

Early Access  
Edition

# The Anatomy of Go

Phuong Le



Learn Go by understanding how the language works internally

Early Access  
Edition

# The Anatomy of Go

Phuong Le



Learn Go by understanding how the language works internally

 byteSizeGo

[OceanofPDF.com](http://OceanofPDF.com)

# Table of Contents

1. [Foreword](#)
2. [Preface](#)
  1. [Is This Book Right for You](#)
  2. [Content Guide](#)
  3. [Who Am I & Why This Book](#)
  4. [Acknowledgments](#)
  5. [Feedback](#)
3. [Chapter 1: Go Thinks Simple Works Better](#)
  1. [1. What Makes Go Special](#)
    1. [1.1 Good Code Needs More Than Code](#)
    2. [1.2 The Go Way of Building Software](#)
    3. [1.3 Go's Design Decisions](#)
      1. [Why Go Skips the Ternary Operator](#)
      2. [How Go Handles Initial Values](#)
      3. [When Simplicity Met Generics](#)
  2. [2. How Go Runs Your Code](#)
    1. [The Heart of Go: Compiler & Runtime](#)
    3. [Summary](#)
    4. [References](#)
4. [Chapter 2: Basic Data Types, Variables, and Constants](#)
  1. [1. Integer Types: Bit-Width, Semantics, and System Architecture](#)
    1. [Integer Literals and Number Formats](#)
    2. [Overflow and Underflow: Behavior and Pitfalls](#)
    3. [Explicit Conversion Between Integer Types](#)
    4. [Division by Zero: Compile-Time and Runtime Safeguards](#)
  2. [2. Floating-Point Types: Precision, Representation, and IEEE-754 Semantics](#)
    1. [2.1 Floating-Point Literals and Notation Styles](#)
    2. [2.2 Type Conversion and Precision Trade-Offs](#)
    3. [2.3 IEEE-754 Compliance and Internal Behavior](#)
      1. [Special Floating-Point Values: +Inf, -Inf, and NaN](#)
      2. [Rounding Errors, Approximation, and Precision Loss](#)
  3. [3. Boolean Type: 1 Byte Memory Representation](#)

4. [4. Pointer Types: Memory Addresses, Dereferencing, and Allocation](#)
  1. [4.1 Understanding Pointers](#)
  2. [4.2 Type Safety, Nil Pointers, and Dereference Panics](#)
5. [5. Variables & Constants: Declarations, Scoping, and Compiler Behavior](#)
  1. [5.1 Variable Declarations, Scope, and Initialization](#)
    1. [Variable Shadowing and Scope Resolution](#)
    2. [Compiler Enforcement of Unused Variables](#)
  2. [5.2 Constants: Fixed Values, Type Semantics, and Optimization](#)
    1. [Typed Constants and Compile-Time Evaluation](#)
    2. [Untyped Constants and Type Inference Rules](#)
    3. [iota: Implicit Sequencing and Declarative Patterns](#)
6. [6. Bonus: Unsafe Pointer](#)
  1. [6.1 Risks of Unsafe Pointers and uintptr-Based Manipulation](#)
  2. [6.2 Using unsafe Safely: Guidelines and Best Practices](#)
    1. [a. Conversion of a \\*T1 to Pointer to \\*T2](#)
    2. [b. Conversion of a Pointer to a uintptr \(but not back to a pointer\)](#)
    3. [c. Conversion of a Pointer to a uintptr and Back, with Arithmetic](#)
    4. [d. Conversion of a Pointer to a uintptr for System Calls](#)
    5. [e. Conversion of the Result of reflect.Value.Pointer or reflect.Value.UnsafeAddr from uintptr to Pointer](#)
7. [7. Bonus: Addressable and Unaddressable Values](#)
8. [8. Summary](#)
9. [References](#)
5. [Chapter 3: Arrays, Slices, Strings, and Maps](#)
  1. [1. Arrays: Fixed-Size Sequences and Memory Semantics](#)
    1. [1.1 Understanding Array Memory Layout](#)
    2. [1.2 Array Initialization](#)
      1. [Zeroing Small Memory Blocks \(0-79 Bytes\)](#)
      2. [Zeroing Medium Memory Blocks \(80-1039 Bytes\)](#)
      3. [Zeroing Large Memory Blocks \(1040 Bytes and Above\)](#)

- 3. [1.3 Working with Array Literals](#)
- 4. [1.4 Small vs. Large Array Initialization Strategies](#)
- 5. [1.5 Array Length and Capacity](#)
- 2. [2. Slices: Structure, Semantics, and Behavior](#)
  - 1. [2.1 Slice Expressions and Indexing Semantics](#)
  - 2. [2.2 Append: Mechanics of Slice Growth](#)
  - 3. [2.3 Allocation Strategies and Memory Layout](#)
- 3. [3. Strings: Structure and Behavior](#)
  - 1. [3.1 UTF-8 Encoding and Internal Representation](#)
  - 2. [3.2 String Conversion and Memory Semantics](#)
  - 3. [3.3 Efficient String Concatenation Techniques](#)
- 4. [4. Maps: Internal Design and Runtime Behavior](#)
  - 1. [4.1 Internal Structure and Memory Layout](#)
    - 1. [Buckets and Collision Management](#)
    - 2. [Top Hash Optimization](#)
  - 2. [4.2 Operational Semantics and Usage Patterns](#)
    - 1. [Map Initialization and Allocation](#)
    - 2. [Value Lookup and Retrieval](#)
    - 3. [Insertion and Updating of Entries](#)
    - 4. [Deletion and Slot Reclamation](#)
    - 5. [Growth Strategy and Rehashing](#)
    - 6. [For-Range Loop Over a Map](#)
    - 7. [Experiment: Map Modifications During Iteration](#)
- 5. [5. Summary](#)
  - 1. [Arrays](#)
  - 2. [Slices](#)
  - 3. [Strings](#)
  - 4. [Maps](#)
- 6. [References](#)
- 6. [Chapter 4: Structs, Interfaces and Generics](#)
  - 1. [1. Structs: Composition, Layout, and Performance](#)
    - 1. [1.1 Introduction to Structs](#)
      - 1. [Declaring Struct Types](#)
      - 2. [Initializing Structs Using Composite Literals](#)
      - 3. [Anonymous Structs and Their Use Cases](#)
      - 4. [Accessing Struct Fields](#)
    - 2. [1.2 Advanced Struct Concepts](#)

1. [Struct Comparability and Equality Semantics](#)
2. [Composition via Embedding and Field Promotion](#)
3. [Method Shadowing in Embedded Structs](#)
3. [1.3 Memory Layout and Performance Considerations](#)
  1. [Understanding Word Size](#)
  2. [Cache Line Alignment](#)
  3. [Alignment and Padding in Structs](#)
  4. [Working with Empty Structs](#)
  5. [Zero-Size Values on the Stack](#)
  6. [Zero-Size Values on the Heap](#)
2. [2. Interfaces: Contracts, Composition, and Runtime Mechanics](#)
  1. [2.1 Interfaces as Behavioral Contracts and Implicit Satisfaction](#)
  2. [2.2 Composing and Embedding Interfaces](#)
  3. [2.3 Internal Structure of Empty Interfaces](#)
  4. [2.4 Non-Empty Interfaces and Method Dispatch via itab](#)
    1. [Automatic Receiver Method Handling](#)
    2. [How Go Builds Itabs](#)
  5. [2.5 Type Assertions and Runtime Behavior](#)
3. [3. Generics: Type Parameters, Constraints, and Compiler Internals](#)
  1. [3.1 Motivation for Generics and Basic Syntax](#)
  2. [3.2 Defining and Using Type Constraints](#)
    1. [Constraints Are Not Interfaces](#)
  3. [3.3 Type Inference: From Arguments to Complex Relationships](#)
    1. [Constraint Type Inference](#)
    2. [Preserving Types in Generics](#)
  4. [3.4 Compiler Internals: How Go Implements Generics](#)
    1. [Code Generation via Stenciling and Dictionaries](#)
    2. [Understanding Type Shapes and Shape Inference](#)
    3. [Dictionary Passing and Function Dispatch](#)
  4. [Summary](#)
    1. [Structs](#)
    2. [Interfaces](#)
    3. [Generics](#)
  5. [References](#)
7. [Chapter 5: How Go Code Turns Into Assembly](#)

1. [5.1 Inspecting the Compilation Process](#)
  1. [Compiling the `foo` Package](#)
  2. [Compiling the `main` Package](#)
  3. [Linking the Final Binary](#)
2. [5.2 Overview of Compiling a Package](#)
3. [5.3 Stage 1: Parsing and Building the Syntax Tree](#)
  1. [Tokenization with the Scanner](#)
  2. [Constructing the Syntax Tree with the Parser](#)
4. [5.4 Stage 2: Type Checking and Scope Resolution](#)
  1. [File Validation and Package Consistency](#)
  2. [Gathering Package-Level Declarations](#)
  3. [Type Checking Top-Level Declarations](#)
  4. [Type Checking of Function Bodies](#)
  5. [Resolving Initialization Order for Package-Level Variables](#)
5. [5.5 Stage 3: IR Construction](#)
6. [Export Data & Relocations](#)
  1. [Generating Export Data](#)
    1. [Encoding Package Element](#)
    2. [Encoding Constant Object](#)
    3. [Encoding Function Body Relocations](#)
  2. [Decoding Export Data into IR Nodes](#)
7. [5.6 Stage 4: Optimization: Dead Code, Devirtualization, Inlining, Escape Analysis](#)
  1. [5.6.1 Early Dead Code Elimination](#)
  2. [5.6.2 Devirtualization](#)
  3. [5.6.3 Inlining](#)
    1. [Determine Inlinable Functions](#)
    2. [Inlining at the Call Site](#)
    3. [Fast Path Optimization](#)
    4. [Call Site Scoring](#)
  4. [5.6.4 Escape Analysis](#)
8. [5.7 Stage 5: Walk \(Middle end\)](#)
  1. [Order Phase](#)
    1. [For-Range Order](#)
  2. [Walk Phase](#)
    1. [For-Range & Loop-var Problem](#)
    2. [Other Statements](#)

9. [5.8 Stage 6: Static Single Assignment \(SSA\) Generation](#)
  1. [SSA Idea](#)
  2. [SSA Syntax](#)
    1. [Value & Operation](#)
    2. [Phi Function](#)
    3. [Memory](#)
    4. [Block and Function](#)
  3. [Building SSA From IR Nodes](#)
  4. [SSA Passes](#)
    1. [Optimization Phase](#)
    2. [Lower Passes](#)
    3. [Layout & Schedule Passes](#)
    4. [Register Allocation Passes](#)
10. [5.9 Stage 7: Machine Code Generation](#)
11. [Wrapping Up](#)
12. [References](#)
8. [Chapter 6: Functionality](#)
  1. [1. Preliminaries](#)
    1. [Prerequisite: MPG Model Overview](#)
    2. [Prerequisite: Stack Pointer, Program Counter, and Argument Pointer](#)
      1. [Stack Pointer \(SP\)](#)
      2. [Argument Pointer \(argp\)](#)
      3. [Program Counter \(PC\)](#)
  2. [2. Function Basics to Advanced](#)
    1. [Functions as First-Class Citizens](#)
      1. [Pass-by-Value Semantics](#)
      2. [Closures: Capturing and Lifetime](#)
    2. [Treating Functions as Data: Function Values](#)
    3. [How Closures Work](#)
    4. [Variadic Function Design](#)
    5. [The init Function](#)
  3. [3. Defer: Patterns and Performance](#)
    1. [The Role of Defer](#)
    2. [How Defer Is Implemented](#)
    3. [Heap-Allocated Defers](#)
      1. [Per-Processor Pooling \(Defer Pool\)](#)

- 2. [Executing Deferred Calls \(Defer Return\)](#)
- 4. [Stack-Allocated Defers](#)
- 5. [Open-Coded Defers](#)
- 4. [4. Panic & Recover](#)
  - 1. [Usage Scenarios](#)
  - 2. [What the Runtime Does on Panic](#)
  - 3. [Recovery and Argument Pointer](#)
- 5. [5. Profiling](#)
  - 1. [Profile Collection & Access Methods](#)
    - 1. [Profiling with go test](#)
    - 2. [Profiling via runtime/pprof API](#)
    - 3. [Profiling over HTTP \(net/http/pprof\)](#)
  - 2. [Using go tool pprof](#)
    - 1. [Interactive Shell](#)
    - 2. [HTTP UI Mode](#)
  - 3. [Reading Call Graph Visualizations](#)
    - 1. [Labels](#)
    - 2. [Reading Flame Graphs](#)
  - 4. [Memory \(Heap/Alloc\) Profiling](#)
    - 1. [Bucket](#)
    - 2. [Sampling & Scaling](#)
  - 5. [CPU Profiling](#)
    - 1. [Process-Wide CPU Profiling](#)
    - 2. [Thread-Wide CPU Profiling](#)
    - 3. [Interpreting CPU Profiles](#)
  - 6. [Mutex \(Lock Contention\) Profiling](#)
    - 1. [Understanding Contention Time](#)
    - 2. [Sampling Fraction](#)
    - 3. [Defer Interactions](#)
  - 7. [Block \(Wait\) Profiling](#)
    - 1. [Sampling Rate](#)
  - 8. [Goroutine Profiling](#)
  - 9. [Threadcreate Profiling](#)
  - 10. [Delta \(Diff\) Profiling](#)
- 6. [6. Trace: Timeline & Internals](#)
  - 1. [Turning On Tracing \(go test, HTTP, code\)](#)
  - 2. [Visualizing Traces](#)

1. [Proc-Oriented View \(per-P lanes & flows\)](#)
2. [Thread-Oriented View \(per-thread lanes\)](#)
3. [Goroutine Analysis Tools \(summary, breakdown, ranges\)](#)
4. [Profiles Generated From Traces \(scheduler, sync, syscall, net\)](#)
5. [Adding Semantics: Tasks, Regions, Logs](#)
6. [Under the Hood](#)
  1. [Buffering & Generations \(full/empty queues\)](#)
  2. [Trace Locker](#)
  3. [Trace Writer](#)
  4. [Event Pipeline](#)
7. [7. Profile-Guided Optimization \(PGO\)](#)
  1. [Using PGO in Practice](#)
  2. [PGO Mechanics](#)
    1. [Build Process \(what the toolchain does\)](#)
    2. [Preprocess: GO PREPROFILE V1](#)
    3. [Matching: Call Edges & Offsets](#)
    4. [Hot Callsite Policy & Inline Budget](#)
    5. [Devirtualization \(guided by profiles\)](#)
8. [Summary](#)
9. [References](#)

# Foreword

By Matt Boyle

Founder, bytesizego.com

When I first had the idea for bytesizego.com, I was driven by a simple goal: to create the kind of resources I wish I had when I was trying to level up from being a junior Go developer to a senior.

When you're first starting out, there's no shortage of tutorials and guides to help you get your feet wet. But once you've got the basics down and you're ready to dive deeper, it can feel like you're on your own.

That's where Phuong comes in. I first came across his work on Twitter, and I was immediately struck by the way he tackled complex topics with such clarity and depth. He has a gift for taking the most daunting concepts and breaking them down in a way that made them feel accessible and even - dare I say it - fun.

I knew right away that I wanted to work with Phuong. His passion for diving deep into the nuts and bolts of Go was exactly what I was looking for and when we started talking about the idea for this book, it was clear that we were on the same page.

Our goal with this book is simple: to create a resource for Go engineers who are ready to take their skills to the next level. Whether you're an expert looking to expand your knowledge or an intermediate learner eager to tackle more advanced topics, we've got you covered.

The book you are about to read is a testament to Phuong's incredible talent as a teacher and a programmer. He has a way of making even the most complex topics feel approachable, and his enthusiasm for the material is infectious. I've learned so much from working with him, and I know you will too.

So if you're ready to dive deep into Go, you've come to the right place. With Phuong as your guide, you'll gain a new appreciation for the power and beauty of this wonderful language.

Thank you for supporting this book - seriously. It takes a lot of effort to put a book together and most publishers have hundreds of employees. The one you're about to read has had everything done by just two people - Phuong in particular has been working on his mornings, evenings and weekends for over a year.

We can't wait to hear what you think.

[OceanofPDF.com](http://OceanofPDF.com)

# Preface

# Is This Book Right for You

This book is designed to get you up to speed with Go. It's aimed at readers who already have some familiarity with the language. If you're new to Go and haven't completed the Go Tour yet, some sections might feel overwhelming. That's not an issue—take the time to go through the Go Tour (<https://go.dev/tour/>), practice coding, and gain some hands-on experience. Once you've built a foundation, come back to this book. The concepts will start to make more sense, and the advanced topics will feel much more approachable.

Each section moves from basic ideas to more advanced concepts in a clear, logical flow. If you're already comfortable with Go and find the early chapters too familiar, feel free to skip ahead. Even so, the simpler sections might still offer insights or details you haven't encountered before.

## Content Guide

This book isn't here to teach you how to write perfect or idiomatic Go code, and it's not a guide for 'good code' practices. The focus is on making concepts clear and useful. Some examples are simplified for clarity, even if they don't follow best practices. For example:

```
f, _ := os.Open("filename.txt") // ignore error
```

In real-world code, handling errors is a must. But here, the focus is on the core idea, so some details are intentionally left out to keep things simple and easy to grasp. Explanations are kept straightforward—no complicated annotations or overly academic language. Take this definition from the **Go Specification** for floating-point literals:

Figure 1. Floating-point literals (The Go Programming Language Specification)

```
float_lit      = decimal_float_lit | hex_float_lit .
decimal_float_lit = decimal_digits "." [ decimal_digits ]
```

```

] [ decimal_exponent ] |
    decimal_digits decimal_exponent |
    "." decimal_digits [
decimal_exponent ] .
decimal_exponent = ( "e" | "E" ) [ "+" | "-" ]
decimal_digits .

hex_float_lit     = "0" ( "x" | "X" ) hex_mantissa
hex_exponent . .
hex_mantissa      = [ "_" ] hex_digits "." [ hex_digits
] |
                    [ "_" ] hex_digits |
                    "." hex_digits .
hex_exponent      = ( "p" | "P" ) [ "+" | "-" ]
decimal_digits .

```

This level of precision is necessary for formal documentation but not for this book. It might not always be technically perfect—for example, calling a **goroutine** a lightweight thread isn’t entirely accurate—but the comparison is clear enough to convey the idea. That’s what matters.

This book covers Go 1.23, focusing primarily on ARM64, with some examples in AMD64. Differences between the two architectures are noted where relevant. The code and tests should work the same across operating systems and architectures. While some differences exist under the hood, the core concepts remain unchanged. Future updates will track new Go versions.

In addition, edge cases aren’t the focus here either. While exceptions to the rules exist, they often distract from the main ideas. If an edge case is critical for understanding, it will be mentioned—otherwise, it’s left out. The goal remains simplicity and clarity.

## Who Am I & Why This Book

I’m Phuong Le, a writer who’s spent the last few years deeply involved with Go. I’m also an active member of the Gopher community. Right now, I work at VictoriaMetrics, where Go powers our core projects—VictoriaMetrics monitoring solution, our time-series database, and VictoriaLogs, a high-performance log management tool.

This book began with a conversation I had with Matt Boyle, an Engineering Manager at Cloudflare. He said, "*Hey Phuong, I saw your post about Go internals where you mentioned reading the Go source code. How about turning that into a book?*" That suggestion sparked the idea—and here we are.

My first attempt was a book called *100 Practical Tips for Writing Go*, inspired by my own journey with the language. But it didn't take long to realize that many of those tips weren't just mine—they came from the collective wisdom of the Go community. It felt more honest to share them openly. So, I created a GitHub guide with over 80 tips (<https://github.com/func25/go-practical-tips>), posted them one by one on X, and eventually turned them into a long-form guide for easier reading. It's now 5,378 lines long. That experience, combined with the community's feedback and Matt's encouragement, led me to write this book.

This time, the focus is different—understanding Go's internals from the ground up. Writing it wasn't easy. Learning a programming language goes far beyond syntax. To truly understand how Go works under the hood, I had to dive into core principles, explore related domains, and wrestle with complex ideas. And to be clear—I'm not claiming to have mastered everything. My goal is to break down what I've learned into something clear, practical, and easy to digest.

**Disclaimer:** I'm not affiliated with Google or the official Go team. Everything in this book comes from my own research and hands-on experience over the years.

## Acknowledgments

I would like to express my deepest gratitude to Matt Boyle for his unwavering support and guidance throughout the creation of this book. His vision for bytesizego.com and his belief in my ability to contribute something meaningful to the Go community were instrumental in bringing this project to life.

A heartfelt thank you to the reviewers whose keen insights and meticulous feedback helped shape this book. Their expertise and attention to detail have not only refined its content but also elevated its clarity and precision:

- Michael Bang is a pragmatic developer who loves to spend his time building maintainable software with a tiny footprint. He has worked on large parts of the stack, from R&D on NVMe disks at Samsung to web development and competitive hacking. He loves writing tests that actually provide value—a feat that is much harder than most developers believe. He fell in love with Go more than a decade ago and has used it happily ever since.
- Nicolas Leiva is a technology professional with 20 years of experience in enterprise architecture and IT infrastructure. As a Senior Cloud Network Architect at Bloomberg’s CTO Office, he helps design and improve a scalable and programmable network that supports Bloomberg’s global data and analytics platform. His work focuses on network automation, cloud networking, and data center solutions for virtual machines and Kubernetes. He holds industry certifications, including CCDE, CCIE, and RHCA, and is the author of Network Automation with Go. He also writes about technology and speaks at conferences on cloud computing, open-source software, and network automation.
- Aliaksandr Valialkin is a software engineer with more than 20 years of experience. He has been working mostly with Go for the last 10 years. He thinks Go is the best programming language, since it encourages writing clear maintainable code without unnecessary abstractions. Aliaksandr is an author of performance-oriented packages in Go such as `fasthttp`, `fastjson` and `quicktemplate`. Now he works on VictoriaMetrics and VictoriaLogs - open-source resource-efficient observability solutions for metrics and logs.
- Matthias Riegler is an engineer passionate about building efficient, scalable and reliable systems. He has built both production and development platforms from the ground up at multiple companies and now leads the Application Platform team at DE-CIX, one of the largest IXPs in the world. He loves troubleshooting and fixing low level issues and discovered his passion for Golang while building tooling for and around Kubernetes in 2017.

# Feedback

I welcome any feedback, bug reports, or corrections to improve this book. If you spot an issue or have suggestions for enhancements, please feel free to reach out. Your input is highly valued and helps make this book better for everyone.

You can file an issue on the GitHub repository (<https://github.com/func25/the-anatomy-of-go>), or just drop by to share your thoughts on the content, express how you felt while reading the book. I'd love to hear from you!

*[OceanofPDF.com](#)*

# Chapter 1: Go Thinks Simple Works Better

Go's syntax is intentionally minimalist, which leads to some important questions:

- Why does Go leave out common features like default parameter values, optional function parameters, method overloading, operator overloading and so on?
- What core ideas shaped Go's design choices?
- Does Go's simplicity limit it to beginner-friendly or small-scale projects?
- Can Go handle the demands of building complex systems?

Go's simplicity isn't a limitation—it's a strength. By cutting out unnecessary complexity, Go makes code easier to manage and lets developers focus on solving real-world problems instead of fighting with the language.

"Eliminating choices can greatly reduce anxiety"

~ Barry Schwartz *The Paradox of Choice*

## 1. What Makes Go Special

Discussions about whether a programming language is good or bad often get stuck on technical details: performance benchmarks, syntax clarity, concurrency handling, typing, garbage collection, and so on. While these factors matter, they barely scratch the surface of how a language **shapes the way** we think, design, and build software.

### 1.1 Good Code Needs More Than Code

A language can have elegant syntax and extensive features, but if the bigger picture—the development process—is flawed, it can still lead to a frustrating experience.

In this context, *process* covers the entire software development lifecycle: writing, testing, maintaining, and improving code over time. The biggest challenges

developers face often come from inefficiencies in these processes, not from the language itself.

Here are a few examples that highlight why Go was created:

1. **Compilation time:** Imagine waiting 45 minutes for your code to compile. Rob Pike, one of Go's creators, faced this exact problem ([Less is exponentially more \[less\]—Rob Pike](#)). Long compilation times kill momentum, making it harder to test ideas and iterate quickly.
2. **Tooling and libraries:** Without solid tools for tasks like refactoring or a strong library ecosystem, you waste time reinventing the wheel or fighting with inflexible code as your project grows.
3. **Too many choices:** Sometimes, having too many options becomes a burden. In JavaScript, for example, checking for `null` or `undefined` can be done with logical OR (`||`), nullish coalescing (`??`), the ternary operator (`?:`), or `if` statements. Knowing which to use—and when—can be unnecessarily complicated.

So, how can a programming language help solve these very human challenges?

## 1.2 The Go Way of Building Software

As software projects grow in size and complexity, the real challenge shifts from language features to managing that complexity effectively.

In today's distributed, highly concurrent computing environments, language features alone aren't enough. The tools and development practices built into the language play a crucial role in keeping projects manageable. Go was designed with this in mind, encouraging code that is maintainable and scalable.

That doesn't mean you can't write spaghetti code in Go—you can. Anyone can write bad code. But the language is built to steer you toward better habits. These core ideas define what's called "*the Go way*":

- **Specification:** Go began with a clear, detailed specification [[spec](#)]. This ensures all Go compilers follow the same rules, allowing developers to build their own compilers without breaking compatibility.
- **Libraries:** Go's standard libraries are well-documented, efficient, and easy to use. They cover everything from networking and cryptography to file I/O, concurrency, testing, and profiling.

- **Tools:** Go comes with a powerful set of built-in tools: automated testing (`go test`), code coverage, benchmarking, vetting (`go vet`), formatting (`go fmt`), documentation (`go doc`), dependency management (`go mod`), code generation (`go generate`), and more ([golang.org/x/tools](https://golang.org/x/tools)).
- **Concurrency:** Go simplifies concurrency with goroutines and channels (Chapter 8), moving away from traditional thread-based systems. This makes handling multiple tasks at once more straightforward—without the usual pitfalls.

Let's look at a simple yet practical example that shows how the Go compiler encourages better coding habits:

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    var message string = "Hello, Go!"
    fmt.Println(message)

    var unusedVariable int
}
```

The Go compiler will immediately throw an error and stop you from running this code because both the `net/http` import and `unusedVariable` are unused.

This strict behavior has been a topic of debate in the Go community for years. Some developers find it frustrating, especially during development—commenting out a few lines can leave variables unused, forcing you to clean up just to compile. A proposal to relax this rule was even put forward but was quickly rejected [[unw](#)]. On the other hand, many appreciate this rigidity, as it leads to cleaner, less error-prone code. It discourages bad habits and prioritizes long-term maintainability over short-term convenience.

Why not just issue a warning instead of an error?

The reasoning behind unused variable errors is explained in the Go FAQ [[faq](#)].

Warnings often get ignored, they fade into background noise. When you're focused on getting work done, how often do you actually check your IDE's

warnings tab? In Go, if something is worth mentioning, it's worth fixing. There are no warnings, only errors that demand attention.

You might consider adding a flag to disable this behavior temporarily, but it wouldn't take long for that safeguard to become permanent background noise.

Convenience can easily enable bad practices. A well-designed language isn't just about making things easy—it's about making it hard to do the wrong thing.

## 1.3 Go's Design Decisions

Design choices are always controversial. Maintainers and users often disagree because their priorities differ.

Maintainers focus on code complexity, maintainability, long-term impact, and how well a feature aligns with the project's philosophy. Users care about immediate needs, making things more convenient or solving specific problems.

At the end of the day, users rely on the software, but maintainers are responsible for keeping it running. It's up to maintainers to strike the right balance between these two sides.

### Why Go Skips the Ternary Operator

The ternary operator offers a quick way to write conditional statements in a single line—a compact alternative to an `if-else`:

```
// Ternary operator example
var max = (a > b) ? a : b;

// If-else equivalent
max := a
if a < b {
    max = b
}
```

The Go team left out the ternary operator because they believed it would encourage unnecessarily complex code. Go doesn't need two ways to handle the same logic—especially when one could lead to bad habits.

As the Go FAQ explains:

*"The reason ?: is absent from Go is that the language's designers had seen the operation used too often to create impenetrably complex expressions. The if-else form, although longer, is unquestionably clearer. A language needs only one conditional control flow construct." — Go FAQ*

Many of us like to believe we'd use the ternary operator wisely. But when deadlines are tight or you're rushing through code, it's easy to fall into the trap of writing tangled expressions like this:

```
int n = (x > 10) ? (y < 5) ? 100 : 200 : (z == 0) ? 300 :  
400;
```

Even with the best intentions, pressure can push developers toward shortcuts that make code harder to read and maintain. And this happens more often than you'd think. Since code is read far more often than it's written, this goes against Go's core philosophy of clarity and simplicity.

That said, there are some creative ways developers try to mimic the ternary operator's behavior in Go:

```
func Ternary[T any](cond bool, a, b T) T {  
    if cond {  
        return a  
    }  
    return b  
}  
  
max := Ternary(a > b, a, b)  
  
// Another approach  
max := map[bool]int{true: a, false: b}[a > b]
```

The problem with these patterns is that all arguments are evaluated as soon as the function is called—or when accessing the map—before any decision is made about which value to use. This can lead to unexpected issues:

```
b := Ternary(a != nil, a.Thing, "default")
```

In this example, `a.Thing` will be evaluated no matter what—even if `a` is `nil`. If `a` is `nil`, the program will crash with a panic.

Unlike a true ternary operator, these workarounds evaluate both branches regardless of the condition, which can lead to bugs if you're not careful. To avoid

this, you can delay execution using functions:

```
func Ternary[T any](cond bool, f1 func() T, f2 func() T) T {
    if cond {
        return f1()
    }
    return f2()
}

aThing := func() string { return a.Thing }
defaultValue := func() string { return "default" }

Ternary(a != nil, aThing, defaultValue)
```

At that point, the extra effort defeats the purpose of using a simple conditional expression. The added complexity makes it more trouble than it's worth.

## How Go Handles Initial Values

In C# or JavaScript, you can set default values for struct fields directly when defining them. This makes it easy to assign initial values without manually setting them every time you create a new instance:

C#

```
public class Example {
    private int attribute1 = 10;
    private String attribute2 = "default string";
}
```

JavaScript

```
class Example {
    constructor(attribute1 = 10, attribute2 = "default
string") {
        this.attribute1 = attribute1;
        this.attribute2 = attribute2;
    }
}
```

Go takes a different approach. Instead of letting you define custom default values, Go automatically assigns a **zero value** to any fields you don't explicitly set. The zero value depends on the type: numbers default to `0`, strings to an empty string (`""`), booleans to `false`, and slices, maps, channels, and pointers default to `nil`.

There was a proposal to allow custom default values for struct fields in Go 2 [\[def\]](#), but it was ultimately rejected.

To simulate default values in Go, a common pattern is to use a `New` function:

```
type Person struct {
    Name string
    Age  int
}

func NewPerson() *Person {
    return &Person{Name: "Unknown", Age: 18}
}
```

Another method is to create an `Init` function for manual initialization:

```
func (p *Person) Init() {
    p.Name = "Unknown"
    p.Age = 18
}

func main() {
    p := new(Person)
    p.Init()
}
```

Each method has its downsides:

- There's no guarantee users will use `NewPerson()` instead of directly initializing with `Person{}`.
- Clients of your code must remember to call `Init()`. Skipping this step can lead to unexpected behavior or incomplete initialization.

One way to enforce proper initialization is to make the `Person` struct unexported by renaming it to `person`. However, this creates a new issue—users won't be able to reference `person` elsewhere if they need to pass it to a function or access it directly.

The reason behind this design choice, according to the Go team's stance: the zero value should be meaningful [[szv](#)]. This principle is built on a few core ideas:

- Structs should be designed so that their zero value is valid and usable.
- While this might feel restrictive in some cases, it eliminates the need for complex constructor patterns, recursive initializers, or handling partially initialized objects.
- Allowing custom default values would introduce runtime overhead, as the system would need to handle additional setup before a struct could be used.

Adding that level of complexity would go against Go's core philosophy of simplicity and efficiency without providing enough benefits to justify the trade-off.

The takeaway is clear: instead of relying on explicit initialization, Go encourages struct designs where the zero value is valid and meaningful. This reduces the need for constructors and minimizes the risk of errors caused by uninitialized fields.

## When Simplicity Met Generics

Generics (Chapter 4) finally arrived in Go with version 1.18 [[n1.18](#)]—a long-awaited feature that quickly became one of the most significant additions to the language. It had been one of the community's most requested features for good reason: it dramatically expands what Go can do.

As a statically typed language, Go traditionally required writing separate functions or data structures for each type, even when the logic was identical. This led to repetitive code, increased maintenance effort, and made reusability harder.

Consider a simple `max` function. Without generics, you'd have to write multiple versions, `maxInt`, `maxFloat`, `maxString`, even though the core logic remains the same.

While you could use `interface{}` with type assertions, generics let you achieve the same outcome in a cleaner, type-safe way:

```
func Max[T constraints.Ordered](a, b T) T {
    if a > b {
        return a
    }
    return b
}
```

```
func main() {
    fmt.Println(Max(1, 2))
    fmt.Println(Max(1.0, 2.0))
    fmt.Println(Max("Hello", "World"))
}
```

Generics allow you to write functions and data structures without tying them to specific types. This makes your code more flexible, easier to maintain, and eliminates redundancy—you write it once, and it works for any compatible type.

That flexibility comes with added complexity, particularly in how Go's type system now operates:

- **Type parameters:** Functions, types, and methods can now accept type parameters, introducing a new layer of complexity for both the compiler and the developer.
- **Type inference:** The compiler must determine which types to use based on how generic functions and types are called, adding complexity during compilation.
- **Constraints and type bounds:** Generics don't accept just any type—you must use constraints (`comparable`, `any`, `constraints.Ordered`, etc.) to define allowed types. These constraints often rely on interfaces that serve as contracts for valid types.

Generics bring both power and also complexity. Take this overloaded example:

```
func MergeAndConvert[T1, T2, R any]{
    slice1 []T1,
    slice2 []T2,
    convert1 func(T1) R,
    convert2 func(T2) R,
} []R {
    ...
}
```

Even just reading the function signature can feel overwhelming. Despite this complexity, generics were introduced because the demand from the community was clear—and justified.

The need for generics came from practical challenges, particularly from developers tired of writing repetitive boilerplate code that could be simplified. The Go team was cautious about adding generics, aiming to avoid unnecessary complexity that could compromise Go's hallmark simplicity and readability.

Over time, it became clear that generics would help developers write cleaner, more reusable code—especially for libraries built to handle different types. With version 1.18, the Go team introduced a well-considered generics model that stayed true to Go’s core philosophy of simplicity while meeting the needs of the community.

One way Go maintains this simplicity is through type inference, making generic functions feel as natural to use as regular ones:

```
func printSomething(t string) {
    print(t)
}

func printSomethingG[T any](t T) {
    print(t)
}

func main() {
    printSomething("Hello, World!")
    printSomethingG("Hello, World!")
}
```

Calling `printSomethingG` feels no different from calling a regular function—no need to specify the type manually; it just works.

Whether generics are a good or bad addition depends on your perspective. Developers come from different backgrounds and bring unique experiences to the table. What’s clear is that the introduction of generics reflects an ongoing, passionate debate within the Go community [[gnrp](#)].

## 2. How Go Runs Your Code

Go shares some DNA with its older sibling, C. Both are compiled and statically typed languages, but understanding what that actually means—and how they differ—sets the foundation for how Go works under the hood.

### Compiled Languages

Go, C, and languages like Rust are compiled. The code you write doesn’t run directly. Instead, it goes through a compilation process, translating it into machine code—a set of low-level instructions the CPU can execute directly.

In contrast, programs written in interpreted languages are executed at runtime using an interpreter, which processes the code dynamically rather than compiling it beforehand. (Some interpreters execute high-level code directly, while others run bytecode—an intermediate representation of the source code.)

The key advantage of compiled languages is speed. Since the code is already translated into machine code before execution, there's no overhead during runtime. No interpreter, no delays—just raw performance.

However, the trade-off is *portability*. A compiled program is tied to the architecture it was built for. To run it on another platform, you'll need to recompile it for that specific system.

Static typing is another core feature. Go and C are statically typed, meaning variable types are determined at compile time. You declare the type upfront, and it doesn't change. This stability allows the compiler to optimize more aggressively, generating faster, more efficient machine code. It also prevents type-related errors before the program even runs, making the code more predictable and reliable.

To see how Go and C compare in practice, let's look at something simple: a basic “Hello, World!” program. Even this small example highlights some key differences between the two.

C	Go
<pre>#include &lt;stdio.h&gt;  int main() {     printf("Hello, World!\n");     return 0; }</pre>	<pre>package main  import "fmt"  func main() {     fmt.Println("Hello, World!") }</pre>

Compiling this simple program in both Go and C takes only a fraction of a second. Here's a comparison of the binary sizes produced on a local machine:

```
Binary sizes:
- C binary size: 33K
- Go binary size: 2.1M
- Go binary size (no debug): 1.4M
```

*These results were obtained using Go 1.23.5 and Apple Clang version 16.0.0 (clang-1600.0.26.6).*

The second Go binary is built with the `-s` flag, which strips the symbol table and debug information, reducing the size.

## Stripping debug information

The `-s` flag disables **symbol table generation**. The symbol table holds information about functions, variables, and their locations in the binary. This data is stored in the `.symtab` and `.strtab` sections for ELF (Unix-like systems) binaries. On macOS (Mach-O binaries), it suppresses `STAB` symbols used for debugging.

Using `-s` automatically implies `-w` unless explicitly overridden (`-w=false`). The `-w` flag disables **DWARF** generation, which removes detailed debug information like source code, line numbers, and variable data.

The main benefit of these flags is reducing binary size, which is particularly useful in production environments where debug data isn't necessary or when working with limited storage. The reduction depends on the size and complexity of your codebase.

Go's binary is noticeably larger—up to 63 times larger than C's. Even with debug information stripped, this version still comes in at 42 times larger.

This size difference exists because every Go binary is statically linked with the Go runtime, even for simple programs. That means the binary includes essential components like the garbage collector, memory allocator, goroutine scheduler, and so on by default.

However, this is a **one-time cost**. As your application grows, the runtime overhead becomes less significant relative to the overall size of the program.

## The Heart of Go: Compiler & Runtime

The Go compiler transforms high-level code into a standalone binary that runs directly on your machine. Go favors statically linked binaries, meaning everything your program needs—your code, libraries, and the Go runtime itself—is bundled into a single file.

To see this in action, let's revisit the classic "*Hello, World!*" example, but this time, we'll skip the `fmt` package and use `print` instead:

```
package main

func main() {
    print("Hello, World!")
}
```

You can inspect what's happening under the hood by compiling this program with flags that reveal the assembly output (`go build -gcflags "-S"`). The output might look cryptic—it's Go assembly, after all—but it reveals how the runtime operates beneath the surface:

```
main.main STEXT size=80 args=0x0 locals=0x18 funcid=0x0 align=0x0
0x0000 00000 (main.go:3) TEXT main.main(SB), ABIIInternal, $32-0
...
0x0018 00024 (main.go:4) CALL runtime.printlock(SB)
0x001c 00028 (main.go:4) MOVD $go:string."Hello, World!"(SB), R0
0x0024 00036 (main.go:4) MOVD $13, R1
0x0028 00040 (main.go:4) CALL runtime.printstring(SB)
0x002c 00044 (main.go:4) CALL runtime.printunlock(SB)
...
0x0030 00048 (main.go:5) LDP -8(RSP), (R29, R30)
...
0x003c 00060 (main.go:3) MOVD R30, R3
0x0040 00064 (main.go:3) CALL runtime.morestack_noctxt(SB)
0x0044 00068 (main.go:3) PCDATA $0, $-1
...
print("Hello, World!")
```

### Illustration 1. A Quick Look at the Go Assembly Code

Even for something as simple as printing a string, Go's runtime handles a surprising amount of work. The `print` function gets translated into several calls within the `runtime` package. The compiler first hoists all argument evaluations before acquiring a lock. Then, based on the argument's type, it selects the appropriate runtime function:

- For booleans: `runtime.printbool`
- For integers: `runtime.printint`
- For strings: `runtime.printstring`
- For floats: `runtime.printfloat`
- For slices: `runtime.printslice`
- And more, found in `src/runtime/print.go`.

Each of these functions runs between `runtime.printlock()` and `runtime.printunlock()` to ensure thread safety, preventing output from overlapping when multiple goroutines run concurrently.

At the end, `runtime.morestack_noctxt` steps in to handle stack growth automatically. This mechanism allows Go to adjust a goroutine's stack size on the fly—an essential feature we'll cover in more detail later.

### 3. Summary

Go was designed with a clear mission: make coding more productive and enjoyable. It doesn't aim to impress with flashy features or complex syntax. Instead, it's a solid, dependable tool built to get the job done.

What sets Go apart is its willingness to break its own rules when it makes sense. Take generics, for example. For years, Go resisted adding this feature, despite heavy demand from developers. But when it finally arrived in version 1.18, it stayed true to Go's core philosophy—simple, practical, and focused on solving real problems without adding unnecessary complexity.

Beneath its clean surface, Go has a lot more going on than it seems. Even a basic "*Hello, World!*" program taps into serious under-the-hood power, including garbage collection and built-in concurrency.

In the end, Go's simplicity is intentional, not a limitation. It's about making code predictable and manageable, even as projects grow. In a tech landscape where complexity often spirals out of control, Go offers something rare: a reminder that sometimes, less really is more.

## References

- [szv] object/struct default initialization: <https://groups.google.com/g/golang-nuts/c/ZwXGldT4FOM>
- [unw] proposal: spec: make unused variables not an error: <https://github.com/golang/go/issues/43729>
- [gnrp] Generics in Go: <https://github.com/golang/go/issues/15292>
- [less] Less is exponentially more by Rob Pike. 2012: <https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html>

- [gomm] Updating the Go Memory Model by Russ Cox. 2021: <https://research.swtch.com/gomm>
- [n1.18] Go 1.18 Release Notes: <https://tip.golang.org/doc/go1.18>
- [faq] Go Frequently Asked Questions (FAQ): <https://go.dev/doc/faq>
- [def] proposal: Go 2: default initializers for types #32076: <https://github.com/golang/go/issues/32076>
- [spec] The Go Programming Language Specification: <https://go.dev/ref/spec>

*[OceanofPDF.com](#)*

# Chapter 2: Basic Data Types, Variables, and Constants

## 1. Integer Types: Bit-Width, Semantics, and System Architecture

Go offers several integer types, with the number in their name indicating how many bits they use—and, by extension, how much memory they occupy.

There are types for both signed (`int`, `int8`, `int16`, `int32`, `int64`) and unsigned (`uint`, `uint8`, `uint16`, `uint32`, `uint64`) integers. Go also includes specialized types: `byte` and `rune`, which are aliases for `uint8` and `int32`, respectively, and `uintptr`, a distinct integer type large enough to hold the bit pattern of any pointer.

The value ranges for each type are as follows:

```
int8: -128 to 127
int16: -32,768 to 32,767
int32: -2,147,483,648 to 2,147,483,647
int64: -9,223,372,036,854,775,808 to
      9,223,372,036,854,775,807

uint8: 0 to 255
uint16: 0 to 65,535
uint32: 0 to 4,294,967,295
uint64: 0 to 18,446,744,073,709,551,615
```

The `int` and `uint` types work a bit differently compared to other types. They don't have a fixed size:

```
fmt.Println("- GOARCH:", runtime.GOARCH)
fmt.Println("- int size:", unsafe.Sizeof(int(0)))
```

**64-bit systems**

**32-bit systems**

- GOARCH: arm64  
- int size: 8 bytes

- GOARCH: 386  
- int size: 4 bytes

Rather, their size is determined by the system's architecture:

- On a **32-bit system**: an `int` takes up 4 bytes, matching `int32` and shares the same range of values.
- On a **64-bit system**: an `int` takes up 8 bytes, like `int64`, which allows it to handle much larger numbers. The same behavior applies to `uint`.

The `int` type offers a flexible, architecture-dependent size that lets the compiler optimize for the underlying hardware. In most cases, `int` is both sufficient and efficient for common tasks like array indexing, loop counters, or general arithmetic; situations where the exact size of the integer isn't critical.

When you need precise control over integer size, such as when parsing protocol buffers or working with hardware APIs, Go provides explicitly sized types like `int32` and `int64`.

The reason `int` and `uint` depend on system architecture comes down to **word size** (also known as **native size**). This refers to the largest chunk of data the CPU can process in a single operation, as well as the amount of memory it can handle in one read or write.

- On a 32-bit system, the word size is 32 bits (4 bytes).
- On a 64-bit system, the word size jumps to 64 bits (8 bytes).

## Word Size and Performance

Two key aspects of word size impact performance: general-purpose registers and memory address space. Both scale with the system's word size:

- **CPU's general-purpose registers**: These registers act as the CPU's workspace, temporarily holding data during program execution. They handle tasks like calculations and memory addressing. Their size typically matches the system's word size, allowing the CPU to process data efficiently without extra conversions or adjustments.
- **Memory address space**: This defines the range of memory locations available for storing data. A 32-bit system limits the address space to  $2^{32}$  locations, which allows access to up to 4GB of memory. A 64-bit system expands this

limit to  $2^{64}$  locations, theoretically supporting up to 16 exabytes (EB) of memory.

When an integer fits within the CPU's general-purpose registers, the processor can load, store, and manipulate that value in a single operation. There's no need to split the data into smaller pieces or reassemble it later, which makes operations faster and more efficient.

## Integer Literals and Number Formats

You're not limited to writing numbers in standard decimal form when working with integers. Go lets you represent numbers in different formats, depending on what makes the most sense for your code. These are called **int literals**, and they can be written as:

- **Decimal**: Standard base-10 format, just write the number directly.
- **Binary**: Base 2, starting with `0b` or `0B`, followed by a sequence of 0s and 1s.
- **Octal**: Base 8, starting with `0o` or `0O`, followed by digits 0 through 7. You can also use a leading `0`, but that can be confusing and is generally avoided.
- **Hexadecimal**: Base 16, starting with `0x` or `0X`, using digits 0-9 and letters `a - f` (or `A - F`).

*These formats require Go version 1.13 or later [\[g1.13\]](#).*

```
a := 10          // Decimal  
  
b1 := 0b1010    // Binary  
b2 := 0B1010    // Binary  
  
o1 := 0o12      // Octal  
o2 := 0012      // Octal  
o3 := 012       // Octal (not recommended)  
  
h1 := 0xface    // Hexadecimal  
h2 := 0XFaCe    // Hexadecimal
```

Different formats serve different purposes:

- **Binary** is useful for bit manipulation, setting flags, or working directly with bits.
- **Octal** is often used for file permissions.
- **Hexadecimal** is ideal for memory addresses, color values in graphics, and low-level programming.

```

// Binary usage
num := 29          // 11101 in binary
mask := 0b11        // 00011: Mask for the last 2 bits
lastTwoBits := num & mask // 01: Extract the last 2 bits with
bitwise AND

// Octal usage for file permissions
permission := 0755 // Owner: rwx, Group: r-x, Others: r-x

// Hexadecimal usage
address := 0x1400004e740 // Memory address
color := 0x0000FF        // Blue color code

```

Starting with Go 1.13, you can use underscores in numeric literals to improve readability. This is especially useful for long numbers:

Valid	Invalid
<pre> a := 1_000_000 b := 0b_1010_1010 c := 0x_1_2_3_4_5_6_7_8 </pre>	<pre> d := _1234      // Cannot start with an underscore e := 0_x1234    // Cannot place an underscore after the prefix g := 1_000_000_ // Cannot end with an underscore h := 1__0       // Cannot use consecutive underscores </pre>

However, this comes with some obvious restrictions:

- You can't start or end a number with an underscore.
- You can't place underscores directly after a prefix (like `0x` or `0b`).
- Double underscores in sequence are not allowed.

## Overflow and Underflow: Behavior and Pitfalls

When a value exceeds the limits of its data type, it causes either overflow (when the value is too large) or underflow (when it's too small).

Take `int8` as an example, it can hold values from `-128` to `127`. If you push beyond that range, the value wraps around:

```

var a int8 = 127
a++ // Wraps around to -128 (overflow)
a-- // Wraps back to 127 (underflow)

```

Go handles overflow differently depending on the type. For integers, it uses two's complement arithmetic for signed integers and modulo arithmetic for unsigned integers. This is quite different from the behavior of floating-point numbers and constants, which we'll cover later.

It's up to developers to manage overflow. While Go can catch some cases at compile time, it won't cover everything, especially for calculations that happen at runtime. The following example shows both compile-time overflow errors and runtime overflow (which does not produce an error):

```
func main() {
    // Compile error: 128 is too large for int8
    var a int8 = 128

    // This runs, but overflows at runtime
    var b int8 = 127
    var c = b + 1 // Wraps around to -128
}
```

This wrapping behavior is not introduced by Go but rather follows the hardware's behavior. It occurs because most CPUs use fixed-width registers to store numbers, meaning there's a limit to how many bits are available. When a value exceeds that limit, the processor discards the extra bits, causing the number to wrap around instead of throwing an error.

## Explicit Conversion Between Integer Types

Go doesn't allow automatic conversion between different integer types—you have to be explicit. This might make your code a bit more verbose, but it prevents silent mistakes that could slip through with automatic conversions.

For those new to Go, this might feel a bit frustrating at first:

```
var a int8 = 10
var b int16 = int16(a)      // Explicit conversion
var c = int32(int16(a) + b) // More explicit conversion
```

If Go allowed automatic conversions, you'd risk issues like truncation, unexpected sign changes, or other subtle bugs. By forcing explicit conversions, the language makes your intentions clear and reduces the chances of unexpected behavior. As the Go FAQ [[cfg](#)] puts it:

*"The convenience of automatic conversion between numeric types in C is outweighed by the confusion it causes. When is an expression unsigned? How*

*big is the value? Does it overflow? Is the result portable, independent of the machine on which it executes?"*

## ~ Go FAQ

Being explicit doesn't mean that Go guarantees safety. Converting between types with different ranges can still lead to data loss. The upside is that this risk is more obvious, making it easier to catch and handle carefully:

```
var a int16 = 130
var b int8 = int8(a) // Overflow (>127), wraps around, b
becomes -126

var u int8 = -1
var v uint8 = uint8(u) // Underflow (<0), wraps around, v
becomes 255
```

One exception is constants. Since they are handled at compile time, they don't risk overflow or underflow during runtime:

```
const a = 255

var b uint8 = a
var c uint8 = a + 1 // cannot use a + 1 (untyped int constant
256) as uint8 value in variable declaration (overflows)
```

Of course, this doesn't apply to constants in expressions that can't be evaluated at compile time. We'll revisit this topic when we dive deeper into how constants work.

## Division by Zero: Compile-Time and Runtime Safeguards

Go takes division by zero seriously. If you divide by zero using integers, Go compiler stops you immediately with a compile-time error. And if, by some chance, the check slips past during compilation, you'll hit a runtime panic:

```
func main() {
    a := 0
    println(a / 0) // compile error: invalid operation:
division by zero (compile-time error)
    println(10 / a) // panic: runtime error: integer
divide by zero
}
```

The example above shows both scenarios clearly: a compile-time error and a runtime panic. How does the runtime detect this and turn it into a panic?

On most processors, dividing by zero triggers a hardware fault and sends a signal to the application. The runtime catches that via a signal handler and converts it into a panic by calling `panicdivide()`. On platforms without reliable hardware faults for division by zero, the compiler inserts an explicit check: if the divisor is zero, execution jumps directly to the `panicdivide()` path instead of attempting the divide.

Figure 2. Panic for Integer Division by Zero (runtime/panic.go)

```
package runtime

func panicdivide() {
    panicCheck2("integer divide by zero")
    panic(divideError)
}
```

At a lower level, Go's assembly code follows the same logic. This offers a clear look at how Go handles errors at the system level. If you're new to assembly, it's worth paying attention, as more examples like this will appear throughout the book:

```
00112 CBNZ      R2, 120 ; if R2 (the divisor) is not zero,
jump to 120
00116 JMP       164      ; if zero, jump to 164, which
triggers the panic

00164 CALL      runtime.panicdivide(SB)
```

Here, the `R2` register holds the divisor. The `CBNZ` (*Compare and Branch on Non-Zero*) instruction checks whether it's zero. If it is, the program jumps directly to `panicdivide()` at address `00164`.

While Go keeps integer division safe and simple, floating-point operations require a more nuanced understanding.

## 2. Floating-Point Types: Precision, Representation, and IEEE-754 Semantics

Go provides two floating-point types: `float32` and `float64`. These correspond to single-precision and double-precision floating-point numbers, following the IEEE-754 standard:

```
var a float32 = 0.1
var b float64 = 0.0001
```

Despite using the same number of bits as integers (`int32`, `int64`), floating-point numbers can represent a much larger range of values at the cost of precision:

```
float32: 1.4e-45 to 3.4e+38  
float64: 4.9e-324 to 1.8e+308
```

In Go's early days, before version 1 was released in 2012, there was actually a `float` type, similar to how `int` works today. It would adjust to the system's architecture. But in practice, this flexibility didn't offer any real advantage. Instead, it made the type system unnecessarily complex. To keep things simple, the Go team dropped `float` in 2011 and stuck with explicit `float32` and `float64` types.

Floating-point numbers work differently from integers at a fundamental level. Precision matters in every calculation, no matter how small, because even tiny rounding errors can affect the result.

With integers, the compiler can lean on the hardware for most operations. But floating-point math often requires balancing precision and performance, depending on the context. On mobile devices or GPUs, `float32` might be favored for speed and efficiency. In contrast, some applications demand the extra accuracy that comes with `float64`. Even with small values, the precision gap between `float32` and `float64` becomes noticeable. For those interested in the finer details, the discussion on **Unsized integer types** in the *Golang Nuts group* [div] offers more insight.

When a floating-point type isn't explicitly defined, the default is always `float64`, regardless of the architecture:

```
a := 0.1  
fmt.Printf("Type of a is %T\n", a) // "Type of a is float64"
```

The default `float64` type is chosen for precision. `float64` is the standard in Go's library because it offers greater accuracy for floating-point calculations. In contrast, `float32` loses precision quickly.

## 2.1 Floating-Point Literals and Notation Styles

Floating-point numbers, or literals, can be written in three main formats: decimal, scientific notation, and hexadecimal.

- **Decimal form:** The standard format, with a dot separating the whole number and fractional part.

- **Scientific notation:** Useful for very large or very small numbers, written as a base number followed by an `e` or `E` and an exponent.
- **Hexadecimal form:** Similar to hexadecimal integers, but used for floating-point numbers, with a base of `2` instead of `10`.

```
// Decimal form
d1 := 0.1
d2 := 1.
d3 := .1

// Scientific notation
s1 := 1.2e3 // 1.2 * 10^3 = 1200
s2 := 1.E-3 // 1 * 10^-3 = 0.001
s3 := .2e3 // 0.2 * 10^3 = 200

// Hexadecimal form
h1 := 0x1.2p3 // (1 + 2/16) * 2^3 = 1.125 * 8 = 9
h2 := 0x1P3 // 1 * 2^3 = 8
h3 := 0X1.p3 // 1 * 2^3 = 8
```

Each format has its place, but they don't mix. For example, using scientific notation (`e` or `E`) in a hexadecimal number isn't valid.

The term **floating-point literals** can be misleading—it sounds like these formats are only for floats. In reality, both integers and floating-point numbers can use some of the same formats when it makes sense:

```
var a int = 1e3 // 1000
var b float64 = 0x1 // 1
```

Of course, you can't assign a fractional value to an integer type:

```
// Error: cannot use 1e-1 (untyped float constant 0.1)
// as int value in variable declaration (truncated)
var a int = 1e-1

var b int = 1.2e3 // This works fine, equals 1200
```

Go treats these number formats as untyped constants by default, meaning they can be converted to whatever type fits your program—as long as the value makes sense for that type.

## 2.2 Type Conversion and Precision Trade-Offs

When converting between different floating-point types, you need to do it explicitly, just as you would with `int` types. However, floats come with their own

challenges—subtle details that can lead to unexpected results if you’re not careful.

Converting from `float32` to `float64` is generally safe since `float64` can handle larger numbers with higher precision. The problem arises when going the other way—from `float64` to `float32`.

Here, you risk losing precision because `float32` simply can’t store as much detail:

```
// Precision loss
var a float64 = 1.123456789
var b float32 = float32(a) // 1.1234568 (some precision is
lost)
```

Converting a `float` to an `int` is much more straightforward, and harsher. Go doesn’t round the number; it simply cuts off the decimal part.

Likewise, converting from an `int` to a `float` can lead to precision loss, especially with larger numbers:

```
var a float64 = 1.999999
var b int = int(a) // 1 (decimal is dropped, not rounded)

var c int = 123456789
var d float32 = float32(c) // 123456792 (some rounding
happens)
```

## Using Floats for Money

A common mistake is using floats to handle money. At first impression, it might seem like a natural fit for dollars and cents. The problem is, floats aren’t precise enough for financial calculations—rounding errors can accumulate quickly, leading to inaccurate results. A better approach is to use integers (for example, store cents as an `int`), fixed-point arithmetic, or specialized decimal types built for financial precision.

Be mindful when converting between types that handle different levels of precision and range. A careless conversion can introduce subtle bugs that are really hard to track down.

### 2.3 IEEE-754 Compliance and Internal Behavior

Now, Go follows the IEEE-754 standard (IEEE Standard for Floating-Point Arithmetic) [ieee] for storing floats in memory. These quirks aren't unique to Go, they're common across most programming languages that follow the same standard. If you're familiar with IEEE-754, this will feel familiar. If not, some of these behaviors might seem counterintuitive.

We won't dive too deep into the IEEE-754 standard here. Our focus is on what matters most for working with floating-point numbers in Go. Still, there are a few key points worth knowing about how Go handles floats under this standard.

## Special Floating-Point Values: +Inf, -Inf, and NaN

Floating-point division by zero works differently from integer division. Instead of triggering an error, Go returns special values depending on the situation:

- Dividing a positive float by zero returns `+Inf` (positive infinity).
- Dividing a negative float by zero returns `-Inf` (negative infinity).
- Dividing zero by zero returns `NaN`, which stands for "Not a Number."

```
func main() {
    zero := 0.0

    println(1 / zero)      // +Inf
    println(-1 / zero)    // -Inf
    println(zero / zero) // NaN
}
```

`+Inf`, `-Inf`, and `NaN` have unique behavior in both operations and comparisons. Any operation involving `NaN` will always result in `NaN`:

```
func main() {
    NaN := math.NaN()
    Inf := math.Inf(1)
    NegInf := math.Inf(-1)

    println(NaN - 100)    // NaN - 100 = NaN
    println(NaN + Inf)    // NaN + +Inf = NaN
    println(NaN + NegInf) // NaN + -Inf = NaN
}
```

When working with `+Inf` or `-Inf`, the result depends on the operation and the values involved. You can end up with `+Inf`, `-Inf`, or `NaN`:

```
println(Inf + Inf)      // +Inf + +Inf = +Inf
println(NegInf + NegInf) // -Inf + -Inf = -Inf
println(NegInf + Inf)   // -Inf + +Inf = NaN
```

Comparisons involving `Nan` behave differently from any other value. Comparing `Nan` to anything, including another `Nan`, always returns `false`. `Nan` refuses to equal anything, not even itself:

```
println(Nan == Nan) // false
println(Nan != Nan) // true
```

`+Inf` is greater than any finite number, and `-Inf` is smaller than any finite number. Comparing `+Inf` to `+Inf` or `-Inf` to `-Inf` returns `true`. As expected, `+Inf` and `-Inf` don't match:

```
println(Inf > 10)        // true
println(Inf == Inf)       // true
println(NegInf < 10)      // true
println(NegInf == NegInf) // true
```

Since `Nan` doesn't even equal itself, the simplest way to check for `Nan` is by using the condition `if n != n`:

```
n := math.NaN()
if n != n {
    fmt.Println("n is NaN")
}
```

For clearer and more readable code, use the built-in `math` package:

```
import "math"

math.IsNaN(n)
math.IsInf(n, 0) // 0: any infinity, -1: negative infinity,
1: positive infinity
```

Behind the scenes, `math.IsNaN` works the same way as `n != n`, but using the explicit function makes the intention clearer.

## Rounding Errors, Approximation, and Precision Loss

Floating-point numbers come with hidden pitfalls, especially when it comes to rounding errors and precision loss. These issues are common in the Go community, often showing up when calculations produce results that seem incorrect at first glance.

A frequent example of this occurs when `a + b` doesn't equal `c` as expected:

```
a := 0.1
b := 0.2
c := 0.3

a + b == c // false
a + b      // 0.30000000000000004
c          // 0.3
```

The issue lies in how floating-point numbers are stored. Values like `0.1` and `0.3` can't be represented exactly in binary floating-point format. Just as `1/3` becomes an infinite repeating decimal (`0.33333...`) in base 10, some numbers can't be stored precisely in binary due to the limited number of bits available.

When comparing `a + b` to `c`, you're actually comparing two approximations. Small differences in how these numbers are represented can cause equality checks to fail, even if the result looks correct when printed.

Another clear example of precision loss happens with large floating-point numbers:

```
a := float32(16777216)
b := a + 1

a == b // true
b      // 16777216
```

This behavior follows the IEEE-754 standard. A `float32` can accurately represent whole numbers only up to `16,777,216`. Beyond that, it loses the ability to distinguish between closely spaced values.

In this case, adding `1` should produce `16,777,217`. However, `float32` lacks the precision at this range, so it rounds the result back to `16,777,216`. This is why `a == b` evaluates to `true`.

The table below shows how `float32` starts losing precision as numbers grow larger, eventually grouping distinct values into the same representation:

## Number    Float32

16777216	16777216
16777217	16777216
16777218	16777216
16777219	16777216
16777220	16777216
16777221	16777216
16777222	16777216
16777223	16777216
16777224	16777216
16777225	16777216
16777226	16777216
16777227	16777216
16777228	16777216
16777229	16777216
16777230	16777216
16777231	16777216
16777232	16777216
16777233	16777216
16777234	16777216
16777235	16777216
16777236	16777216
16777237	16777216
16777238	16777216
16777239	16777216
16777240	16777216
16777241	16777216
16777242	16777216
16777243	16777216
16777244	16777216
16777245	16777216
16777246	16777216
16777247	16777216
16777248	16777216
16777249	16777216
16777250	16777216
16777251	16777216
16777252	16777216
16777253	16777216
16777254	16777216
16777255	16777216
16777256	16777216
16777257	16777216
16777258	16777216
16777259	16777216
16777260	16777216
16777261	16777216
16777262	16777216
16777263	16777216
16777264	16777216
16777265	16777216
16777266	16777216
16777267	16777216
16777268	16777216
16777269	16777216
16777270	16777216
16777271	16777216
16777272	16777216
16777273	16777216
16777274	16777216
16777275	16777216
16777276	16777216
16777277	16777216
16777278	16777216
16777279	16777216
16777280	16777216
16777281	16777216
16777282	16777216
16777283	16777216
16777284	16777216
16777285	16777216
16777286	16777216
16777287	16777216
16777288	16777216
16777289	16777216
16777290	16777216
16777291	16777216
16777292	16777216
16777293	16777216
16777294	16777216
16777295	16777216
16777296	16777216
16777297	16777216
16777298	16777216
16777299	16777216
16777300	16777216
16777301	16777216
16777302	16777216
16777303	16777216
16777304	16777216
16777305	16777216
16777306	16777216
16777307	16777216
16777308	16777216
16777309	16777216
16777310	16777216
16777311	16777216
16777312	16777216
16777313	16777216
16777314	16777216
16777315	16777216
16777316	16777216
16777317	16777216
16777318	16777216
16777319	16777216
16777320	16777216
16777321	16777216
16777322	16777216
16777323	16777216
16777324	16777216
16777325	16777216
16777326	16777216
16777327	16777216
16777328	16777216
16777329	16777216
16777330	16777216
16777331	16777216
16777332	16777216
16777333	16777216
16777334	16777216
16777335	16777216
16777336	16777216
16777337	16777216
16777338	16777216
16777339	16777216
16777340	16777216
16777341	16777216
16777342	16777216
16777343	16777216
16777344	16777216
16777345	16777216
16777346	16777216
16777347	16777216
16777348	16777216
16777349	16777216
16777350	16777216
16777351	16777216
16777352	16777216
16777353	16777216
16777354	16777216
16777355	16777216
16777356	16777216
16777357	16777216
16777358	16777216
16777359	16777216
16777360	16777216
16777361	16777216
16777362	16777216
16777363	16777216
16777364	16777216
16777365	16777216
16777366	16777216
16777367	16777216
16777368	16777216
16777369	16777216
16777370	16777216
16777371	16777216
16777372	16777216
16777373	16777216
16777374	16777216
16777375	16777216
16777376	16777216
16777377	16777216
16777378	16777216
16777379	16777216
16777380	16777216
16777381	16777216
16777382	16777216
16777383	16777216
16777384	16777216
16777385	16777216
16777386	16777216
16777387	16777216
16777388	16777216
16777389	16777216
16777390	16777216
16777391	16777216
16777392	16777216
16777393	16777216
16777394	16777216
16777395	16777216
16777396	16777216
16777397	16777216
16777398	16777216
16777399	16777216
16777400	16777216
16777401	16777216
16777402	16777216
16777403	16777216
16777404	16777216
16777405	16777216
16777406	16777216
16777407	16777216
16777408	16777216
16777409	16777216
16777410	16777216
16777411	16777216
16777412	16777216
16777413	16777216
16777414	16777216
16777415	16777216
16777416	16777216
16777417	16777216
16777418	16777216
16777419	16777216
16777420	16777216
16777421	16777216
16777422	16777216
16777423	16777216
16777424	16777216
16777425	16777216
16777426	16777216
16777427	16777216
16777428	16777216
16777429	16777216
16777430	16777216
16777431	16777216
16777432	16777216
16777433	16777216
16777434	16777216
16777435	16777216
16777436	16777216
16777437	16777216
16777438	16777216
16777439	16777216
16777440	16777216
16777441	16777216
16777442	16777216
16777443	16777216
16777444	16777216
16777445	16777216
16777446	16777216
16777447	16777216
16777448	16777216
16777449	16777216
16777450	16777216
16777451	16777216
16777452	16777216
16777453	16777216
16777454	16777216
16777455	16777216
16777456	16777216
16777457	16777216
16777458	16777216
16777459	16777216
16777460	16777216
16777461	16777216
16777462	16777216
16777463	16777216
16777464	16777216
16777465	16777216
16777466	16777216
16777467	16777216
16777468	16777216
16777469	16777216
16777470	16777216
16777471	16777216
16777472	16777216
16777473	16777216
16777474	16777216
16777475	16777216
16777476	16777216
16777477	16777216
16777478	16777216
16777479	16777216
16777480	16777216
16777481	16777216
16777482	16777216
16777483	16777216
16777484	16777216
16777485	16777216
16777486	16777216
16777487	16777216
16777488	16777216
16777489	16777216
16777490	16777216
16777491	16777216
16777492	16777216
16777493	16777216
16777494	16777216
16777495	16777216
16777496	16777216
16777497	16777216
16777498	16777216
16777499	16777216
16777500	16777216
16777501	16777216
16777502	16777216
16777503	16777216
16777504	16777216
16777505	16777216
16777506	16777216
16777507	16777216
16777508	16777216
16777509	16777216
16777510	16777216
16777511	16777216
16777512	16777216
16777513	16777216
16777514	16777216
16777515	16777216
16777516	16777216
16777517	16777216
16777518	16777216
16777519	16777216
16777520	16777216
16777521	16777216
16777522	16777216
16777523	16777216
16777524	16777216
16777525	16777216
16777526	16777216
16777527	16777216
16777528	16777216
16777529	16777216
16777530	16777216
16777531	16777216
16777532	16777216
16777533	16777216
16777534	16777216
16777535	16777216
16777536	16777216
16777537	16777216
16777538	16777216
16777539	16777216
16777540	16777216
16777541	16777216
16777542	16777216
16777543	16777216
16777544	16777216
16777545	16777216
16777546	16777216
16777547	16777216
16777548	16777216
16777549	16777216
16777550	16777216
16777551	16777216
16777552	16777216
16777553	16777216
16777554	16777216
16777555	16777216
16777556	16777216
16777557	16777216
16777558	16777216
16777559	16777216
16777560	16777216
16777561	16777216
16777562	16777216
16777563	16777216
16777564	16777216
16777565	16777216
16777566	16777216
16777567	16777216
167775	

**Number** **Float32**

16777217 16777216

16777218 16777218

16777219 16777220

16777220 16777220

16777221 16777220

16777222 16777222

...

...

66777310 66777312

66777311 66777312

66777312 66777312

66777313 66777312

66777314 66777312

66777315 66777316

As numbers increase, precision degrades further. Eventually, multiple consecutive values collapse into the same `float32` representation, making it impossible to distinguish between them.

The `math` package offers the `Nextafter32` function, which helps visualize this precision loss by finding the next closest representable value:

```
math.Nextafter32(66777310, float32(math.Inf(+1))) // 66777316
math.Nextafter32(66777312, float32(math.Inf(+1))) // 66777316
```

Given a starting value and a target, `Nextafter32` returns the **smallest representable** `float32` that moves closer to the target—in this case, `+Inf`.

When floating-point numbers grow too large in magnitude, they overflow to `+Inf` or `-Inf` depending on the sign. Conversely, values whose magnitude is too small to represent as normal numbers underflow toward zero (possibly via subnormal values), reaching signed zero ( $\pm 0$ ). Here's how to create a `+Inf` without using the division-by-zero trick:

```
max := float32(math.MaxFloat32) // 3.4028235e+38
max + 1                         // 3.4028235e+38
max + max                        // +Inf
```

Adding `1` to the largest possible `float32` doesn't change the value. At that scale, the difference is too small to be represented, similar to tossing a grain of sand onto a mountain. Only when the added value is large enough to shift the magnitude, as with `max + max`, does the result overflow into `+Inf`.

### 3. Boolean Type: 1 Byte Memory Representation

Go has a boolean type, `bool`, and it's as straightforward as it gets—it can be either `true` or `false`:

```
var a bool = true   // true
var b bool          // false (default zero value)
c := !a            // false
```

Working with booleans in Go feels familiar if you've used other languages. They control flow in `if`, `for`, and `switch` statements and are used in logical operations like `&&` (and), `||` (or), and `!` (not).

In practice, you'll often see expressions like this:

```
if a && !b || c {  
    ...  
}
```

One thing that sets Go apart from some other languages is its strict handling of boolean values. You can't convert a `bool` to an integer, and there are no shortcuts or implicit conversions.

```
var a bool = true  
var b int = int(a) // Error: cannot convert a (type bool) to  
type int  
  
if b { // Error: non-boolean condition in if statement  
    // ...  
}
```

There was some discussion about allowing conversions from `bool` to `int`, but the proposal was rejected. If you want to dig into the details, check out the proposal "`#6428 spec: support int(bool) conversions`" [\[bint\]](#).

Now, a `bool` takes up 1 byte (8 bits) of memory. While this might seem excessive —after all, a boolean only needs 1 bit to represent `true` or `false`. Every variable has a memory address, and on most modern systems, the smallest addressable unit is 1 byte. You can't directly access a single bit in memory. Specialized systems might handle this differently, but for practical purposes in Go, that's not something we need to worry about.

## 4. Pointer Types: Memory Addresses, Dereferencing, and Allocation

### 4.1 Understanding Pointers

Every variable in Go has a place in memory, and that place has an address. A pointer is a variable that stores this address instead of holding the value directly. While a pointer could technically store any address in memory, in practice, it's primarily used to reference the address of a variable.

Consider this example to see how pointers work in Go:

```
func main() {  
    var a int = 10  
    var b *int // Declaring a pointer to an int  
  
    b = &a      // Assigning b the address of a
```

```

        println(b) // Prints the memory address of a, e.g.,
0x14000050738
        println(*b) // Prints 10, the value stored at the
memory address b points to

        *b = 20      // Updates the value at that address (a
becomes 20)
        println(a) // Prints 20
        println(&b) // Prints b's own memory address, e.g.,
0x14000050748
}

```

This is what's happening step by step:

- **Declaring a pointer:** Use `*` before the type (`*int`, `*float64`, `*[]int`) to declare a variable that holds a memory address of that specific type.
- **Getting a variable's memory address:** Use `&` before a variable's name (`&a`) to retrieve its address in memory.
- **Dereferencing a pointer:** Use `*` before a pointer (`*b`) to access the value stored at the memory address it points to.
- **Updating the value at a pointer's target:** Assign a new value to `*b` to update the original variable (`a`).

The exact memory address, such as `0x14000050738`, will vary depending on your machine—there's no need to focus on the specific number. What matters is understanding that it represents a location in memory:

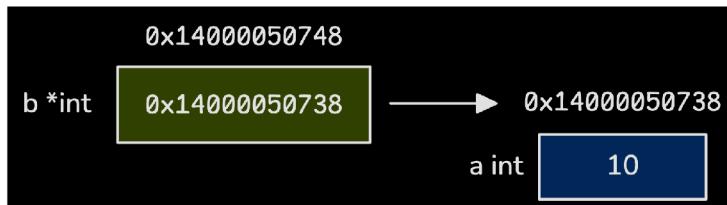


Illustration 2. `b` is a pointer that stores an address.

Since `b` is also a variable, you can retrieve its own memory address using `&b`. This might show up as something like `0x14000050748`, likely close to `a`'s address since they're stored near each other in the stack frame.

There's no guarantee of address proximity or stability.

While hexadecimal memory addresses might look strange, remember that they are just numbers. Go displays memory addresses in hexadecimal by default because it is more concise and easier to read for larger values. If you prefer to see the address in decimal, you can use `fmt.Printf("%d", b)`, which might output something

like `1374389864264`. This shows that a pointer, behind the scenes, is simply an address-sized number, while Go enforces reference semantics for how pointers are used.

We've already covered word size and memory addresses when discussing integers, but it's worth reinforcing: a pointer is represented as an unsigned integer (`uint`). On a 32-bit system, it occupies 4 bytes; on a 64-bit system, it occupies 8 bytes.

Another way to create a pointer in Go is by using the built-in `new()` function. The `new()` function allocates memory for a zero value of the specified type and returns a pointer to it. This is typically used for non-composite types like `int`, `float64`, or `bool`. In contrast, the `&` operator is more commonly used for composite types like structs (e.g., `&Book{}`), even when initializing them with zero values.

`new()` both allocates a zero value for the given type and provides a pointer pointing to it:

```
var a *int
var b *int = new(int)

a // nil
b // 0x14000110018
*b // 0
```

- `var a *int` : Declares `a` as a pointer to an `int` but doesn't assign it a value, so it defaults to `nil`.
- `var b *int = new(int)` : Allocates memory for an `int` initialized to `0` and returns a pointer to that memory.

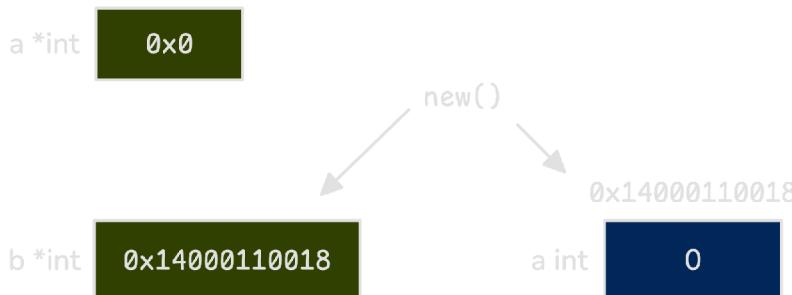


Illustration 3. Visualizing `new()` creating a pointer for `b`

It's a common misconception that `new()` always allocates memory on the heap just because it returns a pointer. While many functions that return pointers do involve heap allocation, Go's compiler is smart enough to optimize this.

If a pointer created with `new()` is only used within a function and doesn't "escape" to other parts of the program, the compiler may allocate it on the stack instead. This makes the operation faster and more efficient since stack memory is automatically cleaned up when the function ends.

## 4.2 Type Safety, Nil Pointers, and Dereference Panics

Pointers in Go are type-specific, meaning they're type-safe. When you have a pointer to a value of type `T`, it's written as `*T`—where `T` is the base type and `*T` is the pointer type.

You can't mix and match pointers of different types. If you try, Go will stop you at compile time:

```
var a int = 10
var b *string = &a // cannot use &a (type *int) as type
                     *string in assignment
```

A memory address is just a number under the hood. But Go enforces precision about what type of data your pointer references—it won't let you sidestep this rule.

By default, a pointer's zero value is `nil`, meaning it doesn't point to any valid memory location (often represented as `0x0`). If you're familiar with `null` from other languages, it works the same way in Go.

You might have heard Go developers yell about errors like "*nil pointer dereference*" or "*nil pointer panic*." This happens when you try to access the value stored at a `nil` pointer, which triggers a runtime error—and yes, it's a serious one:

```
func main() {
    var a *int

    println(*a) // panic: runtime error: invalid memory
                 address or nil pointer dereference
}
```

Since a `nil` pointer doesn't reference any valid memory, Go can't retrieve a value from it. Instead of letting the program continue and risk unpredictable behavior, Go immediately stops execution and triggers a panic to prevent further damage.

However, the message is not just about a '*nil pointer dereference*' panic; it also includes an '*invalid memory address*' panic. That means, besides a nil pointer dereference, there is another type of error that we haven't covered yet.

Go prevents pointer arithmetic, such as incrementing a pointer (`pointer = pointer + 1`). Allowing this could lead to pointers referencing invalid memory, resulting in unpredictable behavior or crashes. That said, if you truly need to bypass these safety checks, Go provides the `unsafe` package, which enables direct memory manipulation.

Pointers can also reference other pointers. Every variable, including a pointer, has a memory address. When you store the address of a pointer in another pointer, you create what's called a **double pointer**. This lets you reference the memory address of another pointer variable.

You can even take it further—a pointer to a pointer, a pointer to a pointer to a pointer, and so on:

```
func main() {
    var a int = 25
    var b *int = &a
    var c **int = &b

        println(&c) // 0x140000a4010: the memory address of
c itself
        println(c) // 0x140000a4018: the address stored in
c (which points to b)
        println(*c) // 0x1400009e018: the address stored in
b (which points to a)
        println(**c) // 25: the value stored at the memory
address of a
}
```

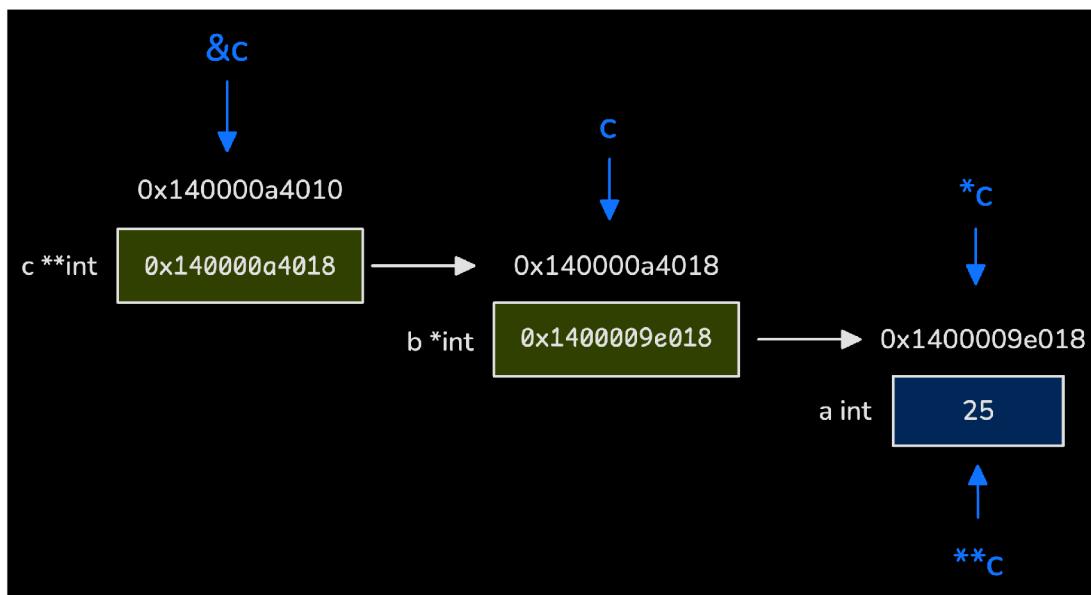


Illustration 4. How double pointers work

And that's all we need to know about pointers. There will be a bonus section dedicated entirely to `unsafe.Pointer` and its companion, `uintptr`.

## 5. Variables & Constants: Declarations, Scoping, and Compiler Behavior

This section gets more interesting because the compiler does more than just set aside memory and run calculations. We'll look at snippets of compiler code, but the focus will stay on the core ideas without overcomplicating things.

You've already seen how to declare and initialize variables in Go and explored some of the basic types. Now, let's take a closer look at what's happening—both in your code and behind the scenes in the compiler.

### 5.1 Variable Declarations, Scope, and Initialization

In Go, variables are typically declared using the `var` keyword. You can also assign an initial value right away:

```
var a int
var h = 2
```

Thanks to type inference, you don't always need to specify the type explicitly. In this example, Go automatically recognizes that `h` is an `int` based on the assigned value.

You can also declare multiple variables in a group. This avoids repeating `var` for each declaration:

```
var (
    b int
    c string
    d float64
)
```

Grouping related variables like this can keep your code organized. Just don't overuse it—only group variables that are logically connected.

For more clarity, you can explicitly declare and initialize variables by specifying both the type and value:

```
var e int = 1
var f, g int = 2, 3
```

Let's break down what happens during this process. A variable declaration involves three main components: the **variable name** (`obj`), the **type** (`typ`), and the **initialization expression** (`init`).

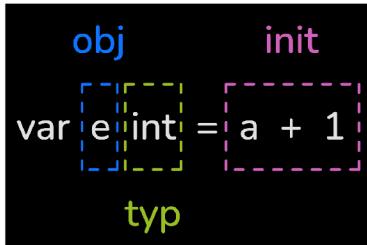


Illustration 5. obj, typ, and init in variable declaration

If both `typ` and `init` are missing, the variable's type (`obj.type`) is flagged as invalid, and the code won't compile. The Go compiler checks for this during variable declaration:

Figure 3. Compiler Checks for Invalid Type  
(src/cmd/compile/internal/types2/decl.go)

```
func (check *Checker) varDecl(obj *Var, lhs []*Var, typ, init
syntax.Expr) {
    assert(obj.typ == nil)

    // determine type, if any
    if typ != nil {
        obj.typ = check.varType(typ)
    }

    // check initialization
    if init == nil {
        if typ == nil {
            // error reported before by
arityMatch
            obj.typ = Typ[Invalid]
        }
        return
    }

    if lhs == nil || len(lhs) == 1 {
        assert(lhs == nil || lhs[0] == obj)
        var x operand
        check.expr(newTarget(obj.typ, obj.name), &x,
init)
        check.initVar(obj, &x, "variable
declaration")
        return
    }
}
```

```
}
```

...

This is our first look at Go's compiler code. Therefore, let's build some context around the compilation process.

Compilation happens in phases. The first phase, *parsing*, transforms your source code from plain text into a structured format called a *syntax tree*. Next comes *type checking*. During this phase, the compiler walks through the syntax tree, determines the types of variables, constants, and expressions, and ensures your code follows Go's type rules.

The function above is part of the type-checking phase. It validates variable declarations, such as `var a int = 1 + 2`.

So far, so good. Let's refocus on the first section of that logic:

```
// determine type, if any
if typ != nil {
    obj.typ = check.varType(typ)
}

// check initialization
if init == nil {
    if typ == nil {
        // error reported before by arityMatch
        obj.typ = Typ[Invalid]
    }
    return
}
```

If the type is explicitly provided, like in `var x int`, the compiler assigns that type to the variable.

When both `typ` and `init` are missing, Go has no type information to work with. The second `if` statement effectively prevents a scenario like:

```
var a
```

The compiler then flags the variable as having an *invalid type*, which results in a compilation error. There are two possible scenarios when declaring variables:

1. The variable has an explicitly declared type (e.g., `var a int`).
2. The type is inferred from the initialization expression (e.g., `var a = 1 + 2.2`).

```

if lhs == nil || len(lhs) == 1 {
    assert(lhs == nil || lhs[0] == obj)
    var x operand
    check.expr(newTarget(obj.typ, obj.name), &x, init)
    check.initVar(obj, &x, "variable declaration")
    return
}

```

*In most simple cases, `lhs` will be `nil`, so it can be ignored (unless you're dealing with multiple variable declarations or a function returning multiple values on the right-hand side).*

If the variable already has a type, the compiler checks whether the assigned value matches it (`check.expr`). If no type is specified, the compiler infers it from the `init` expression and assigns that type to the variable (`check.initVar`).

There's also a quicker way to declare and initialize variables in Go using a short assignment (`:=`):

```

func main() {
    g := 10
    h, i := 20, "example"
    h, k := 30, 40
}

```

Short assignment is simple and convenient. If a variable on the left-hand side doesn't already exist in the current scope, `:=` creates it. If the variable already exists, `:=` will reassign it. This is known as **re-declaration**, as long as at least one of the variables on the left side is new.

Keep in mind that short assignments can only be used inside functions. The reason is straightforward: *"Outside a function, every statement begins with a keyword (`var`, `func`, etc.), so the `:=` construct is not available."* This statement comes directly from *The Tour of Go* [[tour](#)].

From the outset, short assignments might seem similar to `var` declarations with inferred types, but there are important differences:

Aspect	Short Variable Declaration ( <code>:=</code> )	Variable Declaration with <code>var</code> (without type)
Where it can be used	Only inside function bodies	Allowed at package level and inside functions

Aspect	Short Variable Declaration (:=)	Variable Declaration with var (without type)
Type annotation	Not allowed (type is inferred from the right-hand side)	Allowed (you may specify a type or omit it)
Grouped declarations	Not supported	Supported with <code>var (...)</code> blocks
In the short statement before if/for/switch	Allowed (e.g., <code>if x := f(); x &gt; 0 {}</code> )	Not allowed (you cannot write <code>if var x = f(); {}</code> )
New-variable requirement	Must introduce at least one new non-blank name; otherwise error	Always declares new bindings; no “must be new” rule
Re-declaration in the same block	May reuse existing names only if at least one name on the left is new	Redeclaring an existing name in the same block is an error
Typical pitfall	Accidental shadowing of an outer variable	Can also shadow, but commonly used with explicit types where intent is clearer

It's easy to assume that `:=` allows re-declaring variables as long as one variable on the left side is new. But there's a catch: all items on the left must be plain variable names. If not, the compiler will throw a *bad declaration* error.

Consider the following examples, where this restriction comes into play:

```
person.Age, name := 20, "John" // non-name person.Age on left
side of :=
a[0], b := 1, 2                // non-name a[0] on left side
```

```
of :=  
m["key"], n := 1, 2           // non-name m["key"] on left  
side of :=  
*ptr, c := 1, 2               // non-name *ptr on left side  
of :=
```

In these cases, the expressions on the left aren't valid variable names. Instead, they belong to specific categories:

- `person.Age` : a selector expression (accessing a field within a struct).
- `a[0]` and `m["key"]` : index expressions (accessing elements in an array, slice, or map).
- `*ptr` : a pointer dereference operation (accessing the value stored at a memory address).

These expressions can't be used for variable declarations, even when combined with valid variable names.

## Variable Shadowing and Scope Resolution

Variable shadowing can catch both beginners and experienced developers off guard. It occurs when you declare a variable within a specific scope, such as inside a function or a block, and give it the same name as a variable defined in an outer scope. The new variable shadows the outer one, meaning that any reference to that name within the inner scope points to the newly declared variable, not the original.

This can lead to unexpected behavior if you're not careful. Take a look at how shadowing can create subtle bugs:

```
func main() {  
    var discount int  
    cartTotal := 500  
  
    if cartTotal > 100 {  
        discount, err := db.GetDiscount()  
        if err != nil {  
            panic(err)  
        }  
  
        recordDiscount(discount)  
    }  
  
    // ... process cartTotal and discount  
    fmt.Printf("You saved %d\n", discount)  
}
```

```
// Output: You saved 0
```

In `main()`, `discount` is declared without an initial value, so it defaults to `0`. Inside the `if` block, a **new** `discount` variable is accidentally created using `:=` instead of `=`. This inner `discount` **shadows** the outer one and exists only within the block.

When `recordDiscount(discount)` runs, it uses the inner `discount`. But as soon as the block ends, that inner variable disappears. The outer `discount` remains unchanged at `0`, leading to an output that doesn't reflect the intended behavior.

This mistake is surprisingly common. The `:=` operator is convenient, but that convenience can backfire. It's easy to unintentionally shadow an outer variable, especially when your focus is on solving the bigger problem rather than tracking variable scopes.

The Go compiler handles shadowing by keeping track of scopes. Every function, `if` block, `else` block, `switch` statement, or standalone code block defines its own scope, thanks to Go's use of curly braces:

```
if condition { // new scope
}
for condition { // new scope
}
{ // manual scope block
}
```

The compiler builds a hierarchical structure of scopes. Each scope maintains three main components: a parent scope (the outer scope), any child scopes (blocks nested inside), and a map that links variable names to their corresponding objects.

Figure 4. Scope Structure (src/compile/cmd/internal/types2/scope.go)

```
type Scope struct {
    parent      *Scope
    children   []*Scope
    elems      map[string]Object
    isFunc     bool
    ...
}
```

This illustration shows how the compiler maps variables to their scopes:

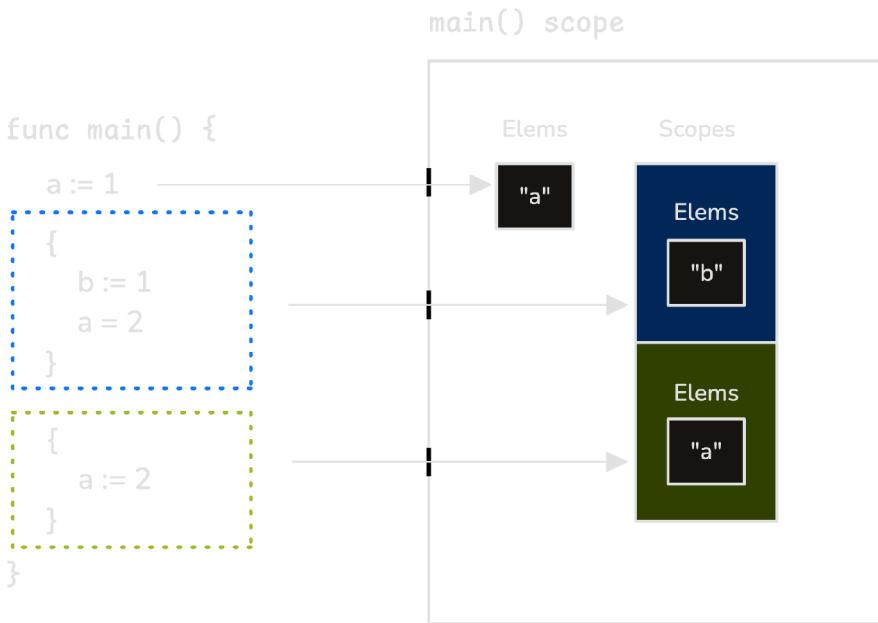


Illustration 6. Compiler mapping variables to their scopes

When the compiler looks up a variable, it starts with the current scope. If it doesn't find the variable there, it moves up to the parent scope and continues until it finds the variable or reaches the top level. This guarantees that the most recent declaration always takes priority within the current scope before looking outward.

If the compiler finds the variable, it returns both the scope where it was found and the variable itself:

Figure 5. Scope Lookup (src/compile/cmd/internal/types2/scope.go)

```
// Lookup returns the object in scope s with the given name
// if such an
// object exists; otherwise, the result is nil.
func (s *Scope) Lookup(name string) Object {
    return resolve(name, s.elems[name])
}

func (s *Scope) LookupParent(name string, pos syntax.Pos)
(*Scope, Object) {
    for ; s != nil; s = s.parent {
        if obj := s.Lookup(name); obj != nil && ... {
            return s, obj
        }
    }
    return nil, nil
}
```

```
}
```

This structure naturally leads to variable shadowing. When a new variable with an existing name is declared in the current scope, it takes priority over any outer declarations. The result is that the inner variable effectively hides—or "shadows"—the outer one.

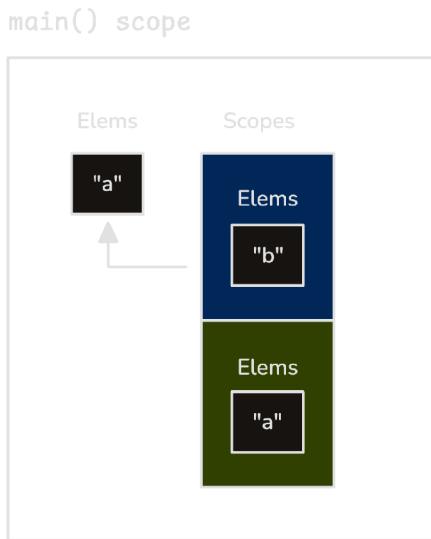


Illustration 7. Inner blocks override outer variable declarations

## Compiler Enforcement of Unused Variables

If you declare a variable and don't use it within that scope, the compiler will stop and throw an error. This rule doesn't just apply to variables—Go also flags unused imported packages and labels.

Unused variables by themselves aren't always a problem, but they can be a sign of something bigger, like a logical oversight or incomplete code:

```
var global int = 10

func (receiver AnyType) checkUnused(arg int) (named int) {
    var local int = 20 // Compiler error: local declared and
not used

    {
        var local int = 30 // Compiler error: local declared
and not used
    }
}
```

```
    return  
}
```

In this case, `global`, `arg`, `receiver`, and `named` are also unused, but the compiler only flags `local`. Why does this happen?

It comes down to when and where the compiler expects usage. Behind the scenes, each variable in Go is represented by a `Var` structure with a `used` field—a simple boolean that tracks whether the variable has been used:

Figure 6. Var Struct (src/compile/cmd/internal/types2/object.go)

```
type Var struct {  
    object  
    used     bool // Set if the variable was used or  
function arguments were accessed  
    isField   bool // Set if the variable is a struct  
field  
    embedded bool // Set if the variable is an embedded  
struct field, with name as the type name  
  
    ...  
}  
  
type PkgName struct {  
    object  
    imported *Package  
    used     bool // Set if the package was used  
}  
  
type Label struct {  
    object  
    used bool // Set if the label was used  
}
```

Go applies the same rule to imported packages and labels—they also can't go unused. Both `PkgName` and `Label` structures have a `used` field, just like `Var`.

As discussed earlier, the Go compiler tracks every scope and every object within that scope using a map that links names to objects. This system is also how the compiler identifies unused variables. It checks each variable in its scope—and any child scopes—by going through this list and inspecting the `used` field:

Figure 7. Checking Unused Variables  
(src/compile/cmd/internal/types2/stmt.go)

```

func (check *Checker) usage(scope *Scope) {
    ...

