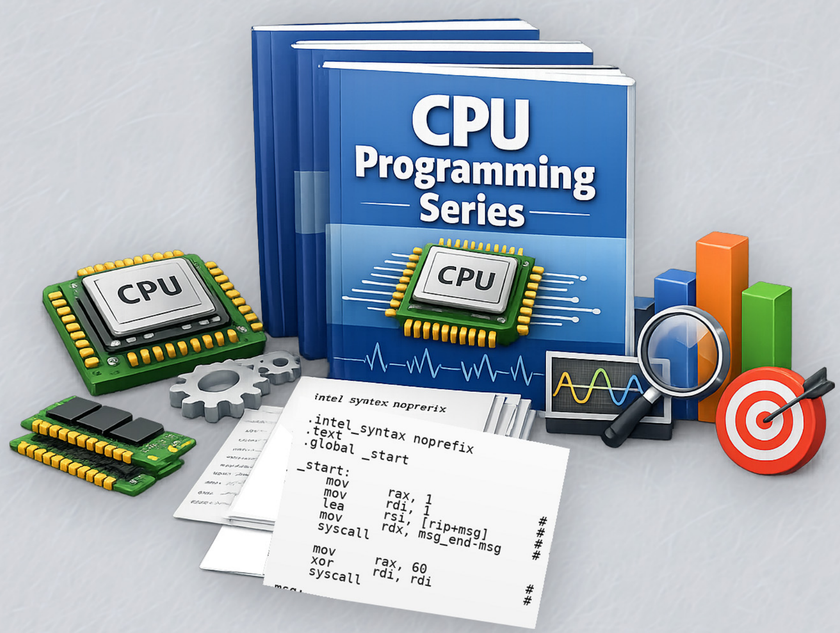
https://simplifycpp.org

CPU Programming Series

IA-32 Assembly in Practice

bit Stack-Based ABI & Interoperability-32



```
.intel_syntax noprefix
.text
.global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [rip+msg]
    mov     rdx, msg_end-msg
    syscall

    mov     rax, 60
    xor     rdi, rdi
    syscall
```

7

Prepared by Ayman Alheraki

CPU Programming Series

IA-32 Assembly in Practice

32-bit Stack-Based ABI & Interoperability

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Why IA-32 Still Matters in 2026

IA-32 (x86-32) is no longer the mainstream desktop/server deployment target, yet it remains a *high-value practical skill* in 2026 for one simple reason: it is the cleanest real-world environment where the ABI is visibly *stack-centric*. If you can reason correctly about 32-bit stack-based calling conventions, you can debug and verify interoperability at a level that makes 64-bit ABIs (with more register passing) feel easier rather than harder.

- **Legacy compatibility is still a business reality.** Large codebases, vendor SDKs, and industrial deployments keep 32-bit components alive because rewriting and recertifying is expensive.

- **Security and reverse engineering still rely on IA-32.** Many malware samples, packed binaries, and older commercial software remain 32-bit; analysts must read IA-32 call stacks and conventions fluently.

- **The stack tells the truth.** IA-32 makes argument passing, cleanup rules, and hidden ABI parameters obvious. This builds strong intuition for correctness, debugging, and toolchain behavior.

- **Interoperability is an engineering necessity.** Mixing C, C++, and assembly (and sometimes older binary-only libraries) is common in systems programming, emulation, tooling, and compatibility layers.

This booklet treats IA-32 as a *laboratory for ABI discipline*: correct stack layout, correct cleanup responsibility, correct register preservation, and correct C/C++ interoperability.

Mini Example: The ABI as a Contract (Not a Style Choice)

Two functions with identical source intent can be binary-incompatible if they disagree on the calling convention (who cleans the stack, which registers must be preserved, and how names are exported).

```
extern "C" int __cdecl   f_cdecl(int a, int b);
extern "C" int __stdcall f_stdcall(int a, int b);
```

If you link a caller compiled to call `__cdecl` with a callee implemented as `__stdcall`, the program may appear to work briefly, then crash later from a corrupted stack pointer. This is not a "bug" in either function; it is a broken ABI contract.

When and Where 32-bit x86 Is Still Used

In 2026, IA-32 persists in several practical environments. The exact mix varies by organization, but the underlying reasons are stable: compatibility, cost, hardware constraints, and certification inertia.

- **Long-lived industrial PCs and kiosks:** systems that were deployed years ago and are maintained rather than replaced.
- **Embedded x86 and appliance-like deployments:** special-purpose devices built around x86 SoCs where the software stack is frozen.

- **Legacy commercial software and plugins:** binary-only components that exist only in 32-bit form.
- **Compatibility layers and tooling:** emulators, dynamic instrumentation, debuggers, unpackers, and reverse-engineering workflows.
- **Cross-compilation and multilib userlands:** 64-bit hosts that still build/run 32-bit binaries (toolchains, tests, compatibility, older dependencies).

Mini Example: 32-bit on a 64-bit Host (Multilib Build)

On many development hosts, you can compile a 32-bit binary from a 64-bit toolchain (when 32-bit libraries are installed).

```
# Build 32-bit on a 64-bit host (multilib required)
gcc -m32 -O2 -c add.c -o add.o
gcc -m32 add.o -o add32
```

This booklet focuses on the binary interface rules that make such builds correct and interoperable.

Scope and Assumptions of This Booklet

This booklet is about **IA-32 in practice**: how real compilers and linkers shape function calls, stack frames, and interoperability boundaries. The aim is **ABI correctness**, not instruction memorization.

Primary Scope

- IA-32 calling convention(s) used by mainstream toolchains.

- Stack argument passing and real stack layouts.
- Stack cleanup rules (caller vs callee cleanup).
- Register preservation rules (caller-saved vs callee-saved).
- Return-value rules including structure returns and hidden parameters.
- C / C++ interoperability boundaries:
  - name mangling vs `extern "C"`
  - matching calling conventions across translation units
  - struct layout and alignment expectations

Assumptions

- You understand the **conceptual CPU execution model** (fetch/decode/execute/retire) from earlier booklets.
- You know basic **register and flag meaning** and can read simple IA-32 assembly.
- You can read C/C++ function declarations and understand types, pointers, and basic structs.
- You want **correctness-first** reasoning: stack integrity, ABI compliance, and predictable interoperability.

Example Convention Used in This Booklet

To keep examples concrete, many sections demonstrate a cdecl-style stack discipline (arguments on stack, caller cleanup), then contrast it with stdcall/fastcall where relevant. Assembly is shown in Intel syntax, with GAS-style comments using `#`.

```
# IA-32 example (Intel syntax shown), comments use '#'
# Conceptual: call add(a, b) where a and b are on the stack
```

What This Booklet Does Not Cover

This is a focused ABI and interoperability booklet. Many advanced topics are intentionally excluded to keep the mental model clean and the page count within target.

- **No OS internals or system call ABI.** We do not cover kernel entry mechanisms, interrupt/trap frames, or OS-specific syscall conventions.
- **No deep coverage of segmentation and legacy protected-mode details.** We only touch what is required to reason about normal user-space code generation and stack behavior.
- **No floating-point deep dive.** We mention return/parameter realities only as needed, but do not fully teach x87/SSE calling details.
- **No exception-unwinding ABI deep dive.** C++ exceptions, unwinding metadata, and personality routines are not the goal here.
- **No inline-assembly dialect tutorial.** We focus on ABI behavior and interoperability, not on teaching a specific compiler's inline-asm syntax.
- **No comprehensive instruction reference.** You should already have an instruction reference strategy from prior booklets; we use only what is needed.

Why These Exclusions Are Healthy

ABIs are hard mainly because engineers mix concerns. This booklet isolates the ABI contract: calling convention, stack layout, register preservation, and cross-language linking. Once that is mastered, adding OS details, exceptions, and SIMD rules becomes structured work instead of guesswork.

How This Booklet Fits in the CPU Programming Series

This booklet is the **first architecture-specific** step where the series becomes *binary-interface practical*. It bridges:

- the earlier ISA-independent foundations (execution model, data representation, flags, stack concepts), and
- the real-world constraints of toolchains (compilers, assemblers, linkers) and cross-language integration.

Series Position and Dependency Map

- **You should already be comfortable with:**
  - binary data representation (signed/unsigned, carry vs overflow)
  - stack growth, call/return concept, stack frames conceptually
  - reading small assembly snippets and relating them to source intent
- **This booklet prepares you for:**
  - x86-64 ABI work (more register passing, stronger alignment rules, different red-zone / shadow-space realities depending on platform)
  - calling-convention debugging in real crash dumps and traces
  - writing ABI-correct assembly that interoperates with C/C++ reliably

Practical Outcome Checklist

After finishing this booklet, you should be able to do all of the following *without guessing*:

1. Predict the **stack layout** at a call site and at function entry.
2. Determine who must **clean the stack** for a given calling convention.
3. Identify which registers must be **preserved** by the callee.
4. Explain how **struct returns** work, including hidden pointers when required.
5. Make C and C++ interoperate safely using `extern "C"` and explicit calling-convention annotations.
6. Recognize the common signatures of ABI mismatch: stack imbalance, corrupted return address, and silent data corruption.

Extensive Interoperability Preview (One Small, Real Pattern)

A common pattern is to expose a stable C ABI entry point from C++ so that assembly (or other languages) can call it reliably.

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

int __cdecl sum2(int a, int b);

#ifdef __cplusplus
}
#endif
```

Then implement it in IA-32 assembly with ABI-correct behavior: read arguments from the stack and return in `EAX`. (Exact prologue style may vary with optimization, but the ABI rules do not.)

```
# sum2(a, b) using a stack-based ABI (cdecl-style)
# On entry (conceptually):
#   [esp+4] = a
#   [esp+8] = b
# Return:
#   eax = a + b

sum2:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]     # a
    add  eax, dword ptr [ebp+12]    # b
    pop  ebp
    ret
```

This booklet explains every hidden rule behind that small example: why those offsets exist, what changes under different conventions, what must be preserved, how the symbol must be exported, and how to debug failures when the contract is violated.

Chapter 1

IA-32 Execution Model Recap

1.1 32-bit Mode vs Compatibility Mode

The term **IA-32** refers to the 32-bit x86 programming model: 32-bit general-purpose registers, 32-bit linear addresses (with architectural mechanisms that can alter the effective address size), and a 32-bit stack discipline. In practice, you will encounter IA-32 code running in two common environments:

- **Legacy 32-bit mode (native IA-32):** the processor executes with a 32-bit execution environment where the default operand and address sizes are 32-bit (subject to instruction prefixes), and the OS is 32-bit.

- **Compatibility mode under x86-64:** a 64-bit CPU runs a 32-bit user program while the OS kernel is 64-bit. The program still sees the IA-32 register set and the 32-bit ABI rules, but it is hosted by a 64-bit OS environment that may impose additional system-level constraints (loader, syscall interface, library availability). For ABI and interoperability inside user-space, the *function call contract remains IA-32*.

1.1.1 What does `call` / `ret` mean in both cases?

In both native IA-32 and compatibility execution of 32-bit code, `call` pushes a return address and transfers control, and `ret` pops that return address to resume execution. This stack-based control flow is the backbone of every 32-bit calling convention.

```
# Conceptual view (IA-32):
# call target
#   1) push return_address
#   2) EIP <- target
#
# ret
#   1) EIP <- pop()
```

The exact encodings differ (`call rel32`, `call r/m32`, `ret`, `ret imm16`), but the architectural contract is stable and is what debuggers, unwinders, and ABIs rely on.

1.2 General-Purpose Registers in IA-32

IA-32 exposes eight general-purpose registers (GPRs), each 32-bit wide:

- `EAX`, `EBX`, `ECX`, `EDX`
- `ESI`, `EDI`
- `EBP` (frame/base pointer by convention)
- `ESP` (stack pointer)

These registers have historical partial-register aliases:

- `EAX` → `AX` → `AH`/`AL`

- `EBX` → `BX` → `BH/BL`   (and similarly for `ECX`, `EDX`)
- `ESI` → `SI`, `EDI` → `DI`, `EBP` → `BP`, `ESP` → `SP`

ABI relevance: calling conventions define *who must preserve what*. A typical IA-32 contract is:

- **Caller-saved (volatile):** `EAX`, `ECX`, `EDX`
- **Callee-saved (non-volatile):** `EBX`, `ESI`, `EDI`, `EBP`

This division is not a stylistic preference. It is what allows separately compiled modules to call each other safely.

1.2.1 Example: Correct Register Preservation

Suppose a function uses `EBX`. If the ABI says `EBX` is callee-saved, the function must restore it before returning.

