISA-specific details here, the reader risks learning one platform's conventions as if they were universal truths.

The goal of this booklet is to teach **invariants and contracts** that remain stable across architectures:

- the logical meaning of a call boundary,
- stack-frame responsibilities,
- preservation rules as contracts,
- argument and return protocols as ABI obligations,
- and alignment discipline as a correctness requirement.

ISA-specific facts (exact registers used for arguments/returns, exact prologue sequences, exact stack layout conventions) belong in later booklets where they can be presented precisely for each architecture without contaminating the core model.

14.2 Why No Syscalls

System calls are a separate interface layer with a different purpose and different contracts. A syscall boundary is not the same as a normal function-call boundary:

- A syscall crosses **user/kernel** privilege boundaries and follows an OS-defined ABI.
- Its argument/return conventions are platform- and OS-specific.
- It interacts with kernel-managed state, error reporting conventions, and security/permission rules.

Including syscalls here would blur the boundary between:

- **ABI for calls between code components** (functions, libraries, languages),
- and **OS ABI for entering the kernel**.

This booklet focuses only on the call/return machinery needed for reliable *user-space* binary interoperability and compiler-generated execution structure.

14.3 Scope Discipline and Learning Order

A disciplined scope is not a limitation; it is a learning strategy. The stack and calling conventions are foundational. They must be understood first as a **portable model**, otherwise the learner becomes trapped in memorization:

- memorizing register lists without understanding preservation responsibility,
- copying prologue sequences without understanding invariants,
- and reading assembly without knowing what is guaranteed versus accidental.

The intended learning order is:

1. Understand **execution contracts** (this booklet).
2. Map those contracts onto a specific ABI (architecture-specific booklet).
3. Only then study system interfaces (syscalls) as a separate OS-defined contract.

This order minimizes confusion and prevents "false universals" learned from one platform from being misapplied to another.

14.4 Preparing for Architecture-Specific Analysis

You are ready for architecture-specific ABI study when you can answer these questions without using any register tables:

- What must the caller establish before a call so the callee can run safely?
- What must the callee preserve so the caller can resume safely?
- What does it mean for arguments and return values to be ABI-defined?
- Why is stack alignment a contract rather than an optimization?
- What kinds of errors are ABI violations versus language-level bugs?

In the next (architecture-specific) phase, you will take this exact conceptual framework and map it to concrete rules:

- specific argument/return locations,
- specific preserved register sets,
- exact stack layout and alignment requirements,
- real prologue/epilogue patterns under different optimization levels,
- and practical boundary cases (varargs, aggregates, vector arguments, unwinding conventions where applicable).

Key Takeaway

This booklet excludes ISA-specific rules and syscalls to preserve a clean separation:

- **here:** universal execution contracts and ABI reasoning,
- **later:** precise architecture/OS mappings and syscall interfaces.

That separation is what makes the knowledge transferable across x86-64, ARM64, RISC-V, and beyond.

Appendices

Appendix A — Mental Models for Debugging

Visualizing Stack Growth

A useful debugging mental model is to treat the stack as a **timeline of activations**:

- **Each function call** creates a new frame on top of the current one.
- **Each return** removes the most recent frame and restores the previous context.
- The **topmost frame** is always the currently executing function.

Think in two layers simultaneously:

- **Logical stack:** the nested call structure (which function called which).
- **Physical stack:** a memory region managed by the ABI where frames are realized (sometimes partially or not at all under optimization).

Key invariant for reasoning (ISA-independent):

- "*Growth*" means "more recent activations are closer to `SP`." Whether addresses numerically increase or decrease is irrelevant to the model.

Frame Contents: What to Picture

When you visualize a frame, think of **roles**, not offsets:

- return linkage (how control gets back),
- preserved state (callee-saved responsibilities),
- automatic storage (locals/temporaries),
- spill/outgoing areas (used by the compiler when needed),
- alignment padding (to maintain ABI invariants).