    // Collect all unused variables in the current scope
    for name, elem := range scope.elems {
        elem = resolve(name, elem)
        if v, _ := elem.(*Var); v != nil && !v.used {
            unused = append(unused, v)
        }
    }
    ...

    // Throw an error for each unused variable, but still
    continue checking
    for _, v := range unused {
        check.softErrorf(v.pos, UnusedVar, "%s
declared and not used", v.name)
    }

    // Recursively check child scopes, but skip function
    scopes
    for _, scope := range scope.children {
        if !scope.isFunc {
            check.usage(scope)
        }
    }
}

```

So, Go compiler finds unused variables, flags them with an error, and then moves on to check child scopes. Function scopes are skipped here because, when a function is defined, its body is handled separately. A different method, `funcBody`, takes care of checking variable usage within a function's scope.

Not every unused variable triggers an error. The compiler doesn't flag global variables or function parameters during this check.

- All variables—whether global or local—start with their `used` field set to `false`. However, since this unused check only applies within function scopes, global variables are ignored.
- Function parameters are automatically marked as `used`. Even if they aren't explicitly referenced in the function body, they may be required to satisfy an interface.

This is how Go creates variables with `NewVar` (for regular variables) and `NewParam` (for function arguments):

Figure 8. NewVar and NewParam  
(src/compile/cmd/internal/types2/object.go)

```
func NewVar(...) *Var {
    return &Var{object: object{...}}
}

func NewParam(...) *Var {
    return &Var{object: object{...}, used: true} // Parameters are always 'used'
}
```

Function receivers and named return values are also created using `NewParam( . )`, which automatically marks them as "used":

```
func (receiver AnyType) doSomething(arg int) (named int) {
    return 0
}
```

In this example, the compiler doesn't flag `receiver`, `arg`, or `named` as unused, even though none of them are referenced in the function body, because they're automatically considered used.

However, a variable that is assigned a value but not used later in the code is still flagged as unused:

```
func main() {
    var a int = 10 // a declared and not used
    var b int = 10

    a = b * 2
}
```

Even though `a` is assigned a new value (`b * 2`), it doesn't actually contribute to the program. The compiler expects a variable to be used in a meaningful way— affecting the program's behavior or state. Since `a` isn't used beyond the assignment, the compiler still considers it unused.

## 5.2 Constants: Fixed Values, Type Semantics, and Optimization

Constants are fixed values that cannot be changed once set. They make code clearer by replacing raw numbers or text with meaningful names. Using constants also centralizes values, so updating them in one place automatically applies the change everywhere:

## Not Bad

```
func calculatePrice(price
float64) float64 {
    // hard-coded
    value with no context
    return price + (price
* 0.07)
}
```

## Better

```
const taxRate = 0.07 // clear, named constant for the tax rate
```

```
func calculatePrice(price
float64) float64 {
    return price + (price
* taxRate)
}
```

When the code is compiled into binary, constants are placed in read-only memory segments, and the instructions reference them using pointers. However, this primarily applies to larger constants, like string literals.

For smaller constants, the compiler often optimizes performance by **embedding their values directly** into the instruction stream when possible:

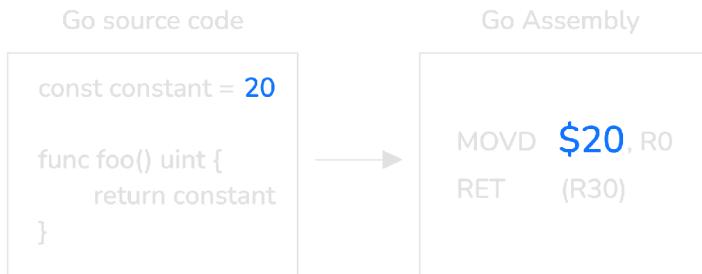


Illustration 8. Constants embedded directly into assembly instructions

In the `foo` function above, which simply returns the constant `20`, the compiler skips storing the value in memory. Instead, it embeds the constant directly into the instruction, making the function more efficient.

Setting up constants in Go is straightforward:

```
const a int = 10      // a is an int constant with value 10
const b = 20          // b is an untyped int constant with value 20
const c, d = 30, 40  // multiple constants in one line
```

Constants can only be defined using basic types: numbers (integers, floats), booleans, and strings. You can't declare constants for more complex structures like arrays, slices, maps, or structs. There are ongoing proposals [[imm](#)] to introduce

constant, immutable, or read-only versions of these types, but they're still under consideration.

The following excerpt from the compiler shows how Go defines its basic types:

Figure 9. Basic Types (src/compile/cmd/internal/types2/basic.go)

```
type BasicKind int

const (
    Invalid BasicKind = iota // type is invalid

    // predeclared types
    Bool
    Int
    Int8
    Int16
    Int32
    Int64
    Uint
    Uint8
    Uint16
    Uint32
    Uint64
    uintptr
    Float32
    Float64
    Complex64
    Complex128
    String
    UnsafePointer

    // types for untyped values
    UntypedBool
    UntypedInt
    UntypedRune
    UntypedFloat
    UntypedComplex
    UntypedString
    UntypedNil

    // aliases
    Byte = Uint8
    Rune = Int32
)
```

These types fall into three main categories:

- **Predeclared types:** The standard types Go recognizes by default (integers, floats, booleans, and strings).

- **Types for untyped values:** Values that don't have a specific type until they're used. For example, writing `10` instead of explicitly declaring `int(10)` uses an untyped integer.
- **Aliases:** Alternative names for existing types. For instance, `byte` is an alias for `uint8`, and `rune` is another name for `int32`.

This gives us a solid foundation for understanding constants in Go. But constants have more depth than it might seem from the outset. Just like variables, constants come in two forms: typed and untyped.

## Typed Constants and Compile-Time Evaluation

Typed constants behave like variables with a fixed type—such as `int32`, `int`, `float64`, `string`, or `bool`—and must follow the rules of their assigned type. For example, you can't assign an `int64` value to a constant typed as `int`. This is where they differ from untyped constants, which are more flexible.

When declaring a typed constant, you have two main options:

- Explicitly define the constant's type.
- Let the compiler infer the type from the assigned value.

```
// Explicitly specify the type
const a int = 10

// Let the compiler infer the type
const b = int(20)
```

Go uses a process called *constant folding* to evaluate constant expressions at compile time. This means the compiler resolves the value of expressions, such as `const c = a + b`, before the program even runs. This approach also helps catch errors early, ones that wouldn't appear with variables.

For example, the compiler will flag constants that overflow or get truncated—something that does not happen with variables:

Valid	Invalid
<pre>var c int8 = 127 var d int8 = c + 1</pre>	<pre>const a int8 = 127 const b int8 = a + 1</pre>

In the case of constants, the Go compiler immediately complains: "*a + 1 (constant 128 of type int8) overflows int8.*"

The overflow error only triggers with constants because their values are evaluated at compile time. Variables, on the other hand, are checked at runtime, allowing assignments that would otherwise fail during compilation.

## Untyped Constants and Type Inference Rules

In Go, numeric literals like `10` and `20.` are untyped constants by default. When you use these untyped constants in variable declarations without explicitly specifying a type, Go follows specific type inference rules:

```
var a = 10 // a is an int
var b = 20. // b is a float64
```

In these examples, the variables `a` and `b` receive specific types (`int` and `float64` respectively), but this doesn't mean the literals themselves become typed. The literals `10` and `20.` remain untyped constants.

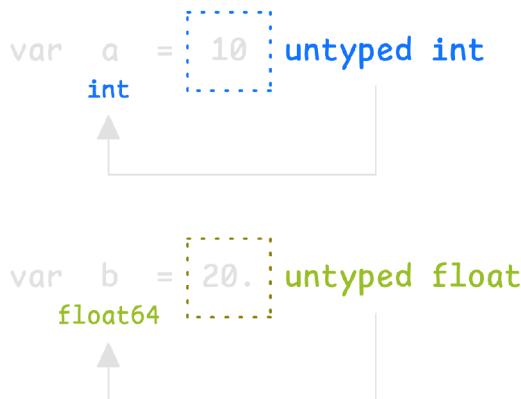


Illustration 9. Go infers types from untyped constants

This untyped nature gives Go flexibility, allowing you to use the same constant with various compatible types without explicit conversion:

```
var a int = 10
var b uint = 10
var c int8 = 10
```

Each variable has a different type, but all can use the same untyped constant `10`. This might seem like an obvious feature, but it's only possible because of how Go handles untyped constants.

Untyped constants interact with the type system by adapting to the context where they're used. There are clear rules around what values they can hold and how they can be converted to other types.

Before going deeper into these rules, let's look at how Go recognizes literal values as untyped constants:

Figure 10. Literal Values as Untyped Constants  
(src/compile/cmd/internal/types2/operand.go)

```
func (x *operand) setConst(k syntax.LitKind, lit string) {
    switch k {
    case syntax.IntLit:
        kind = UntypedInt
    case syntax.FloatLit:
        kind = UntypedFloat
    case syntax.ImagLit:
        kind = UntypedComplex
    case syntax.RuneLit:
        kind = UntypedRune
    case syntax.StringLit:
        kind = UntypedString
    default:
        unreachable()
    }

    ...
}
```

Inside Go's compiler, the `setConst` function identifies the type of literal and tags it as an untyped category. When you input values like `10`, `20.`, `1i`, `'A'`, or `"hello"`, Go automatically classifies them as untyped integers, floats, complex numbers, runes, or strings.

To understand why `var a = 10` results in `a` being treated as an `int`, we can look directly at how the compiler handles it. Instead of just relying on the *Go Language Specification* to explain default types for untyped constants, it's more engaging to see how the compiler applies these defaults:

Figure 11. Go Compiler Default Types  
(src/cmd/compile/internal/types2/predicates.go)

```
// Default returns the default "typed" type for an "untyped"
// type;
// it returns the incoming type for all other types. The
// default type
// for untyped nil is untyped nil.
```

```

func Default(t Type) Type {
    // Alias and named types cannot denote untyped types
    // so there's no need to call Unalias or under,
below.
    if t, _ := t.(*Basic); t != nil {
        switch t.kind {
        case UntypedBool:
            return Typ[Bool]
        case UntypedInt:
            return Typ[Int]
        case UntypedRune:
            return universeRune // use 'rune'
name
        case UntypedFloat:
            return Typ[Float64]
        case UntypedComplex:
            return Typ[Complex128]
        case UntypedString:
            return Typ[String]
        }
    }
    return t
}

```

The `Default` function assigns the appropriate type to an untyped constant by mapping it to its corresponding typed version. When you write `var a = 10`, the compiler uses this function to assign `a` the type `int`:

```

var a = 10      // a is an int
var b = 20.    // b is a float64
var c = 'A'    // c is a rune
var d = "hello" // d is a string
var e = 1 + 2i // e is a complex128
var f = true   // f is a bool

```

The story of untyped constants doesn't stop at default types. While they have defaults, these types aren't locked in—they can still adapt based on how and where they're used.

For example, even though the default type for an untyped integer is `int` (with a type of `UntypedInt`), it can still be assigned to other types like `float32`, `float64`, and various integer sizes. All the following assignments are valid:

```

const a = 10 // a is UntypedInt

var _ int8 = a
var _ int16 = a
var _ int32 = a
var _ int64 = a

```

```
var _ uint = a
var _ uint8 = a
var _ uint16 = a
var _ uint32 = a
var _ uint64 = a
var _ uintptr = a
var _ float32 = a
var _ float64 = a
```

This works because the value `10` comfortably fits within the range of both integer and floating-point types. But what happens when dealing with a literal like `10.`, which is an `UntypedFloat`?

In this case, whether the assignment is valid depends on the actual value of the floating-point literal. Go determines if the value can safely convert to an integer type without losing precision:

```
const a = 10. // a is UntypedFloat
var _ int = a // valid

const b = 10.1
var _ int = b // invalid: cannot use b (untyped float
constant 10.1) as int value in variable declaration
(truncated)
```

The first assignment works even though `a` is an `UntypedFloat` because the value `10.0` can be exactly represented as an integer. The second assignment fails because `10.1` can't be converted to an integer without losing data.

The error refers to **truncation**—converting `b` from a float to an integer would cut off the decimal portion, losing precision. Go doesn't allow this for constants. Just like with overflow checks, the compiler ensures that constant values stay within the valid range of their target type.

To handle this, the compiler uses a function that verifies whether a constant's value can be correctly represented by the target type when assigning a value, such as `var _ typ = x`, where `x` is a constant:

Figure 12. Representability of Constants  
(src/compile/cmd/internal/types2/const)

```
func representableConst(x constant.Value, ..., typ *Basic,
rounded *constant.Value) bool {
    ...

    switch {
        // The 'typ' is an integer type (int8, int16, int32,
```

```
int64, etc.)
    case isInteger(typ):
        ...

            // Try to get value as int64 first for
efficient checks
            if x, ok := constant.Int64Val(x); ok {
                switch typ.kind {
                    // For platform-dependent int,
calculate bit size from sizeof
                    case Int:
                        var s = uint(sizeof(typ)) * 8
                        // Check if value fits in s
                        return int64(-1)<<(s-1) <= x
&& x <= int64(1)<<(s-1)-1
                        // ... similar checks for other
integer types
                }
            ...
        }

    }

// The 'typ' is a floating-point type (float32,
float64)
    case isFloat(typ):
        ...

            switch typ.kind {
            case Float32:
                // Just check if value fits in
float32
                if rounded == nil {
                    return fitsFloat32(x)
                }

                // Try to round value to float32
precision
                r := roundFloat32(x)
                if r != nil {
                    *rounded = r // Store rounded
value
                    return true
                }
            // ... similar checks for other floating-
point types
        ...
    }
```

This function guarantees that a constant's value stays within the boundaries of its target type:

- **For integers:** The value must fall within the valid range of the type, preventing overflow or underflow.
- **For floats:** The value must stay within the precision limits of a `float32` or `float64`. If it doesn't, Go attempts to round the value. If rounding still leads to lost precision, the assignment is rejected—similar to the earlier example where assigning `10.1` to an `int` failed.

From the user's perspective, untyped numeric constants feel limitless—you can define massive constants without issue:

```
const a = (1<<64 - 1) * 5 // 5 times the max value of uint64
const b =
1234567890123456789012345678901234567890123456789012345678901
23456789012345678901234567890
```

Untyped numeric constants aren't bound by type size. They're limited not by the language itself but by the compiler's internal constraints. In practice, the Go compiler allows untyped integer constants to grow quite large, but there is a limit.

The following function shows how the compiler enforces these boundaries:

Figure 13. Overflows of Numeric Constants  
(src/compile/cmd/internal/types2/const.go)

```
func (check *Checker) overflow(x *operand, opPos syntax.Pos) {
    ...
    if isTyped(x.typ) {
        check.representable(x, under(x.typ).(*Basic))
        return
    }

    // Untyped integer values must not grow arbitrarily.
    const prec = 512 // 512-bit precision limit
    if x.val.Kind() == constant.Int &&
constant.BitLen(x.val) > prec {
        ...
        check.errorf(atPos(opPos), InvalidConstVal,
"constant %soverflow", op)
        ...
    }
}
```

The `overflow` function, part of the `Checker`, ensures that constants remain within valid size limits:

- **Typed constants:** The function checks whether the value fits within the constraints of its type.
- **Untyped constants:** The function ensures that the size of the constant doesn't exceed 512 bits.

*There's ongoing discussion about increasing this limit to 1100 bits, led by Robert Griesemer [[pbit](#)].*

In the compiler's world, constants are not truly limitless. Whether it happens directly in your code or during constant folding, exceeding 512 bits hits a hard boundary. The same applies to floating-point precision—if the value surpasses what the type can handle, Go rejects it:

```
// Invalid: number exceeds allowed size
const a = 1234567890... // (repeated 16 times)

// Invalid: too small for float64 to represent accurately
smallestFloat64 = 1.0 / (1<<(1023 - 1 + 52))
```

## iota: Implicit Sequencing and Declarative Patterns

`iota` is a built-in identifier in Go that simplifies creating number sequences, especially within groups of constants.

When a `const` block starts, `iota` begins at zero and automatically increments by one with each new constant in that block. Starting a new `const` block resets `iota` back to zero:

```
const (
    a = iota // 0
    b = iota // 1
    c = iota // 2
)

const (
    d = iota // 0
    e        // 1
    f        // 2
)
```

In the second block, there's no need to repeat `iota`. Go implicitly reuses the previous expression when no new value is assigned. This feature, known as

**implicit repetition**, isn't unique to `iota` but works seamlessly with it.

When you skip assigning a value to a constant, Go automatically reuses the last expression:

```
const (
    a = 1 // 1
    b    // 1 (implicit repetition)
    c = 2 // 2
    d    // 2 (implicit repetition)
)
```

`iota` isn't just for simple counting—you can use it in expressions to generate patterns:

```
const (
    a = 2 * iota // 0
    b             // 2
    c             // 4
)

const (
    e = 2*iota + 1 // 1
    f             // 3
    g             // 5
)
```

Internally, `iota` has a type and it's an untyped integer constant. At this point, this type should feel intuitive to you. Due to the nature of untyped constants, `iota` works seamlessly with different integer types and even floats:

```
const (
    a int8 = iota // 0
    b             // 1
    c             // 2
)

const (
    d float32 = iota // 0
    e             // 1
    f             // 2
)
```

There are some basic rules for using `iota`: it only works within `const` blocks. But what happens if you interrupt the sequence?

```
const (
    a = iota // 0
    b = 10   // 10
```

```
    c      // 10 (implicit repetition, not iota)
    d = iota // 3
)
```

In this case, assigning a specific value like `10` breaks the automatic `iota` sequence. When the sequence resumes, `iota` picks up where it left off, based on its position in the block.

Under the hood, the compiler handles `iota` through logic that tracks where each constant appears in a declaration block. The function below gives us a glimpse of how it works:

Figure 14. `collectObjects` (cmd/compile/internal/types2/resolver.go)

```
func (check *Checker) collectObjects() {
    ...
    // Iterate over files to populate package and file
scopes
    for fileNo, file := range check.files {
        ...

            for index, decl := range file.DeclList {
                if _, ok := decl.(*syntax.ConstDecl);
!ok {
                    first = -1 // we're not in a
constant declaration
                }

                switch s := decl.(type) {
                case *syntax.ImportDecl:
                ...
                case *syntax.ConstDecl:
                    ...
                    iota :=
constant.MakeInt64(int64(index - first))

                    // Determine which
initialization expressions to use
                    inherited := true
                    switch {
                    case s.Type != nil ||
s.Values != nil:
                        last = s
                        inherited = false
                    case last == nil:
                        last =
new(syntax.ConstDecl) // Ensure last exists
                        inherited = false
                    }
                }
}
}
```

```
// Declare all constants
values :=

syntax.UnpackListExpr(last.Values)
    for i, name := range
s.NameList {
    obj :=
NewConst(name.Pos(), pkg, name.Value, nil, iota)

    var init syntax.Expr
    if i < len(values) {
        init =
values[i]
    }

    d := &declInfo{file:
fileScope, vtyp: last.Type, init: init, inherited: inherited}

check.declarePkgObj(name, obj, d)
}

// Constants must always have
init values.

    check.arity(s.Pos(),
s.NameList, values, true, inherited)
    case *syntax.VarDecl:
    ...
    case *syntax.TypeDecl:
    ...
    case *syntax.FuncDecl:
    ...
    default:
        check.errorf(s,
InvalidSyntaxTree, "unknown syntax.Decl node %T", s)
    }
}
}
```

Overwhelming, isn't it? This part of the compiler processes global constant declarations during the type-checking phase. We will break down the code step by step.

First, the key part of how `iota` works lies in how it's assigned:

```
iota := constant.MakeInt64(int64(index - first))
```

`iota` can be used as a variable name. It's not a keyword in Go like `for`, `if`, or `range`. Instead, it's a special identifier, similar to the blank identifier (`_`).

The value of `iota` is determined purely by its position within a `const` block, starting from zero. It doesn't need to begin at the top of the block—it simply reflects the constant's order in the declaration:

```
const (
    a = 10 // 10
    b = iota // 1
    c = 20 // 20
    d = iota // 3
    e // 4
)
```

This example also highlights another important concept: **inheritance**. This is why `e` doesn't need an explicit value—it inherits the behavior of the previous constant. The Go compiler uses the following logic to determine when a constant should inherit:

```
// Determine which initialization expressions to use
inherited := true
switch {
case s.Type != nil || s.Values != nil:
    last = s
    inherited = false
case last == nil:
    last = new(syntax.ConstDecl) // Ensure last exists
    inherited = false
}
```

Inheritance applies when a constant has neither an explicit type (`s.Type != nil`) nor an explicit value (`s.Values != nil`). In that case, it automatically takes the value or behavior from the last explicitly defined constant in the same group. Here, `e` effectively becomes `e = iota`.

This feature allows Go to handle grouped constant declarations without requiring repeated values:

```
const (
    A = 10
    B // Inherits the value of A (10)
    C // Inherits the value of B (10)
    D = 11
    E // Inherits the value of D (11)
)
```

When a constant has an explicit value or type, like `D`, it becomes the new reference point. Any following constants without explicit values will inherit from it.

Go also applies inheritance when handling multiple constants on the same line:

```
const (
    a, b = iota, iota + 1 // 0, 1
    c, d           // 1, 2
)
```

In this case, `c` and `d` inherit the pattern from `a` and `b`, continuing the sequence based on their positions:

```
values := syntax.UnpackListExpr(last.Values)
for i, name := range s.NameList {
    obj := NewConst(name.Pos(), pkg, name.Value, nil, iota)

    var init syntax.Expr
    if i < len(values) {
        init = values[i]
    }

    d := &declInfo{file: fileScope, vtyp: last.Type, init:
init, inherited: inherited}
    check.declarePkgObj(name, obj, d)
}
```

The `values` are unpacked from the last constant declaration—in this case, `a, b = iota, iota + 1`. This means `c` and `d` are effectively treated as if they were written explicitly: `c, d = iota, iota + 1`.

## 6. Bonus: Unsafe Pointer

Both `unsafe.Pointer` and `uintptr` are concepts that many developers, even experienced ones, don't encounter often. While this section isn't essential, these types appear frequently in Go's source code, so they're worth understanding. In fact, they're built-in types within Go's type system:

Figure 15. Go's type system (src/internal/abi/type.go)

```
type Kind uint8

const (
    Invalid Kind = iota
    Bool
    Int
    Int8
    Int16
    Int32
```

```
    Int64
    Uint
    Uint8
    Uint16
    Uint32
    Uint64
    Uintptr
    Float32
    Float64
    Complex64
    Complex128
    Array
    Chan
    Func
    Interface
    Map
    Pointer
    Slice
    String
    Struct
    UnsafePointer
)
```

Before getting into what makes something 'unsafe,' it's important to understand what Go considers 'safe.' Regular pointers in Go, whether `*int`, `*string`, `*struct`, or others, are safe because of two core rules baked into Go's design. We actually mentioned both of them in the pointer section:

First, Go doesn't allow direct conversion between pointer types. If you have a pointer of type `*int64`, you can't simply convert it to a `*float64`. The compiler enforces type safety and will throw a compile-time error:

```
func main() {
    a := new(int)
    b := new(float64)

    // cannot convert a (variable of type *int) to type
    *float64
    b = (*float64)(a)

    // invalid operation: a == b (mismatched types *int and
    *float64)
    if a == b { ... }

    // cannot convert b (variable of type *float64) to type
    *int
    if (*int)(b) == a { ... }
}
```

You also can't compare pointers of different types using `==` or `!=`, nor can you assign a pointer of one type to another type directly.

At a glance, this might seem obvious—especially if you're familiar with Go's strict typing. But fundamentally, a pointer is just a number—an integer representing a memory address. So, why can't you assign it to a different type's memory address as you would with any other number?

The second rule is that Go doesn't allow arithmetic on pointers. You can't manually shift or adjust memory addresses like you might in languages such as C:

```
func main() {
    a := new(int)

    // invalid operation: a++ (non-numeric type *int)
    a++

    // invalid operation: a += 1 (mismatched types *int and
    // untyped int)
    a += 1
}
```

The `unsafe` package changes that. With `unsafe.Pointer`, you gain much more freedom—but that freedom comes with risk. Using `unsafe.Pointer`, you can convert between pointer types without any restrictions, bypassing Go's usual safety checks and breaking the first rule entirely.

## 6.1 Risks of Unsafe Pointers and `uintptr`-Based Manipulation

Let's look at an example where this freedom becomes risky. In this case, we bypass Go's type safety by assigning a pointer to a different pointer type:

```
import "unsafe"

func main() {
    a := new(int64)
    b := new(float64)

    // Convert a to unsafe.Pointer
    b = (*float64)(unsafe.Pointer(a))

    println("a", a, *a)
    println("b", b, *b)
}

// a 0x1400004e740 0
```

```
// b 0x1400004e740 +0.000000e+000
```

In this example, `a` is an `int64` pointer and `b` is a `float64` pointer, but both end up pointing to the same memory address. This means that any change made through one pointer can unintentionally affect the other, making bugs incredibly hard to track down.

When memory is shared this way, it opens the door to data corruption. While `int64` and `float64` both occupy 8 bytes, they store and interpret those bytes differently. To see this in action, let's set the `float64` value to `1` and observe what happens to the `int64`:

```
func main() {
    a := new(int64)
    b := new(float64)

    // Convert a to unsafe.Pointer
    b = (*float64)(unsafe.Pointer(a))

    *b = 1
    println("a", a, *a)
    println("b", b, *b)
}

// Output
// a 0x1400004e740 4607182418800017408
// b 0x1400004e740 +1.000000e+000
```

The result is unpredictable because the memory has been corrupted. Even though both types use the same 8 bytes in memory, they interpret those bytes differently. A `float64` stores `1.0` in IEEE 754 format, while an `int64` reads the same binary data as a completely different integer value:

```
00111111111000...000 = 1
                           float64

00111111111000...000 = 4607182418800017408
                           int64
```

Illustration 10. Data corruption risk with shared memory usage

By using `unsafe.Pointer`, we've already broken Go's first rule of pointer safety. To get around the second rule, which forbids pointer arithmetic, we need to convert a pointer to `uintptr`.

Go 1.17 introduced `unsafe.Add`, which helps you avoid converting a pointer to `uintptr`. This function is treated as a compiler intrinsic that performs byte-wise pointer arithmetic:

```
// The function Add adds len to ptr and returns the updated
pointer
// [Pointer](uintptr(ptr) + uintptr(len)).
// The len argument must be of integer type or an untyped
constant.
// A constant len argument must be representable by a value
of type int;
// if it is an untyped constant it is given type int.
// The rules for valid uses of Pointer still apply.
func Add(ptr Pointer, len IntegerType) Pointer
```

`uintptr` is an integer type that represents a memory address as a number. However, it lacks the properties of a pointer: it cannot be dereferenced, and it doesn't carry any memory management capabilities. Simply put, it's just a number. This conversion allows us to bypass the compiler's restrictions, enabling actions like accessing private (unexported) fields in a struct. Suppose we have a `User` struct in another package, and we want to access its unexported `password` field:

```
type User struct {
    Username string
    password string
}
```

Let's say we write a function, `exploitPassword()`, that retrieves the `password` field without directly accessing it:

```
func main() {
    u := User{
        Username: "func25",
        password: "123456",
    }

    println(&u, &u.Username, &u.password)
    println(exploitPassword(&u))
}

func exploitPassword(u *User) string {
    ... // this is where uintptr comes into play
}

// Output:
// 0x1400004e700 0x1400004e700 0x1400004e710
// 123456
```

To complete this function, we need to understand how structs are laid out in memory. When you define a struct, Go allocates memory for the entire struct as a single block, placing each field in sequence, with possible padding between them. This means the `Username` and `password` fields are stored next to each other in memory, with each field located at a fixed offset based on its order and size.

For example, if `u` starts at `0x1400004e700`, `Username` might be located right at `0x1400004e700`, while `password` would immediately follow at `0x1400004e710`. This matches exactly what we see in the output:

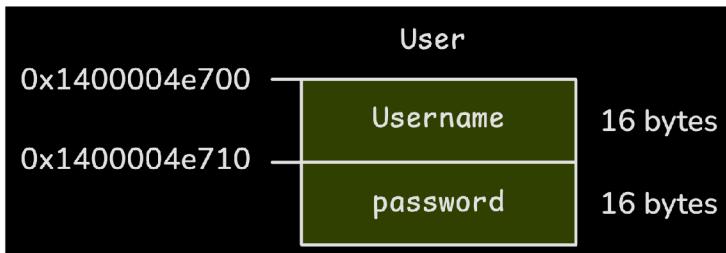


Illustration 11. Username and password fields aligned in memory

Under the hood, a string in Go is actually a struct with two fields: a pointer to underlying byte array, which points to the first byte of the string, and an `int`, which stores the string's length. This means a string takes up 8 bytes for the pointer and another 8 bytes for the length, totaling 16 bytes. That's why `u.password` sits 16 bytes away from the start of the `User` struct.

It's also important to note that the first field of a struct shares the same memory address as the struct itself. This makes it possible to exploit the memory layout with just a single line of code using the `unsafe` package:

```
type User struct {
    Username string
    password string
}

func main() {
    u := User{
        Username: "func25",
        password: "123456",
    }

    passwordUintptr := uintptr(unsafe.Pointer(&u)) + 16
    passwordPtr := (*string)
    (unsafe.Pointer(passwordUintptr))

    println("password:", *passwordPtr)
```

```

        println("-- debug --")
        println("u:", &u)
        println("u.Username:", &u.Username)
        println("u.password:", &u.password)
    }

// Output:
// 123456
// -- debug --
// u: 0x1400004e6e8
// u.Username: 0x1400004e6e8
// u.password: 0x1400004e6f8

```

In this example, we don't access `u.password` directly. Instead, we use the memory address of `u` to calculate where `password` should be. Since `password` sits 16 bytes from the start of the `User` struct, we can compute its address and access its value through `unsafe.Pointer`.

```
passwordUintptr := uintptr(unsafe.Pointer(&u)) + 16
```

The result would be the same if we used `&u.Username` instead of `&u` because the first field of a struct shares the same address as the struct itself.

Now, even though `uintptr` and `unsafe.Pointer` might seem similar, they behave differently in terms of memory management—especially concerning garbage collection (GC) and stack growth.

Despite its name, `unsafe.Pointer` is still recognized as a pointer by Go's type system. It follows specific safety rules, even when bypassing normal type checks. In contrast, `uintptr` is treated purely as an integer, nothing more. The Go runtime does not track `uintptr` values as memory references, meaning the garbage collector is unaware that a `uintptr` might still reference important data. This can lead to unexpected memory cleanup, as the runtime may incorrectly assume the data is no longer in use.

To see how risky this difference can be, let's look at an example that demonstrates what happens during stack growth:

```

func growStack() {
    x := [1024]byte{}
    _ = fmt.Sprint(x)
}

func main() {
    x := 1

```

```

        xUnsafePtr := unsafe.Pointer(&x)
        xUintptr := uintptr(xUnsafePtr)

        println("Before: xPtr", &x, "xUnsafePtr", xUnsafePtr,
"xUintptr", xUintptr)
        growStack()
        println("After: xPtr", &x, "xUnsafePtr", xUnsafePtr,
"xUintptr", xUintptr)
    }

// Output:
// Before: xPtr 0x1400007af38 xUnsafePtr 0x1400007af38
xUintptr 1374390038328
// After: xPtr 0x14000115f38 xUnsafePtr 0x14000115f38
xUintptr 1374390038328

```

## Stack growth in Go

Every goroutine has its own memory space, called a goroutine stack. This stack starts with an initial size, and when it nears capacity, the runtime automatically doubles it. For example, if your initial stack size is 4 KB and it needs to grow, the runtime increases it to 8 KB, then 16 KB, and so on.

During this resizing, Go allocates a larger stack and copies all the data from the old stack to the new one. It then updates all valid pointers to reflect the new memory location, except for `uintptr`, which the runtime does not recognize as a pointer.

If you examine the output, both the type-safe pointer (`&x`) and the `unsafe.Pointer` (`xUnsafePtr`) correctly reflect the new stack address after growth. However, the `uintptr` value (`xUintptr`) still points to the old memory location, which is now invalid. This creates a serious risk of using stale memory.

With great power comes great responsibility. So, why use `unsafe.Pointer` and `uintptr` at all? And, more importantly, how can they be used safely without risking memory corruption?

### 6.2 Using `unsafe` Safely: Guidelines and Best Practices

`unsafe.Pointer` can be a powerful tool for performance optimization in high-efficiency applications. One common use case is converting between `[]byte` and `string` without triggering additional memory allocations:

```

func String(b []byte) string {
    return unsafe.String(unsafe.SliceData(b), len(b))
}

```

```

func Bytes(s string) []byte {
    return unsafe.Slice(unsafe.StringData(s), len(s))
}

func StringSlicing(data []byte, start, end int) string {
    n := int(end - start)
    return unsafe.String((*byte)(unsafe.Pointer(uintptr(unsafe.Pointer(unsafe.SliceData(data))) + uintptr(start))), n)
}

```

These functions convert between a `[]byte` and a `string` without copying the data. To understand how this works, you need to know how slices and strings represent data in memory—something we’ll cover in the next chapter.

Normally, converting a `[]byte` to a `string` requires creating a new string and copying the byte data over. This ensures that strings remain immutable. However, using `unsafe` allows you to skip this copy. Instead of creating new memory, these functions simply reinterpret the existing data, making the operation faster but also riskier.

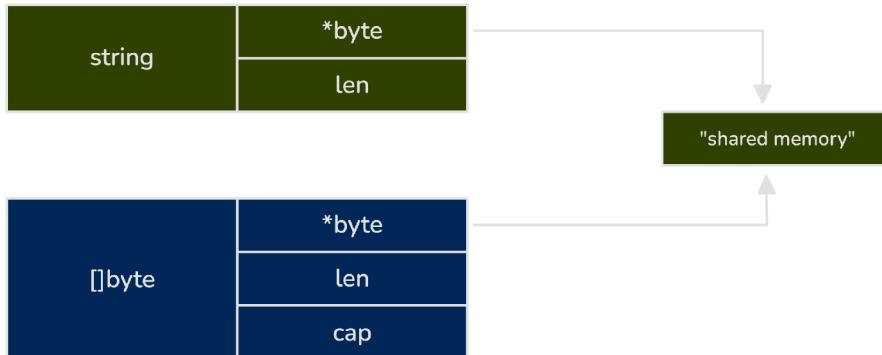


Illustration 12. Strings and slices sharing underlying byte arrays

When you use this method, the `[]byte` and the resulting `string` share the same underlying memory. This means that modifying the byte slice will also change the content of the string—a clear violation of Go’s immutability guarantee for strings.

In Go, strings are designed to be immutable; their underlying data should never change. However, `unsafe` bypasses this restriction, allowing direct manipulation of the data.

That said, to use `unsafe` correctly—and more importantly, safely—you need to follow clear guidelines. The Go team has outlined 5 essential rules for using unsafe pointers without risking memory corruption or unexpected behavior.

### a. Conversion of a `*T1` to Pointer to `*T2`

The first rule of unsafe pointer conversion allows you to take a pointer of one type, `*T1`, and reinterpret it as a pointer to another type, `*T2`. In practice, this means viewing the data that `*T1` points to as if it were `*T2`. However, this only works safely when `T1` and `T2` have compatible memory layouts, meaning the binary data of `T1` must make sense when read as `T2`.

For instance, both `float64` and `uint64` are 8 bytes in size, so converting between them is possible without violating memory boundaries:

```
func Float64bits(f float64) uint64 {
    return *(*uint64)(unsafe.Pointer(&f))
}
```

We've previously looked at this as an example of how risky it can be to convert between `float64` and `uint64`. While they share the same memory size, they interpret the data differently. A floating-point number and an integer don't have the same meaning at the binary level.

That said, this type of conversion doesn't corrupt memory or lead to invalid data access—it simply changes how the existing data is interpreted. As long as the memory layouts are compatible, this operation remains 'safe' within the context of `unsafe`.

### b. Conversion of a Pointer to a `uintptr` (but not back to a pointer)

The second rule allows you to convert a pointer into a `uintptr`. This conversion takes the memory address of a value and represents it as a plain integer.

However, this conversion is intended to be one-way—you can go from a pointer to a `uintptr`, but not directly back from a `uintptr` to a pointer. As we saw in the goroutine stack resizing example, `uintptr` lacks the "pointer semantics" that keep memory references valid during operations like garbage collection or stack growth:



Illustration 13. Converting pointers to `uintptr` in Go

Once converted to `uintptr`, the runtime no longer tracks that value as a valid memory reference. If the memory address changes, as it might during stack

resizing, your `uintptr` will still point to the old, invalid location. Converting it back to a pointer can lead to dangerous, unpredictable behavior.

However, just because it's risky doesn't mean it's entirely off-limits. This brings us to the third rule.

### c. Conversion of a Pointer to a `uintptr` and Back, with Arithmetic

Converting a pointer to `uintptr` allows you to perform arithmetic on memory addresses—adding, subtracting, and then converting it back to a pointer. We've already seen this in action when accessing unexported struct fields:

```
type User struct {
    Username string
    password string
}

func main() {
    u := User{
        Username: "func25",
        password: "123456",
    }

    passwordUintptr := uintptr(unsafe.Pointer(&u)) + 16
    passwordPtr := (*string)
    (unsafe.Pointer(passwordUintptr))

    println("password:", *passwordPtr)
}
```

In this example, we're actually breaking one of Go's rules for pointer arithmetic. These are the core rules to follow:

1. **Stay within bounds:** You can adjust a pointer's position as long as you stay within the allocated memory. This allows safe navigation through an object's memory, such as accessing struct fields or array elements.
2. **Convert in a single expression:** The conversion from a pointer to `uintptr` and back must happen within a single expression. Since `uintptr` is treated as a plain integer, not a memory reference, it can become invalid if stored separately before being converted back.
3. **No nil pointers:** You can only perform arithmetic on valid, non-nil pointers. Arithmetic on a `nil` pointer is invalid since `nil` doesn't refer to any actual memory.

In this case, we're breaking the second rule by storing the `uintptr` in a variable (`passwordUintptr`) before converting it back to a pointer.

Here's how valid and invalid usage differ:

Valid	Invalid
<pre>p = unsafe.Pointer(uintptr(p) + offset)</pre>	<pre>u := uintptr(p) p = unsafe.Pointer(u + offset)</pre>

To stay within Go's memory safety guarantees, both conversions—from pointer to `uintptr` and back—must occur in the same expression.

#### d. Conversion of a Pointer to a `uintptr` for System Calls

This rule is more specialized and typically applies in low-level programming scenarios, such as making system calls.

Certain system calls expect arguments as raw memory addresses, typically in `uintptr` form. In these cases, you must convert a pointer directly to `uintptr` within the function call itself. This ensures that Go's garbage collector recognizes the memory as "in use" for the duration of the call.

Valid	Invalid
<pre>syscall.Syscall(     SYS_READ,     uintptr(fd),     uintptr(unsafe.Pointer(p)),     uintptr(n), )</pre>	<pre>u := uintptr(unsafe.Pointer(p))  syscall.Syscall(SYS_READ,     uintptr(fd), u,     uintptr(n))</pre>

Go's compiler includes specific logic to recognize when a `uintptr` is passed directly in a `syscall.Syscall` call. When used this way, Go treats the pointer as a valid memory reference for the entire duration of the syscall, preventing premature garbage collection or memory invalidation.

#### e. Conversion of the Result of `reflect.Value.Pointer` or `reflect.Value.UnsafeAddr` from `uintptr` to Pointer

The fifth rule focuses on the `reflect` package's `Pointer` and `UnsafeAddr` methods, which return memory addresses as `uintptr`.

This design choice is intentional. By returning `uintptr` instead of `unsafe.Pointer`, the `reflect` package forces developers to explicitly use the `unsafe` package to convert the address into a usable pointer. To do this safely, the conversion must happen immediately—within the same expression where the `uintptr` is obtained.

This means that calling `reflect.ValueOf(...).Pointer()` must feed directly into `unsafe.Pointer(...)`, like this:

```
p := (*int)
(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer()))
```

To summarize, all these rules come down to three key principles:

- Always stay within the bounds of allocated memory when performing pointer arithmetic.
- Avoid storing `uintptr` values in variables.
- Keep conversions between `uintptr` and pointers within a single expression.

There's also a sixth rule related to converting the `Data` field of a `reflect.SliceHeader` or `reflect.StringHeader` to or from a pointer. However, these headers are now deprecated, and using them is discouraged. You can find more details in the Go documentation for the `unsafe` package [\[usf\]](#).

Earlier in this section, we looked at examples of optimizing code and avoiding unnecessary allocations using the `unsafe` package. Every one of those examples follows the rules we've covered so far.

## 7. Bonus: Addressable and Unaddressable Values

Not every value in Go has a memory address. *Addressable values* are those tied to a specific memory location—these are the ones we've worked with so far:

- Variables declared at the package or function level.
- Fields inside a struct (as long as the struct itself is addressable).
- Elements of an array that has an address.

- Elements in slices, whether the slice itself is addressable or not.

Take a look at this example to see how addressable values behave:

```
var global int = 10

func main() {
    var local int = 20
    var array = [3]int{1, 2, 3}
    var slice = []int{1, 2, 3}

    println(&global) // 0x100738388
    println(&local) // 0x1400004e6d0

    println(&array[0]) // 0x1400004e6d8
    println(&array[1]) // 0x1400004e6e0

    println(&slice[0]) // 0x1400004e6f0
    println(&slice[1]) // 0x1400004e6f8
}
```

In contrast, unaddressable values can't be referenced using the `&` operator. These values often appear when Go creates temporary results, such as the return value of a function, that aren't tied to a stable memory address.

Consider this example:

```
func returnArray() [3]int {
    return [3]int{1, 2, 3}
}

func returnStruct() Person {
    return Person{Name: "John", Age: 20}
}

func main() {
    // Error: cannot take the address of returnArray() (value
    // of type [3]int)
    array := &returnArray()

    // Error: cannot take the address of returnArray()[0]
    // (value of type int)
    arrayElement := &returnArray()[0]

    // Error: cannot take the address of returnStruct()
    // (value of type Person)
    person := &returnStruct()

    // Error: cannot take the address of returnStruct().Name
    // (value of type string)
```

```
    personName := &returnStruct().Name
}
```

The reason function return values are unaddressable comes down to how Go handles memory. This involves some deeper memory concepts, but don't worry if you're not familiar with them yet—we'll cover stack and heap memory in more detail later.

## Function Return Values Explained

A function's return value is unaddressable because it doesn't have a stable memory location you can reliably reference. While the value exists in registers or memory at the moment it's returned, that storage is temporary. Small return values are typically held in **registers**, while larger ones are written directly into the **caller's stack frame**.

Registers don't have memory addresses, making them inherently unaddressable.

For larger return values, the caller allocates space on the stack **before** the function call, reserving memory for the result. You can think of this pre-allocated space as the variable `receive` in the example below:

```
func callee() int {
    a := 10
    return a
}

func caller() {
    receive := callee()
}
```

In this example, `callee` writes its return value directly into the caller's pre-allocated stack space (`receive`). Inside `callee`, the result might temporarily exist in its own stack frame (`a`), but once the function returns, that stack frame is discarded. Assuming `callee` were to return the address of `a` to the `caller`, trying to use that address (`&a`) would be invalid because that memory no longer valid once `callee` finishes executing.

Other values can also be unaddressable, even outside of function returns:

```
func main() {
    const constant = 10

    // Error: cannot take address of constant (untyped)
```

```

int constant 10)
    constantAddress := &constant

        // Error: cannot take address of 10 (untyped int
constant)
    literalAddress := &10

        // Error: cannot take address of (10 + 20) (untyped
int constant 30)
    expressionAddress := &(10 + 20)

    aMap := map[int]string{1: "one"}
        // Error: cannot take address of aMap[1] (map index
expression of type string)
    mapElementAddress := &aMap[1]
}

```

Constants and literals are unaddressable, and the Go compiler enforces this rule, even though they may be stored in a read-only memory section. In some cases, constants and literals are embedded directly into the instruction stream, meaning they have no memory address at all.

Map elements are a different story. In Go, map elements are intentionally unaddressable to prevent accidental pointer errors. Since maps can reorganize their internal storage when elements are added or removed, taking the address of a map element could leave you with an invalid pointer. This design choice helps keep your code safe from unpredictable behavior caused by underlying map operations, a concept we touched on in Chapter 3.

Interestingly, elements in slices behave differently. They're always addressable, even if the slice itself isn't:

```

func returnSlice() []int {
    return []int{1, 2, 3}
}

sliceElement := &returnSlice()[0] // valid: 0x14000102000
slice := &returnSlice()          // Error: cannot take
address of returnSlice() (value of type []int)

```

A slice is essentially a wrapper around an array. While the slice itself might be unaddressable, it still points to an underlying array that has a valid memory address. This connection allows you to take the address of individual elements, even when the slice as a whole is temporary.

When we take the address of a slice element, we're actually pointing to a part of the underlying array. This array is already allocated in memory and forms the

foundation of the slice.

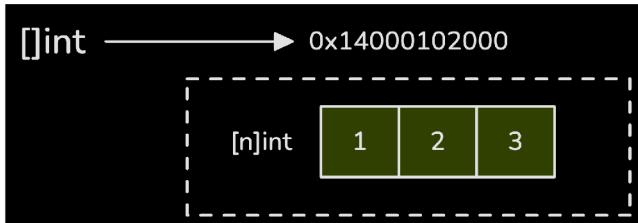


Illustration 14. Slice points to its underlying array

Now, what happens if the slice is inside an unaddressable struct? Let's see how Go handles this situation:

```
type Person struct {
    Name      string
    Age       int
    Friends   []string
    Array     [3]int
}

personFriends := &returnStruct().Friends           // Error:
cannot take address of returnStruct().Friends (value of type
[]string)
personArrayElement := &returnStruct().Array[0]        // Error:
cannot take address of returnStruct().Array[0] (value of type
int)
personFriendsElement := &returnStruct().Friends[0] // Valid
```

Since the struct field itself is unaddressable, trying to take the address of `person.Friends` fails. The same goes for `person.Array[0]`. But there's an exception—individual elements inside the slice, like `person.Friends[0]`, are addressable.

This happens because the `Person` struct and its fields are treated as a temporary block of data without a stable memory address. You can't take the address of a field within that temporary block. However, the slice `person.Friends` isn't stored inside the struct itself—it points to its own underlying array, which lives elsewhere in memory and does have a valid address.

This is similar to the earlier example with slices. The underlying array that `person.Friends` points to exists independently of the temporary struct, meaning you can still take the address of its elements.

## 8. Summary

Go offers a variety of integer types in different sizes, ranging from 8 to 64 bits, available in both signed and unsigned forms. The size of `int` and `uint` depends on the system architecture—32 bits on a 32-bit system and 64 bits on a 64-bit system. This design choice aligns with the CPU’s word size.

For floating-point numbers, Go follows the IEEE-754 standard with `float32` and `float64`. Unlike integers, dividing by zero doesn’t trigger a panic. Instead, Go returns special values like `+Inf`, `-Inf`, or `Nan`. Keep in mind that floating-point arithmetic can introduce small precision errors due to how binary systems represent decimal numbers. This is a common issue across most programming languages, and it’s something to be aware of when working with calculations that require high precision.

When dealing with pointers, you’re storing the memory address of a variable. Every pointer has a specific type, meaning it can only point to variables of that type. The zero value of a pointer is `nil`, and dereferencing a `nil` pointer will trigger a runtime panic. Go enforces memory safety by disallowing pointer arithmetic—unless you’re working with the `unsafe` package.

Variables are declared using either the `var` keyword or the shorthand `:=`. The compiler actively tracks variable usage, flagging any declared but unused variables within function scopes. This helps keep your code clean and efficient. Shadowing—when a new variable in an inner scope shares the name of one from an outer scope—can lead to subtle bugs if not handled carefully.

Constants are determined at compile time and are limited to basic types like numbers, strings, and booleans. They can be either typed or untyped. Untyped constants are more flexible and can adapt to different types, as long as the value fits within the bounds of the target type. However, Go imposes a precision limit of 512 bits for untyped integer constants—they can be large, but not unlimited.

Finally, Go provides `iota`, a convenient tool for generating related constant sequences. `iota` starts at 0 in each `const` block and increments by 1 for each new line of declarations; all identifiers on the same line share the same `iota` value. Because its value corresponds to the constant’s position in the block, `iota` is especially useful for creating enumerations.

## References

- [g1.13] Go 1.13 Release Notes: <https://go.dev/doc/go1.13>

- [cfq] Go FAQ, Why does Go not provide implicit numeric conversions?  
<https://go.dev/doc/faq#conversions>
- [ieee] IEEE 754: [https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754)
- [div] Unsized integer types: <https://groups.google.com/g/golang-nuts/c/IpzCvzWyYMM/m/WGk1Djgem3kJ>
- [bint] proposal: spec: support int(bool) conversions:  
<https://github.com/golang/go/issues/64825>.
- [df] Weekly Snapshot History: <https://go.dev/doc/devel/weekly#2011-01-20>
- [usf] The `unsafe` Package: <https://pkg.go.dev/unsafe>
- [tour] A Tour of Go, Short variable declarations: <https://go.dev/tour/basics/10>
- [pbit] cmd/compile, types2: consider increasing the internal precision limit for integer values: <https://github.com/golang/go/issues/44057>
- [imm] Go immutable value proposal list:  
<https://github.com/go101/go101/wiki/Go-immutable-value-proposal-list>

[OceanofPDF.com](http://OceanofPDF.com)

# Chapter 3: Arrays, Slices, Strings, and Maps

Let's take a step back and look at the bigger picture of what we've covered so far. We've gone through the basics—numbers, booleans, and even touched on a more advanced type: pointers.

The behavior of these basic types is tightly connected to how the compiler translates your code into assembly language. But now, we're entering territory where things get more abstract. Some types in Go don't reveal their structure as clearly, yet they play a significant role in how your code behaves. Understanding this can help make sense of many quirks you might encounter while working with Go.

At this point, it helps to think in terms of two layers: the "wrapper" and the "internal" structure—sometimes called a descriptor.

The wrapper is what you interact with directly in your code, like the exterior of a box. The internal part is what's hidden inside, managing the actual data and behavior behind the scenes.

Take `slice []int` as an example. The wrapper is the slice syntax you normally use (`[]int`), and its internal structure looks like this:

```
type Slice struct {
    array    unsafe.Pointer
    len, cap int
}
```

Internally, a slice is a structure that holds a pointer to an underlying array, along with two integers that track its length and capacity. This design, an external wrapper managing an internal structure, also applies to other core types like maps and strings, though we'll get into those details later.

To understand slices better, let's first examine their underlying structure: arrays.

# 1. Arrays: Fixed-Size Sequences and Memory Semantics

Arrays in Go are collections of elements that all share the same type. When you create an array, you define its size upfront, and that size is fixed—it can't be changed later.

In fact, the size of an array is also part of its type:

```
var a [5]int // a's type is [5]int
var b [3]int

a == b // invalid operation: a == b (mismatched types [5]int
and [3]int)
```

In this example, `[5]int` doesn't just mean 'an array of integers'—it means 'an array of exactly 5 integers.' The number 5 is baked into the type itself.

This leads to a couple of important details that might seem familiar but deserve a closer look:

- Since the size is part of the type, arrays of different sizes are considered entirely different types. That's why, in the example above, you can't compare or assign `a` and `b` directly—they're treated as distinct types by the compiler.
- When passing an array to a function, the size must be included in the parameter type. You can't pass a `[5]int` to a function expecting a `[3]int`.

From the compiler's perspective, an array can be represented like this:

```
type Array struct {
    len int64
    elem Type
}
```

This isn't the exact structure used by Go under the hood but rather a simplified representation of how the compiler interprets an array **during compilation** (not runtime). In fact, an array's elements are stored in a contiguous block of memory.

## 1.1 Understanding Array Memory Layout

Arrays are contiguous sequences of elements stored directly next to each other in memory. There are no gaps—each element follows the previous one in a straight line:

```
func main() {
    a := [5]byte{0, 1, 2, 3, 4}
    println("a", &a)
    println("a[0]", &a[0])
    println("a[1]", &a[1])
    println("a[2]", &a[2])
    println("a[3]", &a[3])
    println("a[4]", &a[4])
}

// Output:
// a      0x1400004e723
// a[0]   0x1400004e723
// a[1]   0x1400004e724
// a[2]   0x1400004e725
// a[3]   0x1400004e726
// a[4]   0x1400004e727
```

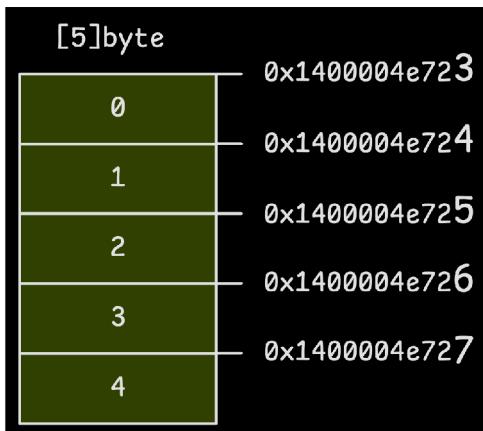


Illustration 15. Memory layout of a [5]byte array

The memory addresses of the array elements line up sequentially, each exactly 1 byte apart. The address of the entire array matches the address of its first element, `a[0]`. This behavior is similar to how memory is handled for structs in Go.

This layout has an important consequence: assigning one array to another or passing an array to a function results in copying the entire array—not just a reference.

When you assign an array, Go creates a shallow copy of the original array under the hood:

```

func main() {
    var a [3]int // [0 0 0]

    b := a
    b[0] = 10

    fmt.Println(a) // [0 0 0]
    fmt.Println(b) // [10 0 0]
}

```

In this example, changing `b` does not affect `a` because `b` is a copy of `a`, not a reference to it. Each element in `b` starts with the same value as the corresponding element in `a`. However, since `b` is a separate copy, any modifications to `b` leave `a` unchanged.

When an array is created, all elements are automatically initialized to their zero value, which depends on the element type.

## 1.2 Array Initialization

When memory is allocated for an array, let's say an integer array, it doesn't naturally come pre-filled with `0` values. It's also not as straightforward as simply moving the program counter to a new memory address or running a basic loop to set each element to zero. This leads to an important detail: *how the Go compiler handles zeroing out a block of memory*.

*This process matters not just for arrays but across many areas where memory needs to be initialized.*

Consider a typical example of manually zeroing an array in Go using a loop:

```

for i := range arr {
    arr[i] = 0
}

```

This loop is not efficient and not the best way to zero an array because it involves many things: a counter, a condition check, setting every element, etc. Instead, Go uses different strategies to zero memory depending on the size of the memory block:

- **Small memory blocks:** The compiler typically uses multiple direct instructions to zero out small memory blocks. For smaller arrays, this method can be faster than managing a loop.

- **Medium memory blocks:** In these cases, Go applies a technique called Duff's Device, using a function named `runtime.duffzero`. This clever loop unrolling strategy speeds up the zeroing process by reducing the overhead of loop management.
- **Large memory blocks:** For large arrays, Go defaults to using a straightforward loop as we saw previously to clear memory efficiently.

As we move into discussing Go assembly in the next section, the syntax might feel complex at first. You don't need to master assembly language to be an effective Go programmer—grasping the core ideas is what matters.

That said, I recommend reading through the examples, as they'll come up frequently throughout the discussion.

## Zeroing Small Memory Blocks (0-79 Bytes)

For small memory blocks, Go uses a simple method: it zeroes out memory chunk by chunk using a series of instructions. The larger the block, the more instructions are needed. For small blocks, this method is efficient and doesn't risk bloating the executable.

- When clearing an **8-byte block**, the compiler uses the `MOVD` (*Move Doubleword*) instruction. This instruction moves an 8-byte value—specifically, 8 bytes of zeroes—from the zero register to the target memory address in a single step.
- For blocks of **16 bytes**, it switches to `STP` (*Store Pair of Registers*), which writes two 8-byte chunks to consecutive memory addresses.

If Go assembly language feels unfamiliar, think of it like this: `MOVD` moves 8 bytes of data at a time, and when zeroing memory, it moves 8 zeroed bytes directly to the destination:

Go	Go Assembly
<code>var a [8]byte</code>	<code>MOVD ZR, main.a-72(SP)</code>

We will dive deeper into Go assembly as we go, and this is a good opportunity to start with simple examples. Let's break this down:

- `main.a-72(SP)` refers to the memory address of the array `a`. The `-72(SP)` notation indicates that `a` is located 72 bytes below the stack pointer (`SP`). This stack pointer points to the top of the current function's stack frame.
- `ZR` is the zero register, an 8-byte register that always holds the value zero.
- `MOVD` (Move Doubleword) moves 8 bytes of data from the zero register into the specified memory address. In this case, it writes 8 bytes of zeroes directly into the memory allocated for `a`.

To make this clearer, here's a simplified pseudo-code representation of what's happening in the assembly:

Go Assembly	Pseudo-Code
<code>MOVD ZR, main.a-72(SP)</code>	<code>ZR = [0 0 0 0 0 0 0 0] a = ZR</code>

The pseudo-code shows the core idea behind the assembly instruction: it moves 8 bytes of zeroes from the zero register directly into the array `a`, efficiently clearing that block of memory in one step.

Let's work through a few more examples to solidify this concept. On the right is the Go declaration, and on the left is the corresponding assembly code generated by the compiler:

Go	Go Assembly
<code>var b [16]byte</code>	<code>STP (ZR, ZR), main.b-88(SP)</code>

The `STP` instruction, short for **Store Pair of Registers**, zeroes out 16 bytes in a single operation by storing a pair of zero registers:



Illustration 16. Zeroing 16 bytes in a single instruction

For a 16-byte array `b`, one `STP` instruction is enough to clear all 16 bytes in a single step. Things get more interesting with a slightly larger array:

Go	Go Assembly
<pre>var c [17]byte</pre>	<pre>STP      (ZR, ZR), main.c- 105(SP) MOVD    ZR,     main.c- 96(SP)</pre>

With a 17-byte array, the compiler takes a two-step approach. First, it uses `STP` to zero out the first 16 bytes. Then, it follows up with a `MOVD` to handle the remaining byte.

However, this doesn't mean `MOVD` is used to clear just one byte. Instead, these two instructions end up overlapping in memory:

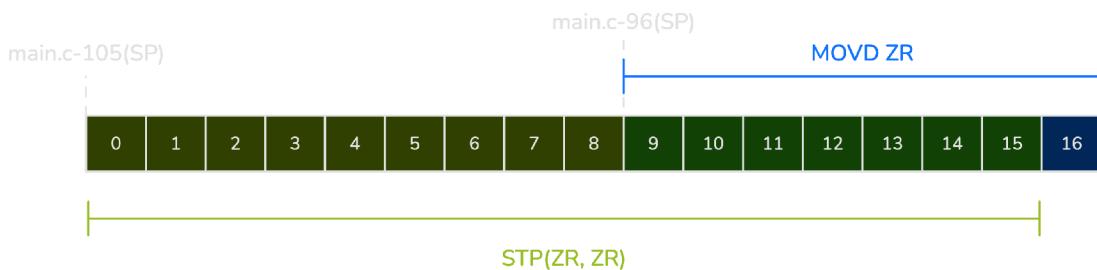


Illustration 17. Zeroing 17 bytes using STP and MOVD

After `STP` writes up to `SP-90`, `MOVD` starts at `SP-96` and writes up to `SP-89`. This means the first 7 bytes of the `MOVD` operation overwrite memory that `STP` has already cleared. While this might seem inefficient, it's actually faster than writing a special instruction just for that last byte.

As the array size increases, so does the number of instructions—but not without limits. Go changes its strategy based on the size category of the array. Let's move on to the next approach and see how Go handles medium-sized memory blocks.

## Zeroing Medium Memory Blocks (80-1039 Bytes)

For medium-sized memory blocks, those between 80 and 1039 bytes, Go uses an efficient technique called *Duff's Device*, combined with store instructions like `MOVD` or `STP` to clear memory.

This strategy is implemented in the `runtime.duffzero` function, which is essentially a long, unrolled sequence of instructions designed to write zeros directly into memory:

Figure 16. `runtime.duffzero` (src/runtime/duff\_arm64.s)

```
TEXT runtime·duffzero(SB), NOSPLIT|NOFRAME, $0-0
    STP.P  (ZR, ZR), 16(R20) // Store pair of zeros,
increment pointer
    STP.P  (ZR, ZR), 16(R20) // Same instruction repeated
many times...
    STP.P  (ZR, ZR), 16(R20)
    STP.P  (ZR, ZR), 16(R20)
    STP.P  (ZR, ZR), 16(R20)
    STP.P  (ZR, ZR), 16(R20)
    STP.P  (ZR, ZR), 16(R20)
// ... many more identical instructions ...
    STP     (ZR, ZR), (R20)
RET
```

This function clears memory in 16-byte chunks using repeated `STP` instructions. With 64 such instructions, the function can zero out up to 1024 bytes in a single pass.

What makes this technique efficient is how it's applied. Instead of running a loop that executes store instructions repeatedly, the compiler calculates how many store instructions are needed and **jumps directly to the right point** in the sequence.

For example, to clear 80 bytes, the compiler jumps to `runtime·duffzero + 236` in the assembly code:

```
TEXT runtime·duffzero(SB), NOSPLIT|NOFRAME, $0-0
    STP.P  (ZR, ZR), 16(R20)
    STP.P  (ZR, ZR), 16(R20)
    STP.P  (ZR, ZR), 16(R20)
    ...
236  STP.P  (ZR, ZR), 16(R20) ←————— 80 bytes
240  STP.P  (ZR, ZR), 16(R20)
244  STP.P  (ZR, ZR), 16(R20)
248  STP.P  (ZR, ZR), 16(R20)
252  STP     (ZR, ZR), (R20)
RET
```

Illustration 18. `runtime.duffzero`: Zeroing Out 80 Bytes

But how does the compiler know where to jump? such as 236?

Each instruction in the assembly is 4 bytes long, with 64 instructions in total. Since each clears 16 bytes, you can calculate the jump offset like this:

```
offset = 4 * (64 - size/16)
```

The real advantage of this method comes from eliminating loop overhead. A traditional loop requires checking a counter and branching back to the top on each iteration—operations that add unnecessary work and slow things down. Duff's Device skips this entirely. Instead, the CPU runs through a straight, uninterrupted sequence of instructions, speeding up execution and avoiding costly branch mispredictions.

Note that `runtime.duffzero` only processes memory blocks in multiples of 16 bytes since it exclusively uses the `STP` instruction. For any leftover bytes, the system relies on `MOVD` for handling 8 bytes or another `STP` for clearing up to 16 bytes:

Normal Loop	Unrolled Loop
<pre>var a [80]byte</pre>	<pre>MOVD    \$main.e-234(SP), R20 ADR     92, runtime.duffzero+236(R27)(R27) (REG) ... DUFFZERO runtime.duffzero(SB)</pre>

Let's make a rundown of what's happening here:

- `MOVD` moves the address of the array `a` into register `R20`. This address serves as the starting point for zeroing.
- `ADR` calculates the specific entry point within `runtime.duffzero` to jump to, storing that address in register `R27`.
- `DUFFZERO` triggers the `runtime.duffzero` function, kicking off the memory-clearing process.

The compiler generates code that jumps directly into the middle of `runtime.duffzero`, allowing it to execute just enough instructions to zero out the entire 80-byte block efficiently. The jump is calculated so that the function runs exactly the number of operations needed—no more, no less.

Now, consider what happens when the array size isn't a multiple of 16. For example, take a 95-byte array. The first 80 bytes are cleared using Duff's Device since 80 is a multiple of 16:

```
; Zeroing out the first 80 bytes
MOVD    $main.f-341(SP), R20
ADR     116, runtime.duffzero+236(R27)(R27)(REG)
STP     (R29, R27), -24(RSP)
SUB    $24, RSP, R29
DUFFZERO runtime.duffzero(SB)
SUB    $8, RSP, R29

; Zeroing out the remaining 15 bytes
STP    (ZR, ZR), main.f-266(SP)
```

To handle the remaining 15 bytes, the compiler uses an `STP` instruction. This efficiently zeroes out the leftover memory without requiring a separate loop or extra logic.

Why use this technique for memory blocks from 80 up to 1039 bytes?

Duff's Device works best for medium-sized memory blocks—those that are multiples of 16 bytes, larger than 64 bytes but no more than 1024 bytes. The unrolled loop is hard-coded with enough instructions to handle this range efficiently. By combining `runtime.duffzero` with `STP`, Go can handle blocks up to 1039 bytes.

## Zeroing Large Memory Blocks (1040 Bytes and Above)

For memory blocks larger than 1039 bytes, the strategy shifts. Instead of using Duff's Device, the compiler switches to a loop to clear memory.

In each loop iteration, it zeroes out 16 bytes at a time. If any bytes are left over—fewer than 16—the compiler uses either a `MOVD` or `STP` instruction to handle the remainder, just as it does for smaller blocks.

Two examples will help clarify how this works.

*Zeroing out 1,040 bytes (`var h [1040]byte`):*

## Go Assembly

```
160: MOVD    $main.h-
2443(SP), R16
164: MOVD    $main.h-
1419(SP), R2
168: STP.P   (ZR, ZR),
16(R16)
172: CMP     R2, R16
176: BLE     168
```

## Pseudo Code

```
R16 = address(main.h)
R2 = address(main.h +
1024)
while R16 <= R2
  *R16 = [0 ... 0] // 16 zeroes
  R16 += 16
```

Zeroing out 1,041 bytes (`var h [1041]byte`):

## Go Assembly

```
180: MOVD    $main.i-
3484(SP), R16
184: MOVD    $main.i-
2460(SP), R2
188: STP.P   (ZR, ZR),
16(R16)
192: CMP     R2, R16
196: BLE     188
200: MOVD    ZR, main.i-
2451(SP)
```

## Pseudo Code

```
R16 = address(main.i)
R2 = address(main.i +
1024)
while R16 <= R2
  *R16 = [0 ... 0] // 16 zeroes
  R16 += 16
[address(main.i + 1033)] =
[0 ... 0] // 8 zeroes
```

This pseudo-code simplifies what's happening in the assembly but captures the core logic behind the operation.

- `MOVD` loads the address of the array into register `R16` and the end address of the relevant memory block into register `R2`.
- `STP.P (Store Pair and Post-Index)` writes 16 bytes of zeros to the memory location pointed to by `R16`. After storing, it automatically increments `R16` by 16 bytes to point to the next memory block.
- `CMP` compares the current address in `R16` with the end address in `R2` to determine if the loop should continue.
- `BLE (Branch if Less or Equal)` sends execution back to the `STP.P` instruction, repeating the loop until all bytes are cleared.

If there are leftover bytes after zeroing in 16-byte chunks—as with the final byte in a 1041-byte array—the compiler uses a `MOVD` or `STP` instruction to clear the

remaining space.

This approach isn't limited to arrays. It applies to **any memory block**, including structs with many fields.

## 1.3 Working with Array Literals

Value literals in Go offer several ways to initialize arrays while setting their elements at the same time:

```
// Value-based initialization
a := [3]int{1, 2, 3}          // [1 2 3]

// Inferred size
b := [...]int{1, 2, 3}        // [1 2 3]

// Index-based initialization
c := [3]int{0: 1, 2: 3}       // [1, 0, 3]

// Mixed: Inferred size and index-based
d := [3]int{2: 2, 0: 1, 1}    // [1, 1, 2]

// Mixed: Inferred size and value-based
e := [...]int{0: 1, 2: 3}     // [1, 0, 3]

// Mixed: Inferred size, index-based and value-based
f := [...]int{0: 1, 2: 3, 1}  // [1, 0, 3, 1]
```

This example highlights the flexibility Go offers when creating arrays:

- The array `b` uses the `...` syntax, letting Go infer the array size based on the number of elements provided.
- In array `c`, index-based initialization allows you to assign values to specific positions. Here, position 0 holds a `1`, position 2 holds a `3`, and any unspecified positions, like index 1, are automatically set to `0`.
- The array `d` mixes methods, assigning values by index while using a non-sequential order.

The way Go initializes memory for array literals goes beyond simply zeroing out memory. Take a closer look at how a simple array like `a := [3]int{1, 2, 3}` is initialized:

## Go Assembly

```
STP    (ZR, ZR), main.a-  
40(SP)  
MOVD   ZR, main.a-24(SP)  
  
MOVD   $1, R2  
MOVD   R2, main.a-40(SP)  
MOVD   $2, R3  
MOVD   R3, main.a-32(SP)  
MOVD   $3, R3  
MOVD   R3, main.a-24(SP)
```

## Pseudo Code

```
a[0], a[1] = 0, 0  
a[2] = 0  
  
r2 = 1  
a[0] = r2  
r3 = 2  
a[1] = r3  
r3 = 3  
a[2] = r3
```

The compiler first zeroes out the memory, then sets each element's value individually.

For a small array like this, with just three integers, Go's compiler uses the strategy designed for small memory blocks (up to 79 bytes). It relies on `STP` and `MOVD` instructions, as shown in the assembly code above.

Earlier, we covered how Go uses different strategies for zeroing memory based on block size. Similarly, Go applies various strategies for setting array values. While we won't dive into every detail, we'll cover the essentials so the core concept is clear.

## 1.4 Small vs. Large Array Initialization Strategies

For arrays with **four or fewer elements**, Go treats them as 'small,' and the compiler generates a separate instruction for each element to set its value. This strategy is known as **local-code initialization** because the setup happens directly within the generated code, inside the function where the array is used.

```
arr := [3]byte{1, 2, 3}  
  
var arr [3]byte  
arr[0] = 1  
arr[1] = 2  
arr[2] = 3
```

When arrays have more than four elements, Go takes a different approach to make initialization more efficient. For example, with an array like `[5]byte{1, 2, 3, 4, 5}`, the compiler might set the first four elements all at once using a single value, `67305985`, and then assign `5` separately.

The number `67305985` might seem strange at first glance, but it's actually the result of packing four bytes into one integer. This allows Go to set multiple array elements simultaneously:

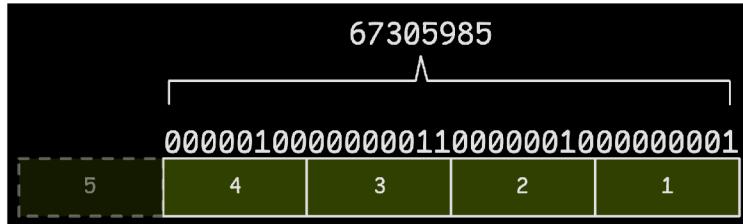


Illustration 19. Compact initialization of a [5]byte array using 67305985

To find this number, Go calculates this packed value using bitwise operations:

```
67305985 = 1 | (2 << 8) | (3 << 16) | (4 << 24)
```

For larger or more complex arrays, Go sometimes initializes values by copying from a **statically allocated temporary block**. This block is a read-only section of memory, pre-defined by the compiler and embedded directly into the binary.

If you switch the array type from `byte` to `int`, you'll see the compiler create a static, read-only memory block for `[5]int{1, 2, 3, 4, 5}` in the assembly output:

```
main..stmp_0 SRODATA static size=40
    0x0000 01 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00
00  .....
    0x0010 03 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
00  .....
    0x0020 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....
```

In this case, initializing the array becomes a matter of copying data from this predefined static block.

These rules don't just apply to arrays—they extend to any block of memory on the stack. For instance, if you create a struct with five integers, Go handles it just like it would an array:

```
type Struct struct {
    A int; B int; C int; D int; E int;
}
```

```
func main() {
    b := Struct{A: 1, B: 2, C: 3, D: 4, E: 5}
}
```

Even though this looks different from an array, under the hood, the compiler treats it the same way. Note that, if both this struct and the previous array appear together in your code, they may not share the same read-only memory block.

Large arrays are allocated on the heap, but this behavior isn't exclusive to arrays. Any large block of memory—whether it's a struct, an array, or another data type—can be moved to the heap if its size exceeds a certain threshold.

In the current version of Go, that threshold is around 10 MB:

Figure 17. MaxStackVarSize (cmd/compile/internal/ir/cfg.go)

```
var (
    // MaxStackVarSize is the maximum size variable
    // which we will allocate on the stack.
    // This limit is for explicit variable declarations
    // like "var x T" or "x := ...".
    // Note: the flag smallframes can update this value.
    MaxStackVarSize = int64(10 * 1024 * 1024)
)
```

Let's run a quick experiment to see how this limit affects memory allocation:

```
func main() {
    a := [10 * 1024 * 1024]byte{} // Allocated on the
    stack
    b := [10*1024*1024 + 1]byte{} // Exceeds the limit
    by 1 byte
}

$ go build -gcflags="-m" main.go
./main.go:5:2: moved to heap: b
```

Using the `-gcflags="-m"` command, we can confirm that `b` gets moved to the heap—exceeding the threshold by just one byte triggers this behavior.

There's another detail worth noting: You can directly take the address of array value literals, which isn't something you can do with other basic types like `int`, `string`, etc.

```
a := &[3]int{1, 2, 3} // tmp = [3]int{1, 2, 3}; a = &tmp
```

```
b := &1           // invalid
c := &"hello"    // invalid
```

This ability to directly reference array literals—and this extends to slices, maps, and structs—is a convenient bit of syntactic sugar offered by the Go compiler. It simplifies working with pointers to composite data structures that are initialized at declaration.

## 1.5 Array Length and Capacity

The length and capacity of an array are always the same and remain constant. This makes sense since an array’s length is fixed and defined at the time of declaration:

```
a := [5]int{1, 2, 3, 4, 5}

len(a) // 5
cap(a) // 5
```

Both `len(a)` and `cap(a)` are resolved as constant values at compile time, specifically as `int` constants. To see how the compiler treats these values without diving directly into the internals, consider this example:

```
a := [5]int{1, 2, 3, 4, 5}

// valid
const b = len(a)

// valid
var _ int = b

// invalid: cannot use b (constant 5 of type int) as int64
// value in variable declaration
var _ int64 = b
```

By assigning `len(a)` to the constant `b`, we can see that `len(a)` must be treated as a constant. It can be assigned to a variable of type `int` but not directly to `int64`—because `len(a)` is specifically of type `int`, not an untyped integer.

From the compiler’s perspective, both `len(array)` and `cap(array)` are treated as constants and embedded directly into the generated code.

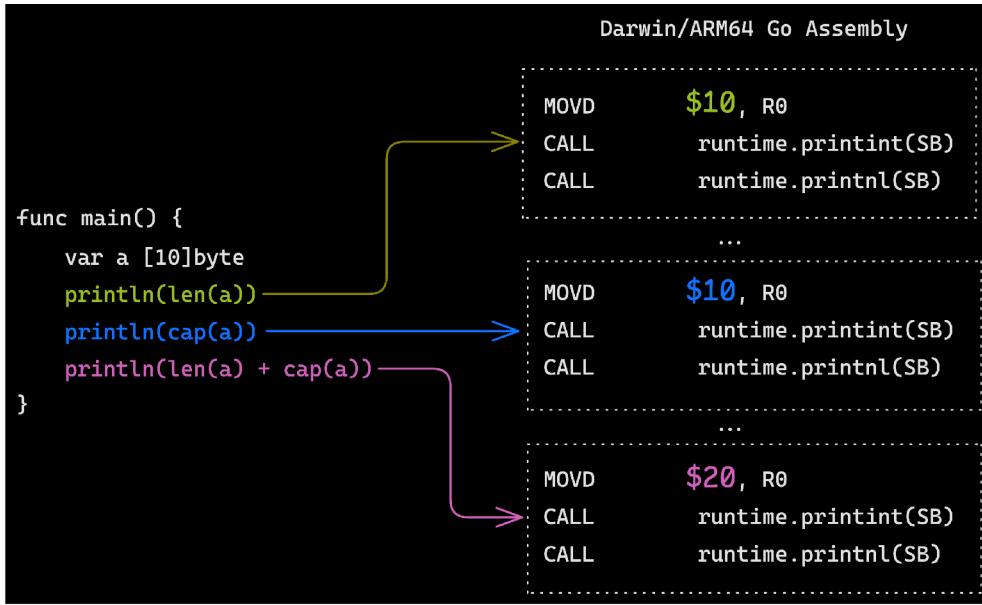


Illustration 20. Compiler embedding len and cap of an array as constants

When `len(a)` or `cap(a)` is used in expressions, Go applies a technique called 'constant folding'. This means the compiler directly writes the computed result into the generated code, as seen with 20 in the illustration above.

However, using `len()` on an array doesn't always return a constant value; particularly when the array comes from a function call:

```

func getArray() [5]int {
    return [5]int{1, 2, 3, 4, 5}
}

const length = len(getArray()) // invalid: len(getArray())
                             (value of type int) is not constant

```

Even though the length of the array is clearly 5, the compiler doesn't treat it as a constant. The reason comes down to how Go handles expressions internally:

```

func (check *Checker) builtin(x *operand, call
*syntax.CallExpr, id builtinId) (_ bool) {
    ...
    switch id {
    case _Cap, _Len:
        case *Array:
            ...
            if !check.hasCallOrRecv {
                mode = constant_
                if t.len >= 0 {

```

```
constant.MakeInt64(t.len)
}
} else {
    val =
constant.MakeUnknown()
}
}
...
}
...
}
```

The snippet above shows that if you use `len(..)` on the result of a function call or a receive operation from a channel (like `len(<-ch)`), the compiler won't treat it as a constant. Only when the length is clear and straightforward—with no function calls, channel receives, or other dynamic behavior—will the compiler resolve it as a constant value.

## 2. Slices: Structure, Semantics, and Behavior

Slices are more flexible than arrays and offer greater convenience from a programmer's perspective. Unlike arrays, slices can adjust their size dynamically by referencing portions of an underlying array.

```
func main() {
    s := []byte{0, 1, 2}

    // Append 3 and 4 to the slice
    s = append(s, 3, 4)
    fmt.Println(s)

    // Remove the last element from the slice
    s = s[:len(s)-1]
    fmt.Println(s)
}

// Output:
// [0 1 2 3 4]
// [0 1 2 3]
```

In this example, the slice is declared without a fixed length, unlike an array. We add elements using `append` and remove one with a simple slicing operation. While it might seem like the slice itself is resizing, that's not exactly what's happening. A slice doesn't hold elements directly—it just offers a flexible view of the underlying array.

When you create a slice, Go allocates an array beneath the surface and sets up the slice to reference that array. Think of a slice as a dynamic window into a portion of that array, adjusting how much of it you can see or work with:

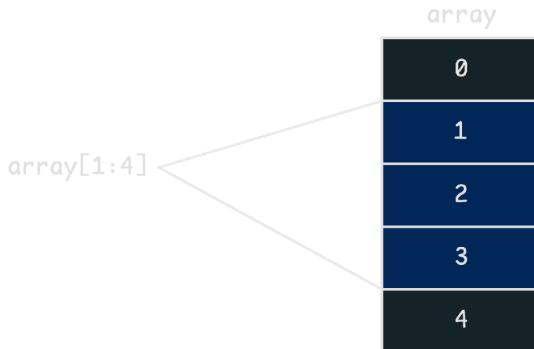


Illustration 21. Slice views part of the underlying array

This window can be resized using slicing operations. You'll often see this written as `[start:end]`, where `start` is the index of the first element and `end` marks the position **after** the last element you want to include.

Under the hood, Go manages slices using a structure defined in its runtime:

Figure 18. Slice Internal Structure (src/runtime/slice.go)

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

A slice consists of three components: a pointer to the start of the underlying array (`array`), the current length (`len`), and the capacity (`cap`)—the maximum number of elements the slice can hold before it needs to grow.

```
func main() {
    arr := []byte{0, 1, 2, 3, 4}
    slice := arr[2:4]
}
```

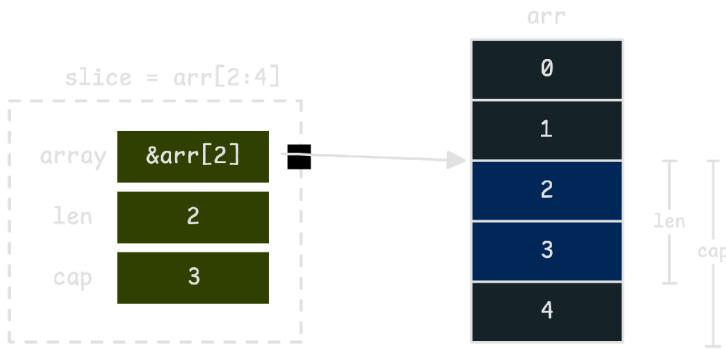


Illustration 22. Slice header with pointer, length, and capacity

At this point, it's clear that `slice.array` points directly to the memory address of `arr[2]` in the underlying array. There are several ways to confirm this, and we'll walk through three of them next.

Luckily, the first method can be done quickly. Although the slice struct is part of Go's internal implementation, you can print it directly without any tricks or unsafe code—just use `print` or `println`:

```
func main() {
    arr := [5]byte{0, 1, 2, 3, 4}
    slice := arr[2:4]

    println(slice)
    println(&arr[2])
}

// Output:
// [2/3]0x1400004e70d
// 0x1400004e70d
```

This output reveals important details about the slice in the format `[2/3]0x1400004e70d` (`[len/cap]array`). It shows how slices are structured under the hood. With this in mind, the diagram becomes even more intuitive:

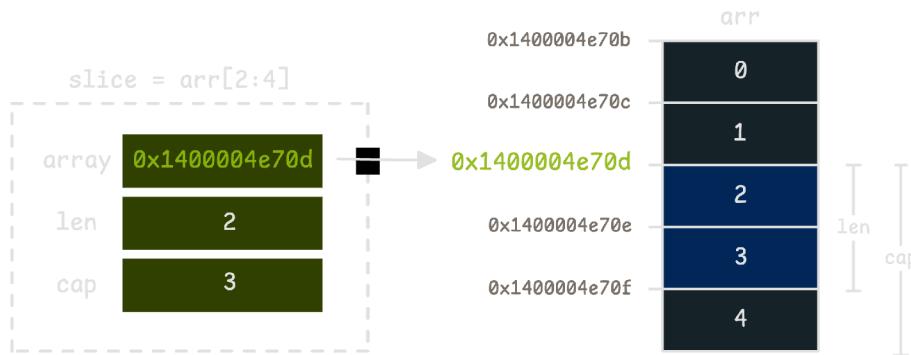


Illustration 23. Memory address tracks start

Unlike arrays, when you copy or pass a slice, you're only copying the slice header, not the entire underlying array. Regardless of how large the array is, the slice header remains the same size: 24 bytes on a 64-bit machine.

Here's how that plays out in practice:

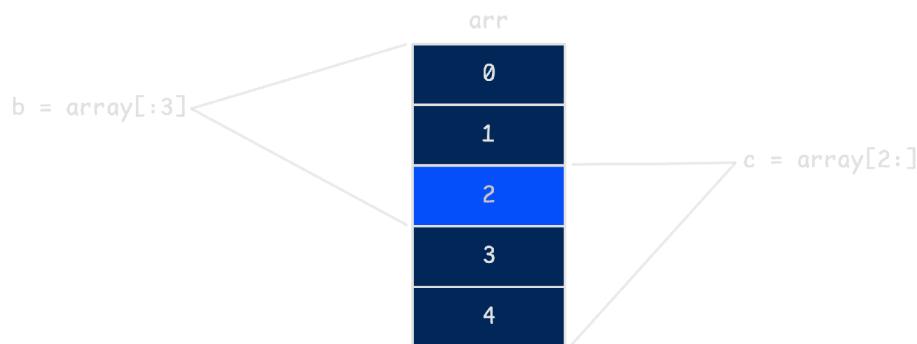
```
func main() {
    a := [5]byte{0, 1, 2, 3, 4}

    b := a[:3]
    c := a[2:]

    a[2] = 10

    fmt.Println("a:", a)
    fmt.Println("b:", b)
    fmt.Println("c:", c)
}

// Output:
// a: [0 1 10 3 4]
// b: [0 1 10]
// c: [10 3 4]
```



## Illustration 24. Slices share the same underlying array

As the diagram shows, both slices share the same underlying array and overlap at `a[2]`. Changing the value at `a[2]` affects both slices since they reference the same memory. This is why both `b` and `c` reflect the update in the output.

Moving on to a second approach. Earlier, we examined `unsafe` pointers and how to use them for pointer arithmetic, including accessing unexported struct fields. We can apply that same method here to inspect the slice's internal structure and see how it is laid out in memory:

```
func main() {
    mySlice := make([]byte, 3, 5)

    sliceArray := uintptr(unsafe.Pointer(&mySlice))
    sliceLength := unsafe.Pointer(sliceArray +
        uintptr(8))
    sliceCapacity := unsafe.Pointer(sliceArray +
        uintptr(16))

    println("array:", *(*unsafe.Pointer)
        (unsafe.Pointer(sliceArray)))
    println("length:", *(*int)(sliceLength))
    println("capacity:", *(*int)(sliceCapacity))

    println("mySlice:", mySlice)
}

// Output:
// array: 0x1400004e6fc
// length: 3
// capacity: 5
// mySlice: [3/5]0x1400004e6fc
```

The snippet above is purely educational. It bypasses Go's memory safety mechanisms and breaks the usual guidelines for using the `unsafe` package.

When running `make([]byte, 3, 5)`, Go creates an array with a capacity of 5 bytes and sets up a slice that initially references just the first 3 bytes. This gives you direct control over both the length and capacity of the slice while keeping the underlying array intact.

The `sliceArray` variable captures the address of the slice header, which also corresponds to the address of the slice header's first element, `slice.array`. Since `sliceArray` is a pointer that takes up 8 bytes, adding 8 bytes moves us to

`slice.len`, and adding 16 bytes reaches `slice.cap`. Visually, the memory layout looks like this:

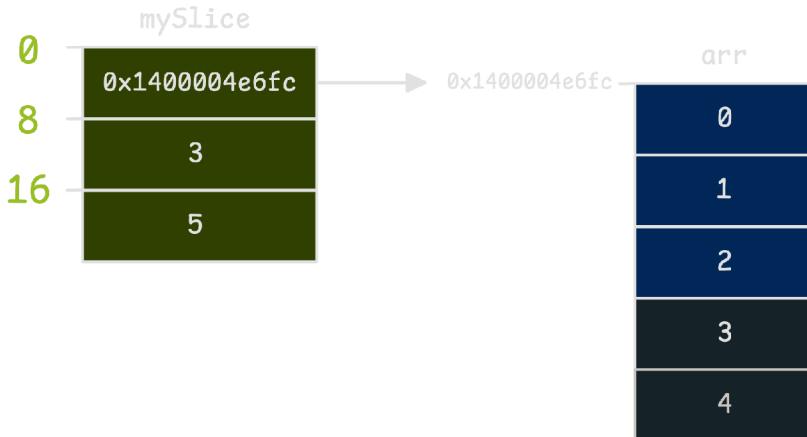


Illustration 25. Header offsets store slice info

An older, now-deprecated method for accessing the slice header involves using `reflect.SliceHeader`. This struct shares the same memory layout as the `slice` struct from the `runtime` package. You can define your own slice header struct with the same structure to inspect the slice's underlying data:

```
type sliceHeader struct {
    array unsafe.Pointer
    len   int
    cap   int
}

func main() {
    slice := make([]byte, 3, 5)
    println("slice", slice)

    header := (*sliceHeader)(unsafe.Pointer(&slice))
    println("sliceHeader:", header.array, header.len,
header.cap)
}

// Output:
// slice [3/5]0x1400004e71b
// sliceHeader: 0x1400004e71b 3 5
```

If you need to inspect the internals of any type, you can create your own struct with the same memory layout and use `unsafe.Pointer` to parse it. However, this method comes with risks. The internal layout of Go's types isn't guaranteed to

remain consistent across different versions, making this approach unreliable for consistent use.

## 2.1 Slice Expressions and Indexing Semantics

Slicing creates a new slice that contains a specific section of an existing slice or array. The full syntax is `a[low : high : cap]`, where `a` is the array or slice, `low` is the starting index, `high` is the ending index (excluded from the result), and `cap` sets the capacity limit of the new slice.