```
# int uses_ebx(int x)
# Returns: x + 1, but uses EBX internally.
uses_ebx:
    push ebp
    mov  ebp, esp

    push ebx                  # preserve callee-saved EBX
    mov  ebx, dword ptr [ebp+8]   # x
    add  ebx, 1
    mov  eax, ebx             # return in EAX
    pop  ebx                  # restore EBX

    pop  ebp
    ret
```

If you omit the save/restore and the caller expected `EBX` to remain unchanged, the failure can show up later in unrelated code (the most dangerous kind of bug: **silent corruption**).

1.3 Stack Pointer (`ESP`) and Base Pointer (`EBP`)

IA-32 function calling is fundamentally stack-based. Two registers dominate this model:

- **`ESP`:** points to the top of the current stack (the next location affected by `push`/`pop`).
- **`EBP`:** commonly used as a stable *frame pointer* to reference arguments and locals at fixed offsets.

1.3.1 Canonical Frame Setup

A classic prologue establishes `EBP` as a stable anchor:

```
# Prologue (canonical)
push ebp
mov  ebp, esp
sub  esp, 32       # reserve 32 bytes for locals (example)
```

With this frame, the layout is conceptually:

- `[ebp+4]` : return address
- `[ebp+8]` : first argument
- `[ebp+12]` : second argument
- `[ebp-4]`, `[ebp-8]`, ... : local variables

1.3.2 Example: Stack Frame With Locals and Two Arguments

Consider a C-style function:

```
extern "C" int __cdecl f(int a, int b);
```

A typical ABI-correct assembly implementation:

```
# int f(int a, int b)
# a at [ebp+8], b at [ebp+12]
f:
    push ebp
    mov  ebp, esp
    sub  esp, 8                 # reserve 8 bytes for locals

    mov  eax, dword ptr [ebp+8]   # eax = a
    mov  dword ptr [ebp-4], eax   # local0 = a

    mov  eax, dword ptr [ebp+12]  # eax = b
    add  eax, dword ptr [ebp-4]   # eax = b + local0

    mov  esp, ebp
    pop  ebp
    ret
```

Key point: `EBP`-relative addressing provides stability even as `ESP` moves due to pushes, pops, and local allocation.

1.3.3 Frame Pointer Omission (FPO)

Optimizing compilers may omit `EBP` as a frame pointer to free a register and shorten prologues/epilogues. When this happens, arguments and locals are addressed relative to `ESP`, and offsets can vary across the function as `ESP` changes. For ABI reasoning, the *external contract is unchanged*; only the internal addressing strategy changes.

1.4 Instruction Pointer (`EIP`) and Control Flow

`EIP` holds the address of the next instruction to execute. Control-flow instructions modify `EIP` directly or indirectly:

- **Direct control flow:** `jmp label`, `call label`
- **Indirect control flow:** `jmp r/m32`, `call r/m32`
- **Return:** `ret` (pops target from the stack)
- **Conditional flow:** `jcc label` based on flags

1.4.1 Example: Why the Return Address Matters

A single stack imbalance can break `ret`, because `ret` trusts the top of stack to be the correct return address.

```
# Wrong: mismatched stack cleanup (conceptual)
# If a callee uses 'ret 8' (callee cleanup) but the caller also adds
 ↪  esp, 8,
# ESP will skip over the real return address at some later point.
```

This is why calling convention mismatches often crash at `ret` or shortly after: `EIP` is loaded from an invalid location, transferring control to nonsense.

1.4.2 Direct vs Indirect `call`

Indirect calls are central to C++ virtual dispatch and function pointers.

```
# Direct call
call do_work

# Indirect call via pointer in EAX
call eax

# Indirect call via memory operand
call dword ptr [ebp-4]      # call through local function pointer
```

From an ABI standpoint, these calls still push a return address and transfer control; argument passing and register preservation rules still apply.

1.5 Stack Growth and Alignment Rules

The IA-32 stack grows **downward** toward lower addresses. A `push` decrements `ESP` and stores a value; a `pop` loads a value and increments `ESP`.

1.5.1 Push/Pop Mechanics

```
# push r32:
#   esp = esp - 4
#   [esp] = r32

# pop r32:
#   r32 = [esp]
#   esp = esp + 4
```

1.5.2 Alignment: The Practical Rule

Alignment requirements can vary by ABI and toolchain, but the engineering rule is:

- Maintain stack alignment as required at **call boundaries**.
- Do not assume you may freely misalign `ESP` inside a function if you will call other functions.

Some code works *until* a callee uses instructions or conventions that assume stronger alignment, at which point you see crashes or data faults that look unrelated. Therefore, this booklet enforces a discipline: **treat stack alignment as part of the ABI contract.**

1.5.3 Example: Enforcing Alignment Before a Call

The following example shows a conservative alignment strategy: adjust stack space so that after pushing arguments, the stack remains aligned as required for the environment, then undo the adjustment after the call.

```
# Conceptual pattern:
#   1) reserve alignment padding if needed
#   2) push arguments
#   3) call target
#   4) clean arguments + padding (cdecl-style)

caller_example:
    push ebp
    mov  ebp, esp

    sub  esp, 12             # padding/reserved space (example
     ↪  strategy)
```

```
push 2                     # arg2
push 1                     # arg1
call callee
add  esp, 8                # caller cleans 2 args (cdecl)

add  esp, 12               # remove padding/reserved space

pop  ebp
ret
```

Important: the exact padding amount and the required alignment depend on the target ABI and the caller's stack state. The stable lesson is the method: *alignment is maintained at call boundaries*, because that is where interoperability is tested.

1.5.4 A Minimal Stack-Integrity Checklist

When you debug IA-32 interoperability issues, check these in order:

1. Are caller and callee using the **same calling convention** (cleanup responsibility matches)?
2. Does the callee preserve all **callee-saved registers** required by the ABI?
3. Are argument offsets correct relative to the established frame (`EBP` or `ESP`)?
4. Is stack alignment maintained correctly at **every call boundary**?

If these rules are satisfied, IA-32 calls across C, C++, and assembly become mechanically reliable rather than fragile.

Chapter 2

The IA-32 Stack in Practice

2.1 Stack Layout at Function Entry

In IA-32, the stack is the primary transport for **control flow** (return addresses) and, in common conventions, for **arguments**. At the moment a function begins executing, the stack already contains at least the **return address**. If the caller passed arguments on the stack, they are located above that return address.

2.1.1 Immediate Effects of `call`

Architecturally, a near `call` performs two essential actions:

1. Push the return address on the stack.

2. Load `EIP` with the call target.

```
# IA-32 conceptual: call target
#   esp = esp - 4
#   [esp] = return_address
#   eip = target
```

2.1.2 Typical Entry Stack Layout (Stack Arguments)

Assume a common stack-argument convention (e.g., cdecl/stdcall style) with two 32-bit integer arguments. Right after the callee begins (before it changes `ESP`), the layout is conceptually:

```
# On entry to callee (before prologue), conceptual:
#   [esp+0]  = return_address
#   [esp+4]  = arg1
#   [esp+8]  = arg2
```

If the function establishes a frame pointer using `EBP`, these become fixed offsets from `EBP`:

```
# After:
#   push ebp
#   mov  ebp, esp
#
# Layout:
#   [ebp+4]  = return_address
#   [ebp+8]  = arg1
#   [ebp+12] = arg2
```

2.1.3 Example: Reading Arguments Correctly

Goal: implement `int add2(int a, int b)` with ABI-correct stack usage.

```
# int add2(int a, int b)
# a = [ebp+8], b = [ebp+12], return in eax
add2:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # a
    add  eax, dword ptr [ebp+12]     # + b

    pop  ebp
    ret
```

This example is intentionally minimal: it demonstrates the most important invariant for correctness across separately compiled code: **arguments are read from the ABI-defined locations**, not from assumptions.

2.2 Stack Frames and Prologue/Epilogue Patterns

A **stack frame** is the region of stack memory a function uses for:

- saving the caller's state (e.g., `EBP`, preserved registers),
- local variables,
- temporary storage and spill slots,
- outgoing call arguments (in some compiler strategies).

2.2.1 Canonical Frame (Debug-Friendly, Stable Offsets)

The classic IA-32 prologue/epilogue pattern:

```
# Prologue
push ebp
```

```
mov  ebp, esp
sub  esp, N          # allocate N bytes for locals

# Epilogue (one of the common forms)
mov  esp, ebp
pop  ebp
ret
```

Why it matters: `EBP` provides stable addressing even if `ESP` changes due to pushes/pops or dynamic allocation.

2.2.2 Compact Epilogue Using `leave`

Many assemblers and compilers use `leave`, which is equivalent to: `mov esp, ebp` then `pop ebp`.

```
# Equivalent epilogue
leave
ret
```

2.2.3 Frame Pointer Omission (FPO) Pattern

Optimizing builds may omit the frame pointer and address locals/args relative to `ESP`. This is correct but harder to reason about manually because `ESP` can move.

```
# Example shape only (offsets are compiler-dependent):
#   sub esp, N
#   mov eax, dword ptr [esp + K]   # arg/local via esp-relative
↪  addressing
```

For ABI work, remember: **FPO changes internal bookkeeping, not the external calling contract**.

2.3 Saved Registers and Caller vs Callee Responsibilities

Every calling convention is a contract that defines:

- **where arguments live** (stack, registers, or both),
- **who cleans the arguments** (caller or callee),
- **which registers must be preserved** across the call.

2.3.1 Register Volatility: The Interoperability Core

A widely used IA-32 division is:

- **Caller-saved (volatile):** `EAX`, `ECX`, `EDX`
- **Callee-saved (non-volatile):** `EBX`, `ESI`, `EDI`, `EBP`

Meaning:

- If the caller needs a volatile register after a call, the caller must save it before the call.
- If the callee wants to use a non-volatile register, it must save and restore it.

2.3.2 Example: Caller Saves Volatile Registers

Caller wants to preserve a value held in `ECX` across a function call:

```
# Caller side
caller:
    push ebp
    mov  ebp, esp
    mov  ecx, 1234              # important value in a volatile
     ↪  register
    push ecx                    # caller saves ECX
    push 2
    push 1
    call add2
    add  esp, 8                 # caller cleans args (cdecl-style)
    pop  ecx                    # restore ECX

    pop  ebp
    ret
```

This is correct regardless of what the callee does, because the ABI says `ECX` is not preserved by the callee.

2.3.3 Example: Callee Preserves Non-Volatile Registers

Callee uses `EBX` (non-volatile) and must restore it:

```
# int use_ebx(int x)
use_ebx:
    push ebp
    mov  ebp, esp

    push ebx                    # callee preserves EBX
    mov  ebx, dword ptr [ebp+8]
    add  ebx, 10
    mov  eax, ebx               # return in EAX
    pop  ebx                    # restore EBX
    pop   ebp
    ret
```

If EBX is not restored, the caller may malfunction later in ways that are difficult to attribute to the call site.

2.3.4 Stack Cleanup Responsibility (Caller vs Callee)

There are two major cleanup models for stack arguments:

- **Caller cleanup:** caller adjusts ESP after the call (common for cdecl, required for variadic).
- **Callee cleanup:** callee adjusts ESP before returning (common for stdcall).

```
# Caller cleanup pattern (cdecl-style)
push arg2
push arg1
call func
add  esp, 8      # caller removes 2 args (2 * 4 bytes)

# Callee cleanup pattern (stdcall-style, conceptual)
push arg2
push arg1
call func
# callee returns with ret 8 (pops return address and discards 8 bytes
↪  of args)
```

Rule: if caller and callee disagree on cleanup, the stack becomes unbalanced and ret will eventually jump to the wrong address.

2.4 Local Variables and Temporary Storage

Locals and temporaries live in the callee's stack frame. Compilers use stack storage for:

- local variables (`int local;`),
- spilled registers (when registers are insufficient),
- temporary values created during expression evaluation,
- outgoing arguments for nested calls (in some strategies).

2.4.1 Example: Locals at Negative Offsets

A function with two locals (8 bytes total) uses `[ebp-4]` and `[ebp-8]`.

```
# int g(int a)
# locals: local0 at [ebp-4], local1 at [ebp-8]
g:
    push ebp
    mov  ebp, esp
    sub  esp, 8

    mov  eax, dword ptr [ebp+8]      # a
    mov  dword ptr [ebp-4], eax      # local0 = a
    lea  eax, dword ptr [eax+eax]    # eax = 2*a
    mov  dword ptr [ebp-8], eax      # local1 = 2*a

    mov  eax, dword ptr [ebp-4]
    add  eax, dword ptr [ebp-8]      # return local0 + local1
    mov  esp, ebp
    pop  ebp
    ret
```

This yields a stable, debuggable layout: arguments at positive offsets, locals at negative offsets.

2.4.2 Spill Slots: When the Compiler Runs Out of Registers

Even hand-written assembly often needs temporary stack storage when preserving values across calls.

```
# Example: preserve an intermediate across a call by storing it on
↪  the stack
h:
    push ebp
    mov  ebp, esp
    sub  esp, 4

    mov  eax, 777
    mov  dword ptr [ebp-4], eax      # spill: save intermediate

    push 2
    push 1
    call add2
    add  esp, 8

    add  eax, dword ptr [ebp-4]      # use spilled value after call

    mov  esp, ebp
    pop   ebp
    ret
```

This is the simplest reliable pattern when you must protect a value from being overwritten by a call (especially if that value would otherwise reside in a volatile register).

2.5 Stack Alignment Constraints

Alignment is part of interoperability. While IA-32 is tolerant of many unaligned memory accesses, **ABIs and generated code may assume specific stack alignment at call boundaries**. Misalignment can cause:

- slower accesses (penalties on some microarchitectures),
- faults in code that uses stronger-alignment instructions or conventions,
- subtle corruption when a callee assumes an alignment that the caller did not maintain.

2.5.1 Practical Rule for ABI-Safe Calls

Treat stack alignment as a **call-site invariant**:

- before calling another function, ensure `ESP` meets the platform's ABI alignment requirement,
- after returning, restore `ESP` exactly to the value the caller expects (balanced pushes/pops and correct cleanup).

2.5.2 Example: Balanced Stack Discipline

The strongest universal discipline for correctness is **balance**: every `push` must be matched by a corresponding stack adjustment before returning, and cleanup responsibility must match the chosen convention.