Conceptual Diagram

```
Top (most recent activation)
+---------------------------+
| Current frame             |
| - return linkage          |
| - saved state (if needed) |
| - locals/temps/spills     |
+---------------------------+
| Caller frame              |
+---------------------------+
| Older frames              |
+---------------------------+
Bottom (older activations)
```

Tracing Calls Without Tools

When tools are unavailable (or when optimized builds obscure frames), you can still trace execution by applying the **call-boundary contract** mentally:

Step 1: Build a call chain from the source

- Identify the current function and list the functions it can call along the active path.
- For each call, record: arguments, expected return value, and which variables remain live after the call.

Step 2: Identify state that must survive calls

- Mark values that the caller uses *after* a call.
- Mentally classify them as "must survive" across the boundary.
- This directly predicts preservation pressure (values likely kept in callee-saved locations or spilled to the stack by the compiler).

Step 3: Apply LIFO reasoning

- For nested calls, always unwind in reverse order.
- If behavior is wrong after a return, suspect:
  - corrupted preserved state,
  - wrong argument interpretation (prototype mismatch),
  - broken stack alignment or frame teardown.

Step 4: Use invariants to narrow failures

In stack/ABI debugging, ask invariant questions:

- Did the callee promise to preserve something that changed?
- Did the caller assume a value survives even though it was caller-saved?

- Did a boundary cross languages or compilation units with mismatched signatures?
- Did any low-level code adjust `SP` manually?

Preparing to Read Compiler-Generated Assembly

Compiler-generated assembly is easier to read if you know what to look for and what *not* to assume.

What to look for first

1. **Call sites:** identify the `call`-like transfer and examine what was prepared immediately before it.
2. **Prologue/epilogue patterns:** locate frame establishment and teardown (even if minimized).
3. **Preservation actions:** find saves/restores that correspond to callee-saved responsibilities.
4. **Stack adjustments:** locate where `SP` changes, and ensure changes are reversed symmetrically along all exit paths.

How to interpret registers without guessing a platform

Even without register tables, you can reason by **role**:

- values prepared before a call are likely *arguments* or *hidden ABI parameters*,
- values used after a call are likely *caller-live state* that must be preserved somehow,
- stores to the stack near entry often represent *saves* or *spills*,
- loads near exit often represent *restores*.

Optimization warnings (what not to assume)

- A function may have **no visible frame** (leaf, inlined, frame-pointer omitted) yet still obey the ABI.
- Locals may never appear on the stack if held in registers.
- Call/return structure can be transformed (tail calls) while preserving the ABI-visible contract.

Conceptual Assembly Reading Checklist

```
1) Find the call boundary
2) Determine which values are prepared as inputs
3) Determine which state must survive across the call
4) Identify saves/restores (callee-saved or spills)
5) Check SP adjustments are balanced on all exits
6) Confirm return value is produced in the ABI-defined manner
 ↪  (concept)
```

Key Takeaways

- Visualize frames as activation records stacked in LIFO order; ignore numeric address direction.
- Debugging without tools relies on call-boundary contracts and invariants, not on memorizing layouts.
- Reading compiler assembly becomes systematic when you focus on roles: inputs, preserved state, stack adjustments, and exit symmetry.

Appendix B — Common Misconceptions

“The stack is just memory”

This statement is technically incomplete and therefore misleading. While the stack is implemented using memory, it is not an arbitrary memory region. It is a **structured execution mechanism** governed by strict ABI rules.
Key clarifications:

- The stack enforces **LIFO execution order** for nested calls; arbitrary memory does not.
- Stack memory has **implicit semantics**: return linkage, preserved state, alignment guarantees, and frame boundaries.
- Stack misuse does not merely corrupt data; it can corrupt **control flow**, because return behavior depends on stack-managed metadata.