```
func main() {
    var a = [5]int{0, 1, 2, 3, 4}

    fmt.Println(a[1:4])    // [1, 2, 3]
    fmt.Println(a[1:4:5]) // [1, 2, 3]
    fmt.Println(a[1:4:4]) // [1, 2, 3]
}
```

In the examples so far, the slices created share the same capacity as their underlying arrays. A slice can grow only up to this full capacity. Once it exceeds that limit, a growth process is triggered, which we will cover in the next section.

Even though slices like `a[1:4]`, `a[1:4:5]`, and `a[1:4:4]` look identical when printed, their underlying capacities differ:

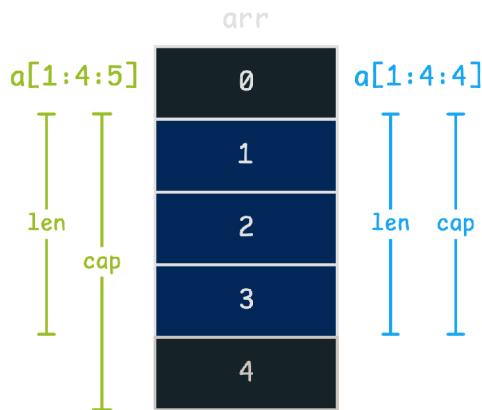


Illustration 26. Slices with different capacities

In this case, `a[1:4:4]` has already reached its maximum capacity (`len == cap`), meaning it can't be extended further using slicing. Trying to do so will result in a runtime error. The only way to extend it is with `append`—but that's tied to the concept of slice growth, which we'll tackle in the next section.

On the other hand, `a[1:4:5]` still has room to grow, allowing further extension through slicing:

```
func main() {
    var a = [5]int{0, 1, 2, 3, 4}

    b := a[1:4:5] // [1, 2, 3]
    c := a[1:4:4] // [1, 2, 3]

    fmt.Println(b[:4]) // [1, 2, 3, 4]

    // runtime error: slice bounds out of range [:4]
    with capacity 3
    fmt.Println(c[:4])
}
```

You can't directly access an element beyond a slice's length (`b[4]` would fail). However, when using a slice operation, you can extend the `high` value beyond the current length—as long as it stays within the slice's capacity.

We don't often write out the full form of a slice expression, but every slice operation is expanded into its complete form behind the scenes:

### Go Sugar Syntax

`b := a[1:4]`

### Underlying Syntax

`b := a[1:4:cap(a)]`

`b := a[:4]`

`b := a[0:4:cap(a)]`

`b := a[1:]`

`b := a[1:len(a):cap(a)]`

`b := a[:]`

`b := a[0:len(a):cap(a)]`

It's important to remember that the new slice `b` still shares the underlying array with the original slice `a`. This connection matters more than it might seem at first. Operations like `append(..)` can break this link, changing how the data is stored and potentially leading to unexpected behavior.

When it comes to how the compiler handles slicing, there isn't much complexity. Slices in Go have a clear structure under the hood, so the compiler's role is

simply to set up a new slice header with the right pointer, length, and capacity.

Here's how the compiler processes a slice operation like `a[i:j:k]`:

1. The compiler identifies the type of `a`—whether it's a slice, a string, or a pointer to an array.
2. If `i`, `j`, or `k` aren't specified, they default to `0`, `len(a)`, and `cap(a)`.
3. The compiler checks that the indices stay within valid bounds (`i <= j && j <= k`).
4. It calculates the new length as `j - i` and the new capacity as `k - i`.
5. A new base pointer is set for the slice, and a slice header is built with this pointer, the new length, and capacity. This header is then returned as the new slice.

Up to this point, we've covered how slicing works and how the compiler manages it internally. But there's more to capacity than just limiting how much of the underlying array you can access.

Capacity also plays a crucial role in how slices grow—and when that growth happens, it can silently break the connection between slices and their original arrays. This subtle behavior is really important to note down.

## 2.2 Append: Mechanics of Slice Growth

With `append()`, expanding a slice in Go requires no extra effort:

```
func main() {
    s := append(nil, 0)
    s = append(s, 1, 2, 3, 4)

    b := [3]int{5, 6, 7}
    s = append(s, b...)

    fmt.Println(s) // [0, 1, 2, 3, 4, 5, 6, 7]
}
```

`append()` doesn't just add elements to the end of a slice—it does more, especially when the slice runs out of capacity.

Since a slice's underlying array has a fixed size, it can't grow on its own. When there's no room left, `append()` automatically handles resizing without requiring any input from the developer. This automatic behavior is convenient but can also catch newcomers off guard:

```

func main() {
    s := make([]int, 0, 1)

    s = append(s, 0)
    println(s)

    s = append(s, 1)
    println(s)
}

// Output:
// [0/1]0x1400004e738
// [1/1]0x1400004e738
// [2/2]0x14000102000

```

In this example, we start by creating a slice `s` with a capacity for one element but no elements added yet.

When the first element is appended, the underlying array's memory address remains the same (`0x1400004e738`) since there's still enough room. But after appending a second element, the slice exceeds its initial capacity. Because arrays in Go are fixed in size, the runtime allocates a new, larger array (`[2/2]0x14000102000`) and copies the existing elements into it.

This is why the memory address changes after adding the second element.

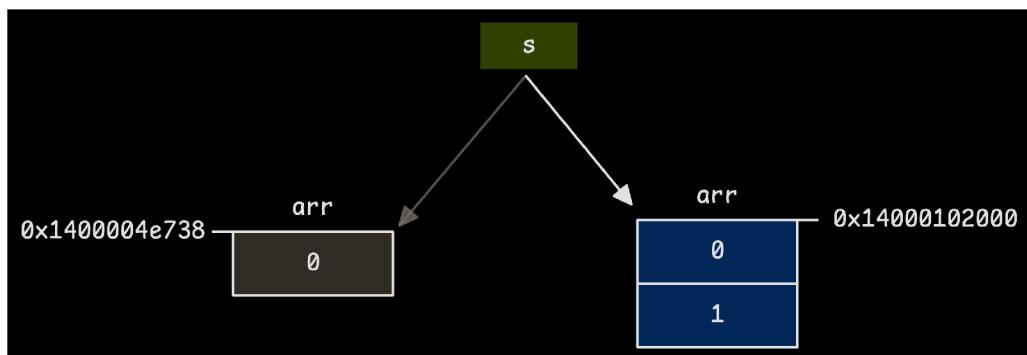


Illustration 27. Capacity exceeded, data is copied over

Once the capacity is exceeded, the original connection between the slice and its underlying array breaks. The slice now points to a new, larger array, and the old array becomes irrelevant to that slice.

This often leads to a common mistake: relying on the connection between slices, especially when passing slices to a function. That connection can break unexpectedly when the slice grows:

```
func main() {
    a := []int{1, 2}
    appendToSlice(a)

    fmt.Println(a)
}

func appendToSlice(s []int) {
    s = append(s, 3)
}

// Output:
// [1, 2]
```

It's safe to modify the elements of a slice directly, but avoid appending to it or replacing it with a new slice inside a function. If you need to modify the slice's structure, it's better to return the updated slice from the function. This approach keeps your code more predictable and easier to maintain, with minimal performance cost.

When you notice the slice's capacity doubling during appends, you're catching on to how Go manages slice growth. Let's run a simple experiment to see how the capacity changes as elements are added, starting from zero and continuing until the slice holds 2048 elements:

```
Initial capacity: 0
New capacity: 1
New capacity: 2
New capacity: 4
New capacity: 8
New capacity: 16
New capacity: 32
New capacity: 64
New capacity: 128
New capacity: 256
New capacity: 512
New capacity: 848
New capacity: 1280
New capacity: 1792
New capacity: 2560
```

At first glance, Go doubles the slice's capacity each time it reaches its limit. However, once the capacity goes beyond 512 elements, the growth rate starts to slow down.

The threshold is not 512; it is 256, and it will be explained further below.

This strategy reduces memory waste and limits how often resizing occurs. Resizing can be expensive because it involves allocating a new, larger array and copying over all existing elements.

If you have a good idea of how many elements you'll need, it's practical to preallocate the slice with enough capacity upfront. This reduces the need for resizing altogether:

```
a := make([]int, 1000)      // Preallocate with length and
                           capacity
b := make([]int, 0, 1000)    // Preallocate with zero length
                           but set capacity
c := make([]int, len(b))    // Create a slice with the same
                           length as b
```

If you preallocate a slice with a fixed capacity, like `1000` in the example above, Go will allocate the underlying array on the stack—provided the total size doesn't exceed 64KB. Once the slice grows beyond this limit, the new array is allocated on the heap.

Not Bad	Better
<pre>a := []int{} for i := range 1000 {     a = append(a, i) }</pre>	<pre>a := make([]int, 0, 1000) for i := range 1000 {     a = append(a, i) }</pre>

The way Go grows a slice isn't as simple as just doubling the capacity every time. To understand how it actually works, we need to break down two key points: when the capacity growth starts to slow down, and by how much the growth rate decreases as the slice gets larger.

When you compile your Go code, the compiler rewrites `append` calls behind the scenes. While `append(slice, elements...)` might seem like a straightforward function call, it actually triggers a more complex series of operations. The following pseudo-code gives a clearer picture of what's really happening under the hood:

```
func append(src []Type, elements ...Type) []Type {
    ...
    // Initialize the slice to append to
```

```

    s := src

    // Grow the slice if needed
    newLen := len(s) + len(elements)
    if newLen <= cap(s) {
        s = s[:newLen]
    } else {
        s = growSlice(s, newLen)
    }

    // Append new elements by copying them to the end of the
    slice
    copy(s[len(src):], elements)

    return s
}

```

## Determining the new capacity

Go checks whether the slice has enough capacity for the new elements by comparing the target length after appending (`newLen`) with the slice's current capacity. If there's enough space, the slice simply expands by updating its length, adds the new elements, and moves on.

If `newLen` exceeds the current capacity, Go triggers the growth process through `runtime.growslice`. At this point, it calculates the new capacity using the `runtime.nextsliceCapacity(newLen, oldCap int)` function, which handles the resizing logic:

Figure 19. `runtime.nextsliceCapacity` (`src/runtime/slice.go`)

```

func nextsliceCapacity(newLen, oldCap int) int {
    newcap := oldCap
    doublecap := newcap + newcap
    if newLen > doublecap {
        return newLen
    }

    const threshold = 256
    if oldCap < threshold {
        return doublecap
    }
    for {
        newcap += (newcap + 3*threshold) >> 2

        if uint(newcap) >= uint(newLen) {
            break
        }
    }
}

```

```
    }

    if newcap <= 0 {
        return newLen
    }
    return newcap
}
```

If `newLen` is more than twice the current capacity (`oldCap`), Go sets the new capacity to match `newLen`. Otherwise, the capacity initially doubles. But once the capacity reaches 256, the growth rate slows down, using this formula:

```
cap += (cap + 3 * 256) / 4
```

This adjustment provides additional space without growing as aggressively as before, based on the assumption that the larger the slice gets, the less immediate room it needs for future growth. At first glance, it might seem that the slice capacity simply doubled from 256 to 512 in our earlier experiment, but in reality, this jump follows a specific formula.

The process does not end there. The new capacity is then rounded up to match the sizes that the memory allocator is designed to handle. Go's memory allocator does not allocate arbitrary sizes; instead, it uses a predefined set of size classes.

For example, if you request 7 bytes, it rounds up and allocates 8 bytes instead. If you're curious before diving into *Chapter 7: Memory*, here's a quick breakdown of how this works:

- For **small allocations**, Go rounds up the memory size to fit within a specific size class. These predefined sizes are designed to be small and efficient—think blocks of 16 bytes, 32 bytes, and so on.
- For **large allocations** that do not fit within these small classes, Go allocates memory in full pages. In Go, a page is usually 8 KB, which is different from an OS page.

Different slice types grow at different rates. A slice of `int64` values will grow slower than a slice of `byte` because an `int64` is 8 times larger. The table below shows how various types scale as they grow:

**[]int64 (8 bytes) []int32 (4 bytes) []int8 (1 byte)**

0

0

0

**[]int64 (8 bytes) []int32 (4 bytes) []int8 (1 byte)**

1

2

2

4

4

8

8

8

16

16

16

32

32

32

64

64

64

128

128

128

256

256

256

512

512

512

848

864

896

`[]int64 (8 bytes) []int32 (4 bytes) []int8 (1 byte)`

1280

1344

1408

1792

2048

2048

2560

3072

3072

3072

When growing from zero, you won't see a slice with just one element for `[]int8` or `[]int32` because Go's memory allocator uses a minimum size class of 8 bytes. This means that instead of increasing by just one element, a `[]int8` slice will jump straight from 0 to 8 elements to align with the smallest allocation size. The same logic applies to `[]int32`, which grows to a capacity of 2 elements immediately.

Keep in mind that Go has a hard limit on memory allocation size, capped at  $2^{48}$  bytes (not elements) on most 64-bit architectures. If a slice exceeds this limit during growth, Go will trigger a panic. This restriction doesn't just apply to slices; it also affects other allocations, including channels, maps, and similar structures.

## Allocate and copy

At this stage, Go allocates a new underlying array with the expanded capacity by calling `mallocgc`, which triggers a heap allocation.

Once the memory is allocated, Go copies the elements from the old slice into the new array and returns the updated slice:

Figure 20. growslice (src/runtime/slice.go)

```
func growslice(oldPtr unsafe.Pointer, newLen, oldCap, num
int, et *_type) slice {
    ...
    newcap := nextslice(cap(newLen, oldCap))

    ...

    var p unsafe.Pointer
    if et.PtrBytes == 0 {
        p = mallocgc(capmem, nil, false)
        memclrNoHeapPointers(add(p, newlenmem),
        capmem-newlenmem)
    } else {
        p = mallocgc(capmem, et, true)
        ...
    }
    memmove(p, oldPtr, lenmem)

    return slice{p, newLen, newcap}
}
```

This is what happens under the hood when you call `append()` and it triggers a growth: a new array is allocated, existing elements are copied over, and the slice is returned with its new capacity.

## 2.3 Allocation Strategies and Memory Layout

When the compiler determines that a slice needs to escape to the heap—either due to its size or because it will outlive the function where it was created—it replaces the `make([]T)` call with a runtime call to `runtime.makeslice`. Whether you write `make([]T, len)` or `make([]T, len, cap)`, the allocation process works the same way behind the scenes.

In fact, `make([]T, len)` is automatically translated by the compiler into `make([]T, len, len)`. This ensures that both length and capacity are set equally unless specified otherwise:

Figure 21. makeslice (src/runtime/slice.go)

```
func makeslice(et *_type, len, cap int) unsafe.Pointer {
    mem, overflow := MulUintptr(et.Size_, uintptr(cap))
```

```

        if overflow || mem > maxAlloc || len < 0 || len >
cap {
            mem, overflow := MulUintptr(et.Size_,
uintptr(len))
            if overflow || mem > maxAlloc || len < 0 {
                panicmakeslicelen()
            }
            panicmakeslice(cap())
}

return mallocgc(mem, et, true)
}

```

If an overflow or required memory exceeds  $2^{48}$  bytes (or any other invalid conditions arise as the above checks), the function triggers a panic. Whether the issue is with `len` or `cap` determines the specific panic raised.

If all checks pass, memory allocation proceeds using `mallocgc`. This function takes the calculated memory size, the element type, and a flag that indicates *whether the allocated memory needs to be zeroed out*. And that's where things get interesting.

Normally, when a slice is created, the underlying array is zeroed to ensure safety and prevent data leakage. However, things change when you create a slice and immediately populate it by copying data from another array or slice:

```

func main() {
    a := [5]int{1, 2, 3, 4, 5}

    b := make([]int, 5)
    copy(b, a)
}

```

In this case, you might expect Go to create slice `b` with `makeslice`, zero out the memory, and then perform a `memmove` to copy values from `a`. However, there is a smarter optimization for pointer-free slice types, such as `[]int`. Since the slice will be immediately overwritten with values from `a`, zeroing out the memory first is unnecessary.

Go recognizes this pattern and skips zeroing altogether by using a specialized function: `runtime.makeslicecopy`. This function allocates memory without clearing it, using `mallocgc(mem, et, false)` to save time and resources:

Figure 22. makeslicecopy (src/runtime/slice.go)

```
func makeslicecopy(et *_type, tolen int, fromlen int, from
unsafe.Pointer) unsafe.Pointer {
    ...
    if !et.Pointers() {
        to = mallocgc(tomem, nil, false)
        if copymem < tomem {
            memclrNoHeapPointers(add(to,
copymem), tomem-copymem)
        }
    } else {
        to = mallocgc(tomem, et, true)
    }
    ...
    memmove(to, from, copymem)
    return to
}
```

If the slice's element type doesn't contain pointers, Go skips zeroing the memory altogether. Since non-pointer types don't pose a risk of being misinterpreted by the garbage collector, this optimization speeds up memory allocation without compromising safety.

However, if the destination slice is larger than the source, any extra bytes that aren't populated by `memmove` still need to be zeroed out using `memclrNoHeapPointers`. This prevents leaving behind uninitialized memory, which could lead to unexpected behavior.

For slices whose element type contains pointers (like `*int`, `string`, or structs with pointer fields), Go must zero out the memory during allocation.

Between allocation (`mallocgc`) and copying data (`memmove`), there's a brief window where the garbage collector can see the newly allocated memory. If this memory isn't zeroed out, the garbage collector might mistake random, uninitialized data for valid pointers, leading to unpredictable behavior.

This behavior applies specifically to slices allocated on the heap. When a slice is allocated on the stack, things are simpler—the memory is zeroed out first and then populated by `memmove` as usual.

For arrays, anything larger than 10 MB automatically gets allocated on the heap. Slices, however, follow a different rule. **When the size is determined at compile time**, their underlying array is capped at 64 KB before triggering heap allocation:

```
type A struct {
    slice [64 * 1024 + 1]byte
}

func main() {
    stackSlice := make([]byte, 64 * 1024)
    heapSlice := make([]byte, 64 * 1024 + 1)
    heapStruct := new(A)
    ...
}

$ go build -gcflags=-m
./main.go:4:20: make([]byte, 65536) does not escape
./main.go:5:19: make([]byte, 65537) escapes to heap
./main.go:10:19: new(A) escapes to heap
```

In this example, `stackSlice` stays on the stack because its size is exactly 64 KB. But exceeding that limit by even a single byte forces the allocation to the heap, as seen with `heapSlice`. This applies to more than just slices—the `A` struct also ends up on the heap due to its large size.

This behavior is controlled by a runtime constant called `MaxImplicitStackVarSize`, which defines the largest variable size allowed on the stack:

Figure 23. `MaxImplicitStackVarSize`  
(src/cmd/compile/internal/ir/cfg.go)

```
// MaxImplicitStackVarSize is the maximum size of implicit
variables allocated on the stack.
// p := new(T)           allocating T on the stack
// p := &T{}            allocating T on the stack
// s := make([]T, n)      allocating [n]T on the stack
// s := []byte("...")    allocating [n]byte on the stack
// Note: the smallframes flag can adjust this value.
MaxImplicitStackVarSize = int64(64 * 1024)
```

There's also a way to bypass this limit:

```
func main() {
    largeSlice := []byte{ 1024 * 1024 * 1024: 10 }
    ...
}
```

```
$ go build -gcflags=-m  
./main.go:8:22: []byte{...} does not escape
```

Even with 1 GB of data, a slice literal does not force the array onto the heap. However, running this code will cause either a runtime stack overflow error or, at compile time, a "stack frame too large" error for anything exceeding 1 GB. While these details rarely matter in everyday coding, they reveal just how many exceptions and edge cases the Go compiler handles behind the scenes.

## 3. Strings: Structure and Behavior

At its core, a string is a sequence of characters, but technically, it behaves more like a read-only slice of bytes.

While strings share some similarities with slices in how they reference data, there's a fundamental difference: strings are **immutable**. Once a string is created, its data cannot be changed through normal operations. This immutability is a key feature and shapes how strings function in Go.

To understand this better, let's look at the internal structure of a string in Go:

Figure 24. stringStruct (src/runtime/string.go)

```
type stringStruct struct {  
    str unsafe.Pointer  
    len int  
}
```

In this structure, the `str` field is a pointer to the first byte of the string's data in memory—it marks where the data begins, similar to how a `slice` works. The `len` field holds the length of the string in bytes. Unlike slices, strings don't have a capacity field in their header, which means they occupy just 16 bytes on a 64-bit system.

When creating strings, Go offers two types of literals:

- **Raw string literals:** Enclosed in backticks (`), these capture everything exactly as written, including spaces, tabs, and line breaks. What you see in the code is exactly what you get.
- **Interpreted string literals:** Enclosed in double quotes (""), these support escape sequences. An escape sequence starts with a backslash (\) and represents specific characters, such as \n for a newline or \t for a tab.

```
func main() {
    raw := `String is a
sequence of characters`

    interpreted := "String is a \nsequence of
characters"
}
```

The `raw` string preserves the formatting exactly as it appears in the code, including line breaks and indentation. The `interpreted` string uses escape sequences like `\n` to represent formatting characters, meaning it prints as a single line with a newline inserted where specified.

These two strings might seem similar, but they're not identical. To highlight the difference, you can use `fmt.Printf("%q", raw)` to reveal hidden formatting details like escape characters.

However, it becomes even clearer when looking at the Go assembly output:

```
MOVD    $go:string."String is a \n\tsequence of characters"
(SB), R0
...
MOVD    $go:string."String is a \nsequence of characters"
(SB), R0
```

The output shows that the `raw` string includes an actual newline and tab space before "sequence of characters." To match this formatting precisely, you would need to adjust the code:

```
func main() {
    raw := `String is a
sequence of characters`


    ...
}
```

The assembly output shown here isn't the final machine code—it represents an intermediate stage in Go's compilation process. It provides insight into how the compiler interprets strings before generating the final assembly instructions.

Now, when you assign a string to another variable or slice a string, do they share the same underlying byte array, or does Go duplicate the data?

```
func main() {
    a := "Hello, World!"
    b := a[:2]
```

```
        println(unsafe.StringData(a)) // 0x16f8fb4d8
        println(unsafe.StringData(b)) // 0x16f8fb4d8 (same)
    }
```

In this example, both `a` and `b` point to the same memory location. This shows that assigning a string to another variable—or slicing it—doesn’t duplicate the underlying data, regardless of the string’s size. Only the string header is copied.

This might seem risky since shared memory typically means changes to one variable could affect the other. However, with strings, this isn’t an issue because any operation that modifies a string **creates a new string**. Go enforces immutability by preventing direct changes to string data through standard operations.

An interesting detail is that you can’t use `&a[0]` to get the address of the first byte, as you would with a slice. Allowing that would expose the underlying bytes and open the door to unintended modifications, breaking immutability.

Instead, Go provides `unsafe.StringData`, which safely returns the address of the first byte as a `*byte`. The `unsafe` prefix is a clear signal that you’re bypassing Go’s usual safety checks. That said, even with `unsafe.StringData`, you still can’t modify the contents of a string:

```
func main() {
    a := "Hello, World!"
    bptr := unsafe.StringData(a)

    *bptr = 'h'
}

// unexpected fault address 0x102f5c7bc
```

Attempting to change a string’s value directly like this may cause an immediate, unrecoverable crash. This happens because string literals are stored in a dedicated, read-only section of memory.

However, if a string is created dynamically at runtime, there are ways to manipulate its data using unsafe tricks—but at that point, you’re stepping outside Go’s safety guarantees and risking undefined behavior.

### 3.1 UTF-8 Encoding and Internal Representation

By default, strings in Go use UTF-8 encoding. So far, we've looked at how a string's header is stored in memory, but not how its actual content is represented in bytes. To understand this fully, it's important to break down a few key concepts that often cause confusion: ASCII, Unicode, and UTF-8.

This might seem like a detour, but it's essential to understanding how strings, bytes, and runes work together in Go. If you're already familiar with these terms, feel free to skip ahead.

ASCII (American Standard Code for Information Interchange) was one of the earliest character encoding standards. It used 7 bits per character, allowing for  $2^7$  (128) unique symbols.

Take the string "Hi" as an example. Each ASCII character has a corresponding decimal code: "H" maps to 72, and "i" maps to 105.

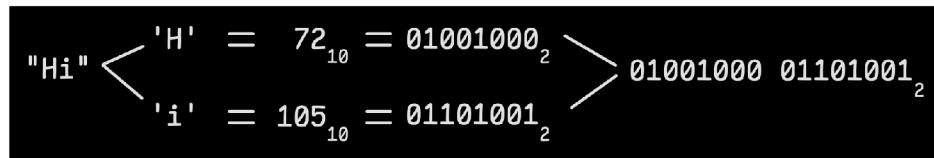


Illustration 28. ASCII encodes 'H' and 'i' numerically.

In terms of bytes, these decimal values translate to `010010002` (or `0x48`) for "H" and `011010012` (`0x69`) for "i". In memory, "Hi" is stored as the byte sequence `01001000 011010012`, with each character taking up one byte.

ASCII worked well for English text, keeping every character within the 0-127 range. Printing ASCII characters in Go is straightforward:

```
func main() {
    println(string(65)) // A is the 65th character in
    ASCII
    println(string(97)) // a
    println(string(48)) // 0
}
```

ASCII's simplicity made it widely adopted, becoming the standard for early computing.

As computers spread globally, it became clear that a more flexible system was needed to support languages beyond English. This led to **Unicode**, designed to accommodate a truly global character set. Unicode now includes over 140,000

characters and continues to grow, covering everything from English letters to Chinese characters, emojis, and specialized symbols.

Unicode assigns each character a unique number, known as a code point. However, Unicode itself doesn't dictate how these code points are stored in memory. That's where UTF-8 comes in.

We've already seen how ASCII works—it uses a single byte for each character since there are only 128 characters to represent. Each character's decimal value is simply converted into binary for storage.

Now, let's take something more complex, like the emoji "😊". This smiley face has the Unicode code point `U+1F60A`.

```
😊 = U+1F60A = 12852210 = 0001 1111 0110 0000 10102
      ||  
      UTF-8  
      ||
representation in memory?
```

Illustration 29. Code point U+1F60A 😊 stored in binary

Allocating 4 bytes for every Unicode character might seem like a simple solution since Unicode covers over 140,000 characters. But it's not that straightforward—or efficient.

The issue is that most Unicode characters don't need 4 bytes. Take the letter "H" from earlier—it only requires 1 byte. In fact, for the majority of everyday text, especially in English or other Latin-based languages, characters can be stored using just 1 or 2 bytes. Forcing every character into 4 bytes would waste a significant amount of memory.

This is where UTF-8 shines. It saves space by adjusting the storage size based on what each character actually needs. Instead of using a one-size-fits-all approach, UTF-8 uses a variable-width encoding system. Depending on the character, it can use anywhere from 1 to 4 bytes:

- Basic ASCII characters (U+0000 to U+007F) fit into 1 byte.
- Many European language characters use 2 bytes.
- Most East Asian characters need 3 bytes.

- Supplementary characters, like emojis, require 4 bytes.

For clarity, in `U+0000` to `U+007F`, the "U" stands for Unicode, and the "+" separates the Unicode prefix from the hexadecimal code point.

You can see this behavior in action with a simple Go program:

```
func main() {
    char1 := "A"
    char2 := "ñ"
    char3 := " "
    char4 := "😊"

    fmt.Printf("Character: %s, Bytes: %d\n", char1,
len(char1)) // 1 byte
    fmt.Printf("Character: %s, Bytes: %d\n", char2,
len(char2)) // 2 bytes
    fmt.Printf("Character: %s, Bytes: %d\n", char3,
len(char3)) // 3 bytes
    fmt.Printf("Character: %s, Bytes: %d\n", char4,
len(char4)) // 4 bytes
}

// Output:
// Character: A, Bytes: 1
// Character: ñ, Bytes: 2
// Character: , Bytes: 3
// Character: 😊, Bytes: 4
```

Another advantage of UTF-8 is that it's backward-compatible with ASCII. ASCII characters remain encoded in a single byte, just as they always were.

Interestingly, UTF-8 was co-developed by Ken Thompson and Rob Pike—two of the same minds behind the Go language. It's no coincidence that Go handles strings efficiently and practically.

UTF-8 follows a clear set of bit patterns to map Unicode code points into bytes effectively:

- **1-byte character:** For code points in the `U+0000` to `U+007F` range, UTF-8 uses a single byte. The bit pattern follows  $0xxxxxx_2$ , where the first bit is `0` and the remaining 7 bits store the character.
- **Multi-byte characters:** Characters needing 2, 3, or 4 bytes follow specific patterns:
  - $110xxxxx_2$  for 2-byte characters
  - $1110xxxx_2$  for 3-byte characters

- $11110xxx_2$  for 4-byte characters
- For multi-byte sequences, continuation bytes start with  $10xxxxxx_2$ .

```
B = 010000102
Ø = 11000011 100110002
字 = 11100101 10101101 100101112
😊 = 11110000 10011111 10011000 100010102
```

Illustration 30. UTF-8 adapts to characters with variable bytes

Working with UTF-8 encoded strings in Go is straightforward. You don't need to manage encoding manually, as Go handles it under the hood. The `range` keyword makes looping through strings intuitive—it iterates over Unicode code points instead of raw bytes. This allows you to handle characters directly without getting caught up in byte-level details:

```
func main() {
    s := "Hello,   "

    for i, r := range s {
        fmt.Printf("%#U starts at byte position
%d\n", r, i)
    }
}

// U+0048 'H' starts at byte position 0
// U+0065 'e' starts at byte position 1
// U+006C 'l' starts at byte position 2
// U+006C 'l' starts at byte position 3
// U+006F 'o' starts at byte position 4
// U+002C ',' starts at byte position 5
// U+0020 ' ' starts at byte position 6
// U+4E16 ' ' starts at byte position 7
// U+754C ' ' starts at byte position 10
```

This output shows each character's Unicode code point along with its starting byte position in the string. The variable `r` is a **rune**—Go's type for representing a Unicode code point.

What happens when you call `len(..)` on a string? More specifically, what does the `len` field of the string header actually count?

```
func main() {
    s := "Hello, "
    println(len(s))           // 13
    println(utf8.RuneCountInString(s)) // 9
    println(len([]rune(s)))    // 9
}
```

Calling `len` on a string returns the number of bytes, not the number of characters. This difference becomes clear when the string contains non-ASCII characters, which often use more than one byte.

To count actual Unicode characters, use `utf8.RuneCountInString`. Another approach is to convert the string into a slice of runes and measure its length—this also gives the correct character count.

There's a nuance to keep in mind: some characters, like certain emojis, can actually be **made up of multiple Unicode code points**.

Take the "𩿱" icon as an example:

```
len(s)           // 25
utf8.RuneCountInString(s) // 7
len([]rune(s))  // 7
```

Even though it visually appears as a single emoji, it's composed of multiple Unicode code points combined to create one symbol. That's why the byte length is much larger than the rune count.

In Go, the `for-range` loop doesn't iterate through raw bytes—it's designed to work with Unicode code points directly. This allows you to handle full characters without worrying about the underlying byte structure.

Behind the scenes, each loop iteration involves a few extra steps. Go creates three temporary variables during each cycle:

- `hv1` : the current byte position.
- `hv1t` : a copy of `hv1`, marking the starting byte position of the current character.
- `hv2` : the value of the current Unicode character (`rune`).

Here's a simplified version of what's happening internally—this is roughly how Go handles the `for-range` loop to process full characters instead of raw bytes:

```

for hv1 := 0; hv1 < len(s); {
    hv1t := hv1
    hv2 := rune(s[hv1])
    if hv2 < utf8.RuneSelf {
        hv1++
    } else {
        hv2, hv1 = decoderune(s, hv1)
    }
    v1, v2 = hv1t, hv2
}
// the loop body
}

```

The loop starts with `hv1 = 0` and continues while `hv1 < len(s)`. If the byte at `hv1` represents a single-byte character, `hv1` increments by 1. For multi-byte characters, Go uses `decoderune` to decode the entire Unicode character, returning both the rune and the index of the next character.

The `decoderune` function, found in the runtime package, handles the heavy lifting. It's what allows `for-range` to work seamlessly with full Unicode characters, rather than just raw bytes.

## 3.2 String Conversion and Memory Semantics

Go allows you to convert several types into strings—integers, rune characters, slices of bytes, and slices of runes. However, you can't directly convert an array to a string.

When converting an integer to a string, it's not about turning the number `10` into `"10"`. Instead, the conversion maps the integer to its corresponding Unicode character. The same logic applies to runes:

```

func main() {
    println(string(65)) // A
    println(string([]byte{72, 101, 108, 108, 111})) //
Hello
    println(string('H')) // H
    println(string([]rune{72, 101, 108, 108, 111})) //
Hello
}

```

Both `string(65)` and `string('H')` convert a number or rune into a character based on its Unicode code point. Behind the scenes, these conversions rely on Go's `runtime.intstring()` function:

Figure 25. intstring (src/runtime/string.go)

```
func intstring(buf *[4]byte, v int64) (s string) {
    var b []byte
    if buf != nil {
        b = buf[:]
        s = slicebytetostringtmp(&b[0], len(b))
    } else {
        s, b = rawstring(4)
    }
    if int64(rune(v)) != v {
        v = runeError
    }
    n := encoderune(b, rune(v))
    return s[:n]
}
```

Under the hood, the compiler sets up a 4-byte buffer **on the stack** and passes it to `intstring()`. This function writes the UTF-8 bytes of the integer into the buffer, which is stored in the `stringHeader.str` field.

Other types—like slices of bytes or runes—are handled in a similar way, each with their own optimized runtime function. For instance, converting a slice of bytes to a string uses `runtime.slicebytetostring`, while converting a slice of runes relies on `runtime.slicerunetostring`.

When converting a byte slice to a string in Go, the runtime uses a buffer that can handle up to 32 bytes. If the slice you’re converting is larger than that, it has to allocate a new buffer **on the heap** to fit everything:

Figure 26. slicebytetostring (src/runtime/string.go)

```
func slicebytetostring(buf *tmpBuf, ptr *byte, n int) string
{
    if n == 0 {
        return ""
    }
    ...
    if n == 1 {
        p := unsafe.Pointer(&staticuint64s[*ptr])
        if goarch.BigEndian {
            p = add(p, 7)
        }
        return unsafe.String((*byte)(p), 1)
    }
    var p unsafe.Pointer
```

```

        if buf != nil && n <= len(buf) {
            p = unsafe.Pointer(buf)
        } else {
            p = mallocgc(uintptr(n), nil, false)
        }
        memmove(p, unsafe.Pointer(ptr), uintptr(n))
        return unsafe.String((*byte)(p), n)
    }
}

```

This function handles four different scenarios when converting a byte slice into a string with specific optimizations:

- Empty byte slice:** Returns an empty string immediately. This early check is optimized for common cases in text processing.
- Single-byte slice:** Uses a preallocated array called `staticuint64s` to quickly convert a single byte into a string without extra memory allocation.
- Small byte slice ( $\leq 32$  bytes):** Uses a buffer provided by the caller or a pre-allocated stack buffer to store the string directly.
- Large byte slice ( $> 32$  bytes):** Allocates a new buffer on the heap if no existing buffer is available or the slice exceeds 32 bytes.

A key detail here is that even if the underlying bytes are allocated on the heap, the string itself isn't always heap-allocated. If certain conditions are met, the string remains stack-allocated, with `stringHeader.str` (an `unsafe.Pointer`) pointing to the heap buffer:

Figure 27. staticuint64s (src/runtime/iface.go)

```

// staticuint64s is used to avoid allocating in convTx for
// small integer values.
var staticuint64s = [...]uint64{
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
    0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
    0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f,
    0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
    0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f,
    0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f,
    0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
    0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f,
}

```

```

    0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
    0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
    0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
    0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
    0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
    0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
    0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
    0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf,
    0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
    0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf,
    0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7,
    0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
    0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
    0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed, 0xee, 0xef,
    0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
    0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
}

```

The `staticuint64s` array is a clever optimization. It holds precomputed values for every possible single-byte integer (0-255). When Go needs to convert a single-byte slice into a string, it can skip memory allocation entirely by pulling from this array directly. This speeds up conversions for small values and avoids unnecessary memory overhead.

To sum up, when you assign or create a new string from byte slice, Go typically allocates fresh memory and copies the bytes during the assignment. Once a string is created, it's fully isolated from any changes to the original byte array—thanks to its immutability.

This concept becomes clearer with a bit of pseudo-code:

Go	Pseudo Code
<pre> var s1 []byte s2 := string(s1) </pre>	<pre> s2Bytes := make([]byte, len(s1)) copy(s2Bytes, s1)  s2 = stringHeader{&amp;s2Bytes[0], len(s1)} </pre>

This is a simplified version to illustrate the idea. In reality, Go's implementation is more complex, as we've discussed earlier.

However, not every conversion from a byte slice to a string requires copying bytes into new memory. In some cases, Go can **reuse** the original byte slice directly as the string's underlying data. This optimization happens when the runtime can guarantee that the byte slice remains unchanged for the string's lifetime, making it safe to reference without copying.

This behavior often appears in scenarios like comparisons or switch statements:

```
func main() {
    ...

    switch string(anyBytes) {
    case "Hello":
        print("World")
    case "World":
        print("Hello")
    default:
        print("default")
    }

    if string(anyBytes) == "Hello" {
        print("Hello")
    }
}
```

In this example, `anyBytes` is converted to a string within the `switch` or `if` condition. Go's compiler recognizes that `anyBytes` isn't being modified in this context, so it safely uses the original byte slice without allocating new memory.

Instead of calling `runtime.slicebytetostring`, Go uses `runtime.slicebytetostringtmp`, which skips memory allocation when possible:

Figure 28. `slicebytetostringtmp` (`src/runtime/string.go`)

```
// slicebytetostringtmp returns a "string" referring to the
// actual []byte bytes.
//
// Callers need to ensure that the returned string will not
// be used after
// the calling goroutine modifies the original slice or
// synchronizes with
// another goroutine.
//
// The function is only called when instrumenting
// and otherwise intrinsified by the compiler.
//
```

```

// Some internal compiler optimizations use this function.
// - Used for m[T1{..., Tn{..., string(k), ...} ...}] and
m[string(k)]
// where k is []byte, T1 to Tn is a nesting of struct
and array literals.
// - Used for "<"+string(b)+">" concatenation where b is
[]byte.
// - Used for string(b)=="foo" comparison where b is
[]byte.
func slicebytetostringtmp(ptr *byte, n int) string {
    ...
    return unsafe.String(ptr, n)
}

```

The comment highlights where `slicebytetostringtmp` can optimize conversions—such as during comparisons or specific string concatenations. By avoiding unnecessary allocations, this function saves both time and memory when a full copy isn't needed.

### 3.3 Efficient String Concatenation Techniques

String concatenation is the process of joining two or more strings into one. In Go, the more efficient way to handle multiple concatenations is by using `strings.Builder` instead of the ```` operator. Since strings in Go are immutable, every time you use ```` to concatenate, a new string gets allocated. This can lead to unnecessary memory allocations and slow down performance, especially in loops or large-scale operations.

The compiler does try to optimize concatenation, particularly when working with constant strings:

```

func main() {
    const c1 = "Hello"

    str1 := "normal string"
    str2 := c1 + " World" + str1 + "!"

    print(str2)
}

```

In this example, `str2` is created by joining four strings: the constant `c1`, the constant `"World"`, the variable `str1`, and the constant `"!"`.

If we take a closer look at how the compiler handles this, the assembly output reveals some interesting optimizations:

## Go Assembly

```
MOVD $main..autotmp_2-
48(SP), R0
MOVD $go:string."Hello
World"(SB), R1
MOVD $11, R2
MOVD $go:string."normal
string"(SB), R3
MOVD $13, R4
MOVD $go:string."!"(SB),
R5
MOVD $1, R6
CALL
runtime.concatstring3(SB)
MOVD R0, main.str2.ptr-
8(SP)
MOVD R1, main.str2.len-
16(SP)
```

## Pseudo-code

```
{R1, R2} = string{"Hello
World", 11}
{R3, R4} = string{"normal
string", 13}
{R5, R6} = string{"!", 1}

R0, R1 =
concatstring3(R1, R3, R4)
main.str2 = R0
main.str2.len = R1
```

During compilation, Go optimizes the code by merging constant or literal strings ahead of time. For example, the compiler combines `c1 ("Hello")` and `"World"` into a single string, reducing the need for multiple concatenation steps at runtime. Instead of joining four strings, the compiler simplifies the operation into a single call to `runtime.concatstring3`, which joins three strings efficiently.

The reason `"normal string"` isn't merged with the other constants is that Go only combines strings known at compile time—constants and literals. Since `str1` is a variable, its value could change at runtime, and Go avoids merging it to ensure correctness.

The `concatstring3` function is an optimization designed for efficiently joining three strings. However, the real work happens inside `runtime.concatstrings`:

Figure 29. concatstring3 & concatstrings (src/runtime/string.go)

```
func concatstring3(buf *tmpBuf, a0, a1, a2 string) string {
    return concatstrings(buf, []string{a0, a1, a2})
}

func concatstrings(buf *tmpBuf, a []string) string {
    ...
    // Step 1: Skip empty strings
    for i, x := range a {
        n := len(x)
```

```

        if n == 0 {
            continue
        }
        ...
    }
    if count == 0 {
        return ""
    }

    // Step 2: If there's only one string and memory
    // conditions are met, return it directly
    if count == 1 && (buf != nil || !stringDataOnStack(a[idx])) {
        return a[idx]
    }

    // Step 3: Allocate or reuse buffer
    s, b := rawstringtmp(buf, 1)

    // Step 4: Copy the bytes into the buffer
    for _, x := range a {
        copy(b, x)
        b = b[len(x):]
    }

    return s
}

```

This function takes each string, joins them together, and writes them into a **preallocated buffer on the stack**, copying the bytes sequentially into that space.

When Go detects that a temporary string, such as one created from a `[]byte` to `string` conversion or a quick concatenation, doesn't need to persist beyond the immediate operation, it avoids heap allocation. Instead, the compiler sets up a small 32-byte buffer on the stack and passes it to functions like `slicebytetostring` or `concatstrings`.

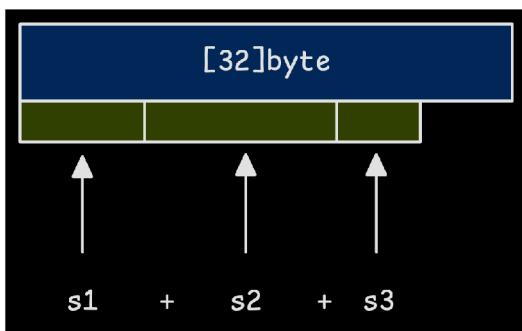


Illustration 31. Strings concatenated into a stack buffer

If the result fits within that 32-byte buffer, the runtime uses it directly on the stack, avoiding heap allocation altogether. The reason behind the 32-byte buffer isn't arbitrary, it's practical. Most short-lived strings are 32 bytes or fewer. This size efficiently handles the majority of temporary strings without adding the overhead of heap allocation.

For now, Go offers several built-in optimizations for string concatenation, each tailored to handle different numbers of strings efficiently:

Figure 30. concatstring2 to concatstring5 (src/runtime/string.go)

```
func concatstring2(buf *tmpBuf, a0, a1 string) string {
    return concatstrings(buf, []string{a0, a1})
}

func concatstring3(buf *tmpBuf, a0, a1, a2 string) string {
    return concatstrings(buf, []string{a0, a1, a2})
}

func concatstring4(buf *tmpBuf, a0, a1, a2, a3 string)
string {
    return concatstrings(buf, []string{a0, a1, a2, a3})
}

func concatstring5(buf *tmpBuf, a0, a1, a2, a3, a4 string)
string {
    return concatstrings(buf, []string{a0, a1, a2, a3,
a4})
}
```

The Go compiler automatically selects the most efficient function based on the number of strings being concatenated:

Figure 31. walkAddString (src/cmd/compile/internal/walk/expr.go)

```
func walkAddString(n *ir.AddStringExpr, init *ir.Nodes)
ir.Node {
    ...

    var fn string
    if c <= 5 {
        // small numbers of strings use direct
        // runtime helpers.
        // note: order.expr knows this cutoff too.
        fn = fmt.Sprintf("concatstring%d", c)
    } else {
        // large numbers of strings are passed to
        // the runtime as a slice.
    }
}
```

```

        fn = "concatstrings"

        t :=
types.NewSlice(types.Types[types.TSTRING])
    // args[1:] to skip buf arg
    slice := ir.NewCompLitExpr(base.Pos,
ir.OCOMPLIT, t, args[1:])
    slice.Prealloc = n.Prealloc
    args = []ir.Node{buf, slice}
    slice.SetEsc(ir.EscNone)
}
...
}

```

When working with five or fewer strings (`c <= 5`), the compiler calls specialized helper functions like `concatstring3` or `concatstring5`. This makes the process faster and reduces overhead by avoiding unnecessary allocations.

If there are more than five strings, the strategy changes. Instead of calling a fixed helper function, Go builds a slice containing all the strings and passes it to `concatstrings`.

With concatenation covered, that wraps up our deep dive into strings.

## 4. Maps: Internal Design and Runtime Behavior

A map is one of Go's core data types, designed to hold key-value pairs. It's an unordered collection where each key must be unique and comparable.

In this context, **comparable** means that Go must be able to determine if two keys are equal using `==` or `!=`. While values can be of nearly any type, keys need to support this comparison. If you're familiar with dictionaries in Python, objects in JavaScript, `HashMap` in Java, or `unordered_map` in C++, this concept should feel familiar.

```

func main() {
    m := map[string]int{
        "key1": 1,
        "key2": 2,
    }

    fmt.Println(m) // map[key1:1 key2:2]
}

```

There are two main ways to declare a map. You can use map literals, as shown above, or the `make` function, which works similarly to how slices are initialized. Unlike slices, maps don't have a fixed size. Instead, you can provide a **capacity hint**:

```
func main() {
    m := make(map[string]int, 10)
}
```

In this case, you're giving Go a rough estimate of how many elements you plan to store. This isn't a strict limit—maps automatically grow as needed. This idea will make more sense once we get into how maps work internally.

A crucial detail is that keys must be unique. If you insert a key that already exists, Go will overwrite the existing value with the new one.

For a map to determine whether a key exists, the key type must be comparable. Therefor, some data types—such as slices, maps, and functions—cannot be used as keys because they are not comparable or do not support `==` or `!=`. In such cases, Go has no way to determine whether two values are the same.

*While the types we just listed above aren't generally comparable, they can still be compared to nil; just not to themselves.*

```
func main() {
    // int
    m1 := map[int]int{}

    // string
    m2 := map[string]int{}

    // array
    m3 := map[[1]int]int{}

    // slice
    // compile error: invalid map key type []int
    m4 := map[[]int]int{}

    // map
    // compile error: invalid map key type
    map[string]int
    m5 := map[map[string]int]int{}

    // function
    // compile error: invalid map key type func()
```

```
m6 := map[func()]int{}  
}
```

This example above shows how Go handles different types as keys. Simple, comparable types like integers, strings, and arrays work just fine. But when you try to use uncomparable types, you'll run into compile-time errors.

Even though the Go compiler does a solid job of catching uncomparable keys at compile time, there's still a way to slip uncomparable types into a map: by using `interface{}` as the key type.

In this case, Go won't block you during compilation because `interface{}` bypasses the usual type restrictions. It's essentially a free pass to use any type, including those that aren't normally comparable:

```
func main() {  
    m := map[interface{}]int{  
        1: 1,  
        "key": 2,  
        [1]int{1}: 3,  
  
        // panic: runtime error: hash of unhashable  
        type [1]func()  
        [1]func(){func() {}}: 4,  
  
        // panic: runtime error: hash of unhashable  
        type []int  
        []int{1, 2}: 5,  
  
        // panic: runtime error: hash of unhashable  
        type map[string]int  
        map[string]int{"key": 1}: 6,  
  
        // panic: runtime error: hash of unhashable  
        type func() int  
        func() int { return 1 }: 7,  
    }  
  
    fmt.Println(m)  
}
```

Instead of catching the issue upfront, Go lets you find out that the key type is uncomparable at runtime.

The application panics because some of these keys can't be hashed, and hashing is essential for keys to work in a map. This applies any time you use

`interface{}` as a key and try to add incomparable values—not just at declaration.

Another important behavior to understand is how Go handles `nil` maps. Interestingly, you can still read from it without triggering an error; it behaves as if it's an empty map and simply returns zero values for any keys you query. The problem arises when you try to write to a `nil` map:

```
func main() {
    var m map[string]int

    fmt.Println(m)          // map[]
    fmt.Println(m["key"])   // 0

    m["key"] = 1           // panic: assignment to entry in nil
    map
}
```

Reading from a `nil` map works without issue—Go treats it as empty, though it isn't usable in the full sense. But as soon as you attempt to write to it, the application panics.

This behavior reflects Go's strict handling of types. A `nil` map behaves like a read-only map. When printed, it also shows up as an empty map (`map[]`). You can still check for values or loop over it with `for-range` without needing to verify if it's `nil` first.

In addition, when initializing a map with values from the start, you don't always need to specify the types of the keys and values explicitly:

```
type Person struct {
    Name string
    Age  int
}

func main() {
    // Explicit type declaration
    expM := map[Person][]string{
        Person{Name: "Alice", Age: 30}:
    []string{"Apple", "Banana"},
        Person{Name: "Bob", Age: 25}:
    []string{"Cherry", "Date"},
    }

    // Implicit type declaration
}
```

```

    impM := map[Person][]string{
        {Name: "Alice", Age: 30}: {"Apple",
"Banana"},
        {Name: "Bob", Age: 25}: {"Cherry",
"Date"},
    }
}

```

Go compiler can infer types based on the key-value pairs provided during initialization. Both `expM` and `impM` work the same way, but `impM` is shorter and cleaner. When the context clearly shows what types you're working with, this approach is good practice; it reduces redundancy without sacrificing clarity.

## 4.1 Internal Structure and Memory Layout

Under the hood, Go maps use a structure called a hashtable. The Go runtime handles hashing keys, managing collisions, resizing, and distributing key-value pairs efficiently.

If you're not familiar with hashtables, think of them as structures with a fixed number of **buckets** and a hash function (`hash(key)`) that determines where each key-value pair will be stored:

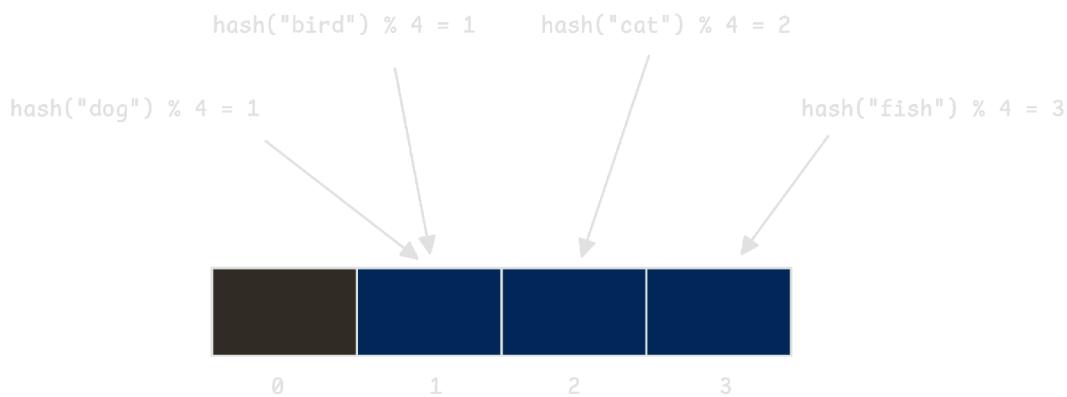


Illustration 32. Map keys distributed using a hash function

Imagine a hashtable with 4 buckets, and you're adding the keys `"cat"`, `"dog"`, `"fish"`, and `"bird"`. A hash function might distribute them like this:

- `hash("cat") % 4 → slot 2`
- `hash("dog") % 4 → slot 1`
- `hash("fish") % 4 → slot 3`
- `hash("bird") % 4 → slot 1` (collides with `"dog"`)

When a collision occurs, Go uses **chaining**. Instead of limiting each bucket to a single item, each bucket becomes a list that can hold multiple elements. This allows both `"dog"` and `"bird"` to share slot 1 by forming a simple linked list:

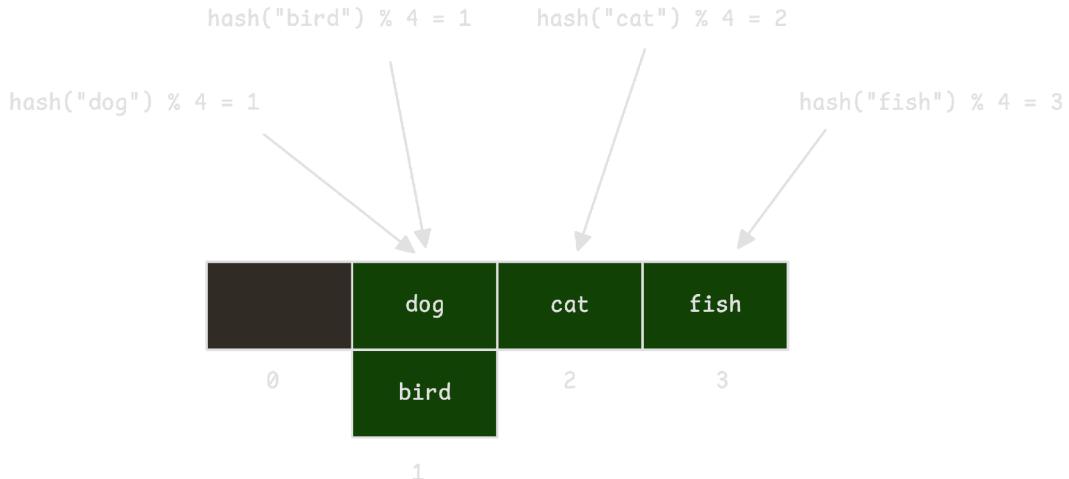


Illustration 33. Collisions handled using chaining in buckets

To find `"bird"` in this setup, the search starts at slot 1 and moves through the list until the key is found.

- In most cases, Go maps offer  $O(1)$  lookup time due to efficient hashing.
- In the worst case, when too many keys collide in the same bucket, lookup time can degrade to  $O(n)$  for that bucket ( $n$  being the number of elements in that particular bucket—not the total number of elements in the map).

While this gives a basic understanding of how a hashtable functions, Go's map implementation adds more complexity. The runtime manages much of this automatically, adjusting the map's structure dynamically during execution rather than enforcing strict rules at compile time.

Like slices (`[]T`) and strings (`string`), maps (`map[K]V`) also have an internal representation. We can use `println` to inspect what the type looks like:

```
func main() {
    m := map[string]int{
        "key1": 1,
        "key2": 2,
    }
    println(m)
}

// Output: 0x1400004e728
```

Instead of the map's contents, you see a memory address. This reveals that the `map[K]V` type you normally use is actually a pointer to a deeper structure. That structure is called `hmap`, and it manages the map's internal mechanics:

Figure 32. type hmap struct (src/runtime/map.go)

```
// A header for a Go map.
type hmap struct {
    count      int      // # live cells == size of map.
    Must be first (used by len() builtin)
    flags      uint8
    B          uint8    // log_2 of # of buckets (can hold
    up to loadFactor * 2^B items)
    noverflow uint16   // approximate number of overflow
    buckets; see incrnoverflow for details
    hash0      uint32   // hash seed

    buckets     unsafe.Pointer // array of 2^B Buckets.
    May be nil if count == 0.
    oldbuckets unsafe.Pointer // previous bucket array,
    non-nil only when growing
    nevacuate  uintptr      // progress counter for
    evacuation (buckets less than this have been evacuated)

    extra *mapextra // optional fields
}
```

The `hmap` struct manages an array of buckets, and the number of buckets always follows a power of two: 1, 2, 4, 8, 16, and so on. Each time the map grows because it has too many elements, the bucket count doubles, making sure that there is enough space to handle new items efficiently.

This is not required reading for now, but I can give a rough explanation of each field:

- `count` : Tracks the number of items currently in the map. Since it's the first field, Go can quickly retrieve the map size when calling `len()`.
- `flags` : Stores status information about the map. Whether the map is resizing, in the middle of an iteration, or being accessed concurrently, `flags` keeps track.
- `B` : Represents the power of two for the number of buckets. If `B` is 4, the map has  $2^4$ , or 16 buckets.
- `noverflow` : Gives an estimate of how many overflow buckets are in use because the main buckets are full. It's not exact, but accurate enough for tracking.

- `hash0` : A unique seed used for hashing keys. Every map has its own hash seed to ensure even distribution.
- `buckets` : Points to the array of buckets where map entries are stored.
- `oldbuckets` : Keeps a reference to the previous bucket array during resizing, allowing data to transfer gradually instead of all at once.
- `nevacuate` : Acts as a progress counter for moving data during resizing, ensuring the transfer is happening smoothly.
- `extra` : Holds optional fields for additional storage or configuration when needed.

Looking closer at the `hmap` structure reveals a few important insights:

- Each map instance has a different seed value (`hash0`) for the hash function, helping distribute keys evenly across buckets.
- Each bucket has a limit on the number of elements it can hold. When these buckets fill up due to a high collision rate (like `bird` and `dog` in the example above), overflow buckets step in, acting as temporary storage for the overflowed elements.

Now, the main focus here is on the `buckets`—where the actual data lives.

## Buckets and Collision Management

Going back to the hashtable example, think of those 4 slots we discussed earlier. In Go, those slots are called **buckets**. Each bucket can hold up to `8` key-value pairs.

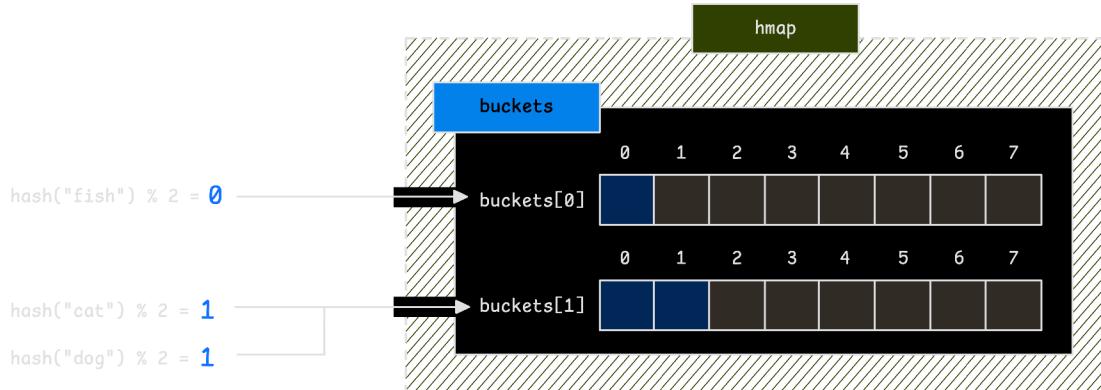


Illustration 34. Each bucket can store up to 8 pairs.

When collisions occur, these 8 slots begin to fill. However, that doesn't mean a bucket is limited to just 8 pairs.

When bucket A fills up and another key-value pair lands in the same slot, a collision occurs, and the new pair also maps to bucket A. Instead of resizing the map immediately, Go creates an overflow bucket linked to the original bucket A. This overflow chain can keep growing, with overflow buckets linking to additional overflow buckets as needed:

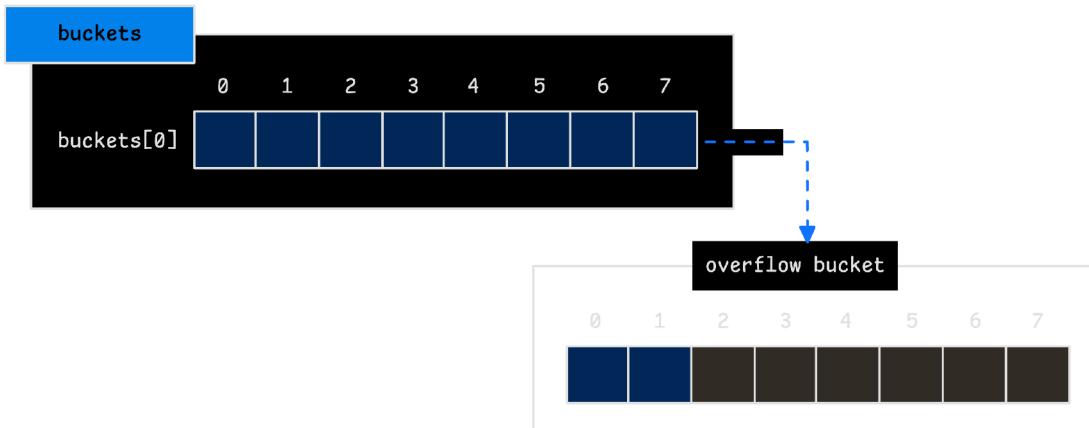


Illustration 35. Overflow buckets prevent immediate map resizing

Each bucket is managed by a structure called `bmap` (bucket map), which looks like this in Go's runtime:

Figure 33. type bmap struct (src/runtime/map.go)

```
// A bucket for a Go map.
type bmap struct {
    tophash [bucketCnt]uint8
}
```

Beyond just key-value pairs, each bucket contains an array of bytes called `tophash` (`tophash`), which stores the top byte of each key's hash value.

Wait a minute, we obviously see the top-hash, an array of 8 bytes (`tophash [bucketCnt]uint8`), but where are the keys and values in the `bmap` structure?

It turns out the keys and values are stored directly after the top-hash array in memory. However, this data isn't directly accessible—you have to use unsafe pointer arithmetic to access it:

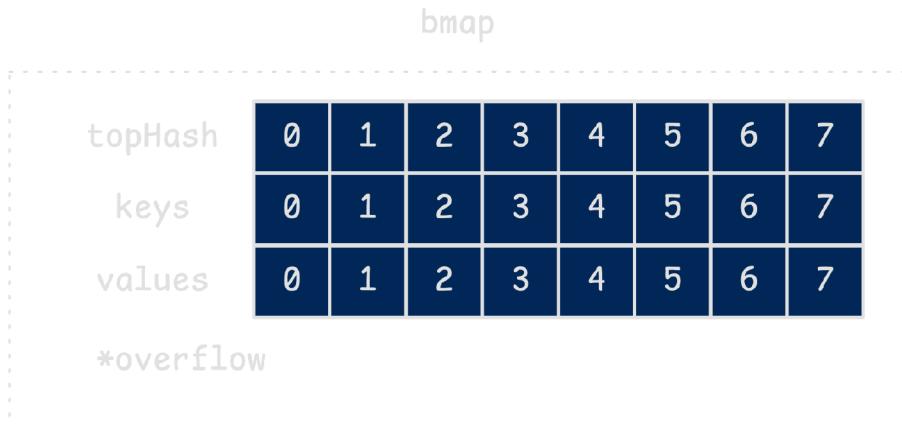


Illustration 36. Keys and values stored after the top-hash array

The reason lies in how Go's map implementation was designed. The built-in map type (`map[K]V`) was developed long before generics were introduced in Go 1.18, so it needed a way to handle different types of keys and values dynamically. That's why it can't define specific key and value types in the `bmap` structure.

If we imagine how a more generic bucket structure might look using modern Go features, it could resemble this:

```
type bmap[K comparable, V any] struct {
    tophash [bucketCnt]uint8
    keys     [bucketCnt]*K // or [bucketCnt]*K
    values   [bucketCnt]*V // or [bucketCnt]*V
    overflow *bmap[K, V]
}
```

*This isn't how Go actually implements maps internally, but it illustrates how generics could theoretically work.*

In this structure, `K` and `V` represent the key and value types, respectively. In fact, when dealing with large keys, Go doesn't store the entire key directly in the bucket. Instead, it stores a pointer to the key to keep the bucket size manageable (`keys [bucketCnt]*K` or `values [bucketCnt]*V`).

So, what does Go consider "large"? This is determined at compile time. If a key or value exceeds 128 bytes, Go stores a pointer in the bucket instead of the actual data.

The following code shows how Go makes that decision:

Figure 34. map: MapOf (src/reflect/type.go)

```
func MapOf(key, elem Type) Type {
    ...

        if ktyp.Size_ > maxKeySize { // maxKeySize is
            currently set to 128
                mt.KeySize = uint8(goarch.PtrSize)
                mt.Flags |= 1 // indirect key
        } else {
            mt.KeySize = uint8(ktyp.Size_)
        }

        if etyp.Size_ > maxValSize { // maxValSize is
            currently set to 128
                mt.ValueSize = uint8(goarch.PtrSize)
                mt.Flags |= 2 // indirect value
        } else {
            mt.MapType.ValueSize = uint8(etyp.Size_)
        }

    ...
}

func (mt *MapType) IndirectKey() bool { // store ptr to key
    instead of key itself
    return mt.Flags&1 != 0
}
func (mt *MapType) IndirectElem() bool { // store ptr to
    elem instead of elem itself
    return mt.Flags&2 != 0
}
```

The flags `1` and `2` in `MapType` indicate whether the map stores a pointer to the key or value instead of the actual data. Meanwhile, `maxKeySize` and `maxValSize` act as thresholds (currently set at 128 bytes) to determine when a key or value is considered "large."

And that's the story of key and value storage in a bucket. Now, let's talk about the usage of the top-hash.

## Top Hash Optimization

When the Go runtime hashes a key, it extracts the top byte of the hash value to use as a shorthand identifier—this is known as the **tophash**.



Illustration 37. Tophash uses the top byte of the hash

Searching for a key in a bucket can be slow, especially if the keys are large, like structs packed with fields. This is where `tophash` comes in—it acts as a quick filter to narrow down potential matches. It doesn't guarantee an exact match, but it quickly eliminates irrelevant entries, making the search process far more efficient.

Imagine a scenario where a map has only one bucket, and two keys, `"dog"` and `"cat"`, collide and end up in the same bucket:

```
func main() {
    m := map[string]int{"dog": 1, "cat": 2}

    println(m["dog"]) // lookup "dog"
}
```

When searching for the key `"dog"` (`println(m["dog"])`), Go doesn't compare `"dog"` directly with every key in the bucket. First, the runtime hashes the key, extracts the **top** byte of the hash, and compares it against the `tophash` values stored in the bucket:

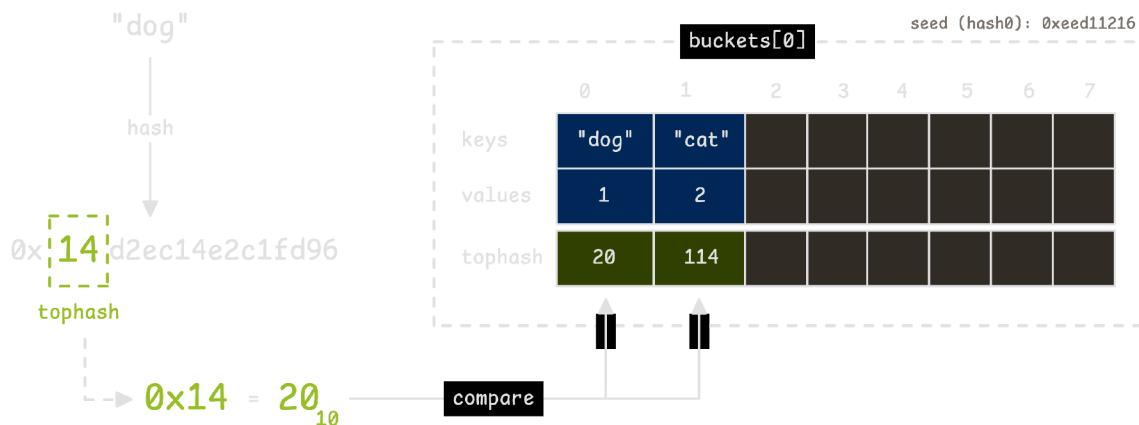


Illustration 38. Key 'dog' hashed and tophash compared

If the top hash matches, Go then checks the actual key to confirm the match. But what if an empty slot's top hash is 0, and a key's top hash is also 0? In this case,

the Go runtime reserves top-hash values from 0 to 4 for special purposes. If a key's top hash is 0, it is adjusted to 5 instead:

Figure 35. func tophash(hash uintptr) uint8 (src/runtime/map.go)

```
const minTopHash = 5

func tophash(hash uintptr) uint8 {
    top := uint8(hash >> (goarch.PtrSize*8 - 8))
    if top < minTopHash {
        top += minTopHash
    }
    return top
}
```

Besides representing the top-hash of a key, it also indicates *the state of each element* in a bucket—such as whether it is empty, has been moved, and so on.

These specific states are defined in Go's runtime as constants:

Figure 36. Top-hash preserved values (src/runtime/map.go)

```
const (
    emptyRest      = 0 // Cell is empty, with no non-
empty cells in higher indexes or overflows.
    emptyOne       = 1 // Cell is empty.
    evacuatedX     = 2 // Entry has been evacuated to
the first half of a larger table.
    evacuatedY     = 3 // Entry evacuated to the second
half of a larger table.
    evacuatedEmpty = 4 // Cell is empty, and the bucket
has been evacuated.
    minTopHash     = 5 // Minimum tophash for a normally
filled cell.
)
```

Many of these values relate to something new called 'evacuation.' This process handles moving key-value pairs from old buckets to new ones when the map resizes or reorganizes. We will cover this in detail in the next section.

## 4.2 Operational Semantics and Usage Patterns

Most map operations are handled behind the scenes by the `runtime` package. The Go compiler rewrites certain parts of map-related code to call specific runtime functions.

The table below breaks down how this works in simplified pseudo-code:

Go Code	Pseudo Code
<code>m = make(map[K]V, hint)</code>	<code>runtime.makemap(m, hint) hmap</code>
<code>m[key] = value</code>	<code>runtime.mapassign(m, key, value)</code>
<code>v = m[key]</code>	<code>runtime.mapaccess1(m, key, &amp;value)</code>
<code>v, ok = m[key]</code>	<code>runtime.mapaccess2(m, key, &amp;value, &amp;ok)</code>
<code>delete(m, key)</code>	<code>runtime.mapdelete(m, key)</code>
<code>clear(m)</code>	<code>runtime.mapclear(m)</code>

*The actual code generated by the compiler isn't identical to this pseudo-code, but it follows the same structure.*

Go also adjusts its behavior based on the key type. While we won't get into every variation here, it's important to know that. For example, `map[string]V` behaves slightly differently from `map[OtherType]V`. We focus on the general version of map operations, but the underlying concepts apply to other key types as well.

## Map Initialization and Allocation

When using `make(map[K]V, hint)` to create a new map, we provide the Go runtime with an estimate of how much space it **might** need to store elements. This estimate is called a 'hint' rather than 'capacity' or 'length,' as it would be with slices or arrays:

```
func main() {  
    m := make(map[string]int, 10)
```

```
        fmt.Println(len(m)) // 0
    }
```

Even though we set a hint of 10, calling `len(m)` right after creation returns 0. That's because the map's length reflects the actual number of key-value pairs stored, which is zero at this point. The hint doesn't affect the reported length or capacity directly.

What it does influence is the internal allocation strategy. By pre-allocating buckets based on the hint, the runtime reduces the need for immediate resizing, improving performance.

This behavior becomes clearer when looking at the internals of `make(map)`, specifically the `runtime.makemap` function:

Figure 37. map: makemap (src/runtime/map.go)

```
func makemap(t *maptype, hint int, h *hmap) *hmap {
    ...

    if h == nil {
        h = new(hmap)
    }

    // 1. Set the hash seed
    h.hash0 = uint32(rand())

    // 2. Determine the number of buckets based on the
    // hint
    B := uint8(0)
    for overLoadFactor(hint, B) {
        B++
    }
    h.B = B

    // 3. Create the buckets, potentially preallocating
    // overflow buckets
    if h.B != 0 {
        var nextOverflow *bmap
        h.buckets, nextOverflow = makeBucketArray(t,
h.B, nil)
        if nextOverflow != nil {
            h.extra = new(mapextra)
            h.extra.nextOverflow = nextOverflow
        }
    }
}
```

```
        return h  
    }
```

The Go runtime uses the `hint` to estimate the initial number of buckets, setting  $B$  so the map starts with  $2^B$  buckets.

In the earlier example, where the hint is `10`, Go initially creates 2 buckets. Since each bucket can hold up to 8 key-value pairs, this setup offers 16 slots in total.

However, it's not just about fitting the hint exactly. For instance, with a hint of `14` elements, Go doesn't stop at `2` buckets—even though `16` slots would technically cover it. Instead, a 'load factor' comes into play, dictating how full the buckets should be to maintain performance. In this case, the runtime allocates `3` buckets to avoid hitting that threshold too soon.

If no `hint` is given, the Go runtime simplifies the process by using a leaner version of the `makemap` function called `runtime.makemap_small`. This function is optimized for creating small maps and is triggered when the hint is either missing or small enough—typically fewer than 8 elements, which aligns with the initial bucket size:

Figure 38. Small Map Creation (src/runtime/map.go)

```
func makemap_small() *hmap {  
    h := new(hmap)  
    h.hash0 = uint32(rand())  
    return h  
}
```

In this case, Go skips early bucket allocation, and the map naturally starts with one bucket ( $2^0$ ).

## Value Lookup and Retrieval

When accessing a value in a map using `v := m[key]`, the Go compiler steps in and rewrites the operation as a call to `runtime.mapaccess1`. For efficiency, specialized versions handle different key types—such as `runtime.mapaccess1_fast32`, `runtime.mapaccess1_fast64`, and `runtime.mapaccess1_faststr`.

The `1` in `mapaccess1` signals that there's another variant—and there is. The `mapaccess2` function returns both the value and a boolean indicating whether the

key exists in the map. This two-part return is useful when you need to know not just the value, but also whether the key was found.

This gives us two ways to access map values:

- `v := m[key]` retrieves the value.
- `v, ok := m[key]` retrieves the value and checks if the key exists.

If you read from `m[key]` on a `nil` or empty map (`h.count == 0`), Go immediately returns the zero value for that type. As mentioned earlier, reading from a `nil` map won't cause a panic, but writing to one will. There's one exception: if the key isn't comparable, accessing it will trigger a panic, whether the map contains data or not.

This behavior differs from earlier Go versions, where reading an incomparable key from a `nil` or empty map didn't cause a panic (see: Issue #23734 [[mapiter](#)]):

Figure 39. map: Preparing for Map Access - mapaccess1  
(src/runtime/map.go)

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...
    // 1. Check if the map is empty or if the key is
invalid (uncomparable)
    if h == nil || h.count == 0 {
        if err := mapKeyError(t, key); err != nil {
            panic(err)
        }
        return unsafe.Pointer(&zeroVal[0])
    }

    // 2. Detect concurrent write operations
    if h.flags&hashWriting != 0 {
        fatal("concurrent map read and map write")
    }
    ...
}
```

Go also enforces strict rules against concurrent reads and writes on a map. If it detects any simultaneous operation involving a write (`h.flags&hashWriting != 0`), it throws a fatal error—not a panic—leaving no chance for recovery.

Go runtime then computes a hash for the key using a hash function designed for that key's type. This hash is a large number, but it's not used directly. Instead, Go

applies a mask (`m`) to the hash to determine which bucket might contain the key.

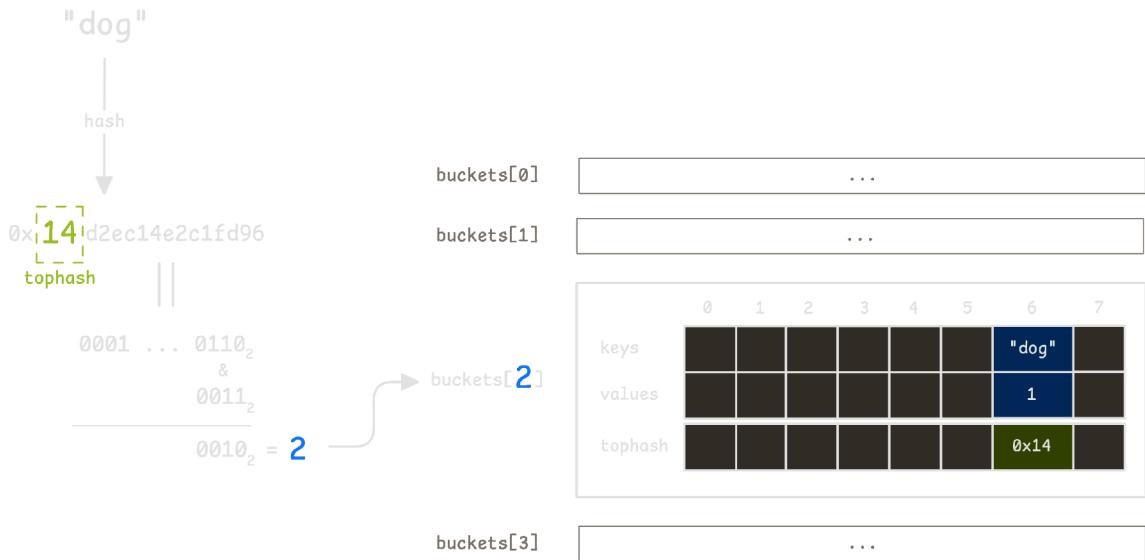


Illustration 39. Key 'dog' maps to bucket[2]

Imagine you have 4 buckets. The mask would be `3` (`011` in binary). Applying the mask produces a result between `0` and `3`, effectively selecting one of the four buckets. If the result is `2`, Go targets the third bucket.

After computing the hash, Go extracts the top-hash from the hash value:

Figure 40. Hash Calculation - mapaccess1 (src/runtime/map.go)

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...
    // 3. Get the hash, apply bucket mask, calculate
    //      tophash, and find the bucket
    hash := t.Hasher(key, uintptr(h.hash0))
    m := bucketMask(h.B)
    b := (*bmap)(add(h.buckets,
        (hash&m)*uintptr(t.BucketSize)))
    ...
    top := tophash(hash)
    ...
}
```

By adding an offset to the base address of the buckets through pointer arithmetic (`(add(h.buckets, (hash&m)*uintptr(t.BucketSize))`), Go locates the exact

memory address where that bucket begins.

Once the target bucket is found, Go searches through its 8 slots and compares the top-hash values in each slot. If there's a match, Go then compares the actual key to confirm the hit. This quick top-hash check eliminates unnecessary full-key comparisons, speeding up the lookup process:

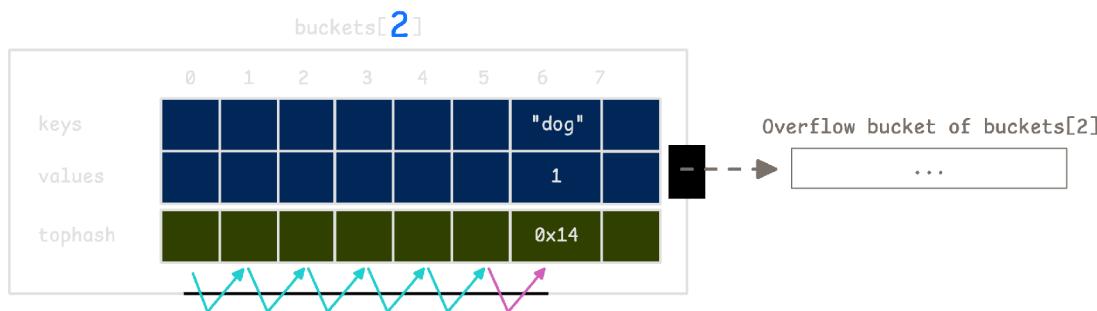


Illustration 40. Key 'dog' matched by tophash and key check

If the key isn't found in the current bucket, Go moves on to any overflow buckets and repeats the same checks until it either finds the key or exhausts all possibilities:

Figure 41. Map Access (Iterating) - mapaccess1 (src/runtime/map.go)

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...

    // 4. Iterate over the buckets and their overflow
buckets
bucketloop:
    for ; b != nil; b = b.overflow(t) {
        for i := uintptr(0); i < abi.MapBucketCount;
i++ {
            // 4.1 Check the tophash
            if b.tophash[i] != top {
                if b.tophash[i] == emptyRest
{
                    break bucketloop
                }
                continue
            }

            // 4.2 Get the key from the bucket
using address arithmetic
            k := add(unsafe.Pointer(b),
dataOffset+i*uintptr(t.KeySize))
```

```

        if t.IndirectKey() {
            k = *(*unsafe.Pointer)(k))
        }

        // 4.3 Compare the key
        if t.Key.Equal(key, k) {
            e := add(unsafe.Pointer(b),
dataOffset+bucketCnt*uintptr(t.KeySize)+i*uintptr(t.ValueSiz
e))
            if t.IndirectElem() {
                e = *
(*unsafe.Pointer)(e))
            }
        }
    }

    // 5. Key not found
    return unsafe.Pointer(&zeroVal[0])
}

```

Earlier, we talked about reserved values for top-hash. One of them, `emptyRest`, acts as a signal that the current slot—and everything after it—is empty. This lets Go break out of the loop early, saving time by avoiding unnecessary checks.

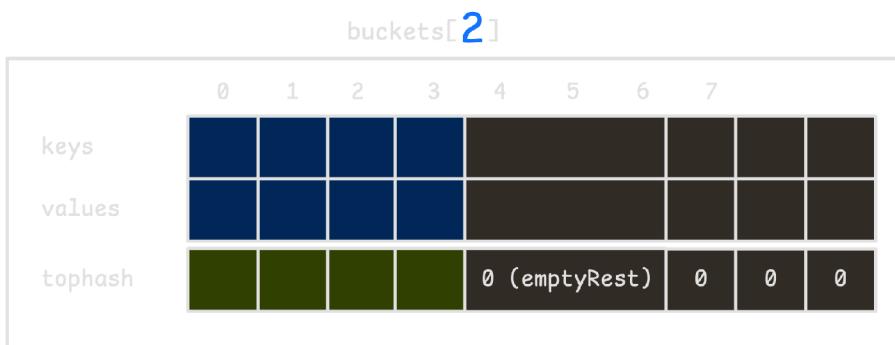


Illustration 41. Top-hash 0 signals end of search.

If the key isn't found after scanning the bucket and all overflow buckets, Go returns a pointer to a predefined zero value, indicating that the key doesn't exist in the map.

The `mapaccess2` function works much like `mapaccess1`, but with one key difference: it returns a boolean alongside the value. If the key is found, it returns `e, true`; if not, it returns `unsafe.Pointer(&zeroVal[0]), false`. This allows you to check both the value and whether the key was present.

Beyond these two core functions, Go also uses specialized implementations to optimize performance for specific key types. For example, `mapaccess1_fast32` is tailored for integer keys (`int` or `uint32`), while `mapaccess1_faststr` is optimized for string keys. These specialized functions take advantage of the unique characteristics of each key type—such as memory layout, hashing behavior, and comparison methods—to speed up operations.

For instance, `mapaccess1_faststr`, designed for string keys, skips the top-hash check entirely for short strings (under 32 bytes) and compares the keys directly. For longer strings, Go uses an optimization: instead of comparing the entire string, it checks the beginning and the end first to quickly rule out non-matching keys.

## Insertion and Updating of Entries

Adding a key-value pair to a map is managed by the `mapassign` function, which handles both inserting new pairs and updating existing ones. What makes this operation more complex is that it requires both reading and writing.

First, Go checks if the key already exists (reading); only then can it insert a new value or update the existing one (writing). Since we've already covered how reading works, understanding the write operation should now feel more intuitive.

However, this is also where the map's growth process starts to come into play, setting the stage for a deeper look at how resizing works.

### 1. Checking the Map's State

As mentioned earlier, trying to write to a `nil` map triggers a panic. But writing to a map from multiple goroutines at the same time is more serious—it results in a fatal error.

So, what's the difference between a panic and a fatal error?

Assigning to a `nil` map triggers a panic because it's an issue confined to a single goroutine—something that can be caught and handled. Concurrent writes reflects a fundamental concurrency issue. Go doesn't allow any recovery from these race conditions; instead, it stops execution entirely, signaling a critical flaw in the program's design:

Figure 42. Map Write Panic & Fatal - mapassign (src/runtime/map.go)

```

func mapassign(t *maptpe, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    if h == nil {
        panic(plainError("assignment to entry in nil
map"))
    }
    ...
    if h.flags&hashWriting != 0 {
        fatal("concurrent map writes")
    }
    ...
    h.flags ^= hashWriting
}

```

## 2. Calculate the Hash and Find the Target Bucket

Just like when accessing a key, adding a key-value pair also finds the hash, determines the target bucket, and calculates the top-hash.

However, before proceeding with this, Go checks whether the map is in the middle of a growth phase. If it is, the write operation helps move that process forward:

Figure 43. Map Write - Calculating Hash (src/runtime/map.go)

```

func mapassign(t *maptpe, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...
    hash := t.Hasher(key, uintptr(h.hash0))
again:
    bucket := hash & bucketMask(h.B)
    if h.growing() {
        growWork(t, h, bucket)
    }
    b := (*bmap)(add(h.buckets,
bucket*uintptr(t.BucketSize)))
    top := tophash(hash)
    ...
}

```

Map growth doesn't happen all at once; it's a *gradual process that unfolds over multiple operations*. Each time a new entry is added, the map takes a step forward in resizing if a growth phase is underway.

Once the hash and target bucket are set, the next step is to search through the bucket slots and any overflow buckets. This determines whether the key already exists, allowing Go to decide whether to update an existing value or insert a new one.

### 3. Iterate Over the Buckets

This step mirrors what happens in `mapaccess`, where the top-hash of the key is compared to the top-hash in each slot (`b.tophash[i]`). If there's no match, Go checks if the slot is empty. If it is, and we're in insert mode, the slot becomes a candidate for adding the new key-value pair.

However, there are up to three states for empty slots. Go uses three specific top-hash values to indicate empty slots, but only two are valid for new entries:

- `evacuatedEmpty (4)` : The slot has already been migrated (during map growth) to a new bucket and should no longer be used for new entries.
- `emptyRest (0)` : Signals that there are no more non-empty slots beyond this point, allowing us to stop searching.
- `emptyOne (1)` : The slot is empty and available for new entries, but there might be more keys further along. So, we need to keep looping through the remaining slots to ensure we don't miss a potential match.

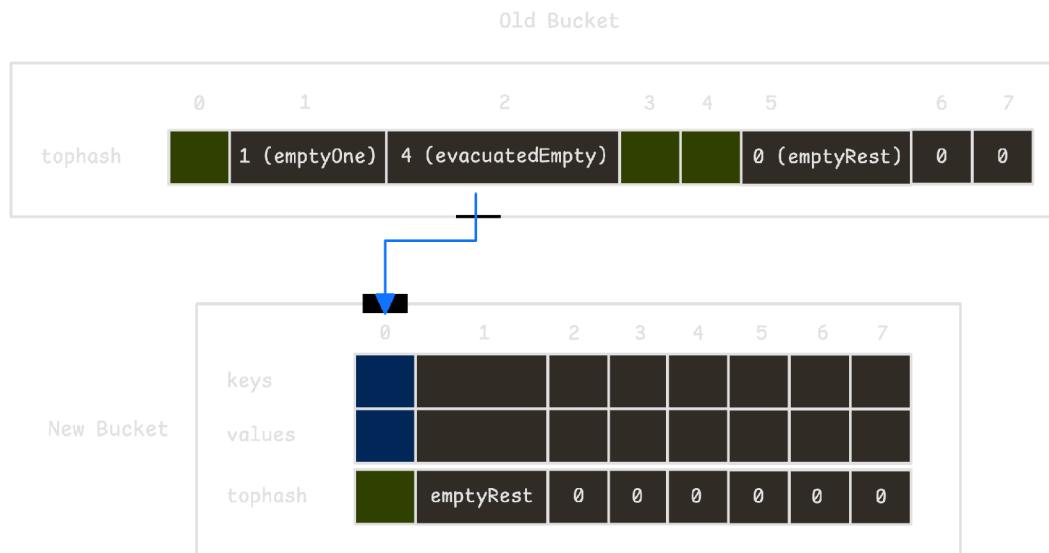


Illustration 42. Three empty states manage bucket entries

The Go runtime goes through eight bucket slots in both the target bucket and the overflow buckets to check the top-hash value:

Figure 44. mapassign: Map Write - Not Matched Slot  
(src/runtime/map.go)

```

func mapassign(t *maptYPE, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...
    for {
        for i := uintptr(0); i < bucketCnt; i++ {
            // Check the tophash
            if b.tophash[i] != top {
                if isEmpty(b.tophash[i]) &&
inserti == nil {
                    inserti =
&b.tophash[i]
                    insertk =
add(unsafe.Pointer(b), dataOffset+i*uintptr(t.KeySize))
elem =
add(unsafe.Pointer(b),
dataOffset+bucketCnt*uintptr(t.KeySize)+i*uintptr(t.ValueSiz
e))
                }
            if b.tophash[i] == emptyRest
{
                break bucketloop
            }
            continue
        }

        // Retrieve and double-check the key
using address arithmetic
        k := add(unsafe.Pointer(b),
dataOffset+i*uintptr(t.KeySize))
        if t.IndirectKey() {
            k = *((*unsafe.Pointer)(k))
        }
        if !t.Key.Equal(key, k) {
            continue
        }

        // Replace the key if necessary
        if t.NeedKeyUpdate() {
            typedmemmove(t.Key, k, key)
        }

        // Retrieve the value slot using
address arithmetic
        elem = add(unsafe.Pointer(b),
dataOffset+abi.MapBucketCount*uintptr(t.KeySize)+i*uintptr(t
.ValueSize))
        goto done
    }
}

```

```

        // Move to the overflow bucket
        ovf := b.overflow(t)
        if ovf == nil {
            break
        }
        b = ovf
    }
    ...
}

```

Still, we're not done yet. Adding a new entry might trigger map growth, depending on a few conditions:

- The map's load factor (how full the map is) exceeds its limit.
- Too many overflow buckets also indicate it's time to redistribute the map's key-value pairs.

When either of these conditions is met, the map grows, and the insertion process starts over from scratch:

Figure 45. map: Growing Check in mapassign (src/runtime/map.go)

```

func mapassign(t *maptypes, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...

    if !h.growing() && (overLoadFactor(h.count+1, h.B)
|| tooManyOverflowBuckets(h.overflow, h.B)) {
        hashGrow(t, h)
        goto again // Growing the table invalidates
everything, so try again
    }
}

```

When growth kicks in, the map resets, and the process jumps back to recalculating the hash, restarting the insertion from the beginning.

Back to the regular flow—if we find an empty slot, we're ready to add the new key-value pair. If there's no space left, Go creates an overflow bucket, linking it back to the target bucket to accommodate the new entry:

Figure 46. map: Map Write - Overflow Bucket (src/runtime/map.go)

```

func mapassign(t *maptypes, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ...
}

```

```

        // If there is no room in the current bucket,
        // allocate an overflow bucket and link back to the
target bucket
        if inserti == nil {
            newb := h.newoverflow(t, b)
            inserti = &newb.tophash[0]
            insertk = add(unsafe.Pointer(newb),
dataOffset)
            elem = add(insertk,
bucketCnt*uintptr(t.KeySize))
        }

        // Store new key/elemt at insert position
        if t.IndirectKey() {
            kmem := newobject(t.Key)
            (*unsafe.Pointer)(insertk) = kmem
            insertk = kmem
        }
        if t.IndirectElem() {
            vmem := newobject(t.Elem)
            (*unsafe.Pointer)(elem) = vmem
        }

        typedmemmove(t.Key, insertk, key)
        *inserti = top
        h.count++

done:
        if h.flags&hashWriting == 0 {
            fatal("concurrent map writes")
        }
        h.flags &^= hashWriting
        if t.IndirectElem() {
            elem = *((*unsafe.Pointer)(elem))
        }
        return elem
    }
}

```

And that's it! Once the key-value pair is inserted or updated, the final step is to check the concurrency flag again and clear it, signaling that the write operation is complete.

## Deletion and Slot Reclamation

To remove a key-value pair from a map in Go, use the built-in `delete` function:

```

func main() {
    m := make(map[string]int)
}

```

```

        m["key"] = 1
        delete(m, "key") // valid
        delete(m, "key") // still valid
    }
}

```

The `delete` function has a simple signature: `func delete(m map[Type]Type1, key Type)`. If the key exists, `delete` clears both the key and its associated value. If the key isn't present, it does nothing—no errors, no warnings.

Under the hood, `delete` works similarly to `mapassign`, following many of the same steps: calculating the hash, locating the bucket, and even handling growth if necessary. When the key is found, `delete` clears the key-value pair's memory and sets the slot's top-hash to `emptyOne`, marking it as an available slot for future entries.

Since `mapdelete` closely mirrors `mapassign`, we will only discuss the differences.

After a key-value pair is deleted, the cleanup isn't complete. If an `emptyOne` slot appears at the end of a bucket or overflow chain, it is converted to `emptyRest`:

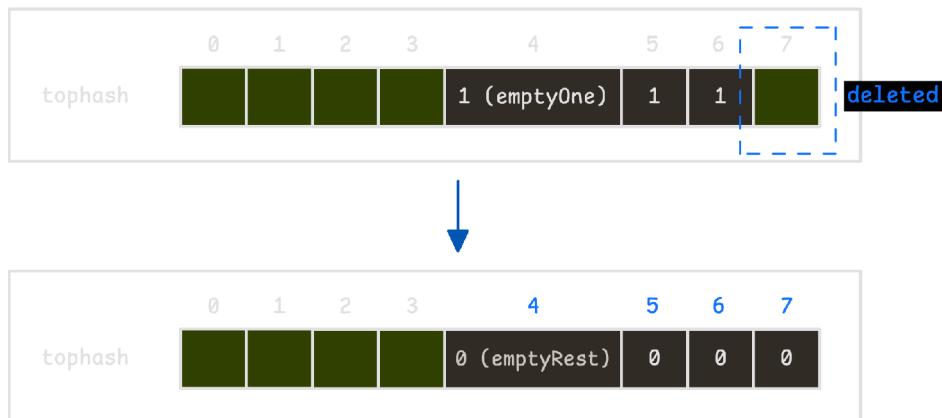


Illustration 43. Bucket cleanup converts emptyOne to emptyRest

This signals that not only is the current slot empty, but every slot after it is as well. Here's the code shows that process:

Figure 47. map: Found Key in mapdelete (src/runtime/map.go)

```

func mapdelete(t *maptype, h *hmap, key unsafe.Pointer) {
    search:
        for ; b != nil; b = b.overflow(t) {

```

```
for i := uintptr(0); i < abi.MapBucketCount;
i++ {
    ...
        // Convert trailing emptyOne slots
        // into emptyRest for cleanup
        if i == abi.MapBucketCount-1 {
            if b.overflow(t) != nil &&
b.overflow(t).tophash[0] != emptyRest {
                goto notLast
            }
        } else {
            if b.tophash[i+1] != emptyRest {
                goto notLast
            }
        }
    for {
        b.tophash[i] = emptyRest
        if i == 0 {
            if b == borig {
                break //
}
Reached the start of the initial bucket
}
// Move to the
previous bucket and continue cleanup
c := b
for b = borig;
b.overflow(t) != c; b = b.overflow(t) {
}
i =
abi.MapBucketCount - 1
} else {
    i--
}
if b.tophash[i] != emptyOne
{
    break
}
notLast:
h.count--
// Reset the hash seed when the map
is empty to prevent collision attacks
if h.count == 0 {
    h.hash0 = uint32(rand())
}
break search
}
}
```

## Growth Strategy and Rehashing

Overflow buckets offer a quick, temporary fix for storing extra items when a bucket is full, but they're bad for performance. The more overflow buckets a map has, the longer it takes to find a key-value pair since search time increases linearly with each overflow. Even worse, this slowdown affects every map operation—lookups, insertions, and deletions.

To fix this, the map needs to grow. Growing means creating more buckets and redistributing existing key-value pairs into them to balance the load. Sometimes, the map grows without increasing the total number of buckets—a process called **same-size grow**.