```
# Caller: balanced stack, caller cleanup (cdecl-style)
balanced_call:
    push ebp
    mov  ebp, esp

    push 20
    push 10
    call add2
    add  esp, 8          # remove two arguments (2 * 4)

    pop  ebp
    ret
```

If you follow balanced stack discipline *and* preserve registers per the ABI, your IA-32 assembly becomes reliably interoperable with C and C++ across toolchains and build modes.

2.5.3 A Minimal Stack Correctness Checklist

For every function and every call site, verify:

1. **Entry invariants:** return address is at the expected location; arguments are at ABI-defined locations.
2. **Frame correctness:** if using `EBP`, establish and tear down the frame correctly.
3. **Register contract:** callee preserves all required non-volatile registers it modifies.
4. **Cleanup contract:** caller vs callee cleanup matches the convention (never both, never neither).
5. **Alignment at calls:** maintain the ABI-required stack alignment at every call boundary.

Chapter 3

IA-32 Calling Conventions

3.1 Overview of Common IA-32 ABIs

An IA-32 **calling convention** is an ABI contract that answers five questions:

1. **Where are arguments placed?** (stack, registers, or both)
2. **In what order are stack arguments placed?** (typically right-to-left)
3. **Who cleans the stack arguments?** (caller or callee)
4. **Which registers must be preserved?** (callee-saved vs caller-saved)
5. **How are return values produced?** (registers, hidden pointers for aggregates)

In practice, most real-world IA-32 code you will analyze falls into a small set of conventions:

- **cdecl:** stack arguments, caller cleanup, flexible (required for variadic).
- **stdcall:** stack arguments, callee cleanup, common in many OS/API boundaries.

- **fastcall:** some arguments in registers, remaining on stack, cleanup rules vary by toolchain.

The **most important engineering principle** is not memorizing names, but proving correctness by inspecting:

- call site stack adjustments (e.g., `add esp, N` after `call`),
- return instruction form (e.g., `ret` vs `ret imm16`),
- register save/restore behavior.

3.1.1 A Unifying Mental Model

Think of the call boundary as a handshake:

```
# Caller prepares inputs  ->  Callee consumes inputs
# Caller expects outputs  <-  Callee produces outputs
# Both sides obey:
#   - stack cleanup responsibility
#   - register preservation rules
```

3.2 `cdecl` Calling Convention

cdecl is the classic C calling convention on IA-32 in many toolchains.

3.2.1 Core Rules

- **Arguments:** passed on the stack (typically right-to-left).
- **Stack cleanup: caller** cleans the stack arguments after the call.

- **Return value:** integer/pointer returns in `EAX`.
- **Variadic support: required** for variadic functions because only the caller knows how many arguments were pushed.

3.2.2 Example: `cdecl` Caller and Callee

C signature:

```
extern "C" int __cdecl sum3(int a, int b, int c);
```

Caller side (push right-to-left, caller cleanup):

```
# int r = sum3(10, 20, 30);
push 30                 # c
push 20                 # b
push 10                 # a
call sum3
add  esp, 12            # caller removes 3 args (3 * 4)
# result now in eax
```

Callee side (reads args at fixed `EBP` offsets, returns in `EAX`):

```
# int __cdecl sum3(int a, int b, int c)
sum3:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # a
    add  eax, dword ptr [ebp+12]     # + b
    add  eax, dword ptr [ebp+16]     # + c
    pop   ebp
    ret                             # callee does NOT discard args
```

3.2.3 Variadic Function Reality

For variadic functions, the callee cannot discard arguments because it does not know how many were pushed. Therefore, **caller cleanup is mandatory**.

```
# Caller knows argument count, so caller cleanup is the only safe
 ↪  rule.
# push fmt, push arg1, push arg2, ...
# call printf_like
# add esp, N
```

3.3 `stdcall` Calling Convention

stdcall is widely used for fixed-parameter APIs where a stable binary boundary is desired.

3.3.1 Core Rules

- **Arguments:** passed on the stack (typically right-to-left).
- **Stack cleanup: callee** cleans the stack arguments.
- **Return value:** integer/pointer returns in `EAX`.
- **Fixed parameter count:** expected (not suitable for true variadic interfaces).

3.3.2 Key Signature in Assembly: `ret imm16`

The common mechanical difference is in the return instruction:

- `ret` pops only the return address.
- `ret 12` pops the return address *and* discards 12 bytes of arguments.

3.3.3 Example: `stdcall` Callee Cleanup

C signature:

```
extern "C" int __stdcall sum3_std(int a, int b, int c);
```

Caller side (no `add esp, 12`):

```
push 30
push 20
push 10
call sum3_std
# no stack adjustment here (callee discards args)
```

Callee side (returns with `ret 12`):

```
# int __stdcall sum3_std(int a, int b, int c)
sum3_std:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    add  eax, dword ptr [ebp+16]
    pop   ebp
    ret   12                    # callee discards 3 args (3 * 4)
```

3.3.4 The Classic Failure Mode

If a caller also does `add esp, 12` after calling a `stdcall` callee, the stack becomes imbalanced. The failure often appears later when a `ret` loads an incorrect return address.

3.4 `fastcall` Calling Convention

fastcall is a performance-oriented convention that reduces memory traffic by passing some arguments in registers.

3.4.1 Core Rules (General Form)

- **First arguments:** passed in registers (toolchain-defined; commonly `ECX`, `EDX` for the first two integer/pointer args).
- **Remaining arguments:** passed on the stack.
- **Cleanup:** toolchain-defined; in many cases callee cleanup is used for fixed signatures, but you must verify at the binary boundary.
- **Return value:** integer/pointer returns in `EAX`.

3.4.2 Example: Two Args in Registers

Signature:

```
extern "C" int __fastcall add2_fast(int a, int b);
```

Typical call shape (conceptual):

```
mov  ecx, 10              # a
mov  edx, 20              # b
call add2_fast
# result in eax
```

Callee reads from registers instead of stack:

```
# int __fastcall add2_fast(int a, int b)
# a in ecx, b in edx (common fastcall form)
add2_fast:
    push ebp
    mov  ebp, esp

    mov  eax, ecx
    add  eax, edx

    pop  ebp
    ret
```

3.4.3 Mixing Register and Stack Arguments

With more than two integer/pointer args, remaining args go on the stack. The exact offsets depend on whether the callee sets up `EBP` and whether there is additional ABI-required bookkeeping. The **safe method** is: verify by reading the call site and the callee simultaneously.

3.5 Compiler-Specific Variations

On IA-32, calling convention names can hide **toolchain-specific** rules. Variations commonly appear in:

- **Register assignment:** which registers receive the first arguments under `fastcall`-like rules.
- **Name decoration:** how symbols are exported and how the convention is encoded in symbol names.
- **Struct return conventions:** small aggregates in registers vs hidden pointers for larger aggregates.
- **Frame pointer usage:** `EBP`-framed vs frame-pointer omission, changing how offsets appear.
- **Alignment policy:** required alignment at call boundaries and how compilers maintain it.

3.5.1 Example: Detect Convention by Mechanics (Not by Guessing)

You can often infer the convention from the callee's return instruction and the caller's stack adjustment:

```
# If you see:
#   call foo
#   add esp, 8
# then caller cleanup is happening -> cdecl-like for two stack args.

# If you see:
#   call foo
# and NO add esp, 8 at the call site, but callee ends with:
#   ret 8
# then callee cleanup is happening -> stdcall-like for two stack
↪  args.
```

For register-passing conventions, you will see argument setup in registers at the call site:

```
# Register argument setup often indicates fastcall-like behavior:
mov ecx, 1
mov edx, 2
call foo
```

3.6 When and Why Conventions Differ

Calling conventions differ to optimize for different constraints:

3.6.1 Variadic vs Fixed Signatures

Variadic interfaces require caller cleanup because only the caller knows how many arguments were passed. Fixed signatures can choose caller or callee cleanup.

3.6.2 Performance and Code Size

Register passing reduces memory traffic and can improve performance. Callee cleanup can reduce caller code size for fixed signatures by removing `add esp, N` sequences at each call site.

3.6.3 Binary Compatibility Boundaries

System APIs and stable binary interfaces often standardize a convention to ensure that independently compiled modules can interoperate safely across versions.

3.6.4 Language Semantics

C++ introduces additional ABI realities:

- member functions may have an implicit object pointer (commonly passed in a register or as a hidden argument depending on ABI),
- exceptions and RTTI involve additional runtime mechanisms (not covered here), but they influence how toolchains standardize ABI choices.

3.6.5 The Practical Rule for Engineers

You never “choose” a calling convention in isolation. You choose it because you must match an existing ABI boundary: an OS API, a library, a compiler default, or a previously shipped binary interface. In interoperability work, correctness comes from **matching the boundary**, then verifying by the mechanical evidence in the call sites and returns.

Chapter 4

Passing Arguments on the Stack

4.1 Argument Order (Right-to-Left Rule)

For the dominant IA-32 stack-based conventions, stack arguments are placed **right-to-left**: the *last* source-level argument is pushed first, so the *first* argument ends up closest to the return address. This rule makes **fixed offsets** from a frame pointer (`EBP`) stable across call sites and enables C-style variadic functions to locate the first named argument reliably.

4.1.1 Example: Two Arguments, Right-to-Left

Source intent:

```
extern "C" int __cdecl f(int a, int b);
int r = f(10, 20);
```

Typical stack setup:

```
# Push right-to-left:
push 20                    # b
push 10                 # a
call f
add  esp, 8             # caller cleanup for cdecl-style
```

Callee view after establishing `EBP`:

```
# On entry after:
#   push ebp
#   mov  ebp, esp
#
# [ebp+8]  = a
# [ebp+12] = b
```

4.1.2 Why This Order Matters (Variadic Reality)

With variadic functions, the callee must locate the first named argument and then walk to additional arguments using fixed-width rules and type promotions. A consistent placement rule is essential.

4.2 Primitive Types on the Stack

In IA-32 ABIs, the stack is addressed in **32-bit units** (4 bytes). Most primitive integer-like arguments are passed as 4-byte values on the stack, even when the source type is smaller, due to standard argument passing rules and promotions in many calling contexts.

4.2.1 32-bit Units and Stack Slots

A practical rule for ABI reasoning in IA-32:

- Each stack argument occupies one or more **4-byte slots**.

- Smaller integer types (`char`, `short`) are typically **extended** to a 4-byte value when placed on the stack for calls.
- 64-bit integers occupy **two slots** (8 bytes) and are laid out as two adjacent 32-bit words.

4.2.2 Example: `char`, `short`, `int`

Source intent:

```
extern "C" int __cdecl g(char c, short s, int i);
```

Conceptual stack layout (after `EBP` is set):

```
# [ebp+8]  = c extended to 32-bit
# [ebp+12] = s extended to 32-bit
# [ebp+16] = i (32-bit)
```

Callee reading them as 32-bit values:

```
g:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]     # c (extended)
    mov  edx, dword ptr [ebp+12]    # s (extended)
    add  eax, edx
    add  eax, dword ptr [ebp+16]    # + i

    pop  ebp
    ret
```

4.2.3 Example: 64-bit Integer Argument

Source intent:

```
extern "C" int __cdecl h(long long x);
```

A 64-bit integer occupies two 32-bit stack slots. The callee sees adjacent words at fixed offsets:

```
# Conceptual:
# [ebp+8]  = low 32 bits of x
# [ebp+12] = high 32 bits of x
```

Reading both parts:

```
h:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]     # low
    mov  edx, dword ptr [ebp+12]    # high
    # Now EDX:EAX holds x in a common IA-32 pairing model
    # (how you use it depends on your goal)

    pop  ebp
    ret
```

4.3 Pointers and References

On IA-32, **pointers are 32-bit** values. They occupy one 4-byte stack slot and are passed like integers.

4.3.1 Pointers

Source intent:

```
extern "C" int __cdecl sum2_ptr(const int* p);
```

Callee receives `p` as a 32-bit address at `[ebp+8]`:

```
sum2_ptr:
    push ebp
    mov  ebp, esp

    mov  edx, dword ptr [ebp+8]      # edx = p
    mov  eax, dword ptr [edx]        # eax = p[0]
    add  eax, dword ptr [edx+4]      # eax += p[1]

    pop  ebp
    ret
```

4.3.2 References (C++ ABI Reality)

In practice, a C++ reference parameter is passed as an **address** (like a pointer) at the ABI boundary. This is why references are ABI-friendly: they have a concrete machine representation.

```
extern "C" int __cdecl inc_ref(int& x);
```

```
# int inc_ref(int& x)  # x is received as a pointer-like address
inc_ref:
    push ebp
    mov  ebp, esp
    mov   edx, dword ptr [ebp+8]      # edx = &x
    mov   eax, dword ptr [edx]        # eax = x
    add   eax, 1
    mov   dword ptr [edx], eax        # x = x + 1

    pop   ebp
    ret
```

The C++ type system enforces non-null semantics at the source level, but at the ABI level it is still an address.

4.4 Floating-Point Arguments

Floating-point argument passing in IA-32 has two important realities:

- In many classic IA-32 ABIs, floating-point arguments are **passed on the stack** as their memory representation (`float` 4 bytes, `double` 8 bytes).
- The computation may use x87 stack registers or SSE registers internally depending on compiler and settings, but the **call boundary** still frequently uses stack slots for floats/doubles in stack-based conventions.

4.4.1 Example: `double` Occupies Two Stack Slots

Source intent:

```
extern "C" int __cdecl take_double(double x);
```

A `double` is 8 bytes, so it occupies two 4-byte stack slots:

```
# Conceptual layout:
# [ebp+8]  = low 32 bits of x
# [ebp+12] = high 32 bits of x
```

Loading the raw bits (for demonstration) without performing floating arithmetic:

```
take_double:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]     # low  part
    mov  edx, dword ptr [ebp+12]    # high part
    # EDX:EAX now holds the raw 64-bit payload of x

    pop  ebp
    ret
```

4.4.2 Example: Passing a `double` from Caller

Caller reserves 8 bytes and stores the value, then passes it by pushing the two words (or by reserving and copying, depending on compiler strategy). A conceptual push-based view:

```
# Conceptual only: pushing the 64-bit value as two 32-bit words.
# The exact order is ABI-defined; treat as two adjacent stack slots.
push dword ptr [x_high]    # high 32
push dword ptr [x_low]     # low  32
call take_double
add  esp, 8
```

In real compiler output, you often see stack space reserved then a store into `[esp]` and `[esp+4]` before the call. The ABI-relevant point remains: **the callee sees two adjacent 32-bit words**.

4.5 Structs Passed by Value

Passing structs by value is where IA-32 ABIs become visibly strict. Two rules dominate:

- Small aggregates may be passed in one or more stack slots directly (by copying their bytes onto the stack).
- Larger aggregates are still passed by value *semantically*, but the mechanism is a **copy onto the caller's stack** (or sometimes a hidden pointer depending on ABI/toolchain rules).

4.5.1 Example: 8-Byte Struct by Value

Assume:

```
struct Pair { int x; int y; };
extern "C" int __cdecl sum_pair(Pair p);
```

The struct is 8 bytes, so it occupies two stack slots. Callee reads fields at fixed offsets:

```
# Conceptual layout after EBP set:
# [ebp+8]  = p.x
# [ebp+12] = p.y
```

```
sum_pair:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # p.x
    add  eax, dword ptr [ebp+12]     # + p.y

    pop  ebp
    ret
```

4.5.2 Example: Larger Struct by Value (Copy Semantics)

For a larger struct, the caller typically **copies the struct bytes** into the outgoing argument area on the stack (or into a temporary then onto the stack). The callee still reads its fields from the stack argument region.