Treating the stack as “just memory” leads to unsafe assumptions, such as writing past local storage or assuming frame layout stability across builds. Correct reasoning requires viewing the stack as *an execution protocol realized in memory*, not as a free-form buffer.

“Calling conventions don’t matter”

Calling conventions are often invisible in high-level source code, which creates the illusion that they are optional. In reality, they are the **binary contract** that makes function calls correct across compilation boundaries.
Why this misconception is dangerous:

- The compiler relies on the convention to decide where arguments live, where results appear, and which registers survive calls.
- Debuggers, profilers, and unwinders rely on the same rules to reconstruct call chains.

- Mismatched conventions typically compile and link, but produce undefined behavior at runtime.

Calling conventions matter most when:

- mixing languages or toolchains,
- using inline or hand-written assembly,
- working with varargs,
- or debugging optimized builds.

Ignoring calling conventions is equivalent to assuming that two independent pieces of code "just agree" by accident.

"C and C++ calls are always compatible"

C and C++ are closely related languages, but their binary interfaces are not automatically interchangeable. Compatibility exists *only* when both sides intentionally target the same ABI boundary.
What is compatible by design:

- Functions with **C linkage** using C-compatible types.
- Shared data structures that obey C layout rules.

What is not inherently compatible:

- C++ function overloading and name mangling.
- C++ classes with constructors, destructors, virtual dispatch, or non-trivial layout.
- Exceptions crossing language or runtime boundaries.

- Templates and inline functions as exported binary interfaces.

The correct model is:

> C and C++ interoperability is explicit and contractual, not automatic.

Correct Mental Model

- The stack is a structured execution mechanism, not just memory.
- Calling conventions are mandatory ABI contracts, not compiler trivia.
- C/C++ compatibility exists only across intentionally designed C-style ABI boundaries.

Understanding and correcting these misconceptions is essential before moving on to architecture-specific ABIs or low-level debugging.

Appendix C — Preparation for Architecture-Specific Study

What to Look For in x86-64, ARM64, and RISC-V

When transitioning from an architecture-neutral model to a concrete platform, focus on *mapping roles to rules*, not memorizing instruction sequences.
For each architecture, identify:

- **Argument locations:** which registers are designated for arguments, how many are used before spilling to the stack, and how aggregates are handled.
- **Return mechanisms:** fixed return locations for scalars and the protocol for indirect (caller-provided) returns.
- **Register preservation sets:** which registers are caller-saved vs callee-saved under the platform ABI.

- **Stack invariants:** required alignment at call boundaries, red zones or reserved areas (if any), and minimum frame guarantees.
- **Frame conventions:** presence/absence of a frame pointer by convention, and how unwind/debug metadata is expected to describe frames.

Across x86-64, ARM64, and RISC-V, the *details differ*, but the underlying contracts (call boundary, preservation responsibility, alignment discipline) are the same concepts presented in this booklet.

Mapping This Booklet to Real ABIs

Use this booklet as a translation key from *concept* to *specification*:

- **Prologue/Epilogue** → ABI-mandated preservation and alignment rules, not a fixed instruction template.
- **Caller-saved vs Callee-saved** → concrete register lists defined by the platform ABI.
- **Argument Passing** → ordered mapping from parameters to registers and stack slots as defined by the ABI.
- **Return Values** → designated return locations and indirect return protocols.
- **Stack Discipline** → exact alignment guarantees required before calls and at function entry.

If an ABI rule seems arbitrary, trace it back to one of these conceptual needs: preservation, alignment, determinism, or interoperability.

How to Read ABI Documents Effectively

ABI documents are precise but dense. Read them with a purpose-driven strategy:

Step 1: Identify the Call Boundary Contract

Locate sections that define:

- function call sequences,
- argument and return placement,
- preserved vs volatile state.

Ignore instruction encodings initially; focus on *what must hold true at entry and exit*.