That said, growth is only triggered when adding new elements:

Figure 48. map: mapassign & hashGrow (src/runtime/map.go)

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer)
unsafe.Pointer {
    ... // Check the map state, calculate the hash, find
    the bucket, iterate over the buckets

    if !h.growing() && (overLoadFactor(h.count+1, h.B)
|| tooManyOverflowBuckets(h.overflow, h.B)) {
        hashGrow(t, h)
        goto again // Growing the table invalidates
everything, so try again
    }
    ...
}

func hashGrow(t *maptype, h *hmap) {
    ...
    bigger := uint8(1)
    if !overLoadFactor(h.count+1, h.B) {
        bigger = 0
        h.flags |= sameSizeGrow
    }
    oldbuckets := h.buckets
    newbuckets, nextOverflow := makeBucketArray(t,
h.B+bigger, nil)
    ...

    h.B += bigger // Increase the number of buckets
    h.flags = flags
    h.oldbuckets = oldbuckets // Store the old buckets
    h.buckets = newbuckets // Update to the new
```

```

buckets
    h.nevacuate = 0           // Reset evacuated bucket
count
    h.noverflow = 0           // Reset overflow bucket
count
}

```

Notice that maps don't shrink in Go; they either grow or stay the same. When elements are deleted using `delete()`, the count is decremented, and the slot is marked as empty, but the bucket array size (controlled by `h.B`) is never reduced.

There was an open issue "runtime: shrink map as elements are deleted" [\[shrinkmap\]](#) that still open until Go 1.23.

Growth doesn't happen all at once; it's a gradual process. Go resizes the map lazily, handling only what's necessary with each operation. There are two main conditions that trigger map growth:

1. **High load factor:** The map is too full, meaning there are too many key-value pairs for the available space.
2. **Too many overflow buckets:** A sign of poor distribution, leading to frequent collisions.

In the first case, growth adds more buckets. In the second, the map redistributes existing elements into the new buckets without increasing their number—a **same-size grow**.

The load factor determines when the map needs to grow. It's a ratio that measures how full the buckets are allowed to get before resizing begins. In Go, the threshold sits around 80%—if the map's buckets are more than 80% full on average, growth is triggered.

Figure 49. map: overLoadFactor (src/runtime/map.go)

```

const (
    // Maximum average load of a bucket that triggers
    // growth is bucketCnt*13/16 (about 80% full)
    // Because of minimum alignment rules, bucketCnt is
    // known to be at least 8.
    // Represented as loadFactorNum/loadFactorDen to
    // allow integer math.
    loadFactorDen = 2
    loadFactorNum = loadFactorDen * bucketCnt * 13 / 16
)

```

```

// overLoadFactor reports whether count items placed in 1<<B
buckets exceed the load factor.
func overLoadFactor(count int, B uint8) bool {
    return count > bucketCnt && uintptr(count) >
loadFactorNum*(bucketShift(B)/loadFactorDen)
}

// bucketShift returns 1<<b, optimized for code generation.
func bucketShift(b uint8) uintptr {
    return uintptr(1) << (b & (goarch.PtrSize*8 - 1))
}

```

The map doesn't consider growing until it has more elements than one bucket can handle (8 items per bucket). Once it hits that point, the total number of elements is compared against 80% of the maximum capacity for the current setup. If the count exceeds this threshold, the map grows.

Let's break down how this works with actual numbers:

### Buckets Max Elements in Buckets Growth Trigger (80%)

1	8	0
2	16	13
4	32	26
8	64	52
16	128	104
32	256	208
64	512	416
128	1024	832

## Buckets Max Elements in Buckets Growth Trigger (80%)

256	2048	1664
-----	------	------

512	4096	3328
-----	------	------

The first row is an exception. From the second row onward, the 'Growth Trigger' column shows the threshold where the map starts growing. For instance, a map with 2 buckets can hold up to 16 elements, but if it exceeds 13, it will expand to 4 buckets to maintain balance.

This ties back to our earlier example where we initialized a map with a hint of 14 elements:

```
m := make(map[string]int, 14)
```

Although 2 buckets can technically hold 14 elements, surpassing the 80% threshold ([13](#)) prompts the map to expand to 4 buckets for better distribution.

Even if a map isn't completely full, poor distribution can slow things down. When too many elements cluster in just a few buckets, lookups become slower since Go has to sift through more entries per bucket—far from the quick access expected from a well-balanced hash map.

Overflow buckets can also trigger growth. When too many of them accumulate, it signals that collisions are increasing and performance is likely degrading.

But how many overflow buckets are "too many"?

Overflow buckets become excessive when *their count matches or exceeds the number of main buckets*. To avoid runaway growth, this check is capped at 15 overflow buckets:

Figure 50. map: tooManyOverflowBuckets (src/runtime/map.go)

```
func tooManyOverflowBuckets(noverflow uint16, B uint8) bool
{
    if B > 15 {
        B = 15
    }
}
```

```
        return noverflow >= uint16(1) << (B & 15)
    }
```

When growth begins, the `buckets` field in `hmap` points to the new set of buckets, while `oldbuckets` keeps a reference to the old ones. The process unfolds gradually, managed by functions like `growWork()` and `evacuate()`. This lazy resizing spreads the workload across multiple operations, ensuring the map remains responsive and doesn't freeze during expansion.

Two main actions drive map growth: **assigning** to the map and **deleting** from it. Each time one of these operations occurs, the bucket involved is fully migrated to the new bucket array:

Figure 51. `growWork` (src/runtime/map.go)

```
{ // mapassign & mapdelete
    if h.growing() {
        growWork(t, h, bucket)
    }
}

func growWork(t *maptpe, h *hmap, bucket uintptr) {
    evacuate(t, h, bucket & h.oldbucketmask())

    if h.growing() {
        evacuate(t, h, h.nevacuate)
    }
}
```

Each unit of work handles two types of bucket evacuations:

1. **Targeted evacuation:** Focuses on the current bucket. If the key "dog" is lying in bucket 3, then the targeted evacuation will be bucket 3.
2. **Steady evacuation:** Keeps the growth moving forward, evacuating at least one additional old bucket—even if the targeted one has already been evacuated.

All elements in the chosen bucket, including any in its overflow buckets, are moved to new locations.

## 1. Same-size Grow

Over time, repeated insertions and deletions can lead to overflow buckets piling up due to hash collisions. Even if items are removed through deletion, those overflow buckets remain, consuming memory and reducing efficiency.

To resolve this, instead of deleting overflow buckets directly—avoiding corruption or peak performance degradation—Go takes advantage of the growth process in a special way called same-size grow:

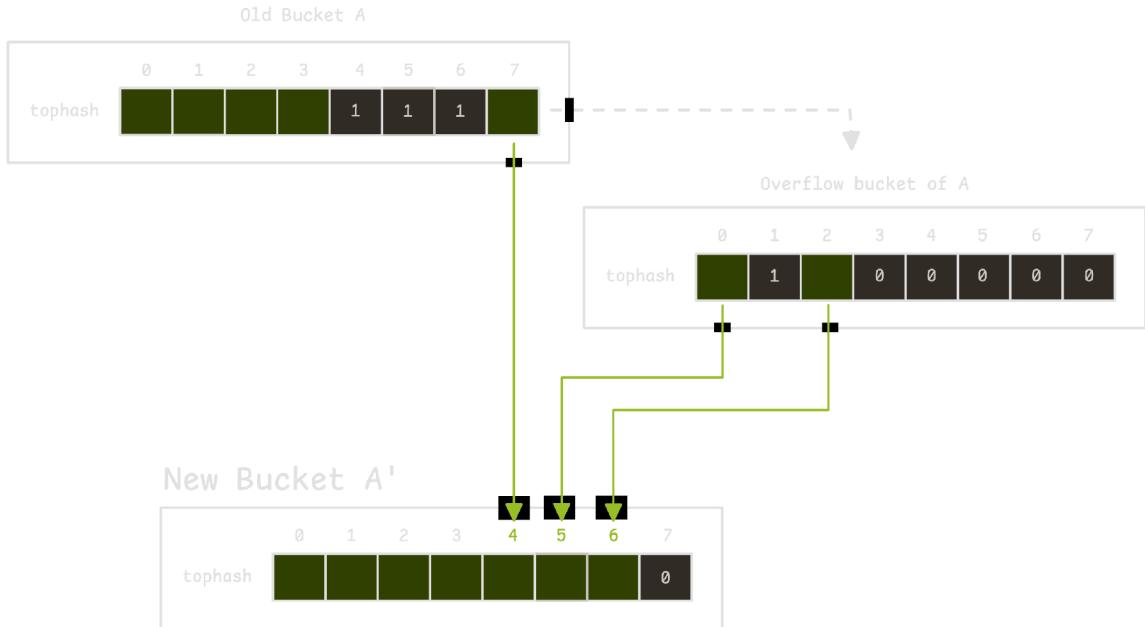


Illustration 44. Same-size grow redistributes map bucket entries

This process redistributes elements into a clean set of buckets, leaving the old overflow buckets behind. The garbage collector then clears out these abandoned buckets. This approach cleans up the map without breaking iteration or adding unnecessary complexity.

## 2. Double-Size Grow

When a map doubles in size, all existing elements must move from the old buckets to the new ones. This might seem like every key needs to be rehashed and reinserted from scratch—but that's not the case.

Since the hash function and seed remain the same, each entry from the old map has only **two possible destinations** in the larger map. This significantly simplifies redistribution.

Imagine a map with 4 buckets that grows to 8 buckets:



Illustration 45. Keys redistributed between two potential buckets

With 4 buckets, only the last 2 bits of the hash value are used to determine where each key belongs. For example, a hash ending in  $11_2$  would land in bucket 3 (since  $11_2$  equals 3 in decimal). When the map doubles to 8 buckets, Go starts using the last 3 bits of the hash value instead of 2.

- If the additional bit is  $0$ , the key stays in its original bucket.
- If the additional bit is  $1$ , the key moves to a new bucket that is offset by the size of the old map (in this case, bucket 7).

So, a key that initially landed in bucket 3 now has two potential destinations: it either stays in bucket 3 or moves to bucket 7, depending on that extra bit:

Figure 52. 2 Possible New Buckets (src/runtime/map.go)

```
func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
    newbit := h.noldbuckets() // Number of old buckets
    in the new map

    ...
    x.b = (*bmap)(add(h.buckets,
    oldbucket*uintptr(t.BucketSize)))
    x.k = add(unsafe.Pointer(x.b), dataOffset)
    x.e = add(x.k, bucketCnt*uintptr(t.KeySize))

    ...
    y.b = (*bmap)(add(h.buckets,
    (oldbucket+newbit)*uintptr(t.BucketSize)))
    y.k = add(unsafe.Pointer(y.b), dataOffset)
    y.e = add(y.k, bucketCnt*uintptr(t.KeySize))
}
```

This diagram illustrates how elements are redistributed when the map grows from 4 to 8 buckets:

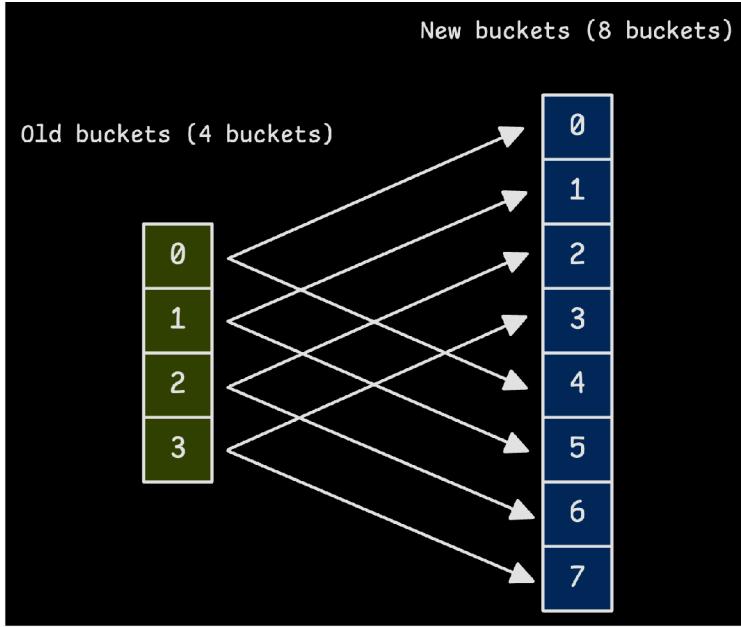


Illustration 46. Each old bucket maps to two new buckets

Once the new locations are determined, Go begins moving the elements. It iterates through the old bucket and its overflow buckets, relocates each key-value pair, and marks the old buckets as evacuated.

The logic behind this redistribution gets more intricate in the actual implementation:

Figure 53. evacuate - Moving (src/runtime/map.go)

```
func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
    for ; b != nil; b = b.overflow(t) {
        for i := 0; i < bucketCnt; i, k, e = i+1,
add(k, uintptr(t.KeySize)), add(e, uintptr(t.ValueSize)) {
            ...
            // dst is the destination { b
            (bucket), i (slot index), k (key), e (element/value) }
            // If dst reaches the end of the
            bucket, create a new overflow bucket
            if dst.i == bucketCnt {
                dst.b = h.newoverflow(t,
dst.b)
                dst.i = 0
                dst.k =
            add(unsafe.Pointer(dst.b), dataOffset)
                dst.e = add(dst.k,
bucketCnt*uintptr(t.KeySize))
            }
        }
    }
}
```

```

        // Move the key to the new bucket
        dst.b.tophash[dst.i&(bucketCnt-1)] =
    top
        if t.IndirectKey() {
            (*unsafe.Pointer)(dst.k) =
    k2
        } else {
            typedmemmove(t.Key, dst.k,
    k)
        }

        // Move the value to the new bucket
        if t.IndirectElem() {
            (*unsafe.Pointer)(dst.e) =
    e)
        } else {
            typedmemmove(t.Elem, dst.e,
    e)

        // Prepare for the next slot
        dst.i++
        dst.k = add(dst.k,
        dst.e = add(dst.e,
    uintptr(t.KeySize))
    uintptr(t.ValueSize))
    }
    ...
}

}

```

Go loops through the old bucket and any overflow buckets, using three pointers—`i` (slot index), `k` (key), and `e` (element/value)—to keep track of the current position. A `dst` variable manages the destination bucket and slot for each entry. If the destination bucket fills up, a new overflow bucket is created, and elements continue filling from there.

Depending on how the keys and values are stored—directly or indirectly as pointers—Go either moves the actual data with `typedmemmove` or copies the pointer.

Once all elements are relocated, the old buckets are marked as evacuated and left for Go's garbage collector to clean up, freeing memory and ensuring efficient map performance.

## For-Range Loop Over a Map

One of the more intriguing aspects of Go's maps is how iteration works. Specifically, the order in which elements appear during a `for-range` loop isn't fixed—it changes from one loop to the next.

Using `for-range` to iterate over a map is straightforward and feels natural:

```
func main() {
    m := make(map[string]int)

    m["key1"] = 1
    m["key2"] = 2
    m["key3"] = 3
    m["key4"] = 4
    m["key5"] = 5

    for k, v := range m {
        fmt.Println(k, v)
    }
}

// Example output:
// key4 4
// key5 5
// key1 1
// key2 2
// key3 3
```

The iteration order isn't tied to the sequence of insertion. However, the output isn't entirely random either—it follows a pattern.

Each time you print the map, iteration begins at what seems like a random point and continues in the order of insertion, wrapping around at the end. You might see output like `2 3 4 5 1` or `3 4 5 1 2`. This happens because, in this example, the map has only one bucket under the hood.

## How Iteration Works Internally

Go uses an iterator to manage the current position as it traverses the map's hash table structure. Because of how hashing works, maps aren't laid out in a simple, linear order—so iterating over them doesn't follow a straightforward path.

Adding to the complexity, Go allows modifications to a map while iterating over it—as long as those changes happen within the same goroutine.

Note that "*Go allows modifications to a map while iterating over it*" does not mean you can modify the map from another goroutine. If you do, you'll encounter a fatal error. The following example demonstrates a valid modification:

```
func main() {
    m := make(map[string]int)

    for k, v := range m {
        m["key6"] = 6 // Valid, since it's in the
same goroutine
    }
}
```

The `for-range` loop over a map consists of two phases:

1. **Initialization:** The runtime sets up the iteration, determining the starting point and other details.
2. **Iteration:** The runtime steps through the map, slot by slot, updating the iterator as it progresses.

## 1. Initialization Phase

When iteration starts, Go initializes an iterator represented by the `hiter` struct:

Figure 54. map: hiter (src/runtime/map.go)

```
type hiter struct {
    key         unsafe.Pointer // Pointer to the
current key
    elem        unsafe.Pointer // Pointer to the
current value
    t           *maptype      // Map type, containing
info about key and value types
    h           *hmap         // Reference to the map
being iterated over
    buckets     unsafe.Pointer // Pointer to the map's
bucket array
    bptr        *bmap         // Pointer to the
current bucket being processed
    overflow    *[]*bmap      // Overflow buckets for
the map
    oldoverflow *[]*bmap      // Overflow buckets from
a previous map grow
    ...
}
```

The iterator captures the map's current structure, including the number of buckets and a pointer to them. This isn't a deep copy—it's more like bookmarking the map's layout and status at the time the loop starts.

The iteration doesn't begin at bucket 0 and move sequentially. Instead, it starts at a **random bucket** and a **random slot** within that bucket. This randomness discourages developers from relying on predictable iteration order and reinforces that map iteration in Go is inherently non-deterministic.

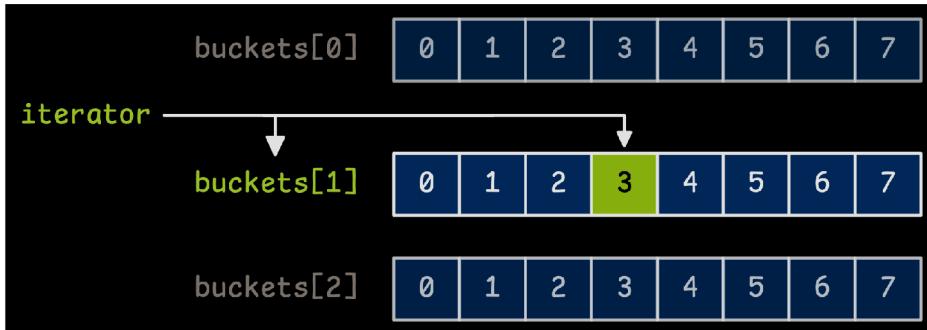


Illustration 47. Iteration starts at a random bucket and slot

This initialization is handled by the `mapiterinit` function in the `runtime` package:

Figure 55. `mapiterinit` (`src/runtime/map.go`)

```
func mapiterinit(t *maptype, h *hmap, it *hiter) {
    ...
    // Take a snapshot of the map's state
    it.B = h.B
    it.buckets = h.buckets
    if t.Bucket.PtrBytes == 0 {
        h.createOverflow()
        it.overflow = h.extra.overflow
        it.oldoverflow = h.extra.oldoverflow
    }

    // Start at a random bucket and random slot within
    // that bucket
    r := uintptr(rand())
    it.startBucket = r & bucketMask(h.B) // Random
    bucket position
    it.offset = uint8(r >> h.B & (bucketCnt - 1)) // Random
    slot in the bucket

    ...
}
```

With more buckets, randomness spreads across a larger space, making the iteration order appear even less structured.

## 2. Iteration Phase

As mentioned before, Go maps aren't safe for concurrent use. You can have multiple readers or run multiple `for-range` loops on the same map simultaneously. But the moment you introduce writes, you need external synchronization to prevent race conditions.

Iteration starts at a random slot within a randomly selected bucket and scans through all non-empty slots in that bucket. Once finished, it moves to the next bucket (`bucket++`), wrapping around when necessary. Interestingly, even when it moves to a new bucket, the iteration doesn't begin at the start of that bucket—it continues from the same random slot as the initial bucket.

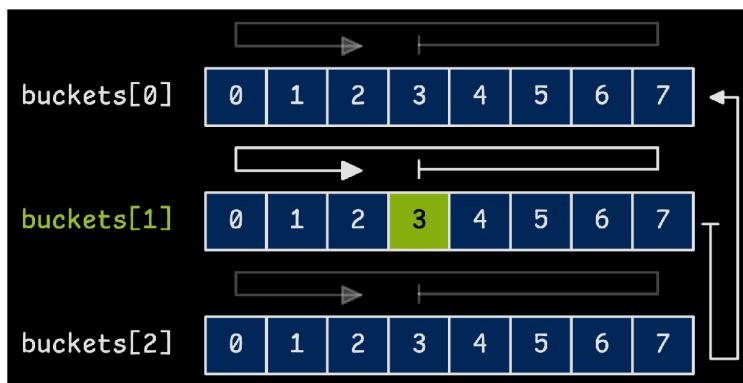


Illustration 48. Each bucket iterates from the same random slot

One of the more impressive aspects of Go's map iteration is how it handles resizing during iteration. Even if the map grows mid-loop, the iterator keeps track of both the old and new bucket arrays to ensure no elements are skipped or visited twice.

This ties back to the evacuation process. When a map grows and elements from an old bucket redistribute into two new buckets, the original bucket index can still be determined using a modulo operation with the old bucket count.

For example, when a map grows from 4 to 8 buckets:

- Old bucket 3 maps to buckets 3 and 7 in the new array.
- Using `bucket % oldBucketCount`, both 3 (`3 % 4`) and 7 (`7 % 4`) point back to old bucket 3.

If the iterator lands on a bucket that hasn't finished evacuating its elements, it falls back to the corresponding old bucket. It then filters out entries, processing only those meant for that specific new bucket. For instance, when iterating over new bucket 7, the iterator only picks elements from the old bucket that belong in bucket 7 and ignores those meant for bucket 3.

## Experiment: Map Modifications During Iteration

Now, since Go allows map modifications during iteration within the same goroutine, what happens if you insert new key-value pairs inside the loop? Let's put that to the test:

```
func main() {
    m := map[int]int{0: 1}

    for i := range m {
        fmt.Println(i)

        for k := range 100 {
            m[k] = k + 1
        }
    }
}
```

In this example, the map starts with a single entry `{0: 1}`, and during each iteration, 100 new key-value pairs are added. What's the result? How many lines get printed? Just one? Or does it run into an infinite loop?

When the loop starts, the iterator takes a snapshot of the map's current state, capturing the number of buckets (`h.B`) and a pointer to the bucket array (`h.buckets`). Initially, the map has one bucket with 8 slots, but only one slot is occupied. As the loop runs and new elements are added, the bucket fills up and could even trigger a resize. However, since the iterator snapshot only captures the current state of the buckets, new elements added to newly allocated buckets won't be part of the current loop.

The output depends on where the iterator begins. There are two main scenarios:

- The iterator starts at **slot 0**: it prints 8 lines. After processing slot 0, new elements fill the remaining slots in the same bucket, and the iterator continues through them.
- The iterator starts at a **different slot**: it prints just 1 line. Let's say the iterator starts at slot 1—since there's nothing in this slot, it runs through the

bucket, wraps back to slot 0, prints that, and stops. Even though the bucket fills up after printing slot 0, the loop ends because slot 1 has already been visited.

The second scenario is illustrated below:

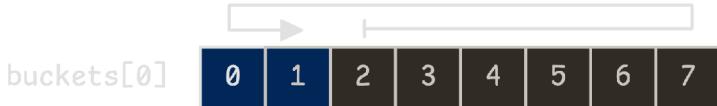


Illustration 49. If the iterator starts at slot 1

And that wraps up the iteration phase—and with it, our deep dive into how maps work in Go. From storage and resizing to iteration mechanics, we've covered the full picture of what makes Go's map implementation efficient, yet flexible enough to handle dynamic changes on the fly.

## 5. Summary

### Arrays

When you create an array in Go, you're setting aside a continuous block of memory where all elements are stored side by side. The size of the array becomes part of its type, meaning `[5]int` and `[3]int` are considered entirely different types.

Go uses different strategies to initialize arrays depending on their size: - **Small arrays** (up to 79 bytes) are zeroed out directly using simple instructions. - **Medium arrays** (80-1039 bytes) use an optimization known as **Duff's Device**, which speeds up zeroing memory by unrolling loops. - **Large arrays** (1040+ bytes) fall back on a straightforward loop to zero out the memory.

You can initialize arrays in several ways: by explicitly setting values (`[3]int{1, 2, 3}`) or letting Go infer the size (`[...]int{1, 2, 3}`). Under the hood, Go first zeroes out the memory, then assigns the specified values. For larger arrays, Go uses optimizations like combining multiple assignments into a single operation or copying from a pre-initialized memory template.

One important detail: when you pass an array to a function, assign it to another variable, or iterate over it with a `for-range` loop, Go copies the entire array.

This behavior differs from other types like slices and can impact performance with large arrays.

An array's length and capacity are always the same and are determined at compile time. Because of this, `len(a)` and `cap(a)` are constants and get embedded directly into the compiled code.

## Slices

When you create a slice, Go sets up a three-part structure and allocates an underlying array to hold the data.

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

The clever part is that multiple slices can share the same underlying array. When you create a new slice from an existing one, you're just creating a new view into the same block of memory. Changes to the shared elements will reflect across all slices that share that section of the array.

Things get interesting with the `append` function. If there's room in the underlying array, Go simply adds the new elements and updates the slice's length. But when the capacity is exceeded, Go creates a new, larger array, copies the existing elements over, and points the slice to this new array. The growth pattern isn't always a straightforward doubling—it depends on the current capacity and the type of elements being stored.

For smaller slices (up to 256 elements), Go typically doubles the capacity. Beyond that, it grows more conservatively, balancing memory usage with performance. The type of elements can also influence growth behavior because Go's memory allocator groups allocations into size classes that handle certain data types more efficiently.

Memory allocation for slices follows specific rules. If the slice size can be determined at compile time and is smaller than 64KB, the slice can be allocated on the stack. Larger slices are moved to the heap. When creating a slice that will immediately be filled with data, Go optimizes by skipping the initial zeroing of memory—unless the slice contains pointers, in which case zeroing is required for garbage collection safety.

## Strings

Under the hood, strings in Go are immutable, read-only slices of bytes. Each string consists of two parts: a pointer to the memory location where the bytes are stored and the length of the string. This simple structure always takes up 16 bytes, no matter how long the string is.

All strings in Go are encoded in UTF-8. Basic ASCII characters, like standard English letters, use just one byte each. More complex characters—such as Chinese symbols or emojis—can take up to 4 bytes. This variable-width encoding efficiently supports a wide range of characters without wasting memory.

Strings in Go are immutable, meaning once a string is created, it can't be changed. Assigning a string to another variable doesn't copy the actual bytes—it just copies the pointer and length. This makes string operations efficient since no extra memory is allocated. However, any operation that would modify a string instead creates a new one.

Go also includes smart optimizations for handling strings. When converting small byte slices (up to 32 bytes) into strings, Go uses a pre-allocated stack buffer instead of allocating heap memory. It even maintains a pre-made array of all possible single-byte values for faster conversions.

String concatenation is optimized too. For combining up to five strings, Go uses specialized helper functions. When dealing with more than five, it builds a slice of strings and merges them in one pass. If the concatenation is temporary—like in a comparison—Go often avoids memory allocation by using a small buffer on the stack.

The `for-range` loop handles UTF-8 complexity automatically by iterating over complete characters, known as runes, instead of raw bytes. This simplifies text processing and ensures correct handling of multi-byte characters.

## Maps

A map is essentially a pointer to a complex structure called `hmap`, which handles storage and operations behind the scenes. Internally, Go uses an array of buckets, with the total number of buckets always being a power of two (1, 2, 4, 8, 16, and so on). Each bucket can store up to 8 key-value pairs. When a bucket fills up, Go creates overflow buckets to handle the extra data temporarily.

When you insert a key-value pair, Go hashes the key to determine the correct bucket. If multiple keys hash to the same bucket—a situation called a collision—they share that bucket. Each bucket also stores a small fragment of the key’s hash, known as the **tophash**, which speeds up searches by allowing quick comparisons before checking full keys.

For larger keys or values (over 128 bytes), Go doesn’t store them directly in the bucket. Instead, it stores pointers to those values, keeping bucket sizes manageable and improving lookup performance.

When a map needs to grow, Go doesn’t move everything at once. Instead, it creates a new, larger array of buckets and gradually migrates data over time. This process, known as **gradual migration**, spreads out the workload across multiple operations, preventing sudden performance hits.

Go maps are not safe for concurrent access. If one goroutine writes to a map while another reads from it simultaneously, Go will detect the conflict and stop the program.

Map iteration order is intentionally randomized. When you loop through a map, Go takes a snapshot of its current state. Changes made during iteration won’t be visible until the next loop begins, ensuring consistent and predictable behavior across iterations.

## References

- [mapiter] runtime: map lookup of non-comparable value doesn’t panic:  
<https://github.com/golang/go/issues/23734>
- [shrinkmap] runtime: shrink map as elements are deleted:  
<https://github.com/golang/go/issues/20135>

# Chapter 4: Structs, Interfaces and Generics

## 1. Structs: Composition, Layout, and Performance

Unlike arrays, which store multiple values of the same type, structs allow you to combine different types of data into a single unit. While both arrays and structs store data in a contiguous block of memory, structs are specifically designed to handle mixed data types.

If you compare how arrays and structs are laid out in memory, their structure is surprisingly similar. In both cases, data is arranged linearly—one block after another:

```
type Struct struct {
    First  int
    Second bool
    Third  string
}
```

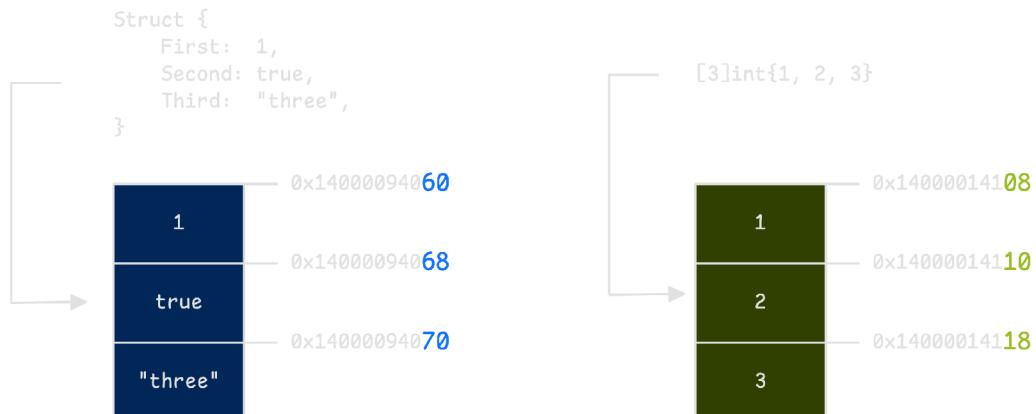


Illustration 50. Memory layout of structs vs arrays

Looking at the memory layout of the `Struct` above, something unexpected happens. The field `Second bool` starts at memory address `0x14000094068`. Some might expect the next field to begin right after, at `0x14000094069`. Instead, it jumps to `0x14000094070`. This leaves a 7-byte gap after the 1-byte `bool` field:



Illustration 51. Memory layout showing a 7-byte gap after a 1-byte `bool` field

This happens because of memory **alignment** and **padding**. Go aligns data in memory to improve performance. It sometimes adds unused space, called padding, to make access more efficient.

We will explain why this gap appears in the section on memory alignment and padding. But first, let's look at how structs are declared and used.

## 1.1 Introduction to Structs

### Declaring Struct Types

Defining a struct in Go begins with the `type` keyword. You assign a name to the struct, use curly braces, and list its fields inside. Each field has a name and a type:

```
type Book struct {
    ID      int
    Title   string
    Author  string
}
```

There is also a less common but valid way to declare multiple fields on a single line by separating them with semicolons:

```
type Book struct {
    ID      int; Title string
    Author  string `tagKey:"tagValue"`
}
```

While this syntax is valid, it is generally discouraged for readability. This is where `go fmt` (Go format) helps. It automatically formats your code for better clarity. In practice, placing each field on its own line makes the code easier to read and maintain.

Notice the use of struct tags in this example (`tagKey:"tagValue"`). Struct tags are string literals attached to fields, like the one next to `Author`. They act as metadata and can be used by your Go code or by external tools. This enables extra behavior such as encoding, validation, or database mapping.

A common use case for tags is in serialization, where data is converted between different formats. Tags guide how a struct's fields are encoded or decoded when using libraries such as `encoding/json`:

```
type User struct {
    Username string `json:"custom_username"`
    Age      int    `json:"age"`
}

// {"custom_username": "Bob", "age": 18}
```

The `json` tag is not a special keyword; it is simply a convention recognized by the `encoding/json` package. You can define custom tags to support other use cases as needed.

Go's `reflect` package allows you to inspect and manipulate types at runtime, including reading struct tags:

```
type Person struct {
    Name string `json:"name" xml:"nameElement"`
    Age  int    `json:"age,omitempty"`
}

func main() {
    p := Person{Name: "John Doe", Age: 30}
    t := reflect.TypeOf(p)

    for i := 0; i < t.NumField(); i++ {
        field := t.Field(i)
        jsonTag := field.Tag.Get("json")
        xmlTag := field.Tag.Get("xml")
        fmt.Printf("%q field: json tag: %q, xml tag: %q\n",
            field.Name, jsonTag, xmlTag)
    }
}
```

```
// Output:  
// "Name" field: json tag: "name", xml tag: "nameElement"  
// "Age" field: json tag: "age,omitempty", xml tag: ""
```

This example shows how the `encoding/json` package uses tags for serialization, but the same principle can be applied to any feature that benefits from runtime metadata.

## Initializing Structs Using Composite Literals

Now that we've covered how structs are defined, let's look at how to initialize them using composite literals. This shorthand syntax allows you to assign values directly to a struct at the point of creation.

There are two common approaches:

- Provide values in the exact order the fields appear in the struct.
- Name each field explicitly when assigning values.

Naming fields is particularly useful when working with larger structs or when you want to set only a few fields and let the rest default to their zero values:

```
a := Book{ID: 100, Title: "Great Expectations", Author:  
"Charles Dickens"}  
b := Book{Title: "The Anatomy of Go"}
```

In the first example, all fields of the `Book` struct are assigned values and stored in variable `a`. When fields are named explicitly, their order does not matter—Go matches values based on field names. In the second example, only the `Title` field is set; the others are left at their default zero values.

If you prefer a more concise style, you can omit field names and list values in the exact order the fields are defined. This is known as position-based initialization:

```
b := Book{100, "Great Expectations", "Charles Dickens"}
```

This approach is more compact, but it has some potential pitfalls. It works best for small structs with a clear and fixed field order. As structs get larger or include several fields of the same type, it becomes easier to assign values incorrectly. For example, you might accidentally swap two `int` fields.

That said, position-based initialization can be practical in specific cases, such as:

- Small structs with obvious field order, like `Point{X, Y}` or `Pair{Key, Value}`.
- Table-driven tests, where creating many struct instances quickly without repeating field names helps keep the code brief.

Here is an example of a table-driven test for a simple `Add` function:

```
func TestAdd(t *testing.T) {
    // Define a slice of test cases
    var tests = []struct {
        input1 int
        input2 int
        expected int
    }{
        {1, 1, 2},
        {2, 3, 5},
        {10, -5, 5},
        {0, 0, 0},
    }

    // Iterate over test cases
    for _, tt := range tests {
        result := Add(tt.input1, tt.input2)
        if result != tt.expected {
            t.Errorf("Add(%d, %d) = %d; want %d", tt.input1,
tt.input2, result, tt.expected)
        }
    }
}
```

Now, how does Go prepare memory and make sure everything is initialized correctly when using a composite literal?

You can get the idea by recalling how arrays were explained in Chapter 3. The Go compiler might use direct instructions, loop unrolling, or simple loops to set up memory. Structs follow a similar idea. Since they use a single block of memory, the compiler generates instructions to assign values to each field or to set them to zero if no value is provided.

## Anonymous Structs and Their Use Cases

Anonymous structs are a flexible feature in Go that let you define a struct inline without assigning it a formal name. They are especially useful when you need to group related data for short-term use, without creating a full type definition:

```

type Person struct {
    Name      string
    SelectedBook struct {
        Title  string `json:"title"`
        Author string `json:"author"`
    }
}

func main() {
    a := struct {
        ID    int   `json:"id"`
        Title string `json:"title"`
    }{ID: 100, Title: "Great Expectations"}

    ...
}

```

In the `Person` struct, the `SelectedBook` field is an anonymous struct, defined directly within the parent type. In the `main` function, another anonymous struct is both declared and initialized at the same time.

Although they may seem unnecessary at first glance, anonymous structs are quite practical in specific situations. A common use case is table-driven testing, where temporary struct definitions help avoid cluttering your code with additional named types:

```

func TestAdd(t *testing.T) {
    tests := []struct {
        name      string
        a, b      int
        expected int
    }{
        {name: "Add 1 and 2", a: 1, b: 2, expected: 3},
        {name: "Add 3 and 4", a: 3, b: 4, expected: 7},
        {name: "Add 5 and 6", a: 5, b: 6, expected: 11},
    }
    ...
}

```

Anonymous structs are also effective when working with JSON. They allow you to build temporary structures for marshaling or unmarshaling data without modifying existing types:

```

type User struct {
    Name      string `json:"name"`
    Email     string `json:"email"`
    Password  string `json:"password"`
}

func (u User) MarshalJSON() ([]byte, error) {
    return json.Marshal(&struct {
        Name      string `json:"name"`
        Email     string `json:"email"`
    }{
        Name: u.Name,
        Email: u.Email,
    })
}

```

In this example, a custom `MarshalJSON` method is used to control how the `User` struct is converted to JSON. It creates an anonymous struct with only the fields that should be included. This allows the `Password` field to be left out of the JSON output without changing the original `User` type.

Internally, anonymous structs behave just like named ones. They are blocks of memory with fields laid out according to their types. The main limitation is that you cannot define methods on an anonymous struct.

For example, the following will not compile:

```

// INVALID
func (s struct{}) Print() {
    fmt.Println("Hello, World!")
}

```

Another benefit of anonymous structs is that they can be compared directly. If two structs—whether named or anonymous—have the same field types, order, and tags, they can be compared without issue:

```

type Person struct {
    Name string `json:"name" xml:"nameElement"`
    Age  int
}

func main() {
    a := struct {
        Name string `json:"name" xml:"nameElement"`
        Age  int
    }()
}

```

```

    b := struct {
        Name string `json:"name" xml:"nameElement"`
        Age  int
    }{}

    fmt.Println(a == b)          // true
    fmt.Println(a == Person{}) // true
}

```

As long as the structure, field types, and tags match, Go allows the structs to be compared and treats them as equivalent.

## Accessing Struct Fields

Struct fields are accessed using dot (.) notation, whether you are reading or updating values:

```

type User struct {
    ID  int
    Name string
}

func main() {
    b1 := User{ID: 100, Name: "John"}
    b2 := &User{ID: 101, Name: "Doe"}

    b1.Name = "Jane"
    fmt.Println(b1.Name) // Output: Jane
    fmt.Println(b2.Name) // Output: Doe
}

```

The compiler knows the exact location of each field within a struct. It calculates the offset of each field from the beginning of the struct in memory:

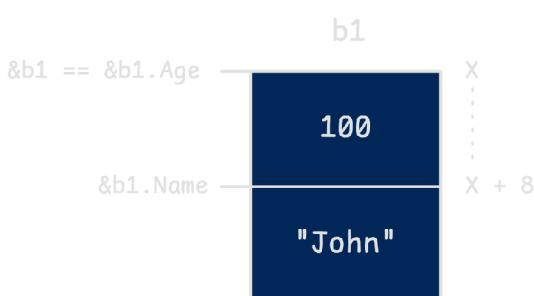


Illustration 52. Struct field memory offsets in Go

When you access a field using dot notation, Go computes the field's offset from the base address of the struct. Take `b2` for example, which is a pointer to a struct. You might expect to dereference it manually, like `(*b2).ID`.

While that syntax is valid, Go handles the dereferencing automatically. The compiler detects that `b2` is a pointer and transparently resolves the field access. This means you can simply write `b2.ID`, and Go will perform the necessary pointer dereference. That's why if `b2` is `nil`, trying to access a field will cause a nil pointer dereference at runtime.

Here's how the compiler handles field access during type checking stage:

Figure 56. DotField (src/cmd/compile/internal/typecheck.go)

```
// DotField returns a field selector expression that selects
// the
// index'th field of the given expression, which must be of
// struct or
// pointer-to-struct type.
func DotField(pos src.XPos, x ir.Node, index int)
*ir.SelectorExpr {
    op, typ := ir.ODOT, x.Type()
    if typ.IsPtr() {
        op, typ = ir.ODOPTPTR, typ.Elem()
    }
    if !typ.IsStruct() {
        base.FatalfAt(pos, "DotField of non-struct:
%L", x)
    }

    types.CalcSize(typ)

    field := typ.Field(index)
    return dot(pos, field.Type, op, x, field)
}
```

When the compiler recognizes a pointer to a struct, it changes the operation from a regular dot node (`ODOT`) to a dot pointer node (`ODOPTPTR`). This distinction is important in the code generation phase, where the abstract operation is translated into lower-level instructions:

Figure 57. exprCheckPtr (src/cmd/compile/internal/ssagen/ssa.go)

```
func (s *state) exprCheckPtr(n ir.Node, checkPtrOK bool)
*ssa.Value {
    ...
}
```

```

        switch n.Op() {
        ...
        case ir.ODOTPTR:
            n := n.(*ir.SelectorExpr)

            // Get the pointer value
            p := s.exprPtr(n.X, n.Bounded(), n.Pos())
            // Add the field's offset to the pointer
            p = s.newValue1I(ssa.OpOffPtr,
types.NewPtr(n.Type(), n.Offset(), p)
            // Load the value from memory (dereference)
            return s.load(n.Type(), p)
        }
        ...
    }
}

```

Here, the compiler retrieves the pointer value, adds the field's offset, and then loads the data from memory by dereferencing the computed address.

You may notice through practice that you cannot index a pointer to a slice or a map. Only pointers to arrays are supported:

```

func main() {
    var a *[5]int
    var b *[]int
    var c *map[string]int

    fmt.Println(a[2])    // valid
    fmt.Println(b[2])    // invalid operation: cannot
index b (variable of type *[]int)
    fmt.Println(c["a"]) // invalid operation: cannot
index c (variable of type *map[string]int)
}

```

This behavior exists because slices and maps are fundamentally different from simple, contiguous memory types like arrays and structs. They already include indirection as part of their design. Although slices are not pointers themselves, they are backed by a descriptor that contains a pointer to an underlying array, along with length and capacity information.

If Go allowed indexing a pointer to a slice, the compiler would need to perform two levels of dereferencing: first to access the slice header (`data`, `len`, `cap`), and second to retrieve the element from the `data` pointer. The same logic applies to maps, which internally reference a hash table (`hmap`).

Allowing this kind of syntax would add extra complexity to the language. It would introduce another layer of indirection without providing a clear benefit, making the behavior of slices and maps less predictable and harder to reason about.

## 1.2 Advanced Struct Concepts

### Struct Comparability and Equality Semantics

Go provides a clear set of types that are inherently comparable: integers, floats, strings, pointers, channels, arrays, and structs. Slices, maps, and functions are not comparable.

However, if a struct or array contains any of these non-comparable types, it becomes non-comparable and cannot be used with `==` or `!=`.

That said, for a struct to be comparable, all of its fields must themselves be comparable:

```
type Person struct {
    Emails []string
}

func main() {
    a := Person{}
    b := Person{}

    fmt.Println(a == b)
}

// compile error: invalid operation: a == b (struct
containing []string cannot be compared)
```

Even if two structs look the same, they cannot always be compared. This is especially true for named structs with identical fields but different type names:

```
type Person struct {
    Name string
}

type Employee struct {
    Name string
}

func main() {
```

```

    p1 := Person{Name: "John"}
    p2 := Person{Name: "Doe"}
    e := Employee{Name: "John"}

    fmt.Println(p1 == p2) // Output: false
    fmt.Println(p1 == e) // invalid operation: p1 == e
(mismatched types Person and Employee)
}

```

Although `Person` and `Employee` share the same field and type, Go treats them as entirely separate types.

The type names are significant. Having identical fields is not enough if the type definitions are named differently. This contrasts with anonymous structs, where comparison is based solely on structure.

Anonymous structs can be compared to named structs if their fields match exactly in type, name, and order:

```

type Person struct {
    Name string
}

type Employee struct {
    Name string
}

func main() {
    p1 := Person{Name: "John"}
    p2 := Employee{Name: "Doe"}
    e := struct{ Name string }{Name: "John"} // Anonymous struct

    fmt.Println(p1 == e) // Output: true
    fmt.Println(p2 == e) // Output: false
    fmt.Println(p1 == p2) // invalid operation: p1 == p2
(mismatched types Person and Employee)
}

```

This behavior can feel unintuitive. The anonymous struct `e` can be compared with both `Person` and `Employee` as long as the fields match exactly. However, named structs like `Person` and `Employee` cannot be compared directly, as Go enforces strict type boundaries based on names.

Here's an interesting question: if two structs are comparable, can you assign one to the other? It's a useful point to pause and consider.

According to the Go specification:

"In any comparison, the first operand must be assignable to the type of the second operand, or vice versa."

## ~ The Go Programming Language Specification

```
a := struct{ Name string }{Name: "John"}  
b := struct{ Name string }{Name: "Doe"}  
  
a = b // This works just fine
```

This assignment works because the structure of the two values is identical. When comparing or assigning unnamed structs, the compiler checks that the field names, types, order, and tags are all aligned.

Assignability plays a key role here, but it does not cover everything. Tags are considered during comparison, but they are ignored during assignment:

Figure 58. assignableTo (src/cmd/compile/internal/types2/operand.go)

```
func (x *operand) assignableTo(check *Checker, T Type, cause  
*string) (bool, Code) {  
    ...  
    V := x.typ  
  
    // x's type V and T have identical underlying types  
    // and at least one of V or T is not a named type  
    // and neither V nor T is a type parameter.  
    if Identical(Vu, Tu) && (!hasName(V) || !hasName(T))  
&& Vp == nil &&Tp == nil {  
        return true, 0  
    }  
    ...  
}  
  
func (c *comparer) identical(x, y Type, p *ifacePair) bool {  
    ...  
    case *Struct:  
        if y, ok := y.(*Struct); ok {  
            if x.NumFields() == y.NumFields() {  
                for i, f := range x.fields {  
                    g := y.fields[i]  
                    if f.embedded !=  
g.embedded ||  
!c.ignoreTags && x.Tag(i) != y.Tag(i) ||
```

```

!f.sameId(g.pkg, g.name) ||
!c.identical(f.typ, g.typ, p) {
    }
}
return true
}
}

return false
}

```

The `identical` method in the compiler shows how this decision is made. The `c.ignoreTags` flag determines whether tags are included in the comparison:

- When set to `false`, tags are checked during comparison or type identity checks.
- When set to `true`, tags are ignored, which is the case during conversion or assignment.

As a result, Go allows assignment and type conversion between structs with matching field names and types, even if their tags differ:

```

func main() {
    p := Person{Name: "John"}
    a := struct {
        Name string `json:"hello"`
    }{Name: "John"}

    fmt.Println(Person(a)) // Output: {John }
    fmt.Println(a == p)   // invalid
    operation: a == p (mismatched types struct{Name string
    "json:\\"hello\\\""} and Person)
}

```

This may seem like a subtle technical detail, but it has practical benefits. Ignoring tags during conversion allows more flexibility when working with compatible structs that originate from different sources. For example, you may receive data from external systems that use different struct tags for serialization, while the core field definitions remain the same.

## Composition via Embedding and Field Promotion

Embedding allows one struct to include another, enabling them to work together in a clean and efficient way. In Go, the fields and methods of the embedded struct are automatically **promoted**, making them accessible as if they were part of the outer struct.

These embedded structs are often referred to as 'anonymous fields' because they are included without assigning them a separate name. Technically, they are not nameless—their type serves as the field name.

If needed, you can still access them directly using their type name:

```
type Engine struct {
    Horsepower int
}

type Car struct {
    Engine
    Brand string
}

func main() {
    myCar := Car{
        Engine: Engine{
            Horsepower: 200,
        },
        Brand: "Toyota",
    }

    // Accessing Engine's fields directly on Car
    fmt.Println(myCar.Horsepower) // Outputs: 200
    fmt.Println(myCar.Brand)     // Outputs: "Toyota"

    // Or accessing the fields through the embedded
    // struct
    fmt.Println(myCar.Engine.Horsepower) // Outputs: 200
}
```

In this case, `Car` embeds the `Engine` struct, which allows direct access to `Engine`'s fields, such as `Horsepower`, without requiring an explicit field name.

You can still refer to the embedded struct directly using `myCar.Engine.Horsepower`, but thanks to field promotion, that level of detail is often unnecessary.

If you are familiar with inheritance from object-oriented languages, this behavior may feel similar. However, Go does not support inheritance in the traditional sense. Instead, it relies on composition. The distinction becomes more apparent when looking at how Go performs field and method resolution.

When both the outer and embedded structs define fields or methods with the same name, Go follows a specific set of rules to resolve the conflict. This behavior is known as *shadowing*:

- 1. Outer struct has priority:** Go first checks the fields of the outer struct. If an embedded struct is encountered, Go doesn't dive into it right away. Instead, it notes the embedded struct for later evaluation. If the desired field is found in the outer struct, that result is used immediately.
- 2. Checking embedded structs:** If the field is not found in the outer struct, Go proceeds to check the embedded structs. It does so in the order they appear.
- 3. Resolving ambiguity:** If a potential match is located, Go verifies that no other matches exist at the same depth. If multiple candidates are found at that level, a compile-time error is triggered due to ambiguity.
- 4. Returning the resolved match:** Once a unique match is identified, Go selects it and uses it as the result.

The following snippet shows how shadowing works when field names overlap:

```
type Base struct {
    A
    B
    Name string
}

type A struct {
    Shared string
    Name   string
}

type B struct {
    Shared string
}

base.Name      // valid
base.Shared    // invalid: Error: Ambiguous (Shared exists
in both A and B)
base.A.Shared  // valid
base.B.Shared  // valid
```

Here's how Go resolves each case:

- Accessing `base.Name` works without issue. Go locates the `Name` field directly in the `Base` struct, so it returns that value immediately.
- Attempting to access `base.Shared` fails with a compile-time error. Both `A` and `B` define a `Shared` field, and since they are embedded at the same level, Go cannot determine which one to use. This results in ambiguity.

To avoid the conflict, you need to be explicit. Use `base.A.Shared` or `base.B.Shared` to clearly indicate which field you intend to access. This removes any ambiguity for the compiler and improves code readability.

## Method Shadowing in Embedded Structs

Developers with an object-oriented programming (OOP) background often expect Go to handle methods similarly to classical inheritance. At first glance, embedding might resemble inheritance, but Go emphasizes composition rather than inheritance—a key conceptual difference.

Methods in Go follow the same shadowing behavior as fields. Take the following example:

```
type Vehicle struct {
    Name string
}

func (v Vehicle) Operate() string {
    return fmt.Sprintf("%s operates on %s", v.Name,
    v.Surface())
}

func (v Vehicle) Surface() string {
    return "land"
}

Vehicle{Name: "Car"}.Operate()
// Output: "Car operates on land"
```

In this case, the `Operate` method calls `Surface`, which returns `"land"`. The result is predictable: *"Car operates on land"*.

Now, suppose we create a `Boat` type that embeds `Vehicle` and defines its own `Surface` method:

```
type Boat struct {
    Vehicle
}

func (b Boat) Surface() string {
    return "water"
}
```

This might feel like method overriding. Calling `boat.Surface()` will return `"water"` as expected. However, Go does not support traditional method overriding. Instead, the `Surface` method in `Boat` shadows the one in `Vehicle`.

The important distinction appears when calling `boat.Operate()`. Even though `Boat` defines its own `Surface` method, the `Operate` method being used belongs to `Vehicle`. Inside that method, the call to `Surface()` refers to `Vehicle.Surface`, not `Boat.Surface`.

This is because methods defined on the embedded type only see their own method set:

```
func main() {
    boat := Boat{Vehicle{Name: "Speedboat"}}
    fmt.Println(boat.Surface()) // Output: water
    fmt.Println(boat.Operate()) // Output: Speedboat
operates on land
}
```

As a result, the output might be surprising: "*Speedboat operates on land.*" Even though `Boat` has its own `Surface` method, `Operate` does not use it because it belongs to the embedded `Vehicle` type and resolves method calls within that scope.

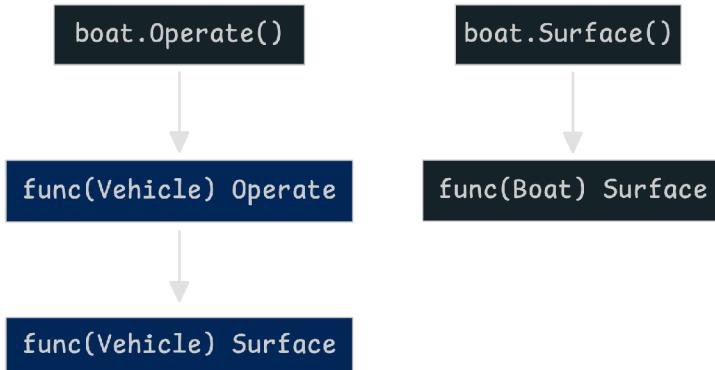


Illustration 53. `Operate` uses `Vehicle`'s own method set

## 1.3 Memory Layout and Performance Considerations

When you define a struct, Go allocates memory for its fields based on their types and arranges them in a specific sequence. Unlike arrays, which are packed tightly in memory, structs may include unused gaps. These gaps are introduced due to alignment requirements that ensure efficient access to data.

To meet alignment constraints, the compiler may insert **padding bytes** between fields. These padding bytes are not visible in code but are necessary to align data in a way that optimizes performance for the CPU.

The trade-off for better access speed is increased memory usage:

```

type A struct {
    A byte
    B int32
    C byte
    D int64
    E byte
}

type OptimizedA struct {
    D int64
    B int32
    A byte
    C byte
    E byte
}

unsafe.Sizeof(A{}) // 32
unsafe.Sizeof(OptimizedA{}) // 16

```

Although both `A` and `OptimizedA` contain the exact same fields, their memory usage is very different. `A` occupies 32 bytes, while `OptimizedA` only takes up 16. The difference comes down to **field ordering**.

By placing fields with stricter alignment requirements (such as `int64`) before those with looser requirements (like `byte`), the compiler can reduce the amount of padding needed. This helps lower the overall memory footprint and may also improve performance by optimizing how data is accessed in memory.

This layout illustrates four key memory-related concepts:

1. **Word size:** The amount of data the CPU can read or write in a single operation, based on the system architecture.
2. **Cache line:** A block of memory the CPU loads into its cache in one go, typically 64 or 128 bytes on modern systems.
3. **Alignment:** Each field must begin at a memory address that matches its type's alignment requirement (e.g., `int32` must start on a 4-byte boundary).
4. **Padding:** Extra space automatically added by the compiler to satisfy alignment rules and enable fast, aligned memory access.

If this feels a bit abstract, let's break these ideas down further.

## Understanding Word Size

We briefly introduced word size in Chapter 2 when covering integer types. Let's revisit and expand on that concept here.

Word size refers to the number of bits a CPU can process in a single operation. It influences how data is handled, stored, and addressed within a system. The processor's architecture defines the word size. For instance, on a 32-bit Linux system, the word size is 32 bits. On a 64-bit system, it's 64 bits.

This difference means that a 32-bit processor uses 32-bit general-purpose registers, while a 64-bit processor operates with 64-bit registers.

These registers handle arithmetic, logic operations, and memory addressing. As a result, a 64-bit machine can perform operations on 64-bit integers in a single step. In contrast, working with 128-bit numbers may require multiple instructions.

Performance is generally better when data aligns with the CPU's native word size. Take this C code as an example:

```
var a uint64 = 100
var b uint64 = 200
var result uint64 = a + b
```

On a 64-bit system, this addition is completed in a single CPU instruction since `uint64` matches the word size. If you were to use a larger type like `complex128` (which is 128 bits), the CPU would need additional processing to handle operations beyond the native word size.

Word size also directly affects memory addressing. A 32-bit system can access up to 4 GB of memory:

```
2^32 = 4,294,967,296 bytes = 4 GB
```

In contrast, a 64-bit system can theoretically access:

```
2^64 = 18,446,744,073,709,551,616 bytes
```

While most 64-bit systems do not utilize the full  $2^{64}$  range due to hardware constraints, the addressable space is still vastly larger than that of a 32-bit system.

## Cache Line Alignment

**Cache line** is a performance-critical concept at the hardware level. The L1 cache—the smallest and fastest cache closest to the CPU core—stores data in fixed-size blocks called cache lines. When the CPU reads from or writes to memory, it operates on entire cache lines, not individual bytes.

The size of a cache line depends on the processor architecture. Common sizes include 32, 64, 128, or even 256 bytes:

Figure 59. Cache Line Size in Different Architectures  
(src/internal/cpu/cpu\_\*.go)

```
src/internal/cpu/cpu_loong64.go: const CacheLinePadSize = 64
src/internal/cpu/cpu_ppc64x.go: const CacheLinePadSize = 128
src/internal/cpu/cpu_riscv64.go: const CacheLinePadSize = 64
src/internal/cpu/cpu_mips.go: const CacheLinePadSize = 32
src/internal/cpu/cpu_wasm.go: const CacheLinePadSize = 64
src/internal/cpu/cpu_arm.go: const CacheLinePadSize = 32
src/internal/cpu/cpu_mips64x.go: const CacheLinePadSize = 32
src/internal/cpu/cpu_mipsle.go: const CacheLinePadSize = 32
src/internal/cpu/cpu_arm64.go: const CacheLinePadSize = 128
```

```
src/internal/cpu/cpu_s390x.go: const CacheLinePadSize = 256  
src/internal/cpu/cpu_x86.go: const CacheLinePadSize = 64
```

We're focusing on ARM64, so we'll use 128 bytes as the reference cache line size. Consider the following snippet:

```
var arr [200]byte  
x := arr[10]
```

Although only a single byte is accessed, the CPU fetches the full 128-byte cache line that contains `arr[10]`. This fetch includes a range of nearby values:

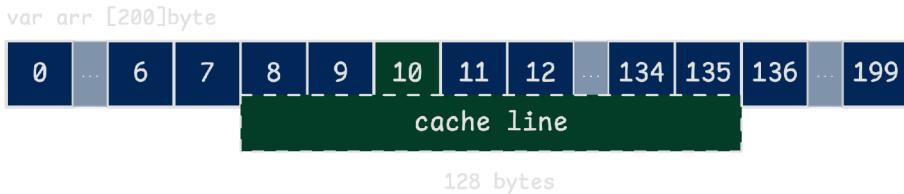


Illustration 54. Cache line brings nearby values into memory

This approach benefits from a principle called **spatial locality**, which assumes that nearby memory locations will likely be used soon. If values like `arr[8]`, `arr[9]`, `arr[11]`, or `arr[12]` are accessed shortly after, they are already loaded in the cache. This reduces the number of memory fetches and improves performance.

But how can we tell whether a cache line begins at `arr[8]` or any other specific position?

A cache line **never** starts at an arbitrary byte offset. It always begins at a memory address that is a multiple of the cache line size. This property is known as **alignment**. For instance, with a 64-byte cache line, valid starting addresses are 0, 64, 128, and so on. This alignment is a fundamental part of how modern memory systems are designed. In our earlier `arr[8]` example, it just happened to fall at the start of a cache line, but that is not guaranteed.

When a data structure crosses a cache line boundary—meaning it starts near the end of one cache line and extends into the next—the CPU must fetch both lines to access it. This is referred to as a **split load** or **split access**.

For example, suppose your system uses 128-byte cache lines and you're accessing a 32-byte structure, like our `A` struct from earlier. If that struct begins at offset 112:

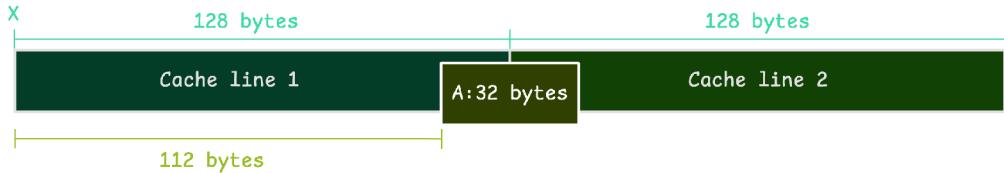


Illustration 55. Structure `A` crosses two cache lines

Since cache lines are aligned to addresses where `address % 128 == 0`, the struct starts 16 bytes before the end of one cache line and extends 16 bytes into the next. As a result, the CPU must load two cache lines to retrieve the full structure.

Proper alignment helps prevent this issue. If a structure begins at a cache-aligned address and is small enough to fit entirely within a single cache line, the CPU can access it with just one memory load. This reduces latency and improves overall performance.

One strategy for avoiding split access is to reduce the size of the structure. Here's the optimized layout of the `A` struct:

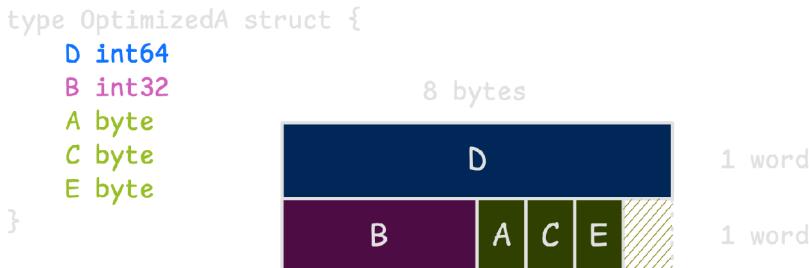


Illustration 56. Optimized struct layout reduces padding

When a struct is smaller than the cache line and does not cross its boundary, it can be loaded into the L1 cache in a **single** memory access. While a few extra memory operations may seem minor, these small inefficiencies can add up quickly—especially when processing large amounts of data or accessing memory repeatedly in performance-critical code.

## False Sharing

Cache lines can introduce performance issues such as false sharing. This occurs when multiple CPU cores access different variables that happen to be stored in the same cache line, with at least one core writing to its variable.

The reason is that any write operation causes the entire cache line to be invalidated across all cores, forcing unnecessary cache reloads on cores that only access unrelated variables within the same line. This creates contention that significantly degrades performance in multithreaded applications despite no actual data dependency between threads.

Consider two threads running on different cores:

```
type SharedData struct {
    x int32
    y int32
}
```

If each thread frequently updates either `x` or `y`, and both fields are located in the same cache line, false sharing becomes a real concern. Updating `x` invalidates the entire cache line, impacting the thread accessing `y`, despite `y` not being changed.

To mitigate false sharing in Go, you can introduce padding between fields:

```
type SharedData struct {
    x int32
    _ [124]byte // Padding to separate x and y (assuming a
    y int32
}
```

This design ensures that `x` and `y` are stored in different cache lines, preventing cache invalidation between threads that access them concurrently. A more reusable and reliable approach to avoid false sharing is to use a dedicated padded type:

```
type PaddedInt32 struct {
    value int32
    _     [124]byte // Ensure total size is 128 bytes
}

type SharedData struct {
    x PaddedInt32
    y PaddedInt32
}
```

This padding technique is also applied in the Go standard library. For example, in the `sync.Pool` implementation:

Figure 60. `sync.Pool`'s `poolLocal` struct padding (`src-sync-pool.go`)

```
type poolLocal struct {
    poolLocalInternal
    // Prevents false sharing on widespread platforms
    with
        // 128 mod (cache line size) = 0.
        pad [128 -
unsafe.Sizeof(poolLocalInternal{})%128]byte
}
```

The padding guarantees that each `poolLocal` instance is cache-aligned, avoiding interference between different processors accessing their respective data.

A similar pattern appears in the `runtime` package's semaphore implementation:

Figure 61. Semaphore table padding (`src-runtime-sema.go`)

```
type semTable [semTabSize]struct {
    root semaRoot
    pad  [cpu.CacheLinePadSize -
unsafe.Sizeof(semaRoot{})]byte
}
```

You might wonder how we know that struct `A` starts at offset 112 within a cache line instead of 110, 111, or 113. The answer comes down to alignment.

## Alignment and Padding in Structs

Alignment ensures that memory addresses meet the size requirements of each field. When a type's size is equal to or smaller than the CPU's word size, its alignment is typically based on its own size. For example, a 4-byte `int32` is aligned to 4-byte boundaries. This means its address should be a multiple of 4—such as 0, 4, 8, and so on.

The same principle applies to smaller types. A `byte` (1 byte) aligns on 1-byte boundaries, and an `int16` (2 bytes) aligns on 2-byte boundaries.

For types larger than the system's word size, alignment usually follows the word size rather than the type's actual size. On a 64-bit system, types like `complex128`

(16 bytes) or larger structs typically align on 8-byte boundaries. This is because the CPU is optimized to access memory in chunks that match the word size.

To maintain alignment, the compiler inserts **padding** between fields. The purpose of padding is to ensure each field begins at a properly aligned memory address.

The `A` struct provides a good example:

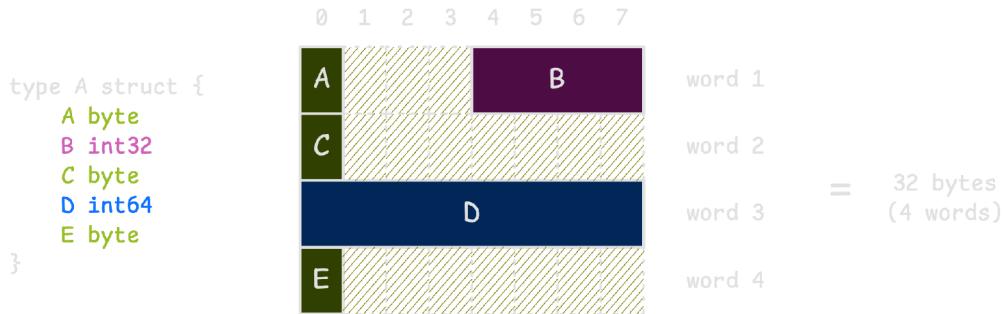


Illustration 57. Padding increases struct memory usage

Alignment does not happen by chance. Padding is what fills the unused space between fields to meet alignment rules. Here is how it plays out in the `A` struct:

- `A`: A `byte` starts at offset 0, which is fully aligned.
- `B`: An `int32` needs to start at a 4-byte boundary. Since `A` only used 1 byte, the compiler inserts 3 bytes of padding after it.
- `C`: Another `byte`, placed right after `B`, without needing additional padding.
- `D`: An `int64` requires 8-byte alignment. After `C`, 7 bytes of padding are added so that `D` begins at a valid 8-byte boundary.
- `E`: A final `byte`, placed immediately after `D`.

Altogether, the `A` struct consumes 32 bytes of memory. A considerable portion of that is padding, not actual data. This extra space is necessary to satisfy alignment requirements and help the CPU access fields efficiently.

For reference, the following table shows the size and alignment requirements of common Go data types on both 64-bit and 32-bit architectures:

Type	64-bit		32-bit	
	Size	Align	Size	Align
<code>byte</code>	1	1	1	1
<code>int8</code>	1	1	1	1
<code>int16</code>	2	2	2	2
<code>int32</code>	4	4	4	4
<code>int64</code>	8	8	8	8
<code>float32</code>	4	4	4	4
<code>float64</code>	8	8	8	8
<code>bool</code>	1	1	1	1
<code>string</code>	variable	variable	variable	variable
<code>slice</code>	variable	variable	variable	variable
<code>map</code>	variable	variable	variable	variable
<code>chan</code>	variable	variable	variable	variable

bool, uint8, int8	1	1	1	1
uint16, int16	2	2	2	2
uint32, int32	4	4	4	4
uint64, int64	8	8	8	4
int, uint	8	8	4	4
float32	4	4	4	4
float64	8	8	8	4
complex64	8	4	8	4
complex128	16	8	16	4
uintptr, *T, unsafe.Pointer	8	8	4	4
array of n T	n * T	T	n * T	T
struct { F T; G U }	varies	max(T, U)	varies	max(T, U)
other types	8	8	4	4

You may have noticed that the `A` and `OptimizedA` structs aligned to 8 bytes, but this is not always the case. A struct's alignment is determined by the largest alignment requirement among its fields:

```
// size = 1, align = 1
type A struct {
    A byte
}

// size = 12, align = 4
type B struct {
    A byte
    B int32
    C int32
}
```

Here, `A` requires only 1-byte alignment since it contains a single `byte`. On the other hand, `B` aligns to 4 bytes because it contains two `int32` fields, which each require 4-byte alignment. This difference directly affects how memory is allocated for the struct as a whole:

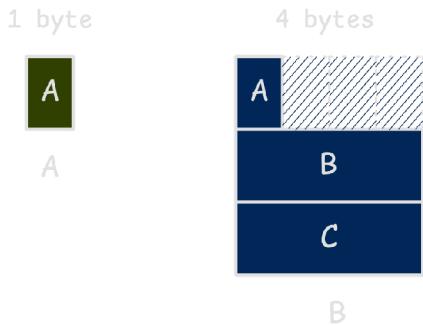


Illustration 58. Struct alignment based on largest field

When embedding structs like `A` or `B` within larger types, the overall memory layout is influenced by the alignment of the embedded fields. The compiler ensures that all fields meet their alignment requirements by adjusting the layout accordingly.

Alignment and padding may appear to waste memory—adding unused bytes between fields—but they serve a critical performance purpose. When data is aligned to its natural boundary, the CPU can read or write it more efficiently using fewer instructions.

## Instruction & Memory Fetching

There are two levels to consider when the CPU accesses memory: the cache line transfer handled by hardware, and the instruction-level memory access performed by the CPU.

At the hardware level, the memory subsystem loads full cache lines (often 64 or 128 bytes) from RAM into the CPU's cache. This is done by the cache controller.

At the instruction level, operations like `MOV`, `LOAD`, or `ADD` work on word-sized units. On a 64-bit system, reading a value of type `int64` triggers an 8-byte load instruction. Even though a 64-byte cache line was loaded, only the necessary 8 bytes are accessed by the CPU instruction.

Consider a struct with two fields: `A`, which is 1 byte, and `B`, which is 8 bytes. If no padding were added, the memory layout would appear as follows:

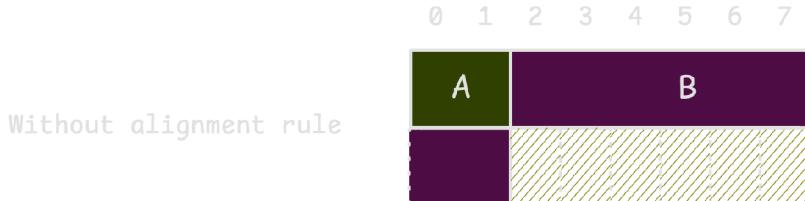


Illustration 59. Memory layout without alignment rules

On certain architectures, this kind of misalignment prevents the CPU from reading all 8 bytes of `B` in a single operation. The reason is that CPUs are built to fetch data in word-sized chunks, which means they expect to load 8 bytes (in 64-bit systems) starting at aligned addresses (0, 8, 16, etc.).

Even though the full 8 bytes are present in memory, the hardware cannot treat the access as a single, efficient load. It must retrieve the overlapping memory regions in multiple steps and piece the value together using internal registers. This process increases latency and reduces performance.

## Working with Empty Structs

Starting with Go 1.22, iterating a fixed number of times has become more straightforward. You can now use `for range` directly with an integer:

```
for range 1000 {  
    // ...  
}
```

This syntax is concise and easy to read, with no need for extra tricks. But why mention this?

Before this feature was added, there was a common trick that used an array of empty structs to achieve the same result. It allowed a `for range` loop to be written in a short and concise way:

```
empty := [1000]struct{}{}
for range empty {
    // ...
}

unsafe.Sizeof(empty) // 0
```

Even though the array contains 1000 elements, it occupies zero bytes in memory. The call to `unsafe.Sizeof` confirms this. It works because `struct{}` has no fields and therefore takes up no space.

Empty structs remain useful beyond iteration. A common usage pattern is in maps, such as `map[T]struct{}`, to represent sets where only the keys matter. This approach is both memory-efficient and clear in intent. In contrast, using `map[T]bool` stores additional data that is often unnecessary.

However, things become more subtle when empty structs are included in a larger struct. If an empty struct appears at the end, the size of the containing struct might be larger than expected:

```
type First struct {
    B struct{}
    A int64
}

type Last struct {
    A int64
    B struct{}
}

// unsafe.Sizeof(First{}) // 8
// unsafe.Sizeof>Last{}) // 16
```

Although both `First` and `Last` contain the same fields and the empty struct `B` technically takes up no space, their sizes differ. `First` is 8 bytes, while `Last` occupies 16 bytes.

This happens because Go computes field addresses by starting at the struct's base and adding each field's offset. In `First`, the empty struct is placed at the beginning, so its address aligns with the start of the struct. In `Last`, the empty struct is at the end, which causes its address to fall right at the end of the allocated memory block.

This creates a potential issue. If a pointer is taken to `Last.B`, it would point just beyond the memory reserved for `Last`. In Go, that could lead to unsafe or undefined behavior by potentially overlapping with unrelated memory.

To avoid this, the compiler adds padding to the struct to ensure it ends at a safe boundary. This adjustment increases the size of `Last` to 16 bytes:

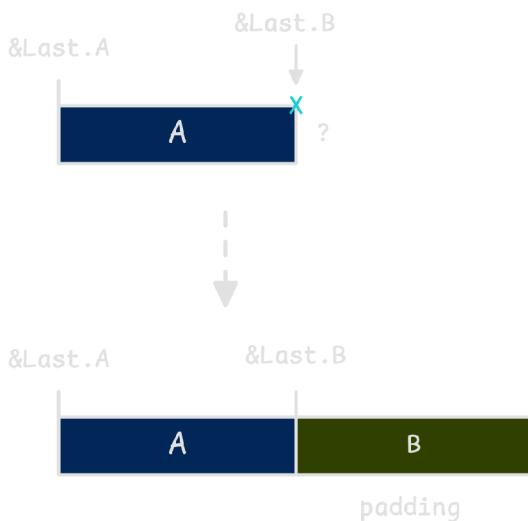


Illustration 60. Zero-size values and memory alignment

Keep in mind, zero-size values aren't limited to empty structs. Arrays with zero elements (e.g., `[0]T`) are also treated as having zero size—anything that does not reserve space in memory falls into this category.

## Zero-Size Values on the Stack

Another interesting behavior of zero-size values is that multiple variables may share the same memory address. This is a compiler optimization designed to reduce memory usage when dealing with values that require no storage.

While helpful to understand, this behavior is an implementation detail and should not be relied upon across all situations:

```

func main() {
    var a, b = struct{}{}, struct{}{}
    var c int = 100

    println("a:", &a) // 0x1400004e747
    println("b:", &b) // 0x1400004e747
    println("c:", &c) // 0x1400004e747
}

```

Here, both `a` and `b` are empty structs. Since they occupy no memory, the compiler reuses the same address for both on the stack. Additional zero-size variables may also receive this same address.

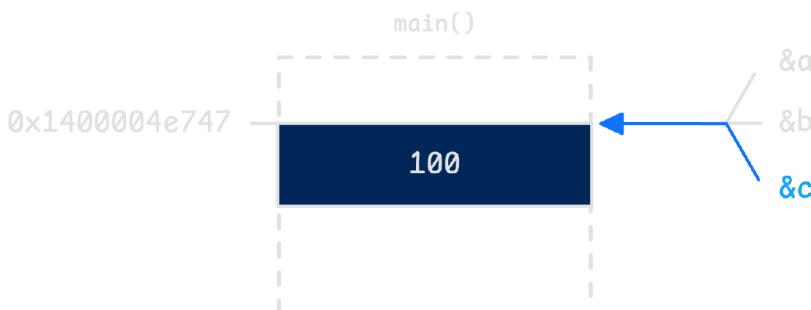


Illustration 61. Zero-size variables may have identical addresses

Now take a look at variable `c`, which is an `int`. Even though it serves a different purpose and holds actual data, it appears to share the same address as `a` and `b`. In fact, `c` occupies the next 8 bytes of memory starting from that location, while `a` and `b` take up no space at all.

This address sharing occurs because the compiler recognizes that zero-size values don't interfere with memory state and can be co-located without any impact.

## Zero-Size Values on the Heap

Allocating zero-size values on the heap involves a slightly different strategy. By using the `//go:noinline` directive and returning pointers, you ensure these values outlive the scope of the function where they are created:

```

type Empty struct{}

//go:noinline
func GetZeroSizeStruct() *Empty {
    return &Empty{}
}

```

```
//go:noinline
func GetZeroSizeArray() *[0]int {
    return &[0]int{}
}

func main() {
    // heap
    println("zero-size struct:", GetZeroSizeStruct()) // 0x100859a40
    println("zero-size array:", GetZeroSizeArray()) // 0x100859a40
}
```

In this example, both heap-allocated zero-size values point to the same address. This is not a coincidence—the Go runtime uses a predefined memory location for all zero-size heap allocations. This special location is referred to as the '**zero base address**'.

If you're interested in the internal details, this address is managed by the `zerobase` variable in the `runtime` package. It's part of Go's runtime optimizations to avoid unnecessary memory use when allocating values that require no space.

## 2. Interfaces: Contracts, Composition, and Runtime Mechanics

At its core, an interface is a contract: a collection of method signatures that a type can implement. If a type provides implementations for all the methods defined in an interface, it is said to satisfy that interface. These method signatures must match exactly in name, parameter types, and return types—though parameter names themselves are not part of the contract and can differ.

### 2.1 Interfaces as Behavioral Contracts and Implicit Satisfaction

An interface defines a set of expectations for behavior. It specifies which methods a type must provide, but not how those methods are implemented:

```
type Worker interface {
    Work()
}
```

Any type—whether a struct or a user-defined type—that implements the `Work()` method is considered a `Worker`.

One distinguishing feature of Go's interface system, compared to languages like Java or C#, is its use of implicit implementation. There's no need to explicitly state that a type satisfies an interface. As long as the required methods are present, the type automatically conforms to the interface:

```
type Developer struct {
    Name string
    Skill string
}

func (d Developer) Work() {
    fmt.Printf("%s is coding in %s\n", d.Name, d.Skill)
}

type Designer struct {
    Name string
    Tool string
}

func (d Designer) Work() {
    fmt.Printf("%s is designing with %s\n", d.Name,
d.Tool)
}
```

Both `Developer` and `Designer` fulfill the `Worker` interface simply by implementing a `Work()` method with a matching signature. This allows values of either type to be assigned to a variable of type `Worker`:

```
var work Worker

work = Developer{"Alice", "Go"}
work.Work()

work = Designer{"Bob", "Figma"}
work.Work()
```

What's missing? There's no explicit declaration like, "*Developer implements Worker.*" The method signatures match, and that's all Go needs.

Although this implicit mechanism might seem like it could lead to confusion, in practice, it works cleanly and efficiently. You don't need to import the interface's defining package just to implement it. This design choice minimizes

dependencies and helps keep your code organized. It's a practical and lightweight approach that distinguishes Go from more verbose, declaration-heavy languages.

For instance, consider the `bufio` package. When performing buffered reading, it expects an input that satisfies the `io.Reader` interface:

```
package bufio

// NewReader returns a new [Reader] whose buffer has the
// default size.
func NewReader(rd io.Reader) *Reader {
    return NewReaderSize(rd, defaultBufSize)
}
```

Your type doesn't need to directly import the `io` package to be compatible. It only needs to implement the `Read([]byte) (int, error)` method. Go handles the rest automatically:

```
type MyReader struct {}

func (r MyReader) Read(p []byte) (n int, err error) {
    // Custom reading logic goes here
}

func main() {
    reader := bufio.NewReader(MyReader{})
}
```

In this example, `bufio` accepts `MyReader` without issue. The concrete type is irrelevant—what matters is that it provides a method matching the `io.Reader` interface.

That said, there are situations where you might want to explicitly assert that a type satisfies a particular interface. This typically arises when the compiler can't infer the relationship, such as when working with reflection or code generated at runtime.

In these scenarios, you can enforce the contract with a straightforward check:

```
import "io"

// Syntax: var _ <interface> = (*type)(nil)
var _ io.Reader = (*MyReader)(nil)
```

This line tells the compiler: "I expect `*MyReader` to implement `io.Reader`. Validate this now." If the implementation is incomplete or incorrect, Go will produce a compile-time error:

```
cannot use (*MyReader)(nil) (value of type *MyReader) as
io.Reader value in variable declaration: *MyReader does not
implement io.Reader (missing method Read)
```

Now that the fundamentals are clear, let's return to the `Worker` interface and see it applied in a real scenario:

```
func main() {
    var workers = []Worker{
        Developer{Name: "Alice", Skill: "Go"},
        Designer{Name: "Bob", Tool: "Photoshop"},
    }

    for _, w := range workers {
        w.Work()
    }
}
```

Executing this code produces the following output:

```
Alice is coding in Go
Bob is designing with Photoshop
```

Here, the `workers` slice holds values of different types: `Developer` and `Designer`. Both implement the `Worker` interface, illustrating polymorphism in action. Interfaces let you write code that operates on various types uniformly, while each type still defines its own unique behavior.

Go doesn't maintain a master list of all types that implement a given interface. If a type is never used with an interface, there's no relationship between them. Instead, when a type is actually assigned to an interface, the compiler verifies whether it meets the interface's requirements. If it does, Go generates what's called an '**interface table**' (`itab`). This structure maps the interface to the concrete type and associates the required method implementations.

To understand how this works behind the scenes, let's take a brief look at the assembly generated from an earlier example:

## Go Code

```
var workers
= []Worker{

Developer{
    Name:
    "Alice",
    Skill: "Go"
        },
Designer{
    Name:
    "Bob",
    Tool:
    "Photoshop"
        },
}
```

## Go Assembly

```
MOVD    $type:main.Developer(SB), R0
MOVD    $go:itab.main.Developer,main.Worker(SB),
R2
MOVD    $type:main.Designer(SB), R0
MOVD    $go:itab.main.Designer,main.Worker(SB),
R2
```

The `$type` and `$go:itab` symbols are embedded directly into the compiled binary. These are not computed during program execution; they're resolved at compile time. By the time your code runs, Go has already established the mappings between concrete types and the interfaces they implement. At runtime, these connections are simply referenced.

With that mechanism in mind, let's look at a practical use of interfaces:

```
func announce(s Worker) {
    s.Work()
}

func main() {
    announce(Developer{Name: "Alice", Skill: "Go"})
    announce(Designer{Name: "Bob", Tool: "Photoshop"})
}
```

The `announce` function accepts any value that satisfies the `Worker` interface. This promotes flexibility while keeping the contract clear—any type implementing `Work()` is valid here.

Another important concept is the empty interface, written as `interface{}` or `any` (introduced in Go 1.18). This interface doesn't require any methods, so it can hold values of any type. Even beginners often use it without realizing.

A well-known example is the `fmt.Println` function:

Figure 62. `fmt.Println` (`src/fmt/print.go`)

```
package fmt

func Println(a ...any) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

Here, `any` is just a more concise alias for `interface{}`, defined like this:

Figure 63. `any` Type (`src/builtin/builtin.go`)

```
type any = interface{}
```

While the empty interface offers maximum flexibility, it also introduces limitations. Once a value is stored in an `interface{}`, its concrete type is no longer directly accessible.

To interact with it meaningfully, you need to retrieve its original type using a type assertion or a type switch:

```
func doSomething(i interface{}) {
    switch v := i.(type) {
    case string:
        fmt.Println("String:", v)
    case int:
        fmt.Println("Int:", v)
    default:
        fmt.Println("Unknown type")
    }
}
```

The `doSomething` function checks the underlying type at runtime and acts accordingly.

Many developers use `interface{}` without knowing its formal name: the **anonymous empty interface**. Technically, you could define your own named version, but in practice, the built-in type covers all use cases and is universally preferred.

## 2.2 Composing and Embedding Interfaces

In Go, interface embedding appears in two primary forms: embedding one interface within another, or embedding an interface into a struct.

Embedding interfaces into other interfaces allows you to construct more flexible and modular designs by composing smaller, purpose-specific interfaces. Rather than grouping numerous methods into a single large interface, you can divide behavior into focused components and combine them as needed:

```
type Speaker interface {
    Speak() string
}

type Mover interface {
    Move() string
}

// Embedding interfaces
type Animal interface {
    Speaker
    Mover
}
```

Here, the `Animal` interface embeds both `Speaker` and `Mover`. To satisfy `Animal`, a type must implement both `Speak()` and `Move()`. This promotes modularity by encouraging composition over duplication. Instead of repeating method signatures, shared capabilities can be embedded and reused.

It also allows for greater precision when defining API boundaries. For instance, a function that only needs speaking behavior can accept the `Speaker` interface, regardless of whether the passed-in type supports other capabilities.

This pattern is widely adopted in Go's standard library, particularly in the `io` package:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

```

// ReadWriter is the interface that groups the basic Read
and Write methods.
type ReadWriter interface {
    Reader
    Writer
}

// ReadCloser is the interface that groups the basic Read
and Close methods.
type ReadCloser interface {
    Reader
    Closer
}

// WriteCloser is the interface that groups the basic Write
and Close methods.
type WriteCloser interface {
    Writer
    Closer
}

// ReadWriteCloser is the interface that groups the basic
Read, Write and Close methods.
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}

```

Things become even more interesting when embedded interfaces include methods with the same name. This behaves similarly to struct embedding, but introduces some additional considerations:

```

type Speaker interface {
    Speak() string
}

type Singer interface {
    Speak() string
}

type DoubleSpeak interface {
    Speaker
    Singer
}

```

Prior to Go 1.14, embedding interfaces that declared methods with identical names would result in a conflict, preventing compilation. Since Go 1.14 [rln], this

restriction has been lifted. As long as the overlapping methods have identical signatures, the embedded interfaces can coexist without issue [ovl].

In the example above, `DoubleSpeak` is valid because both `Speak()` methods share the same definition.

So far, so good. Let's move on to the second scenario: 'embedding an interface in a struct'.

This might seem a bit unconventional. Why embed an interface into a struct? After all, interfaces don't store data or provide concrete behavior. So what's the benefit?

Let's walk through a basic example to understand what's really happening:

```
type Speaker interface {
    Speak() string
}

type Dog struct {
    Speaker
    Name string
}
```

Here, the `Dog` struct embeds the `Speaker` interface. At a glance, this appears reasonable—if a `Dog` can speak, embedding `Speaker` should imply that.

However, things take a turn when you attempt to use it:

```
func main() {
    dog := Dog{Name: "Molly"}
    fmt.Println(dog.Speak()) // panic: runtime error:
    invalid memory address or nil pointer dereference
}
```

Rather than printing a friendly bark, the program crashes with a runtime panic. The issue lies in how Go handles embedded interfaces inside structs.

The embedded interface doesn't bring along an implementation. If no concrete value is assigned to the interface, it defaults to `nil`. This subtle detail is critical—without an actual type assigned to the interface, calling its method results in a `nil` pointer dereference.

Let's fix the issue so the code behaves as expected. One solution is to implement the `Speak` method directly on the `Dog` type:

```
func (d Dog) Speak() string {
    return "Woof!"
}

func main() {
    dog := Dog{}
    fmt.Println(dog.Speak()) // Output: Woof!
}
```

This version works and gives the intended output. But it can lead to a misunderstanding. It might look like embedding an interface requires the struct to implement the interface's methods. That's not actually the case.

The original problem isn't with `Dog` itself—it's with the embedded `Speaker` field, which remains uninitialized:

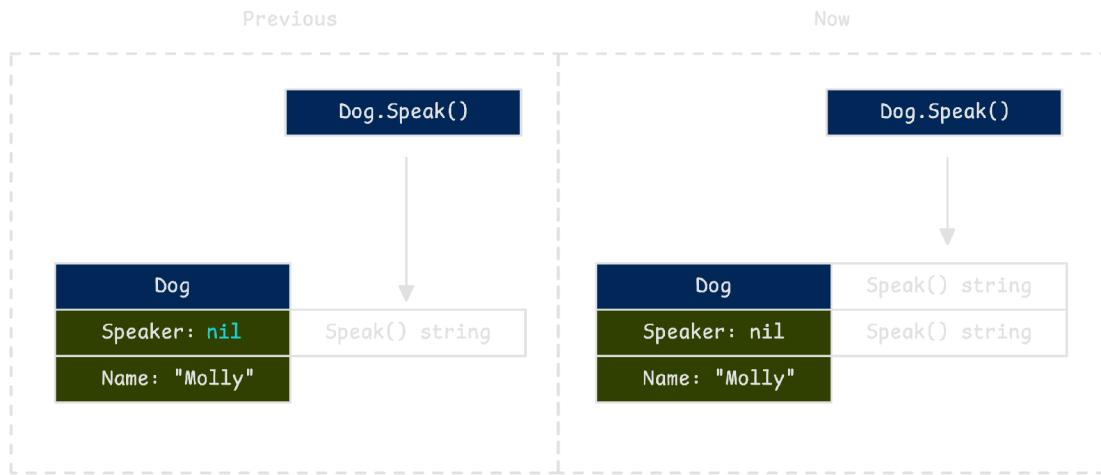


Illustration 62. Fixing nil interface method call

By adding `Speak` directly to `Dog`, you're not fulfilling the embedded interface. You're simply defining a method with the same name, which shadows the one expected from `Speaker`. This fixes the panic in this context, but doesn't resolve the underlying issue.

If you attempt to call `Speak` through the `Speaker` field, the original panic returns:

```
func main() {
    dog := Dog{}
    fmt.Println(dog.Speak())           // Output: Woof!
```

```
        fmt.Println(dog.Speaker.Speak()) // panic: runtime
error: invalid memory address or nil pointer dereference
}
```

To address this correctly, you need to assign a concrete implementation to the embedded interface. Otherwise, the field remains a `nil` interface and will fail at runtime:

```
type DogVoice struct {}

func (w DogVoice) Speak() string {
    return "Grrr!"
}

func main() {
    dog := Dog{Speaker: DogVoice{}}
    fmt.Println(dog.Speak())           // Output: Grrr!
    fmt.Println(dog.Speaker.Speak())   // Output: Grrr!
}
```

Now, the code runs safely. The `Speaker` field is initialized with an actual implementation (`DogVoice{}`), eliminating the risk of a `nil` dereference.

It's worth noting that the earlier behavior—where `Dog` seemed to respond to `Speak()`—wasn't due to interface enforcement. It was simply method promotion. The method defined on `Dog` took precedence and masked the absence of an actual interface implementation.

The true value of embedding an interface in a struct lies in granting the struct access to the interface's methods or enabling it to be used in places where that interface is expected.