```
struct Big { int a; int b; int c; int d; }; // 16 bytes
extern "C" int __cdecl sum_big(Big v);
```

Callee view:

```
# After EBP set:
# [ebp+8]  = v.a
# [ebp+12] = v.b
# [ebp+16] = v.c
# [ebp+20] = v.d
```

```
sum_big:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    add  eax, dword ptr [ebp+16]
    add  eax, dword ptr [ebp+20]

    pop  ebp
    ret
```

The ABI discipline is the same: the callee expects a contiguous, correctly aligned memory image of the struct in the argument region.

4.6 Hidden Arguments and ABI Details

Not all parameters appear explicitly in the source signature. ABIs commonly introduce **hidden arguments** to implement language semantics and efficiency. In IA-32 interoperability work, the two most important hidden-argument cases are:

- **Structure return (sret):** when returning a struct by value that cannot be returned in registers, the caller passes a hidden pointer to return storage.
- **C++ implicit object pointer (`this`):** member functions receive an implicit pointer to the object; its placement depends on the ABI and convention used for member functions.

4.6.1 Example: Hidden Pointer for Struct Return (sret)

Source intent:

```
struct Pair { int x; int y; };
extern "C" Pair __cdecl make_pair(int a, int b);
```

A common ABI strategy: the caller allocates space for the return object and passes a hidden first argument: `Pair* out`. The logical signature becomes conceptually:

```
extern "C" void __cdecl make_pair_sret(Pair* out, int a, int b);
```

Callee view after `EBP` is established:

```
# [ebp+8]  = out (hidden pointer)
# [ebp+12] = a
# [ebp+16] = b
```

Assembly implementation:

```
# void make_pair_sret(Pair* out, int a, int b)
make_pair:
    push ebp
    mov  ebp, esp

    mov  edx, dword ptr [ebp+8]       # out
    mov  eax, dword ptr [ebp+12]      # a
    mov  dword ptr [edx], eax         # out->x = a
    mov  eax, dword ptr [ebp+16]      # b
    mov  dword ptr [edx+4], eax       # out->y = b

    # Return convention for sret commonly returns 'out' in eax, or
     ↪  returns void
    # depending on toolchain/ABI. The critical ABI point is the
     ↪  hidden pointer.
    mov  eax, edx                     # often: return out pointer in
     ↪  eax

    pop  ebp
    ret
```

When you debug C/C++ interoperability, recognizing the **hidden sret pointer** is essential; otherwise argument offsets appear “shifted” and developers misdiagnose the crash as stack corruption.

4.6.2 Example: C++ `this` as a Hidden Argument

For a non-static member function, the machine-level call includes an implicit object pointer. Conceptually:

```
struct Obj { int v; int get() const; };
```

The member call `o.get()` is conceptually like:

```
int Obj_get(const Obj* this_ptr);
```

How `this` is passed (stack vs register) depends on the ABI and convention for member functions in that toolchain. The core lesson for this chapter is: **hidden arguments exist**, and they change stack offsets and register usage. Therefore, always confirm ABI lowering by inspecting both the call site and the callee.

Chapter 5

Stack Cleanup Rules

5.1 Caller Cleanup vs Callee Cleanup

In IA-32 stack-based calling conventions, **stack cleanup** defines who restores `ESP` after arguments are placed on the stack for a call. This rule is a core ABI contract, because `ret` trusts that the stack top contains the correct return address.

5.1.1 Caller Cleanup (cdecl-style)

Caller cleanup means:

- The caller pushes arguments.
- The callee returns with `ret` (pops only the return address).
- The caller restores `ESP` by discarding the argument bytes.

```
# Caller cleanup pattern (cdecl-style)
push argN
```

```
...
push arg2
push arg1
call callee
add  esp, N              # caller discards N bytes of args
```

5.1.2 Callee Cleanup (stdcall-style)

Callee cleanup means:

- The caller pushes arguments.
- The callee returns with `ret imm16`, which pops the return address and discards the argument bytes.
- The caller performs no argument discard.

```
# Callee cleanup pattern (stdcall-style)
push argN
...
push arg2
push arg1
call callee
# no add esp, N here
```

```
# Callee return discards args:
ret  N                   # pop return address + discard N bytes
```

Key invariant: cleanup must happen *exactly once*. If both sides clean, the stack moves too far. If neither cleans, the stack grows and eventually breaks.

5.2 Stack Adjustment Instructions

The IA-32 instruction set provides small, reliable primitives to adjust the stack. These are the basic tools used by compilers and by ABI-correct hand-written assembly.

5.2.1 `add esp, imm`

The most common caller-cleanup form: discard a known number of bytes.

```
# Discard 3 arguments (3 * 4 = 12 bytes)
add  esp, 12
```

5.2.2 `ret` vs `ret imm16`

`ret` pops the return address only. `ret imm16` pops the return address and discards an immediate byte count.

```
ret                         # pop return address only
ret  8                      # pop return address + discard 8 bytes of
↪  args
```

5.2.3 `leave`

`leave` is a canonical epilogue helper for framed functions: it restores `ESP` from `EBP` then pops the old `EBP`.

```
leave                       # mov esp, ebp ; pop ebp
ret
```

5.2.4 Extensive Example: Two Equivalent Epilogues

```
# Form A: explicit
mov  esp, ebp
pop  ebp
ret

# Form B: using leave
leave
ret
```

Both are ABI-correct if the function established `EBP` as a frame pointer.

5.3 Common Bugs from Mismatched Cleanup

Stack-cleanup mismatches produce some of the most deceptive failures in IA-32. Often the crash happens *later*, far from the call site that caused the imbalance.

5.3.1 Bug Type 1: Double Cleanup (Caller + Callee)

Caller performs `add esp, N`, but callee also returns with `ret N`. The stack advances too far.

```
# Caller (wrong for stdcall callee)
push 2
push 1
call callee_stdcall
add  esp, 8              # WRONG: callee already discarded args
```

```
# Callee (stdcall style)
```

```
callee_stdcall:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    pop  ebp
    ret  8                  # callee cleanup
```

Typical symptom: later `ret` jumps to an invalid address because the return address is no longer at the expected stack top.

5.3.2 Bug Type 2: No Cleanup (Neither Side)

Caller fails to discard args, and callee returns with plain `ret`. The stack pointer drifts downward with every call.

```
# Caller (wrong for cdecl callee)
push 2
push 1
call callee_cdecl
# missing: add esp, 8
```

```
# Callee (cdecl style)
callee_cdecl:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    pop  ebp
    ret
```

Typical symptom: program runs for a while, then fails due to stack exhaustion, corrupted locals, or a crash in unrelated code.

5.3.3 Bug Type 3: Wrong Byte Count

Even when the right side cleans, using the wrong byte count is enough to corrupt the call stack.

```
# Caller expects to discard 12 bytes but discards only 8
add  esp, 8                # WRONG: leaves one argument on stack
```

Symptom pattern: the stack becomes gradually misaligned, and a later `ret` loads the wrong return address or a later function reads the wrong argument values.

5.3.4 Extensive Diagnostic Trick: Watch `ESP` Across Call Boundaries

A reliable debugging method is to observe `ESP` immediately:

- before pushing arguments,
- right before `call`,
- immediately after returning.

If cleanup is correct, `ESP` after the full call sequence equals the value before pushing arguments (plus/minus only intentional temporary allocations in the caller).

5.4 Variadic Functions and Cleanup Constraints

Variadic functions impose a strict constraint: **caller cleanup is required**. The callee cannot know how many arguments were passed, because the argument list length is determined at the call site.

5.4.1 Why the Callee Cannot Clean

In a variadic call, only the caller knows the total number and sizes of the passed arguments after the named parameters. Therefore:

- a callee-cleanup `ret N` is not generally possible,
- a fixed cleanup count would be wrong for most calls.

5.4.2 Extensive Example: Variadic-Style Call Shape

Conceptual example:

```
extern "C" int __cdecl vprint(const char* fmt, ...);
```

```
# vprint("%d %d", 10, 20)
push 20
push 10
push fmt_ptr
call vprint
add  esp, 12             # caller discards all pushed arguments
```

Even if a platform uses `stdcall` for many fixed-parameter APIs, variadic functions still require caller cleanup.

5.5 ABI Guarantees vs Compiler Behavior

The ABI defines the externally visible contract between separately compiled modules. Compilers may transform code internally, but they must preserve the ABI at the boundary.

5.5.1 What the ABI Guarantees

- A caller and callee that agree on the calling convention will interoperate correctly.
- Cleanup responsibility is fixed by the convention at that call boundary.
- Register preservation rules are stable across modules that follow the same ABI.

5.5.2 What Compilers May Change

- **Frame pointer usage:** `EBP` frame vs omission (FPO).
- **Argument setup strategy:** pushing arguments vs reserving stack space and storing into it.
- **Epilogue selection:** `leave` vs explicit restore.
- **Tail calls:** converting `call + ret` into a direct `jmp` when ABI rules allow.

5.5.3 Extensive Example: Push-Based vs Store-Based Argument Setup

Both of the following are ABI-equivalent from the callee's perspective:

```
# Style 1: push arguments
push 2
push 1
call add2
add  esp, 8
```

```
# Style 2: reserve space then store
sub  esp, 8
mov  dword ptr [esp], 1
mov  dword ptr [esp+4], 2
```

```
call add2
add  esp, 8
```

The callee still sees a contiguous argument block in the correct order.

5.5.4 Extensive Example: Tail Call and Cleanup Interaction

A tail call replaces the current function's return with a jump to another function. Cleanup rules must still hold.

```
# Conceptual tail call shape (cdecl-style):
# current function discards its frame and jumps to target,
# letting target return directly to the original caller.

tail_caller:
    push ebp
    mov  ebp, esp
    # ... set up args for target ...
    mov  esp, ebp
    pop  ebp
    jmp  target            # no return here; target returns to
     ↪  original caller
```

A correct tail call requires that the outgoing stack state match what the target expects, including argument placement and alignment. This is why aggressive optimizations do not change the ABI contract; they only change how it is implemented.

5.5.5 Practical Rule

When you analyze binaries, trust the ABI contract first, and verify using mechanics:

- Does the caller adjust `ESP` after the call?

- Does the callee return with `ret imm16`?
- Are argument bytes balanced exactly once?

If these are consistent, the cleanup rules are correct even if the compiler used non-obvious instruction sequences.

Chapter 6

Returning Values in IA-32

6.1 ABI-Mandated Return Registers

In IA-32 ABIs, the return mechanism is part of the calling convention contract. The caller assumes the return value will be placed in specific architectural locations immediately after the callee returns.

6.1.1 Core Return Locations (Practical Summary)

- **32-bit integers and pointers:** returned in `EAX`.
- **64-bit integers:** returned in `EDX:EAX` (high:low pairing).
- **Floating-point:** commonly returned via the x87 `ST(0)` register, or via SSE registers in toolchains/environments that use SSE-based floating ABI rules.
- **Aggregates (struct/class) by value:** returned either in registers (small cases) or via a hidden return-storage pointer (large/common cases).

Engineering rule: always treat return locations as *ABI-defined observable behavior*. If you return a value in the wrong place, the caller will read garbage even though the callee "computed" correctly.

6.2 Integer and Pointer Returns

The simplest and most stable IA-32 rule: **a 32-bit integer or pointer return value is in `EAX`**.

6.2.1 Extensive Example: Returning an `int` in `EAX`

```
# int add1(int x) -> return x+1 in eax
# x at [ebp+8] in a classic framed function
add1:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]
    add  eax, 1

    pop  ebp
    ret
```

6.2.2 Extensive Example: Returning a Pointer in `EAX`

Returning a pointer is identical to returning a 32-bit integer at the ABI level.

```
# int* id_ptr(int* p) -> return p
id_ptr:
    push ebp
    mov  ebp, esp
    mov   eax, dword ptr [ebp+8]      # eax = p

    pop   ebp
    ret
```

6.2.3 Extensive Example: Returning a 64-bit Integer in **EDX:EAX**

A 64-bit integer return uses a two-register pair. A common convention is: `EAX` = low 32 bits, `EDX` = high 32 bits.

```
# long long make64(int lo, int hi) -> returns (hi:lo) in edx:eax
# lo at [ebp+8], hi at [ebp+12]
make64:
    push ebp
    mov   ebp, esp

    mov   eax, dword ptr [ebp+8]      # low
    mov   edx, dword ptr [ebp+12]     # high

    pop   ebp
    ret
```

The caller reads `EDX:EAX` as a 64-bit value immediately after return.

6.3 Floating-Point Return Mechanisms

Floating-point return on IA-32 is historically tied to x87, where the primary return location is **`ST(0)`**. Many modern toolchains can also use SSE registers for floating return in some configurations, but the key ABI reality remains:

- The caller expects the floating return value to appear in the ABI-defined floating return location.
- The callee must ensure the floating pipeline state is consistent at the return boundary.

6.3.1 Extensive Example: Returning a `double` via x87 `ST(0)` (Conceptual)

A minimal shape of a function that returns a floating value through x87 is:

```
# double ret_pi()
# Conceptual: load a floating constant into x87 ST(0), then ret.
# Exact constant loading depends on assembler/sections; the ABI point
↪  is ST(0).
ret_pi:
    push ebp
    mov  ebp, esp

    # Conceptual placeholder:
    # fld qword ptr [pi_const]      # ST(0) = pi

    pop  ebp
    ret
```

This booklet focuses on ABI placement (where the caller reads the result), not on teaching x87 instruction programming in depth.

6.3.2 Extensive Example: Returning a Floating Value as Raw Bits (ABI-Useful Technique)

When verifying interoperability, it is often helpful to return floating payload bits as integers to avoid dependence on x87/SSE details. For example, return a `double`'s 64-bit payload using `EDX:EAX`:

```
# Return the raw 64-bit payload of a double argument in edx:eax
# double x passed on stack as two dwords at [ebp+8] and [ebp+12]
ret_double_bits:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # low 32 of x
    mov  edx, dword ptr [ebp+12]     # high 32 of x

    pop  ebp
    ret
```

This is a robust ABI debugging method: it proves argument placement and return placement without depending on floating execution units.

6.4 Small Structure Returns

Returning small structs by value is ABI-sensitive and can vary by toolchain, but there are stable patterns:

- Some small aggregates are returned in registers (typically using `EAX` and optionally `EDX`).

- If the aggregate does not fit in the ABI's register-return scheme, the ABI switches to a hidden return-storage pointer.

6.4.1 Common Practical Pattern: 8-Byte Aggregate in `EDX:EAX`

A very common small-struct return case is an 8-byte struct (two 32-bit fields). Conceptually, the ABI can return: `EAX` = first 32 bits, `EDX` = second 32 bits.