Step 2: Extract Invariants

From the text, extract invariants such as:

- required stack alignment before calls,
- registers that must be preserved by callees,
- assumptions the callee may make about incoming state.

Write these invariants as short rules. These are what compilers and hand-written assembly must obey.

Step 3: Separate Mandatory Rules from Conventions

Distinguish between:

- **mandatory ABI rules** (violating them breaks correctness),
- **recommended conventions** (violating them may break tooling or performance, but not immediate correctness).

This separation helps prioritize what matters when debugging or writing low-level code.

Step 4: Validate with Compiler Output

After reading the ABI rules, examine compiler-generated assembly and verify:

- stack alignment is established as specified,
- preserved registers are saved/restored only when used,
- arguments and returns appear in the documented locations.

Conceptual ABI Reading Checklist

```
1) What must the caller guarantee before the call?
2) What may the callee freely overwrite?
3) What must the callee restore before returning?
4) Where are arguments placed, in what order?
5) Where is the return value produced?
6) What stack alignment must hold at the boundary?
```

Final Guidance

Approach each architecture-specific ABI as an *instantiation* of the same abstract model. When you can explain a concrete rule in terms of preservation, alignment, or call determinism, you are studying the ABI correctly. When rules are memorized without that mapping, understanding will be brittle and non-transferable.

References

All Resources Used for This Booklet

The content of this booklet is derived from long-established, authoritative, and academically accepted resources in computer architecture, compiler design, and systems programming. These sources define the canonical understanding of stacks, calling conventions, and ABIs across modern platforms.

Language Standards and Specifications

- ISO C Language Standard — definitions of object lifetimes, storage duration, and function call semantics at the abstract machine level.
- ISO C++ Language Standard — definitions of automatic storage, object lifetimes, calling semantics, and interoperability rules, including linkage and ABI-relevant constraints.

ABI Specifications

- System V Application Binary Interface — foundational ABI model influencing most UNIX-like platforms.

- Platform-specific ABI documents for x86-64, AArch64 (ARM64), and RISC-V — defining calling conventions, register preservation rules, stack alignment, and binary interoperability.
- Executable and object file format specifications (ELF model) — defining linkage, symbol resolution, and runtime loading behavior.

Compiler Design and Toolchain Documentation

- Compiler internal design documentation from major production compilers — describing stack frame construction, prologue/epilogue generation, register allocation, and ABI enforcement.
- Official compiler manuals covering calling conventions, inline assembly constraints, and ABI compliance requirements.

Computer Architecture References

- Academic textbooks on computer architecture — covering instruction execution, register usage, call/return mechanisms, and memory alignment.
- Processor architecture manuals — defining instruction behavior, calling support mechanisms, and architectural constraints relevant to ABI design.

Operating Systems and Runtime Models

- UNIX and UNIX-like operating system design literature — explaining user-space execution models and binary interfaces.
- Runtime system documentation — covering stack unwinding models, exception handling infrastructure, and debugging metadata requirements.

Debugging and Binary Analysis Sources

- Debugger architecture documentation — explaining stack walking, frame reconstruction, and reliance on ABI-defined invariants.
- Binary analysis and reverse-engineering literature — reinforcing ABI-based reasoning for call boundaries and execution flow.

Academic and Industry Consensus

- Peer-reviewed academic material on calling conventions and ABI stability.
- Industry-standard practices established by decades of compiler and operating system evolution.

Scope Clarification

This booklet intentionally relies only on:

- stable, architecture-neutral principles,
- ABI concepts shared across major modern platforms,
- and long-standing industry and academic consensus.

No experimental, proprietary, or non-standard sources were used. All explanations reflect behavior that production compilers and operating systems depend on today and have depended on for many years.

Final Note

These references form the conceptual backbone for the entire CPU Programming Series. Architecture-specific booklets will build directly on the same sources, mapping their abstract

rules to concrete register sets, instruction sequences, and platform conventions without redefining the underlying principles.