Consider Go's `context` package as a practical example. The `context.Context` interface is embedded into several internal structs like `valueCtx`, `cancelCtx`, and `stopCtx`:

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key any) any
}

type valueCtx struct {
    Context
    key, val any
}
```

```

}

func WithValue(parent Context, key, val any) Context {
    // Implementation details...
    return &valueCtx{parent, key, val}
}

type cancelCtx struct {
    Context

    mu    sync.Mutex
    done atomic.Value
    // Additional fields...
}

```

This approach works because the `Context` interface sets the contract, while types like `valueCtx` and `cancelCtx` manage the internal logic and state.

It's a strong example of separation of concerns: the interface outlines **what** behavior is required, and the struct determines **how** that behavior is carried out.

## 2.3 Internal Structure of Empty Interfaces

One concept that often causes confusion in Go is what's sometimes called the 'nil but not nil' interface. It's a subtle but important distinction, and once understood, it helps clarify some of Go's more nuanced behaviors around interfaces.

Take a look at this example:

```

func main() {
    var x interface{}
    var y *int = nil

    x = y

    if x != nil {
        fmt.Println("x != nil")
    } else {
        fmt.Println("x == nil")
    }

    fmt.Println(x)
}

```

At first glance, you might expect `x` to be `nil` after assigning it the `nil` pointer `y`. However, the output is:

```
"x != nil"
```

Why? Even though `y` is `nil`, `x` holds more than just a `nil` value—it stores the type `*int` along with a `nil` pointer. That type information makes all the difference. Because `x` contains a concrete type, Go treats it as non-nil.

An empty interface (`interface{}`) is only considered truly `nil` when both its type and value are unset. To see this in action, you can inspect the raw contents of the interface using `println`:

```
func main() {
    var x interface{}
    println("prev:", x)

    var y *int = nil
    x = y

    println("next:", x)
}
```

```
prev: (0x0,0x0)
next: (0x1024174a0,0x0)
```

Initially, `x` is empty—no type, no value—represented as `(0x0,0x0)`. Once `y` is assigned, `x` now holds two things: a pointer to the type descriptor for `*int`, and a second pointer to the `(nil)` value. Even though the value is `nil`, the presence of the type means the interface isn't empty. Go treats it as non-nil because it's holding type information.

Now let's look at a more applied example using custom types:

```
type Developer struct {
    Name string
    Skill string
}

type Designer struct {
    Name string
    Tool string
}

func main() {
    var workers = []interface{}{
```

```

        Developer{Name: "Alice", Skill: "Go"},  

        Designer{Name: "Bob", Tool: "Photoshop"},  

        Developer{Name: "Charlie", Skill: "Python"},  

    }  

    for i := range workers {  

        println(workers[i])  

    }  

}

```

This code gives you a peek at how interface values are represented internally:

```
(0x1047ca100, 0x140000606a0) # Alice  

(0x1047ca060, 0x14000060718) # Bob  

(0x1047ca100, 0x140000606f8) # Charlie
```

You'll see that each interface value consists of two pointers. To understand what those pointers are, we can refer to Go's source code:

Figure 64. eface (src/runtime/runtime2.go)

```

type eface struct {
    _type *_type
    data  unsafe.Pointer
}

```

Under the hood, an empty interface is implemented as a two-word structure. The first word, `_type`, points to the type descriptor, and the second points to the actual data. Even when the data is `nil`, the presence of a type makes the interface non-empty.

With this understanding, we can now better interpret the output from our earlier example. The first pointer (e.g., `0x1047ca100`) is the `_type` pointer, which refers to the type descriptor stored in a read-only section of the binary. This section typically resides near the start of the binary, which explains why these addresses are significantly smaller than the second pointer (e.g., `0x140000606a0`). The second pointer is the data pointer. It points to the actual value in memory.

Notice that the first pointer is the same for both Alice and Charlie (`0x1047ca100`). This confirms that the type descriptor for `Developer` is reused across all instances.

Go maintains a single shared descriptor for each concrete type:



Illustration 63. Go interface stores two pointers

To understand how Go builds an interface value from a concrete type, let's look at a breakdown that compares Go assembly with simplified pseudo-code:

### Go Assembly

```
.
.
.
MOVD    $go:string."Alice"
(SB), R2
MOVD    R2, main..autotmp_9-
120(SP)
MOVD    $5, R2
MOVD    R2, main..autotmp_9-
112(SP)

MOVD    $go:string."Go"(SB),
R2
MOVD    R2, main..autotmp_9-
104(SP)
MOVD    $2, R2
MOVD    R2, main..autotmp_9-
96(SP)

MOVD
$type:main.Developer(SB), R2
MOVD    R2, main..autotmp_7-
88(SP)
MOVD    $main..autotmp_9-
120(SP), R2
MOVD    R2, main..autotmp_7-
80(SP)
```

### Pseudo Code

```
var data Developer
r2 = &"Alice"
data.Name.str = r2
r2 = 5
data.Name.len = r2

r2 = &"Go"
data.Skill.str = r2
r2 = 2
data.Skill.len = r2

r2 = &types[Developer]
eface._type = r2
r2 = &data
eface.data = r2
```

This illustrates how the Go compiler creates an interface value. Remember, strings in Go are structs with two fields: a pointer to the underlying data (`str`) and a length (`len`).

Also note that the `eface.data` pointer refers to a copy of the concrete value, not the original variable—unless the stored value is already a pointer:

```
a := 10
var b interface{} = a

a = 20

fmt.Println(a) // 20
fmt.Println(b) // still 10
```

Here, changing `a` to `20` has no effect on `b` because `b` holds a separate copy of `a`'s original value. This is straightforward but not universally consistent. Whether an interface contains a copy of the value or a reference depends on the type being assigned.

To see this more clearly, consider what happens when assigning integers to interfaces:

```
func main() {
    var a int = 5

    var var1 interface{} = a
    var var2 interface{} = a

    println("var1:", var1)
    println("var2:", var2)

    var const1 interface{} = 5
    var const2 interface{} = 5

    println("const1:", const1)
    println("const2:", const2)
}

var1: (0x100265320, 0x1400004e710)
var2: (0x100265320, 0x1400004e708)

const1: (0x100265320, 0x10025a9d8)
const2: (0x100265320, 0x10025a9d8)
```

This output reveals that the type descriptor for `int` (`0x100265320`) is reused in all cases. That's expected—Go reuses the same descriptor globally for each type.

The most interesting detail lies in the `data` pointers. For `var1` and `var2`, the data addresses differ. But for `const1` and `const2`, they are identical:

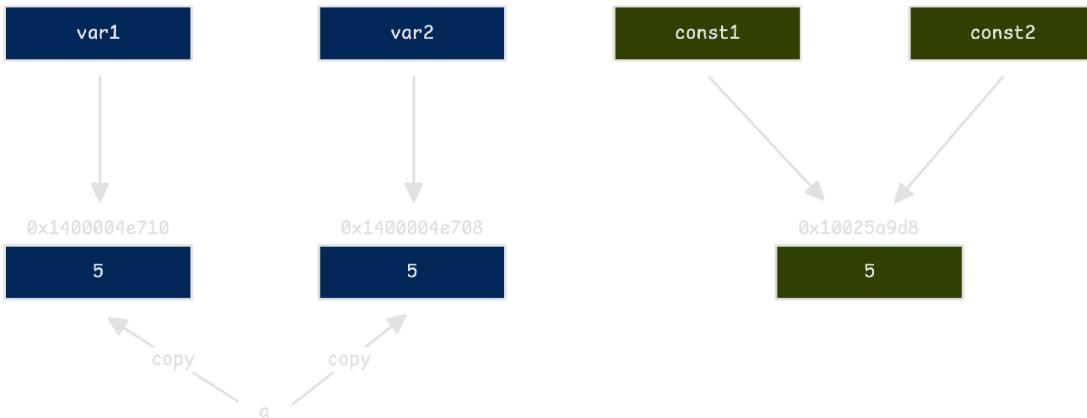


Illustration 64. Variables copy values, constants share memory

This difference highlights how Go handles memory under the hood:

- `var1` and `var2` have distinct data pointers (`0x1400004e710` and `0x1400004e708`), indicating that each assignment results in a separate copy of the value.
- `const1` and `const2`, on the other hand, both point to the same address (`0x10025a9d8`), even though they were assigned independently using the same constant.

The `eiface.data` pointers for constants are also noticeably smaller than those for variables, and even smaller than the type descriptor pointer (`0x100265320`). This is because constants reference preallocated memory in the binary's read-only section.

Go follows a defined strategy when assigning values to an interface's `data` field:

1. If the concrete value is a pointer, Go uses that pointer directly without copying the data. This is an optimization worth remembering.
2. If the value has zero size, `eiface.data` points to a special location called `zerobase`, which is also used for empty structs.
3. For one-byte types like `uint8`, `int8`, or `bool`, `eiface.data` may point to a shared memory location like `staticuint64s`, which we briefly noted earlier when discussing how Go stores strings.
4. If the value is a read-only global (such as a literal or constant), the pointer references its fixed location in the binary's read-only memory.
5. If the value is small ( $\leq 1024$  bytes) and doesn't escape to the heap, the compiler creates a temporary copy on the stack, and `eiface.data` points to that.

## 6. For larger values or values that escape, Go allocates a new copy on the heap.

It's worth noting that while this is how things work in most versions of Go, specific behavior may differ slightly depending on the Go release.

That's quite a bit of theory, so let's see how it behaves in practice:

```
func main() {
    // eface.data uses the original pointer directly
    pointer := new(int)
    pointerEface := interface{}(pointer)
    println("pointer:", pointer)
    println("pointerEface:", pointerEface)

    // eface.data points to zerobase for zero-size types
    println("zeroSizeArrayEface:", interface{}{[0]int{}})
    println("emptyStructEface:", interface{}(struct{}{}))
}
```

```
pointer: 0x1400005e710
pointerEface: (0x100e0fce0, 0x1400005e710)
zeroSizeArrayEface: (0x100e12580, 0x100ea1a20)
emptyStructEface: (0x100e13720, 0x100ea1a20)
```

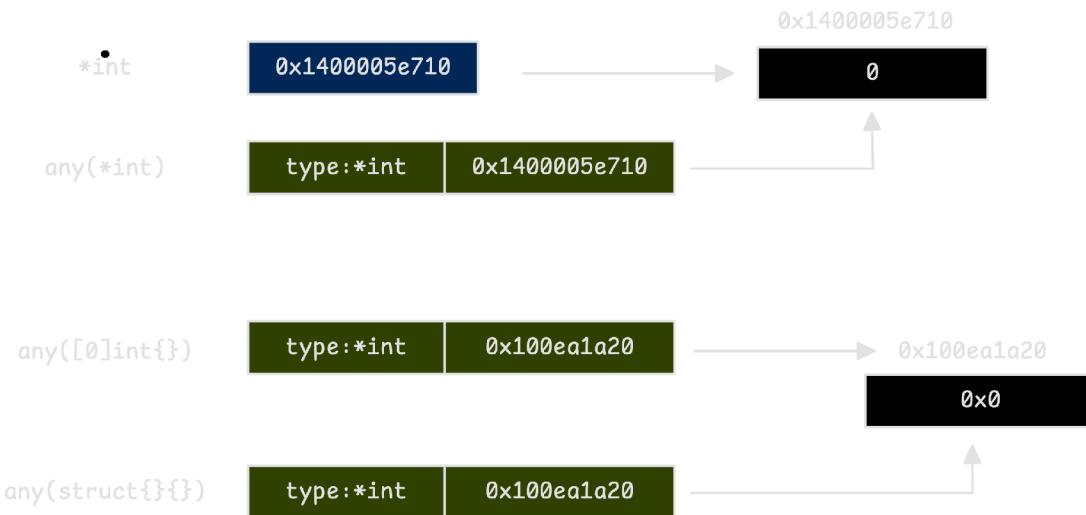


Illustration 65. Go stores zero-size types in zerobase

In the first part, `pointerEface` directly references the memory address of the pointer, exactly as expected.

In the second and third outputs, the data pointer ( `eiface.data` ) points to `runtime.zerobase`. This happens because `[0]int{}` and `struct{}{}` occupy zero bytes. The compiler knows there's no reason to allocate memory for values that hold no data.

Now let's examine how Go handles one-byte types such as `uint8` and `bool`.

Go leverages a preallocated array called `staticuint64s`, which holds values from 0 to 255. This optimization helps avoid unnecessary allocations. The following assembly confirms this:

Go Code	Go Assembly
<pre>func main() {     uint8Eface := any(uint8(1))     boolEface := any(true)      println("uint8Eface:", uint8Eface)     println("boolEface:", boolEface) }</pre>	<pre># uint8Eface := any(uint8(1)) MOVD    \$type:int8(SB), R0 MOVD    \$runtime.staticuint64s+8(SB), R1  # boolEface := any(true) MOVD    \$type:bool(SB), R0 MOVD    \$runtime.staticuint64s+8(SB), R1</pre>
<pre>uint8Eface: (0x1002d5720, 0x1002cd368) boolEface: (0x1002d5aa0, 0x1002cd368)</pre>	

Both values share the same `eiface.data` pointer, despite having different types. This reuse is intentional.

The assembly shows that both `uint8Eface` and `boolEface` reference a shared address in the `runtime.staticuint64s` array:

Figure 65. staticuint64s (src/runtime/iface.go)

```
// staticuint64s is used to avoid allocating in convTx for
// small integer values.
var staticuint64s = [...]uint64{
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
```

```

        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        ...
        0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
        0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
    }
}

```

The compiler pulls from this shared array to store small values efficiently. This avoids duplicate allocations for common cases like small integers and booleans.

Recall that `staticuint64s` was previously mentioned in Chapter 3. Go uses it again when converting a one-byte slice to a string, allowing it to skip allocation by referencing this array directly.

## 2.4 Non-Empty Interfaces and Method Dispatch via itab

Now that we've covered empty interfaces, let's shift to non-empty interfaces, where things become slightly more involved. It's common to describe an interface as a pair of 'type' and 'value'—and that holds true for empty interfaces. But non-empty interfaces introduce an additional layer:

```

type Worker interface {
    Work()
}

func main() {
    println(interface{}(Developer{})) // Empty interface
    println(Worker(Developer{}))     // Non-empty
interface
}

(0x100eab960,0x14000060718)
(0x100eb27b8,0x140000606f8)

```

Both interface values wrap the same concrete type: `Developer`. However, the first pointers differ.

This difference doesn't mean the type descriptor for `Developer` (`$type:Developer`) has changed. Instead, non-empty interfaces use a different mechanism for storing type information.

In a non-empty interface, the first pointer does not refer directly to the type descriptor. Instead, it points to an **interface table**, or `itab`.

Figure 66. iface Struct (src/runtime/runtime2.go)

```
type iface struct {
    tab *itab
    data unsafe.Pointer
}
```

Some might ask why Go doesn't simply add more fields to the `iface` struct—perhaps something like this:

```
type iface struct {
    tab     *itab
    _type   *_type
    data   unsafe.Pointer
}
```

The answer lies in Go's consistent interface representation. While non-empty interfaces contain more information (such as method pointers), the language makes no distinction between empty and non-empty interfaces in terms of structure. Both must occupy the same size, 16 bytes. Adding extra fields would break this consistency.

When a value like `Dog` is assigned to a variable of type `Speaker`, the compiler builds an `iface` structure representing that interface. The `tab` field then points to an `itab`, which includes detailed information about the `Dog` type—specifically, how it satisfies the `Speak` method defined in the `Speaker` interface.

Think of `itab` as a runtime reference that tells Go exactly how a specific type satisfies the requirements of an interface:

Figure 67. `itab` Struct (`src/runtime/runtime2.go`)

```
type itab struct {
    inter *interfacetype // at 0 byte
    _type *_type          // at 8 byte
    hash  uint32           // at 16 byte
    _     [4]byte           // at 20 byte
    fun   [1]uintptr        // at 24 byte
}
```

The `itab` serves as an intermediary. Like the empty interface, it includes a pointer to the type descriptor, but it also carries additional metadata that helps the runtime invoke the correct method implementations.



Illustration 66. Non-empty interfaces use an extra indirection layer

Here's what each field represents:

- `inter` : Points to the `interfacetype`, which holds metadata about the interface, including its required methods.
- `_type` : The descriptor for the concrete type that's being assigned to the interface.
- `hash` : A cached value of `_type.hash`, used for quick lookups, especially in type switches.
- `fun` : An array of function pointers that link directly to the methods needed to satisfy the interface.

Among these, the `fun` array is particularly important. Starting at byte offset 24, it holds the actual function pointers for methods required by the interface. These pointers are used during method dispatch at runtime, and their usage is visible in the compiled assembly.

Let's look at a concrete example:

```

type Worker interface {
    Work()
}

var worker = Worker(
    Developer{
        Name: "Aiden",
        Skill: "Go",
    },
)

func main() {
    worker.Work()
}

```

### Go Assembly Code

### Pseudo Code

MOVD main.worker(SB), R1	→ R1 = worker.itab *itab
MOVD main.worker+8(SB), R0	→ R0 = worker.data *Developer
MOVD 24(R1), R1	→ R1 = worker.itab.fun[0] (*Developer).Work
CALL (R1)	→ R1() (*Developer).Work()

Illustration 67. Go interface method dispatch in assembly

Breaking down the key assembly instructions:

- `MOVD main.worker(SB), R1`: Loads the `itab` pointer from the `iface` structure into register `R1`.
- `MOVD main.worker+8(SB), R0`: Loads the `data` pointer (the concrete `Developer` value) into `R0`.
- `MOVD 24(R1), R1`: Retrieves the method pointer from the `fun[0]` slot in the `itab`, which corresponds to `(*Developer).Work` method.
- `CALL (R1)`: Invokes the method using the function pointer.

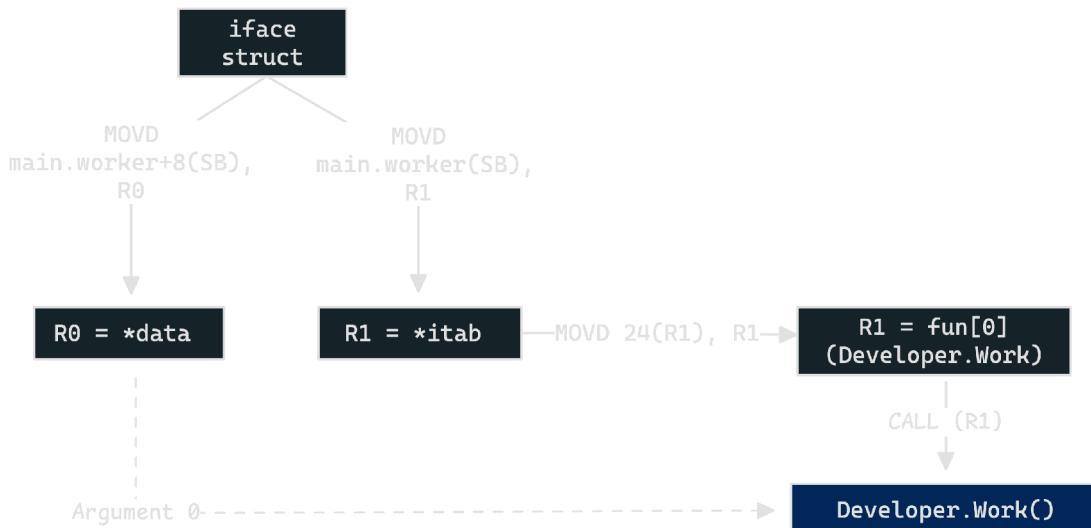


Illustration 68. Go interface method invocation flow

This is how Go performs dynamic dispatch. The `fun[0]` entry in the `itab` provides a direct link to the correct method implementation for the specific concrete type. It's how the runtime bridges the gap between an interface's contract and the actual method logic.

Assembly is fascinating, but let's be honest—it's not something most developers use on a daily basis. To demonstrate the same idea in a more approachable way,

we can rely entirely on Go code. We'll use some of the techniques from the `unsafe` package discussed back in Chapter 2:

```
type iface struct {
    tab *itab
    data unsafe.Pointer
}

type itab struct {
    inter unsafe.Pointer
    _type unsafe.Pointer
    -     uint32
    -     [4]byte
    fun   [1]unsafe.Pointer
}

func main() {
    dev := Developer{Name: "Alice", Skill: "Go"}
    worker := Worker(dev)

    // Accessing iface internals
    ifaceData := (*iface)(unsafe.Pointer(&worker))
    println("- iface.tab.inter:", ifaceData.tab.inter)
    println("- iface.tab._type:", ifaceData.tab._type)
    println("- iface.tab.fun[0]:",
        unsafe.Pointer(ifaceData.tab.fun[0]))

    // Verifying the actual method address
    fmt.Printf("- (*Developer).Work address: %p\n",
        (*Developer).Work)
}
```

From the earlier section on structs, we learned how Go arranges fields in memory. In this example, we rebuild the memory layout of an interface's internal structure by defining our own `iface` struct. Then we use the `unsafe` package to cast the `worker` variable to our custom `iface` type so we can access and examine its internals. By looking directly at the internal structure, we can focus on the parts that matter.

The output below gives us an even closer look:

```
- iface.tab.inter: 0x1007a6f20
- iface.tab._type: 0x1007ae620
- iface.tab.fun[0]: 0x10076e050
- (*Developer).Work address: 0x10076e050
```

As expected, the `fun[0]` entry inside the `itab` struct points to the same memory address as the `(*Developer).Work` method: `0x10076e050`.

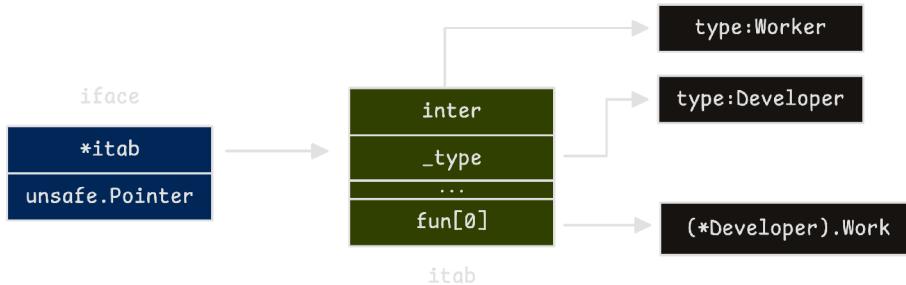


Illustration 69. Go interface method resolution via itab

Now, there's something subtle but important about the `fun` field.

Although declared as `[1]uintptr`, it's not actually limited to one method. This pattern signals that the array is variable in length. The number of entries depends on how many methods the interface defines.

This technique, known as a **variable-length array** or **runtime-sized array**, comes from C and similar low-level languages that shaped Go's internals. You'll often see it used when the number of elements isn't known at compile time. Declaring it with one element marks the starting point, but in memory, it extends further.

For instance, if the interface defines three methods, the actual memory layout will hold three function pointers. The `[1]` just indicates the base of the array.

The Go runtime uses pointer arithmetic to safely access the full array as shown below:

```
ni := len(inter.Methods)
...
methods := (*[1 << 16]unsafe.Pointer)
(unsafe.Pointer(&m.Fun[0]))[:ni:ni]
```

This allows the runtime to treat the `fun` slice as if it were declared with the correct number of entries from the start.

## Automatic Receiver Method Handling

An interesting detail that might not be immediately obvious is how the `fun[0]` field in `itab` points to the `(*Developer).Work` method, even though `Developer.Work` is actually defined with a value receiver:

```
//go:noinline
func (d Developer) Work() {
    println("%s is working", d.Name)
}
```

So, what's going on here?

The Go compiler automatically creates a wrapper method with a pointer receiver that internally calls the original value receiver. This behavior isn't limited to interfaces—it's part of Go's general method handling.

That's why you can call a method defined on a value receiver using a pointer to the type without manually dereferencing it:

```
d := &Developer{Name: "John"}
d.Work() // Works even though Work() has a value receiver
```

The compiler handles the conversion under the hood, ensuring that the method call remains valid regardless of whether the receiver is a value or a pointer.

If you're interested in how the compiler creates the pointer receiver method, we can examine the assembly it generates:

```
main.(*Developer).Work STEXT dupok size=128 args=0x8
locals=0x18 funcid=0x16 align=0x0
    0x0000 00000 (< autogenerated >:1) TEXT
main.(*Developer).Work(SB), DUPOK|WRAPPER|ABIInternal, $32-8
    ...
    0x0020 00032 (< autogenerated >:1) CBZ      R0,
64
    0x0024 00036 (< autogenerated >:1) MOVD
(R0), R2
    0x0028 00040 (< autogenerated >:1) MOVD
8(R0), R1
    0x002c 00044 (< autogenerated >:1) MOVD      R2,
R0
    0x0030 00048 (< autogenerated >:1) CALL
main.Developer.Work(SB)
    0x0034 00052 (< autogenerated >:1) LDP
-8(RSP), (R29, R30)
    0x0038 00056 (< autogenerated >:1) ADD      $32,
RSP
```

```

        0x003c 00060 (<autogenerated>:1)      RET
(R30)
        0x0040 00064 (<autogenerated>:1)      CALL
runtime.panicwrap(SB)
...

```

This function is tagged with `DUPOK|WRAPPER`, meaning it's a wrapper function generated automatically by the compiler and may be deduplicated if needed. The `<autogenerated>` label on each line further confirms that the compiler created this method as part of its internal handling. Let's go together and break down the assembly code further.

This wrapper targets the pointer receiver (`main.(*Developer).Work`) and sets up a stack frame of `48` bytes, with `8` bytes reserved for arguments:

```

TEXT    main.(*Developer).Work(SB),
DUPOK|WRAPPER|ABIInternal, $48-8

```

The method receives a single argument: a pointer to the `Developer` struct in register `R0`. The core logic begins at offset `0x0024` (36):

```

0x0020 00032 (<autogenerated>:1)  CBZ   R0, 64
0x0024 00036 (<autogenerated>:1)  MOVD  (R0), R2
0x0028 00040 (<autogenerated>:1)  MOVD  8(R0), R1
0x002c 00044 (<autogenerated>:1)  MOVD  R2, R0
0x0030 00048 (<autogenerated>:1)  CALL
main.Developer.Work(SB)
...
0x0040 00064 (<autogenerated>:1)  CALL
runtime.panicwrap(SB)

```

At this point, the `CBZ` instruction (Compare and Branch if Zero) checks whether the receiver pointer (`*Developer`) in register `R0` is `nil`.

- If `R0` is `nil`, control jumps to offset `00064` where `runtime.panicwrap` is called. This triggers a panic.
- If the pointer is valid, the method loads the full `Developer.Name` string by moving its data pointer into `R2` and its length into `R1`. Then it moves the data pointer from `R2` into `R0` to match the expected argument order. After that, it continues by calling the original value receiver method using the `CALL` instruction (`CALL main.Developer.Work(SB)`).

This mechanism ensures that whether you call the method on a value or a pointer, Go handles it safely by generating a wrapper that performs the necessary checks

and conversions.

Here's a simplified version of what the compiler-generated function looks like:

```
//go:DUPOK|WRAPPER|ABIInternal
func (d *Developer) Work() {
    if d != nil {
        (*d).Work()
    }
    runtime.panicwrap()
}
```

## How Go Builds Itabs

Now, most `itab` entries are generated at compile time. For instance:

```
var d Worker = Developer{}
```

When a `Developer` value is assigned to a `Worker` interface, the compiler prepares an `itab` that links the two types (`itab: {Developer, Worker}`) during compilation.

However, there are situations where the compiler cannot determine the type-interface relationship in advance. In such cases, `itab` must be created at runtime. This occurs when reflection is used, types are generated dynamically, or the interface conversion involves multiple levels of indirection.

Compile-time `itab` entries are stored in a dedicated binary section called `.itablink`. To confirm this, let's inspect the binary layout:

```
$ objdump -h main

main: file format mach-o arm64

Sections:
Idx Name      Size   VMA           Type
 0 __text     000665e4 0000000100001000 TEXT
 1 __symbol_stub1 0000001f8 0000000100067600 TEXT
 2 __rodata    0001ec44 0000000100067800 DATA
 3 __rodata    00012e40 0000000100088000 DATA
 4 __typelink  000003f4 000000010009ae40 DATA
 5 __itablink  00000018 000000010009b240 DATA
 6 __gosymtab   00000000 000000010009b258 DATA
 7 __gopclntab  000528d0 000000010009b260 DATA
 8 __go_buildinfo 00000150 00000001000f0000 DATA
```

```
9 __go_fipsinfo    00000078 00000001000f0160 DATA
10 __nl_symbol_ptr 00000150 00000001000f01d8 DATA
11 __noptrdata     0000098a 00000001000f0340 DATA
12 __data          00002b32 00000001000f0ce0 DATA
13 __bss           000261f0 00000001000f3820 BSS
14 __noptrbss     00003440 0000000100119a20 BSS
...
```

If you've dealt with binaries before, many of these sections may look familiar: `__text`, `__rodata`, `__data`, `__bss`, and `__noptrbss`. We'll revisit them in more detail later, but for now, focus on the `.itablink` section.

The `.itablink` section has a size of `0x18` (24 bytes) and is located at address `0x000000010009b240`. Since each pointer is 8 bytes on macOS (Darwin), this section can store exactly three `itablink` entries ( $24 / 8 = 3$ ).

And that brings up an interesting question. We only defined a single interface (`Worker`) and used it with the `Developer` type, so why do we see three `itablink` entries?

To find out, let's examine all the symbols in the binary and filter only those related to `itab`:

```
$ go tool nm theanatomyofgo | grep go:itab
10009a7b8 R go:itab.main.Developer,main.Worker
10009a7d8 R go:itab.runtime.errorString,error
10009a7f8 R go:itab.runtime/plainError,error
```

Besides our expected `Developer` to `Worker` entry, we also see two additional entries: `errorString` and `plainError`, both implementing the `error` interface.

These two are always included in Go binaries, even when the program doesn't use the `error` interface directly. They are part of the runtime and help support basic error handling.

That said, bringing in external packages increases the number of `itab` entries. For instance, replacing `println` with `fmt.Println` requires importing the `fmt` package, which adds several additional entries to the binary:

```
1000ec908 R go:itab.*errors.errorString,error
1000ecb08 R go:itab.*fmt.pp,fmt.State
1000ecac0 R go:itab.*internal/bisect.parseError,error
```

```
1000ecaa0 R
go:itab.*internal/godebug.runtimeStderr,internal/bisect.Writer
1000eca60 R
go:itab.*internal/poll.DeadlineExceededError,error
1000ec9a0 R go:itab.*io/fs.PathError,error
1000ec8c8 R go:itab.*os.File,io.Writer
1000ec9c0 R go:itab.*os.LinkError,error
1000ec9e0 R go:itab.*os.SyscallError,error
1000ece08 R go:itab.*os.fileStat,io/fs.FileInfo
1000ecb40 R go:itab.*os.unixDirent,io/fs.DirEntry
1000ed678 R go:itab.*reflect.rtype,reflect.Type
1000eca40 R go:itab.internal/poll.errNetClosing,error
1000ecf88 R
go:itab.internal/reflectlite.rtype,internal/reflectlite.Type
1000ec8e8 R go:itab.main.Developer,main.Worker
1000eca00 R go:itab.runtime.errorString,error
1000eca20 R go:itab.runtime/plainError,error
1000ec980 R go:itab.syscall.Errno,error
1000eca80 R go:itab.time.fileSizeError,error
```

These interface tables are created during compilation but initialized when the runtime starts.

As part of the runtime setup—before your `main()` function executes—Go builds a global hash table called `itabTable`. This table starts with 512 slots, as defined by `runtime.itabInitSize`. It uses a combination of the interface type and the concrete type's hash values to determine the proper storage location for each `itab`.

The runtime walks through all active modules and registers every `itab` into the global table:

Figure 68. itabsinit (src/runtime/iface.go)

```
func itabsinit() {
    ...
    for _, md := range activeModules() {
        for _, i := range md.itablinks {
            itabAdd(i)
        }
    }
    ...
}

// itabLock must be held.
func itabAdd(m *itab) {
    ...
```

```

        t := itabTable
        if t.count >= 3*(t.size/4) { // 75% load factor
            // Grow hash table.
            // t2 = new(itabTableType) + some additional
            entries
            // We lie and tell malloc we want pointer-
            free memory because
            // all the pointed-to values are not in the
            heap.
            t2 := (*itabTableType)
            (mallocgc((2+2*t.size)*goarch.PtrSize, nil, true))
            t2.size = t.size * 2

            iterate_itabs(t2.add)
            ...

            atomicstorep(unsafe.Pointer(&itabTable),
            unsafe.Pointer(t2))
            t = itabTable
        }
        t.add(m)
    }
}

```

For most programs that don't use plugins or shared libraries, there's usually just one active module: `firstmoduledata`. This includes everything from your `main` package to standard libraries such as `fmt` and `runtime`.

This initialization process is how the Go runtime knows which `itab` to use for a specific type-interface pairing.

However, not all type-interface combinations are known during compilation. When the compiler can't anticipate a pairing, the runtime handles it through **lazy initialization**. The `itab` is created only when the combination is actually used.

This logic is handled in `runtime.getitab`. If the requested `itab` isn't found in the global table, the runtime follows a clear sequence:

Figure 69. `getitab` (`src/runtime iface.go`)

```

func getitab(inter *interfacetype, typ *_type, canfail bool)
*itab {
    var m *itab

    // 1. Look in the hash table first
    t := (*itabTableType)
    (atomic.Loadp(unsafe.Pointer(&itabTable)))
}

```

```

        if m = t.find(inter, typ); m != nil {
            goto finish
        }
        ...

        // 2. If not found, create a new itab
        m = (*itab)
(persistentalloc(unsafe.Sizeof(itab{})+uintptr(len(inter.Methods)-1)*goarch.PtrSize, 0, &memstats.other_sys))
        m.inter = inter
        m._type = typ
        m.hash = 0
        m.init()
        itabAdd(m)
        ...
    }
}

```

The runtime first checks the global `itabTable` for an existing match. If found, it returns immediately. If not, it allocates a new `itab`, fills in the required fields, and registers it into the global table for future lookups.

One detail worth highlighting is this line:

```

// Allocates memory for the itab struct plus space for all
// method pointers
persistentalloc(unsafe.Sizeof(itab{})+uintptr(len(inter.Methods)-1)*goarch.PtrSize, 0, &memstats.other_sys)

```

This calculation backs up what we saw earlier: the `fun [1]uintptr` field in the `itab` struct is only a starting point. The actual size of the method pointer array depends on how many methods the interface defines.

Another important aspect is how the memory is reserved. The `persistentalloc` function allocates memory that lasts for the lifetime of the program. Anything allocated this way will not be managed by the garbage collector.

Altogether, this provides a clear view of how the Go runtime builds and manages type-to-interface relationships, both at startup and on demand.

## 2.5 Type Assertions and Runtime Behavior

A type assertion allows you to extract the concrete value stored inside an interface variable.

The syntax is simple: `value.(Type)`. This statement tells Go, "I believe the interface holds a value of this specific type." If the assertion is correct, you gain access to the underlying value and can use its fields or methods.

If the assertion is incorrect—meaning the value does not match the expected type—the outcome depends on how you perform the assertion.

Go provides two ways to write type assertions:

## Single-Value Context

This version assumes the type match will succeed without any fallback. When you write `t := value.(Type)`, Go checks the type at runtime. If the interface holds the expected type, `t` receives the value. If not, the program panics.

Use this form when you're confident the assertion will hold and prefer a crash over silent failure:

```
type Animal interface {}

type Dog struct {}

func (d Dog) Bark() {
    fmt.Println("Woof!")
}

func makeSound(a Animal) {
    dog := a.(Dog)
    dog.Bark()
}
```

## Two-Value Context

This version includes a check: `t, ok := value.(Type)`. If the assertion succeeds, `t` gets the value and `ok` is `true`. If it fails, `ok` becomes `false`, and `t` holds the zero value of the expected type.

This form avoids panics and gives you the chance to handle the failure gracefully:

```
func makeSound(a Animal) {
    if dog, ok := a.(Dog); ok {
        dog.Bark()
    } else {
        fmt.Println("Not a dog")
```

```
    }
}
```

When performing a type assertion from an interface to a concrete type, Go checks the actual type stored inside the interface. For non-empty interfaces, this check looks at `itab._type`. For empty interfaces, it compares the `_type` field directly.

The situation becomes more nuanced when you assert from one non-empty interface to another.

For example, suppose an `Animal` interface holds a `Dog`, and `Dog` also implements another interface—let's say `Barker`. You can assert from `Animal` to `Barker` like this:

```
type Barker interface {
    Bark()
}

func makeSound(a Animal) {
    if dog, ok := a.(Barker); ok {
        dog.Bark()
    } else {
        fmt.Println("Not a barker")
    }
}
```

In this case, the assertion does not check whether the underlying value is a `Dog`. Instead, it verifies whether the value implements the `Barker` interface. It's not about the exact type—it's about the capabilities the value provides.

This behavior aligns with Go's design philosophy: focus on what a type **can do**, not what it **is**. If something can bark, that's all that matters.

This kind of assertion doesn't always rely on a precomputed interface table (`itab`). The compiler can't always predict every possible type-interface combination, especially if the type isn't explicitly referenced in the code. In the example, `Dog` isn't mentioned inside `makeSound`. Instead, the runtime creates the `itab` dynamically and checks whether the concrete type (`Dog`) implements the target interface (`Barker`).

Figure 70. typeAssert (src/runtime/iface.go)

```

// typeAssert builds an itab for the concrete type t and the
// interface type s.Interface. If the conversion is not
possible, it
// panics if s.CanFail is false and returns nil if s.CanFail
is true.
func typeAssert(s *abi.TypeAssert, t *_type) *itab {
    var tab *itab
    if t == nil {
        if !s.CanFail {
            panic(&TypeAssertionError{nil, nil,
&s.Interface.Type, ""})
        }
    } else {
        tab = getitab(s.Interface, t, s.CanFail)
    }

    if !abi.UseInterfaceSwitchCache(GOARCH) {
        return tab
    }

    // Maybe update the cache to speed up future
assertions.
    if cheaprand()&1023 != 0 {
        // Update cache roughly 1 in 1000 times.
        return tab
    }

    // Load the current cache.
    oldC := (*abi.TypeAssertCache)
(atomic.Loadp(unsafe.Pointer(&s.Cache)))

    if cheaprand()&uint32(oldC.Mask) != 0 {
        // As cache grows, update it less frequently
to amortize the cost.
        return tab
    }

    // Build a new cache.
    newC := buildTypeAssertCache(oldC, t, tab)

    // Update the cache with compare-and-swap to handle
concurrency.
    atomic_casPointer((*unsafe.Pointer)
(unsafe.Pointer(&s.Cache)), unsafe.Pointer(oldC),
unsafe.Pointer(newC))

    return tab
}

```

This function has changed over time, especially with the addition of a caching mechanism designed to improve performance.

Calling `getitab` repeatedly can be costly. It involves hash lookups, memory reads, and type comparisons. To reduce this overhead, Go maintains a localized cache for faster access.

Here's how it works:

When the compiler encounters a type assertion like `a.(Barker)`, it creates a type assertion descriptor. This descriptor represents the transition from the source interface (`Animal`) to the target interface (`Barker`) and is linked to a small cache. On each call to `makeSound(a)`, the runtime checks this cache before doing anything more expensive:

Figure 71. TypeAssert (src/internal/abi/switch.go)

```
type TypeAssert struct {
    Cache    *TypeAssertCache
    Inter   *InterfaceType
    CanFail bool
}
type TypeAssertCache struct {
    Mask     uintptr
    Entries [1]TypeAssertCacheEntry
}
type TypeAssertCacheEntry struct {
    // Source type (a *runtime._type)
    Typ     uintptr
    // Target itab (a *runtime.itab), or nil if the
    // conversion would fail
    Itab    uintptr
}
```

The first time the assertion is performed, the cache will be empty. This results in a cache miss and triggers a call to `runtime.typeAssert`, which retrieves or creates the appropriate `itab` using `getitab`.

Once a valid `itab` is obtained, the runtime may update the cache—but not every time. The condition `cheaprand()&1023 != 0` ensures that cache updates happen infrequently (roughly 1 in 1000 times). This reduces overhead from unnecessary writes while still allowing popular assertions to be cached.

When the runtime decides to update, it creates a new cache like this:

```

// New cache with size 2 (minimum size)
newCache := {
    Mask: 1, // Size 2 - 1
    Entries: [
        {Typ: &dogType, Itab: &dogBarkerItab}, // New entry
    for Dog
        {Typ: 0, Itab: 0} // Empty
    slot
    ]
}

```

On later calls, this cache is checked first. If the type is found, the runtime skips the full lookup and uses the cached result. This keeps repeated assertions fast and efficient.

## 3. Generics: Type Parameters, Constraints, and Compiler Internals

### 3.1 Motivation for Generics and Basic Syntax

Suppose you need a function to determine the maximum of two floating-point numbers. A typical approach would be to accept two `float64` values. The downside is that this restricts the function to only that specific type:

```

func MaxFloat64(a, b float64) float64 {
    if a > b {
        return a
    }
    return b
}

```

One way to handle multiple types is by using interfaces:

```

switch f := arg.(type) {
case float32:
    ...
case float64:
    ...
case int:
    ...
}

```

While functional, this method is cumbersome. It involves type assertions, which introduce runtime overhead due to boxing and unboxing, and can lead to subtle

bugs.

An alternative is to write individual functions for each supported type. This eliminates the need for type assertions and keeps the implementation clear:

```
func MaxInt(a, b int) int {
    if a > b {
        return a
    }
    return b
}

func MaxInt64(a, b int64) int64 {
    if a > b {
        return a
    }
    return b
}

func MaxFloat64(a, b float64) float64 {
    if a > b {
        return a
    }
    return b
}
```

This strategy is efficient and easy to follow. However, since Go doesn't allow function overloading, you're forced to create separate versions with distinct names. It's repetitive and becomes increasingly difficult to manage, especially when maintaining shared codebases or working on public libraries.

Generics address this problem by introducing type parameters. Rather than locking a function into one type, generics define a constraint that permits a range of compatible types. In this context, you can instruct the function to work with any type that supports comparison operations—whether it's `int`, `int32`, `float32`, or a user-defined numeric type.

This approach makes the function significantly more flexible. It adapts to different types, as long as they conform to the specified constraint.

With generics, the `Max` function becomes much more concise:

```
type Ordered interface {
    Integer | Float | ~string
}
```

```
func Max[T Ordered](a, b T) T {
    if a > b {
        return a
    }
    return b
}
```

Here, the `Max` function accepts 2 parameters of the generic type `T`. This type must satisfy the `Ordered` constraint, which includes types that support standard comparison operators such as `<`, `>`, `<=`, or `>=`. That covers built-in types like `int`, `float64`, and `string`, as well as custom types that follow the same behavior.

Generics add powerful flexibility to your code. However, they also raise questions about complexity. Writing a type-specific function like one for `string` is straightforward—you immediately know what operations are allowed. In contrast, generic functions trade some transparency for versatility.

It can be harder to follow what's happening when working with more involved generic examples:

```
func DoSomething[T any, K Doer, V string](t T, retry int)
(K, V) {
    ...
}
```

This function introduces multiple layers of complexity for both the developer and the compiler. `T` accepts any type, `K` must implement the `Doer` interface, and `V` is constrained to `string`. Each of these parameters requires the compiler to validate operations, uphold type safety, and maintain efficient code generation without producing unnecessary duplication.

Generics aren't exclusive to functions. They can also be used with structs and interfaces:

```
type List[T any] struct {
    data []T
}

type Container[T any] interface {
    Get() T
    Put(item T)
}
```

In this case, `List` is a generic struct that stores a slice of any type `T`. The `Container` interface defines two methods: `Get`, which returns a value of type `T`, and `Put`, which takes an input of the same type.

We'll stay focused on functions for now, but the same generic principles extend to structs and interfaces.

## 3.2 Defining and Using Type Constraints

Before looking at constraints in detail, it's helpful to understand what you can already do with a type parameter using `any` (`[T any]`):

```
func AnyUsage[T any](t T) {
    // Assign to a variable
    a := t

    // Use with built-in functions
    b := new(T)

    // Get a pointer to a variable
    c := &t

    // Use in composite literals
    d := []T{t}

    // Assert type in a type switch
    var e interface{} = a
    switch t := e.(type) {
    case int:
        ...
    case T:
        ...
    default:
        ...
    }
}
```

This function accepts a value of any type and offers a broad range of uses. You can assign it to a variable, create a pointer, pass it to built-in functions, or include it in composite types like slices. It's also possible to use a type switch to examine its concrete type at runtime.

Although `any` provides a great deal of flexibility, it lacks guarantees about the capabilities of the types it allows.

For instance, not all types can be compared for equality:

```
func Equal[T any](a, b T) bool {
    return a == b // invalid operation: a == b
    (incomparable types in type set)
}
```

This won't compile because some types can't be compared using `==` or `!=`. When you need to ensure a type supports certain operations, you must define boundaries. That's where constraints are essential:

```
func Equal[T comparable](a, b T) bool {
    return a == b
}
```

The `comparable` constraint, introduced in Go 1.18 with generics, restricts the type parameter to types that support equality operators. By narrowing the range of acceptable types, constraints let your code safely depend on specific operations.

So, constraints define which types are allowed as type arguments in a generic function or type. They also control how those types can be used within the function body.

Constraints use standard interface syntax. For example, consider the `comparable` constraint:

Figure 72. comparable (src/builtin/builtin.go)

```
// comparable is an interface that is implemented by all
// comparable types
// (booleans, numbers, strings, pointers, channels, arrays
// of comparable types,
// structs whose fields are all comparable types).
// The comparable interface may only be used as a type
// parameter constraint,
// not as the type of a variable.
type comparable interface{ comparable }
```

Although interfaces can serve as constraints, not every constraint behaves exactly like a regular interface. There are important distinctions, which we'll go over shortly.

A common use of constraints involves creating unions of specific types:

```
type StringOrInt interface {
    string | int
}
```

```

func MustBeStringOrInt[T StringOrInt](t T) {
    fmt.Println(t)
}

func main() {
    MustBeStringOrInt("Hello, World!")
    MustBeStringOrInt(42)
    MustBeStringOrInt(1.23) // error: float64 does not
satisfy StringOrInt (float64 missing in string | int)
}

```

In this case, `StringOrInt` is a user-defined constraint that permits only `string` or `int` types. Any attempt to pass another type, such as `float64`, results in a compile-time error.

The `|` symbol is called the **union operator**. It allows you to combine multiple types into a single constraint. When using unions, keep the following rules in mind:

- Union terms must be disjoint. You can't include both `int` and `-int` since they overlap.
- A union can have up to 100 terms, but no more.
- The `comparable` constraint cannot be part of a union. For example, `interface{int | comparable}` is not allowed.

Earlier, we used the `constraints.Ordered` constraint in the `Max` function example. This constraint is defined in Go's experimental `constraints` package. It ensures that the type parameter supports ordering—that is, it allows comparisons using `<`, `>`, `<=`, or `>=`.

The constraint is defined as follows:

```

type Ordered interface {
    Integer | Float | ~string
}

type Float interface {
    ~float32 | ~float64
}

type Integer interface {
    Signed | Unsigned
}

```

```
type Signed interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}

type Unsigned interface {
    ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 |
~uintptr
}
```

The structure is straightforward. Constraints like `Ordered` are composed by combining smaller, reusable constraints such as `Float` and `Integer`. This modular approach makes type restrictions more expressive without introducing unnecessary complexity.

The `~` symbol, known as the **underlying type operator**, brings added flexibility. It allows a constraint to match a type that has a specific underlying type. For example, `~string` means the type can be a built-in `string` or any custom type defined with `string` as its base—like `type MyString string`.

However, the tilde can't be applied to named types or interfaces directly. For instance, if you define `type MyInt int`, you cannot write `~MyInt`.

As you work with constraints in Go, you'll notice they can combine methods, interfaces, union types, and underlying types. This layered system allows you to express sophisticated rules about type behavior and structure:

```
type MyConstraint interface {
    Print() string
    ~int | ~float64
}

type MyInt int

func (mi MyInt) Print() string {
    return fmt.Sprintf("MyInt: %d", mi)
}

type MyFloat float64

func (mf MyFloat) Print() string {
    return fmt.Sprintf("MyFloat: %f", mf)
}
```

In this example, `MyInt` and `MyFloat` are custom types based on `int` and `float64`, respectively. Both implement the `Print` method required by `MyConstraint`.

This approach goes beyond traditional interfaces. In classic Go, interfaces were centered around method sets. For example, the `Stringer` interface only expects a type to implement a single method: `String() string`. If a type satisfies that requirement, it fulfills the interface.

Generics have extended the role of interfaces. They now define both **behavior** through methods and **type compatibility** through constraints. It's no longer only about what a type can do—it's also about what it is underneath.

This change doesn't replace Go's original philosophy of interfaces as behavioral contracts. It builds on that foundation, giving developers a more precise and expressive way to model type requirements, while still keeping things readable and consistent.

## Constraints Are Not Interfaces

Although constraints use interface syntax, they don't behave like traditional interfaces when declaring variables:

```
type Stringer interface {
    String() string
}

type IntConstraint interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}

type AnInt int

func (i AnInt) String() string {
    return fmt.Sprintf("AnInt: %d", i)
}

var s Stringer = AnInt(42) // VALID
var i IntConstraint = AnInt(42) // error: cannot use type
IntConstraint outside a type constraint: interface contains
type constraints
```

Here, `AnInt` is derived from a basic integer type and satisfies `IntConstraint`. However, you can't use `IntConstraint` to declare a variable. Constraints are not designed for that purpose. Writing something like `var x interface{~int}` is also invalid. Constraints define rules for type parameters, not for variable declarations.

So even though interfaces are used to express constraints, constraints themselves aren't regular interfaces.

Once you add a method to a constraint, it becomes a combination of a type constraint and a behavioral constraint. This was the case in the earlier `MyConstraint` example, shown again below:

```
type MyConstraint interface {
    Print() string
    int | float64
}
```

Now the constraint does two things: it requires a method (`Print`) and enforces type structure (`int | float64`).

Go also supports **inline** or **anonymous constraints**, where you define the constraint directly where it's used:

```
func Execute[T interface {
    Print() string
    ~int | ~float64
}](t T) {
    result := t.Print()
    fmt.Println(result)
}
```

In this function, `T` must satisfy both conditions: it must have a `Print` method and its underlying type must be either `int` or `float64`. This is a combination of logical AND and OR—`T` must meet all criteria defined in the constraint block.

Interestingly, when a constraint is based solely on types, you can omit the `interface{}` wrapper entirely:

```
func Execute[T ~int | ~float64](t T) {
    ...
}
```

This version works the same way and avoids unnecessary syntax, resulting in cleaner code.

Now, we can go further with a slightly more advanced example:

```
type Constraint interface {
    int | string
}
```

```
        string | float64  
    }
```

With both union and intersection logic at play, this constraint calculates the overlap between the two sets. Since `string` appears in both, it becomes the only valid type that satisfies the constraint.

This approach allows you to define precise and flexible constraints. By intersecting typesets, you reduce the acceptable types to only those that meet every listed requirement. In this case, the intersection of `int | string` and `string | float64` yields just `string`:

```
type Constraint interface {  
    int | string  
    string | float64  
}  
  
func Intersecting[T Constraint](t T) {  
    fmt.Println(t)  
}  
  
Intersecting("Hello, World!") // VALID  
Intersecting(42) // error: cannot use 42 (untyped int  
constant) as string value in argument to Intersecting
```

But what happens if two type sets share no common elements—resulting in an empty intersection?

Go handles this scenario gracefully. Declaring a constraint that leads to an empty typeset doesn't produce an error immediately. That's because determining whether a constraint is truly empty would require checking every type across all packages and dependencies. While Go can recognize simple cases, it doesn't raise a compile-time error until you actually try to **use** a function or type constrained by that empty set.

Even a value of type `any` won't satisfy an empty constraint:

```
type EmptyConstraint interface {  
    float32  
    int  
}  
  
func Empty[T EmptyConstraint](t T) {}  
  
func main() {  
    Empty(1)      // compile error: cannot satisfy
```

```
EmptyConstraint (empty type set)
    Empty(any(1)) // compile error: cannot satisfy
EmptyConstraint (empty type set)
}
```

This illustrates how Go deals with empty type sets and concludes the section on constraints. With this core understanding in place, we can move forward.

### 3.3 Type Inference: From Arguments to Complex Relationships

Type inference lets the compiler determine types automatically, so we don't have to specify them every time:

#### Without Type Inference

```
var a int = 10 // a is
defined as int
Max[int](a, 20) // T is
explicitly set as int
```

#### With Type Inference

```
a := 10 // a is
defined as int
Max(a, 20) // T is
inferred as int
```

With inference, the compiler examines the provided values and determines the appropriate types on its own. It applies this logic to both variable declarations and function calls.

This feature plays a key role in keeping generic code concise and pleasant to use. Without inference, generics would become verbose and repetitive—something Go tries to avoid. Instead, the complexity is absorbed by the developer who designs the generic function, not the one who uses it:

```
package main

import "fmt"

// Box is a generic struct with two type parameters: T1 and
T2.
type Box[T1, T2 any] struct {
    First T1
    Second T2
}

// NewBox is a generic constructor for the Box struct.
func NewBox[T1, T2 any](first T1, second T2) *Box[T1, T2] {
    return &Box[T1, T2]{First: first, Second: second}
}
```

```
func main() {
    box := NewBox("hello", 123) // T1 inferred as
string, T2 inferred as int
    box2 := NewBox(3.14, true) // T1 inferred as
float64, T2 inferred as bool
    ...
}
```

In the example above, we define a generic struct `Box` with two type parameters, `T1` and `T2`. The `NewBox` function acts as a constructor, creating an instance of `Box` from two input values. The types for `T1` and `T2` are inferred automatically based on the arguments passed.

While writing `NewBox` requires more effort under the hood, using it stays clean and straightforward. When reading the `main()` function, you wouldn't even realize that `NewBox` is generic. That's the advantage of type inference—it simplifies usage by keeping the generic details out of the way.

Up to now, we've looked at type inference in function arguments. But Go's compiler can handle more complex situations too, including cases involving function parameters or composite types. Even with these added layers, it can often infer the correct type parameters without requiring any explicit input:

```
func Complex[T comparable, K any](t map[T]K, f func(K) T) T
{
    for k, v := range t {
        if f(v) == k {
            return k
        }
    }

    var def T
    return def
}

func main() {
    firstArg := map[string]int{"a": 1}
    secondArg := func(i int) string { return "a" }

    _ = Complex(firstArg, secondArg)
}
```

There's no need to dive into what `Complex` actually does. What matters is how smoothly the compiler infers the types of `T` and `K` from the inputs—even when

the function involves maps, function arguments, and multiple layers of logic. The compiler handles these connections automatically, reducing the burden on the developer.

## Constraint Type Inference

Type inference in Go goes beyond simply deducing types from arguments—you can also infer one type parameter based on another:

```
func Infer[T any, K []T](t T, k K) K {
    return append(k, t)
}

Infer(1, nil)           // = func Infer(t int, k []int)
[]int
Infer(1, []float64{})  // = func Infer(t float64, k
[]float64)
```

This is called **constraint type inference**. Before discussing its practical applications, let's first understand how it works behind the scenes.

In the `Infer` function, we define two type parameters: `T` and `K`. The key relationship is that `K` is constrained to be a slice of `T`, or `K = []T`. This creates a link between the two and knowing one helps determine the other.

Let's walk through the first call: `Infer(1, nil)`

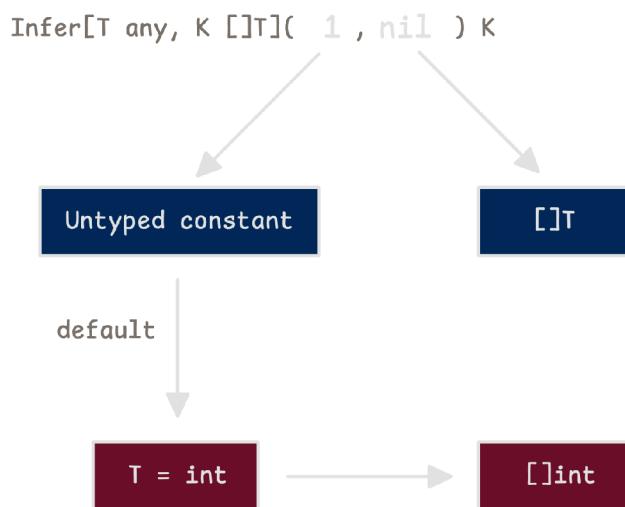


Illustration 70. Go compiler infers types from constraints

1. The compiler sees the untyped constant `1`. Without more context, it postpones assigning a specific type to `T`.
2. The second argument is `nil`, which can match any slice type. Still, there's not enough information to resolve `K`.
3. Since no additional hints are available, the compiler defaults the untyped constant to `int`, updating the type mapping to `{"T": int}`.
4. With `T` now set, and knowing that `K` must be a slice of `T`, the compiler infers `K` as `[]int`.

Now consider the second call: `Infer(1, []float64{})`

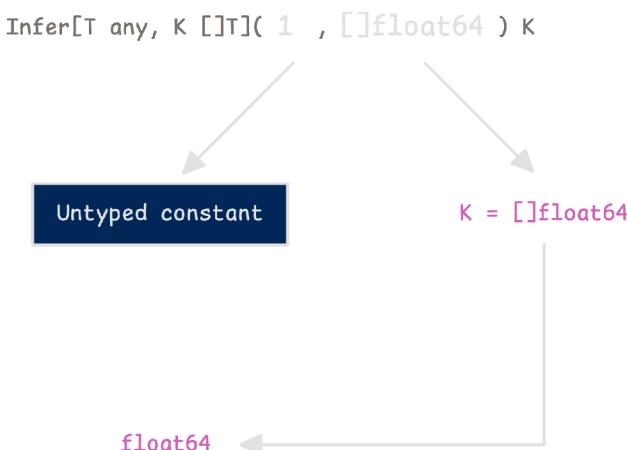


Illustration 71. Type inference from existing slice type

Here, the second argument is a concrete slice. The compiler immediately infers `K` as `[]float64`. From the constraint `K = []T`, it deduces that `T` must be `float64`. It then verifies that the constant `1` can be assigned to a `float64`, which is valid. Type inference succeeds again, but this time the inference flows in the opposite direction.

While this is a helpful demonstration, it's not something you'd write in practice. A simpler and clearer version would remove the second type parameter entirely:

```

func Infer[T any](t T, k []T) []T {
    return append(k, t)
}
  
```

That said, **constraint type inference** is a powerful feature when used appropriately. It enables more advanced generic patterns and offers a cleaner way to express type relationships without requiring redundant type information.

## Preserving Types in Generics

The Go team frequently presents a classic example that highlights this feature's usefulness. For example, imagine you need a function that filters out zero values from a slice of any type:

```
func FilterDefault[T comparable](s []T) []T {
    var result []T
    var def T

    for _, v := range s {
        if v != def {
            result = append(result, v)
        }
    }
    return result
}
```

This works well in typical cases. However, things become less straightforward when custom slice types are involved:

```
type StringSlice []string

func (s StringSlice) Dummy() {
    fmt.Println("Dummy")
}

func main() {
    s := StringSlice{"a", "b", "c", ""}
    noZeroS := FilterDefault(s)

    noZeroS.Dummy() // compiler error: noZeroS.Dummy
undefined (type []string has no field or method Dummy)
}
```

Here's the problem: when calling `FilterDefault(s)`, the input `StringSlice` is inferred as `[]string` because `T` becomes `string`. The return type ends up as a plain `[]string`, and the custom method `Dummy` is lost in the process.

Even though you're passing in a custom slice, the result is a basic slice. This breaks the expectation that the output should retain the original type and its associated behavior.

To fix this, we need to update `FilterDefault` so it returns the same type as the input. The type parameter `T` should represent the entire slice type, and we

introduce another type parameter `E` for the element type. The key is to constrain `T` so it must have an underlying type of `[]E`:

```
// previous: func FilterDefault[T comparable](s []T) []T {}  
func FilterDefault[T ~[]E, E comparable](s T) T {  
    var result T  
    var def E  
  
    for _, v := range s {  
        if v != def {  
            result = append(result, v)  
        }  
    }  
    return result  
}
```

This version is more precise, though slightly more complex. If you pass in a `StringSlice`, you get back a `StringSlice`. The method `Dummy` remains available, and the function behaves exactly as expected.

This example shows how generics can uncover subtle issues. While the surface syntax might feel simple, deeper aspects like type inference and constraints can lead to surprises if not carefully managed.

### 3.4 Compiler Internals: How Go Implements Generics

Finally, let's look at how the Go compiler implements generics. It uses a strategy called **monomorphization**—the term might sound fancy, but the concept is simple.

When you write a generic function, the compiler examines the concrete types you use and generates a specific version of that function for each type.

Take the `Max` function as an example:

- If you call it with integers, the compiler generates a version tailored for `int`.
- If you later use it with `float64`, a separate version is created for `float64`.

Translated into regular Go, it would resemble something like this:

```
func Max(a, b int) int {  
    if a > b {
```

```
        return a
    }
    return b
}

func Max(a, b float64) float64 {
    if a > b {
        return a
    }
    return b
}
```

It's like writing each version manually, but the compiler generates them for you, much like function overloading.

So when you call `Max` with an `int`, Go substitutes in the type-specific version—like `Max(int, int) int`. You don't have to define a function for every type. The compiler generates the necessary code based on usage.

Why use this approach?

By generating concrete implementations at compile time, Go avoids runtime overhead. In other languages, generic functions might rely on **virtual dispatch** to decide which version to use during execution. That adds cost at runtime. Go sidesteps this by resolving everything during compilation.

However, this method comes with a tradeoff: it can lead to **code bloat**.

Each new type used with a generic function results in a separate compiled version. This can increase binary size, add to compile time, and use more memory.

To mitigate this, Go applies a technique known as shape stenciling, often in combination with dictionaries. This approach represents a middle ground—avoiding unnecessary duplication while keeping the benefits of type safety and performance.

## Code Generation via Stenciling and Dictionaries

Building on monomorphization, Go takes a more efficient route by combining **shape stenciling** with **dictionaries** to manage generics.

- **Shape Stenciling:** Instead of generating a completely specialized version of a function for each type, the compiler creates a generic **shape** during compilation. This shared shape supports any type that fits the constraint. Rather than duplicating machine code for each variation, Go reuses a single implementation where possible.
- **Dictionaries:** Alongside the stenciled code, Go passes type-specific metadata at runtime through structures known as dictionaries. These dictionaries include method sets and behavior details for each concrete type, enabling the generic code to work correctly across multiple types.

Together, these mechanisms help reduce code duplication and keep binary sizes smaller compared to full monomorphization. While passing dictionaries introduces a slight runtime cost, the tradeoff is acceptable to preserve performance and maintain compact binaries.

If these two concepts aren't completely clear yet, that's expected. Let's walk through how they work using the `Max` function:

Go	Pseudo Go Code
<pre>func Max[T Ordered](a, b T) T {     if a &gt; b {         return a     }     return b }  Max(1, 2) Max(uint(1), uint(2)) Max(FakeInt(1), FakeInt(2))</pre>	<pre>func Max(dict *dictionary, a, b go.shape.int) {     ... }  main.Max[go.shape.int] (dict.Max[int], 1, 2) main.Max[go.shape.uint] (dict.Max:uint, 1, 2) main.Max[go.shape.int] (dict.Max[main.FakeInt], 1, 2)</pre>

Rather than generating a different function for each of `int`, `uint`, or `FakeInt` (which has the same underlying type as `int`), the compiler produces a single implementation using a common shape—like `go.shape.int`. The dictionary (`dict.Max[int]`) provides the necessary information for the specific type at runtime.

Here, `int` and `FakeInt` have the same underlying type, allowing them to share a shape. In contrast, Go generates a separate shape for `uint`.

This highlights an important distinction: even though `int` and `uint` might both be 64-bit integers (on 64-bit systems), Go treats them as separate shapes due to differences in behavior and type semantics. It's not just about size in memory—type identity and the rules that govern them also play a role.

In contrast, alias types such as `byte` and `rune` share the same shape as their underlying types (`uint8` and `int32`, respectively). Because their behavior is identical to their base types, Go does not treat them as unique shapes.

## Understanding Type Shapes and Shape Inference

The Go compiler uses an internal placeholder package called `go.shape` to manage shape types. This package isn't accessible in regular code—it exists solely for compiler use:

Figure 73. ShapePkg (src/cmd/compile/internal/types/type.go)

```
// Fake package for shape types (see typecheck.Shapify()).  
var ShapePkg = NewPkg("go.shape", "go.shape")
```

The process of assigning a shape to a type is called *shapify*. A type's **underlying type** is what determines its shape. Two types are considered to have the same shape only if their underlying representations are identical. When the compiler encounters a type that hasn't been assigned a shape, it extracts its underlying structure and assigns it under the `go.shape` namespace.

Details like named types, type aliases, or other syntactic forms do not affect the shape. Only the concrete structure matters. This process is also recursive—when shaping a composite type such as a struct, array, or map, the compiler recursively determines the shape of each component.

For example, consider passing a struct to a generic function:

```
type User struct {  
    ID   string  
    Name string  
    Age  int  
}  
  
func Print[T any](t T) {
```

```

        fmt.Println(t)
    }

Print(User{})
// -> Print[go.shape.struct {ID string; Name string; Age
int}](dict, User{})

```

Go generates a version of `Print` using the shape of the `User` struct. Any other type with the exact same structure will share this shape and can reuse the same compiled version of the function.

Here's a quick cheat sheet showing the shape of several common types:

Object	Shape
<pre> type User struct {     ID   string     Name string     Age  int } </pre>	<pre> go.shape.struct {ID string; Name string; Age int} </pre>
<code>[]int{}</code>	<code>go.shape.[]int</code>
<pre> type Worker interface {     Work() } </pre>	<pre> go.shape.interface { Work() } </pre>
<code>*User</code>	<pre> go.shape.*uint8 // or go.shape.*main.User </pre>

Pointers deserve special attention. They are often assigned a shared shape like `go.shape.*uint8`, no matter which type they point to. This generalized shape is used when the compiler doesn't require additional details about the pointed-to type:

```

func Print[T any](t T) {
    fmt.Println(t)
}

Print(&Developer{})
// -> Print[go.shape.*int8](dict, &Developer{})

```

Since the constraint is `any`, the generic function doesn't perform operations that depend on the exact type behind the pointer. In these cases, all pointer types are treated the same way. The compiler selects a minimal, universal pointer shape like `go.shape.*int8`. This allows the generated code to be reused for different pointer types (`*Developer`, `*int`, `*MyStruct`, etc.) as long as the constraint remains general.

But what happens if we change the constraint to an interface with methods and then call one of those methods?

```

type Worker interface { Work() }

func Print[T Worker](t T) {
    t.Work()
}

Print(&Developer{})
// -> Print[go.shape.*int8](dict, &Developer{})

```

We update the constraint to the basic interface `Worker` and call `t.Work()` on the type parameter. Surprisingly, the shape still doesn't change. Even though `T` now satisfies the `Worker` interface, the compiler continues to use the generalized shape `go.shape.*int8`. So how does the call to `t.Work()` succeed?

The answer lies in the dictionary (`dict`) passed to the function. When `t.Work()` is called, the compiler doesn't emit a direct method call. Instead, it uses the dictionary to look up and invoke the appropriate `Work` method for the actual type that `T` represents at runtime.

## Basic Interface

A basic interface is defined purely by its methods. It doesn't include any type constraints.

Non-basic interfaces include type restrictions. For example, `interface{ ~int }` is not a basic interface because it restricts the underlying type. These constraints define a set of allowed types and may permit additional operations.

In such cases, the compiler cannot rely on the dictionary alone. It needs to retain more detailed type information in the shape:

```
type DeveloperPtr interface { *Developer }

func Print[T DeveloperPtr](t T) {
    *t = Developer{Name: "Phuong Le", Skill: "Go"}
}

Print(&Developer{})
// -> Print[go.shape.*main.Developer](dict, &Developer{})
```

Because `DeveloperPtr` includes a specific type constraint, the compiler chooses a more accurate shape: `go.shape.*main.Developer`. This ensures the generated code has access to the full type information needed for operations beyond method dispatch, such as dereferencing the pointer or assigning a new value.

By now, the concept of shape should be clear. But there's one final piece to understand—the dictionary that gets passed to the function

## Dictionary Passing and Function Dispatch

Here's how everything comes together in practice:

```
type Developer struct {
    Name string
    Skill string
}

func (d Developer) Work() {
    fmt.Printf("%s is coding in %s\n", d.Name, d.Skill)
}

type Worker interface {
    Work()
}

//go:noinline
func CallWork[T Worker](worker T) {
    worker.Work()
}
```

```

func main() {
    worker := Developer{Name: "Phuong Le", Skill: "Go"}
    CallWork(worker)
}

```

When `CallWork(worker)` is called, the compiler generates two versions of the generic function: a **wrapper function** and a **shaped function**.



Illustration 72. Shaped version of generic function

Let's begin with the **wrapper**: `main.CallWork[main.Developer]`. This version is created for each concrete type used with the generic function. It doesn't contain the core logic—it simply sets up the call to the shaped function:

```

// WRAPPER
func main.CallWork(arg1 main.Developer) {
    dict := main.getDict(main.Developer)
    main.CallWork(dict, arg1)
}

```

This wrapper retrieves the type-specific dictionary and passes it, along with the function arguments, to the shaped version.

Now, here's the corresponding Go assembly for that wrapper:

```

main.CallWork[main.Developer] STEXT dupok size=144 args=0x20
locals=0x38 funcid=0x16 align=0x0
0x0000 00000 TEXT main.CallWork[main.Developer](SB),
DUPOK|WRAPPER|ABIInternal, $64-32
...
0x0020 00032 MOVD R0, main.worker(FP)
0x0024 00036 MOVD R2, main.worker+16(FP)
0x0028 00040 MOVD R3, R4
0x002c 00044 MOVD R2, R3

```

```

0x0030 00048 MOVD R1, R2
0x0034 00052 MOVD R0, R1
0x0038 00056 MOVD $main..dict.CallWork[main.Developer]
(SB), R0
0x0040 00064 CALL main.CallWork[go.shape.struct { Name
string; Skill string }](SB)
...

```

The wrapper function is marked with the `WRAPPER` flag and serves as the entry point for calls with concrete types. Its job is straightforward: it retrieves the dictionary for the `Developer` type and passes it, along with the actual argument, to the shaped version of the function:

```
0x0038 00056 MOVD $main..dict.CallWork[main.Developer](SB),
R0
```

Here, the compiler loads the dictionary address (`main..dict.CallWork[main.Developer](SB)`) into register `R0` from the `main` package and invokes the shaped function:

```
0x0040 00064 CALL main.CallWork[go.shape.struct { Name
string; Skill string }](SB)
```

This shaped version contains the actual logic of the generic function. It is compiled once for any type that matches the same shape. In this case, the shape corresponds to a struct with two `string` fields.

```

main.CallWork[go.shape.struct { Name string; Skill string }]
STEXT dupok size=112 args=0x28 locals=0x28 funcid=0x0
align=0x0
0x0000 00000 TEXT main.CallWork[go.shape.struct { Name
string; Skill string }](SB), DUPOK|ABIInternal, $48-40
...
0x0018 00024 MOVD R1, main.worker+8(FP) // Store
Developer.Name.data into stack
0x001c 00028 MOVD R3, main.worker+24(FP) // Store
Developer.Skill.data into stack
0x0020 00032 MOVD (R0), R5 // Load method function
pointer from dictionary
0x0024 00036 MOVD R0, R26 // Preserve dictionary
(type info) in R26
0x0028 00040 MOVD R1, R0 // Prepare
Developer.Name.data as first argument
0x002c 00044 MOVD R2, R1 // Developer.Name.len
0x0030 00048 MOVD R3, R2 // Developer.Skill.data
0x0034 00052 MOVD R4, R3 // Developer.Skill.len
0x0038 00056 CALL (R5) // Call the method for

```

```
Developer.Work({R0,R1,R2,R3})
```

```
...
```

Each `string` in Go consists of a pointer and a length, totaling 16 bytes. The `Developer` struct, having two strings, occupies 32 bytes. Adding the 8-byte dictionary pointer brings the total size of the function's arguments to 40 bytes.

You can see this reflected in the assembly metadata:

```
0x0000 00000 (main.go:21)      TEXT
main.CallWork[go.shape.struct { Name string; Skill string }]
(SB), DUPOK|ABIInternal, $48-40
```

The `$48-40` notation indicates that the stack frame uses 48 bytes in total, with 40 bytes reserved for arguments. At this stage, `R0` contains the dictionary pointer, placed there by the wrapper function.

We can represent the shaped function in pseudo-code like this:

```
// DUPOK
func main.CallWork(dict *dictionary, arg1 go.shape.struct
{Name string;Skill string}) {
    methodPtr := dict.typeParamMethodExprs[0] // get
function pointer
    methodPtr(arg1) // call the function with arg1
}
```

The dictionary's layout, conceptually, looks like this:

```
// Simplified representation of dictionary layout
type Dictionary struct {
    typeParamMethodExprs []uintptr // Method pointers for
the type parameter
    subdicts             []uintptr // Nested dictionaries
for inner generic types
    rtypes               []uintptr // Runtime type
information
    itabs                []uintptr // Interface table
    pointers             []
}
```

The key field here is `typeParamMethodExprs`, which stores pointers to the methods defined on the type used as a type parameter. For this example, it would include something like `main.Developer.Work`. These pointers are used to

perform method calls at runtime, based on the actual type passed into the generic function.

If you're familiar with how structs are laid out in Go, you might recall that the first field of a struct shares the same memory address as the struct itself. That behavior applies here: the dictionary pointer stored in `R0` points directly to the first field of the dictionary, which holds the method expression:

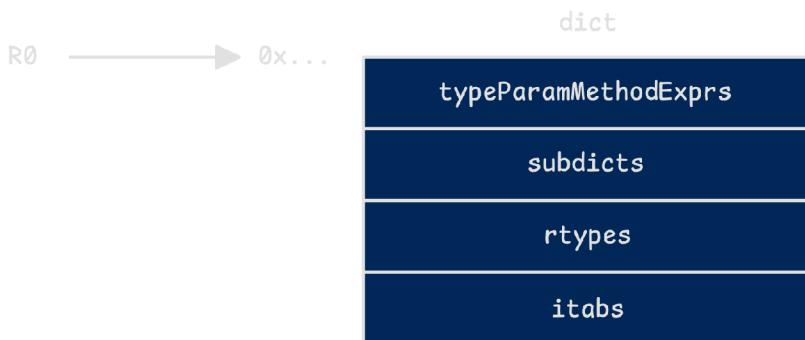


Illustration 73. Dictionary holds runtime type information

Because the first field starts at the struct's base address, it aligns exactly with the address stored in `R0`.

```
0x0020 00032 (main.go:22) MOVD (R0), R5
...
0x0038 00056 (main.go:22) CALL (R5)
```

When the assembly executes `MOVD (R0), R5`, it loads an 8-byte value from the address stored in `R0`, which is the base of the dictionary. This value is placed into register `R5` and corresponds to `dict.typeParamMethodExprs[0]`, the pointer to the `Work` method on `main.Developer`.

The following `CALL (R5)` instruction uses this pointer to invoke the method, effectively executing `Developer.Work()`.

And that's how the shaped function performs method dispatch through the dictionary. What appears to be a simple method call is actually backed by a more intricate process that links generic type handling, method resolution, and runtime behavior.

For those interested in a deeper look into these internals, Vicent Martí's article "*Generics Can Make Your Go Code Slower*" [gsc] provides a solid breakdown of how dictionaries and shaped functions interact. Just note that the article was last

updated in 2022, and Go's implementation of generics has continued to evolve since then, so some details may no longer reflect the current state.

## Summary

Understanding how Go works under the hood gives us more than just surface-level knowledge. It helps us see familiar patterns with fresh clarity, revealing not just **what** the language does, but **why** it works that way.

Before moving on, let's recap what we've covered to keep the key takeaways sharp.

## Structs

Structs organize different types of data into a single, cohesive unit. Defining a struct tells the computer exactly how to arrange data in memory for efficient access. You declare structs with fields and their types, create instances, and access fields using dot notation, like `myStruct.fieldName`.

A critical aspect of structs is memory alignment. To optimize access speed, the compiler may add padding between fields to ensure proper alignment. The order of fields affects memory usage—arranging fields from largest to smallest typically results in the most efficient layout.

Structs can be compared, but only if all their fields are of comparable types. However, two structs must share the same type name to be directly compared—unless one is an anonymous struct. Interestingly, anonymous structs can be compared with named structs if their fields match, even though two different named structs with identical fields can't be compared.

Embedding is another powerful feature. You can embed one struct within another, promoting its fields to the outer struct and allowing direct access. While this might look like inheritance in other languages, it's actually composition. Embedded structs retain their own methods, operating within their original context.

One unique feature is the empty struct (`struct{}`), which takes up no memory space. It's commonly used in maps to create sets or as a signal in concurrency patterns. However, where you place an empty struct inside another can affect

memory layout—especially when positioned at the end of a struct, where padding may be required for memory safety.

The real value of structs lies in organizing data efficiently. Understanding memory alignment, embedding, and field organization directly impacts code clarity, performance, and memory usage in real-world applications.

## Interfaces

Interfaces define behavior through a set of methods that types can implement. If a type has all the methods an interface specifies, it automatically satisfies that interface—no explicit declaration required.

For example, if an interface called `Worker` requires a `Work` method, any type that implements `Work()` automatically becomes a `Worker`. This implicit implementation keeps Go's interface system flexible and reduces dependencies across code.

Interfaces can also embed other interfaces, allowing you to build more complex behaviors. For instance, a `ReadWriter` interface can combine both `Reader` and `Writer`, requiring any implementation to have both `Read()` and `Write()` methods. This pattern is used extensively in Go's standard library.

When interfaces are embedded in structs, they require concrete implementations, leaving an interface field uninitialized triggers a runtime panic if accessed without an actual implementation.

Under the hood, interfaces consist of two main components: a type pointer and a data pointer. The type pointer holds metadata about the concrete type, while the data pointer references the actual value. When you assign a value to an interface, Go creates a copy unless it's a constant. For non-empty interfaces, Go uses **interface tables** (`itabs`) to map interface methods to their corresponding concrete type implementations. These tables, generated at compile time, ensure efficient method calls at runtime.

Empty interfaces (`interface{}` or `any`) can store values of any type since they require no methods. However, to retrieve and use a value from an empty interface, you need to extract it using type assertions or type switches.

## Generics

Generics in Go solve a common problem: writing flexible, reusable code without duplicating logic for every type. Before generics, developers had two options—write separate functions for each type or use interfaces with type assertions. Both approaches added complexity and, in the case of interfaces, introduced potential runtime overhead.

With generics, you can write a single function that works with multiple types while maintaining type safety. For instance, a `Max` function can handle integers, floats, or strings using type parameters and constraints. These constraints define which operations are allowed on type parameters, ensuring type-safe code while keeping things flexible.

Under the hood, Go implements generics using a combination of **shape stenciling** and **dictionaries**. When you define a generic function, the compiler generates two versions:

- A **wrapper version** for each specific type used with the generic function. This version prepares the necessary type-specific information.
- A **shaped version** that operates on any type matching the required shape. It relies on a dictionary to handle type-specific operations.

The dictionary contains metadata such as method pointers and runtime type information, allowing the shaped version to perform type-safe operations without generating redundant code for every type.

This system strikes a balance between code size and performance. Instead of fully specializing code for every type (full monomorphization), Go reuses the shaped version across compatible types, passing type-specific details through dictionaries. The compiler also optimizes this process by treating similar types—like pointers—as having the same shape when possible.

**Type inference** makes generics even more seamless. The compiler automatically determines the correct types based on the function arguments, so you often don't need to specify type parameters explicitly when calling a generic function.

---

At this point, we've explored the inner workings of several Go types and even touched on the more advanced topic of generics. If you're wondering about channels, we'll cover them in the concurrency section.

# References

- [rln] Go 1.14 Release Notes: <https://golang.org/doc/go1.14>
- [ovl] Proposal: Permit embedding of interfaces with overlapping method sets: <https://github.com/golang/proposal/blob/master/design/6977-overlapping-interfaces.md>
- [gsc] Generics can make your Go code slower: <https://planetscale.com/blog/generics-can-make-your-go-code-slower>
- [est] runtime: pointer to struct field can point beyond struct allocation: <https://github.com/golang/go/issues/9401>

*OceanofPDF.com*

# Chapter 5: How Go Code Turns Into Assembly

## 5.1 Inspecting the Compilation Process

Go has a really useful command, `go build -n`, that lets you peek under the hood of the build process without actually compiling anything. Instead of producing a binary, it prints out the exact steps the compiler would take. It's a great way to get a feel for what's happening behind the scenes when you build a Go application.

Let's walk through a simple example:

```
package main

import (
    "theanatomyofgo/foo"
)

func main() {
    foo.Foo()
}
```

In this example, the `main` package calls `foo.Foo()`, so we need to import our local package `theanatomyofgo/foo`. Now let's try out the special command we mentioned earlier:

```
$ go build -n .
```

The output can vary depending on the state of the build cache. If the `foo` package was already compiled, Go might just reuse the cached version instead of recompiling it.

The `-n` flag puts the build in dry-run mode; it shows you all the commands that would be executed, but without actually running them. It's totally safe to run as many times as you want without changing anything. If you want to see the build commands and actually compile the code, use the `-x` flag instead.

Alright, let's see what happens:

### Compiling the `foo` Package

Here are the commands that the Go compiler would execute to compile the `foo` package:

```
#  
# theanatomyofgo/foo  
  
$ mkdir -p $WORK/b057/  
  
$ echo '# import config' > $WORK/b057/importcfg # internal  
  
$ bin/compile -o $WORK/_pkg_.a -trimpath "$WORK/b057=>"  
-p theanatomyofgo/foo -lang=go1.23 -complete -builddid  
FZDoZhkkf1DU6lDFwHOV/FZDoZhkkf1DU6lDFwHOV -goversion go1.23.5  
-c=4 -shared -nolocalimports -importcfg $WORK/b057/importcfg  
-pack ./foo/foo.go  
  
$ bin/builddid -w $WORK/_pkg_.a # internal
```

Let's go through these steps one by one. The first command creates a temporary directory to store intermediate build files:

```
$ mkdir -p $WORK/b057/
```

The Go build system first creates a temporary build directory `$WORK/b057/`. The `$WORK` directory serves as a temporary workspace where the Go build system stores intermediate files, object files, and other build artifacts during compilation. Within this workspace, the build system creates numbered object directories for each action being executed. An action in the Go build system represents any unit of work that needs to be performed during the build process.

These directories follow a naming pattern in which each gets a sequential three-digit number starting from `001`, so you'll see directories like `b001`, `b002`, `b057`, etc.

In dry run mode, you won't see what `$WORK` is—`$WORK` serves as a placeholder. If you really want to know the exact commands and what `$WORK` looks like, use the `-x` flag instead of `-n` when running `go build`.

The `$WORK/b057` directory mentioned above corresponds to the compilation action for the `foo` package. It contains compiled object files, generated Go files, import configuration files, symbol files, and other package-specific build artifacts.

The next step is the creation of the import configuration file, commonly referred to as `importcfg`. This file serves as a map that tells the compiler where to find

compiled packages that your code imports:

```
$ echo '# import config' > $WORK/b057/importcfg # internal
```

In other words, when compiling package A that imports package B, the compiler needs to know where to find the compiled version of package B so it can access its exported types, functions, and constants.

In this case, the `importcfg` file for the `foo` package is essentially empty. It just contains a single comment line: `# import config`. That's because `foo` doesn't import any other packages, so there's nothing for the compiler to map. Later, we'll look at the `importcfg` for the `main` package, which will be more populated.

Now comes the actual compilation of the `foo` package:

```
$ bin/compile -o $WORK/b057/_pkg_.a -trimpath "$WORK/b057=>"  
-p theanatomyofgo/foo -lang=go1.23 -complete -builddid  
FZDoZhkkf1DU6lDFwHOV/FZDoZhkkf1DU6lDFwHOV -goversion go1.23.5  
-c=4 -shared -nolocalimports -importcfg $WORK/b057/importcfg  
-pack ./foo/foo.go
```

The command is a bit long, so let's break down the key parts:

\$ bin/compile -o \$WORK/b057/_pkg_.a		
import path	-p	theanatomyofgo/foo
build id	-builddid	FZDoZhkkf1DU6lDFwHOV/FZDoZhkkf1DU6lDFwHOV
dependencies	-importcfg	\$WORK/b057/importcfg
source files		./foo/foo.go

Illustration 74. Go compiler command breakdown for foo package

First, we need to tell the compiler where to write the compiled output. The `_pkg_.a` file is an archive containing the compiled package. The `.a` extension indicates this is an archive file that can contain multiple object files.

The flag `-p theanatomyofgo/foo` sets the import path of the package being compiled. This path determines how the package's exported names (such as functions and types) are uniquely identified, and how other packages can import and use them. If another package contains import `"theanatomyofgo/foo"`, then this package must be compiled with `-p theanatomyofgo/foo`. This creates the connection between the import statement and the compiled package.

The build ID is where things start to get a little more interesting. When you run `go build` on a package, and then run it again without changing any of its source files, Go will skip recompiling and just reuse the previous build result. This clever behavior is powered by the build ID.

Here's what the build ID looks like in our case:

```
FZDoZhkkf1DU6lDFwHOV/FZDoZhkkf1DU6lDFwHOV
```

It is a unique identifier consisting of two parts, separated by a slash `/`:

- **Action ID:** A hash that represents everything that went into the build—like the Go version, compiler flags, source code, build tags, dependency paths, and more.
- **Content ID:** A hash of the actual compiled output, meaning the `.a` archive or binary produced.

You might've noticed something curious: in this case, both the action ID and the content ID are the same. Why is that?

At the beginning of compilation, the actual contents of the final output aren't known yet; because the build hasn't happened. So Go temporarily uses the action ID as a placeholder for the content ID. Once the compilation finishes, Go goes back and updates the archive with the correct content hash:

That happens in this step:

```
$ bin/buildid -w $WORK/b057/_pkg_.a # internal
```

This command reads the temporary build ID from the `_pkg_.a` file, computes a hash of the file's full contents, and then rewrites the archive (`-w`) to replace the placeholder with the correct content ID.

So, does that mean the archive file contains its own build ID? Yes, and quite a bit more:

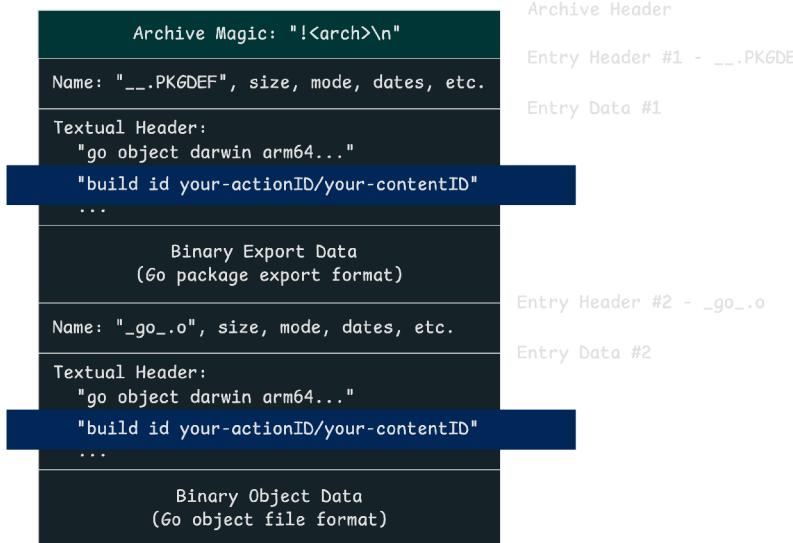


Illustration 75. Archive file contents

These `.a` archive files are at the heart of Go's build system. They store compiled packages that can later be linked into executables or imported by other packages. Each archive typically includes two key entries:

- 1. Package definition (`___.PKGDEF`)**: This section holds metadata about the package and export data, which includes details about all the functions, types, constants, and variables the package exposes to others.
- 2. Object file (`go.o`)**: This contains the compiled Go object code with the actual machine instructions and data.

## Compiling the `main` Package

Packages in Go are compiled in sequence. That means the dependencies of a package are always compiled before the package itself. In our case, the `main` package is compiled last.

Let's quickly summarize what we've learned so far:

- The compiler scans and resolves all the dependencies of the `main` package.
- Each imported package is compiled into a separate archive file (`.a`).
- After a package is compiled, its file path is added to the import configuration file (`importcfg`) so the compiler knows where to find it.
- Once all dependencies are compiled, the compiler uses the import configuration to compile the `main` package itself.

5. Finally, all compiled packages are linked together into a single executable binary that can be run.

Now let's look at how compiling the `main` package differs from compiling the `foo` package:

```
$ mkdir -p $WORK/b001/  
  
$ cat >$WORK/b001/importcfg << 'EOF' # internal  
# import config  
packagefile theanatomyofgo/foo=$WORK/b056/_pkg_.a  
packagefile runtime=$WORK/b003/_pkg_.a  
EOF  
  
$ bin/compile -o $WORK/b001/_pkg_.a -trimpath "$WORK/b001=>"  
-p main -lang=go1.23 -complete -builddid i-  
ZHRMyDc5JYCIpQhZes/i-ZHRMyDc5JYCIpQhZes -goversion go1.23.5 -  
c=4 -shared -nolocalimports -importcfg $WORK/b001/importcfg -  
pack ./main.go  
  
$ bin/builddid -w $WORK/b001/_pkg_.a # internal
```

This time, the import configuration file is not empty. It includes instructions for the compiler to locate the precompiled archive files for both the `foo` and `runtime` packages:

```
$ cat >$WORK/b001/importcfg << 'EOF' # internal  
# import config  
packagefile theanatomyofgo/foo=$WORK/b056/_pkg_.a  
packagefile runtime=$WORK/b003/_pkg_.a  
EOF
```

In some cases, instead of seeing a plain `.a` file, you might encounter a longer filename like this:

```
packagefile runtime=/Library/Caches/go-  
build/4b/4b581d4600faf043433da171b86652e79bb831bdb585418c111c  
1673b6ab4fde-d
```

What we're looking at is still just an archive file—it's been renamed and tucked away in the build cache. Go's cache uses a consistent naming pattern: `<first-two-hex-digits>/<full-hash>-<suffix>`. The cache stores two main types of files:

- Action files (with the `-a` suffix) contain metadata about build actions, including references to output files.
- Data/output files (with the `-d` suffix) store the actual compiled artifacts, such as `.a` archive files.

For now, we'll set aside the inner workings of the cache system, as it's outside the scope of this chapter.

## Linking the Final Binary

Finally, we reach the linking phase. Linking is the process of combining multiple compiled object files into a single executable program. Think of it as the final assembly step where all the pieces of your program come together. When you write a program that uses functions from different files or external libraries, each piece gets compiled separately into object files. The linker's job is to connect these pieces and resolve all the references between them.

The Go linker is a separate binary called `link` that's invoked as `go tool link`. Like the compiler, the linker uses its own import configuration file (`importcfg.link`) that maps package names to their corresponding compiled archive files:

```
$ cat >$WORK/b001/importcfg.link << 'EOF' # internal
packagefile theanatomyofgo=$WORK/b001/_pkg_.a
packagefile theanatomyofgo/foo=$WORK/b002/_pkg_.a
packagefile runtime=$WORK/b003/_pkg_.a
...
EOF

$ mkdir -p $WORK/b001/exe/

$ GOROOT='/theanatomyofgo/sdk/go1.23.5'
/tool/darwin_arm64/link -o $WORK/b001/exe/a.out -importcfg
$WORK/b001/importcfg.link -buildmode=pie -
buildid=1_SU2e8NHDbjQQdFs88j/i-ZHrMyDc5JYCIpQhZes/i-
ZHrMyDc5JYCIpQhZes/1_SU2e8NHDbjQQdFs88j -extld=clang
$WORK/b001/_pkg_.a

$ mv $WORK/b001/exe/a.out main
```

The link tool operates in two distinct modes depending on the context: internal linking mode and external linking mode. For pure Go programs, internal linking is the default and preferred option.

In internal linking mode, which we are focusing on here, the Go linker handles the entire process itself. It reads the main package's `.a` file, processes the `importcfg.link` file that maps dependencies to their archive files, loads all compiled packages, resolves symbols, performs relocations, and produces the final executable binary. Notably, it does not generate another `.a` archive file. Instead, it directly creates a platform-specific executable. On Unix-like systems such as

macOS and Linux, this is typically an ELF or Mach-O file. On Windows, it's a PE (Portable Executable) file. By default, the output is named `a.out`, though this can be changed with the `-o` flag.

External linking mode, on the other hand, introduces additional complexity. The Go linker still performs most of the work internally but then generates a temporary object file called `go.o` in a temporary directory. It also copies any host object files resulting from `cgo` compilation into this directory. Finally, it invokes an external linker such as `ld`, `clang`, or `gcc` to complete the linking process.

The most common scenario for external linking is when using `cgo` with C/C` code. When you import "C" and include C or C source files, the `cgo tool compiles them into "external objects" or "host objects." Since the Go linker cannot process these directly, it switches to external linking mode to finish the job.

So, what's the key takeaway after all this?

The unit of compilation in Go is the **package**. Each package refers to its dependencies through an import configuration file. These dependencies must be compiled first. Only then can the package itself be compiled and eventually linked into an executable.

In the next section, we will look more closely at how the compiler turns a package into an executable binary.

## 5.2 Overview of Compiling a Package

Go is a compiled language. This means the code you write is translated into machine code before it can run. The Go compiler, which is part of the official Go toolchain, handles this translation process. It takes Go source files (with a `.go` extension) and compiles them into binary executables. Go also supports cross-compilation. You can compile your code for different operating systems and architectures without needing to leave your own machine.

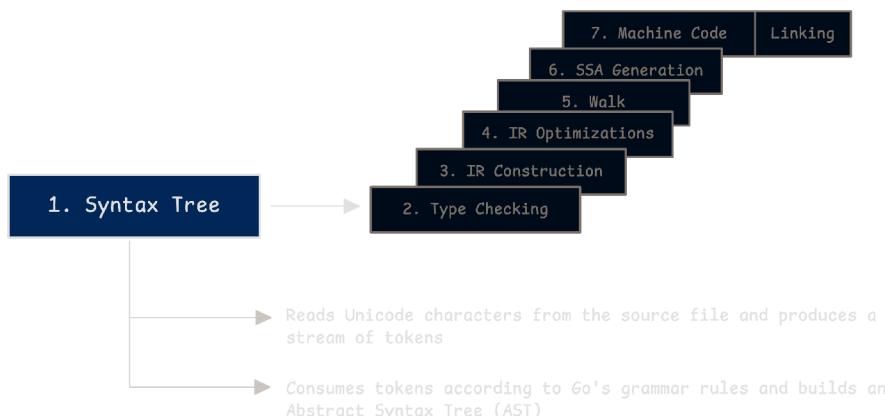
To convert high-level code into low-level instructions, the Go compiler follows a series of well-defined steps:

1. **Syntax Tree:** The compiler reads your source code and converts it into a syntax tree. This tree breaks down your program into its core components, such as declarations, expressions, and statements.

2. **Type Checking:** The compiler verifies the types of all variables, constants, functions, and expressions. It ensures types are used correctly—for example, it checks that you’re not assigning a string to an integer variable.
3. **IR Nodes Construction:** After type checking, the compiler constructs its own Intermediate Representation (IR) by converting the syntax tree and type information into a format tailored for compilation. The original AST isn’t suitable for the transformations required during later stages, so IR provides a mutable structure that supports further analysis and rewriting.
4. **IR Optimizations:** A series of optimizations are applied to the IR. These include removing dead code, inlining small functions, escape analysis (deciding between heap and stack allocation), and devirtualization (replacing interface calls with direct function calls when possible).
5. **Walk Phase:** The IR is further simplified by rewriting complex expressions into more primitive operations, preparing them for lower-level processing. For instance, a map access like `m[key]` might be rewritten into a runtime call such as `runtime.mapaccess`.
6. **SSA Generation (Static Single Assignment):** The IR is then converted into SSA form, where each variable is assigned exactly once. This makes the code easier to analyze and optimize by simplifying data flow.
7. **Machine Code & Linker:** The compiler translates the SSA form into architecture-specific assembly code. The assembler processes this into object files (`.o`), the linker then combines these with other compiled packages and the Go runtime to produce a final executable.

In general, the compiler is organized into clear phases. Each phase may have its own representation of the code, so it is easier to focus on its specific task without being constrained by the limitations of previous stages.

## 5.3 Stage 1: Parsing and Building the Syntax Tree



## Illustration 76. Compiler starts by building the syntax tree

In the first stage, the compiler reads the source code, which is usually spread across multiple files. It performs syntax parsing using limited concurrency through the `syntax.Parse()` function:

```
func LoadPackage(filenames []string) {
    base.Timer.Start("fe", "parse")

    // Limit the number of simultaneously open files.
    sem := make(chan struct{}, runtime.GOMAXPROCS(0)+10)

    ...
    // Run the syntax processing in a separate goroutine
    // to avoid blocking on the semaphore.
    go func() {
        for i, filename := range filenames {
            filename := filename
            p := noders[i]
            sem <- struct{}{}
            go func() {
                defer func() { <-sem }()
                defer close(p.err)

                fbase :=
syntax.NewFileBase(filename)
                f, err := os.Open(filename)
                ...

                // Parse the file. Errors are
                // reported via p.error.
                p.file, _ =
syntax.Parse(fbase, f, p.error, p.pragma,
syntax.CheckBranches)
                }()
            }
        }()
    ...
}
```

This parsing process (`syntax.Parse()`) involves two main components: the scanner and the parser.

## Tokenization with the Scanner

The scanner, also known as a lexer, is a common and essential component in query processing, especially in systems like SQL engines, compilers, and interpreters. It reads the raw source code and breaks it into smaller meaningful parts called *tokens*.

Tokens are the basic building blocks of the language. They include keywords (`func`, `for`, `if`), operators (`+`, `-`, `<=`), delimiters (`;`, `:`, `.`), names, and literals:

Figure 74. Scanner tokens (cmd/compile/internal/syntax/tokens.go)

```
const (
    _token = iota
    _EOF           // end of file

    // names and literals
    _Name          // identifier
    _Literal        // constant value

    // operators
    // _Operator does not include '*' (which is _Star)
    _Operator      // basic operator
    _AssignOp     // operator with assignment (like +=)
    _IncOp        // increment or decrement (like ++)

    ...
    // delimiters
    _Lparen        // (
    _Lbrack        // [
    _Lbrace        // {
    _Rparen        // )
    ...