```
struct Pair { int x; int y; };
extern "C" Pair __cdecl make_pair_small(int a, int b);
```

Assembly returning two fields in registers:

```
# Pair make_pair_small(int a, int b)
# Return:
#   eax = x
#   edx = y
make_pair_small:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # a -> x
    mov  edx, dword ptr [ebp+12]     # b -> y

    pop  ebp
    ret
```

The caller must interpret `EDX:EAX` as the returned struct. If the caller expects an sret-pointer return but the callee returns in registers (or vice versa), results are invalid even if both sides are "correct" internally.

6.5 Large Structure Returns via Hidden Pointers

For larger aggregates, IA-32 ABIs commonly use a **hidden first argument** that points to caller-allocated storage for the return object. This mechanism is often called **sret** (structure return).

6.5.1 Conceptual Lowering

Source signature:

```
struct Big { int a; int b; int c; int d; };   // 16 bytes
extern "C" Big __cdecl make_big(int x);
```

Common ABI lowering:

```
extern "C" void __cdecl make_big_sret(Big* out, int x);
```

The caller allocates `Big` storage and passes its address as a hidden argument.

6.5.2 Extensive Example: sret Implementation

Callee view after `EBP` set:

```
# [ebp+8]  = out (hidden return storage pointer)
# [ebp+12] = x
```

```
# void make_big_sret(Big* out, int x)
make_big:
    push ebp
    mov  ebp, esp
    mov   edx, dword ptr [ebp+8]      # edx = out
    mov   eax, dword ptr [ebp+12]     # eax = x

    mov   dword ptr [edx],     eax    # out->a = x
    add   eax, 1
    mov   dword ptr [edx+4],   eax    # out->b = x+1
    add   eax, 1
    mov   dword ptr [edx+8],   eax    # out->c = x+2
    add   eax, 1
    mov   dword ptr [edx+12],  eax    # out->d = x+3

    # Many ABIs/toolchains also place 'out' in eax as a convenience
     ↪  return value.
    # The correctness-critical ABI element is the hidden pointer
     ↪  itself.
    mov   eax, edx

    pop   ebp
    ret
```

6.5.3 Caller-Side Shape (Conceptual)

The caller reserves storage and passes its address:

```
# Conceptual caller-side:
#   Big tmp;
#   make_big(&tmp, x);
sub  esp, 16             # reserve return storage (example strategy)
lea  eax, dword ptr [esp] # eax = &tmp (return storage)
push x
push eax                    # hidden sret pointer
call make_big
add  esp, 8                 # discard explicit x + hidden pointer
 ↪  (cdecl-style)
# tmp object bytes are now in [esp .. esp+15] (until stack changes)
add  esp, 16                # release storage when done with it
```

Compilers may allocate the return object in caller locals rather than directly on `ESP`, but the ABI principle is identical: **caller provides storage, callee writes into it**.

6.6 Putting It Together: Return Rules You Can Validate Mechanically

When analyzing or writing IA-32 interoperability code, validate returns by inspection:

1. **Integer/pointer return:** confirm final value is in `EAX`.
2. **64-bit integer return:** confirm `EDX:EAX`.
3. **Floating return:** confirm the ABI's floating return location (often x87 `ST(0)` in classic IA-32).
4. **Small aggregate return:** confirm whether the ABI uses registers (commonly `EAX` and `EDX`).
5. **Large aggregate return:** confirm the hidden sret pointer shifts argument offsets and that the callee writes into caller-provided storage.

If the call boundary agreement is correct, the program behaves correctly even under optimization, frame-pointer omission, and different internal code-generation strategies.

Chapter 7

IA-32 Function Prologues and Epilogues

7.1 Canonical Prologue Forms

A **prologue** establishes the callee's stack frame and preserves any callee-saved state the function will modify. A **canonical** (debug-friendly) IA-32 prologue uses `EBP` as a frame pointer, giving stable offsets for arguments and locals.

7.1.1 Minimal Canonical Frame (No Locals)

```
# Canonical prologue/epilogue with EBP frame, no local allocation
func:
    push ebp
    mov  ebp, esp
    # body...
    pop  ebp
    ret
```

7.1.2 Canonical Frame With Locals

Local storage is reserved by subtracting a constant from `ESP`.

```
# Canonical: reserve N bytes for locals (N chosen by compiler/author)
func_locals:
    push ebp
    mov  ebp, esp
    sub  esp, 32            # 32 bytes locals/spills/temps (example)
    # body...
    mov  esp, ebp
    pop  ebp
    ret
```

7.1.3 Canonical Frame With Callee-Saved Register Preservation

If the function modifies non-volatile registers, it must save/restore them.

```
# Uses EBX and ESI (callee-saved), so preserve them
func_saves:
    push ebp
    mov  ebp, esp
    push ebx
    push esi
    sub  esp, 16            # locals

    # body may freely use ebx/esi...

    add  esp, 16            # release locals
    pop  esi
    pop  ebx
    pop  ebp
    ret
```

Key invariant: the epilogue must restore the stack pointer to the exact value expected by the ABI for the chosen cleanup convention and must restore any callee-saved registers it modified.

7.2 Frame Pointer Omission

Frame pointer omission (FPO) means the function does not use `EBP` as a dedicated frame pointer. This frees `EBP` as a general register and can reduce instruction count, but it makes manual analysis harder because `ESP`-relative offsets can change throughout the function.

7.2.1 Typical FPO Shape

```
# FPO-style skeleton: no push ebp / mov ebp, esp
func_fpo:
    sub  esp, 32              # allocate locals
    # body uses [esp + K] addressing
    add  esp, 32
    ret
```

7.2.2 Extensive Example: Addressing Locals With FPO

With FPO, locals and spills are addressed relative to `ESP`. The offsets are compiler-defined, but the key idea is stable:

```
# Example: keep a local at [esp+0] after allocation
func_fpo_local:
    sub  esp, 8
    mov  dword ptr [esp], 123     # local0 = 123
    mov  eax, dword ptr [esp]
    add  esp, 8
    ret
```

Important: if the function performs `push` instructions or makes calls that require temporary stack arguments, `ESP` changes, and the compiler will adjust addressing or use additional bookkeeping to keep references correct.

7.2.3 Practical Consequence for ABI Reasoning

FPO does *not* change:

- the calling convention,
- the return mechanism,
- the register preservation contract.

It only changes how the callee internally addresses stack data.

7.3 Optimized Prologue/Epilogue Variants

Optimizers change prologues/epilogues to reduce instruction count and memory traffic while preserving the ABI contract.

7.3.1 Using `leave`

`leave` is a compact epilogue for framed functions.

```
func_leave:
    push ebp
    mov  ebp, esp
    sub  esp, 16
    # body...
    leave                   # mov esp, ebp ; pop ebp
    ret
```

7.3.2 Epilogue With Direct Stack Restoration

If locals are a fixed size, compilers may restore `ESP` using an add:

```
func_add_epilogue:
    push ebp
    mov  ebp, esp
    sub  esp, 32
    # body...
    add  esp, 32
    pop  ebp
    ret
```

7.3.3 Saving Multiple Registers Efficiently

A function that uses multiple callee-saved registers may push them early and pop them late. The exact order is not ABI-visible; the requirement is full restoration before return.

```
func_multi_save:
    push ebp
    mov  ebp, esp
    push ebx
    push esi
    push edi
    sub  esp, 24

    # body...

    add  esp, 24
    pop  edi
    pop  esi
    pop  ebx
    pop  ebp
    ret
```

7.3.4 Tail Calls and Prologue Elision

A tail call can eliminate the current function’s epilogue/return by jumping directly to the target after restoring the caller-visible state.

```
# Tail-call shape: restore state, then jmp to target
func_tail:
    push ebp
    mov  ebp, esp
    # ... prepare args for target ...
    mov  esp, ebp
    pop  ebp
    jmp  target
```

This is ABI-correct only if the outgoing stack state matches what the target expects.

7.4 Debug vs Release Code Differences

Debug and release builds differ mainly in goals:

- **Debug:** stable frames, easy symbolic debugging, predictable variable locations.
- **Release:** speed and size, aggressive register allocation, fewer memory accesses, fewer instructions.

7.4.1 Typical Debug Characteristics

- `EBP` is commonly used as a frame pointer.
- Locals are stored on the stack more often (even if they could stay in registers).
- Prologue/epilogue is more canonical and consistent.

7.4.2 Typical Release Characteristics

- Frame pointer omission is common.
- Locals may live in registers (no stack slot exists).
- Prologues/epilogues may be merged, shortened, or removed for leaf functions.
- Tail-call optimizations may replace `call + ret` with `jmp`.

7.4.3 Extensive Example: Leaf Function (Release-Style Minimalism)

A leaf function makes no calls and uses no stack locals. It can omit a frame entirely.

```
# int id(int x) -> return x
# Release-style: no frame, no locals
id:
    mov  eax, dword ptr [esp+4]     # x is above return address
    ret
```

This is ABI-correct: the function reads its argument directly from the entry stack layout.

7.4.4 Extensive Example: Same Function With Debug-Friendly Frame

```
# Debug-style: explicit frame
id_dbg:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    pop  ebp
    ret
```

Both are correct; they reflect different priorities.

7.5 Stack Unwinding Implications

Stack unwinding is the process of walking back through call frames to recover return addresses and (sometimes) saved registers and local layout information. Unwinding is critical for:

- crash reporting and backtraces,
- debugging,
- exception handling mechanisms that must restore control and state.

7.5.1 EBP-Framed Code Is Easy to Unwind

When a function uses `EBP` as a frame pointer, the frame chain is explicit:

- `[ebp]` typically holds the previous frame pointer,
- `[ebp+4]` holds the return address.

```
# Typical framed layout:
# [ebp]   = old ebp
# [ebp+4] = return address
```

A debugger can follow the chain reliably, even without deep analysis of instruction streams.

7.5.2 FPO Requires Metadata or Heuristics

With frame pointer omission, there is no guaranteed linked list of frames. Unwinding then depends on:

- toolchain-provided unwind metadata (when present),
- conservative scanning and heuristics (less reliable),
- conventions used by the specific build configuration.

7.5.3 Extensive Example: Why Stack Imbalance Breaks Unwinding Immediately

Unwinding assumes the return address is at the correct position relative to the current stack state. A cleanup mismatch shifts `ESP`, so:

- the unwinder reads the wrong return address,
- the backtrace becomes incorrect or stops,
- debugging becomes misleading.

```
# If arguments were not cleaned correctly, ESP is wrong.
# Then [esp] at a 'ret' boundary is not the true return address.
# Unwinding and backtraces fail or become garbage.
```

7.5.4 Practical Engineering Rule

If your goal is **interoperability and diagnosability**, prefer:

- ABI-correct stack cleanup,
- correct callee-saved register restoration,
- predictable frame setup in low-level boundary functions (especially when mixing C/C++ with assembly).

Optimizations are valuable, but at ABI boundaries (public entry points, cross-language hooks, plugin interfaces), stable frames and strict ABI discipline reduce time-to-debug and prevent silent corruption.

Chapter 8

Interfacing with C and C++

8.1 Name Mangling in C vs C++

Name mangling is how a compiler encodes additional information into symbol names so the linker can distinguish entities that would otherwise collide. The key interoperability fact:

- **C:** external function names are typically exported as the plain identifier (subject to platform decoration rules).
- **C++:** function names are mangled to encode **overloads**, **namespaces**, **classes**, and sometimes calling convention and parameter types.

Therefore, a C++ function cannot be called from assembly (or C) by writing its source-level name unless you know the exact mangled symbol spelling produced by that toolchain and build mode.

8.1.1 Extensive Example: Why Overloads Force Mangling

These are legal in C++:

```
int f(int);
int f(double);
```

Both must exist as distinct linker symbols. C++ achieves that by mangling. C cannot do this at the ABI level, so C has no function overloading.

8.1.2 Practical Rule

If you need a stable symbol name across compilers and languages, do **not** export raw C++ functions directly. Export a C ABI boundary.

8.2 `extern "C"` and ABI Stability

`extern "C"` tells the C++ compiler to use **C linkage** for the declared function(s):

- no C++ name mangling,
- C-style linkage identity at the linker boundary,
- a stable entry point that assembly and C can call by a predictable symbol name.

8.2.1 Extensive Example: C ABI Wrapper Around C++ Implementation

Expose a stable C entry point, implement internally in C++:

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

int __cdecl sum2_c(int a, int b);
#ifdef __cplusplus
}
#endif
```

```
static int sum2_impl(int a, int b) noexcept {
    return a + b;
}

extern "C" int __cdecl sum2_c(int a, int b) {
    return sum2_impl(a, b);
}
```

Why this is stable: the exported symbol is C-linked and has an explicit calling convention. The internal C++ function can evolve (namespaces, overloads, templates) without changing the ABI boundary.

8.2.2 Do Not Assume C++ ABI Stability

Across different C++ compilers (and sometimes different versions or flags), the mangling scheme and certain ABI details can differ. `extern "C"` is the practical mechanism to define a stable, toolchain-agnostic boundary.

8.3 Matching Calling Conventions Explicitly

A correct link is not enough. The caller and callee must also agree on:

- argument placement (stack vs registers),
- stack cleanup responsibility (caller vs callee),
- register preservation rules at the boundary.

Therefore, every exported cross-language boundary should specify the calling convention explicitly.

8.3.1 Extensive Example: Declaring the Convention in C/C++

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

int __cdecl    add_cdecl(int a, int b);
int __stdcall add_stdcall(int a, int b);
int __fastcall add_fastcall(int a, int b);

#ifdef __cplusplus
}
#endif
```

8.3.2 Extensive Example: Assembly Implementations Must Match

cdecl implementation (caller cleanup, `ret`):

```
# int __cdecl add_cdecl(int a, int b)
add_cdecl:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    pop  ebp
    ret
```

stdcall implementation (callee cleanup, `ret 8`):

```
# int __stdcall add_stdcall(int a, int b)
add_stdcall:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    pop  ebp
    ret  8
```

fastcall implementation (common form: first two args in `ECX`, `EDX`):

```
# int __fastcall add_fastcall(int a, int b)
# a in ecx, b in edx (common fastcall lowering)
add_fastcall:
    push ebp
    mov  ebp, esp
    mov  eax, ecx
    add  eax, edx
    pop  ebp
    ret
```

8.3.3 Failure Mode: Mismatched Cleanup

If a caller treats a callee as cdecl and performs `add esp, 8` while the callee returns with `ret 8`, the stack becomes corrupted. If neither side cleans, the stack drifts. This is the most common interoperability bug in mixed C/C++/assembly IA-32 systems.

8.4 Data Layout Compatibility

Even if symbol naming and calling convention match, interoperability fails if both sides disagree on the **memory layout** of data types.

8.4.1 Stable Assumptions in IA-32

- **Pointer size:** 4 bytes.
- **int:** typically 4 bytes in mainstream IA-32 ABIs.
- **Alignment:** each type has an alignment requirement; structs gain padding to satisfy field alignment and overall alignment.

The critical engineering rule: **both sides must use the same definitions and the same packing/alignment policy**.

8.4.2 Extensive Example: Matching a Shared Struct Definition

Use one shared header for C and C++:

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

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

int __cdecl sum_point(Point p);
#ifdef __cplusplus
}
#endif
```

Assembly callee assumes `Point` is 8 bytes (two 32-bit fields) passed by value on the stack:

```
# int __cdecl sum_point(Point p)
# p.x at [ebp+8], p.y at [ebp+12]
sum_point:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    pop  ebp
    ret
```

If one side changes the struct (adds a field, changes packing), the offsets change and the function silently computes wrong results.

8.5 Struct Packing and Alignment Issues

Packing controls whether a compiler is allowed to insert padding bytes between fields for alignment. Changing packing changes field offsets, struct size, and alignment. ABI interoperability requires that both sides agree exactly.

8.5.1 The Default Reality: Padding Exists

Compilers insert padding so that:

- each field meets its alignment requirement,

- the overall struct size is a multiple of the struct's alignment (so arrays of the struct remain aligned).

8.5.2 Extensive Example: Padding Changes Offsets

Consider:

```
typedef struct S {
    char  a;    // 1 byte
    int   b;    // 4 bytes, typically aligned
    char  c;    // 1 byte
} S;
```

In default layout, the compiler typically inserts padding:

- padding after `a` so `b` begins at a 4-byte boundary,
- padding at the end so `sizeof(S)` is a multiple of the struct alignment.

Assembly that assumes `b` is immediately after `a` (offset 1) is wrong under default alignment rules and will read garbage.

8.5.3 Extensive Example: ABI-Safe Approach for Packed Data

If a binary format or network protocol requires packed layout, the ABI-safe approach is:

- use an explicitly packed definition in the shared header,
- treat packed fields carefully (potential unaligned accesses),
- prefer copying to aligned temporaries before heavy arithmetic if required by platform rules.

```
#pragma pack(push, 1)
typedef struct PackedS {
    char  a;
    int   b;
    char  c;
} PackedS;
#pragma pack(pop)
```

Then assembly offsets match the packed layout, but you must accept that `b` may be unaligned in memory.

8.5.4 Alignment Mismatch Is a Silent Interoperability Bug

Two modules can compile and link successfully but interpret the same bytes differently if:

- packing pragmas differ,
- compiler options change default alignment,
- types differ (`long` size differences across environments),
- one side uses C++ features that change layout (inheritance, virtual functions, non-standard-layout classes).

8.5.5 Practical ABI Boundary Rule

For cross-language and assembly boundaries in IA-32:

1. Prefer **C-compatible** structs with simple fields and no inheritance.
2. Keep the boundary types **standard-layout** and **trivially copyable** in C++ terms.
3. Use a **single shared header** to define struct layouts.

4. Avoid exporting C++ classes across the boundary; export C functions that operate on opaque pointers instead.
5. If packing is required, enforce it explicitly and document it as part of the ABI contract.

If you treat symbol naming, calling convention, and data layout as one unified contract, IA-32 interoperability becomes predictable and mechanically verifiable.

Chapter 9

Writing IA-32 Assembly for C/C++ Programs

9.1 Writing ABI-Correct Assembly Functions

ABI-correctness means your assembly function behaves exactly as the calling convention promises, regardless of compiler, optimization level, or who the caller is. For IA-32 stack-based interoperability, verify five invariants:

1. **Correct argument access:** read arguments from ABI-defined locations.
2. **Correct return placement:** return value in ABI-mandated register(s).
3. **Correct stack cleanup:** caller vs callee cleanup matches the declared convention.
4. **Correct register preservation:** restore all callee-saved registers you modify.
5. **Correct stack alignment at call boundaries:** preserve required alignment when you call other functions.

9.1.1 Extensive Example: Minimal ABI-Correct `cdecl` Function

C declaration (stable boundary):

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

int __cdecl add2(int a, int b);

#ifdef __cplusplus
}
#endif
```

Assembly implementation: reads args at `[ebp+8]` and `[ebp+12]`, returns in `EAX`, uses `ret` (caller cleanup).

```
# int __cdecl add2(int a, int b)
add2:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # a
    add  eax, dword ptr [ebp+12]     # + b

    pop  ebp
    ret
```

9.1.2 Extensive Example: ABI-Correct Callee-Saved Register Use

If you use `EBX`, preserve it.

```
# int __cdecl add_with_bias(int a, int b, int bias)
# Uses EBX (callee-saved) so must preserve it
add_with_bias:
    push ebp
    mov  ebp, esp
    push ebx

    mov  eax, dword ptr [ebp+8]     # a
    add  eax, dword ptr [ebp+12]    # + b
    mov  ebx, dword ptr [ebp+16]    # bias
    add  eax, ebx

    pop  ebx
    pop  ebp
    ret
```

9.1.3 Extensive Example: Correct `stdcall` Cleanup

C declaration:

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

int __stdcall add2_std(int a, int b);

#ifdef __cplusplus
}
#endif
```

Assembly ends with `ret 8` (callee cleanup for two 4-byte args):

```
# int __stdcall add2_std(int a, int b)
add2_std:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]

    pop  ebp
    ret  8
```

9.2 Exporting Symbols for the Linker

To link assembly with C/C++ object files, your assembly must export symbols using the assembler directives and naming required by the platform/toolchain. The main interoperability rule is:

- The symbol name in assembly must match the **linker-visible** name produced/expected by the C/C++ compiler for that declaration.

9.2.1 Using a Global Symbol

A common directive shape is to mark the function as global:

```
# Export the symbol so the linker can resolve it from C/C++ code
.global add2
add2:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    pop  ebp
    ret
```

If the linker reports an unresolved external, you must verify the exact spelling (including any platform decoration conventions) and ensure the symbol is exported exactly as expected.

9.2.2 Extensive Example: Export One Symbol, Hide Helpers

Expose a single ABI boundary, keep helpers local to the assembly file:

```
.global sum4
sum4:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]
    add  eax, dword ptr [ebp+12]
    add  eax, dword ptr [ebp+16]
    add  eax, dword ptr [ebp+20]

    pop  ebp
    ret

# Local helper (not exported) - label without .global
local_helper:
    ret
```

9.3 Using Assembly in Mixed Translation Units

A robust mixed-language workflow uses three files:

- a shared header that declares the ABI boundary,
- a C/C++ source file that calls it,
- an assembly source file that implements it.

9.3.1 Extensive Example: Minimal Mixed Unit Layout

Header (C and C++ compatible):

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

int __cdecl mul_add(int a, int b, int c);

#ifdef __cplusplus
}
#endif
```

C++ caller:

```
#include <cstdio>

extern "C" int __cdecl mul_add(int a, int b, int c);

int main() {
    int r = mul_add(2, 3, 4);   // 2*3 + 4 = 10
    std::printf("%d\n", r);
    return 0;
}
```

Assembly implementation:

```
.global mul_add
# int __cdecl mul_add(int a, int b, int c) => (a*b)+c
mul_add:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]      # a
    imul eax, dword ptr [ebp+12]     # eax = a*b
    add  eax, dword ptr [ebp+16]     # + c

    pop  ebp
    ret
```

This pattern ensures:

- stable symbol naming (C linkage),
- explicit calling convention,
- ABI-correct argument offsets and return register.

9.4 Common Interoperability Errors

Interoperability failures in IA-32 mixed code are usually mechanical and repeatable. The highest-impact errors:

9.4.1 Error 1: Wrong Calling Convention

Symptoms: stack imbalance, crash at `ret`, crash later in unrelated code, corrupted locals.

```
# Example mismatch:
# C++ declares __cdecl, but assembly returns with ret 8 (stdcall
↪  cleanup)
# => caller also adds esp, 8 => double cleanup => stack corruption
```

9.4.2 Error 2: Wrong Argument Offsets

Reading `[ebp+4]` as the first argument is wrong; `[ebp+4]` is the return address.

```
# Correct offsets in framed function:
# [ebp+8]  = arg1
# [ebp+12] = arg2
# [ebp+4]  = return address
```

9.4.3 Error 3: Failing to Preserve Callee-Saved Registers

Symptoms: silent corruption after returning, incorrect data in unrelated code paths.

```
# If you modify EBX/ESI/EDI/EBP (callee-saved), you must restore
↪  them.
```

9.4.4 Error 4: Data Layout Mismatch

Symptoms: wrong field values, wrong sizes, memory corruption when structs are passed by value or accessed by pointer.

```
# If C/C++ struct packing differs between modules, assembly field
↪  offsets are wrong.
```

9.4.5 Error 5: Hidden Arguments Not Accounted For

Symptoms: all argument offsets appear shifted; returned structs are garbage; crashes in constructors/destructors when crossing boundaries.

```
# Common hidden argument: sret pointer for large struct return.
# If present, it becomes the first argument at [ebp+8],
# shifting user-declared args to [ebp+12], [ebp+16], ...
```

9.5 Debugging ABI Mismatches

When mixed C/C++ and assembly fails, do not guess. Prove the call boundary mechanically.

9.5.1 Step 1: Verify the Convention by Evidence

- **Caller cleanup evidence:** caller performs `add esp, N` after the call.
- **Callee cleanup evidence:** callee ends with `ret N`.
- **Register-arg evidence:** caller loads `ECX`/`EDX` before the call (fastcall-like).

9.5.2 Step 2: Validate Stack Balance

A correct call sequence restores `ESP` to the exact expected value after cleanup. A simple invariant:

```
# If caller pushed N bytes of args in total:
# - cdecl-style: caller must add esp, N after return
# - stdcall-style: callee must ret N and caller must not add esp, N
```

9.5.3 Step 3: Confirm Register Preservation

If you suspect corruption, temporarily add explicit save/restore around calls in the caller, and save/restore in the callee for non-volatile registers. If the bug disappears, you likely violated the register contract.

```
# Caller-side defensive test: preserve volatile register across call
push ecx
call some_func
pop  ecx
```

9.5.4 Step 4: Confirm Argument Offsets With a Known Pattern

Use a test function that returns one of its arguments directly. If returning the "first" argument yields the wrong value, your offsets or hidden arguments are wrong.

```
# int __cdecl echo1(int a, int b) -> returns a
echo1:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]
    pop  ebp
    ret
```

9.5.5 Step 5: Isolate ABI Boundaries

For stable engineering:

- keep public entry points small and canonical (EBP frame, explicit saves),
- avoid exporting C++ classes directly; export C functions,
- add tiny adapter wrappers if you must bridge conventions.

9.5.6 Extensive Example: Convention Bridge Wrapper

If you must call a stdcall callee from a cdecl boundary, wrap it once, and keep the rest consistent.

```
# extern "C" int __cdecl call_std_from_cdecl(int a, int b);
# Internally calls: int __stdcall add2_std(int a, int b);

.global call_std_from_cdecl
call_std_from_cdecl:
    push ebp
    mov  ebp, esp

    push dword ptr [ebp+12]      # b
    push dword ptr [ebp+8]       # a
    call add2_std                # stdcall callee cleans its args (ret
     ↪  8)
    # no add esp, 8 here

    pop  ebp
    ret                          # cdecl boundary: caller cleans
     ↪  wrapper args
```

This wrapper ensures one clear ABI boundary for callers while safely invoking a differently-conventioned callee.

Chapter 10

Real-World ABI Pitfalls

10.1 Stack Corruption Patterns

In IA-32 interoperability, **stack corruption** is the highest-impact failure class because it destroys the control-flow contract (`call`/`ret`) and invalidates argument/local addressing. The most common patterns are mechanical and repeatable.

10.1.1 Pattern 1: Cleanup Mismatch (Double or Missing Cleanup)

Double cleanup happens when both caller and callee discard arguments. **Missing cleanup** happens when neither does. Both violate the invariant that the caller's `ESP` must be restored to the expected value after the call boundary is completed.