    // keywords
    _Break         // break
    _Case          // case
    _Chan          // chan
    _Const         // const
    _Continue      // continue
    ...
    // prevents this value from being printed in .String
    tokenCount //
)
```

Let's look at a simple example to understand how the scanner works:

```
if num > 5 {
    fmt.Println("The number is greater than 5")
}
```

After scanning, the code above is transformed into a stream of tokens. These tokens are passed to the parser, which builds the abstract syntax tree (AST). For visualization, here's how the tokens might look:

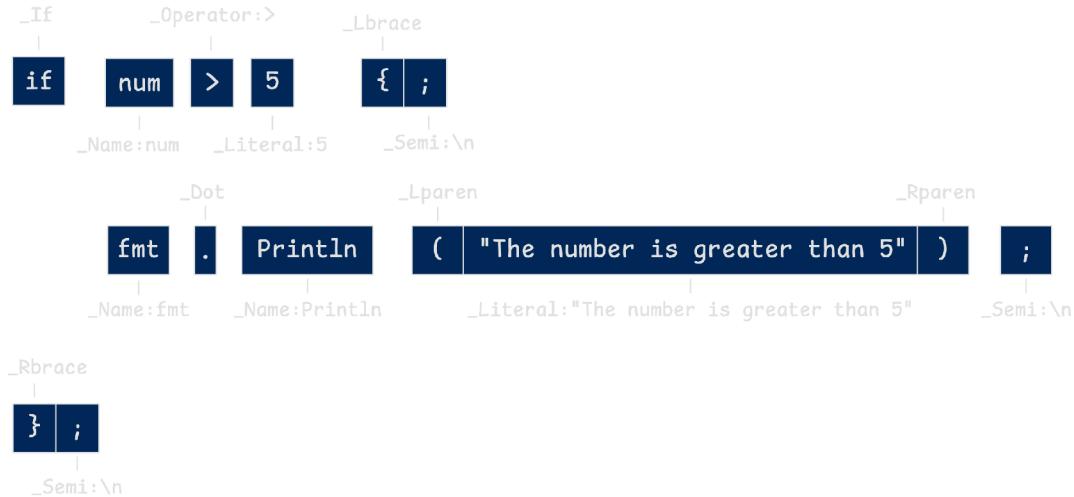


Illustration 77. Tokens corresponding to the if block's content and closing.

The scanner ignores what it considers insignificant characters. These include spaces, tabs, and newlines. It treats them as whitespace and skips over them without producing tokens.

The scanner also encounters comments in two forms: line comments starting with `//` and block comments written between `/*` and `*/`. Most comments are ignored just like whitespace. But there is a special category of comments called **directives**, which carry specific meaning for the Go compiler.

Directives are used to guide how the compiler should handle certain pieces of code. Some common examples include:

- `//go:noinline` or `/*go:noinline*/` : tells the compiler not to inline a function.
- `//go:generate` : provides instructions for generating code using an external tool.

Each token produced by the scanner has a specific type and associated metadata. For tokens like names or literals, the actual text value is stored along with the token. (You can see this in the earlier diagram.)

The special `_Semi` token explains why Go often allows you to omit semicolons at the end of statements. The scanner automatically inserts this token in two main cases:

- after a complete statement followed by a newline (resulting in `Semi:\n`), or

- at the very end of the file (resulting in `Semi:EOF`).

In practice, however, you'll rarely encounter `Semi:EOF` because the standard `go fmt` tool always ensures Go files end with a newline. Consequently, the token sequence at the file's conclusion is almost always `[_Semi:\n, _EOF]`.

## Why a line should end with a newline

This behavior aligns with the POSIX standard: a proper text file must have every line, including the last one, terminated by a newline. If the final line is missing a newline, it's treated as incomplete. This isn't just a technicality. Many text processing tools operate line by line, and when a file doesn't end with a newline, that last line might get skipped or misinterpreted.

A good example of why trailing newlines matter is `Buffer.ReadString` from the `bytes` package:

```
// ReadString reads until the first occurrence of delim in
// the input,
// returning a string containing the data up to and including
// the delimiter.
// If ReadString encounters an error before finding a
// delimiter,
// it returns the data read before the error and the error
// itself (often [io.EOF]).
// ReadString returns err != nil if and only if the returned
// data does not end
// in delim.
func (b *Buffer) ReadString(delim byte) (line string, err
error) {
    slice, err := b.readSlice(delim)
    return string(slice), err
}
```

When you use `Buffer.ReadString('\n')` to read lines from a file that doesn't end with a newline, the final line will be returned along with an `io.EOF` error. This happens because the line doesn't end with the expected delimiter (`\n`).

## Constructing the Syntax Tree with the Parser

After the scanner tokenizes your source code, the parser takes the resulting sequence of tokens. Its primary function is to construct a **syntax tree**, a hierarchical structure that reflects the code's grammar. This tree shows the code's organization,

such as how expressions are nested within statements and how statements are grouped within functions.

To understand what this tree looks like, let's expand our earlier example to show a complete and valid Go program:

```
package main

import "fmt"

func main() {
    num := 10

    if num > 5 {
        fmt.Println("The number is greater than 5")
    }
}
```

The resulting syntax tree includes details such as the function declaration (`FuncDecl`), the `if` statement (`IfStmt`), and the call to `fmt.Println` (represented within nodes like `CallExpr` and `SelectorExpr`). The parser provides a way to dump this syntax tree in a readable format.

While the Go parser can generate a detailed textual dump of this tree, the full output is often extensive, even for simple programs. The following block shows a simplified textual representation of the syntax tree for our example code, focusing on the core structure:

```
*syntax.File {
. PkgName: main
. DeclList: {
. . *syntax.ImportDecl {
. . . Path: *syntax.BasicLit { Value: "\"fmt\"" }
. . }
. . *syntax.FuncDecl {
. . . Name: main
. . . Body: *syntax.BlockStmt {
. . . . List: {
. . . . . *syntax.AssignStmt {
. . . . . . Op: :=
. . . . . . Lhs: num
. . . . . . Rhs: *syntax.BasicLit { Value: "10" }
. . . . .
. . . . *syntax.IfStmt {
. . . . . Cond: *syntax.Operation { // Represents: num > 5
. . . . . . Op: >
. . . . . . X: num
. . . . . . Y: *syntax.BasicLit { Value: "5" }
. . .
. .
. }
```

To better visualize this hierarchical structure, the diagram below presents the same syntax tree graphically:

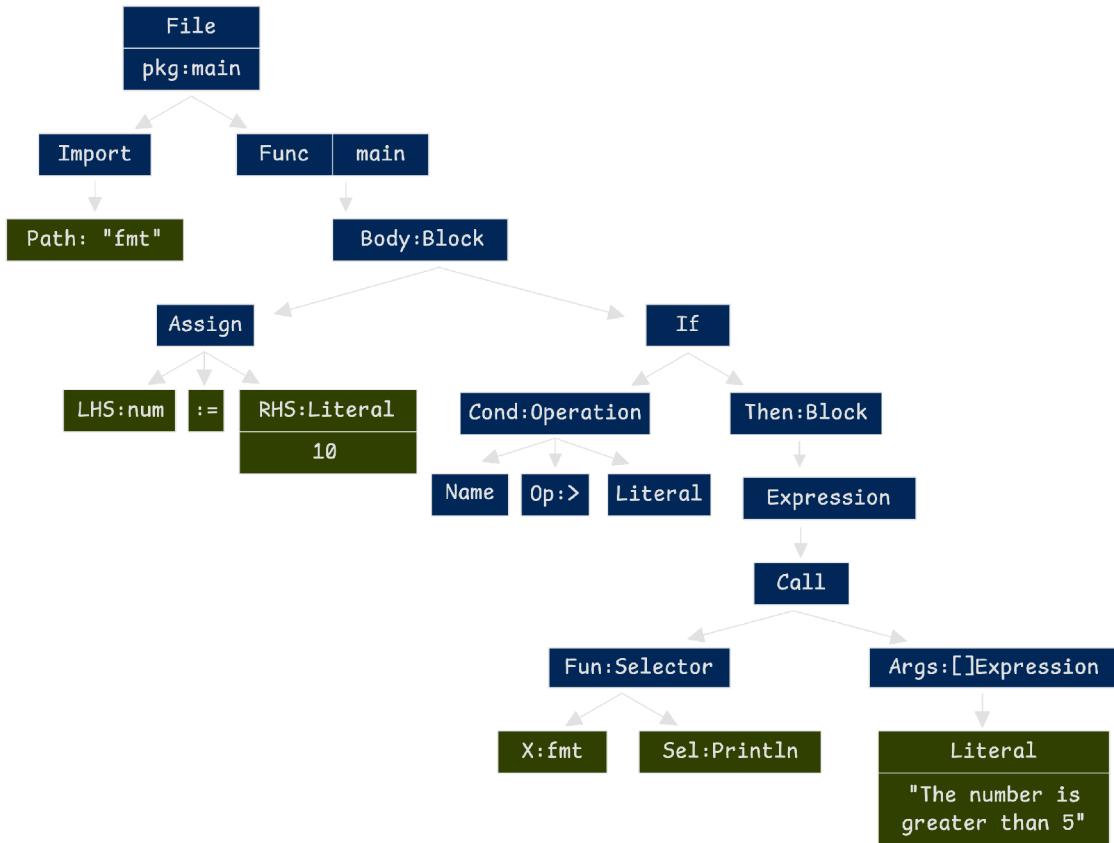


Illustration 78. Visual syntax tree of a Go program

Even though there are many nodes, most of them fall into just three main categories:

- **Expression** (`syntax.Expr`): These represent values and computations. Examples include names, literals, function calls, operators, type assertions, and composite literals. Expressions can be evaluated to produce values when the program runs.
- **Statement** (`syntax.Stmt`): These are the executable parts of the code. They include assignments, loops, conditionals, expression statements, and return statements.
- **Declaration** (`syntax.Dec1`): These define things like variables, constants, types, and functions. Declarations are how you introduce new names into your program.

Let's examine a few key node types from the `syntax` package to understand how Go source code is represented structurally:

Figure 75. Selected Node Type Definitions  
(cmd/compile/internal/syntax/nodes.go)

```

// Represents a selector expression like X.Sel
type SelectorExpr struct {
    X   Expr // Expression on the left side
    Sel *Name // Identifier being selected
    expr      // Embedded field for expression properties
}

// Represents an if statement: if Init; Cond { Then } else { Else }
type IfStmt struct {
    Init SimpleStmt // Optional initialization statement
    Cond Expr      // The condition expression
    Then *BlockStmt // The block executed if true
    Else Stmt      // Optional else block (nil, *IfStmt,
or *BlockStmt)
    stmt          // Embedded field for statement
properties
}

// Represents a function or method declaration
type FuncDecl struct {
    P Pragma      P Pragma // Documentation comments, if any
    R Recv        *Field   // Receiver (for methods), nil
for functions
    N Name         *Name    // Function or method name
    T ParamList []*Field // Type parameters (for
generics), nil if none
    Y Type         *FuncType // Signature (parameters and
results)
    B Body         *BlockStmt // Function body, nil for
forward declarations
    d decl        // Embedded field for declaration
properties
}

```

Each field within these struct definitions corresponds directly to a component of Go's grammar. For instance, the `IfStmt` struct has fields for the optional initialization (`Init`), the condition (`Cond`), the main block (`Then`), and the optional `Else` branch.

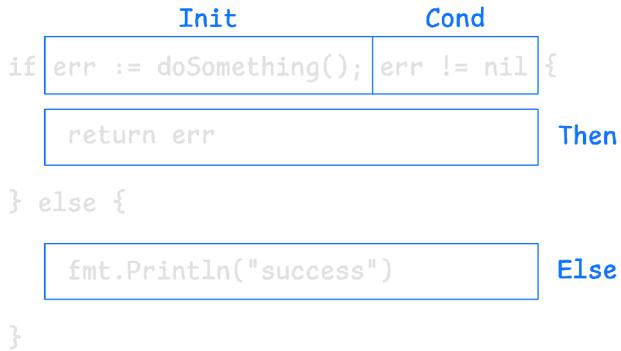


Illustration 79. Structure of an if statement in Go

The embedded types seen at the end of each struct (`expr`, `stmt`, `decl`) serve to categorize these nodes and provide common functionality.

To generate and inspect syntax trees programmatically, developers can use packages provided by the Go standard library, primarily within the `go/*` family (such as `go/parser` and `go/ast`). These packages are designed for analyzing, inspecting, or manipulating Go source code, often for building tools like linters, formatters, or code generators.

While these public packages are distinct from the compiler's internal `syntax` package shown above, they are built upon the same fundamental principles of representing Go code as syntax trees. The official Go documentation acknowledges this connection, stating that the compiler's internal APIs have gradually evolved to become more aligned with the structure and concepts familiar to users of the standard `go/*` packages.

The following Go program demonstrates how to parse source code and print its corresponding syntax tree using standard library packages:

```

package main

import (
    "go/ast"
    "go/parser"
    "go/token"
    "log"
)

// Sample Go source code provided as a string literal.
const src = `
package main

import "fmt"

```

```

func main() {
    num := 10

    if num > 5 {
        fmt.Println("The number is greater than 5")
    }
}

func main() {
    // A FileSet is needed to track position information.
    fset := token.NewFileSet()

    // Parse the source code string into an AST node.
    // The "" filename indicates the source is not from a
    file.
    // The 0 mode means no special parsing modes are
    enabled.
    fileNode, err := parser.ParseFile(fset, "", src, 0)
    if err != nil {
        log.Fatalf("Failed to parse source: %v", err)
    // Use Fatalf for formatted error
    }

    // Print the generated AST to standard output.
    ast.Print(fset, fileNode)
}

```

Key packages involved in this process:

- `go/token` : Defines constants representing lexical tokens (such as keywords and operators) and provides types for managing source code positions (`token.FileSet`, `token.Pos`).
- `go/scanner` : Reads the source text (`src`) and converts it into a sequence of tokens based on the definitions in `go/token`. (Note: `go/parser` uses the scanner internally).
- `go/parser` : Takes the token sequence generated by the scanner and builds the Abstract Syntax Tree (AST).
- `go/ast` : Defines the node types (like `ast.File`, `ast.FuncDecl`, `ast.IfStmt`) used to represent the structure of the Go code within the AST.

The `ast.Print` function provides a human-readable representation of the AST. While its output format might differ slightly from the internal representations used directly by the Go compiler, it accurately reflects the same hierarchical structure derived from the source code.

## 5.4 Stage 2: Type Checking and Scope Resolution

Type checking is the stage where the compiler verifies that all variables, expressions, and function calls follow the rules of the Go language. It also determines the type of every expression and variable in the program.

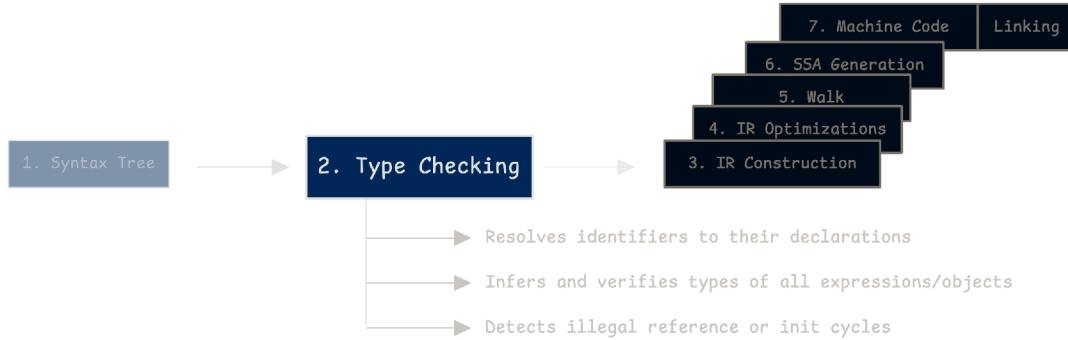


Illustration 80. Type checking stage in Go compiler

For example, the following function triggers a compile-time error. The function is supposed to return a `string`, but it actually returns an `int`:

```
func add(x int, y int) string {
    return x + y
}

// compile error:
// cannot use x + y (value of type int) as string value in
return statement
```

Since this process happens right after building the syntax tree, the syntax tree is used as the input for type checking. The type checker walks through the tree and enriches it with additional information. It transforms the raw syntax into a more structured, scope-aware representation.

When you're using your editor or IDE, you might notice that the compiler doesn't stop checking a file after finding just one error. Instead of failing fast, it tries to continue checking the rest of the code. This way, it can report multiple errors in one pass rather than forcing you to fix them one by one.

## File Validation and Package Consistency

Every Go file belongs to a package, and all files in the same package must declare the same package name. Before type checking begins, the compiler loops through all the provided files and examines their package names.

The package name is determined from the first valid file. If a file has a different package name than the one already determined, the compiler reports an error and ignores that file. If the name matches, the file is added to the list of valid files for the package.

In a directory, you can have two packages: one for the main code and one for tests. Even if the files are in the same directory, they can belong to different packages if they are part of test code.

Files that end with `_test.go` are treated as test files, and they can belong to either of the following:

- **Internal test files:** use the same package name as the code they are testing. These files are compiled along with the package and can access unexported identifiers.
- **External test files:** use a package name that ends with `_test`, such as `mypackage_test`. These are compiled as a separate package and can only access exported identifiers. They are useful for testing the public API of a package from an outside perspective.

Because of this, internal and external test files are treated as different packages and compiled separately.

To understand why this validation step matters, it's important to know how Go versions interact with packages and files. Each Go file can declare a specific minimum Go version using a `//go:build` directive. At the same time, the entire package also gets assigned a version. The compiler keeps track of both.

A file's version is determined based on any `//go:build` directive it contains. For example:

```
//go:build go1.21
```

This tells the compiler that the file requires at least Go 1.21. Different files in the same package may require different Go versions. The version of the whole package is determined by the highest Go version required by any of its files. If even one file needs a newer version than the compiler supports, the build will fail with an error. This prevents code from being compiled on an incompatible version of Go.

The `//go:build` directive was introduced in Go 1.17 [go117] to replace the older `// +build` syntax. It is more readable and flexible.

Starting from Go 1.21 [go121], this directive can also specify a minimum Go version required for the file. This helps maintain compatibility and allows conditional inclusion of code based on the Go version.

## Gathering Package-Level Declarations

The next step in the compilation process is collecting global objects. These include variables, constants, types, functions, and other top-level declarations found in the source files. When we say 'global' here, we mean anything that is declared outside of a function. These are typically placed at the top level of a file.

```
package something

const (
    true  = 0 == 0
    false = 0 != 0
)

type AnotherInt int

func append(slice []Type, elems ...Type) []Type
```

Illustration 81. Package-level declarations: const, type, and func

To organize and manage these declarations, the compiler uses scopes. A scope is a container that keeps track of declared names, such as variables or functions, so that the compiler knows where and how each name is used. There are two main types of scopes in this context:

- **Package scope:** covers all files in the package. Declarations here are visible across the whole package.
- **File scope:** is specific to one file and is mainly used for resolving names locally within that file.

Let's look at how the compiler collects these global objects:

Figure 76. collectObjects (cmd/compile/internal/types2/resolver.go)

```
func (check *Checker) collectObjects() {
    ...
    // Iterate over files to populate package and file
    scopes
    for fileNo, file := range check.files {
        ...
        // Create a file-scope
        fileScope := NewScope(pkg.scope,
```

```

syntax.StartPos(file), syntax.EndPos(file),
check.filename(fileName)
    fileScopes = append(fileScopes, fileScope)

        for index, decl := range file.DeclList {
            if _, ok := decl.(*syntax.ConstDecl);
!ok {
                first = -1 // we're not in a
constant declaration
            }

            switch s := decl.(type) {
            case *syntax.ImportDecl:
            ...
            case *syntax.ConstDecl:
            ...
            case *syntax.VarDecl:
            ...
            case *syntax.TypeDecl:
            ...
            case *syntax.FuncDecl:
            ...
            default:
                check.Errorf(s,
InvalidSyntaxTree, "unknown syntax.Decl node %T", s)
            }
        }
    }
}

```

The compiler walks through the second level of the syntax tree—just below the `File` node we saw earlier—and gathers all top-level declarations. These are then inserted into their appropriate scopes. For each file, the compiler creates a file-specific scope. But when it encounters top-level declarations such as global constants, types, or functions, it puts them into the package scope. This is why you cannot define two global variables or functions with the same name in different files of the same package. They all share the same package scope.

The purpose of this stage is to collect all the global objects. While doing so, the compiler attaches metadata to each node. For example, when processing constant declarations, it records the `iota` position within the group.

This position-based `iota` value and inheritance logic was already explained in Chapter 2, where we covered how constant declarations work.

Here is an example that defines several global constants:

```

const (
    one, two = iota, iota + 1 // 0, 1
    three      = 20
    four       = iota          // 3
)

```

This snippet results in a syntax tree with three constant declaration nodes and one function declaration node, as shown in the previous stage:

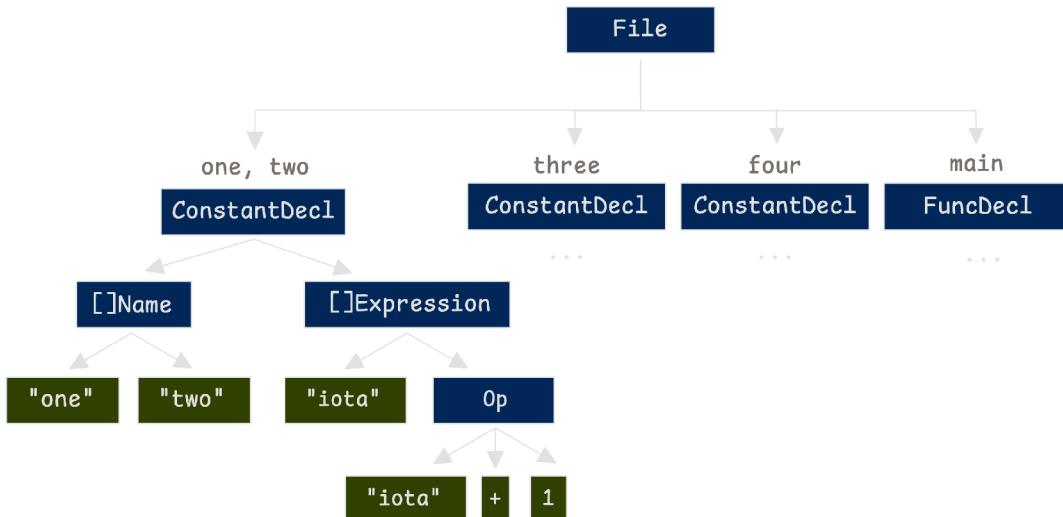


Illustration 82. Constant declaration tree for global scope

The type checker uses these declaration nodes from the syntax tree and creates 4 new `Constant` objects using `NewConst()` and one new `Function` object using `NewFunc()`:

Figure 77. `collectObjects` (cmd/compile/internal/types2/resolver.go)

```

func (check *Checker) collectObjects() {
    ...
    // Iterate over files to populate package and file
    scopes
    for fileNo, file := range check.files {
        ...
        switch s := decl.(type) {
        case *syntax.ConstDecl:
            ...
            // declare all constants
            values :=
syntax.UnpackListExpr(last.Values)
                for i, name := range s.NameList {
                    obj := NewConst(name.Pos(),
pkg, name.Value, nil, iota)

```

```

        var init syntax.Expr
        if i < len(values) {
            init = values[i]
        }

        d := &declInfo{file:
fileScope, vtyp: last.Type, init: init, inherited: inherited}
check.declarePkgObj(name,
obj, d)
    }

    ...
}

}

```

Each object is then registered into the package scope using `declarePkgObj(obj, name, d)`. This step allows the compiler to track all top-level declarations, such as constants, variables, types, and functions:

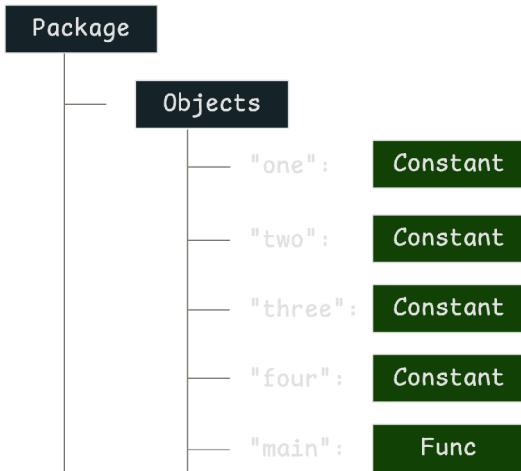


Illustration 83. Package scope with declared objects

The original syntax tree declarations (such as `syntax.ConstDecl`) are now converted into scope-based representations (such as `types2.Const`). These new representations carry additional information. They are specifically designed for the next stages of compilation, such as type checking and semantic analysis. At this point, the compiler begins to understand scopes, types, and the relationships between different parts of the code.

However, we haven't yet seen much about the file scope. This scope is used for objects that are limited to a single file and can appear in multiple files within the same package without causing "already declared" errors.

One example of file-scope objects is `import` declarations. When you import a package like `fmt` at the top of a Go file, it is only available within that file. If you want to use `fmt` package in another file, you have to import it there again.

During this stage, when the compiler processes an import declaration, it loads the associated package's export data into memory.

## Reminder about export data

We briefly touched on export data in the previous section when discussing archive files. For example, when you import the `fmt` package, your current compiling package receives its export data, which includes:

- Type information for all exported declarations, such as the `Printf`, `Println`, and `Fprintf` functions.
- Function signatures, bodies, and associated metadata.
- Generic function bodies used for instantiations.
- Escape analysis results.

The compiler reads the full export data string and parses its index structure—such as the version, flags, section boundaries, and element end positions. However, the actual elements are still stored in their encoded string form. They won't be decoded immediately. Instead, the compiler decodes each element on demand, only when that specific piece of information is needed.

This concept will become clearer in the next section, where we explore how export data is generated for the current package.

Now, the behavior can feel confusing when it comes to name conflicts between imported packages and other declarations. Let's look at an example to explain this logic better:

<b>main.go</b>	<b>another.go</b>
<pre>import "fmt"  const a = 5  func main() {     sum := add(3, a)     fmt.Println(sum) }</pre>	<pre>package main  var fmt = 5  func add(a int, b int) int {     return a + b }</pre>

**main.go**

**another.go**

The code above fails to compile with this error:

```
fmt already declared through import of package fmt
```

This might seem strange at first. If `fmt` is part of the file scope in `main.go`, why can't we declare a variable named `fmt` in a completely different file like `another.go`?

The answer is that the compiler checks for conflicts not just within a single scope, but across file and package scopes. That means file-scope objects must not conflict with anything in the package scope either.

In this case:

- `main.go` imports the `fmt` package, making `fmt` a file-scope object.
- `another.go` tries to declare a package-level variable named `fmt`.
- This creates a conflict, even though the import was in a different file.

This behavior is enforced in the `collectObjects` function, which we saw earlier. Here's the relevant part of the logic:

Figure 78. `collectObjects`: Conflict Check  
(cmd/compile/internal/types2/resolver.go)

```
func (check *Checker) collectObjects() {
    ...

    // Verify that objects in package and file scopes
    have different names
    for _, scope := range fileScopes {
        for name, obj := range scope.elems {
            if alt := pkg.scope.Lookup(name); alt
            != nil {
                ...
                if pkg, ok := obj.(*PkgName);
                ok {
                    err.errorf(alt, "%s
already declared through import of %s", alt.Name(),
                    pkg.Imported())
                    err.recordAltDecl(pkg)
                } else {

```

```

        err.errorf(alt, "%s
already declared through dot-import of %s", alt.Name(),
obj.Pkg())
err.recordAltDecl(obj)
}
check.report(&err)
}
...
}
}

```

This logic ensures that names introduced by imports (like `fmt`) do not clash with names declared elsewhere in the package.

At this stage, the compiler also begins to collect methods. These are stored in the `check.methods` map. Unlike other top-level declarations, methods are not added to the file or package scope directly. Instead, they are associated with their receiver's base type:

Figure 79. `collectObjects: Methods`  
(cmd/compile/internal/types2/resolver.go)

```

// Associate methods with receiver base type
if methods != nil {
    check.methods = make(map[*TypeName][]*Func)
    for i := range methods {
        m := &methods[i]
        // Determine the receiver base type and
        associate m with it.
        ptr, base := check.resolveBaseTypeName(m.ptr,
m.recv, fileScopes)
        if base != nil {
            m.obj.hasPtrRecv_ = ptr
            check.methods[base] =
append(check.methods[base], m.obj)
        }
    }
}

```

For example, if you have:

```

type Person struct {
    name string
}

```

```
func (p Person) GetName() string {
    return p.name
}

func (p *Person) SetName(name string) {
    p.name = name
}
```

The compiler needs to associate both `GetName` and `SetName` methods with the `Person` type.

```
check.methods[PersonTypeName] = [GetNameFunc, SetNameFunc]
```

This grouping is important because later in the compilation process, when the compiler needs to know what methods a type has, it can simply look up the type in this map and get all its methods at once. This is used for things like interface satisfaction checking, method resolution, and generating method tables.

Another special case is the `init()` function, which follows unique scoping rules. It isn't added to any specific scope, which helps avoid naming conflicts. As a result, you can define multiple `init()` functions within the same package, or even in the same file, without triggering compilation errors:

```
func init() {}
func init() {} // valid
func init() {} // valid
```

## Type Checking Top-Level Declarations

Once all global objects are collected and registered into their respective scopes, the next step is to type-check them. The compiler examines each global object to determine its type and verify that there are no conflicts or invalid declarations.

At this stage, the compiler does not type-check the function bodies. Instead, it focuses solely on top-level declarations—constants, variables, types, and functions. The goal is to establish the "skeleton" of the package (all the types, signatures, and global declarations). This ensures that when the compiler later type-checks function bodies, all global objects are guaranteed to already exist and be properly typed:

Figure 80. `packageObjects` (`cmd/compile/internal/types2/resolver.go`)

```
func (check *Checker) packageObjects() {
    ...
    var aliasList []*TypeName
```

```

    var othersList []Object // everything that's not a
type
    // phase 1: non-alias type declarations
    for _, obj := range objList {
        if tname, _ := obj.(*TypeName); tname != nil
{
            if check.objMap[tname].tdecl.Alias {
                aliasList = append(aliasList,
tname)
            } else {
                check.objDecl(obj, nil)
            }
        } else {
            othersList = append(othersList, obj)
        }
    }
    // phase 2: alias type declarations
    for _, obj := range aliasList {
        check.objDecl(obj, nil)
    }
    // phase 3: all other declarations
    for _, obj := range othersList {
        check.objDecl(obj, nil)
    }
    ...
}

```

The function above shows how the compiler type-checks global objects in three phases, based on their priority:

- 1. Non-alias type declarations:** These are processed first to make sure all base types are fully defined before anything else depends on them.
- 2. Alias type declarations:** These are processed next. Aliases depend on existing type definitions, so they must come after non-alias types.
- 3. Other declarations:** This includes constants, variables, and functions. They are handled after the types have been established.

This specific ordering helps avoid circular references or undefined type errors. For example, an alias must not point to a type that has not been type-checked yet. The main contributor in this process is the `objDecl()` function. It handles type-checking for individual objects. This means figuring out what type the object should have and making sure there are no circular dependencies that could lead to errors.

To manage this process, the Go compiler uses a color-based system to track the state of each object during type checking:

- **White (0)**: The object has not been checked yet.
- **Black (1)**: The object has been fully checked and has a valid or invalid type.
- **Grey (2 and above)**: The object is currently being checked.

This color system serves two purposes: it tracks the current state of each object and helps the compiler detect cycles in type dependencies. Let's say the type of variable `a` depends on the type of variable `b`. That means `b` is a dependency of `a`:

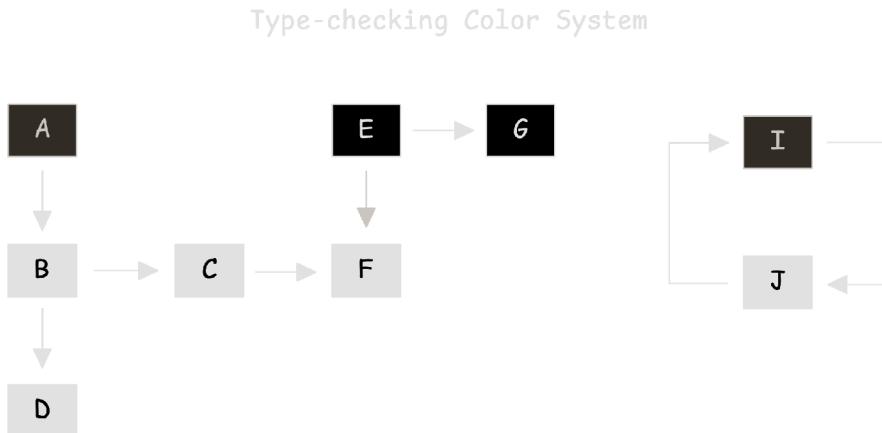


Illustration 84. Type-checking states tracked using color codes

For an object to be marked as black (fully checked), all of its dependencies must also be black. However, there is a special case where an object becomes black, but it still refers to a grey object. This indicates a cycle in the type dependencies. In such cases, the compiler detects the cycle and marks the type as invalid.

To track this process, the compiler maintains a stack of objects it is currently checking. Each time it starts checking a new object, it pushes it onto the stack and **encodes its stack position** into its grey color value:

Figure 81. objDecl (cmd/compile/internal/types2/object.go)

```

// An object may be painted in one of three colors.
// Color values other than white or black are considered
// grey.
const (
    white color = iota
    black
    grey // must be > white and black
)

func (check *Checker) objDecl(obj Object, def *TypeName) {
    ...
    switch obj.color() {
  
```

```

        case white:
            assert(obj.Type() == nil)
            // Mark object as being checked (grey) and
push to stack
            obj.setColor(grey + color(check.push(obj)))
            defer func() {
                check.pop().setColor(black)
            }()
        case black:
            // Already checked, nothing to do
            assert(obj.Type() != nil)
            return
        case grey:
            // Already being checked - cycle detection
            switch obj := obj.(type) {
            case *Const:
                if !check.validCycle(obj) || obj.typ
== nil {
                    obj.typ = Typ[Invalid]
                }
            case *Var:
                if !check.validCycle(obj) || obj.typ
== nil {
                    obj.typ = Typ[Invalid]
                }
            case *TypeName:
                if !check.validCycle(obj) {
                    obj.typ = Typ[Invalid]
                }
            case *Func:
                if !check.validCycle(obj) {
                    // Do not assign Typ[Invalid]
here.
                    // Functions are expected to
have a *Signature type.
                }
            default:
                panic("unreachable")
            }
            assert(obj.Type() != nil)
            return
        }
    }
}

```

As shown above, white is 0, black is 1, and grey **starts** from 2. The following line assigns a unique grey value based on the object's position in the stack:

```
obj.setColor(grey + color(check.push(obj)))
```

If a grey object is the first pushed onto the stack, its color is 2. If it's the second, it gets 3, and so on. To find its position in the stack, simply subtract 2 from the color value.

Consider the following package-level variable declarations:

```
var a = b
var b = d + c
var c = 10
var d int64
```

The Go compiler determines variable types by resolving their initialization dependencies. The variable `a` depends on `b`, and `b` depends on both `c` and `d`. In this group, `c` (initialized directly) and `d` (explicit type) have no further dependencies.

To guarantee a correct initialization order, the compiler processes these declarations using a depth-first search (DFS):

1. The process might start with `a`. The compiler marks `a` grey (2) and explores its dependency, `b`.
2. It marks `b` grey (3) and explores one of its dependencies, perhaps `d`.
3. Marking `d` grey (4), the compiler finds `d` has no further unresolved dependencies in this chain. Thus, the type of `d` is determined, and it's marked black (1).

At this point, the dependency graph resolution state looks like this:

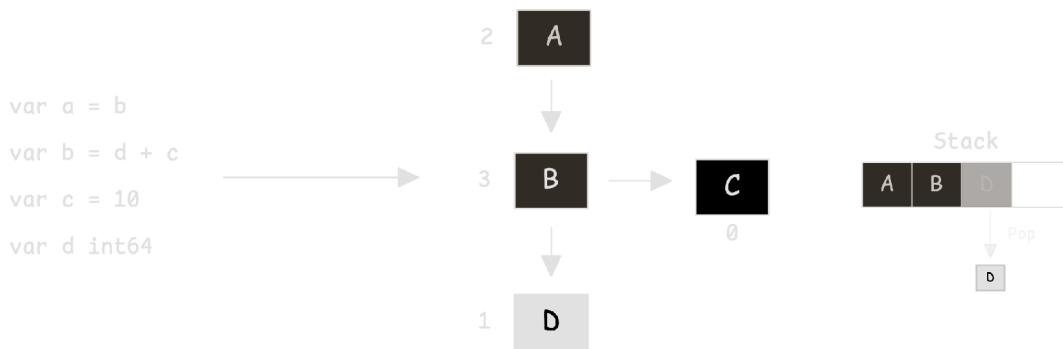


Illustration 85. Type checker resolving variable dependencies

The purpose of this encoding is to track the position of grey objects in the stack and help verify whether a detected cycle is valid.

When the compiler encounters a grey object during type checking, it means the same object has already been seen and is still being processed. This is a sign of a possible cycle. To determine if the cycle is valid or not, the checker needs to know exactly where the object first appeared in the dependency stack. The encoded color value helps identify its stack position.

However, not every cycle detected during type checking is considered invalid. This phase is only focused on determining the **type of an object**, not its initialization value.

There is an important distinction between type checking and initialization checking, which happens at a later stage. During type checking, the compiler only cares about assigning correct types to declarations. It does not evaluate or validate the actual values assigned to them.

Valid Cycle	Invalid Cycle
<code>var a int = b var b = a</code>	<code>var a = b var b = a</code>

In the first case, the cycle is valid. Variable `a` is explicitly declared as an `int`, so its type is known right away. The type of `b` is inferred from `a`, which is already typed. This creates a dependency in values, but not in types:

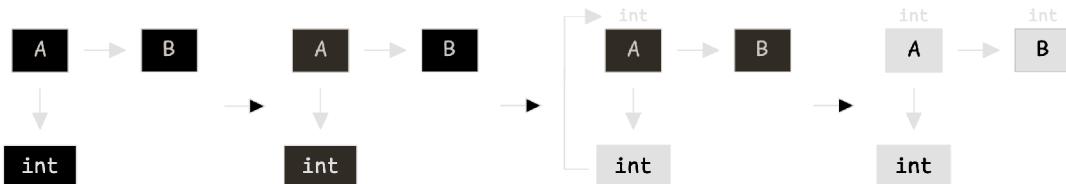


Illustration 86. Type resolution proceeds despite circular references

So even though it looks like a cycle, it's actually an **initialization cycle**, not a type cycle. The type checker does not report an error here because the types can be resolved correctly.

To sum up, the type checker and initialization checker handle different concerns:

- **Type checking:** Determines what type each variable has.
- **Initialization checking:** Determines what order variables should be initialized in, and detects circular dependencies in initialization expressions.

The initialization cycle will be caught later during the initialization check phase. That's where the compiler ensures variables do not rely on each other's values in a circular way.

Now, every object marked as 'white' (unprocessed) must undergo the full type-checking process. This process is recursive and its complexity depends on the specific declaration being checked. The primary dispatch for checking object declarations (like constants, variables, types, and functions) is the last part of `objDecl`:

Figure 82. `objDecl` Dispatcher (cmd/compile/internal/types2/decl.go)

```
func (check *Checker) objDecl(obj Object, def *TypeName) {
    switch obj := obj.(type) {
        case *Const:
            check.decl = d
            check.constDecl(obj, d.vtyp, d.init,
d.inherited)
        case *Var:
            check.decl = d
            check.varDecl(obj, d.lhs, d.vtyp, d.init)
        case *TypeName:
            check.typeDecl(obj, d.tdecl, def)
            check.collectMethods(obj)
        case *Func:
            check.funcDecl(obj, d)
        default:
            panic("unreachable")
    }
}
```

This function uses a type switch to identify the kind of object being declared (`Const`, `Var`, `TypeName`, `Func`) and then delegates to the appropriate specialized checking function—for example, `varDecl` for variables. Let's take a closer look at how the type of a variable is determined:

Figure 83. `varDecl` - Initial Checks (cmd/compile/internal/types2/decl.go)

```
func (check *Checker) varDecl(obj *Var, lhs []*Var, typ, init
syntax.Expr) {
    assert(obj.typ == nil)

    if typ != nil {
        obj.typ = check.varType(typ)
    }
}
```

```
        if init == nil {
            if typ == nil {
                obj.typ = Typ[Invalid]
            }
            return
        }
        ...
    }
```

The compiler first checks whether an explicit type (`typ`) was provided in the source code. For example:

```
var a int
```

This is considered the fast path; the compiler immediately sees that `a` is of type `int` and assigns that type to the variable object. If both the type and the initializer are missing, the compiler marks the variable's type as invalid and returns.

```
var a
```

Next, if the declaration doesn't specify a type, the Go compiler can still infer the type from the initialization expression. The `varDecl` function handles this, distinguishing between declarations with a single variable on the left-hand side (LHS) and those with multiple variables:

Figure 84. varDecl - Single Variable Initialization  
(cmd/compile/internal/types2/decl.go)

```
if lhs == nil || len(lhs) == 1 {
    assert(lhs == nil || lhs[0] == obj)
    var x operand
    check.expr(newTarget(obj.typ, obj.name), &x, init)
    check.initVar(obj, &x, "variable declaration")
    return
}
```

For a single variable, such as:

```
var x = 10 + 0.1
var d = f()
```

The key idea is that if a type isn't explicitly specified, the compiler tries to infer it from the context—usually from the initialization value (`init`). This kind of inference happens in variable declarations, assignments, and function calls. Go looks at how the value is used to determine the appropriate type.

Finally, the function addresses declarations involving multiple variables on the LHS:

Figure 85. varDecl - Multiple Variable Initialization  
(cmd/compile/internal/types2/decl.go)

```
if typ != nil {
    for _, lhs := range lhs {
        lhs.typ = obj.typ
    }
}

check.initVars(lhs, []syntax.Expr{init}, nil)
```

In this case, if an explicit type (`typ`) was provided for the declaration:

```
var a, b int
```

The type (`obj.typ`) is assigned to all variables (`lhs`) listed on the left-hand side.

This section focuses only on the flow for variable declarations (`varDecl`). For details on how constants (`constDecl`), types (`typeDecl`), and functions (`funcDecl`) are handled, see the full source code in [cmd/compile/internal/types2/decl.go](#).

The result of this process is that every variable, constant, expression, etc. in the program is assigned a well-defined type that follows the rules of Go's type system.

In addition to determining their types, the compiler also evaluates the values of constants during this stage. While variables have only their types resolved, constants receive both their types and their fully computed values.

## Type Checking of Function Bodies

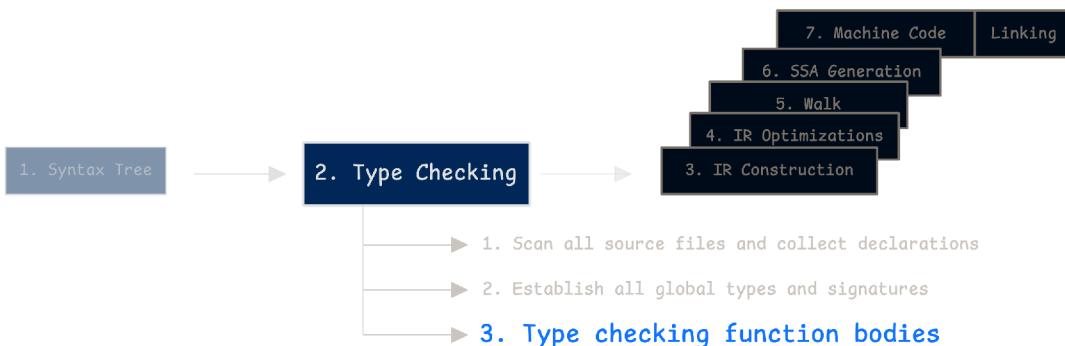


Illustration 87. From global types to full function checks

In the previous step, when we type-checked a function, the compiler only checked the function's **signature**—everything that appears in the function declaration line. This includes the receiver (if it's a method), the input parameters, and the return types.

However, the function **body** is not checked immediately. Instead, the compiler adds it to a list of delayed actions to be processed later:

Figure 86. funcDecl (cmd/compile/internal/types2/decl.go)

```
func (check *Checker) funcDecl(obj *Func, decl *declInfo) {
    ...
    // function body must be type-checked after global declarations
    // (functions implemented elsewhere have no body)
    if !check.conf.IgnoreFuncBodies && fdecl.Body != nil
    {
        check.later(func() {
            check.funcBody(decl, obj.name, sig,
fdecl.Body, nil)
        }).describef(obj, "func %s", obj.name)
    }
}
```

This delayed processing is important because the function body might refer to global declarations that haven't been fully processed yet.

The Go compiler now "pays this debt" by processing the queue of delayed actions. These actions are handled in FIFO order, but while processing each one (typically a function body), new actions can be added to the end of the queue. This often happens when the compiler encounters nested function declarations within a function body, such as function literals or closures. The loop continues until all actions are processed, resulting in a dynamic queue that grows as more nested functions are discovered.

The checker then type-checks each statement and handles the associated scope for constructs like `if`, `switch`, `for`, and so on:

Figure 87. stmt (cmd/compile/internal/types2/stmt.go)

```
// stmt typechecks statement s.
func (check *Checker) stmt(ctx StmtContext, s syntax.Stmt) {
    ...
    switch s := s.(type) {
```

```

        case *syntax.EmptyStmt:
            // ignore
        case *syntax.DeclStmt:
            ...
        case *syntax.LabeledStmt:
            ...
        case *syntax.CallStmt:
            kind := "go"
            if s.Tok == syntax.Defer {
                kind = "defer"
            }
            check.suspendedCall(kind, s.Call)

        case *syntax.ReturnStmt:
            ...
        case *syntax.IfStmt:
            ...
        case *syntax.SwitchStmt:
            ...
        case *syntax.ForStmt:
            ...
            ...
            ...

        default:
            check.error(s, InvalidSyntaxTree, "invalid
statement")
    }
}

```

As with earlier steps, we won't dive into every case here. Instead, let's highlight one example: the call statement. This includes two special forms:

- `go myFunc()` : starts a goroutine.
- `defer myFunc()` : defers execution until the surrounding function returns.

Both are handled as `CallStmt`. Regular function calls like `myFunc()` are treated differently; they're processed as expression statements, represented internally as `ExprStmt{CallExpr}`. This separation exists because `go` and `defer` statements have special execution behavior. They don't immediately execute the function like a regular call. Instead, they schedule it to run later. On the other hand, regular function calls can return values and participate in expressions.

Now, let's see how the checker handles a call statement:

```

func (check *Checker) suspendedCall(keyword string, call
syntax.Expr) {
    code := InvalidDefer
    if keyword == "go" {
        code = InvalidGo
    }
}

```

```

        }

        if _, ok := call.(*syntax.CallExpr); !ok {
            check.errorf(call, code, "expression in %s
must be function call", keyword)
            check.use(call)
            return
        }

        var x operand
        var msg string
        switch check.rawExpr(nil, &x, call, nil, false) {
        case conversion:
            msg = "requires function call, not
conversion"
        case expression:
            msg = "discards result of"
            code = UnusedResults
        case statement:
            return
        default:
            unreachable()
        }
        check.errorf(&x, code, "%s %s %s", keyword, msg, &x)
    }
}

```

First, the expression used in a `go` or `defer` statement must be a call expression (`CallExpr`). But what exactly counts as a call expression? It's any expression followed by parentheses and arguments, such as:

```

uint64(1)

make([]int, 1)

unsafe.Sizeof(1)

new(int)

copy(dst, src)

```

All of these are valid call expressions in the syntax tree. But here's the catch: not all of them are allowed in a `go` or `defer` statement. The second condition is that the call must resolve to an actual function call, not a type conversion or a special built-in.

Although these all look like function calls at a glance, they're treated differently during type checking:

- `uint64(1)` is a type conversion.

- `make`, `unsafe.Sizeof`, and `new` are built-in expressions.
- `copy` is syntactically a call expression, but semantically it's treated as a statement—and this is why it's allowed in `go` or `defer` statements. You can see this in how the compiler handles it in the `suspendedCall` function shown earlier.

When we say an expression is treated as a "conversion" or a "statement," we're not talking about whether it's an expression or a statement in a strict grammar sense. Instead, we're referring to whether it can be used in a **statement context**:

```
// These work because they're classified as "statement" kind
builtins:
copy(dst, src)
print("hello")
panic("error")

// These DON'T work because they're "expression" kind only:
len(slice)           // ERROR: "len(slice) is not used"
make([]int, 5)        // ERROR: "make([]int, 5) is not used"
append(s, x)          // ERROR: "append(s, x) is not used"

// But these are fine when used in an expression context:
n := len(slice)      // OK
s2 := make([]int, 5)  // OK
s3 := append(s, x)   // OK
```

So while all built-ins are syntactically call expressions, only a few are allowed to stand alone as statements. Go applies additional internal rules to make this distinction. Here's how the compiler classifies predeclared functions:

Figure 88. `predeclaredFuncs` (`cmd/compile/internal/types2/universe.go`)

```
var predeclaredFuncs = [...]struct {
    name      string
    nargs    int
    variadic bool
    kind     exprKind
} {
    _Append: {"append", 1, true, expression},
    _Cap: {"cap", 1, false, expression},
    _Clear: {"clear", 1, false, statement},
    _Close: {"close", 1, false, statement},
    _Complex: {"complex", 2, false, expression},
    _Copy: {"copy", 2, false, statement},
    _Delete: {"delete", 2, false, statement},
    _Imag: {"imag", 1, false, expression},
    _Len: {"len", 1, false, expression},
    _Make: {"make", 1, true, expression},
    _Max: {"max", 1, true, expression},
```

```

    _Min:      {"min", 1, true, expression},
    _New:      {"new", 1, false, expression},
    _Panic:    {"panic", 1, false, statement},
    _Print:    {"print", 0, true, statement},
    _Println:   {"println", 0, true, statement},
    _Real:     {"real", 1, false, expression},
    _Recover:   {"recover", 0, false, statement},

    _Add:       {"Add", 2, false, expression},
    _Alignof:   {"Alignof", 1, false, expression},
    _Offsetof:  {"Offsetof", 1, false, expression},
    _Sizeof:    {"Sizeof", 1, false, expression},
    _Slice:     {"Slice", 2, false, expression},
    _SliceData: {"SliceData", 1, false, expression},
    _String:    {"String", 2, false, expression},
    _StringData: {"StringData", 1, false, expression},

    _Assert:   {"assert", 1, false, statement},
    _Trace:    {"trace", 0, true, statement},
}

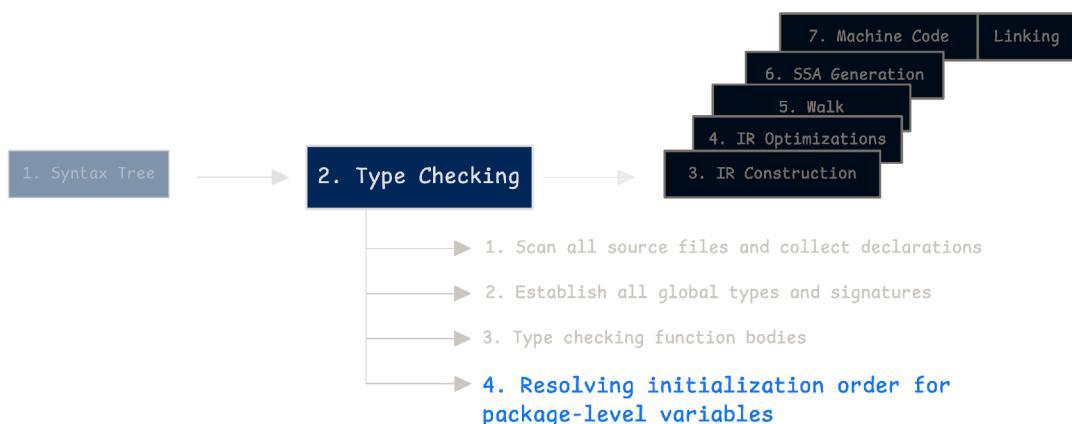
```

A simple way to identify which ones can be used with `go` or `defer` is by checking their return behavior:

- If the built-in function call produces a value, it is treated as an expression and cannot be used with `go` or `defer`.
- If it does not produce a value, it is treated as a statement and can be used with `go` or `defer`.

This is how Go prevents invalid uses of conversions or value-producing expressions inside `go` or `defer` statements, even though they might look like normal function calls.

## Resolving Initialization Order for Package-Level Variables



## Illustration 88. Resolving package-level initialization dependencies

Local variables of function are initialized in the order they are declared. This makes it simple for the Go compiler to check them sequentially. Variable scope is also more strictly controlled within function blocks.

### Package-level variables

```
// valid: 'b' is
initialized before 'a'
// even though it's
declared after 'a' in the
source code
var a = b
var b = 10
```

### Function-level variables

```
func main() {
    // invalid: 'b' is
    not initialized yet,
    compile error
    a := b
    b := 10
}
```

Package-level variables or global variables don't follow this simple sequential rule. The Go compiler must analyze dependencies and determine the correct order of initialization. This final step in the compilation process is responsible for that.

We say 'package-level objects' instead of just 'variables' because this step also applies to constants and functions, which can participate in initialization dependencies:

```
var a = initA()

func initA() int {
    return a
}

// compile error: initialization cycle for a
```

To solve this, the compiler builds a new structure: a **dependency graph**. This graph represents the dependencies between all package-level objects. Then, a **priority queue** is used to process nodes in an order that respects those dependencies. If a cycle is found in this graph, the compiler reports an initialization cycle error.

If that sounds confusing, let's walk through a happy-case example (with no dependency cycle) to make it clearer:

```
var a = f()
var b = a + 1

func f() int {
```

```

    return c
}

var c = 3

```

The process begins by identifying every global object as a node, then drawing connections (edges) between them based on who depends on whom:

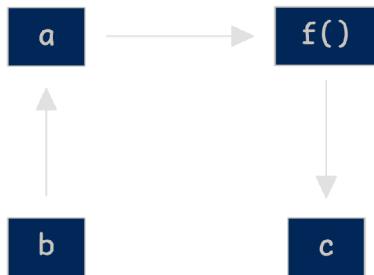


Illustration 89. Dependency graph of global object initialization

This graph shows the dependencies between the global objects. However, functions are special—they can affect the initialization order of other objects, but they themselves do not need to be initialized.

So, the checker removes functions from the graph while still **preserving the dependencies** they contribute:

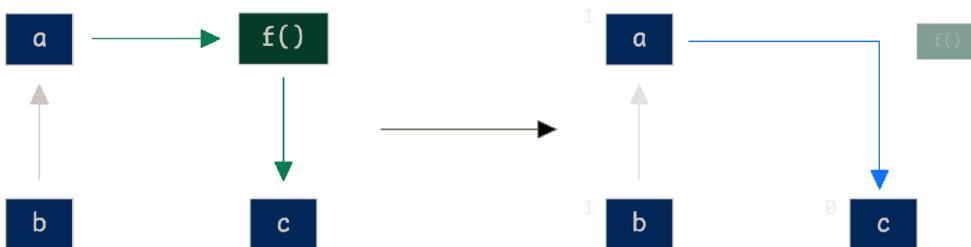


Illustration 90. `f()` replaced with its downstream links

In the original graph, `a` depends on `f()`, and `f()` depends on `c`. After removing `f()`, the compiler creates a direct dependency from `a` to `c`.

At this point, the Go compiler also tracks how many dependencies each node has. It uses this information to determine the final initialization order by repeatedly selecting and removing the node with the fewest remaining dependencies. In the graph above, `c` has the fewest dependencies—ideally zero—so it is initialized first. After removing `c`, `a` no longer has any dependencies and becomes the next node to initialize.

This process continues until all nodes are processed. The final result is the correct order in which global variables should be initialized. This logic is implemented in the `initOrder` function:

Figure 89. `initOrder` (`cmd/compile/internal/types2/initorder.go`)

```
func (check *Checker) initOrder() {
    check.Info.InitOrder = check.Info.InitOrder[:0]

    // build the dependency graph
    // also remove functions from the graph
    pq := nodeQueue(dependencyGraph(check.objMap))
    heap.Init(&pq)
    ...

    for len(pq) > 0 {
        // get the next node with the fewest
        dependencies (highest priority)
        n := heap.Pop(&pq).(*graphNode)

        // if n still depends on other nodes, we have
        a cycle
        if n.ndeps > 0 {
            cycle := findPath(check.objMap,
n.obj, n.obj, make(map[Object]bool))
            if cycle != nil {
                check.reportCycle(cycle)
            }
        }

        // reduce dependency count of all dependent
        nodes
        // and update priority queue
        for p := range n.pred {
            p.ndeps--
            heap.Fix(&pq, p.index)
        }

        // skip if it's not a variable or has no
        initializer
        v, _ := n.obj.(*Var)
        info := check.objMap[v]
        if v == nil || !info.hasInitializer() {
            continue
        }
        ...

        // create an `Initializer` object and store
        it
        // in the final initialization order
        infoLhs := info.lhs
```

```

        if infoLhs == nil {
            infoLhs = []*Var{v}
        }
        init := &Initializer{infoLhs, info.init}
        check.Info.InitOrder =
append(check.Info.InitOrder, init)
    }
    ...
}

```

The checker always picks the node with the fewest dependencies from the priority queue (`heap`). When a variable is ready for initialization, its dependency count should be `0`, meaning everything it depends on has already been initialized.

If the node with the lowest dependency count still has remaining dependencies (`ndeps > 0`), that indicates an initialization cycle. For instance, all the nodes in the following graph have a dependency count greater than `0` and form a cycle:

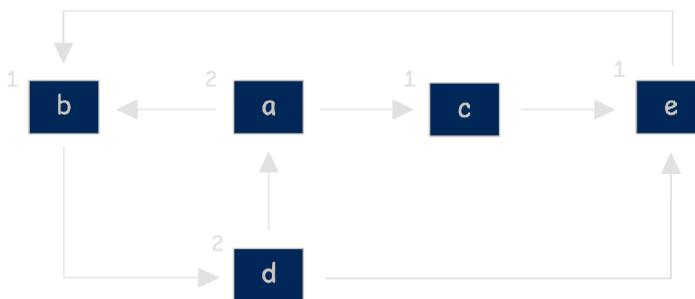


Illustration 91. Cycle detected if dependencies remain

Finally, the checker only records the initialization order of variables, because their initialization values are computed at runtime.

There are a few minor cleanup steps after this stage, but they are skipped here to keep the explanation focused.

## 5.5 Stage 3: IR Construction

The Go compiler includes a middle stage, often referred to as the "middle end." This stage doesn't just work directly with the original syntax tree or the scope-based objects created during earlier phases. Instead, it uses a specialized structure called *Intermediate Representation* (IR).

During this phase, the compiler builds its core IR through two key steps: generating export data and constructing IR nodes from that data.

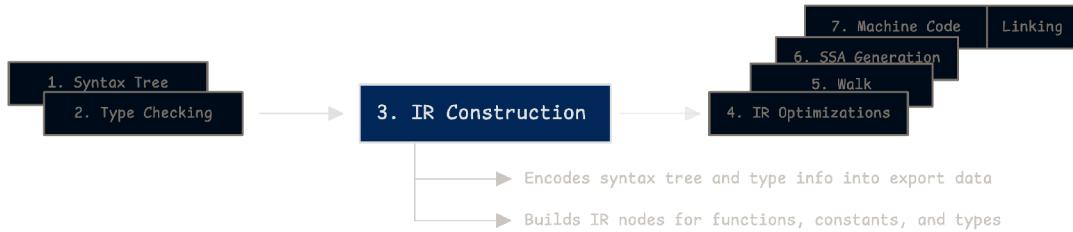


Illustration 92. Compiler builds IR from syntax and types

To understand what export data is, let's revisit something we mentioned earlier in this chapter: the archive file (`.a` file). This file contains *export data* that other packages rely on. You may also recall that during the type-checking phase, when the compiler processes an `import` statement, it uses an importer to load the export data from the archive file of the imported package. That way, when it encounters a reference to something from another package—like a function call—it can look up the necessary information without needing the original source code.

Here are a few key terms to clarify:

- **Export data:** Information about a package that the compiler makes available to other packages, so they can import it without access to the source code.
- **Unified IR (Intermediate Representation):** The structured format used to represent export data internally.
- **Bitstream encoding:** The method used to serialize the Unified IR into a compact format (bitstream) and later deserialize it when needed.

At this stage, the Go compiler encodes the syntax tree (from stage 1) and the type information (from stage 2) into a bitstream format to produce export data. While the process of generating export data is relatively complex, it's a fascinating part of compiler design—especially in how it compresses rich type and syntax information into a compact form. This involves some understanding of encoding techniques, but we'll keep things visual and intuitive with the help of diagrams. It's also worth noting that this part of the compiler is fairly self-contained and doesn't deeply interact with other concepts discussed in this book.

## Export Data & Relocations

Export data is a serialized form of a package's public interface, along with other information needed by any packages that depend on it. This data includes:

- Type definitions for all exported declarations.
- Function signatures and bodies for functions that might be inlined.

- Generic function bodies that may be instantiated in other packages.
- Escape analysis results for function parameters and the inlining cost of each function.

To generate this export data, the Go compiler needs two main inputs: the syntax tree and the type-checked objects. The syntax tree gives structural information, while the type-checked objects contain all type-related information collected in earlier stages.



Illustration 93. From syntax to exportable bitstream

For example, when handling a function, the compiler uses the function's type (its signature) from the type-checking phase and its full body from the syntax tree.

While processing the code, the compiler must keep track of elements like types, functions, variables, and constants, as well as how they relate to each other. Many of these elements are defined once but referenced many times throughout the code.

To manage these references efficiently, the Go compiler uses a system called *relocations*.

Imagine your program contains two identical string values, like `"foo"`. Without relocations, the compiler would store `"foo"` twice in memory—wasteful and unnecessary. Instead, the compiler stores the string once and replaces each usage with a reference to its single location. This method is called a **relocation**.

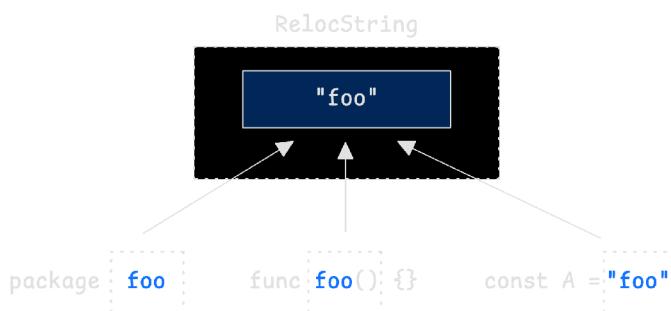


Illustration 94. "foo" stored once, referenced multiple times

Relocations work as a two-part system: the actual data (like strings) is stored in dedicated sections—such as the string section—and other parts of the program use **relocation entries** to point to that data.

For example, if a function signature needs to reference its name, it doesn't embed the string directly in the function's bitstream. Instead, it writes a relocation entry, which typically consists of just two numbers (the string relocation type and the index in the string section) pointing to the location of that string in the string section. The key insight is that relocations provide both deduplication and indirection. The string `"foo"` exists only once in the string section, but it can be referenced as many times as needed through relocations.

If that's still a bit abstract, think of it like writing a book. Instead of copying the full text of another chapter into the current one, you simply write "see Chapter 5." That reference is your relocation—a pointer to content stored elsewhere.

This same idea applies across all kinds of data in the export process—not just strings. At this point in compilation, the Go compiler uses ten types of relocations:

Figure 90. Relocation types (src/internal/pkgbits/reloc.go)

```
const (
    RelocString     RelocKind = iota // 0
    RelocMeta                   // 1
    RelocPosBase                // 2
    RelocPkg                    // 3
    RelocName                   // 4
    RelocType                   // 5
    RelocObj                    // 6
    RelocObjExt                 // 7
    RelocObjDict                // 8
    RelocBody                   // 9

    numRelocs = iota
)
```

Objects can contain relocations that point to other objects, and those other objects can themselves contain relocations pointing to yet other objects. Let's say you have a type like `*int`. In the export data, this gets broken down into separate pieces:

```
Element #1 (PointerType):
- Kind: "pointer"
- ElementType: <RELOCATION to Element #2>

Element #2 (BasicType):
- Kind: "basic"
- Name: "int"
```

Instead of embedding the entire `int` type definition inside the pointer type, `Element #1` contains a relocation entry that says "my element type is whatever you find at `Element #2`."

## Generating Export Data

When the encoding phase finishes, you're left with a complete set of objects representing the package's public interface, each stored in its own indexed location. For example, if you want to read a function signature, you can jump straight to cell #42 in the "function signature" relocation. If that signature references a type, it includes a relocation that says, "the return type is in cell #17 of the type relocation", so you go directly to cell #17 without scanning through unrelated data.

This design enables random access to the export data. It's fundamentally different from a linear format, where you'd need to read through everything in sequence to find what you're looking for.

Before we move on, let's set up the staged example that we'll use throughout this section. We have two packages: the `main` package imports the `foo` package:

```
package main

import "theanatomyofgo/foo"

const a = foo.A + 1
```

```
package foo

const A = 1
```

## Encoding Package Element

A bitstream is just a sequence of bytes that encodes information in a compact binary format. Instead of storing data as readable text, the compiler packs it tightly into bytes using variable-length encoding for numbers, single bytes for booleans, etc. For example, instead of storing the text `func Add(a int, b int) int`, it might encode this as a sequence of bytes representing the function name, parameter types, and return type.

Let's see how this works in practice with our example. The Go compiler encodes the `main` package—along with any imported dependencies—into the bitstream:



Illustration 95. Compact layout for package element

When encoding the `main` and `foo` packages:

- The `main` package has an empty import path ( `""` ) and its package name is `main`.
- The `foo` package has a path of `"theanatomyofgo/foo"` and a package name of `foo`.

At this point, we have 4 strings that need to be encoded. That means 4 string objects in string section:

RelocString (0)	
0	<code>""</code>
1	<code>"main"</code>
2	<code>"theanatomyofgo/foo"</code>
3	<code>"foo"</code>

Illustration 96. String deduplication via reloc index

Consider referencing the package path string `"theanatomyofgo/foo"`. Instead of embedding this raw string directly into the export data where it's referenced, the compiler stores a compact relocation entry consisting of two integers:

1. A relocation type code indicating it's a string reference (e.g., `0`).
2. An index pointing to the actual string's unique entry within a separate string table (e.g., `2` if `"theanatomyofgo/foo"` is the third string added to the table).

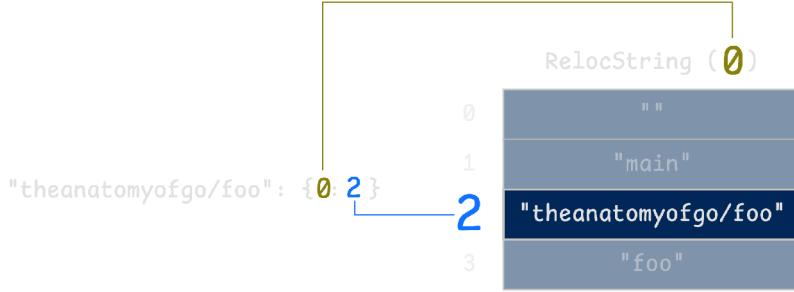


Illustration 97. RelocString type and index reference

When encoding the `main` package which imports `foo`, the export data for the `foo` package itself must first be generated. The encoder responsible for `foo` gathers necessary relocation information, including entries for its own path and name strings:

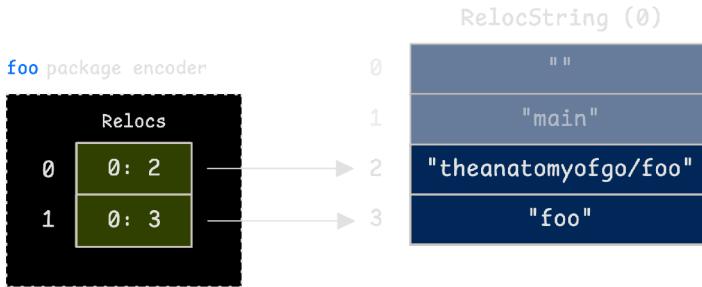


Illustration 98. foo encoder maps strings via relocations

In this scenario, the encoder for the `foo` package might collect two relocation to strings: one for the package path string (perhaps `"theanatomyofgo/foo"` at index `2`, represented as `{0: 2}`) and another for the package name string (`"foo"` at index `3`, represented as `{0: 3}`).

According to the package export data format we saw earlier, the encoded representation of the `foo` package would be the integers: `0 1 0`.

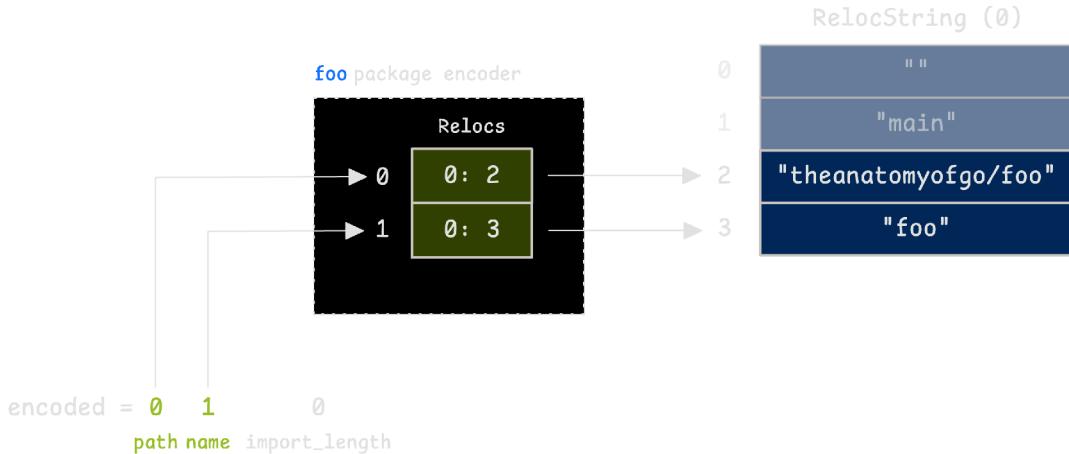


Illustration 99. Encoded header points to relocation indices

Because the `foo` package has no imported packages, its `imports_length` is zero. However, `0 1 0` by itself doesn't provide enough meaning for the decoder—unless we also include the relocation entries (`relocs`) generated by the encoder. That array provides the necessary context.

So, the full encoded form of the `foo` package element looks like this:

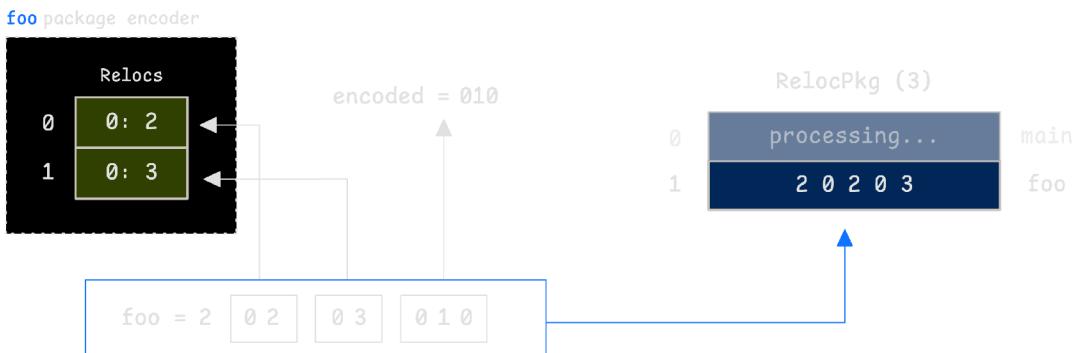


Illustration 100. Final encoded object data for 'foo' package element

The compiler then inserts this representation into the package section. This way, when another package element needs to reference the `foo` package element, it only requires two integers: `3 1` to locate the package object.

This bitstream encoding which combining relocation arrays and specific indices, is not limited to package object. It is used for many different relocation types, so understanding this format is important.

## Reverse operation: decoding the relocation data

If this process seems confusing, that is understandable. Let's look at how Go performs the reverse operation: decoding the relocation data.

First, Go examines the encoded sequence of `foo` package element:

```
2 0 2 0 3 0 1 0
```

It identifies that the relocation array has a length of `2`. This array holds two relocations to strings: `{0 2}` and `{0 3}`. To reconstruct the package details, Go refers to the package relocation format illustrated here:



Illustration 101. Compact layout for export package metadata

According to this format, the path index is `0`, corresponding to string relocation `{0 2}`. The package name index is `1`, corresponding to string relocation `{0 3}`. The number of imported packages is `0`. Finally, Go consults the string section to retrieve the actual string values associated with indices `2` and `3`.

The `main` package requires more relocations because it imports the `foo` package:

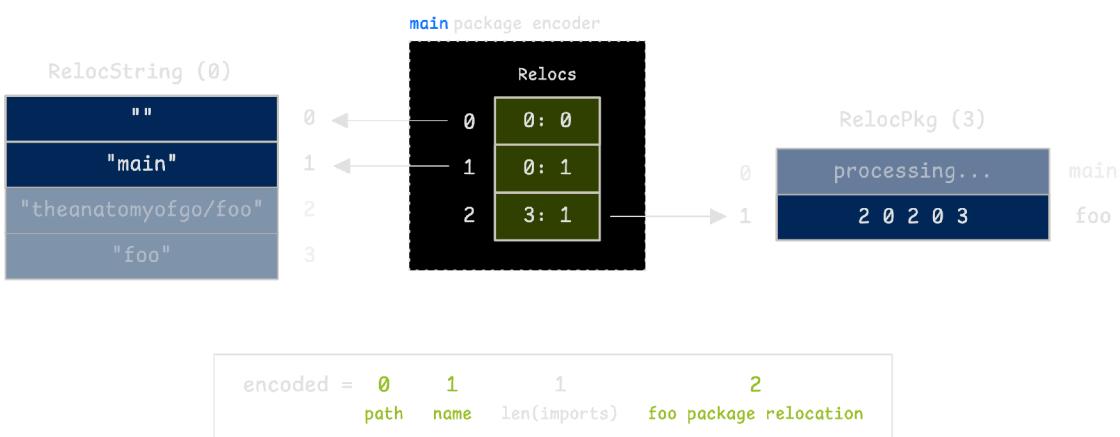


Illustration 102. main requires three relocation values

The encoding process is similar to what we saw with the `foo` package. It can definitely be a bit confusing the first time, so it's worth taking a moment to study

the diagram below and compare it with the earlier encoded version of `foo`.

Here is the final encoded object data for the `main` package:

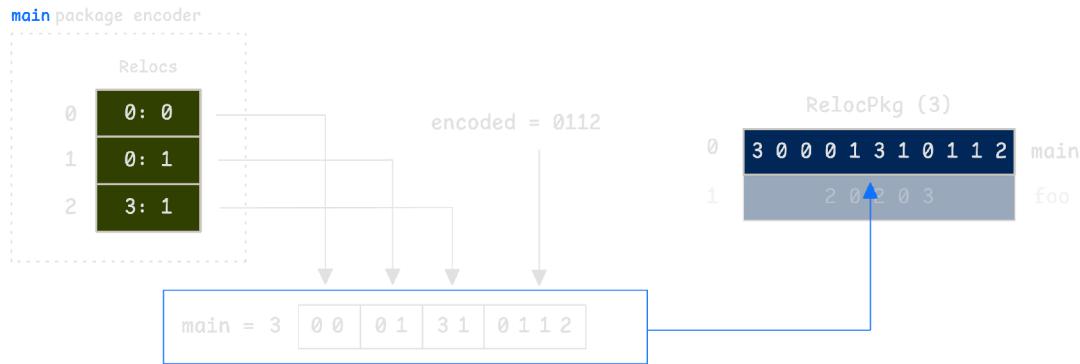


Illustration 103. Final encoded object data for 'main' package

Until now, we have now finished encoding the `main` package element and its imported `foo` package element. The next step involves encoding constant object relocations:

```
package main
import "theanatomyofgo/foo"
const a = foo.A + 1
```

To encode the constant `a`, the compiler first needs its value. We know the type checker calculates this value during its analysis. But calculating `a` requires the value of `foo.A`. How does the compiler find that?

When the type checking phase processes the `import` statements, it loads the export data of other packages into memory. However, this data remains in encoded form. When the type checker encounters `main.a`, it triggers a lazy decoding of the `foo.A` bitstream from the `foo` package export data. Since the encoding process for constants like `foo.A` and `main.a` is the same, let's examine how this encoding works.

## Encoding Constant Object

The bitstream encoding here works much like what we saw in the previous section on package objects. Since the overall structure is the same, we'll move quickly through this part. The goal isn't to analyze every small detail, but to give you a

clear sense of how constant objects are encoded. You don't need to follow every bit —just focus on the bigger picture.

When an object such as a function, variable, constant, or type is written to export data, its information is split across 4 relocation types:

- **Object relocation** (`RelocObj`): Holds core type information, such as the type, value, and source position.
- **Name relocation** (`RelocName`): Contains the kind of object (function, variable, constant, or type) and its fully qualified name (package path and object name).
- **Object extension relocation** (`RelocObjExt`): Stores additional compiler metadata that is not needed during type checking. This includes items like `linkname` directives, pragma flags, and function body relocation (`RelocBody`).
- **Object dictionary relocation** (`RelocObjDict`): Contains data related to generic type parameters and any instantiated types.

Let's begin by resolving the object relocation. Although constants are among the simplest objects to encode, understanding the structure can still be challenging the first time you see it:

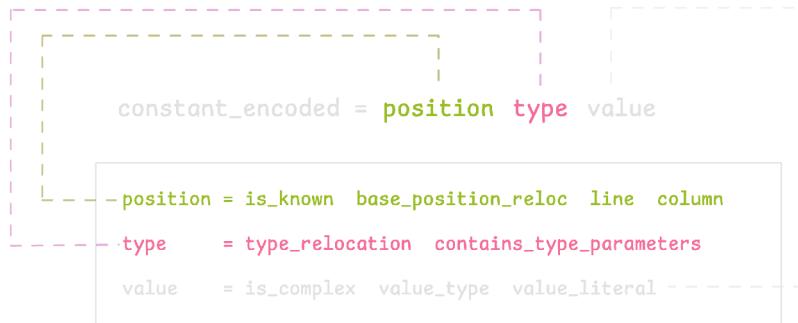


Illustration 104. Structure of a constant's object relocation

The structure above applies specifically to an `untyped int` constant.

Now, let's examine each part of the constant encoding structure. The `position` field has two parts:

- The base position relocation (`base_position_reloc`) identifies the file containing the constant.
- The location (`line` and `column`) specifies the exact line and column within that file.

The relocation base position uses a specific relocation type called **base position relocation** (`RelocPosBase`). In our example, the constant is defined in the `"anatomyofgo/main.go"` file. Therefore, the `RelocPosBase` relocation will point to a string relocation holding this file path:

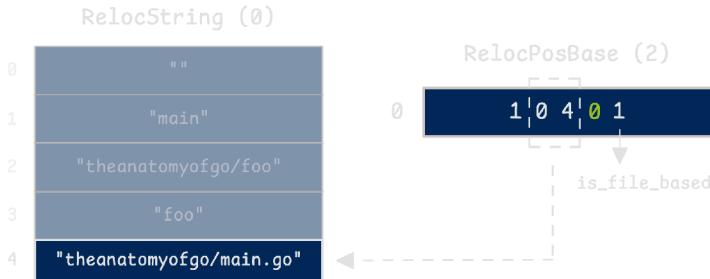


Illustration 105. File-based position encoded using 'RelocPosBase'

Here is how the `position` is encoded:

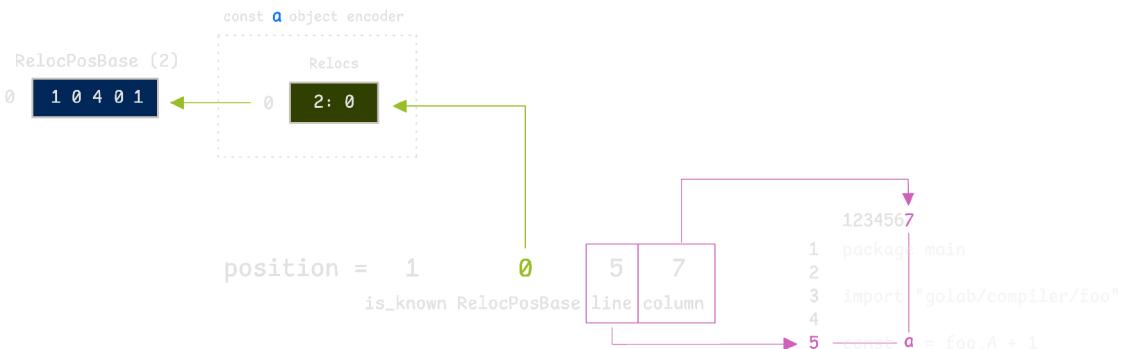


Illustration 106. Position encoded for constant a

Next, we encode the `type` part of the constant:

```
constant_encoded = position type value
```

```
DONE position = 1057
type      = type_relocation contains_type_parameters
value     = is_complex value_type value_literal
```

Illustration 107. Type encoded via relocation reference

This requires a type relocation for the `untyped int` type. The encoding method depends on the specific type being represented. We won't delve into the full details

of type encoding now, as the fundamental concept is clear. Instead, let's look at the resulting type relocation:

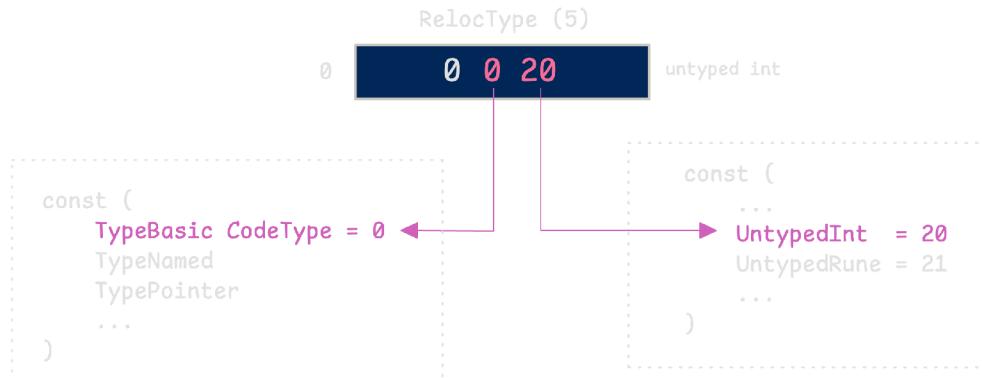


Illustration 108. Type relocation entry for 'untyped int'

After generating the type relocation, we encode the full type segment as: 1 0

- 1 is the reference to the `UntypedInt` type relocation.
- 0 indicates that the type does not contain type parameters (which is true for `untyped int`).

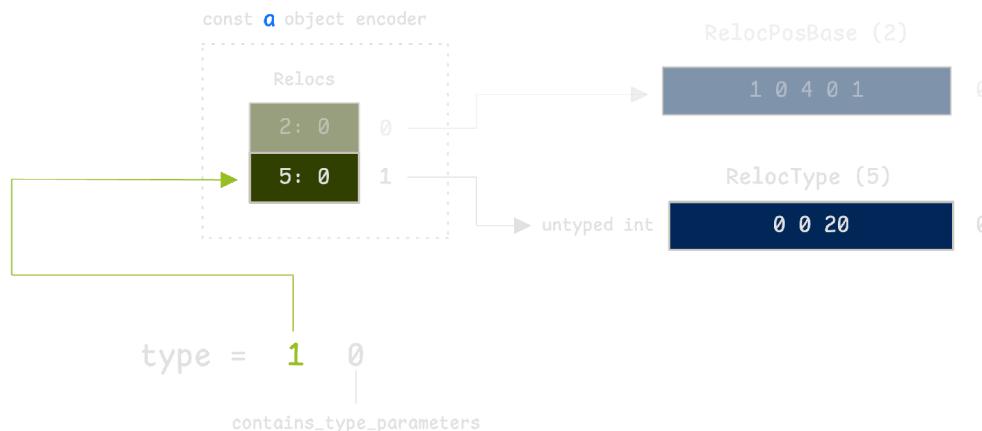


Illustration 109. Constant type encoding complete

Now let's handle the last part of encoding the constant object: the value.