```
# Double cleanup example:
# Caller assumes cdecl and discards args, but callee is stdcall and
 ↪  also discards.
push 2
push 1
```

```
call add_stdcall           # callee ends with: ret 8
add  esp, 8                # WRONG: caller discards again -> stack
 ↪  jumps too far
```

```
# Missing cleanup example:
# Caller assumes stdcall (no add esp, N), but callee is cdecl (ret).
push 2
push 1
call add_cdecl             # callee ends with: ret
# WRONG: no cleanup performed -> stack grows with each call
```

10.1.2 Pattern 2: Wrong Byte Count

Even if the correct side cleans, using the wrong byte count is enough to destabilize the stack gradually.

```
# Wrong byte count: 3 args were pushed (12 bytes), but only 8 bytes
 ↪  removed
add  esp, 8                # WRONG: leaves 4 bytes (one arg) on stack
```

10.1.3 Pattern 3: Using `ESP`-Relative Offsets After Pushes

If you address arguments using `[esp+K]` and then push additional values, `ESP` changes and your offsets become wrong unless you compensate.

```
# Buggy pattern:
# On entry: [esp+4] is arg1, [esp+8] is arg2
buggy:
    push ebx                     # ESP changed, offsets moved
    mov  eax, dword ptr [esp+4]  # WRONG now: this is not arg1
     ↪  anymore
    pop  ebx
    ret
```

Safe pattern: establish `EBP` or compute stable addresses before modifying `ESP`.

```
# Safe pattern:
safe:
    push ebp
    mov  ebp, esp
    push ebx
    mov  eax, dword ptr [ebp+8]    # stable arg1
    pop  ebx
    pop  ebp
    ret
```

10.1.4 Pattern 4: Overwriting the Return Address

A classic catastrophic bug occurs when storing to `[ebp+4]` (return address) or `[esp]` near return.

```
# DO NOT do this:
# [ebp+4] is the return address in a framed function
mov dword ptr [ebp+4], eax    # corruption: next ret jumps to garbage
```

10.2 Incorrect Register Preservation

Register preservation is part of the ABI contract. Violating it often produces **silent and delayed** failures.

10.2.1 Callee-Saved Registers Must Be Restored

A common IA-32 rule set treats `EBX`, `ESI`, `EDI`, `EBP` as callee-saved. If your function modifies any of them, it must restore them before returning.

10.2.2 Extensive Example: Corruption by Failing to Restore `EBX`

Buggy callee:

```
# BUG: modifies EBX but does not restore it
bad_use_ebx:
    push ebp
    mov  ebp, esp

    mov  ebx, 999            # EBX is callee-saved in many IA-32 ABIs
    mov  eax, dword ptr [ebp+8]
    add  eax, ebx

    pop  ebp
    ret
```

Correct callee:

```
good_use_ebx:
    push ebp
    mov  ebp, esp
    push ebx

    mov  ebx, 999
    mov  eax, dword ptr [ebp+8]
    add  eax, ebx
    pop   ebx
    pop   ebp
    ret
```

10.2.3 Caller-Saved Registers Are Not Guaranteed

If the caller needs values in volatile registers (`EAX`, `ECX`, `EDX`) after a call, the caller must save them before calling.

```
# Caller must preserve ECX if needed after call
push ecx
call some_func
pop  ecx
```

10.3 Misaligned Stack Access

IA-32 can often tolerate unaligned memory accesses, but **ABI and toolchain assumptions** about stack alignment can still matter, especially at call boundaries and when code uses instructions that expect stronger alignment.

10.3.1 Alignment Failures Are Often Non-Local

A stack misalignment introduced in one function can crash or corrupt data in a deeper callee that assumes alignment. This is why alignment issues often appear “random”.

10.3.2 Extensive Example: Misalignment by Odd Number of Pushes

If the environment expects a particular alignment at calls, pushing an odd number of 4-byte values can flip alignment. Then calling another function without compensating violates the call-boundary invariant.

```
# Conceptual pitfall: push one extra value, then call without
↪  compensation
misalign_then_call:
    push ebp
    mov  ebp, esp

    push ebx                 # extra push changes ESP
    # if alignment mattered at calls, it may now be wrong
    call callee              # may fail if callee assumes alignment

    pop  ebx
    pop  ebp
    ret
```

Safe discipline: keep stack adjustments balanced and maintain required alignment before each call, especially in ABI boundary functions and assembly stubs.

10.4 Incorrect Struct Handling

Struct passing and returning is where many interoperability systems fail because:

- layout depends on padding and alignment,
- calling convention may introduce hidden arguments (sret),
- the ABI may treat "small" and "large" aggregates differently.

10.4.1 Pitfall 1: Wrong Field Offsets Due to Padding

If the C/C++ struct has padding, assuming a tightly packed layout in assembly reads wrong fields.

```
typedef struct S {
    char a;
    int  b;
} S;
```

In typical default layout, `b` is not at offset 1. Assembly that does `mov eax, [p+1]` for `b` is wrong. The correct offset is determined by the ABI's alignment rules for `int` within structs.

10.4.2 Pitfall 2: Passing by Value vs Passing by Pointer

If one side declares a function taking a struct by value and the other side implements it expecting a pointer, the call boundary is incompatible.

```
typedef struct Point { int x; int y; } Point;
extern "C" int __cdecl sum_point(Point p);    // by value
```

```
# Correct by-value callee reads fields from stack argument block
sum_point:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]      # p.x
    add  eax, dword ptr [ebp+12]     # + p.y
    pop  ebp
    ret
```

If the callee mistakenly treats `[ebp+8]` as a pointer to `Point`, it will load from an arbitrary address and likely fault or corrupt data.

10.4.3 Pitfall 3: Missing Hidden sret Pointer for Large Struct Return

Large struct returns commonly use a hidden first argument: a pointer to return storage. If the assembly implementation ignores it, every argument offset is shifted and the caller reads garbage.

```
# If sret is used:
# [ebp+8]  = hidden out pointer
# [ebp+12] = first user argument
# [ebp+16] = second user argument
```

This pitfall is especially common when developers read a C++ signature and assume the first argument begins at `[ebp+8]` without considering ABI lowering.

10.5 Silent Data Corruption Cases

The most dangerous ABI failures are those that do not crash immediately. They produce plausible but wrong results, pollute state, and explode later.

10.5.1 Case 1: Wrong Cleanup Produces "Delayed Return Address" Failure

A small stack imbalance may still allow the current function to return, but it mispositions the stack for future returns, causing a crash later in unrelated code.

```
# Example: wrong add esp, N leaves extra bytes on stack
# The current return may still work, but future frames are shifted.
```

10.5.2 Case 2: Callee-Saved Register Corruption

Failing to restore `ESI`/`EDI`/`EBX` can silently break:

- pointer iteration in loops,
- string/memory operations that use index registers,
- code that caches important pointers in non-volatile registers across calls.

10.5.3 Case 3: Struct Layout Mismatch With "Plausible Values"

If offsets are wrong but still point into valid memory, you get plausible yet incorrect field values. The program continues with corrupted state.

10.5.4 Case 4: Pointer/Reference Confusion

Treating a by-value struct as a pointer (or vice versa) can sometimes read valid memory by accident, producing wrong computations rather than immediate faults.

10.5.5 Extensive Example: A Classic "Looks Fine" Corruption

Caller expects cdecl cleanup, but callee uses stdcall cleanup. The call returns, but the caller then reads a local variable using an `EBP`-based offset that is now wrong due to stack drift in surrounding code. The value is still "valid", but not the intended one.

```
# The danger: wrong ABI often corrupts the stack in ways that
# still yield in-range memory, causing silent logical failures.
```

10.5.6 Practical Prevention Checklist

For IA-32 C/C++ interoperability, prevent pitfalls by enforcing:

1. **One stable boundary:** export `extern "C"` functions with explicit calling convention.
2. **Mechanical verification:** verify cleanup using `ret imm16` vs `add esp, N`.
3. **Preserve non-volatile registers:** save/restore every callee-saved register you modify.
4. **Use shared headers for structs:** do not duplicate struct definitions across modules.
5. **Assume hidden ABI lowering exists:** especially for struct returns and C++ member functions.

Treat ABI rules as non-negotiable engineering constraints. If you do, these pitfalls become rare, diagnosable, and preventable.

Chapter 11

Practical Case Studies

11.1 Calling C from Assembly

Calling a C function from IA-32 assembly is the most fundamental interoperability case. The assembly code must respect the C ABI: argument order, cleanup responsibility, register preservation, and return placement.

11.1.1 Case Study: Calling a `cdecl` C Function

C declaration (stable ABI boundary):

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

int __cdecl add(int a, int b);

#ifdef __cplusplus
}
#endif
```

Assembly caller (right-to-left arguments, caller cleanup):

```
# int r = add(10, 20);
call_add_from_asm:
    push 20                 # b
    push 10                 # a
    call add
    add  esp, 8             # cdecl: caller cleans args
    # result now in eax
    ret
```

Key verification points:

- arguments pushed right-to-left,
- `ESP` restored by the caller,
- return value read from `EAX`.

11.2 Calling Assembly from C++

Calling assembly from C++ is symmetrical, but the C++ compiler must be told the exact ABI contract. This is done using `extern "C"` and an explicit calling convention.

11.2.1 Case Study: C++ Calls Assembly (`cdecl`)

Header shared by C++ and assembly:

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

int __cdecl mul_add(int a, int b, int c);

#ifdef __cplusplus
}
#endif
```

Assembly implementation:

```
.global mul_add
# int __cdecl mul_add(int a, int b, int c) => (a * b) + c
mul_add:
    push ebp
    mov  ebp, esp

    mov  eax, dword ptr [ebp+8]     # a
    imul eax, dword ptr [ebp+12]    # eax = a * b
    add  eax, dword ptr [ebp+16]    # + c

    pop  ebp
    ret
```

C++ caller:

```
int main() {
    int r = mul_add(2, 3, 4);  // expected: 10
    return r;
}
```

Result: ABI correctness guarantees that the compiler-generated call sequence matches the assembly implementation exactly.

11.3 Mixed Calling Conventions Example

Real systems often require bridging different conventions. The safest approach is to introduce a small adapter function that isolates the mismatch.

11.3.1 Case Study: `cdecl` Wrapper Calling a `stdcall` Function

Target callee (stdcall):

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

int __stdcall add_std(int a, int b);

#ifdef __cplusplus
}
#endif
```

Assembly wrapper exported as `cdecl`:

```
.global add_std_from_cdecl
# int __cdecl add_std_from_cdecl(int a, int b)
# Internally calls: int __stdcall add_std(int a, int b)
add_std_from_cdecl:
    push ebp
    mov  ebp, esp
    push dword ptr [ebp+12]     # b
    push dword ptr [ebp+8]      # a
    call add_std                # stdcall callee cleans args (ret 8)
    # no add esp, 8 here

    pop  ebp
    ret                         # cdecl boundary
```

Engineering benefit: callers only see a single convention, and the ABI complexity is confined to one place.

11.4 Struct Return Case Study

Struct returns are one of the most common sources of ABI confusion. This case study demonstrates a large-struct return using a hidden return-storage pointer.

11.4.1 Case Study: Returning a Struct by Value

C declaration:

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

#ifdef __cplusplus
extern "C" {
#endif
Pair __cdecl make_pair(int a, int b);

#ifdef __cplusplus
}
#endif
```

ABI lowering (conceptual):

- caller allocates storage for `Pair`,
- caller passes a hidden pointer as the first argument,
- callee writes fields into caller-provided storage.

Assembly implementation:

```
.global make_pair
# Pair __cdecl make_pair(int a, int b)
# ABI view:
# [ebp+8]  = out (hidden sret pointer)
# [ebp+12] = a
# [ebp+16] = b
make_pair:
    push ebp
    mov  ebp, esp

    mov  edx, dword ptr [ebp+8]      # out
    mov  eax, dword ptr [ebp+12]     # a
    mov  dword ptr [edx], eax        # out->x = a
    mov  eax, dword ptr [ebp+16]     # b
    mov  dword ptr [edx+4], eax      # out->y = b
    mov   eax, edx                          # commonly return out pointer
    pop   ebp
    ret
```

Critical insight: if the hidden pointer is ignored, every argument offset is wrong and the return value is invalid.

11.5 Debugging a Broken ABI Example

This case study demonstrates a real debugging workflow when ABI assumptions are wrong.

11.5.1 The Symptom

A program crashes during a later return, far from the original call site. Stack traces are corrupted.

11.5.2 The Bug

Caller assumes `cdecl`; callee is implemented as `stdcall`. Both clean the stack.

```
# Caller (wrong)
push 2
push 1
call add_std
add  esp, 8                 # WRONG: callee already cleaned
```

```
# Callee (stdcall)
add_std:
    push ebp
    mov  ebp, esp
    mov   eax, dword ptr [ebp+8]
    add   eax, dword ptr [ebp+12]
    pop   ebp
    ret   8
```

11.5.3 The Diagnosis

1. Inspect the callee epilogue: presence of `ret 8` indicates callee cleanup.
2. Inspect the caller: `add esp, 8` indicates caller cleanup.
3. Conclude: double cleanup is occurring.

11.5.4 The Fix

Choose one cleanup strategy and enforce it consistently. Either:

- remove `add esp, 8` from the caller, or
- change the callee to use `ret`.

11.5.5 Verification Pattern

After the fix, verify mechanically:

- `ESP` before pushing arguments,
- `ESP` after the full call sequence,
- values must match exactly.

11.5.6 Final Engineering Lesson

Practical ABI work succeeds when every boundary is treated as a contract that can be proven by inspection:

- argument order,
- cleanup responsibility,
- register preservation,
- return placement.

When these rules are followed mechanically, mixed IA-32 assembly and C/C++ systems become predictable, debuggable, and robust—even under optimization and across toolchains.

Appendices

Appendix A — ABI Rules at a Glance

Register Usage Summary

The following summary reflects the **most common IA-32 C/C++ ABI expectations** used by mainstream toolchains when using stack-based conventions (`cdecl`/`stdcall`) and conventional register volatility rules.

- **Return registers**
  - 32-bit integer / pointer return: `EAX`
  - 64-bit integer return: `EDX:EAX` (high:low)
  - floating-point return (classic IA-32): x87 `ST(0)` (toolchain may use SSE in some environments)
- **Common register volatility split**
  - Caller-saved (volatile): `EAX`, `ECX`, `EDX`
  - Callee-saved (non-volatile): `EBX`, `ESI`, `EDI`, `EBP`
  - Stack pointer: `ESP` must be restored to the ABI-expected value at the call boundary

Rule: if your function modifies a callee-saved register, it must save and restore it.

```
# Callee-saved register discipline example (uses EBX, must preserve
↪  it)
use_ebx_correctly:
    push ebp
    mov  ebp, esp
    push ebx

    mov  ebx, 123
    mov  eax, dword ptr [ebp+8]
    add  eax, ebx

    pop  ebx
    pop  ebp
    ret
```

Stack Layout Summary

In IA-32, the stack grows downward (toward lower addresses). A near `call` pushes the return address. For a classic framed function:

```
# After:
#   push ebp
#   mov  ebp, esp
#
# Layout:
#   [ebp+0]  = old ebp
#   [ebp+4]  = return address
#   [ebp+8]  = arg1
#   [ebp+12] = arg2
#   [ebp+16] = arg3
#   ...
#   [ebp-4]  = local1 (example)
#   [ebp-8]  = local2 (example)
```

Rule: never treat `[ebp+4]` as an argument; it is the return address.

Argument Passing Rules

Right-to-left pushing (stack arguments) is the dominant rule for stack-based conventions:

- last source argument is pushed first
- first source argument ends up closest to the return address

cdecl (caller cleanup)

```
# cdecl call with two 32-bit args:
push arg2
push arg1
call func
add  esp, 8              # caller discards args (2 * 4)
```

stdcall (callee cleanup)

```
# stdcall call with two 32-bit args:
push arg2
push arg1
call func
# no add esp, 8 here
# stdcall callee returns with:
ret   8                     # callee discards args
```

fastcall (common 2-reg form)

A common fastcall lowering passes the first two integer/pointer arguments in `ECX` and `EDX`, with remaining arguments on the stack. Cleanup rule is toolchain-defined; confirm by `ret imm16` vs caller `add esp, N`.

```
# fastcall-like shape (common):
mov  ecx, arg1
mov  edx, arg2
call func_fast
```

Variadic constraint

Variadic functions require **caller cleanup** because only the caller knows the total number of arguments passed.

```
# variadic-style call shape:
push argN
...
push arg2
push arg1
call variadic_func
add  esp, N                 # caller discards exactly what it pushed
```

Hidden arguments (sret)

Large struct returns commonly use a hidden first argument: a pointer to caller-allocated return storage. This shifts all visible arguments by one slot.

```
# If sret is used:
# [ebp+8]  = hidden out pointer (return storage)
# [ebp+12] = user arg1
# [ebp+16] = user arg2
```

Return Value Rules

Return locations are ABI-mandated observable behavior at the call boundary.

Integer and pointer returns

```
# int/pointer return in EAX
ret_int:
    push ebp
    mov  ebp, esp
    mov  eax, 42
    pop  ebp
    ret
```

64-bit integer returns

```
# 64-bit return in EDX:EAX (high:low)
ret_i64:
    push ebp
    mov  ebp, esp
    mov  eax, 0x89ABCDEF         # low
    mov  edx, 0x01234567         # high
    pop  ebp
    ret
```

Floating-point returns (classic IA-32)

Classic IA-32 ABIs commonly return floating-point values via x87 `ST(0)`. Toolchains may use SSE-based conventions in some environments, but the ABI boundary still defines a specific floating return location.

```
# Conceptual: floating return via x87 ST(0)
# ret_fp:
#     fld qword ptr [const]
#     ret
```

Small struct returns

Small aggregates may be returned in registers (commonly `EAX` and optionally `EDX`), depending on ABI/toolchain rules.

```
# Example: 8-byte aggregate returned as two 32-bit parts
# eax = first 32 bits, edx = second 32 bits
ret_pair_small:
    push ebp
    mov  ebp, esp
    mov  eax, dword ptr [ebp+8]      # x
    mov  edx, dword ptr [ebp+12]     # y
    pop  ebp
    ret
```

Large struct returns via hidden pointers (sret)

Large aggregates are commonly returned by writing into caller-provided storage (hidden pointer argument). The callee fills `*out` and returns.

```
# void make_big_sret(Big* out, int x)
# [ebp+8]  = out (hidden)
# [ebp+12] = x
make_big_sret:
    push ebp
    mov  ebp, esp
    mov  edx, dword ptr [ebp+8]      # out
    mov  eax, dword ptr [ebp+12]     # x
    mov  dword ptr [edx], eax        # out->a = x
    pop  ebp
    ret
```

Golden invariant: after a call boundary completes, the caller's expectations about ESP, preserved registers, and return locations must be satisfied exactly.

Appendix B — Common Mistakes and How to Detect Them

Symptoms of Stack Imbalance

Stack imbalance is the most frequent and destructive ABI error in IA-32 interoperability. It occurs when arguments are discarded incorrectly (double cleanup, missing cleanup, or wrong byte count).

Typical Observable Symptoms

- Crashes at a ret instruction far from the original call site.
- Returning to an invalid address (instruction pointer jumps into unmapped or data memory).
- Corrupted local variables in the caller after a function call.

- Backtraces that abruptly stop or contain nonsensical frames.
- Bugs that disappear when logging or debugging code is added.

Canonical Imbalance Patterns

```
# Double cleanup: caller and callee both discard arguments
call func_stdcall       # callee ends with: ret 8
add  esp, 8             # WRONG: stack advanced too far
```

```
# Missing cleanup: neither side discards arguments
call func_cdecl         # callee ends with: ret
# WRONG: esp not restored, stack grows
```

```
# Wrong byte count: pushed 12 bytes, removed only 8
add  esp, 8             # WRONG: leaves 4 bytes on stack
```

Key invariant: after a full call sequence completes, `ESP` must have the same value it had before arguments were pushed (except for intentional caller-local allocations).

Debugger-Level Detection

Stack-related ABI bugs are best diagnosed at the instruction level, not by source inspection alone.

Step 1: Watch `ESP` Across the Call

Record `ESP` at three points:

- before pushing arguments,
- immediately before the `call`,

- immediately after returning from the call and cleanup.

If the ABI is correct, the final `ESP` must match the initial value.

```
# Diagnostic pattern (conceptual):
# esp_before = esp
# push args...
# call func
# cleanup
# assert esp == esp_before
```

Step 2: Inspect the Callee Epilogue

The epilogue reveals cleanup responsibility:

```
ret             # caller cleanup (cdecl-style)
ret  8          # callee cleanup for 2 args (stdcall-style)
```

If the callee uses `ret imm16`, the caller must not adjust `ESP`.

Step 3: Validate Argument Offsets

In framed functions:

- `[ebp+4]` is the return address,
- `[ebp+8]` is the first argument,
- additional arguments follow at +4 byte increments.

```
# Correct framed access
mov eax, dword ptr [ebp+8]     # arg1
mov ecx, dword ptr [ebp+12]    # arg2
```

Reading arguments relative to `ESP` after pushes is a common source of subtle bugs.

Step 4: Check Callee-Saved Registers

If a function modifies `EBX`, `ESI`, `EDI`, or `EBP` without restoring them, corruption may appear later in unrelated code.

```
# Bug: EBX modified but not restored
mov ebx, 123
ret
```

Compiler Warning Patterns

Many ABI bugs first surface as compiler or linker warnings. Ignoring them is a frequent cause of late-stage failures.

Warning Pattern: Calling Convention Mismatch

Typical meaning:

- caller expects one cleanup rule,
- callee is declared with a different convention.

This almost always leads to stack imbalance at runtime.

Warning Pattern: Incompatible Function Declarations

Occurs when:

- the same symbol is declared differently in different translation units,
- one declaration uses `extern "C"` and another does not,
- parameter lists differ (by-value vs pointer, missing hidden arguments).

Warning Pattern: Size or Alignment Mismatch

Often indicates:

- inconsistent struct packing,
- different type sizes assumed across modules,
- ABI-incompatible data layout.

These warnings frequently correlate with silent data corruption rather than immediate crashes.

Runtime Crash Signatures

ABI bugs tend to produce characteristic crash patterns that can be recognized quickly with experience.

Signature 1: Crash at `ret`

The instruction pointer loads an invalid address from the stack. Common causes:

- stack imbalance,
- overwritten return address,
- incorrect epilogue.

Signature 2: Crash in a Later Function

The call that caused the corruption has already returned successfully. The crash occurs when:

- a later `ret` reads a shifted return address,
- a later function reads corrupted locals or arguments.

This is typical of small but repeated cleanup errors.

Signature 3: Invalid Pointer Dereference

Often caused by:

- misinterpreting a by-value struct as a pointer,
- missing a hidden sret pointer,
- reading arguments at the wrong offsets.

Signature 4: Corrupted Backtrace

Stack unwinding fails or produces nonsensical frames. This strongly suggests:

- stack imbalance,
- incorrect prologue/epilogue,
- overwritten frame pointers or return addresses.

Practical Detection Checklist

When diagnosing an IA-32 ABI issue, apply this checklist in order:

1. Confirm the calling convention on both sides.
2. Inspect the callee epilogue (`ret` vs `ret imm16`).
3. Verify stack balance by tracking `ESP`.
4. Check callee-saved register preservation.
5. Verify argument offsets and hidden arguments.
6. Confirm struct layout and alignment consistency.

Engineering principle: ABI bugs are deterministic. Once you reduce the problem to instruction-level facts—stack pointer movement, register state, and return locations—the root cause becomes mechanically provable rather than speculative.

References

IA-32 Architecture Manuals

The authoritative foundation for IA-32 behavior is the processor architecture documentation that defines:

- instruction semantics and side effects,
- register roles and special-purpose behavior,
- stack operation rules (`push`, `pop`, `call`, `ret`),
- calling and return mechanics at the ISA level (near calls, stack-based control transfer).

For ABI work, the most relevant architectural facts include:

- stack grows toward lower addresses,
- `call` pushes the return address,
- `ret` pops the return address (and optionally an immediate byte count),
- general-purpose register width and pairing (`EDX:EAX` for 64-bit returns),
- x87 floating-point stack behavior (`ST(0)` return semantics).

These manuals are the **ground truth** when reasoning about what instructions do. ABI rules are layered on top of these guarantees, not independent of them.

```
# Architectural invariants used by all IA-32 ABIs
# - call: pushes return address
# - ret: pops return address (optionally discards args)
# - stack grows downward
# - eax is the canonical integer return register
```

Compiler ABI Documentation

Compiler ABI documentation defines how high-level language constructs are lowered onto the IA-32 architecture. This includes:

- calling conventions (`cdecl`, `stdcall`, `fastcall`),
- register volatility rules (caller-saved vs callee-saved),
- stack layout conventions at function entry,
- struct passing and returning rules,
- hidden arguments (such as structure return pointers).

For ABI correctness, compiler documentation clarifies:

- which registers must be preserved by callees,
- how arguments are ordered and aligned,
- how return values are communicated to callers,
- which optimizations are allowed without violating the ABI.

A crucial engineering principle derived from compiler ABI documents:

> Internal code generation may change freely, but externally visible ABI behavior must remain stable.

This explains why:

- frame pointer omission does not change argument offsets at the call boundary,
- tail-call optimizations are only legal when ABI contracts are preserved,
- different optimization levels still interoperate at the binary boundary.

```
# ABI vs optimization reminder:
# - prologue/epilogue shape may change
# - register allocation may change
# - argument and return locations at the boundary must not change
```

Toolchain Calling Convention References

Toolchain-specific calling convention references describe:

- exact register usage for fastcall-style conventions,
- symbol naming and decoration rules,
- default calling conventions for C and C++,
- ABI differences across language modes and compilation flags.

These references are essential when:

- mixing compilers or assemblers,

- interfacing with system libraries,
- writing assembly intended to be portable across toolchains.

From these references, the following practical rules emerge:

- never guess a calling convention—declare it explicitly,
- verify cleanup behavior by inspecting `ret` vs `add esp, N`,
- confirm register-passed arguments by observing call-site setup,
- isolate convention differences using small adapter functions.

```
# Mechanical convention detection checklist:
# - Does callee use ret N? -> callee cleanup
# - Does caller add esp, N? -> caller cleanup
# - Are ecx/edx loaded before call? -> register-passing convention
```

Cross-References to Other Booklets in This Series

This booklet builds directly on concepts established earlier in the CPU Programming Series and prepares the ground for later volumes.

Foundational Dependencies

Readers are expected to be familiar with:

- instruction execution flow (fetch, decode, execute, retire),
- register and flag semantics,
- binary data representation and signed/unsigned behavior,

- basic stack mechanics.

These topics are covered in earlier booklets focused on:

- CPU execution models,
- registers, flags, and data representation,
- stack fundamentals and calling conventions at a conceptual level.

Direct Continuations

The material in this booklet directly feeds into later, architecture-specific and system-level topics, including:

- x86-64 calling conventions and ABI differences,
- SIMD and vector calling rules,
- exception handling and stack unwinding mechanisms,
- operating system ABIs and system call interfaces.

Series Integration Perspective

Within the CPU Programming Series, this booklet serves as:

- the **practical ABI bridge** between pure instruction semantics and real-world compiled programs,
- the point where assembly, C, and C++ interoperability becomes concrete and testable,
- the last architecture-neutral step before transitioning to 64-bit ABIs and OS-level interfaces.

By mastering the rules summarized and applied here, the reader gains the ability to:

- read compiler-generated IA-32 code with confidence,
- write ABI-correct assembly callable from high-level languages,
- diagnose and fix real-world binary interoperability failures.

This makes the booklet a structural hinge in the series: everything before it explains *how the CPU works*, and everything after it builds on *how real programs are composed from those rules*.