```
constant_encoded = position type value

[DONE] position = 1057
[DONE] type      = 10
value      = is_complex value_type value_literal
```

### Illustration 110. Final step: encoding the constant value

In our case, the constant is a small `untyped int` with the value `2`. This is treated as an `int64` for encoding. If the value were larger, it would be treated as a `*big.Int` instead:

Figure 91. Encode scalar value (src/internal/pkgbits/encoder.go)

```
func (w *Encoder) scalar(val constant.Value) {
    switch v := constant.Val(val).(type) {
    ...
    case int64:
        w.Code(ValInt64)
        w.Int64(v)
    case *big.Int:
        w.Code(ValBigInt)
        w.bigInt(v)
    ...
}
```

Now, we have enough information to assemble the encoding for the constant value:

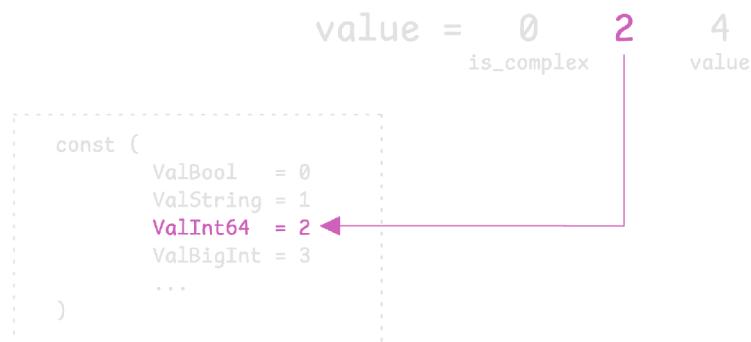


Illustration 111. Constant value encoded as int64 type

You might wonder why the constant value `2` ends up encoded as `4`.

That's because constant values can be negative, and Go uses a special encoding method to handle signed values. This method is called 'Zig-Zag variable-length encoding'. If you've worked with Protocol Buffers (`sint32`, `sint64`), this will sound familiar.

In Zig-Zag encoding:

- `-1` becomes `1`.

- 1 becomes 2.
- -2 becomes 3.
- 2 becomes 4.

This transformation allows negative and positive values to be encoded efficiently in a compact binary format.

It's time to assemble the full encoding for the constant object relocation:

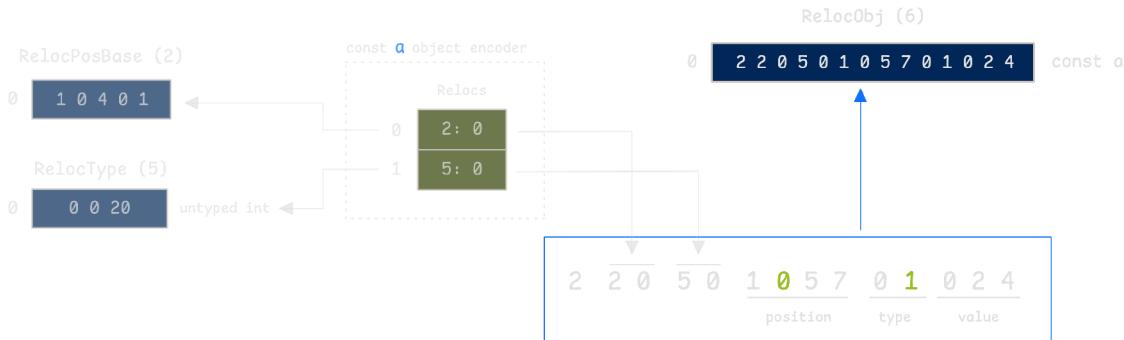


Illustration 112. Full constant object relocation assembled

This completes the discussion of the **object relocation**. While all exported objects can potentially have up to 4 associated relocations, constants typically only require two: the **object relocation** (which we just covered) and the **name relocation**.

Here is the result for the name relocation for constant `a`:

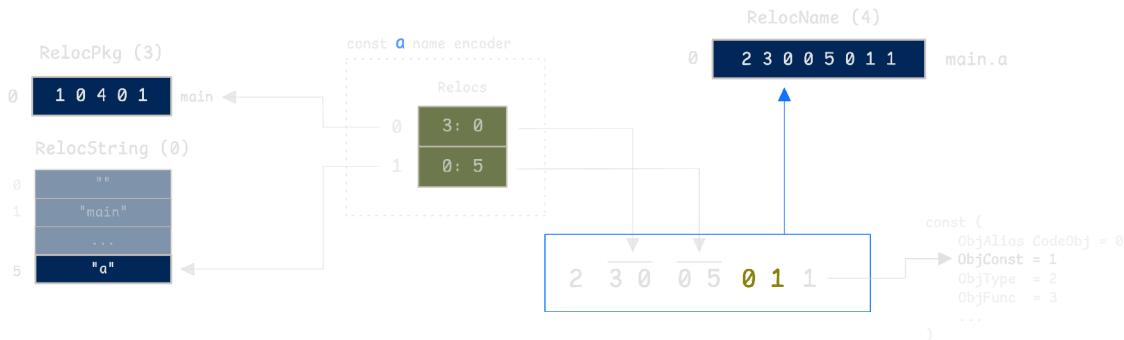


Illustration 113. Name relocation for constant a

So, encoding a constant object like `a` involves splitting its data across these four possible relocations:

- `RelocObj` : the main object data (position, type, value).
- `RelocName` : the name and kind of the object.

- `RelocObjExt` : additional compiler metadata (not needed for constants).
- `RelocObjDict` : generic-related data (not needed for constants).

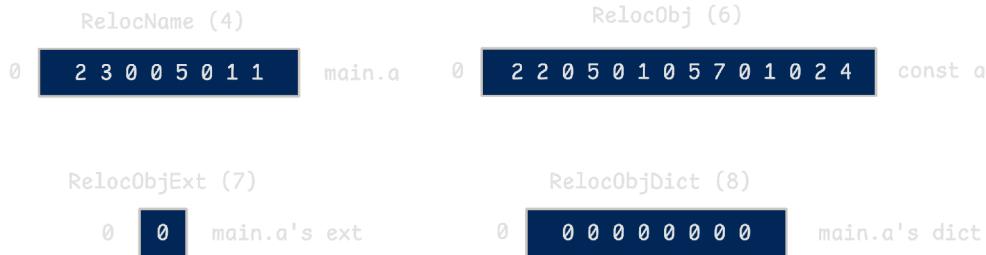


Illustration 114. All four relocation types for constant a

*Note: The encoded format may change across Go versions, but the core structure and purpose of these relocations remain consistent.*

To sum up: relocations in Go's export data format serve as references to shared elements stored in separate sections. This allows the compiler to:

- Cross-reference objects.
- Avoid duplication of common data like strings, types, and positions.
- Organize the export data logically and efficiently.

## Encoding Function Body Relocations

You may have noticed there is a relocation type called `RelocBody`. This means the compiler not only encodes a function's signature for use by other packages, but also its entire function body. But why would other packages need access to the function body of a function defined in another package?

The Go compiler exports function bodies mainly for two key reasons:

1. **Inlining across packages:** Inlining allows a function's body to replace the call site in the caller package. To do that, the compiler needs access to the function body, even if the function is defined in another package.
2. **Generic function instantiation:** Generic functions are instantiated in the binary of the package that uses them, not the package where they are defined. To generate a specialized version, the compiler must have access to the full function body, not just the signature.

When writing the function body, the IR writer processes it from the top down. It begins with high-level constructs, like statements—assignments, conditionals,

loops, returns, and so on. These statements are usually handled through calls like `w(stmt1(n))`:

Figure 92. Statement in IR Writer Pass  
(src/cmd/compile/internal/noder/writer.go)

```
func (w *writer) stmt1(stmt syntax.Stmt) {
    switch stmt := stmt.(type) {
    default:
        w.p.unexpected("statement", stmt)
        ...

    case *syntax.ForStmt:
        w.Code(stmtFor)
        w.forStmt(stmt)

    case *syntax.IfStmt:
        w.Code(stmtIf)
        w.ifStmt(stmt)
        ...

    case *syntax.ReturnStmt:
        w.Code(stmtReturn)
        w.pos(stmt)

        resultTypes := w.sig.Results()
        dstType := func(i int) types2.Type {
            return resultTypes.At(i).Type()
        }
        w.multiExpr(stmt, dstType,
syntax.UnpackListExpr(stmt.Results))
        ...
    }
}
```

When the writer processes a statement, it first identifies its type and writes a marker that represents the statement kind (`w.Code(stmtIf)`, `w.Code(stmtReturn)`, etc.). These markers are simple integer enum values that tell the reader how to interpret the encoded data that follows.

Each statement type has its own encoding format. That format depends on the structure and fields specific to the statement.

For example, an `if` statement includes several parts: an optional initialization statement, a condition expression, and blocks for both the 'then' and 'else' branches:

Figure 93. If Statement in IR Writer Pass  
(src/cmd/compile/internal/noder/writer.go)

```
func (w *writer) ifStmt(stmt *syntax.IfStmt) {
    cond := w.p.staticBool(&stmt.Cond)

    ...
    w.openScope(stmt.Pos())
    w.pos(stmt)
    w.stmt(stmt.Init)
    w.expr(stmt.Cond)
    w.Int(cond)
    if cond >= 0 {
        w.blockStmt(stmt.Then)
    } else {
        w.pos(stmt.Then.Rbrace)
    }
    if cond <= 0 {
        w.stmt(stmt.Else)
    }
    w.closeAnotherScope()
}
```

For an `if` statement, the writer opens a new scope by recording the position of the `if` block:

```
w.openScope(stmt.Pos())
```

Next, it writes the initialization statement using the `stmt1` method we reviewed earlier. Then it encodes the condition expression, the 'then' block, and the 'else' block:

```
func (w *writer) ifStmt(stmt *syntax.IfStmt) {
    ...
    w.stmt(stmt.Init)
    w.expr(stmt.Cond)
    ...
    if cond >= 0 {
        w.blockStmt(stmt.Then)
    }
    ...
    if cond <= 0 {
        w.stmt(stmt.Else)
    }
    ...
}
```

If this idea feels abstract, consider a common error-checking pattern:

```

if err := f(); err != nil {
    return err
}

```

This `if` statement will be broken down into expressions like assignment, function call, comparison, and return:

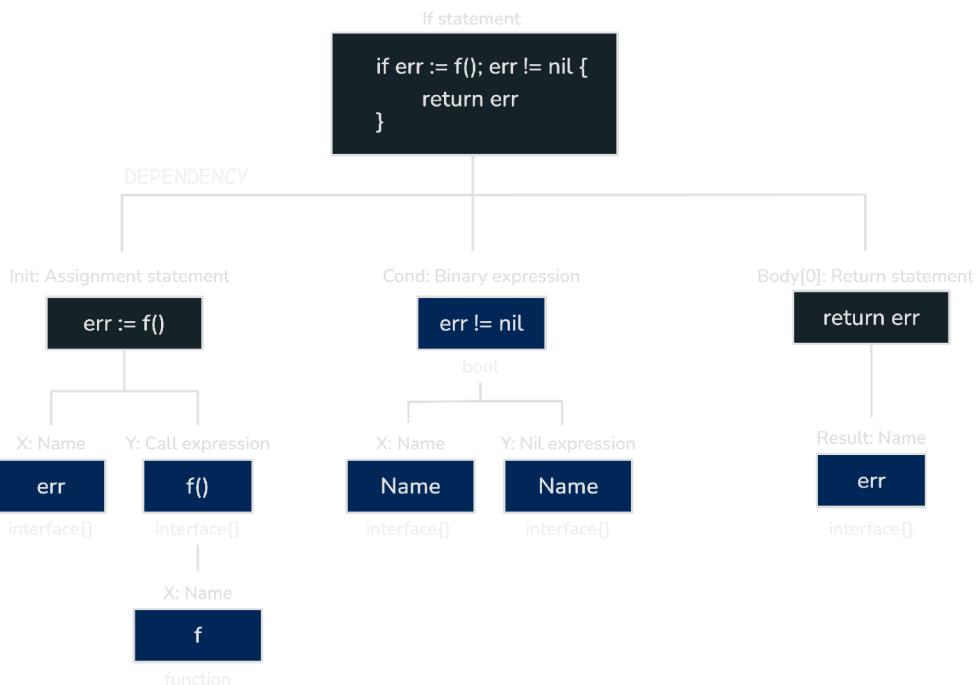


Illustration 115. Breakdown of scoped if error check

In the visual representation, statements and expressions are shown in different colors to help distinguish them. At this stage, the Go compiler has already completed type checking, so it knows the type of every expression. These types are also shown just beneath each expression node in the diagram.

The writer's role is to serialize this graph of statements and expressions while keeping both structure and type information intact. This allows the IR to be reconstructed accurately later on.

Just as with statements, every kind of expression has its own encoding format. Each one starts with a unique marker code that identifies the expression type. This includes slice expressions (`s[l:h:c]`), type assertions (`x.(T)`), call expressions (`f(a, b, c)`), etc.

When the writer encounters a global variable, there is not much work needed. The global object has already been encoded as part of the object relocation. As a result,

the writer simply writes the relocation index of that global object.

## Decoding Export Data into IR Nodes

After being encoded, the export data is later decoded into IR nodes. Let's use the same example as before:

```
func test() error {
    if err := f(); err != nil {
        return err
    }

    return nil
}

func f() error { return nil }
```

Here is how the `test()` function is translated into IR nodes, using the `ir.Dump()` function from the Go compiler source code:

```
1 *ir.Package {
2 . Funcs: []*ir.Func (2 entries) {
3 . . 0: *ir.Func {
4 . . . Body: ir.Nodes (2 entries) {
5 . . . . 0: *ir.IfStmt {
6 . . . . . Cond: *ir.BinaryExpr {
7 . . . . . . X: *ir.Name {
8 . . . . . . Class: PAUTO
9 . . . . . . Defn: *ir.AssignStmt {
10 . . . . . . . X: *(@7)
11 . . . . . . . Def: true
12 . . . . . . . Y: *ir.CallExpr {
13 . . . . . . . . Fun: *ir.Name {
...
40 . . . . . . . . }
41 . . . . . . . . ...
42 . . . . . . . . }
43 . . . . . . . . }
44 . . . . . . . . Curfn: *(@3)
45 . . . . . . . . ...
46 . . . . . . . . }
47 . . . . . . . . Y: *ir.NilExpr {}
48 . . . . . . . . ...
49 . . . . . . . . }
50 . . . . . Body: ir.Nodes (1 entries) {
51 . . . . . . 0: *ir.ReturnStmt {
52 . . . . . . . Results: ir.Nodes (1 entries) {
53 . . . . . . . . 0: *(@7)
54 . . . . . . . . }
55 . . . . . . . }
```

```
56 . . . . . }
57 . . . . . ...
58 . . . . .
59 . . . . 1: *ir.ReturnStmt {
60 . . . . . Results: ir.Nodes (1 entries) {
61 . . . . . . 0: *ir.NilExpr {}
62 . . . . .
63 . . . . }
64 . . . .
...
95 . . .
...
97 . .
...
100 }
```

IR nodes are defined in the `ir` package. At the top level, the compiler builds a node representing the entire package: `ir.Package`. The next level contains function nodes. In this example, we see two: one for `test()` and one for `f()`. The output above shows only the IR of `test()` (index `0`), so we can compare it directly with the visual IR graph we saw earlier.

Looking at the highlighted lines, you can spot familiar patterns. The `if` statement (`ir.IfStmt`) contains:

- An initialization statement (`ir.AssignStmt`).
- A condition expression (`ir.BinaryExpr`).
- A body block that includes one return statement (`ir.ReturnStmt`).

Even though the IR output is more verbose, it follows the same structure as the visualized IR graph. Let's revisit that diagram to compare:

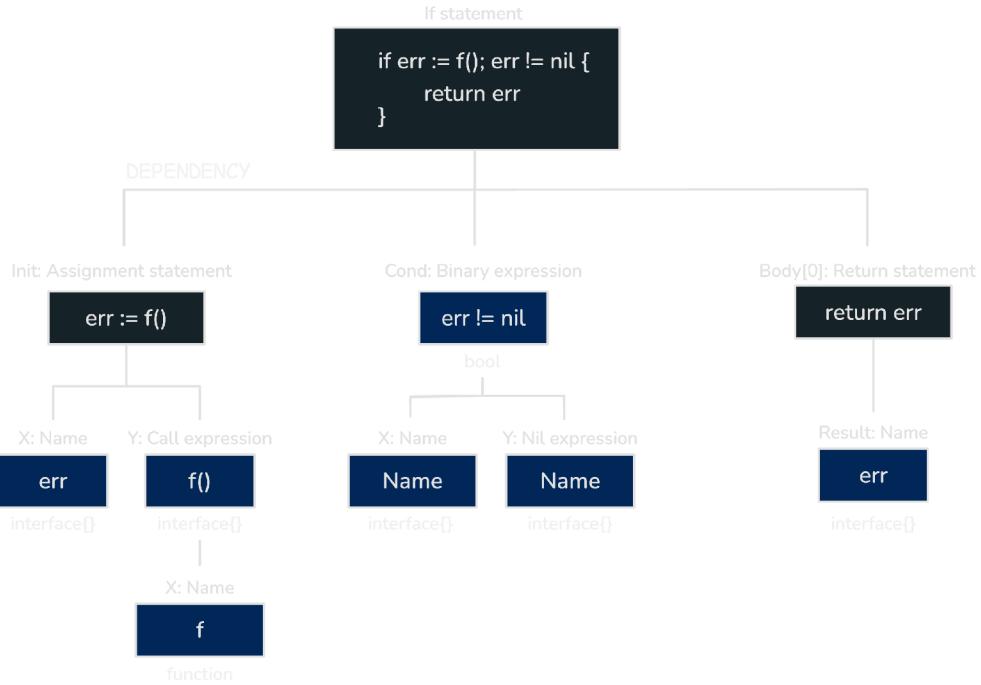


Illustration 116. Breakdown of scoped if error check

And that's how Go constructs the IR node structure from the export data. If it still feels a bit abstract, here's another example—a simple `for-range` loop to help clarify things:

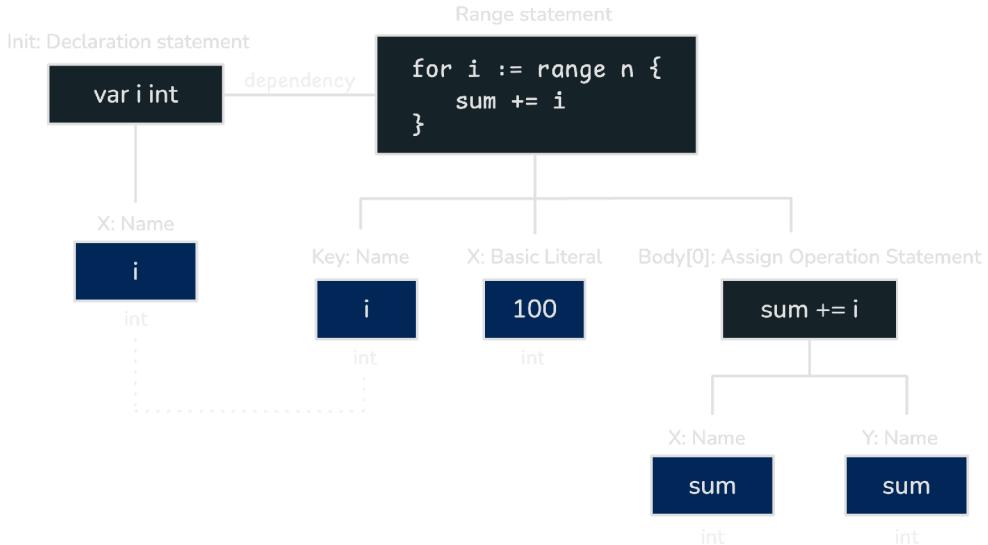


Illustration 117. Range loop broken into IR components

Keep in mind that what you're seeing here is the original IR representation. It is not yet in a form that can be encoded into export data or used by other packages.

Many optimizations and transformations will be applied to this IR graph soon. These steps can significantly change the structure. For example, after inline optimization, a call expression (`ir.CallExpr`) may be replaced with an inlined call expression (`ir.InlinedCallExpr`). This special node embeds the entire body of the called function directly into the IR. As a result, the IR tree becomes larger and more detailed.

These transformations are part of what prepares the program for later stages like code generation and further optimizations.

## 5.6 Stage 4: Optimization: Dead Code, Devirtualization, Inlining, Escape Analysis

Before we proceed to optimizations, it's helpful to revisit the compilation process to establish a clear context:

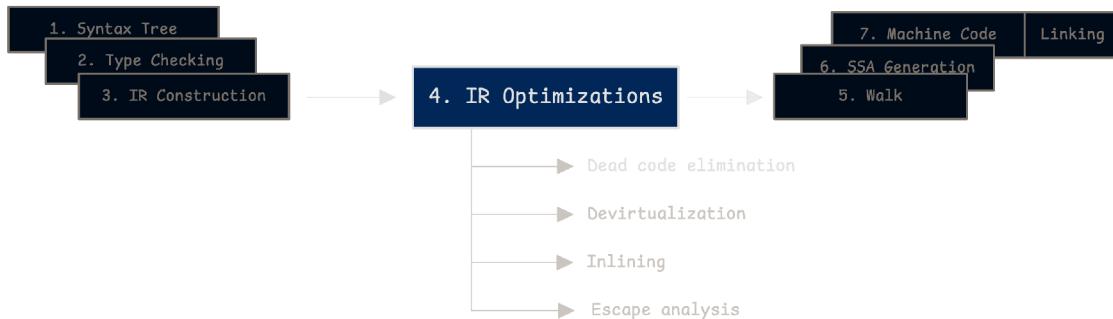


Illustration 118. Optimizations run after IR construction completes

### 5.6.1 Early Dead Code Elimination

At this stage, dead code elimination is relatively straightforward. It operates on the structure of the syntax tree and constant values resolved during type checking.

For example:

```
const alwaysTrue = true

if alwaysTrue {
    fmt.Println("Always runs")
} else {
    fmt.Println("Never runs")
}
```

Since `alwaysTrue` is a constant known at compile time, the `else` block is guaranteed never to run. The compiler can safely remove it early in the process.

Before Go 1.22, dead code elimination occurred after IR construction. Starting in Go 1.22, it has been integrated into the export data generation phase. We're covering it here because it still fits naturally within the broader optimization flow.

When the compiler encodes a function body and encounters an `if` statement, it uses a static check:

```
func (w *writer) ifStmt(stmt *syntax.IfStmt) {
    cond := w.p.staticBool(&stmt.Cond)

    ...
    w.openScope(stmt.Pos())
    w.pos(stmt)
    w.stmt(stmt.Init)
    w.expr(stmt.Cond)
    w.Int(cond)
    if cond >= 0 {
        w.blockStmt(stmt.Then)
    } else {
        w.pos(stmt.Then.Rbrace)
    }
    if cond <= 0 {
        w.stmt(stmt.Else)
    }
    w.closeAnotherScope()
}
```

The call to `w.p.staticBool(&stmt.Cond)` attempts to evaluate the condition at compile time:

- If the condition is always `true` (`cond >= 0`), the compiler skips writing the `else` (`stmt.Else`) block.
- If it's always `false` (`cond <= 0`), it skips the `then` (`stmt.Then`) block.
- If the result is unknown, both branches are included in the IR.

This early simplification helps reduce unnecessary code even before full IR generation.

Let's look at our previous example:

Before	After
<pre>const alwaysTrue = true  if alwaysTrue {     fmt.Println("Always runs") } else {     fmt.Println("Never runs") }</pre>	<pre>println("Else")</pre>

That covers the `if` statement. The handling of `for` loops offers another example of how this kind of dead code elimination works. Here is how the compiler encodes a `for` loop:

Figure 94. Dead `for` in IR Writer Pass  
(src/cmd/compile/internal/noder/writer.go)

```
func (w *writer) forStmt(stmt *syntax.ForStmt) {
    w.Sync(pkgbits.SyncForStmt)
    w.openScope(stmt.Pos())

    if rang, ok := stmt.Init.(*syntax.RangeClause);
    w.Bool(ok) {
        ...
        } else {
            if stmt.Cond != nil &&
w.p.staticBool(&stmt.Cond) < 0 { // always false
                stmt.Post = nil
                stmt.Body.List = nil
            }

            w.pos(stmt)
            w.stmt(stmt.Init)
            w.optExpr(stmt.Cond)
            w.stmt(stmt.Post)
        }
        ...
    }
```

There are two main paths here: one for range-based loops, and one for traditional `for` loops. Dead code elimination occurs in the second path, as highlighted above.

A `for` loop in Go consists of five parts: the initialization (`stmt.Init`), the condition (`stmt.Cond`), the post-statement (`stmt.Post`), the body (`stmt.Body`) and the label (`stmt.Label`). The compiler checks if the condition is always

`false`, just like in the `if` statement case. If so, it can safely remove the loop body and the post-statement, since they will never execute.

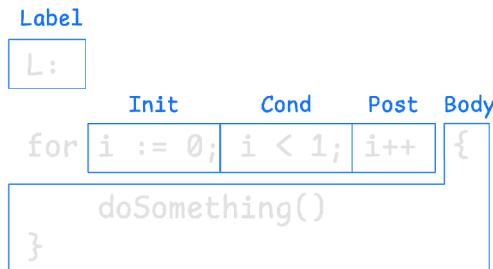


Illustration 119. Compiler view of a Go loop

In the corresponding reader function, the Go compiler decodes the `for` loop from the export data into IR nodes:

Figure 95. Reader Handling of Dead `for` Loop  
(src/cmd/compile/internal/noder/reader.go)

```
func (r *reader) forStmt(label *types.Sym) ir.Node {
    ...
    pos := r.pos()
    init := r.stmt()
    cond := r.optExpr()
    post := r.stmt()
    body := r.blockStmt()
    perLoopVars := r.Bool()
    r.closeAnotherScope()

    if ir.IsConst(cond, constant.Bool) &&
!ir.BoolVal(cond) {
        return init // simplify "for init; false;
post { ... }" into just "init"
    }

    stmt := ir.NewForStmt(pos, init, cond, post, body,
perLoopVars)
    stmt.Label = label
    return stmt
}
```

Notice that the initialization part is preserved. That's because it might include side effects. Even if the loop body and post-statement are skipped, the `init` must still run. Let's look at an example of how a `for` statement is transformed:

Before	After
<pre>for i := 0; false; i++ {     fmt.Println("This will never execute") }</pre>	<pre>i := 0</pre>

However, the initialization part of the `for` loop may not just be a simple assignment like `i := 0`. It could involve function calls or other operations that have side effects:

```
for i := sideEffect(); true; i++ {}
```

Since the compiler cannot always determine whether a side effect exists, it must keep the `init` part just in case.

While later passes handle more complex cases, this early dead code elimination focuses on constant-based structural simplifications. Here are additional examples of code it can simplify:

Before	After
<pre>const value = 2 switch value { case 1:     fmt.Println("One") case 2:     fmt.Println("Two") default:     fmt.Println("Other") }</pre>	<pre>fmt.Println("Two")</pre>
<pre>{     return 42     fmt.Println("This will never execute") }</pre>	<pre>return 42</pre>

## 5.6.2 Devirtualization

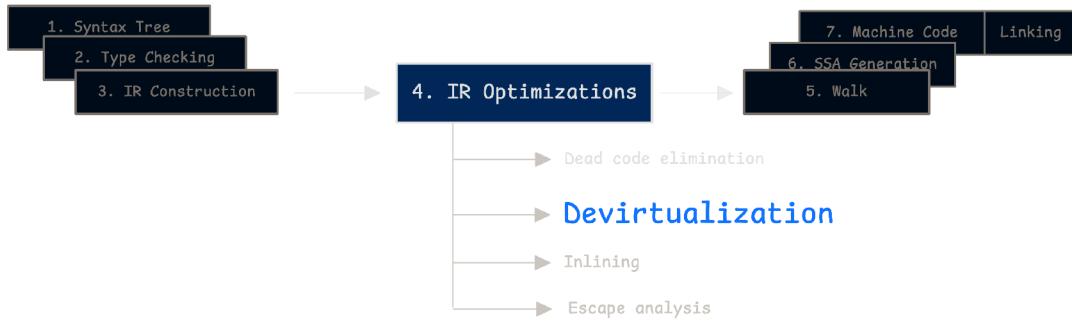


Illustration 120. Devirtualization Occurs After IR Construction

When you call a method using an interface variable, the specific method to execute is determined only when the program runs:

```

type Animal interface{
    Speak()
}

func DoSomething(a Animal) {
    a.Speak() // The actual Speak method depends on 'a's
    concrete type at runtime
}

```

Interface calls like this introduce a small performance cost because the correct method must be looked up during execution. Devirtualization is a compiler optimization designed to eliminate this cost. If the compiler can determine the concrete type of the interface variable at compile time, it replaces the interface call with a direct call to that type's method.

Consider this example:

```

type Dog struct {}

func (d Dog) Speak() {
    println("Gauge!")
}

func main() {
    var d Animal = Dog{} // 'd' has type Animal but holds
    a Dog value

    d.Speak() // Call Speak via the interface
}

```

Here, the `d.Speak()` call appears to use the `Animal` interface, suggesting a runtime lookup. However, because `d` is immediately assigned the concrete type `Dog{}`, the compiler knows the exact type. It sees that `d` is created through a conversion from a `Dog` value to an `Animal` interface. This allows the compiler to perform devirtualization.

In cases like this, the Go compiler can safely replace the indirect method call with a direct call to the method on the concrete type:

```
func main() {
    var d Animal = Dog{}

    d.(Dog).Speak()
}
```

To check whether this optimization is applied, you can run the compiler with static analysis enabled:

```
go build -gcflags="-m"
```

If devirtualization is successful, you should see a message like:

```
./main.go:16:9: devirtualizing d.Speak to Dog
```

This means the compiler has optimized away the interface dispatch, calling `Dog.Speak` directly. However, it is quite easy to break the devirtualization logic by introducing a reassignment that changes the type stored in the interface variable:

```
func main() {
    var d Animal = Dog{}
    d = Cat{}

    d.Speak()
}
```

Even though it is clear to us, that `d` ends up holding a `Cat` value, the compiler does not optimize this case. While it still recognizes the interface conversion, it avoids devirtualizing the method call.

To support these kinds of cases, the compiler would need to track more complex control flow and account for many different possibilities. Reassignments can appear in different forms, and the compiler must make conservative decisions in such scenarios. For example:

- Variables whose addresses are taken are assumed to be potentially modified elsewhere.
- Global variables are always treated as if they could be reassigned at any time.

This logic is part of the compiler's reassignment detection, as shown here:

Figure 96. Reassignment Detection (src/cmd/compile/internal/ir/expr.go)

```
func Reassigned(name *Name) bool {
    ...
    // global objects
    if name.Curfn == nil {
        return true
    }
    // address-taken variables
    if name.Addrtaken() {
        return true
    }
    ...
}
```

That said, what's especially interesting isn't just static devirtualization. The real strength lies in **dynamic devirtualization**, where the compiler uses runtime profiling data to optimize indirect calls. This approach allows optimization decisions to be guided by actual program behavior. We'll cover it in more detail in Chapter 6, in the section on Profile-Guided Optimization (PGO).

### 5.6.3 Inlining

It is good practice to split code into smaller, manageable functions. This makes code easier to read using descriptive names, simpler to change and maintain, and easier to reuse across the codebase.

However, function calls come with a small performance cost. Every time a function is called, the program sets up a new stack frame and then tears it down when the function returns.

In most cases, this overhead is negligible. The benefits of using functions usually far outweigh the cost. But in performance-critical parts of your code, these small costs can add up. Wouldn't it be great if you could keep the benefits of using functions while avoiding the cost of calling them? This is where inlining optimization comes in.

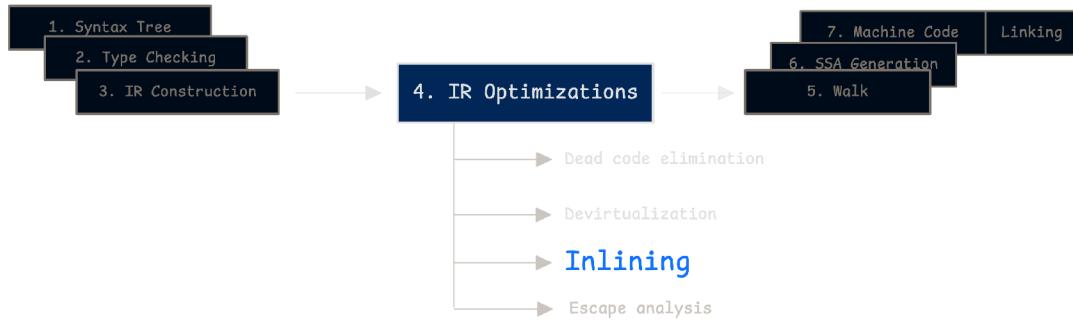
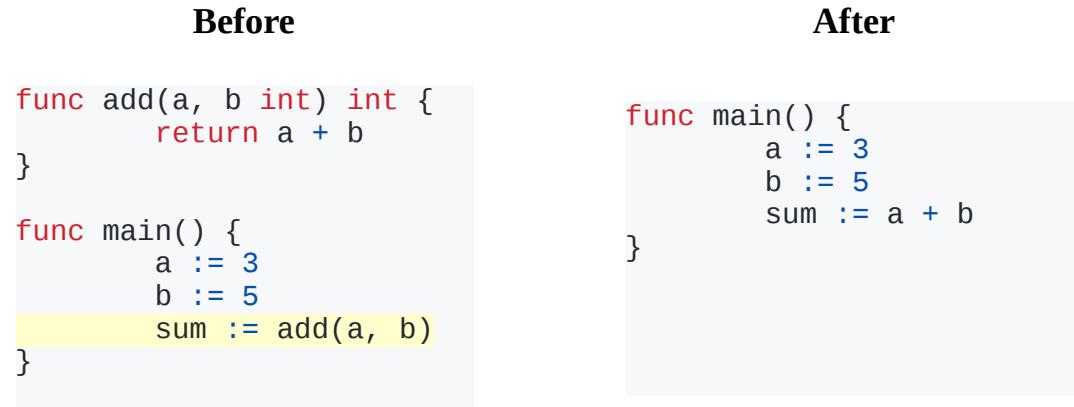


Illustration 121. Compiler pipeline: inlining within IR optimizations

Inlining is a compiler optimization that replaces a function call with the actual body of the called function:



In this example, the compiler recognizes that the `add` function is simple and can be replaced directly at the call site. By doing this, it removes the function call overhead.

Inlining not only improves performance by removing the call, but it also opens up more room for further optimization. Once the function body is inlined, the compiler sees more of the surrounding context and can apply additional optimizations.

In the case above, since both `a` and `b` are values the Go compiler can evaluate at compile time, it can optimize away the addition operation and even eliminate the variables `a` and `b`:

```

func main() {
    sum := 8
}

```

The variables `a` and `b` are not declared as constants, but they hold constant values from the compiler's point of view.

In a later stage of the compilation process, particularly the Single Static Assignment (SSA) stage, the compiler can analyze the program and determine that the values of `a` and `b` do not change after being assigned. Because of this, the compiler is able to treat them as constants for optimization purposes.

Behind the scenes, inlining changes the structure of the Intermediate Representation (IR). Instead of a call node, the compiler inserts the function's implementation directly into the calling function's IR.

Now, let's run a quick benchmark to see if inlining actually makes a difference:

```
//go:noinline
func add(a, b int) int {
    return a + b
}

func addInline(a, b int) int {
    return a + b
}

func BenchmarkAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        add(1, 2)
    }
}

func BenchmarkAddInline(b *testing.B) {
    for i := 0; i < b.N; i++ {
        addInline(1, 2)
    }
}
```

We use the special compiler directive `//go:noinline` to tell the compiler not to inline a specific function. This allows us to control the test more precisely. Let's run the benchmarks and compare the results:

BenchmarkAdd-14	10000000000	0.7240 ns/op
BenchmarkAddInline-14	10000000000	0.2296 ns/op

The results will vary slightly on different machines and runs, but the inlined version is consistently 2 to 3 times faster. This shows how even a small function call can have measurable cost. Keep in mind that while you can prevent inlining, you cannot force the compiler to inline a function. It makes that decision on its own.

## Compiler Directives: `//go:inline`

There was a proposal to introduce a `//go:inline` directive in Go [gin], but it was firmly rejected by the Go team. In general, compiler directives are discouraged in Go's design philosophy because they introduce complexity and shift focus away from clarity and readability.

The emphasis in Go is on improving the compiler's ability to make smart decisions automatically, rather than requiring developers to manage low-level optimizations themselves.

The key factor the compiler looks at is size. Inlining only applies to small functions. The Go compiler estimates the complexity of a function based on factors like the number of statements, basic blocks, and control flow constructs. If the function is simple enough, it may be inlined. Let's try a small experiment:

```
func A() {
    println("A")
    B()
}

func B() {
    defer println("C")
    println("B")
}

func main() {
    A()
}
```

To see which functions the compiler chooses to inline, use this command:

```
go build -gcflags="-m"
```

Sample output might look like this:

```
./main.go:9:2: can inline B.deferwrap1
./main.go:3:6: can inline A
./main.go:13:6: can inline main
./main.go:14:3: inlining call to A
```

Here, the compiler indicates that `A`, `main`, and `B.deferwrap1` can be inlined. However, note that `B` itself is not inlined.

## What is `deferwrap1`?

Deferred calls are wrapped by the compiler into small helper functions that take no arguments and return no values. These wrappers, such as `B.deferwrap1`, are generated automatically and can be inlined if they are simple enough:

```
func B() {
    defer func() { // deferwrap1
        println("C")
    }()
    println("B")
}
```

We'll cover `defer` statements in more detail in the next chapter.

Even though `B` looks simple, the presence of a `defer` statement inside it prevents it from being inlined. At the moment, the Go compiler does not support inlining functions that contain `defer`, regardless of their size, because it requires complex logic to handle stack frames and panic recovery, as discussed in Chapter 6.

That said, if you have a small function that can't panic, try to avoid using `defer` and instead place the deferred code at the end of the function.

## Determine Inlinable Functions

There's a straightforward way to check how much a function "costs," how the compiler makes inlining decisions, and why a function might not be eligible for inlining. Just pass additional `-m` flags to your build command:

```
go build -gcflags="-m=2"
# or
go build -gcflags="-m -m"
```

This gives you detailed output about inlining decisions:

```
./main.go:8:6: cannot inline B: unhandled op DEFER
./main.go:9:2: can inline B.deferwrap1 with cost 2 as: func()
{ println("C") }
./main.go:3:6: can inline A with cost 61 as: func() {
println("A"); B() }
./main.go:13:6: can inline main with cost 63 as: func() { A()
}
./main.go:14:3: inlining call to A
```

The output not only indicates whether a function can be inlined, but also shows the compiler's calculated cost and a simplified version of its internal representation

after parsing and analysis. This gives us insight into how the compiler "sees" the function.

Let's take a closer look at function A. The cost is based on the number of IR (Intermediate Representation) nodes, with some nodes having extra weight depending on what they do:

```
func A() {
    println("A")
    B()
}
```

And here's a snippet of its IR node representation:

```
.  DCLFUNC main.A ABI:ABIInternal FUNC-func() tc(1) #
main.go:3:6
.  DCLFUNC-body
.  .  PRINTN tc(1) # main.go:4:9
.  .  PRINTN-Args
.  .  .  LITERAL-"A" string tc(1) # main.go:4:10
.  .  .  CALLFUNC tc(1) # main.go:5:3
.  .  .  CALLFUNC-Fun
.  .  .  NAME-main.B Class:PFUNC Offset:0 Used FUNC-func()
tc(1) # main.go:8:6
```

The cost breakdown is as follows:

- `DCLFUNC main.A`: Declares the function. This is not included in the inlining budget.
- `PRINTN`: A print statement. Cost: 1.
- `LITERAL-"A"`: A string literal "A". Cost: 1.
- `CALLFUNC`: A function call. Base cost: 1.
- `NAME-main.B`: Refers to function B. Cost: 1.

Unfortunately, any function call (`CALLFUNC`) adds an additional cost of 57 units, defined by the compiler as `inlineExtraCallCost`. This pushes the total cost of function A up to 61.

You can adjust the inlining behavior for function calls cost by using the `-l=4` flag. This reduces the penalty for having multiple function calls within a single function:

- With default settings, a function call costs 58 units: 1 for the base cost and 57 as the extra call cost.

- With `-l=4`, the same call only costs 2 units: 1 for the base cost and 1 for the extra call cost.

This means that with `-l=4`, functions containing two or more calls are still likely to be inlined, as long as they stay within the total budget.

Now let's consider the `main` function. It appears simple but still has a higher inlining cost: 63 units. Here is its IR representation:

```
.  DCLFUNC main.main ABI:ABIInternal FUNC-func() tc(1) #
main.go:13:6
.  DCLFUNC-body
.  .  CALLFUNC tc(1) # main.go:14:3
.  .  CALLFUNC-Fun
.  .  .  NAME-main.A Class:PFUNC Offset:0 Used FUNC-func()
tc(1) # main.go:3:6
```

This shows 2 IR nodes: `CALLFUNC` and `NAME`. Since `A` is inlined, there is no additional call penalty. However, the cost of `main` will include the cost of `A`, plus the cost of its own IR nodes.

Behind the scenes, the Go compiler gives each function a fixed inlining budget. By default, this budget is 80 units. The compiler processes functions from the innermost ones, like `B`, and moves outward to higher-level ones like `main`.

The function that handles budget calculation is relatively straightforward but considers many conditions. Here is a simplified view of how this logic works:

```
func (v *hairyVisitor) doNode(n ir.Node) bool {
    if n == nil {
        return false
    }
opSwitch:
    switch n.Op() {
    case ir.OCALLFUNC:
        ...

        // Intrinsic calls are:
        if ir.IsIntrinsicCall(n) {
            break
        }

        v.budget -= v.extraCallCost

    case ir.OCALL, ir.OCALLINTER:
        v.budget -= v.extraCallCost
```

```

        case ir.ORECOVER:
            base.FatalfAt(n.Pos(), "ORECOVER missed
typecheck")
        case ir.ORECOVERFP:
            v.reason = "call to recover"
            return true

        case ir.OCLOSURE:
            v.budget -= 15

        case ir.OGO, ir.ODEFER, ir.OTAILCALL:
            v.reason = "unhandled op " + n.Op().String()
            return true

        ...
    }

    v.budget--

    return ir.DoChildren(n, v.do)
}

```

There are more cases handled in the full implementation, but this snippet highlights the key ideas. For example:

- A closure reduces the budget by 15 units.
- Any use of `defer`, `go`, `recover`, or `tailcall` will disqualify the function from inlining entirely.

This logic gives the compiler flexibility to avoid inlining complex or risky functions, while still allowing aggressive optimization of simple, frequently used ones.

There is a special case for intrinsic calls, detected using `ir.IsIntrinsicCall(n)`. These are function calls that the compiler recognizes and replaces with optimized, often hardware-specific, inline code rather than generating a regular function call:

Go	ARM64 Assembly Instruction
<code>math.Sqrt(n)</code>	FSQRTD: Floating-point square root, double precision
<code>math.Abs(n)</code>	FABSD: Floating-point absolute value

## Go

## ARM64 Assembly Instruction

```
atomic.StoreInt64(&p,  
42)
```

STLR: Store-Release

```
atomic.StoreInt64(&p,  
42)
```

LDAR: Load-Acquire

These are not functions in the usual sense. Instead, they act as compiler-recognized operations for performance-critical tasks and are assigned an inlining cost of 1 unit.

Now, let's review some situations where the Go compiler prevents inlining. These are based on the compiler behavior at the time this book was written:

- Functions that call `recover()` cannot be inlined. This is because `recover()` needs access to the caller's frame pointer to locate the appropriate panic value. If the function were inlined, the stack frame would be merged and the pointer lost.
- Functions that use reflection internally like `runtime.Caller()` cannot be inlined, as these methods inspect the call stack and rely on a standard frame structure.
- Method calls through interfaces are not inlined, because the actual method is chosen at runtime and cannot be resolved at compile time.
- Function calls made through function values (e.g. variables holding a function) cannot be inlined since the compiler cannot determine the exact function being called.
- Functions that are invoked using `go` or `defer` cannot be inlined. These operations involve extra handling that prevents safe inlining.
- Recursive functions (functions that call themselves) are never inlined to avoid infinite inlining loops and excessive growth.
- A "big function" is one that contains more than 5000 IR nodes. These are treated specially and given a very small inlining budget to discourage inlining.

Inlining decisions may vary from one version of Go to another, as the compiler continues to evolve. If you're curious about which of your functions can be inlined, the most reliable way to check is by using:

```
go build -gcflags="-m"  
# or  
go build -gcflags="-m=2"
```

As explained in the previous section, this will show inlining decisions, cost estimates, and reasons why specific functions are or aren't inlined. It is a really useful tool for performance tuning and understanding how the compiler views your code.

## Inlining at the Call Site

Now that we understand how the compiler calculates the cost of a function, the next question is: how does the compiler transform a regular function call into an inlined version?

Let's walk through a simple experiment:

main package	foo package
<pre>func main() {     result = foo.Add(1, 2) }</pre>	<pre>func Add(a, b int) int {     return a + b }</pre>

Before inlining happens, the assignment statement is represented in the compiler's IR as a normal function call node:

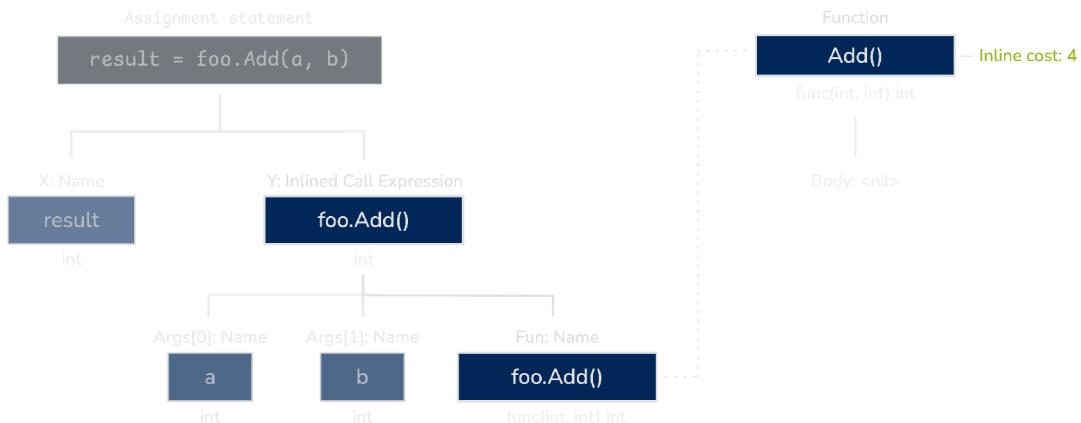


Illustration 122. Inline cost retrieved from export data

Since `foo.Add` comes from another package, its metadata is already available when we import the `foo` package. This metadata includes the function's signature, type information, and most importantly, its inlining cost. However, the body of `foo.Add` is not decoded yet. Instead, the export data includes a reference to the location of its body within the function body relocation table.

The inlining process happens in two major steps:

## 1. Marking Functions as Inlineable

The compiler walks through all functions in the current package and evaluates whether each function is inlineable. It checks the cost, applies any relevant pragmas, and verifies inlining restrictions (such as recursion or `go:noinline`).

If the function qualifies for inlining, the compiler records this information in the function node (`ir.Func`). At this point, only the metadata is marked—no transformation has occurred yet.

## 2. Rewriting Call Sites

In the second phase, the compiler scans all function call sites. If a call is to a function marked as inlineable, it rewrites the call from a regular `ir.CallExpr` node into an `ir.InlinedCallExpr` node. This new node embeds the full inlined body of the target function:

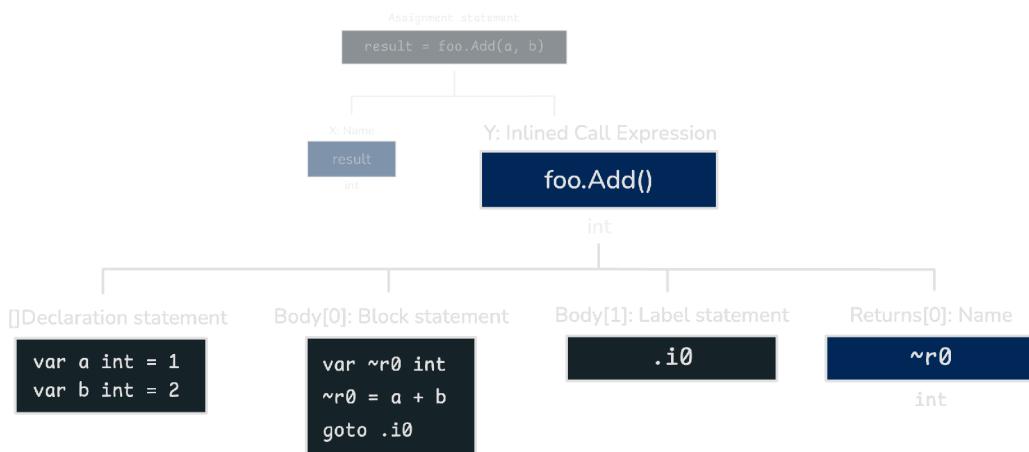


Illustration 123. Inlined body replaces original function call

This might sound complex, but the actual transformation is straightforward. Here's a simplified example, shown in pseudocode. Note that `~r0` is a placeholder name the compiler uses internally for return values:

### Before

```
func main() {
    result =
    foo.Add(1, 2)
}
```

### After

```
func main() {
    a, b := 1, 2
    var ~r0 int
```

Before	After
	<pre> ~r0 = a + b goto .i0  .i0: result := ~r0 } </pre>

During inlining, the compiler does the following:

- It creates temporary local variables for each parameter of the inlined function.
- It assigns the call site arguments to those local variables.
- Each return statement is replaced by:
- Storing the result into a temporary variable (`~r0 = a + b`).
- A `goto` to a common label at the end of the inlined code (`goto .i0`).
- A label placed after the inlined block to resume execution (`.i0:`)

By rewriting the code this way, the compiler preserves the structure and control flow of the original code, while avoiding the actual function call.

## Fast Path Optimization

Let's look at a simple example that becomes surprisingly powerful when we use inlining:

```

var s sync.Mutex

func A() {
    s.Lock()
    println("A")
}

func main() {
    A()
    s.Unlock()
}

```

The `sync.Mutex`'s `Lock` function is inlined into `A`. Although we don't see the cost of `Lock` directly in the code, the compiler logs reveal a clue. (As a bit of insider info, the cost of `Lock` is 66, and `Unlock` is 72.)

What makes this interesting is the actual implementation of the `Lock` function:

```

func (m *Mutex) Lock() {
    // Fast path: grab unlocked mutex.
    if atomic.CompareAndSwapInt32(&m.state, 0,
mutexLocked) {
        if race.Enabled {
            race.Acquire(unsafe.Pointer(m))
        }
        return
    }
    // Slow path (outlined so that the fast path can be
inlined)
    m.lockSlow()
}

```

The `Lock` function begins with a fast path. It performs a simple atomic operation to attempt grabbing the mutex. If the lock is available, the function returns immediately. If not, the call falls through to the slow path, which involves more work and is handled in a separate function called `lockSlow`.

But doesn't `Lock()` contain three function calls?

- `atomic.CompareAndSwapInt32()`
- `race.Acquire()`
- `m.lockSlow()`

Yes, it does. However, only **one** of them is counted as a real function call during cost calculation. Here's why:

- `atomic.CompareAndSwapInt32` is an intrinsic. It is recognized by the compiler and substituted with low-level inline code. Its cost is just 1 unit.
- `race.Enabled` is a compile-time constant. When the race detector is disabled, this condition is always `false`, making the `race.Acquire` call unreachable. As a result, it becomes dead code and is eliminated during compilation.
- `m.lockSlow()` is an only function call, but it only appears on the slow path and is placed after an early return.

This is where the design becomes brilliant. By structuring the `Lock()` function this way, the fast path becomes small and simple enough to be inlined. The slow path remains separate and is only used when necessary.

Here's how the `A` function looks after inlining the fast path of `Lock()`:

```

func A() {
    if atomic.CompareAndSwapInt32(&s.state, 0, mutexLocked) {
    } else {

```

```
        s.lockSlow()
    }
    println("A")
}
```

Now, the most common case, acquiring an uncontended lock, is fully inlined. This means the hot path of execution avoids function call overhead, leading to better performance.

To summarize:

- Each function has a calculated cost (sometimes referred to as "hairiness") based on its IR complexity.
- The compiler uses a fixed inlining budget (`80 units`) to decide if a function is eligible for inlining.
- If the function's cost is within the budget, it will be inlined.
- Carefully designed fast paths, such as in `Lock()`, allow Go to inline only the common case, improving performance while keeping complex logic in the slow path.

## Call Site Scoring

One of the main limitations of the traditional syntax-based inlining approach is that it lacks context awareness.

In other words, the compiler cannot decide to inline a function based on how or where it is called. For example, you might want function `A` to be inlined when used inside a `for` loop, to get better performance, but not when it appears in an `if` statement or regular control flow. The current model treats all call sites the same, regardless of context.

This limitation led to the development of an enhanced inlining system, introduced experimentally in Go 1.21. The new inliner still follows the budget-based model but adds more intelligence. It is designed to improve inlining decisions by considering context. Over time, this system is expected to become the default inliner because it produces better results in many cases.

As of Go 1.23, this new inliner is still experimental and not enabled by default. You can turn it on in one of two ways:

- Set the environment variable: `GOEXPERIMENT=newinliner`.
- Build with the Go tag: `goexperiment.newinliner`.

The inlining process consists of two distinct phases: function scoring and call site scoring. The new inliner focuses particularly on enhancing the call site scoring phase:



Illustration 124. Inlining cost split by function and call site

Let's recall the function scoring phase:

1. The compiler walks through each function and evaluates whether it is eligible for inlining.
2. It checks for disqualifying features such as: use of `recover()`, launching new goroutines, presence of `defer` statements, etc.
3. The compiler calculates the cost of the function, which reflects how complex it is. This value is called the **base cost**. In the traditional inliner, the budget is set to 80 units. However, with the new inliner enabled, this budget is doubled to 160 units.

Previously, the Go compiler would simply replace a call site with the function's body if the function's base cost was below the budget. The new approach, however, also calculates an adjusted cost for the call site itself, considering both the function's base cost and the context surrounding the call. Some situations reduce the cost (making inlining more likely), while others increase it (making inlining less likely).

The budget for each call site is set to 80 units. If the adjusted cost of the call is more than 80, the function will not be inlined at that location. So now, we are working with two separate budgets:

- **Function budget:** This is based on the syntactic complexity of the function. With the new inliner, this budget is increased from 80 to 160 units.
- **Call site budget:** Each call site is evaluated with a budget of 80 units.

The reason for increasing the function budget is to allow more functions to stay eligible for inlining, at least initially. This gives the second phase—the call site scorer—a chance to apply more context-aware decisions.

For example, a function with a cost of 120 would normally be rejected under the traditional inliner. But with the new inliner, it is still considered. When its call sites are evaluated, certain adjustments may lower the effective cost below 80, allowing the function to be inlined in that context.

There are three main types of adjustments applied during call site scoring:

- **Call site context adjustments:** Adjust the score based on where and how the function is called.
- **Parameter usage adjustments:** Reflect how the function uses its input parameters.
- **Return value usage adjustments:** Based on how the caller handles the function's return values.

Here is the adjustment table used during call site scoring:

Figure 97. Adjustment table for call site scoring  
(src/cmd/compile/internal/inline/inlheur/scoring.go)

```
var adjValues = map[scoreAdjustTyp]int{
    panicPathAdj:          40,
    initFuncAdj:           20,
    inLoopAdj:             -5,
    passConstToIfAdj:      -20,
    passConstToNestedIfAdj: -15,
    passConcreteToItfCallAdj: -30,
    passConcreteToNestedItfCallAdj: -25,
    passFuncToIndCallAdj:   -25,
    passFuncToNestedIndCallAdj: -20,
    passInlinableFuncToIndCallAdj: -45,
    passInlinableFuncToNestedIndCallAdj: -40,
    returnFeedsConstToIfAdj: -15,
    returnFeedsFuncToIndCallAdj: -25,
    returnFeedsInlinableFuncToIndCallAdj: -40,
    returnFeedsConcreteToInterfaceCallAdj: -25,
}
```

Let's try a quick experiment to see this in action. One of the adjustment types is `panicPathAdj`, which adds a penalty of 40 units. This applies to function calls on paths that definitely lead to a panic.

Since panic paths are rarely executed, it makes sense to avoid inlining functions there and focus optimization on the common case:

```

1 func example(x int) {
2     doSomething() // NOT on a panic path
3     if x > 0 {
4         return
5     }
6     doSomething() // On a panic path
7     panic("negative x")
8 }
9
10 func doSomething() {
11     noInline()
12 }
13
14 //go:noinline
15 func noInline() string {
16     return "noinline"
17 }

```

In the `example` function, `doSomething()` is called twice, but in two different execution paths. Let's check what the compiler says:

```

$ GOEXPERIMENT=newinliner go build -gcflags="-m=2"
./main.go:17:6: cannot inline noInline: marked go:noinline
./main.go:12:6: can inline doSomething with cost 59 as:
func() { noInline() }
./main.go:3:6: can inline example with cost 130 as: func(int)
{ doSomething(); if x > 0 { return }; doSomething();
panic("negative x") }
./main.go:4:13: inlining call to doSomething with score 59

```

Here's how to interpret the output:

- The `doSomething` function is inlineable and is successfully inlined into the `example` function with a cost of 59 units.
- The `example` function is also considered inlineable with a total cost of 130 units, which is acceptable under the increased function budget of 160 units.

The second call to `doSomething` is on the panic path, and its score is increased due to the 40-unit penalty, making it exceed the 80-unit call site budget. So, it's not inlined because its adjusted cost is 99 units (59 base cost + 40 penalty from `panicPathAdj`), which exceeds the call site budget of 80 units.

Let's move quickly through other adjustment types:

- **`initFuncAdj` (+20)**: Functions called inside initialization functions (`init()` or package-level variable initializers) are less likely to benefit from inlining, as they usually run once.
- **`inLoopAdj` (-5)**: Calls inside loops are encouraged to be inlined, since even small savings can add up over repeated executions.
- **`passConstToIfAdj` (-20) and `passConstToNestedIfAdj` (-15)**: When a constant is passed to a function that uses it in a conditional (`if`) statement, inlining is strongly favored.
- **`passConcreteToIntfCallAdj` (-30) and `passConcreteToNestedIntfCallAdj` (-25)**: When a concrete type is passed to a function that uses it in an interface method call, inlining is favored. This helps convert dynamic dispatch into static calls. For example:

Before	After
<pre>func makeItSpeak(a Animal) string {     return a.Speak() }  func main() {     d := Dog{}     println(makeItSpeak(d)) }</pre>	<pre>func main() {     d := Dog{}     println(d.Speak()) }</pre>

This optimization is powerful because it eliminates the runtime overhead of dynamic dispatch from interface calls (as discussed in Chapter 4), replacing them with direct method calls.

These next adjustments apply a similar idea: turning indirect calls into direct calls known at compile time:

- **`passFuncToIntfCallAdj` (-25) and `passFuncToNestedIntfCallAdj` (-20)**: These apply when a function receives another function as a parameter and then

calls it. Inlining here helps avoid the indirect call overhead.

- `passInlinableFuncToIndCallAdj` (-45) and `passInlinableFuncToNestedIndCallAdj` (-40): These offer the biggest score reductions. If a function is passed as a parameter and is also inlinable, inlining can remove the indirect call and possibly inline the passed function as well.

Finally, there are adjustments based on how return values are used:

- `returnFeedsConstToIfAdj` (-15): Applies when the function returns a constant that is immediately used in a conditional.
- `returnFeedsFuncToIndCallAdj` (-25), `returnFeedsInlinableFuncToIndCallAdj` (-40), and `returnFeedsConcreteToInterfaceCallAdj` (-25): These follow the same principle as the parameter adjustments. They reward inlining when it helps eliminate indirect call overhead in how the result is used.

For instance, consider a function that directly returns a concrete type, which is then converted to an interface type in the return statement:

Before	After
<pre>func newDog() Animal {     return Dog{} }  func main() {     animal := newDog()     animal.Speak() // returnFeedsConcreteToInterfaceCallAdj }</pre>	<pre>func main() {     dog := Dog{}      dog.Speak() }</pre>

The inlining call to `newDog()` receives a negative score (-22), which means it is highly encouraged to be inlined. This is due to both its small size and the fact that it returns a concrete value that is immediately used in a context that could benefit from devirtualization. Let's run the compiler with the new inliner enabled and examine the output:

```
$ GOEXPERIMENT=newinliner go build -gcflags="-m=2"

./main.go:9:6: can inline Dog.Speak with cost 2 as:
method(Dog) func() string { return "Woof!" }
./main.go:13:6: can inline newDog with cost 3 as: func()
```

```

Animal { return Dog{} }
./main.go:17:6: can inline main with cost 69 as: func() {
animal := newDog(); animal.Speak() }
./main.go:18:28: inlining call to newDog with score -22
./main.go:19:14: devirtualizing animal.Speak to Dog
./main.go:19:14: inlining call to Dog.Speak with score 2

```

What stands out here is the devirtualization of `animal.Speak` to `Dog.Speak`. This optimization became possible because `newDog()` was inlined. So in this case, **inlining enabled devirtualization**.

Previously, we mentioned that devirtualization can create new opportunities for inlining, because the compiler learns exactly which function is being called. Now we see the reverse: inlining can reveal more information and open the door for devirtualization.

The Go compiler implements this using an **interleaved** strategy for inlining and devirtualization.

Figure 98. DevirtualizeAndInlineFunc  
(src/cmd/compile/internal/inline/interleaved/interleaved.go)

```

// DevirtualizeAndInlineFunc interleaves devirtualization and
// inlining
// on a single function.
func DevirtualizeAndInlineFunc(fn *ir.Func, profile
*pgoir.Profile) {
    ir.WithFunc(fn, func() {
        ...

        edit := func(n ir.Node) ir.Node {
            call, ok := n.(*ir.CallExpr)
            if !ok { return nil } // previously
inlined

            devirtualize.StaticCall(call) // Try
devirtualization first
            if inlCall :=
inline.TryInlineCall(fn, call, bigCaller, profile); inlCall
!= nil {
                return inlCall // If
inlining succeeds, return the inlined call
            }
            return nil // No changes made
        }

        fixpoint(fn, match, edit) // Apply
repeatedly until no more changes
    }
}

```

```
    })  
}
```

Instead of running inlining and devirtualization as separate passes, the compiler combines them into a single integrated step. It applies both repeatedly on a function until no further changes are detected. Each time the `edit` function returns a non-nil result, the IR node is updated, and another iteration begins. This continues until every call to `edit` returns `nil`, indicating that no more transformations are possible.

#### 5.6.4 Escape Analysis

The last optimization in this stage is **escape analysis**. This process determines whether a variable should be allocated on the stack or the heap.

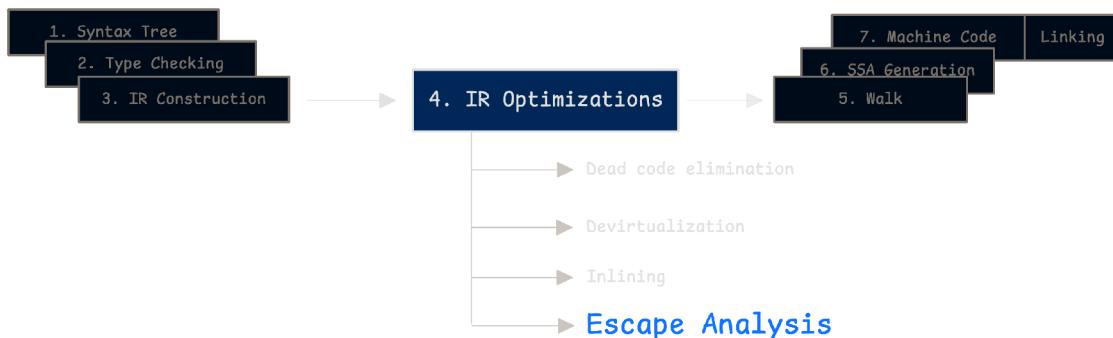


Illustration 125. The compiler applies escape analysis after inlining and devirtualization

We won't go into the deeper mechanics of escape analysis here, since it will be covered more thoroughly in the memory-focused Chapter 7 where it plays a larger role.

The main goal of escape analysis is to understand a variable's **scope** and **lifetime**. The compiler inspects how each variable is used to decide whether it can remain within the stack frame of the function or needs to be accessible beyond it. If a variable must outlive the function where it's declared, it is allocated on the heap instead:

#### Escapes

```
//go:noinline  
func returnPointer()  
*int {  
    a := 3
```

#### May Not Escape

```
func main() {  
    a := 3  
    doSomethingWithPointer(&a)
```

Escapes	May Not Escape
<pre style="margin: 0;">    return &amp;a }</pre>	<pre style="margin: 0;">} //go:noinline func doSomethingWithPointer(ptr *int) {     // do something with     the pointer }</pre>

The first example is a classic case of a variable that escapes. The variable `a` is declared inside `returnPointer` and returned as a pointer. Because the pointer's lifetime extends beyond the scope of `returnPointer` function, `a` must be allocated on the heap.

Or another explanation: When `returnPointer` finishes, its stack frame is no longer available. If `a` had remained on the stack, the returned pointer would point to invalid memory.

In the second example, the variable `a` does not escape just because a pointer to it is passed into another function. The pointer `ptr` is used within `doSomethingWithPointer`, but that function does not store the pointer or extend its lifetime in any way. Therefore, `a` can stay on the stack.

But be cautious—intuition can be misleading. The actual escape behavior depends on context, visibility, and usage. If you care about performance in hot paths, prove it with the static analysis output:

```
go build -gcflags='-m'
```

That said, the second example can still lead to escape if the code evolves. For example:

- If `doSomethingWithPointer()` stores the pointer in a global variable, `a` will escape.
- If it launches a goroutine that uses the pointer, `a` will escape.
- If the function is not inlinable or not visible during compilation (e.g., passed as a function value or called through an interface), the compiler may conservatively assume the pointer escapes.

Unlike inlining or devirtualization, escape analysis does not transform the IR structure. Instead, it updates the existing IR nodes by tagging them with escape

information. These annotations are then used later during code generation to decide whether a variable should go on the stack or be moved to the heap.

## 5.7 Stage 5: Walk (Middle end)

The walk phase is the final pass over the IR nodes of our application. It breaks down complex statements into simpler ones that are easier for the next stage to process.

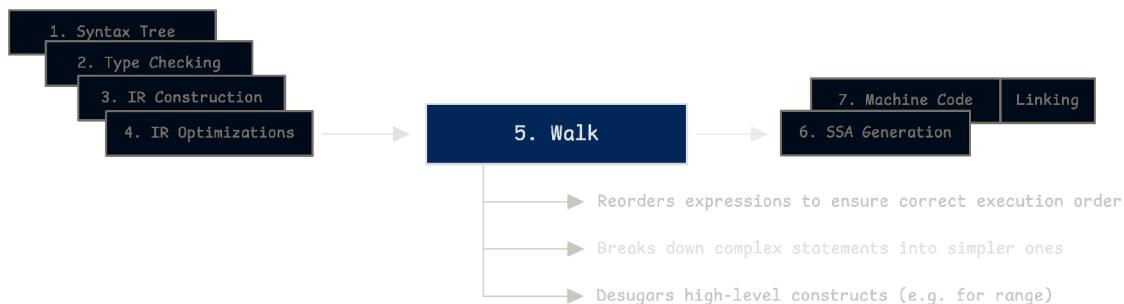


Illustration 126. Walk phase simplifies and flattens IR

If that sounds a bit unclear, let's look at a simple example. You might write a for-range loop like this on the left:

Go	Pseudo-Go
<pre>for v1 := range a {     // loop body }</pre>	<pre>hv1 := 0 hn := len(a) for hv1 &lt; hn; hv1 = hv1 + 1 {     v1 = hv1     // loop body }</pre>

This is a good example of 'desugaring': taking a high-level Go construct and converting it into basic operations that more directly show what happens at runtime.

This stage has two main parts: ordering and transformation (also called walking, the same name as the stage). These parts often overlap, since transformation may call ordering as it works.

## Order Phase

The order phase in the Go compiler is all about making sure that everything in your code happens in the *right sequence*. Think of it like a checklist of tasks, where some tasks depend on the results of others. You need to arrange them so everything runs in the correct order.

The main job of this phase is to walk through a statement, find parts that have *side effects*, and lift those into separate statements.

So, what are side effects? Let's look at a short assignment statement:

```
x := f(g())
```

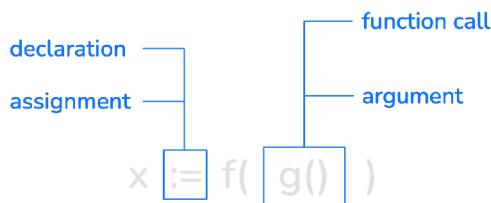


Illustration 127. Assignment decomposed into declaration and call

Here, we look at three parts: the assignment node, the left-hand side, and the right-hand side.

- **The assignment node (`:=`)**: This operator performs both assignment and declaration. Declaring `x` involves initialization, which is considered a side effect in this context.
- **The left-hand side (`x`)**: This is the variable being assigned to. In this case, `x` is just a name, so there are no side effects here.
- **The right-hand side (`f(g())`)**: This is a call to `f()`, with an argument `g()`. Since `g()` is a function call, it can have side effects. That means `f()` has to wait until `g()` runs.

The order phase handles this by lifting any side effects into their own statements. It uses temporary variables to hold intermediate results. If `f()` returns a string, the compiler transforms the original line into something like this:

```
var x string
tmp := g()
x = f(tmp)
```

We'll take a closer look at how this works inside the compiler, using the `for-range` loop as an example. This will help you see what's really going on behind the

scenes and you'll be able to explore the compiler code yourself—even as it changes in future Go versions.

To begin, let's look at a subtle example:

```
func main() {
    arr := [3]int{1, 2, 3}

    for i, v := range arr {
        if i == 1 {
            arr[2] = 99
        }
        println(v) // Print the value
    }
}
```

We start with an array `arr` containing three integers: `[1, 2, 3]`. A for-range loop is used to iterate over the array and print each element's value.

Here's the catch: just before reaching index 2, the code updates that element to `99`. What do you think the output is?

Some might expect `1 2 99`, but the actual output is `1 2 3`. This happens because the compiler uses a protective strategy to avoid unexpected behavior when the loop's source is modified during iteration. That strategy is handled in the order phase.

## For-Range Order

In the ordering phase, the compiler traverses the function's structure. It examines each statement and then the expressions contained within those statements:

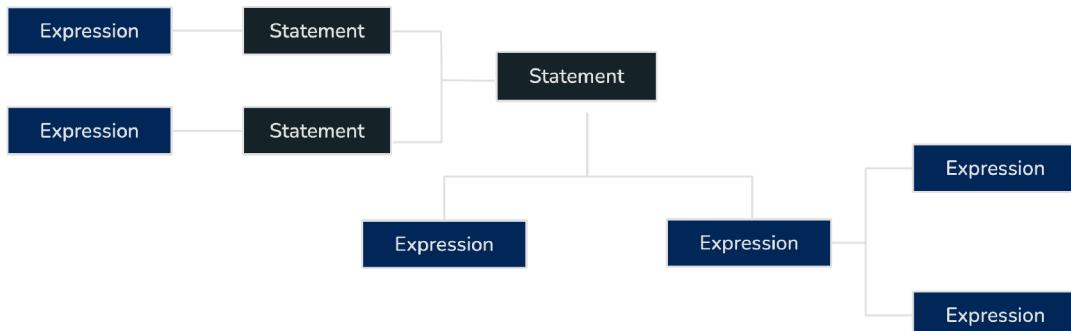


Illustration 128. Statement tree composed of expressions

This traversal process is similar to how the Go compiler handles the encoding and decoding of export data into IR nodes. Let's briefly revisit the structure of IR nodes:

- Functions are composed of a sequence of statements.
- Statements can vary in form and may contain nested statements or expressions.
- Ultimately, all program logic is represented by expressions.

In the IR, expressions and statements are usually represented by different node types. However, some expressions can appear as statements. A common case is function calls. These are 'call expressions', but when used by themselves, they act like standalone statements.

Let's examine the structure of the IR nodes for the for-range loop:

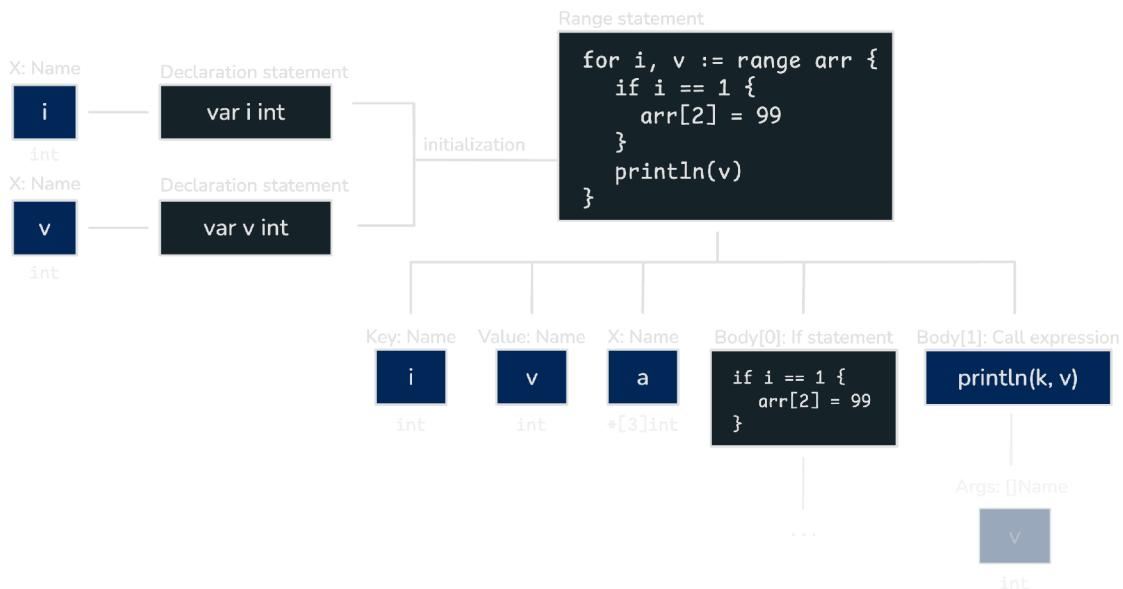


Illustration 129. IR structure of a for-range loop

Each statement in a function is ordered separately. The function below is responsible for ordering individual statements:

Figure 99. Ordering Statements (src/cmd/compile/internal/walk/order.go)

```
func (o *orderState) stmt(n ir.Node) {
    ...
    switch n.Op() {
    default:
        base.Fatalf("order.stmt %v", n.Op())
    ...
}
```

```

        case ir.OCALLFUNC, ir.OCALLINTER:
            n := n.(*ir.CallExpr)
        ...

        case ir.ODEFER, ir.OGO:
            n := n.(*ir.GoDeferStmt)
        ...

        case ir.OFOR:
            n := n.(*ir.ForStmt)
        ...

        case ir.OIF:
            n := n.(*ir.IfStmt)
        ...

        case ir.ORANGE:
            n := n.(*ir.RangeStmt)
        ...

        case ir.ORETURN:
            n := n.(*ir.ReturnStmt)
        ...

        case ir.OSWITCH:
            n := n.(*ir.SwitchStmt)
        ...
    }

    base.Pos = lno
}

```

*This function spans over 600 lines and includes many cases due to bug fixes and optimizations. The excerpt above only shows part of it and not all the supported node types.*

If we assume our `for-range` statement node is `n`, then the structure of the range node (`ORANGE`), as shown in the earlier diagram, would look like this:

```
for n.Key, n.Value = range n.X { n.Body }
```

When the compiler encounters a range statement, it starts by identifying the expression being ranged over. This is represented as `n.X`. The source can be an array, slice, string, map, channel, or even a number (like `for i := range 10`).

One of the first special cases involves ranging over a string-to-bytes conversion:

```

for i, v := range []byte(str) {
    // loop body
}

```

As discussed in Chapter 3, strings in Go are immutable. That means their contents cannot be changed.

When you convert a string to a byte slice using `[]byte(str)`, Go creates a new copy of the string's underlying data. This is necessary because byte slices are mutable. Allowing modifications to reflect directly on the original string would break its immutability rule.

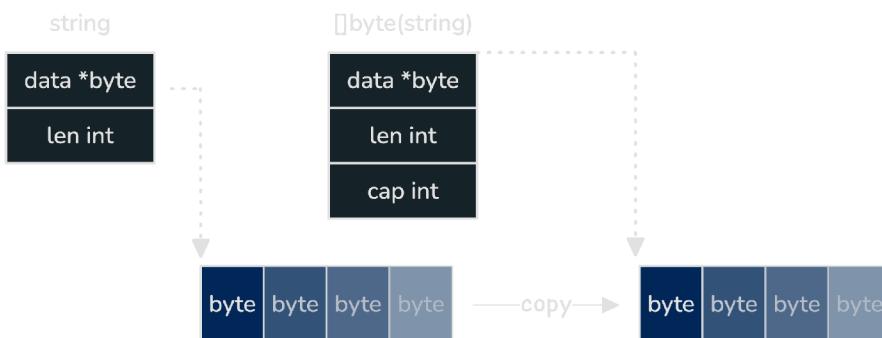


Illustration 130. String converted to byte slice copy

However, what if we convert a string to a byte slice but never mutate it afterward? That is exactly the case in the snippet above.

In such a situation, copying the string's underlying data can be a waste of resources. Back in Chapter 2, we used `unsafe` as a trick to convert a string to a byte slice without copying. But in many cases, we might not need to do that at all.

The Go compiler is smart enough to detect when the conversion is safe to optimize. It marks the expression with a special operation code called `OSTR2BYTESTMP`:

```

case ir.ORANGE:
    n := n.(*ir.RangeStmt)

    if x, ok := n.X.(*ir.ConvExpr); ok {
        switch x.Op() {
            case ir.OSTR2BYTES:
                x.SetOp(ir.OSTR2BYTESTMP)
                fallthrough
            case ir.OSTR2BYTESTMP:
                x.MarkNonNil() // "range []byte(nil)"
        }
    }

```

```
}
```

The `OSTR2BYTESTMP` node tells the SSA generation stage (which comes after the walk phase) that this string-to-byte conversion does not need to allocate a new slice. Instead, it can safely reuse the original data using a zero-copy optimization.

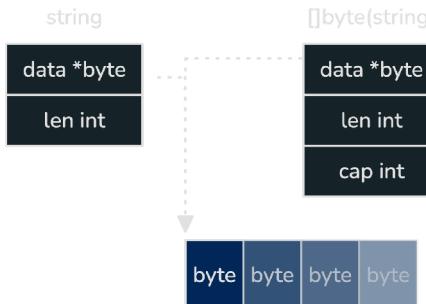


Illustration 131. Zero-copy optimization for underlying bytes

## Zero-copy Optimization

Zero-copy actually happens earlier than this stage, during escape analysis in the optimization phase. That means by the time we reach this code, the operation is often already marked as `ir.OSTR2BYTESTMP`. It will skip the `case ir.OSTR2BYTES` check because escape analysis already changed it.

If you're curious when this optimization kicks in, you can run your build with the `-m` flag to see the compiler's decisions. For example:

```
package main

func main() {
    for _, b := range []byte("hello") {
        _ = b
    }
}

$ go build -gcflags="-m" .

# theanatomyofgo
./main.go:3:6: can inline main
./main.go:4:27: ([]byte)("hello") does not escape
./main.go:4:27: zero-copy string->[]byte conversion
```

So before reaching for unsafe tricks, take a moment to check if the compiler already does the work for you. If it doesn't, it could be because the slice is not safe for this optimization.

Next, let's consider how the compiler processes the expression `n.X`, which represents the array (`arr`) in this loop:

```
func main() {
    arr := [3]int{1, 2, 3}

    for i, v := range arr {
        if i == 1 {
            arr[2] = 99
        }
    }
}
```

The compiler uses the `o.expr` function to handle this expression:

Figure 100. Ordering `n.X` in for-range loop  
(src/cmd/compile/internal/typecheck/stmt.go)

```
case ir.ORANGE:
    ...
    t := o.markTemp()
    n.X = o.expr(n.X, nil)

    orderBody := true
    xt := typecheck.RangeExprType(n.X.Type())
```

When processing complex expressions, the compiler often introduces temporary variables to store intermediate results. This approach guarantees the correct order of evaluation and makes these intermediate values available for later stages of compilation.

For instance, consider how the following expression is broken down using temporary variables:

Go	Pseudo-Go
<code>result = a &gt; 5 &amp;&amp; b()</code>	<code>temp0 := a &gt; 5 if temp0 {     temp1 := b()     temp0 = temp1 }</code>

Go	Pseudo-Go
	result = temp0

Internally, the Go compiler uses a stack to handle temporary variables, often named `temp0`, `temp1`, etc. As it works through code blocks, it places new temporaries onto this stack or reuses existing ones when possible. The call to `o.markTemp()` marks the current point in the stack. Later, the compiler can clean up only the temporaries used within that scope.

Once the marker is set, the compiler orders the `n.X` expression:

Figure 101. Ordering `n.X` in for-range loop  
(src/cmd/compile/internal/typecheck/stmt.go)

```
n.X = o.expr(n.X, nil)
xt := typecheck.RangeExprType(n.X.Type())
```

In this case, nothing really changes in `n.X` because `arr` is just a simple name node. The interesting part is in the `RangeExprType` function:

```
func RangeExprType(t *types.Type) *types.Type {
    if t.IsPtr() && t.Elem().IsArray() {
        return t.Elem()
    }
    return t
}
```

What the code above is saying is: when you have a pointer to an array and use it in a `range` loop, the compiler makes ordering decisions as if it were a regular array. For example:

```
var p *[10]int
for i, v := range p {
    // ...
}
```

In this case, the ordering phase only needs to know that `p` is a pointer to an array in order to make its decisions. The actual dereferencing is handled later during the walk phase. And this is why you're allowed to use `range` over a pointer to an array in Go. However, this behavior is specific to arrays. It's a special case. The same does not apply to pointers to slices or maps, and we'll explore why this exception exists later on.

Now, based on the type of `n.X`, the compiler will handle it in different ways:

```
case ir.ORANGE:
    ...
    switch k := xt.Kind(); {
        case types.IsInt[k]:
        case k == types.TARRAY, k == types.TSLICE:
            if n.Value == nil || ir.IsBlank(n.Value) {
                break
            }
            fallthrough
        case k == types.TCHAN, k == types.TSTRING:
            r := n.X

            if r.Type().IsString() && r.Type() != types.Types[types.TSTRING] {
                r = ir.NewConvExpr(base.Pos,
ir.OCONV, nil, r)
                r.SetType(types.Types[types.TSTRING])
                r = typecheck.Expr(r)
            }

            n.X = o.copyExpr(r)
        case k == types.TMAP:
            ...
            r := n.X
            n.X = o.copyExpr(r)
            ...
    }
```

The goal of this logic is to prevent unexpected behavior when the value being iterated over (`n.X`) is modified during the loop.

If `n.X` is an integer, no extra steps are needed. Integer iteration is simple, and there's no risk of data being changed while iterating. The same applies to arrays or slices when the loop does not use the value part (`v`) of the range expression:

```
for i := range n.X
for i, _ := range n.X
```

In these cases, there is no `v` to be affected by any updates inside the loop. But if `v` is present and used in the loop body, the compiler needs to ensure that its value does not reflect any changes made to `n.X` during iteration.

To do this, the compiler creates a temporary variable using `n.X = o.copyExpr(n.X)`. This holds a snapshot of the original data for iteration:

## Go

```
for i, v := range arr {  
    // loop body  
}
```

## Pseudo-Go

```
temp := arr  
for i, v := range temp {  
    // loop body  
}
```

This explains the behavior we saw earlier when iterating over an array. If you change the element at index 2 of the array `arr` during the loop, the update does not affect `v`. That's because `v` is reading from the temporary copy, not from the original `arr`. In other words, what we modify is different from what the loop is actually iterating over.

Now, because the copy is just an assignment, the behavior changes when we iterate over a slice:

```
1 func main() {  
2     slice := []int{1, 2, 3}  
3  
4     for i, v := range slice {  
5         if i == 1 {  
6             slice[2] = 99  
7         }  
8         println(v)  
9     }  
10 }  
11  
12 // Output: 1 2 99
```

As we know, slices in Go are implemented as headers. When the compiler copies a slice, it only copies the header, not the underlying array. This is much more efficient than copying an entire array.

On the other hand, copying a large array creates a performance cost. This is why Go allows ranging over a pointer to an array (`*[N]T`) but not a pointer to a slice (`*[]T`), as it helps avoid unnecessary memory usage in those cases.

Since only the slice header is copied, the loop operates over the original length and capacity. Even if we append elements to the slice inside the loop, the loop does not grow beyond its initial length:

```
func main() {
    a := make([]int, 3, 5)

    for i, v := range a {
        a = append(a, v)
        println(i)
    }
}

// Output: 0 1 2
```

In this example, we append a value to the slice during each iteration. Still, the loop runs exactly three times, based on the original length of the slice.

Previously, we saw that this copying behavior happens when the value `v` is used in the loop. What happens if we don't use `v` at all?

```
func main() {
    a := make([]int, 3, 5)

    for i, _ := range a {
        a = append(a, i)
        println(i)
    }
}

// Output:
// 0
// 1
// 2
```

Even though we are appending to the slice on every iteration, the loop still runs only three times. So how does Go avoid running the loop forever? The reason lies in the walk phase.

## Walk Phase

The walk phase takes the simplified and ordered IR nodes from the order phase and applies transformations to prepare the code for SSA generation. Its main role is to lower high-level Go constructs into simpler forms that can be directly translated into SSA.

This step is necessary because Go's high-level features are often too abstract or idiomatic to map directly to the machine model that SSA represents. Features such as channels, interfaces, and `defer` statements carry behaviors that are not easily expressed in low-level form without some translation.

The walk phase acts as this crucial translation layer. It converts Go's rich and expressive syntax into simpler operations that align more closely with the expectations of SSA. Here are a few examples of what the walk phase does:

- Channel send operations become runtime calls such as `runtime.chansend1`.
- `defer` statements are transformed into internal bookkeeping logic.
- Interface method calls are lowered into concrete dispatch code.
- Composite literals (like struct or array initializations) are broken into simple assignment statements.

While the order phase ensures that operations are evaluated in the correct sequence and handles side effects using temporary variables, it does not change the meaning of those operations.

In contrast, the walk phase performs deeper changes. It rewrites many of Go's advanced features into lower-level patterns that are easier for later stages of the compiler to handle.

For example, a high-level `for-range` loop is transformed into a more basic `for` loop structure. Similarly, calls like `make(map)` are rewritten into calls to lower-level runtime helpers like `runtime.makemap_small`, `runtime.makemap64`, or `runtime.makemap`. We discussed this behavior earlier in the section on maps.

## For-Range & Loop-var Problem

Just like the order phase, the walk phase starts at the function level, goes through each statement in the body, and applies the necessary transformations:

Figure 102. `walkStmt` (`src/cmd/compile/internal/walk/stmt.go`)

```
func walkStmt(n ir.Node) ir.Node {
    ...
    walkStmtList(n.Init())
    switch n.Op() {
    ...
    case ir.ORANGE:
        n := n.(*ir.RangeStmt)
```

```
        return walkRange(n)
    }
}
```

The `walkRange` function handles the job of converting high-level Go `for-range` loops into lower-level forms:

```
func walkRange(nrange *ir.RangeStmt) ir.Node {
    base.Assert(!nrange.DistinctVars)

    ...
    nfor := ir.NewForStmt(nrange.Pos(), nil, nil, nil,
    nil, nrange.DistinctVars)
    nfor.SetInit(nrange.Init())
    nfor.Label = nrange.Label
}
```

The first thing this function does is lower the high-level range loop into a basic `for` loop structure. This `for` loop carries over the initialization and label from the original range statement. This transformation happens regardless of what type of data is being ranged over.

Inside a for loop structure, you might notice a field named `DistinctVars`. This flag helps the compiler handle a well-known problem in Go called the loop-var problem.

## Loop Variables Issue

The loop-var problem refers to an unexpected behavior in Go where loop variables are not recreated in each iteration. Instead, they reuse the same memory location. This becomes an issue when those variables are captured by closures or used in goroutines.

Here's a common example:

```
func main() {
    funcs := make([]func(), 3)
    for i := 0; i < 3; i++ {
        funcs[i] = func() { fmt.Println(i) }
    }
    for _, f := range funcs {
        f()
    }
}
```

```
// Output: 3 3 3
```

The expected output is `0 1 2`, but what you get is `3 3 3`. This happens because the loop variable `i` is reused across all iterations. Each closure captures a reference to the same memory location. By the time those closures run, the loop has already finished, and `i` holds its final value.

The `DistinctVars` flag is used by the compiler to generate a fresh variable for each iteration when needed. This prevents closures from capturing the wrong value.

The Go team addressed this issue in Go 1.22 by introducing the `DistinctVars` flag in loop nodes, as shown in the earlier code snippet. This flag allows the compiler to enable the new behavior while still preserving compatibility with older code.

Let's now see what happens when we capture the address of the loop variable:

```
func main() {
    loopVarPtrs := make([]*int, 3)
    for i := 0; i < 3; i++ {
        loopVarPtrs[i] = &i
    }

    for _, l := range loopVarPtrs {
        println(*l)
    }
}

// Output: 0 1 2
```

In this case, each iteration creates its own variable, so the output is `0 1 2`. However, the behavior might seem a bit counterintuitive.

It looks like we are taking the address of the same variable `i` in every iteration, but the result shows that `loopVarPtrs` contains three different variables. This is because the compiler internally rewrites the loop to generate distinct variables on each iteration.

Regarding backward compatibility: if your code was written for a Go version earlier than 1.22 but is compiled with Go 1.22 or newer, the compiler keeps the original behavior by default. It does this by not setting the `DistinctVars` flag, which ensures existing programs continue to work the same way.

Internally, the rewritten form of a loop using distinct variables is a bit more complex. But conceptually, you can think of it like this:

For	For-Range
<pre>for i := 0; i &lt; len(a); i++ {     i2 := i     // ... use i2 }</pre>	<pre>for k, v := range a {     k2 := k     v2 := v     // ... use k2, v2 }</pre>

It's important to note that the loop-var issue is not resolved in the walk phase. It is handled before, during the IR optimization phase. Specifically, this happens after inlining and devirtualization steps are complete.

That's why you see this assertion at the beginning of the `walkRange` function:

```
func walkRange(nrange *ir.RangeStmt) ir.Node {
    base.Assert(!nrange.DistinctVars)
    ...
}
```

At this point in the compilation process, the `DistinctVars` flag should be `false` because the transformation has already been applied and the flag is cleared afterward.

Let's refocus on the walk phase of a for-range loop. First, we examine how the compiler handles a for-range loop iterating over integers:

Figure 103. For-Range Over Integer: Desugaring Phase  
(src/cmd/compile/internal/walk/range.go)

```
case types.Int[k]:
    hv1 := typecheck.TempAt(base.Pos, ir.CurFunc, t)
    hn := typecheck.TempAt(base.Pos, ir.CurFunc, t)

    init = append(init, ir.NewAssignStmt(base.Pos, hv1,
nil))
    init = append(init, ir.NewAssignStmt(base.Pos, hn,
a))

    nfor.Cond = ir.NewBinaryExpr(base.Pos, ir.OLT, hv1,
hn)
```

```

nfor.Post = ir.NewAssignStmt(base.Pos, hv1,
ir.NewBinaryExpr(base.Pos, ir.OADD, hv1, ir.NewInt(base.Pos,
1)))

if v1 != nil {
    body = []ir.Node{rangeAssign(nrange, hv1)}
}

```

The compiler translates a `for-range` loop over an integer into a regular for loop using temporary variables:

### Go

```

for i := range n {
    // ... Body
}

```

### Pseudo Go

```

hv1 := 0
hn := n
for ; hv1 < hn; hv1++ {
    i = hv1
    // Body
}

```

We will move quickly here without going too deep into the compiler source. The focus will be on the desugared pseudo-code examples:

### Go

```

// Simple slice loop
for i, v := range slice {
    println(i, v)
}

```

### Pseudo Go

```

sliceTmp := slice
hv1 := 0
hn := len(sliceTmp)
for hv1 < hn {
    tmp := sliceTmp[hv1]
    i, v := hv1, tmp
    println(i, v)
    hv1++
}

```

```

// For-range without value
'v'
for i := range arr {
    doSomething(i)
}

```

```

hv1 := 0
hn := len(arr)
for hv1 < hn {
    i := hv1
    doSomething(i)
    hv1++
}

```

This addresses the earlier question: why does appending to a slice inside a for-range loop (when not using the value `v`) not result in an infinite loop?

The lowered code in the second example provides the answer. The loop iterates using the original length of the slice, which is determined before the loop begins. Therefore, appending new elements during the loop does not affect the number of iterations. This prevents infinite loops and maintains predictable behavior.

## Other Statements

Although we've focused on the for-range loop so far, it's interesting to see how the compiler also handles other kinds of statements. These examples give insight into how Go lowers high-level constructs into something more primitive and concrete.

Let's look at how map iteration is lowered. This brings back memories from Chapter 3 where we explored `runtime.mapiterinit` and `runtime.mapiternext`. Assume our map is of type `map[string]int`:

### Go

```
// for-
range map
without
value 'v'
for k :=
range
myMap {
    //
body
}
```

### Pseudo Go

```
ha := myMap
it := new(hiter)
runtime.mapiterinit(typeof(map[string]int),
ha, &it)

// Loop using the iterator
for it.key != nil {
    k := (*string)(it.key)
    // body
    runtime.mapiternext(&it)
}
```

```
// for-
range map
with
value 'v'
for k, v
:= range
myMap {
```

```
ha := myMap
it := new(hiter)
runtime.mapiterinit(typeof(map[string]int),
ha, &it)

// Loop using the iterator
```

Go	Pseudo Go
<pre>// body {}</pre>	<pre>for it.key != nil {     k := *(*string)(it.key)     v := *(*int)(it.elem)     // body     runtime.mapiternext(&amp;it) }</pre>

When ranging over a map, the Go compiler lowers the loop into runtime support functions. A special iterator (`hiter`) is initialized using `runtime.mapiterinit`. At each step, it extracts the current key and value from the iterator's internal fields (`it.key` and `it.elem`), prints them, and advances the iterator with `runtime.mapiternext`.

Next, the lowering of string iteration:

```
var s string  
  
for i, r := range s {  
    // ... body  
}
```

This high-level construct is lowered into the following form, which explicitly handles UTF-8 decoding and manages byte and rune positions manually:

```
s := "Hello, "  
sCopy := s  
  
hv1 := 0          // Current byte position  
hv1t := 0         // Start position of current rune (i)  
hv2 := rune(0)   // Current rune value (r)  
  
for hv1 < len(sCopy) {  
    hv1t = hv1  
  
    // Read the rune at the current position  
    hv2 = rune(sCopy[hv1])  
  
    // If ASCII (RuneSelf = 128),  
    // move one byte forward  
    if hv2 < utf8.RuneSelf {  
        hv1++
```

```

    } else {
        // Decode multibyte rune
        hv2, hv1 = runtime.decoderune(sCopy, hv1)
    }

    i, r = hv1t, hv2

    // ... body
}

```

Finally, we turn to `switch` statements and how the compiler handles them:

```

switch x {
case 1, 2:
    println("one or two")
case 3:
    println("three")
    fallthrough
case 4:
    println("three or four")
default:
    println("something else")
}

```

The compiler lowers this high-level switch into a series of `if` statements and labeled jumps. Each `case` becomes a label, and `goto` statements are used to direct control flow accordingly:

```

.autotmp_1 := x
goto .switch_dispatch

.s0:
    println("one or two")
    goto .switch_end

.s1:
    println("three")
    // fallthrough - no goto, continues to next case

.s2:
    println("three or four")
    goto .switch_end

.switch_dispatch:
    if .autotmp_1 >= 1 && .autotmp_1 <= 2 {
        goto .s0
    }
    if .autotmp_1 == 3 {
        goto .s1
    }
    if .autotmp_1 == 4 {

```

```

        goto .s2
    }
    goto .switch_default

.switch_default:
    println("something else")
    goto .switch_end

.switch_end:

```

And that brings us to the end of our exploration of SSA preparation. What makes this stage especially important is that SSA form requires a clean and structured representation of control flow and data. The walk phase delivers exactly that by converting Go's expressive and flexible syntax into a consistent, low-level form that works well with SSA's strict requirements.

## 5.8 Stage 6: Static Single Assignment (SSA) Generation

SSA, or Static Single Assignment, restructures our IR nodes to make later optimizations easier to apply. Before SSA, some optimizations were hard to do. SSA changes this by making data flow and code dependencies more visible and explicit. This clarity allows optimizations such as bound check elimination, `nil` check elimination, dead code elimination, constant propagation, etc.

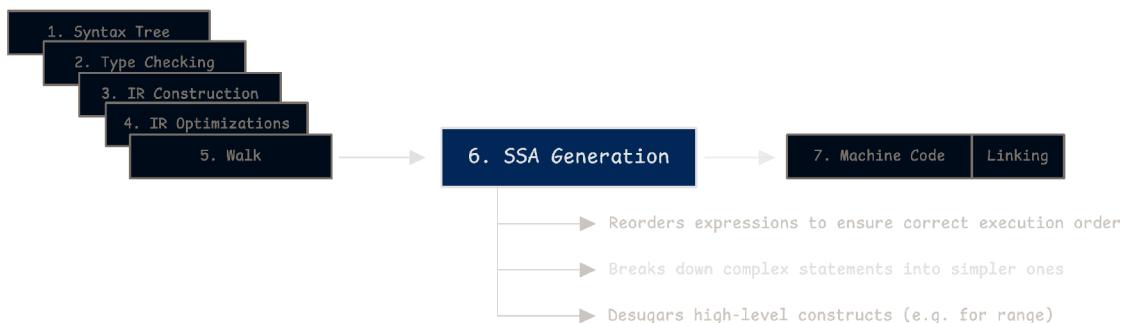


Illustration 132. SSA Generation bridges IR to machine code

Keith Randall (a member of the Go team) proposed and began building the SSA backend, which first appeared in Go 1.7. The initial release supported the AMD64 architecture.

*At GopherCon 2017 [[gpc2017](#)], Keith Randall reported significant performance improvements from this change. Benchmarks for Go 1.7 revealed a 12% increase in execution speed and a 13% reduction in code size. Go 1.8 extended SSA support to ARM, MIPS, and PPC architectures. The impact was*

*particularly notable on ARM. Applications saw a 20% speed increase, and code size decreased by 18%.*

The introduction of SSA marked a significant change in Go's compilation. One might assume that adding this complex stage would slow down the compiler, since more steps usually take more time. However, studies by the Go team, including Keith Randall, showed a more nuanced reality. While SSA introduces some overhead, it also enables optimizations that speed up both the final compiled programs and the compiler itself.

This self-improvement occurs because the compiler binary, like any other Go program, benefits from SSA optimizations. In some cases, these performance gains can outweigh the additional time spent during the SSA phase. Support for the SSA stage was extended to ARM64 in Go 1.8.

Here are some benchmark results from those earlier days:

- For the ARM architecture, the results were quite positive. The ARM compiler built with SSA ran approximately 10% faster. The improvements in the generated code were substantial enough to speed up the compiler, even with the added compilation overhead.
- For the AMD64 architecture, the outcome was different. The compiler experienced a slowdown of about 10%. In this case, the performance improvements from SSA did not fully offset the extra processing time.

*Modern Go compiler releases (Go 1.8 and later) have continued to optimize SSA across architectures, making these early benchmarks largely outdated.*

## SSA Idea

The fundamental principle of SSA form is that each variable is assigned exactly once. This means if a variable `a` is given a value like `10`, it can't be assigned a new value later. In SSA form, variables are effectively read-only after their initial assignment.

In contrast, standard code allows variables to be reassigned multiple times. This repeated assignment makes it challenging for the compiler to precisely track a variable's value at different points in the program. SSA addresses this by creating a new, distinct version of the variable for each assignment. Basically, if a variable's value changes, it gets a new name in the SSA representation.

Let's go through a simple example:

Table 1. SSA Form

Go	Go SSA
<pre>x = 1 ... x = x + 2</pre>	<pre>x1 = 1 ... x2 = x1 + 2</pre>

Instead of reusing `x`, SSA form creates new versions like `x1` and `x2` as needed. This removes any ambiguity about what a variable holds at a given point in the code.

Look at the first example. Can the compiler immediately turn `x = x + 2` into `x = 3`? Not really.

That's because the compiler can't be sure that nothing changed the value of `x` between the two assignments. It would have to scan every line in between, which makes optimization harder. In SSA form, it's much simpler. We can clearly see that `x2` depends only on `x1`, and `x1` is set to `1`. Since `x1` is never modified—thanks to SSA rules—the compiler can confidently rewrite `x2 = x1 + 2` as `x2 = 3`. It doesn't need to worry about any changes in between.

## SSA Syntax

When you write Go code, we work with variables, scopes, and blocks named in ways that make sense to us. SSA removes these concepts. It does not use variable names or traditional scopes. Instead, it works with values and blocks identified by unique, sequential IDs:

Go	Pseudo Go-SSA
<pre>func example(a int) int {     if a &gt; 10 {         return a * 2     }     return a + 2 }</pre>	<pre>entry:     v1 = Arg &lt;int&gt; {a} (a[int])     v2 = 10     v3 = 2     v4 = v1 &gt; v2     If v4 goto b1 else b2  b1:     v5 = v1 * v3     Ret v5  b2:</pre>

**Go**

**Pseudo Go-SSA**

```
v6 = v1 + v3
Ret v6
```

All values are labeled with IDs (`v1`, `v2`, `v3`, etc.) and blocks are named like `entry`, `b1`, and `b2`. The function begins at the `entry` block. It checks if `a > 10`. If `true`, it jumps to block `b1`. Otherwise, it jumps to block `b2`. Everything is spelled out clearly. There are no hidden scopes or implicit flow:

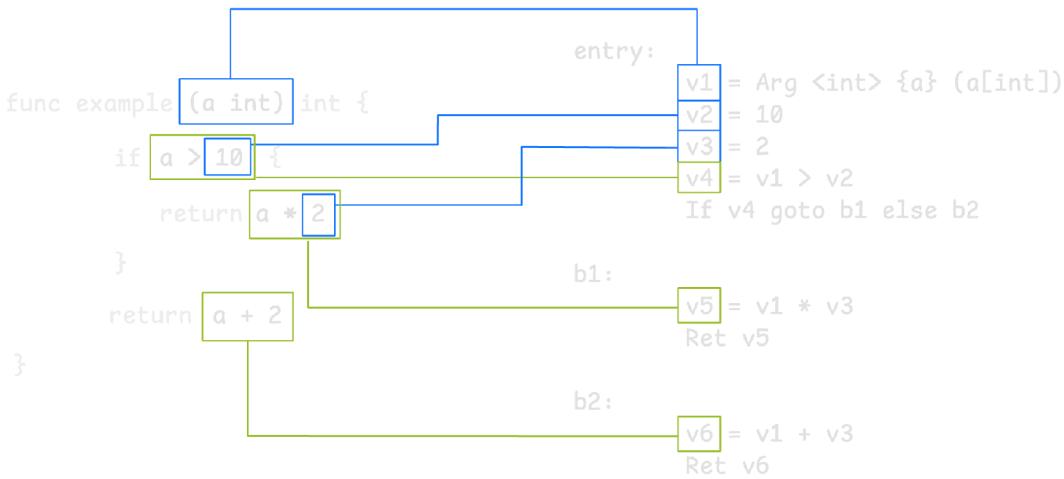


Illustration 133. Go-to-SSA breakdown: clear and explicit

Using sequential IDs also eliminates the need for the compiler to manage mappings between names and values. One part of the SSA syntax in the diagram that might look confusing is the following line:

```
v1 = Arg <int> {a} (a[int])
```

This represents the complete and accurate SSA syntax. Most of the previous examples were simplified for clarity. The actual format used internally by the Go compiler adheres to a consistent structure:

```
v# = opcode <type> [aux] args [: reg] (names)
e.g. v1 = Arg <int> {a} (a[int])
```

Illustration 134. Structure of an SSA value instruction

Here's a breakdown of the components:

- `v#`: A unique value identifier, like `v1`, `v2`, or `v3`.

- `opcode` : The operation being performed, such as `Add8`, `Sub64`, or `Arg` (used for function arguments).
- `<type>` : The data type of the value, for example, `<int>`, `<bool>`, or `<Float32>`.
- `[aux]` : Optional auxiliary information.
  - `[...]` is used for constants like integers `[0]`, `[1024]`, booleans `[true]`, `[false]`, or structured values like `[lsb=4,width=8]`.
  - `{...}` holds symbol-related data such as a symbol with an offset `{mystruct+16}`, a variable symbol `{foo+0}`, or a symbol reference like `{args}` used for globals, functions, and so on.
- `args` : Lists the value IDs used as inputs to the operation.
- `[: reg]` : Optional register allocation information, added later in the compilation process.
- `(names)` : Optional source-level variable names, mainly used for debugging, such as `(foo)`, `(bar)`, or `(args[*uint8])`.

The snippet below shows what the Go compiler actually works with during compilation. It includes full SSA syntax, memory tracking, and other internal details that earlier examples leave out:

```

b1:
    v1 = InitMem <mem>
    v2 = SP <uintptr>
    v3 = SB <uintptr>
    v4 = LocalAddr <*int> {a} v2 v1
    v5 = LocalAddr <*int> {~r0} v2 v1
    v6 = Arg <int> {a} (a[int])
    v7 = Const64 <int> [0]
    v8 = Const64 <int> [10]
    v9 = Less64 <bool> v8 v6
    v11 = Const64 <int> [2]
If v9 → b3 b2 (4)

b2: ← b1
    v15 = Copy <int> v6 (a[int])
    v16 = Add64 <int> v15 v11
    v17 = Copy <mem> v1
    v18 = MakeResult <int,mem> v16 v17
Ret v18 (+7)

b3: ← b1
    v10 (5) = Copy <int> v6 (a[int])
    v12 (5) = Mul64 <int> v10 v11
    v13 (5) = Copy <mem> v1
    v14 (5) = MakeResult <int,mem> v12 v13
Ret v14 (+5)

```

```
name a[int]: v6 v10 v15
```

It might look complex at first, but some parts should look familiar. For example, look at the constant assignments: `v8 = Const64 <int> [10]` and `v11 = Const64 <int> [2]`. These are the same values used in the earlier pseudo SSA example, shown there as `v2` and `v3`. This form may include more structure and internal bookkeeping, but the logic stays the same. With a closer look, it becomes easier to follow.

## Value & Operation

In SSA, values prefixed with `v` are the basic units of computation. Once defined, the value never changes, but it can be used as many times as needed:

```
v8 = Const64 <int> [10]
v11 = Const64 <int> [2]
```

The right-hand side of an SSA instruction shows the operation used to produce the value on the left. This part always begins with an operation code (opcode) which tells the compiler what action or computation is being performed. For example:

- `Const64 [auxint]` : Creates a 64-bit constant value stored in the instruction's `auxint` field.
- `Add64` : Performs 64-bit integer addition between two values, producing their sum.
- `Less64` : Tests if the first argument is less than the second, returning a boolean result.
- `Arg` : Accesses a function parameter.

After the opcode, a type appears in angle brackets, such as `<int>`, to describe the data type of the result.



Illustration 135. How a constant integer value is represented in SSA

Next, some instructions include auxiliary information in brackets. For example:

- Square brackets ( `[]` ) are used for constant or simple values like numbers and booleans (e.g. `[10]` ).

- Curly braces ( `{}` ) are used for symbolic or structured data like names, symbols, and type information (e.g. `{a}` , `{person+8}` ).

Each operator has defined behavior and purpose. These definitions live in files like `opGen.go` and `_gen/*Ops.go` .

Figure 104. SSA Operators (src/cmd/compile/internal/ssa/opGen.go)

```
const (
    OpInvalid Op = iota
    OpAdd8
    OpAdd16
    OpAdd32
    ...
    OpLess64
    OpLess64U
    OpLess32F
    OpLess64F
    ...
    OpConst32F
    OpConst64F
    OpConstInterface
    OpConstSlice
    OpInitMem
    OpArg
    ...
    OpClosureCall
    OpStaticCall
    OpInterCall
    OpTailCall
    OpClosureLECall
    OpStaticLECall
    ...
)
```

## Phi Function

SSA works well for most scenarios, but there is a classic challenge: handling variables that can be assigned along different control flow paths:

Go	Pseudo SSA
<pre>x = 1 if condition {     x = 2 }</pre>	<pre>x1 = 1 if condition {     x2 = 2 }</pre>



In this case, `x` may end up holding either `1` or `2`, depending on whether the condition expression is true or false. Later, `y` takes the value of `x`, but at that point, it is unclear which version of `x` should be used. This creates a problem in SSA, because SSA requires that every value is assigned exactly once. So how does SSA handle a situation where a variable might have different values depending on the path taken?

This is where the phi operation (also called a  $\phi$ -node) comes in. It selects the correct value based on the path taken through the program:

```
y1 = phi(x1, x2)
```

This is SSA's way of solving the 'multiple definitions' problem. Here's what it looks like in actual SSA form:

```
v6 = Const64 <int> [1] // x1
v7 = Const64 <int> [2] // x2
v8 = Phi <int> v6 v7 // y1 is chosen based on control path
```

The phi node allows SSA to keep its rule of assigning each value only once, even when control flow splits and later merges.

## Memory

When a program runs, most computations happen using the processor's registers. Registers are very fast storage locations that the processor can access directly for arithmetic, comparisons, and other operations. Consider a simple calculation:

```
b = 5
a = 10 + b
```

In this case, `b` might be stored in a register or in memory. We do not know for sure yet. However, when computing the value of `a`, the value of `b` must be in a register. The processor loads `b`, performs the addition with constant `10`, and then stores the result in another register for `a`.

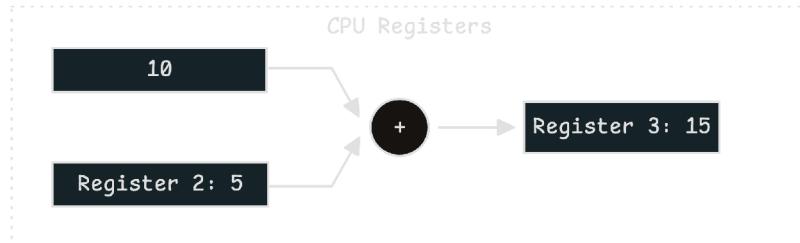


Illustration 136. Register-based arithmetic in action

SSA models this process clearly:

```
v_a = OpAdd v_const_10, v_b
```

This makes it obvious that `v_a` depends on `v_b`. SSA form helps the compiler see these dependencies, which is useful for optimizing register usage.

Now, memory is where data like variables, arrays, and structs are stored long-term. If `a` and `b` are placed in memory, as decided by the compiler, the operation would look more like this:

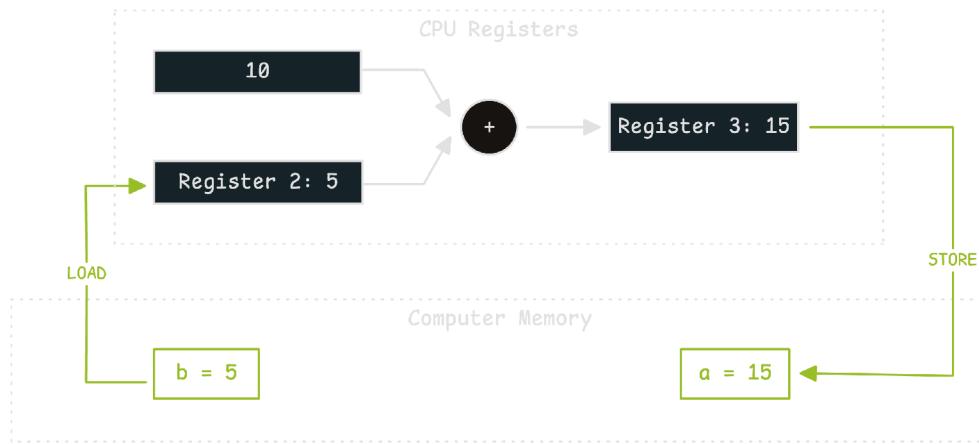


Illustration 137. From memory to registers and back

Before any computation occurs, the necessary values must be loaded from memory into registers. The SSA form operates under the assumption of an infinite supply of registers, represented by abstract SSA values like `v1`, `v2`, `v3`, and so on.

Consequently, every memory access corresponds to moving data between memory and these conceptual SSA values (e.g., loading from memory into `v1`, `v2`, or `v3`).

Consider the following example where variables `a`, `b`, and `c` reside in memory:

## Go

```
a = b + c
```

## Pseudo SSA

```
v1 = load <int> {b}
v2 = load <int> {c}
v3 = v1 + v2
store <int> {a} v3
```

The sequence of operations in SSA form is:

1. Load the value of `b` from memory into register `v1`.
2. Load the value of `c` from memory into register `v2`.
3. Add the values in `v1` and `v2`, storing the result in `v3`.
4. Store the result `v3` back to the memory location associated with `a`.

Although the final placement of variables (in memory or registers) is decided later, the compiler keeps track of all memory operations at this stage. A key factor influencing placement is whether a variable needs a stable memory address:

- Variables whose address is explicitly taken (using `&x`, for instance) must be allocated in memory. This ensures that pointers to the variable remain valid.
- Global variables are always allocated in memory.
- Variables whose address is never taken are candidates for optimization. They are considered 'SSA-eligible' and might be kept entirely in registers, avoiding the need for dedicated memory space.

Go compilers often reorder instructions to improve performance. This is usually safe for operations involving temporary values that do not affect memory directly:

```
// Original order:
v1 = Add64 v2 v3
v4 = Mul64 v5 v6
v7 = Sub64 v1 v4

// Can be reordered to:
v4 = Mul64 v5 v6
v1 = Add64 v2 v3
v7 = Sub64 v1 v4
```

The `Add64` and `Mul64` operations can be reordered because they only involve temporary values and don't affect memory.

However, memory operations are different. When a program reads from or writes to memory—such as storing values in variables, fields, or array elements—the order of those operations matters. If a program writes a value to `x` and then reads from

`x`, it expects to get the value that was just written. If the compiler reorders the read to happen before the write, the result would be incorrect.

Here is an example to show how this matters:

```
func compute(p, q *int) int {
    *p = 10
    temp := *q * 2 // Read through pointer q
    *p = 20         // Write through pointer p again
    return temp
}
```

The assignment `*p = 10` seems unnecessary, as the value is immediately overwritten by `*p = 20`. A common optimization would be to remove the first assignment, potentially simplifying the code to:

```
func compute(p, q *int) int {
    *p = 20
    temp := *q * 2
    return temp
}
```

Behind the scenes, this involves reordering instructions in SSA form:

Before	After
<pre>v1 = Const64 &lt;int&gt; [10] Store &lt;mem&gt; {*p} v1  v2 = Load &lt;int&gt; {*q} v3 = Mul64 &lt;int&gt; v2 2  v4 = Const64 &lt;int&gt; [20] Store &lt;mem&gt; {*p} v4</pre>	<pre>v1 = Const64 &lt;int&gt; [20] Store &lt;mem&gt; {*p} v1  v2 = Load &lt;int&gt; {*q} v3 = Mul64 &lt;int&gt; v2 2</pre>

This looks safe; but **only** if `p` and `q` point to different memory locations. There is a scenario where these two pointers `p` and `q` are pointing to the same memory location. In this case, the calculation will be incorrect and the ordering is breaking the semantics of the program.

To prevent incorrect reordering of memory-sensitive operations, the SSA representation introduces the **memory state**. This state acts as an artificial data dependency. The compiler assumes that any load operation must come after any changes that might affect the value being loaded.

Every operation that reads from or writes to memory, or could potentially do so (such as a function call), takes the current memory state token as an input:

```
m0 = InitMem // initial memory state
v1 = Const64 <int> [10]
m1 = Store <mem> {*p} v1 m0 // store to *p
```

Memory state tokens (e.g. `m0`, `m1`, `m2`) are abstract values that represent the state of all memory at a specific point in program execution. They're not actual memory locations or data; they're compiler bookkeeping mechanisms that ensure memory operations happen in the correct order.

The `Store` operation now takes `m0` as input along with `v1`, and it produces a new memory state token `m1`. The next memory operation must wait for `m1` token and use it as its input:

```
v2 = Load <int> {*q} m1      // v2 = *q = 10
v3 = Mul64 <int> v2 2        // v3 = v2 * 2 = 20
m2 = Store <mem> {*p} v3 m1 // *p = v3 = 20; store to *p
using m1.
```

The load operation takes in the `m1` memory state token, but since it does not modify memory, it does not produce a new memory state. This forms a sequence: `m0`, `m1`, `m2`, and so on—each representing the state of memory after a specific operation.

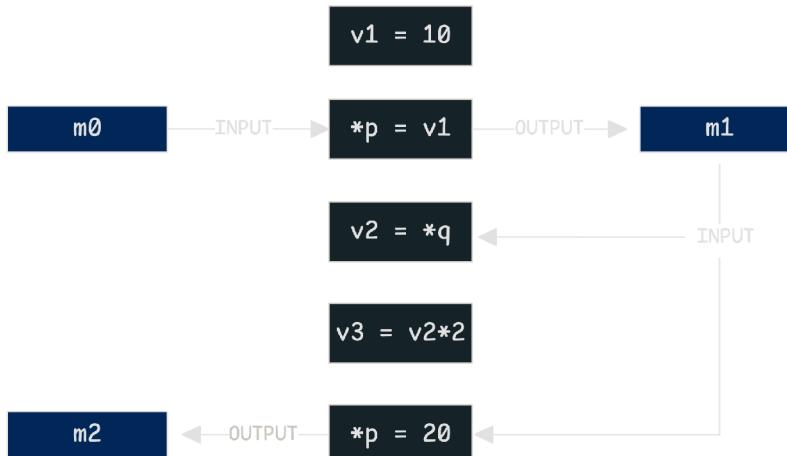


Illustration 138. Memory token flow in SSA form

This chain prevents the compiler from reordering memory operations in a way that could break program correctness. Even though the compiler may still reorder

independent arithmetic operations, the sequence of memory operations is kept intact by following this chain of tokens.

In SSA, memory state is treated like any other value. It can be passed, updated, and versioned just like registers or constants.

We use `m0`, `m1`, `m2` here to represent memory state for learning purposes. In actual SSA form, only `m0` is named this way. Later memory states will be represented as regular SSA values, like `v1`, `v2`, and so on.

The compiler ensures that only one memory state is active at any point in a basic block. There are three major types of operations that involve memory. Two of them we have already seen:

- **Load** operation: Loads a value from memory without changing it. It takes a memory state as input but does not produce a new one. Its form in SSA looks like this:

```
// Read an int from address 'ptr' using memory state 'mem'  
v1 = Load <int> ptr mem
```

- **Store** operation: Writes a value to memory. It takes a memory state as input and produces a new memory state as output. This new value represents the state of memory after the store:

```
// Store 'val' to address 'ptr', memory state transitions  
from 'mem' to 'v2'  
v2 = Store <mem> ptr val mem
```

- **Function call** operation: Calls are more complex because they can both read from and modify memory. The final argument to a call operation is always the current memory state. Call operations return a tuple with result values followed by the updated memory state:

```
// Call function 'fn', memory state transitions from 'mem' to  
'v3'  
v3 = StaticCall <mem> {fn} [argsize] mem
```

When optimizing code, the compiler uses alias analysis to decide whether certain memory operations can be reordered or eliminated. For instance, if two stores write to different memory locations and do not alias, they may be safely reordered for better performance.

However, proving that two pointers like `p` and `q` do not alias is difficult. Pointer aliasing is undecidable in the general case. Since correctness is more important than performance, the compiler takes a conservative approach. If it cannot prove that two memory accesses are independent, it must assume they might interfere.

## Block and Function

In the Go compiler backend, there are three main SSA components: functions (`ssa.Func`), blocks (`ssa.Block`), and values (`ssa.Value`).

A function in SSA represents a single Go function. It holds all of its basic blocks and tracks important metadata such as the entry point, architecture configuration, and flags that determine whether this function should be optimized or if it uses a specific ABI:

```
f := &ssa.Func{
    Name: "add",
    Type: *someSignature, // like func(int, int) int
    Blocks: []*ssa.Block{...},
    Entry: entryBlock,
    Config: ssaConfig, // architecture-specific info
}
```

A block represents a basic block. This is a straight-line sequence of instructions with no internal control flow. It has exactly one entry point and one exit. Blocks are connected through edges that define the control flow graph, using predecessor and successor relationships.

Here's an example of how an `if` statement is transformed into SSA:

Go	Go SSA
<pre>func example(a int) int {     if a &gt; 10 {         return a * 2     }     return a + 2 }</pre>	<pre>b1: v1 = InitMem &lt;mem&gt; v2 = SP &lt;uintptr&gt; v3 = SB &lt;uintptr&gt; v4 = LocalAddr &lt;*int&gt; {a} v2 v1 v5 = LocalAddr &lt;*int&gt; {~r0} v2 v1 v6 = Arg &lt;int&gt; {a} (a[int]) v7 = Const64 &lt;int&gt; [0] v8 = Const64 &lt;int&gt; [10] v9 = Less64 &lt;bool&gt; v8 v6 v11 = Const64 &lt;int&gt; [2]</pre>

Go	Go SSA
	<pre>If v9 → b3 b2 (4)  b2: ← b1       v15 = Copy &lt;int&gt; v6 (a[int])       v16 = Add64 &lt;int&gt; v15 v11       v17 = Copy &lt;mem&gt; v1       v18 = MakeResult &lt;int,mem&gt; v16 v17 Ret v18 (+7)  b3: ← b1       v10 (5) = Copy &lt;int&gt; v6 (a[int])       v12 (5) = Mul64 &lt;int&gt; v10 v11       v13 (5) = Copy &lt;mem&gt; v1       v14 (5) = MakeResult &lt;int,mem&gt; v12 v13 Ret v14 (+5)  name a[int]: v6 v10 v15</pre>

Each block contains a list of values that describe what operations are performed inside that block. Along with this, blocks have unique IDs, a block kind, and a list of successors which show where the control may go next. You can see this with `b2: ← b1` and `b3: ← b1`, meaning both `b2` and `b3` can be reached from `b1`.

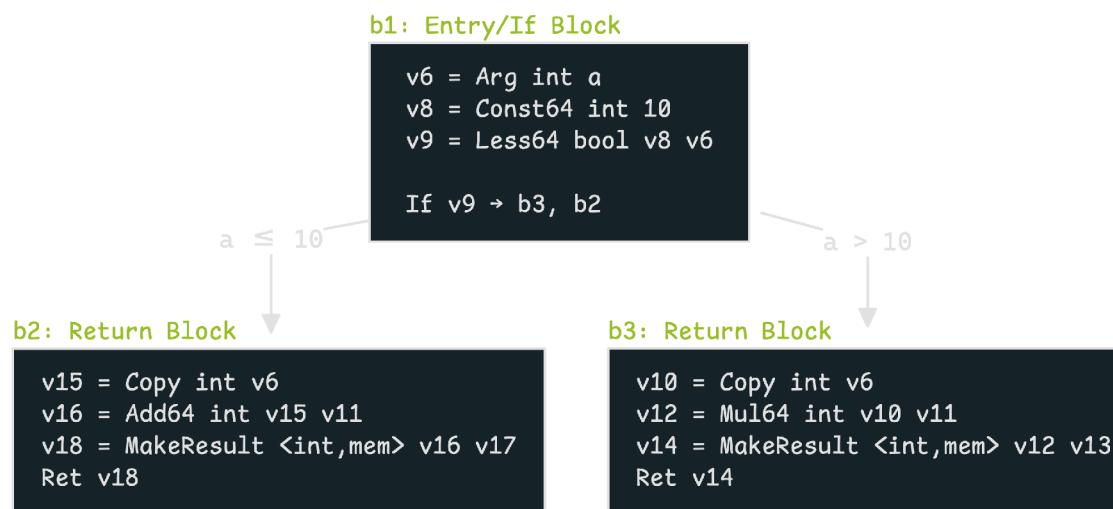


Illustration 139. Conditional branching from entry to return blocks

Let's examine the blocks in this graph:

- `b1` is the entry block and also an `If` block. It performs a comparison, then branches to another block depending on the result. Since it's the entry block, it does not have a predecessor.
- `b2` and `b3` are the two possible branches from `b1`. These are return blocks, as indicated by the `Ret` operation at the end of each one.

Each block also has a block type. Although SSA does not store this explicitly, the Go compiler uses internal logic to determine it. The easiest way to figure out the type of a block is by looking at the last instruction:

- `If` : A block that ends with a conditional branch.
- `Ret` : A block that returns from the function.

This structure helps the compiler understand and transform control flow during optimizations and code generation. Here are some kinds of blocks defined in the source code:

Figure 105. Block Kinds (src/cmd/compile/internal/ssa/opGen.go)

```
const (
    BlockInvalid BlockKind = iota
    ...
    BlockPlain
    BlockIf
    BlockDefer
    BlockRet
    BlockRetJmp
    BlockExit
    BlockJumpTable
    ...
)
```

Below is a quick overview of what each one does:

- `BlockPlain` : Represents a basic block with a single successor and no special control flow. It is used for straight-line code that flows directly to the next block.
- `BlockIf` : Has two successors: one for the `true` path and one for the `false` path. It uses a control value, which is a boolean expression that determines the direction of the flow.
- `BlockDefer` : Manages logic related to Go's `defer` mechanism. It checks whether to continue execution (return `0`) or jump to a deferred function (return `1`).

- `BlockRet` : Marks the point where a function returns to its caller. It usually holds the final memory state and any return values.
- `BlockRetJmp` : Supports tail call optimization. It allows the function to return and immediately jump to another function without an extra return-call step.
- `BlockExit` : Used when a function needs to exit due to a panic or an unrecoverable error.
- `BlockJumpTable` : Handles jump tables, which are often used to implement `switch` statements.

As discussed earlier, function calls are considered memory operations because they may read from or write to memory:

Go	Go SSA
<pre>func anything() {     // do nothing }</pre>	<pre>b1:     v1 = InitMem &lt;mem&gt;     v2 = SP &lt;uintptr&gt;     v3 = SB &lt;uintptr&gt;     v4 = MakeResult &lt;mem&gt; v1     Ret v4</pre>

Although this function does nothing meaningful, it still follows the same SSA layout. The structure includes memory initialization, pointer setup, and a return statement.

To sum up the key SSA relationships:

- **Function to Blocks:** A function is made up of one or more blocks that form its control flow graph.
- **Block to Values:** Each block contains values that represent computations.
- **Value to Block:** Every value belongs to exactly one block.
- **Block to Block:** Blocks are connected through predecessor and successor edges, forming the structure of the control flow.

## Building SSA From IR Nodes

Now that we understand how Go source code is transformed into blocks and values in SSA form, it's time to look at how this happens in the compiler. The transformation from IR nodes to SSA is long and detailed—but skipping it would mean missing out on valuable insight. To make it more engaging, we'll explore helpful domain knowledge along the way.

Let's begin with the entry point: the `ssagen.buildssa` function.

```
func buildssa(fn *ir.Func, worker int, isPgoHot bool)
*ssa.Func {
    name := ir.FuncName(fn)
    abiSelf := abiForFunc(fn, ssaConfig.ABI0,
sssaConfig.ABI1)
    if strings.Contains(ssaDump, name) { ... }
    ...
}
```

This function takes an IR function node and converts it into an SSA function.

The first steps involve setting up the environment. It gets the function name, determines the appropriate ABI (Application Binary Interface), and checks whether SSA output should be printed by looking at the `GOSSAFUNC` environment variable.

One term that might be unfamiliar to readers without a background in compiler design is **ABI**. ABI (Application Binary Interface) in the Go compiler refers to the set of rules that define *how functions interact at the binary level*. This includes how the caller passes arguments, how the callee returns values and how the caller can receive them. In earlier versions of Go, parameters were passed on the stack. Starting around Go 1.17, Go introduced a register-based ABI to improve performance.

ABI will be explained in more detail in the Chapter 7 when we discuss how stack frames work.

Now let's talk about `GOSSAFUNC`. This is an environment variable used by the compiler team and advanced users to debug SSA generation. For us, as people studying how SSA works, it is a very useful tool.

When `GOSSAFUNC` is set to the name of a function (for example, `main`), the compiler will dump the SSA form of that function into an HTML file or print it to standard output. This is especially useful because SSA generation is more than just converting IR to SSA. It also includes many optimization passes, and `GOSSAFUNC` helps us inspect each stage clearly.

You can try this with a simple Go program:

```
func main() {
    var a, b int
    p := &a
    q := &b
```

```

        *p = 42 // store to *p (a)
        *q = 84 // store to *q (b)
        y := *p // load from *p

        println("a:", a)
        println("b:", b)
        println("y:", y)
    }
}

```

To generate the SSA view as an HTML file, run this command:

```
$ GOSSAFUNC=main go tool compile main.go && open ssa.html
```



Illustration 140. SSA visualization using GOSSAFUNC debug flag

The compiler creates an HTML file (`ssa.html`) that shows the SSA representation of the specified function. It captures each phase of compilation, from the initial IR node conversion to the final Go assembly code. What makes it even more helpful is the interactive view. You can click on graph nodes, blocks, and values to highlight them and see how they are connected in the control flow graph, just like in the illustrations we've seen earlier.

Now, back to SSA generation from IR nodes. The first executable SSA block is created, serving as the entry block, similar to the ones shown in earlier examples:

```

func buildssa(fn *ir.Func, worker int, isPgoHot bool)
*ssa.Func {
    ...
    s.f.Entry = s.f.NewBlock(ssa.BlockPlain)
    s.f.Entry.Pos = fn.Pos()

    s.sp = s.entryNewValue0(ssa.OpSP,
types.Types[types.TUINTPTR])
    s.sb = s.entryNewValue0(ssa.OpSB,
types.Types[types.TUINTPTR])
    s.startmem = s.entryNewValue0(ssa.OpInitMem,
types.TypeMem)
}

```

This block sets up three important things:

- The virtual stack pointer ( `SP` ): points to the current execution frame in memory.
  - The static base pointer ( `SB` ): serves as a reference point for accessing global variables and constants.
  - The initial memory state ( `InitMem` ): the initial memory state of the function.

All of these registers will be explained in more detail when we discuss the function frame in the Chapter 7.

This block is required for every function. Even the empty function we saw earlier includes this block:

```
b1:  
    v1 = InitMem <mem>  
    v2 = SP <uintptr>  
    v3 = SB <uintptr>  
    v4 = MakeResult <mem> v1  
    Ret v4
```

Then, the Go compiler analyzes how this function handles parameters and local variables. Parameters may be passed either in registers or on the stack. The backend needs to determine where each parameter lives and whether it can be operated on directly using SSA, or if it must go through memory:

```

// Determine how parameters are passed (registers or stack)
params := s.f.ABISelf.ABIAnalyze(fn.Type(), true)

// Iterate through declared variables to handle function
input parameters
for _, n := range fn.Dcl {
    if n.Class == ir.PPARAM {
        // If the parameter can be represented by an SSA
        value (register)
        if s.canSSA(n) {
            v := s.newValue0A(ssa.OpArg, n.Type(), n)
            s.vars[n] = v
        } else {
            // If not, ask ABI for guidance
            paramAssignment :=
ssा.ParamAssignmentForArgName(s.f, n)
            if len(paramAssignment.Registers) > 0 {
                // ABI allows register passing, check if type
supports SSA
                if sса.CanSSA(n.Type()) {
                    v := s.newValue0A(ssа.OpArg, n.Type(), n)
                    s.store(n.Type(), s.decladdrs[n], v)
                }
            }
        }
    }
}

```

```

        } else {
            // Passed in register, but type is not
            // SSA-able
            // So move from register to stack
            s.storeParameterRegsToStack(s.f.ABISelf,
paramAssignment, n, s.decladdrs[n], false)
        }
    }
}

```

For each parameter, the compiler asks: "Can this be used directly as an SSA value?" If the parameter is simple and remains local, it may qualify. These are the requirements checked by the `canSSA(value)` and `canSSA(type)` functions. For instance:

- The value must be small enough to fit in a register or scalar SSA value (for example, `int`, `float64`, or pointers). Usually, it must be 4 or fewer pointer-sized values.
- Its address is never taken (no usage of `&param`).
- It does not escape to the heap.
- It is not involved in special handling cases like deferred calls.

If the answer is yes, the compiler creates an `Arg` SSA value for it. If not, it uses memory instead.

Now the compiler generates SSA blocks and values for the function body:

```

s.stmtList(fn.Body)
s.insertPhis()

```

*We won't go into detail here because this step resembles earlier phases where code is broken down from statements into expressions.*

We can see the result of this phase using the `GOSSAFUNC` environment variable. It will show the function's SSA form just before and after phi nodes are inserted. This is the first clear view of the transition from IR nodes into SSA blocks and values:

sources	AST	before insert phis	start
<pre> 3   /Users/aiden/Sources/me/golab/main.go 4   func anything(x int) int { 5       if x &gt; 0 { 6           x = 1 7       } else { 8           x = 2 9       } 10      return x 11  } </pre>		<pre> b1: v1 (?) = InitMem &lt;mem&gt; v2 (?) = SP &lt;uintptr&gt; v3 (?) = SB &lt;uintptr&gt; v4 (?) = LocalAddr &lt;=int&gt; {x} v2 v1 v5 (?) = LocalAddr &lt;=int&gt; {~r0} v2 v1 v6 (3) = Arg &lt;int&gt; {x} (x[int]) v7 (?) = Const64 &lt;int&gt; [0] v8 (4) = Less64 &lt;bool&gt; v7 v6 v9 (?) = Const64 &lt;int&gt; [1] (x[int]) v10 (?) = Const64 &lt;int&gt; [2] (x[int]) If v8 ~ b3 b4 (4)  b2: ~ b3 b4 v11 (4#) = FwdRef &lt;int&gt; {{!} x} (x[int]) v12 (4#) = FwdRef &lt;mem&gt; {{!} mem} v13 (10) = MakeResult &lt;int,mem&gt; v11 v12 Ret v13 (~10)  b3: ~ b1 Plain ~ b2 (3#)  b4: ~ b1 Plain ~ b2 (3#)  name x[int]: v6 v9 v10 v11 </pre>	<pre> b1: v1 (?) = InitMem &lt;mem&gt; v2 (?) = SP &lt;uintptr&gt; v3 (?) = SB &lt;uintptr&gt; v4 (?) = LocalAddr &lt;=int&gt; {x} v2 v1 v5 (?) = LocalAddr &lt;=int&gt; {~r0} v2 v1 v6 (3) = Arg &lt;int&gt; {x} (x[int]) v7 (?) = Const64 &lt;int&gt; [0] v8 (4) = Less64 &lt;bool&gt; v7 v6 v9 (?) = Const64 &lt;int&gt; [1] (x[int]) v10 (?) = Const64 &lt;int&gt; [2] (x[int]) If v8 ~ b3 b4 (4)  b2: ~ b3 b4 v11 (4#) = Phi &lt;int&gt; v9 v10 (x[int]) v12 (4#) = Copy &lt;mem&gt; v1 v13 (10) = MakeResult &lt;int,mem&gt; v11 v12 Ret v13 (~10)  b3: ~ b1 Plain ~ b2 (3#)  b4: ~ b1 Plain ~ b2 (3#)  name x[int]: v6 v9 v10 v11 </pre>

Illustration 141. If-else logic converted to SSA structure

## SSA Passes

When the compiler first converts IR to SSA, it creates a conservative representation that preserves all the safety guarantees and behaviors of the original source code. This includes redundant operations, unoptimized control flow, etc.

SSA breaks down a program into simple, atomic operations with clear data dependencies. This structure allows the compiler to analyze and transform the code more effectively. The result is faster, more efficient programs that use less memory and run better overall. These improvements are made through a series of steps called "passes."

This is the main entry point for SSA compilation:

```
// Main call to ssa package to compile function
ssa.Compile(s.f)
```

A pass is like a filter that scans SSA-form code and applies targeted improvements. Each pass focuses on a specific aspect of the code and is applied one at a time, function by function. But how many passes are there?

If we check the Go source code, we can find over 50 different passes. They are too many to list in full here, but below are some examples:

Figure 106. SSA Passes (src/cmd/compile/internal/ssa/compile.go)

```
// list of passes for the compiler
var passes = [...]pass{
    {name: "opt", fn: opt, required: true},
    {name: "sccp", fn: sccp},
    {name: "writebarrier", fn: writebarrier, required: true},
```

```

    {name: "lower", fn: lower, required: true},
    {name: "cse", fn: cse},
    {name: "schedule", fn: schedule, required: true},
    {name: "regalloc", fn: regalloc, required: true},
    // ...
}

```

Each pass includes a name, a function that implements the pass, and optional flags. The `required` flag means the pass must always run, even when optimizations are disabled with the `-N` flag. We will split the passes into four main phases: Optimization, Lowering, Layout and Register Allocation.

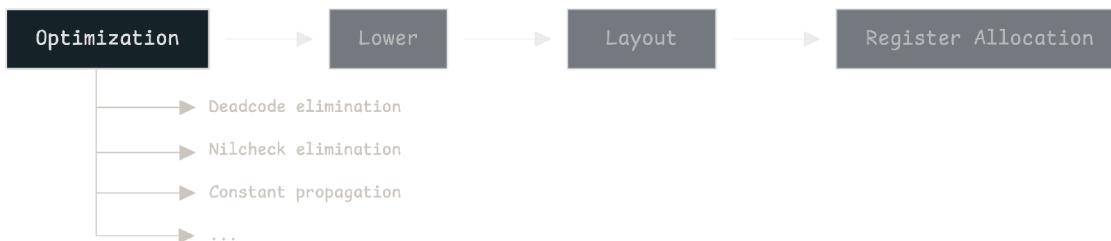


Illustration 142. Optimization begins the SSA pass pipeline

The optimization phase looks for redundant operations, simplifies expressions, removes unnecessary safety checks (when proven safe), and restructures code to reduce extra computation. At this stage, the SSA form is still machine-independent. This means it is written without considering the details of any specific hardware, whether it's ARM, x86, or any other architecture.

To run efficiently on specific hardware, the program must be adapted to the constraints and features of that target architecture. This is the role of the `lower` pass. It converts the machine-independent SSA form into a machine-dependent one:

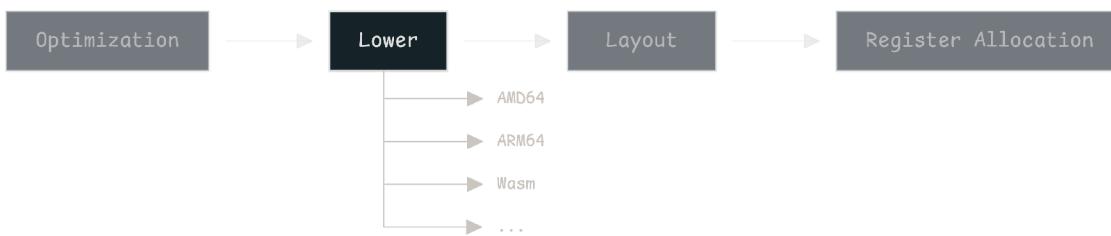


Illustration 143. Lowering adapts SSA to target architecture

During this lowering step, abstract operations are translated into concrete, hardware-specific instructions. For example, a general addition might be replaced with an instruction that directly matches the processor's instruction set.

This step can increase or decrease the number of values. Some high-level operations are broken down into simpler machine instructions, while others are merged into a single, more efficient instruction. The result depends heavily on the target architecture.

Next comes the layout phase, which looks at dependencies between instructions (for example, when one operation needs the result of another) and reorders them to improve performance:

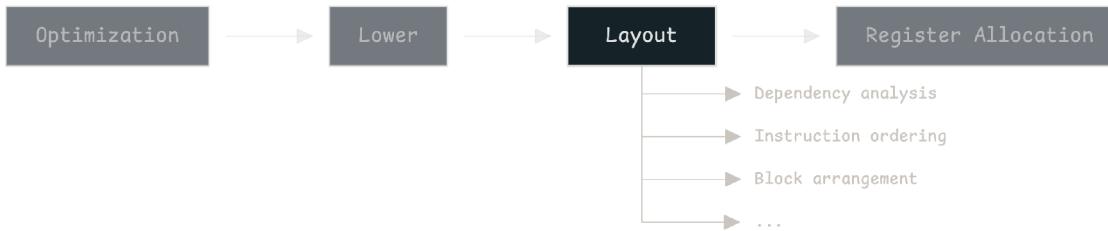


Illustration 144. Layout phase arranges blocks and instructions

It decides the exact order in which instructions are executed within each basic block. This has a significant impact on the final Go assembly code.

The final stage is register allocation. It assigns SSA values to real machine registers:



Illustration 145. SSA values mapped to CPU registers

The Go compiler uses a linear scan algorithm. It walks through the program and makes decisions on where to store each value as it goes. It tracks which registers are available, when values will be used again, and inserts memory load or store instructions (called spilling) when there are not enough registers to hold everything.

In the next section, we will focus on the parts that matter most and are worth understanding.

## Optimization Phase

### Rewrite Rules

Most optimization passes are written directly in Go code, but some use a different form called rewrite rules.

Rewrite rules are patterns that describe how specific code expressions should be transformed. These rules have their own custom syntax and are defined in files located in the `cmd/compile/internal/ssa/_gen` directory. Some examples include:

- `ssa/_gen/dec.rules` (over 50 rules)
- `ssa/_gen/dec64.rules` (over 107 rules)
- `ssa/_gen/generic.rules` (over 950 rules)

These rules help the compiler detect certain patterns in code and replace them with faster or simpler alternatives.

The syntax for these rules uses s-expressions. These are parenthesized expressions, similar to those found in Lisp. Each rule consists of three main parts: a match pattern, optional conditions, and a replacement pattern:

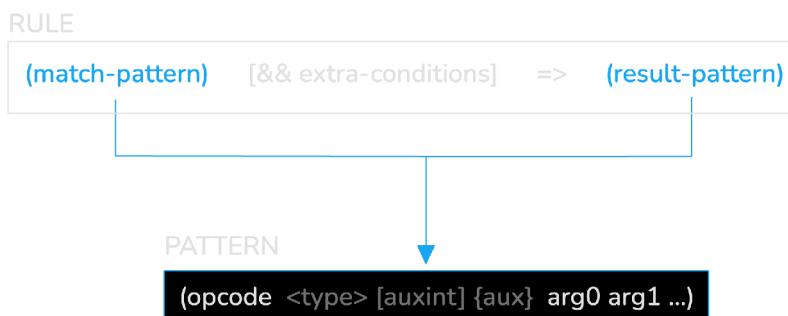


Illustration 146. Pattern-matching rules simplify expressions

You can think of it as saying: "If you see this pattern (`match-pattern`) in the code, replace it with this pattern (`replacement-pattern`)."

Before going deeper into the syntax, here is a simple example:

```
(Mul(8|16|32|64) (Const(8|16|32|64) [1]) x) => x
```

This rule tells the compiler that any multiplication by `1` can be safely removed. Here's how it works:

1. `Mul(8|16|32|64)`: matches any multiplication operation like `Mul8`, `Mul16`, `Mul32`, or `Mul64`. These represent SSA instructions for multiplying values of different bit widths.

2. `(Const(8|16|32|64) [1])` : matches a constant with the value `1` and the same bit width.
3. `x` : represents the other operand in the multiplication.
4. `=> x` : tells the compiler to replace the entire expression with just `x`.

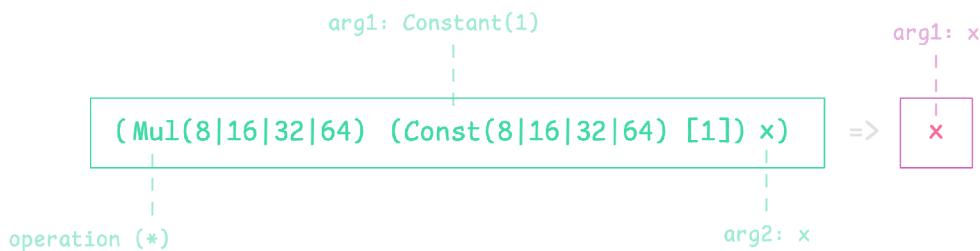
So, for example, the expression `y := x * 1` will be transformed into `y := x`.

If you're thinking about contributing to the Go compiler, working on rewrite rules is a great place to start. But before that, it helps to fully understand how these rules are structured. A complete rule pattern can include many parts, which we will explore next.

Here is the cheatsheet, ordered from most important to least important:

1. **Operation code:** The operation names like `Add64`, `Mul32`, etc. These specify what operation to match or replace.
2. **Variables:** Identifiers such as `x` or `y` that bind to sub-expressions inside the match pattern.
3. **Constants:** Represented using operations like `Const64`, `Const32`, etc., with values in square brackets. For example, `Const64 [1]` is a 64-bit constant with value 1.
4. **Pattern alternatives:** Use the pipe `|` symbol to match multiple similar operations, such as `(Add(8|16|32|64))` to match `Add8`, `Add16`, `Add32`, and `Add64`.
5. **Extra conditions:** Boolean checks added with `&&` to further restrict when a rule applies, like `isPowerOfTwo64(c)` or `d != 0`.
6. **Type annotations:** Written in angle brackets like `<t>` or `<typ.Uint64>` to define or constrain the type of an operation or result.
7. **Auxiliary data:** Extra metadata written in curly braces, such as `{aux}`, that carries additional context for an operation.
8. **Wildcards:** Use `_` to match any value when you do not need to refer to it later.

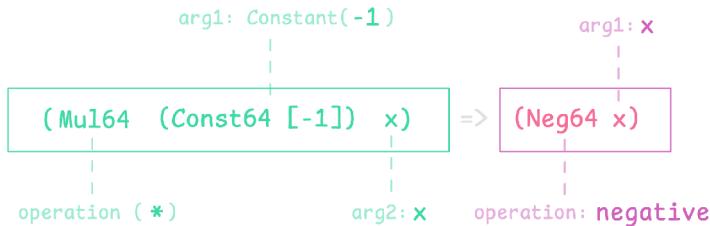
So our previous rule can be broken down as:



### Illustration 147. Pattern match for multiplication by one

Now let's try another example. Earlier, we simplified multiplication by 1. This time, we'll simplify multiplication by -1, which can be replaced with a negation operation:

```
(Mul64 (Const64 [-1]) x) => (Neg64 x)
```



### Illustration 148. Pattern match for multiplication by negative one

Breaking down the rule:

- `Mul64` is the outer operation, representing 64-bit multiplication.
- The first argument is `Const64 [-1]`, which matches a constant with value -1.
- The second argument is `x`, a variable that matches any operand.
- The replacement is `(Neg64 x)`, which applies the 64-bit negation operation.

This rule tells the compiler that any multiplication by -1 should be replaced with a negation.

Before	After
<code>return x * -1</code>	<code>return -x</code>

This rule works for both `-1 * x` and `x * -1` because multiplication is commutative. The metadata for the `Mul64` operation includes the `commutative` flag, which allows the rule to apply regardless of operand order:

Figure 107. Opcodes Mul64 Metadata  
(src/cmd/compile/internal/ssa/opGen.go)

```
{
  name:      "Mul64",
  argLen:    2,
  commutative: true,
  generic:   true,
```

```
},
```

The previous rule only handles 64-bit multiplication. To support other bit widths, we can use pattern alternatives to match all relevant cases in a single rule. This works the same way as in the earlier example:

```
(Mul(8|16|32|64) (Const(8|16|32|64) [-1]) x) =>
(Neg(8|16|32|64) x)
```

So how are these rules used in Go code, and why not just write Go directly?

Behind the scenes, these rules are used to automatically generate Go code. The generation logic is located in `_gen/main.go` and `_gen/rulegen.go`. These files parse the rule definitions, analyze the patterns, and generate the necessary Go compiler code to implement the matching and transformations.

When using pattern alternatives like `Mul(8|16|32|64)`, the generator expands them into separate rules in the output Go code, one for each bit width:

```
// match: (Mul8 (Const8 [-1]) x)
// result: (Neg8 x)
for {
    for _i0 := 0; _i0 <= 1; _i0, v_0, v_1 = _i0+1, v_1,
v_0 {
        if v_0.Op != OpConst8 ||
auxIntToInt8(v_0.AuxInt) != -1 {
            continue
        }
        x := v_1
        v.reset(OpNeg8)
        v.AddArg(x)
        return true
    }
    break
}

// match: (Mul16 (Const16 [-1]) x)
// result: (Neg16 x)
for {
    for _i0 := 0; _i0 <= 1; _i0, v_0, v_1 = _i0+1, v_1,
v_0 {
        if v_0.Op != OpConst16 ||
auxIntToInt16(v_0.AuxInt) != -1 {
            continue
        }
        x := v_1
        v.reset(OpNeg16)
        v.AddArg(x)
```

```
        return true
    }
break
}
```

The main reason the Go team uses rule files is to allow developers to define optimizations through clear pattern-matching transformations. This approach lets contributors focus on what the code should look like before and after the transformation, without having to implement the logic manually in Go.

*"While most compiler passes are implemented directly in Go code, some others are code generated. This is currently done via rewrite rules, which have their own syntax and are maintained in `_gen/*.rules`. Simpler optimizations can be written easily and quickly this way, but rewrite rules are not suitable for more complex optimizations. Similarly, the code to manage operators is also code generated from `_gen/*Ops.go`, as it is easier to maintain a few tables than a lot of code. After changing the rules or operators, run `go generate cmd/compile/internal/ssa` to generate the Go code again."*

## ~ src/cmd/compile/internal/ssa/README.md

We're going to look at another rule that is a bit more complex:

```
(Mul64 <t> n (Const64 [c])) && isPowerOfTwo64(c)
=>
(Lsh64x64 <t> n (Const64 <typ.UInt64> [log64(c)]))
```

This rule matches a multiplication operation `Mul64` between a variable `n` and a constant `c`, but only if `c` is a power of two:

Go	Pseudo SSA
<code>m * 8</code>	<pre>v1: n                      // Original variable v2: (Const64 [8])           // Constant value 8 v3: (Mul64 &lt;int64&gt; v1 v2)  // Multiplication operation</pre>

The multiplication `m * 8` matches the pattern. But the rule operates at the SSA level, so what matters is how the SSA tree looks. The rule pattern checks for the multiplication node `v3` and matches its two arguments.

The functions `isPowerOfTwo64` and `log64` used in the rule already exist in the Go compiler code. They help implement this transformation by detecting when the constant is a power of 2 and calculating its base-2 logarithm:



Illustration 149. Match Mul64 with power-of-two constant

This optimization relies on the mathematical principle that multiplying by a power of two is equivalent to a left bit shift by the exponent of that power. The SSA rule implements this by replacing the multiplication node with a left shift node:



Illustration 150. Multiply replaced with left shift

Let's see how this transformation looks in terms of Go code versus the resulting SSA:

Go	Pseudo SSA
<pre>b &lt;&lt; 3</pre>	<pre>v1: n           // Original variable v2: (Const64 [8]) // Original constant (now unused) v4: (Const64 &lt;typ.UInt64&gt; [3]) // New shift amount constant v3: (Lsh64x64 &lt;int64&gt; v1 v4)    // Transformed operation (Left Shift)</pre>

Effectively, this optimization transforms the original Go source code like this:

Original Go	Optimized Go Equivalent
<pre>b * 8</pre>	<pre>// const redundant = 8 // Original constant becomes unused</pre>

## Original Go

```
b << 3
```

## Optimized Go Equivalent

After this transformation, the original constant `8` (represented by `v2` in SSA) is no longer used. It will likely be removed later by another optimization pass, such as dead code elimination.

Before we move on, one more important rule to mention is **constant folding**. Constant folding is the process of computing constant expressions at compile time instead of generating runtime instructions for them. The `generic.rules` file includes many such rules that simplify expressions involving constants by evaluating them early.

Take some rules in `generic.rules` as example:

```
(Add8   (Const8 [c]) (Const8 [d])) => (Const8 [c+d])
(Mul32  (Const32 [c]) (Const32 [d])) => (Const32 [c*d])
(Sub64  (Const64 [c]) (Const64 [d])) => (Const64 [c-d])
(Div16u (Const16 [c]) (Const16 [d])) && d != 0 => (Const16
[int16(uint16(c)/uint16(d))])
```

These rewrite rules say: if you see an addition, multiplication, subtraction, or division where both operands are constant values, replace the entire operation with a single constant result:

### Before

```
var global = 10

func folded() int {
    a := 10
    b := 5

    x := b + global
    if global > 8 {
        return x
    }
    return a - b +
    global
}
```

### After

```
var global = 10

func folded() int {
    a := 10
    b := 5

    x := b + global
    if global > 8 {
        return x
    }
    c := 5
    return c + global
}
```

Both `a` and `b` are constant values in the case and can be folded in the last return statements (`return a - b + global`). The `global` does not belong to the

function scope, so Go compiler doesn't know the value of `global` at compile time, it can't be folded for `x`.

Now, the variable `a` becomes useless and can be eliminated by deadcode elimination.

When we say "Both `a` and `b` are constant values," we don't mean that the `a` and `b` in the Go code are constants—they're clearly variables. What we mean is that in the SSA representation, the values are constant. Once a value is set in SSA, it doesn't change.

### Deadcode Elimination

The deadcode elimination pass removes code that has no effect on the program's behavior. In the current version of the Go compiler, this optimization runs multiple times during compilation. In fact, we can see it applied in at least eight different places:

```
{name: "early deadcode", fn: deadcode},  
{name: "pre-opt deadcode", fn: deadcode},  
{name: "opt deadcode", fn: deadcode, required: true},  
{name: "gcse deadcode", fn: deadcode, required: true},  
{name: "generic deadcode", fn: deadcode, required: true},  
{name: "lowered deadcode", fn: deadcode, required: true},  
{name: "late deadcode", fn: deadcode},
```

This repetition is intentional and necessary. Many optimization passes can introduce new opportunities for dead code elimination.

For example, rewrite rules involving constant propagation may simplify expressions or resolve conditions at compile time. This can result in entire branches or blocks becoming unreachable:

```
func example() {  
    x := 10  
    y := 20  
    if x + y > 100 {  
        fmt.Println("Always false")  
    }  
}
```

Thanks to two constant propagation rewrite rules, the condition is gradually simplified until the code becomes unreachable and can be removed.

The first rule is constant folding for addition:

```
(Add64 (Const64 [c]) (Const64 [d]))  
=>  
(Const64 [c+d])
```

Before	After
<pre>if x + y &gt; 100 {     fmt.Println("Always     false") }</pre>	<pre>if 30 &gt; 100 {     fmt.Println("Always     false") }</pre>

Next, the compiler applies constant folding for comparisons:

```
(Less(64|32|16|8) (Const(64|32|16|8) [c]) (Const(64|32|16|8)  
[d]))  
=>  
(ConstBool [c < d])
```

Before	After
<pre>if 30 &gt; 100 {     fmt.Println("Always     false") }</pre>	<pre>if false {     fmt.Println("Always     false") }</pre>

Finally, deadcode elimination traverses the control flow graph of the program. It removes instructions and blocks that no longer affect the program's observable behavior. In this case, the entire `if` statement and its contents are eliminated, along with the now-unused assignments to `x` and `y`.

You might ask, why doesn't early deadcode elimination during the IR generation phase remove the `if` block?

At the IR level, the compiler performs only limited static analysis. It can eliminate branches based on known **compile-time constants**, but it does not analyze variable values. It also does not perform data-flow analysis to check whether `x` or `y` might be modified before reaching the `if` statement.

In contrast, SSA provides full visibility into all control paths and tracks how values flow through the program. The compiler can see that `x` and `y` are never modified and that `x + y` always equals `30`.

### Common Subexpression Elimination (CSE)

Common subexpression elimination (CSE) detects repeated computations in a program and removes redundant ones. If an expression `E` is calculated once, and its operands have not changed when it appears again, the compiler can reuse the previously computed value instead of recalculating it:

Before	After
<pre>x := a + b if condition {     return x } return a + b + c</pre>	<pre>x := a + b if condition {     return x } return x + c</pre>

### Comparison & Bounds Check Elimination

Comparison check elimination removes redundant or unreachable conditional checks, especially bounds checks that are proven unnecessary using known conditions.

The Go compiler keeps track of facts it learns about variables throughout the control flow. For example, if the code contains `if i < len(a)` and enters the true branch, the compiler knows from that point on that `i` is definitely less than `len(a)`.

As the compiler moves through different blocks of the program, it updates its knowledge based on conditions it encounters. If it later reaches a condition like `i >= len(a)` within the same control path, it detects a contradiction. Since `i` was already known to be less than `len(a)`, it marks the new condition as unreachable. The unreachable code is then removed by deadcode elimination.

A common case of this optimization is bounds check elimination. In Go, every index access is guarded by an automatic bounds check:

Before	After
a[100] = 100	<pre>if 100 &lt; len(a) {     runtime.panicindex() } a[100] = 100</pre>

However, if the compiler knows that `len(a)` is always greater than `100`, it can safely remove the bounds check. Let's look at a real-world example to see how this affects performance:

No Bound Checks	With Bound Checks
<pre>func WithoutBoundsChecks(a []int, b []int) int {     if len(a) != len(b) {         panic("lengths must be equal")     }     sum := 0     for i := range a {         sum += a[i] + b[i]     }     return sum }</pre>	<pre>func WithBoundsChecks(a []int, b []int) int {     sum := 0     for i := range a {         sum += a[i] + b[i]     }     return sum }</pre>

The key difference between these two functions is why the compiler can eliminate bounds checks in `for` loop of `WithoutBoundsChecks` but not in `WithBoundsChecks` for loop. Let's break down the `WithoutBoundsChecks` function:

- Fact 1:** Since the function starts with an explicit check `len(a) == len(b)`, the compiler understands that both slices have the same length.
- Fact 2:** The loop uses `for i := range a`, which guarantees that `i` is always a valid index for slice `a`.
- Given facts 1 and 2, the compiler can also conclude that `i` is valid for slice `b`. This means both `a[i]` and `b[i]` are always safe, so no bounds checks are needed at runtime.

In the `WithBoundsChecks` version, although `i` is known to be valid for `a` (because of the `range` loop), the compiler cannot be sure about `b`. If `len(b)` is

shorter than `len(a)`, the access to `b[i]` might go out of bounds. To be safe, the compiler inserts a bounds check to ensure `b[i]` will not panic at runtime.

This results in a small performance penalty:

go test -bench=. -benchmem -count=5 -cpu=1 -benchtime=3s		
goos	darwin	
goarch	arm64	
pkg	theanatomyofgo	
<hr/>		
BenchmarkWithBoundsChecks	12068	304553
ns/op		
BenchmarkWithBoundsChecks	12104	298686
ns/op		
BenchmarkWithBoundsChecks	10000	300825
ns/op		
BenchmarkWithBoundsChecks	12192	297409
ns/op		
BenchmarkWithBoundsChecks	12175	300311
ns/op		
<hr/>		
BenchmarkWithoutBoundsChecks	12846	278881
ns/op		
BenchmarkWithoutBoundsChecks	12753	280631
ns/op		
BenchmarkWithoutBoundsChecks	12698	278222
ns/op		
BenchmarkWithoutBoundsChecks	12787	279251
ns/op		
BenchmarkWithoutBoundsChecks	12844	279317
ns/op		

In this benchmark (with `len(a) == len(b) == 1_000_000`), the version without bounds checks (`WithoutBoundsChecks`) performs about 7% faster on average compared to the version with bounds checks (`WithBoundsChecks`). This shows that validating slice lengths early gives the compiler the confidence it needs to eliminate bounds checks, which improves performance even though both versions do the same work.

For those interested in debugging or optimizing the bounds check elimination pass, the Go compiler offers a helpful flag. You can enable it with the following command:

```
go build -gcflags=-d=ssa/check_bce/debug
```

Running the build with this flag provides detailed output, indicating specifically where the compiler inserts or removes bounds checks during compilation.

## Nilcheck Elimination

Nilcheck elimination is similar to bounds check elimination, but applies to pointer dereferences. The Go compiler inserts a nil check before every pointer dereference:

```
if a == nil {  
    runtime.sigpanic()  
}
```

If the compiler can prove that a pointer is never `nil`, it can safely remove the check.

On most modern hardware, nil pointer access results in a hardware fault when trying to read memory at address `0`. This fault is caught by the Go runtime and turned into a panic. Because of this, the compiler can skip the nil check entirely and rely on the hardware to catch errors during execution.

A particularly useful tool for exploring Go's SSA internals is the command `go tool compile -d ssa/help`. This command displays a comprehensive list of all SSA (Static Single Assignment) debug flags supported by the compiler. This list effectively serves as a reference manual for the various `-d=ssa/...` debugging options available for optimization and analysis.

## Lower Passes

Lowering is the process of converting generic SSA operations into architecture-specific operations. SSA includes generic operations like `Add32`, `Add64`, `Load`, `Store`, `Call`, `SelectN`, and others. These operations are not tied to any specific CPU instruction set. You can think of them as a form of portable IR (intermediate representation). Before generating machine code, the compiler must translate them into CPU-specific instructions.

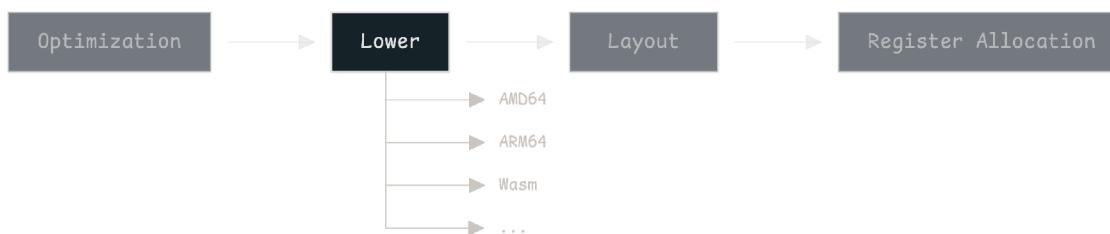


Illustration 151. Lowering adapts SSA to target architecture

The Go compiler again uses rewrite rules to perform this lowering. These rules are defined in architecture-specific files such as `ARM64.rules`, `AMD64.rules`, and so

on.

For example, on ARM64, the compiler might apply the following transformations:

```
(Add64 x y) => (ADD x y)  
(Mul64 x y) => (MUL x y)  
(And64 x y) => (AND x y)
```

Here, `ADD`, `MUL`, and `AND` are pseudo-operations that map directly to real ARM64 instructions.

In some cases, the lowering is a simple 1-to-1 mapping. But often, it includes small optimizations. For instance:

```
(Add64 x (Const64 [5000])) => (ADDconst [5000] x)
```

This transformation generates an ARM64 pseudo-instruction that uses an immediate constant. It avoids the need to load `5000` into a register before adding. There is no actual `Addconst` instruction in ARM64. Instead, `ADDconst` acts as an intermediate step. A later lowering pass will decide whether the constant can be used directly or must be moved into a register.

Not all constants can be used as immediates in ARM64 `ADD` instructions. ARM64 has specific rules about which immediate values can be encoded directly. If the constant does not meet those rules, it will be lowered into a small sequence of instructions to load the value into a register before performing the addition.

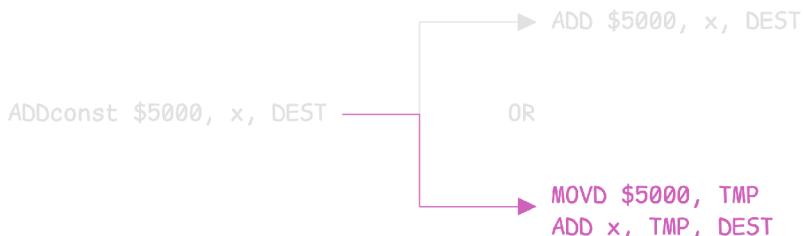


Illustration 152. ARM64 constant add instruction decision

## Layout & Schedule Passes

The layout pass determines the order in which basic blocks of a function are arranged in memory during assembly generation. A basic block is a straight-line sequence of instructions with no jumps or branches, except possibly at the end.

Why does this order matter? Because the CPU executes instructions sequentially. If two blocks that logically follow each other in the program are placed next to each other in memory, the CPU can continue executing without needing to jump. This makes execution faster by reducing control flow jumps and allowing natural 'fall-through' between blocks.

So the question becomes: in what order should we place these blocks in memory during code generation?

Let's consider the following example:

```
func f(x int) int {
    if x != 0 {
        return x
    }

    panic("BUG: x is 0")
}
```

When the Go compiler builds SSA from IR nodes, it creates three blocks: the entry block, a return block, and a panic block. There's no explicit `else` block in this example, but logically, the panic block acts as the fallback case.

The resulting SSA structure looks something like this:

```
b1: -
    v1 (?) = InitMem <mem>
    v6 (14) = ArgIntReg <int> {x+0} [0] (x[int])
NZ v6 → b3 b2 (likely) (+14)

b2: ← b1-
    v3 (?) = SB <uintptr>
    v12 (?) = MOVDaddr <*uint8> {type:string} v3
    v13 (?) = MOVDaddr <*string> {main..stmp_1} v3
    v16 (+18) = CALLstatic <mem>
{AuxCall{runtime.gopanic}} [16] v12 v13 v1
    v17 (18) = SelectN <mem> [0] v16
Exit v17 (18)

b3: ← b1-
    v11 (+15) = MakeResult <int,mem> v6 v1
    Ret v11 (+15)

name x[int]: v6
```

In this SSA:

- Block `b1` is the entry block.
- If `v6 != 0`, control jumps to `b3` (the return block).
- Otherwise, it jumps to `b2` (the panic block).

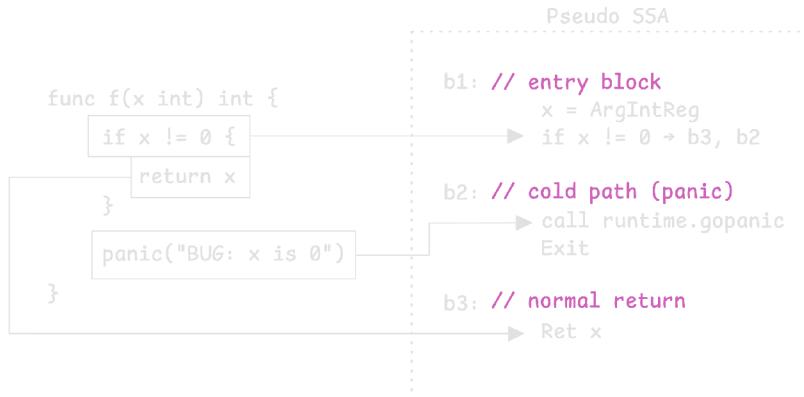


Illustration 153. SSA structure for conditional panic function

If the layout pass places the panic block `b2` immediately after the entry block `b1`, the CPU will have to jump to reach the return block `b3` when executing the common case.

But from experience, we know that the panic path is rarely taken. Most of the time, `x` is not zero and the function simply returns `x`. If this function is executed 100 times and `x` is always non-zero, the CPU would be forced to jump 100 times just to reach the main logic in `b3`.

To avoid this, the layout pass tries to reorder blocks so that likely paths are placed contiguously in memory. That way, the CPU can follow the common case without jumping, improving performance.

## Why Sequential Code Flow Matters

Modern CPUs are optimized for sequential instruction execution due to how their instruction cache (I-cache) and hardware prefetching work:

- **Cache Line Fetching:** CPUs load instructions into the I-cache in fixed-size chunks called cache lines (discussed in Chapter 4). When code is laid out sequentially, the next instructions are often already loaded, making execution more efficient.
- **Prefetching Advantage:** Hardware prefetchers predict upcoming instruction addresses based on sequential patterns. With straight-line execution

(fallthrough), the prefetcher stays ahead and keeps instructions ready in the cache.

- **Jump Disruption:** A jump changes the instruction flow, redirecting the CPU to a new address. If the new location is not in the I-cache, this causes a stall. It also disrupts the prefetcher, which must discard previously fetched lines and restart from the new location.

Arranging code so that the most common execution path flows directly from one block to the next (like `b3` following `b1` in the Go example) works with the CPU's natural optimization strategies. This improves I-cache hit rates and keeps the prefetcher effective, helping performance even when branch prediction is not perfect.

The Go compiler needs to know which paths are taken more frequently. This is known as **branch probability**. There are two main ways compilers obtain this information:

- **Guesses:** Default assumptions based on common patterns.
- **Profile-guided optimization (PGO):** Real profiling data gathered by running the program.

*PGO will be discussed in the next chapter. For now, let's focus on the first method.*

In the example above, the compiler assumes that paths leading to `panic` are **unlikely**. Based on this assumption, it reorders the blocks so that the panic block appears later in memory. This avoids an unnecessary jump in the common case.

That's why you often see `panic` instructions placed at the end of the function in assembly output:

```
00000 TEXT main.f(SB), ABIInternal
00006 CBZ R0, 8
00007 RET
00008 MOVD $type:string(SB), R0
00009 MOVD $main..stmp_0(SB), R1
00011 CALL runtime.gopanic(SB)
00012 HINT $0
00013 END
```

The `CBZ` (Compare and Branch on Zero) instruction checks whether register `R0` (which holds `x`) is zero:

- If so, it jumps to offset `8`, which leads to the panic logic.

- If not, it falls through to `RET`, returning normally.

We can see here that the panic-related code is laid out at the end of the function to favor the more common return path.

Now, let's look at some patterns the Go compiler can recognize to identify **less likely branches**. These blocks are usually placed at the end of the function to improve performance by favoring fallthrough in common cases.

- **Exiting a loop:** The `break` branch is typically less likely:

```
for i := 0; i < 10; i++ {
    if someCondition() {
        break
    }
    // Body is more likely
}
```

- **Function calls inside branches:** The branch with the function call is less likely:

```
if condition {
    doSomething()
} else {
    // This branch without a call is more likely
}
```

- **Error handling:** The error return path is assumed to be less likely:

```
result, err := someOperation()
if err != nil {
    return nil, err
}
```

- **Defer usage:** Branches that include `defer` are treated as less likely:

```
if someCondition() {
    defer cleanup()
}
```

- **Panic or exit paths:** These are always treated as very unlikely. We have already seen this in previous examples.

Once block reordering is complete, the **schedule pass** runs next. It reorders the instructions (values) inside each block. The goal is to prioritize operations based on how critical they are to the control and data flow. The scheduler uses a scoring system to decide how to order instructions:

```
const (
    ScorePhi    = iota // towards top of block
    ScoreArg        // must occur at the top of the
entry block
    ScoreInitMem   // after the args - used as mark by
debug info generation
    ScoreReadTuple // must occur immediately after
tuple-generating insn (or call)
    ScoreNilCheck
    ScoreMemory
    ScoreReadFlags
    ScoreDefault
    ScoreFlags
    ScoreControl   // towards bottom of block
)
```

Over time, the Go team has tuned this scoring system based on real-world experience, learning how different instruction orderings affect program performance and runtime behavior. The source code also contains detailed comments explaining the rationale behind different scores:

- `ScoreArg` : "We want all the args as early as possible, for better debugging"
- `ScoreMemory` : "Schedule stores as early as possible. This tends to reduce register pressure"
- `ScoreReadFlags` : "Schedule flag-reading ops earlier, to minimize the lifetime of flag values"
- `ScoreFlags` : "Schedule flag register generation as late as possible. This makes sure that we only have one live flags value at a time"

## Register Allocation Passes

Register allocation assigns program variables to a limited number of physical CPU registers. In SSA form, the compiler works with an unlimited number of virtual registers. The register allocator is responsible for mapping these virtual registers to a fixed set of physical ones supported by the target architecture.

In SSA, the compiler behaves as if it has infinite registers. Every operation stores its result in a brand new virtual register. There's no concern yet for how many physical registers are actually available on the CPU.

To understand this, let's revisit a simple Go function:

```
func add(x, y int) int {
    a := x + y
    b := a + 1
    c := b * 2
    return c
}
```

When this is converted into SSA form, every assignment creates a new value. There are no variable updates—only new versions:

```
v0 = x
v1 = y
v2 = add v0, v1      // a
v3 = add v2, 1        // b
v4 = mul v3, 2        // c
return v4
```

Each of `v0`, `v1`, `v2`, `v3`, and `v4` is a unique virtual register. These values are not stored in memory. Notice there are no `Load` or `Store` operations here. Instead of reusing variable names, the compiler creates a new identifier for each computed value.

However, physical CPUs only offer a limited number of real registers. The register allocator's job is to decide which values can stay in registers and which ones must be temporarily stored in memory.

First, the compiler needs to know what registers are available on the target architecture. On ARM64, the Go compiler works with 64 registers in total:

- 32 general-purpose integer registers ( `R0 - R31` )
- `R0 - R30` : for holding integer values and addresses
- `R31` : reserved as the stack pointer ( `SP` )
- 32 floating-point registers ( `F0 - F31` ): Used for IEEE 754 floating-point arithmetic (sign, exponent, and significand)

Not all of these registers are available for allocation. Some are reserved or have special roles:

- `R18` : reserved as a platform register.
- `R27` ( `REGTMP` ): used by the assembler as a temporary.
- `R29` : used as the frame pointer, not allocated by the compiler.
- `SP`, `SB`, and `g` : reserved for stack pointer, static base, and goroutine state.

Let's walk through an example to better understand how the register allocation process works:

```
func example(a int, b int) int {
    var y int

    if b > 0 {
        y = a * 2
    } else {
        y = b * 2
    }

    return y + b
}
```

We can translate this code into SSA form. For simplicity, we'll omit metadata and also skip the branch elimination optimization to preserve the full `if/else` structure in the SSA:

```
func example(a int, b int) int {
    var y int
    if b > 0 {
        y = a * 2
    } else {
        y = b * 2
    }
    return y + b
}

b1:
    v8 = ArgInt {b}          // b
    v7 = ArgInt {a}          // a
    v2 = CMPconst <fflags> [0] v8
    GT v2 → b3 b4 // b > 0 (flags)

b3: ← b1 // body (b > 0)
    v19 = SLLconst [1] v7 // y = a*2
    Plain → b2

b4: ← b1 // else (b <= 0)
    v5 = SLLconst [1] v8 // y = b*2
    Plain → b2

b2: ← b3 b4
    v16 = Phi v19 v5      // y
    v18 = ADD v16 v8      // temp = y+b
    v1 = InitMem <mem>
    v20 = MakeResult v18 v1 // return (temp,mem)
    Ret v20
```

Illustration 154. SSA branches for conditional variable assignment

Once SSA form is built, the next step is **liveness analysis**. The allocator needs to determine for each SSA value:

- Whether the value is **live** across basic block boundaries.
- How far ahead (in number of instructions) the value will be used again. This distance is important for deciding which values are better candidates for **spilling** later on.

Let's see how the compiler performs liveness analysis:

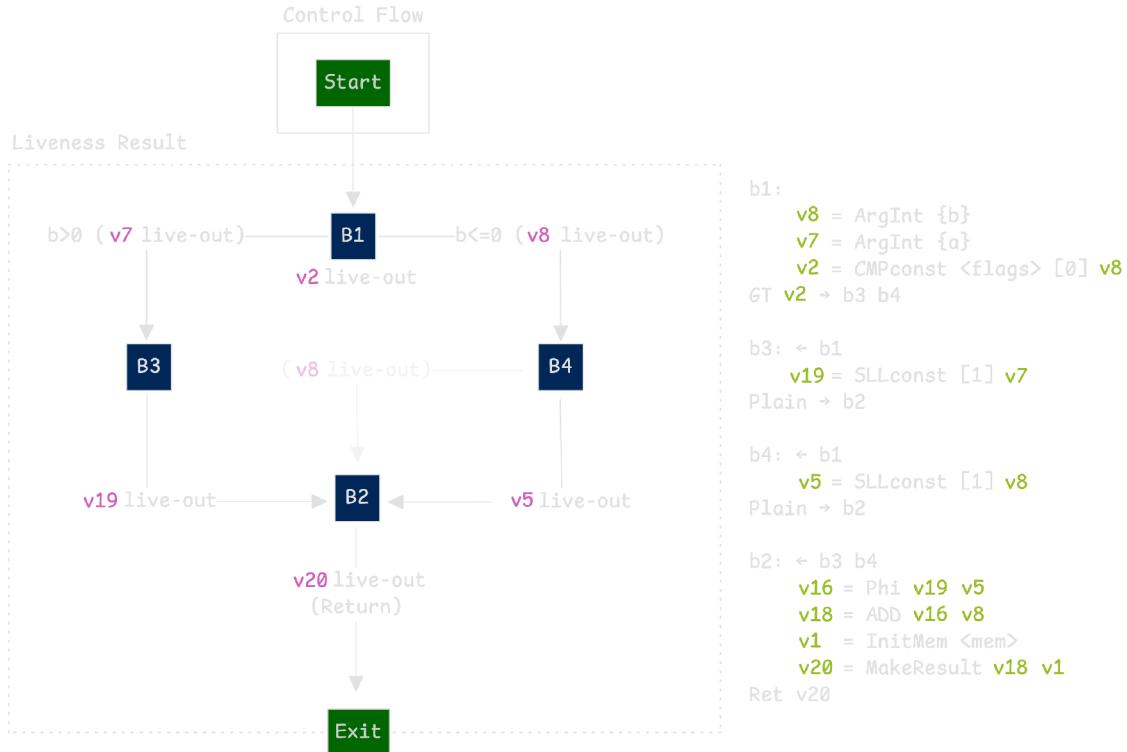


Illustration 155. Liveness analysis across SSA control flow

From the analysis:

- `v7` (which holds `a`) is defined in block `b1` and later used in block `b3`
- `v8` (which holds `b`) is defined in block `b1`, used in block `b4` (as the value for `y`), and also needed again in block `b2` for the final addition
- The results `v19` from `b3` and `v5` from `b4` are inputs to the Phi node `v16` in `b2`
- The final result `v20` is returned from the function

Although the analysis calculates the 'distance to next use' for each value, in our case, it does not make a big impact since there are plenty of available registers in ARM64 compared to the number of SSA values.

After this preparation, the allocator moves into block-by-block processing. Inside each block, the allocator runs the core of the **linear scan algorithm**. This algorithm proceeds greedily, walking through SSA values in order and making allocation decisions step-by-step:

1. The allocator visits each instruction in sequence. At each point, it knows:

- Which physical registers are free.
  - Which ones are currently holding live values.
2. For each instruction, the allocator examines input values first, then output value. For example, in `v3 = add v1, v2`, we have `v1` and `v2` as inputs, and `v3` as the output.
- If a value already lives in a register, that register is used.
  - If not, the allocator tries to find a free, compatible register and assigns it. It prefers the lowest-numbered available register (e.g. `R0`, `R1`, `R2`, ...).
  - If no registers are free, the allocator selects a register holding a value that won't be used soon (based on the liveness analysis). That value is **spilled** to memory to free the register. The needed value is then loaded or assigned into the newly freed register.
3. After processing the instruction (handling inputs and outputs), the allocator updates its internal state: recording which register now holds the output value and marking registers as free if they held input values that are now dead (no longer needed).

That's the general idea behind the linear scan and greedy allocation algorithm. Now, let's see how it works in practice using the earlier example.

The process begins with the entry block `b1`:

```
func example(a int, b int) int {
    var y int
    if b > 0 {
        y = a * 2
    } else {
        y = b * 2
    }
    return y + b
}

b1:
v8 = ArgInt {b}      // b
v7 = ArgInt {a}      // a
v2 = CMPconst <flags> [0] v8
GT v2 → b3 b4      // b > 0 (flags)
```

Illustration 156. The entry block `b1`

The allocator recognizes the incoming arguments `v7` (representing `a`) and `v8` (representing `b`). According to the architecture's Application Binary Interface (ABI), these arguments must be placed in specific physical registers when the function is called.

In this case, the allocator assigns:

- `v7 (a) → R0`

- `v8 ( b ) → R1`

```
b1:
R1 v8 = ArgInt {b}
R0 v7 = ArgInt {a}
v2 = CMPconst <flags> [0] v8
GT v2 → b3 b4
```

Illustration 157. ABI forces v7 to R0, v8 to R1

## Arguments & Return Values Are ABI-Mandated

It's important to understand that placing `a` in `R0` and `b` in `R1` is not a decision made by the allocator during scanning. These assignments are **required** by the ABI for the target architecture.

The ABI defines where arguments should be passed and where return values must be placed. For example, function arguments may always start in `R0`, `R1`, and so on, and return values may be expected in `R0`.

The register allocator must follow these rules. It does not have flexibility when assigning registers for incoming arguments or return values. These ABI requirements must be respected and directly influence how allocation works, especially at the function entry and exit.

Here's an interesting detail: constants like `0`, stored in values such as `v2` (used in comparisons), don't need their own registers. This is possible because most CPU architectures support **immediate mode**, which allows constant values to be embedded directly within the instruction.

Now let's look closer at `v2`, which is used in a comparison:

```
v2 = CMPconst v8, 0 <flags>
```

The key point is the `<flags>` type on the `CMPconst` instruction. In computer architecture, comparison operations usually do not store results in general-purpose registers. Instead, they update bits in a special processor register called the **flags register** (also known as the condition code register).

These flags include:

- **Zero flag (Z)**: set if the result is zero
- **Negative flag (N)**: set if the result is negative

- **Carry flag (C)**: set on unsigned overflow
- **Overflow flag (V)**: set on signed overflow

The subsequent branch instruction `GT` directly checks the flag bits in the CPU's condition code register to decide which path to take.

Next, the allocator processes block `b3`, which is executed if `b` is greater than `0`. It inherits the register state from block `b1`: `(R0 = v7, R1 = v8)`.

```
func example(a int, b int) int {
    var y int
    if b > 0 {
        y = a * 2
    } else {
        y = b * 2
    }
    return y + b
}

b1: (R0=v7, R1=v8)
b3: ← b1 // body (b > 0)
    v19 = SLLconst [1] v7 // y = a*2
Plain → b2
```

Illustration 158. v7 consumed; R0 now holds v19

The instruction `v19 = v7 * 2` uses `v7` from `R0`. Now it needs a register for `v19`. In theory, `R0` could be reused since `v7` is no longer needed after this point:

```
R0 (v19) = R0 (v7) * 2 // v19 = v7 * 2
```

However, liveness analysis has already reserved `R0` for the function's return value to comply with the ABI. So even though `v7` is dead, `R0` is not available for reuse yet. What about `R1`? The register `R1` holds `v8`, and `v8` will be needed again in the final block `b2`. So we can't overwrite it or spill it to memory.

As a result, the allocator assigns `v19` to `R2`, the next available register. By the end of block `b3`, the state is:

- `R1 = v8` (still live)
- `R2 = v19` (newly computed value)

```
b1: (R0=v7, R1=v8)
b3: ← b1
    R2 v19 = SLLconst [1] v7
Plain → b2
```

Illustration 159. Register R2 assigned to value v19

Block `b4` is structured similarly to `b3` and also inherits the state from `b1 : (R1 = v8, R2 = v19)`.

```
func example(a int, b int) int {
    var y int
    if b > 0 {
        y = a * 2
    } else {
        y = b * 2
    }
    return y + b
}

b1: (R0=v7, R1=v8)
b4: ← b1 // else (b <= 0)
v5 = SLLconst [1] v8 // y = b*2
Plain → b2
```

Illustration 160. The block `b4`

But here is a subtle difference. The allocator doesn't just assign the next free register. Instead, it looks ahead to the merge point at block `b2` to determine **preferred registers**. At the merge point, there's a `Phi` operation: one SSA value (`y`) must come from either `v19` (from `b3`) or `v5` (from `b4`):

```
func example(a int, b int) int {
    var y int
    if b > 0 {
        y = a * 2
    } else {
        y = b * 2
    }
    return y + b
}

b2: ← b3 b4
v16 = Phi v19 v5 // y
v18 = ADD v16 v8 // temp = y+b
v1 = InitMem <mem>
v20 = MakeResult v18 v1 // return (temp,mem)
Ret v20
```

Illustration 161. Phi operation in `b2`

Both paths must place their result into the **same physical register** before the merge point. So the allocator checks the `Phi` operation in `b2` while processing `b4`. It notices that `v19` is already assigned to `R2` and therefore tries to assign `v5` to the same register to keep things consistent.

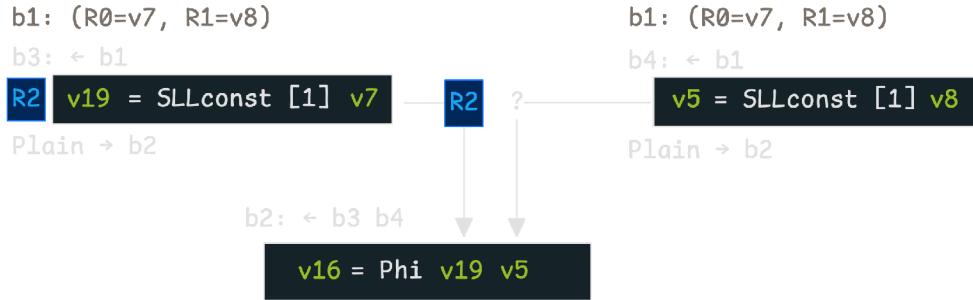


Illustration 162. Allocator prefers same register for Phi

By aligning with the expected register for the Phi node, **v5** is also placed in **R2**. So by the end of block **b4**, the register state is:

- **R1 = v8**
- **R2 = v5**

```

b1: (R0=v7, R1=v8)
b4: ← b1
R2 | v5 = SLLconst [1] v8
Plain → b2

```

Illustration 163. v5 assigned to R2 in b4

Finally, block **b2** becomes straightforward to process. The Phi node output (result of merging **v19** and **v5**) will remain in **R2**, since both incoming values are already in that register. The final return value is moved to **R0**, as required by the ABI.

```

b2: ← b3 b4
R2 | v16 = Phi v19 v5
R0 | v18 = ADD v16 v8
v1 = InitMem <mem>
v20 = MakeResult v18 v1
Ret v20

```

Illustration 164. Register state: R2 = y, R0 = return value

The **MakeResult** operation for **v20** creates a logical grouping of values that will be returned by the function. It doesn't need a physical register itself. Its purpose is to indicate which values make up the function's return tuple.

After register allocation, a few additional steps are performed to finalize the generated code:

- **Spill placement:** If any value was marked for spilling during allocation, this phase inserts a store register instruction (`StoreReg`) instruction. It is placed in the block where the value is still in a register and where it dominates all later uses (restores).
- **Stack allocation:** This phase assigns stack slots to all SSA values that are not held in registers at the end of register allocation. It ensures that every value still needed by the program has a valid memory location. When the value is needed again, it is reloaded with a `LoadReg`.
- **Shuffle phase:** This phase corrects register mismatches at merge points. For example, if the Phi instruction in block `b2` expects both `v19` and `v5` to arrive in the same register (like `R2`), but `v5` is in `R3` at the end of block `b4`, the allocator inserts a move instruction at the end of `b4`: `MOV R3 → R2`. This ensures the Phi input `v5` aligns with the expected register in `b2`, satisfying the Phi requirement.

## Critical Edge Splitting

Register allocation requires that there are no **critical edges** in the control flow graph. To enforce this, the compiler runs a pass called critical edge splitting before register allocation begins.

A critical edge is any edge that goes from a block with multiple successors (a "split point") to a block with multiple predecessors (a "merge point"):

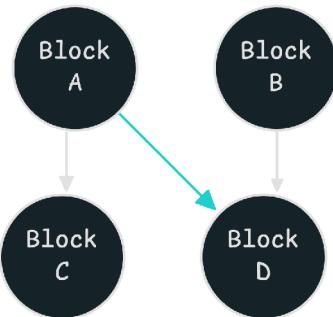


Illustration 165. Critical edge from A to D

In the diagram, the edge `A → D` is critical because:

- `A` has multiple successors (`C`, `D`).
- `D` has multiple predecessors (`A`, `B`).

This situation is problematic for register allocation. Imagine block **D** contains a Phi function. That Phi needs to receive a value in a specific register depending on whether control came from **A** or **B**. For instance:

- From **A**, it expects the value in **R1**.
- From **B**, it expects a different value, also in **R1**.

Now we face a conflict. Where should we place the code that ensures the correct value is in **R1** before entering **D**?

- We can't place it at the end of **A**, because it might interfere with the path from **A** to **C**.
- We also can't place it at the start of **D**, because that would affect the path from **B** to **D**.

The solution is to break the critical edge by inserting a new block between **A** and **D**. This is a small, empty block often called a **pad block**:

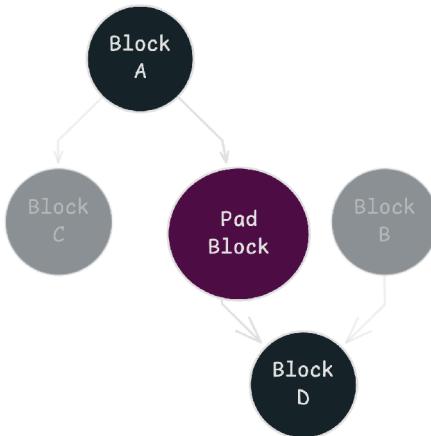


Illustration 166. Pad block splits critical edge

This new pad block creates a dedicated location where the register fix-up can safely happen. Now, if we need to move a value into **R1**, we can do it inside the pad block without affecting other control paths. It cleanly isolates the fix-up logic for just the **A → D** transition.

## 5.9 Stage 7: Machine Code Generation

At the end of the SSA pipeline, after all optimizations and register allocation, we have a fully SSA-transformed function (`*ssa.Func`). This function is no longer high-level or abstract. It consists of concrete operations that are scheduled in a

specific order, with physical registers assigned and values ready for execution. The final step is to walk over this `ssa.Func` and generate actual machine code.

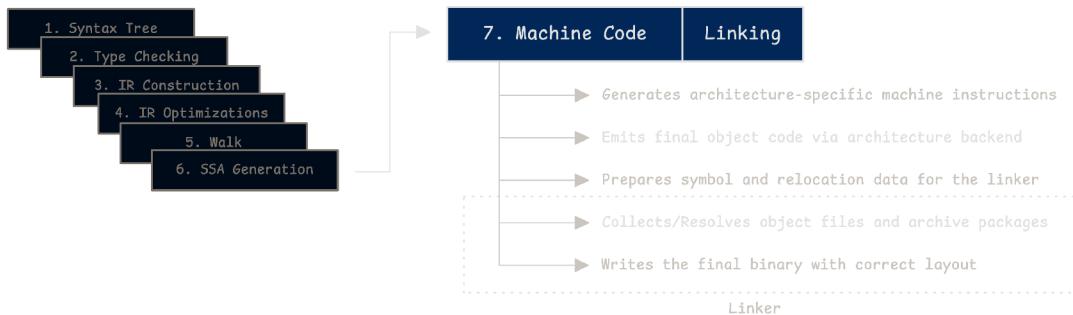


Illustration 167. Final stage: from SSA to binary

The SSA backend hands off the `ssa.Func` to the code generation phase:

```
func genssa(f *ssa.Func, pp *objw.Progs) {}
```

The compiler iterates through each basic block in the SSA function, processing values (SSA instructions) in order. For each value, it uses architecture-specific logic (provided by `Arch.SSAGenValue`) to translate the value into one or more machine instructions. These are not raw bytes yet. They are structured representations of instructions (`obj.Prog`) that closely reflect what will be emitted into the final binary.



Illustration 168. Codegen dispatches per architecture

After processing a block, it calls `Arch.SSAGenBlock` to generate any architecture-specific block-level code. Let's return to our earlier example:

Go	Go SSA
<pre>func example(a int, b int) int {     var y int     if b &gt; 0 {</pre>	<pre>b1:-          v8 = ArgIntReg &lt;int&gt; {b+0} [1] : R1 (b[int])           v7 = ArgIntReg &lt;int&gt; {a+0} [0] : R0 (a[int])</pre>

Go	Go SSA
<pre> a * 2      y = } else { y = b * 2 } return y + b } </pre>	<pre> v2 = CMPconst &lt;flags&gt; [0] v8 GT v2 → b3 b4  b3: ← b1- v19 = SLLconst &lt;int&gt; [1] v7 : R2 Plain → b2  b4: ← b1- v5 = SLLconst &lt;int&gt; [1] v8 : R2 Plain → b2  b2: ← b3 b4- v16 = Phi &lt;int&gt; v19 v5 : R2 (y[int]) v1 = InitMem &lt;mem&gt; v18 = ADD &lt;int&gt; v16 v8 : R0 v20 = MakeResult &lt;int,mem&gt; v18 v1 : &lt;&gt; Ret v20  name a[int]: v7 name b[int]: v8 name y[int]: v16 </pre>

For this SSA example, the branch elimination pass has been disabled. If this pass were active, it could potentially merge some of these blocks into one.

This SSA representation contains three blocks: the entry block (`b1`), the true branch (`b3`), the false branch (`b4`), and the merge or exit block (`b2`). Each block contains a list of values that represent specific operations in the function. Register allocation has already been performed, so each value includes assigned physical registers.

The compiler begins code generation with block `b1`:

```

b1:- 
    v8 = ArgIntReg <int> {b+0} [1] : R1 (b[int])
    v7 = ArgIntReg <int> {a+0} [0] : R0 (a[int])
    v2 = CMPconst <flags> [0] v8
GT v2 → b3 b4

```

The values `v8` and `v7` represent the input arguments `b` and `a`, assigned to registers `R1` and `R0`. The compiler recognizes these arguments at this stage but

defers generating machine code for them. The responsibility for placing arguments into registers lies with the calling function. For instance, when function `A` calls function `B`, `A` prepares the arguments in `R0`, `R1`, etc., following the Application Binary Interface (ABI). The called function (callee), `B`, only needs to know where to find these arguments based on the ABI conventions.

Next, consider the instruction `v2`:

```
v2 = CMPconst <flags> [0] v8
```

This compares the value of `b` (stored in register `R1`) with the constant `0`. The `genssa` function passes this instruction to the ARM64-specific code generator (`Arch.SSAGenValue`), which emits the corresponding machine instruction:

```
CMPconst <flags> [0] v8 → CMP $0, R1
```

Illustration 169. CMPconst to CMP

This becomes `CMP $0, R1`. It compares the immediate value 0 with the contents of register `R1` and updates the processor's condition flags accordingly.

The next part of block `b1` is a conditional branch based on the comparison:

```
GT v2 → b3 b4
```

This means: "If the result of `v2` is greater than zero, jump to block `b3`. Otherwise, jump to block `b4`."

However, remember the layout pass? It reorders blocks to improve performance. In this case, it places block `b3` (the case for `b > 0`) right after block `b1`. This means the program should fall through to `b3` and only jump if the condition is false.

But the logic above is to jump when `b > 0`, which is the opposite of what we want. To fix this, the compiler emits the inverse conditional branch instruction: `BLE` (branch if less than or equal to). This creates the desired control flow:

```
CMP $0, R1  
GT v2 → b3 b4 → BLE <b4>
```

Illustration 170. Branch Inversion for Fallthrough Optimization

It translates to: "If `b ≤ 0`, jump to block `b4`. Otherwise, fall through to block `b3`."

Now in block **b3**, the first instruction is a left shift. However, **SLLconst** is a pseudo-instruction. The ARM64 backend (`Arch.SSAGenValue`) converts it into an actual machine instruction:

```
CMP $0, R1  
BLE <b4>  
v19 = SLLconst <int> [1] v7 : R2 —————→ LSL $1, R0, R2
```

Illustration 171. Instruction selection: shift-left constant resolved

This becomes **LSL \$1, R0, R2**, which shifts the value in **R0** (that is, **v7**) left by one bit and stores the result in register **R2**. Block **b3** ends with a plain jump to **b2**:

```
CMP $0, R1  
BLE <b4>  
LSL $1, R0, R2  
Plain → b2 —————→ JPM <b2>
```

Illustration 172. Plain SSA jump becomes JMP <b2> (placeholder)

This is an unconditional jump. Block **b4** performs a similar operation, but it uses **v8** in register **R1** instead:

```
CMP $0, R1  
BLE <b4>  
LSL $1, R0, R2  
JPM <b2>  
v5 = SLLconst <int> [1] v8 : R2 —————→ LSL $1, R1, R2
```

Illustration 173. Lowered: SSA SLLconst on v8 to ARM64 LSL

Now we reach the final block, **b2**, which contains a Phi node:

```
b2: ← b3 b4  
v16 = Phi v19 v5 : R2  
v18 = ADD v16 v8 : R0  
v1 = InitMem <mem>  
v20 = MakeResult v18 v1
```

Phi nodes do not translate into instructions directly. Instead, earlier compiler passes have already handled register assignment so that the inputs to the Phi (**v19** and **v5**) and the result (**v16**) all share the same physical register (**R2**). As a result, whichever control path was taken, the shifted value is already available in **R2**.

`v18` represents the final addition: `y + b`. The backend (`Arch.SSAGenValue`) translates this into an `ADD` instruction:

```
CMP $0, R1
BLE <b4>
LSL $1, R0, R2
JPM <b2>
LSL $1, R1, R2
v18 = ADD <int> v16 v8 : R0 → ADD R1, R2, R0
v20 = MakeResult <int,mem> v18 v1 : <> RET
Ret v20
```

Illustration 174.  $y + b$  lowered to: `ADD R1, R2, R0`

This becomes `ADD R1, R2, R0`, which adds the value in `R2` (`y`) to the value in `R1` (`b`) and stores the result in `R0`. In the Go ABI, `R0` is the standard reserved return register for integer return values.

Block `b2` is a return block. The backend emits the actual `RET` instruction, returning control to the caller and using the value in `R0` as the function's result.

After all blocks and values are processed, `genssa` enters its final stage. It performs several important tasks to complete the compilation:

1. Generates debugging information (location lists) that map variables and line numbers to machine code locations. This is essential for tools like debuggers and stack traces.
2. Resolves all branch targets. Each block has a corresponding starting `obj.Prog` instruction. Branch instructions like `BLE` and `JMP`, emitted earlier, had their targets left unset. Now, the compiler sets their `prog.target` fields to point to the exact `obj.Prog` representing the destination block.



Illustration 175. Placeholders before linking: symbolic targets inside `obj.Prog`

3. Computes the required stack frame size. This includes space for local variables, argument spill slots, and any temporary values needed for calls. In this simple function, the frame size is minimal, but the step is still necessary.

At this point, the Go assembly representation looks like the following:

```

00000 TEXT main.example(SB), ABIInternal
...
00006 CMP $0, R1
00007 BLE 10
00008 LSL $1, R0, R2
00009 JMP 11
00010 LSL $1, R1, R2
00011 ADD R1, R2, R0
00012 RET
00013 END

```

All these are still `obj.Prog` structures. They are not yet raw machine code. The final step is handled by the `Flush` function, which converts the list of `obj.Prog` entries into actual binary machine code:

Figure 108. Flush (src/cmd/compile/internal/objw/prog.go)

```

// Flush converts from pp to machine code.
func (pp *Progs) Flush() {
    plist := &obj.Plist{Firstpc: pp.Text, Curfn:
    pp.CurFunc}
    obj.Flushplist(base.Ctxt, plist, pp.NewProg)
}

```

The call to `Flushplist` performs the conversion:

```

func Flushplist(ctx *Link, plist *Plist, newprog ProgAlloc)
{}

```

This function walks through the `obj.Prog` list and passes each instruction to the architecture-specific backend. These handlers live in packages like `cmd/internal/obj/x86/` or `cmd/internal/obj/arm64/`, depending on the target platform. Each instruction is converted into its corresponding **binary machine code byte sequence**.

The code is now prepared for linking into the final executable. The assembler generates a symbol for each global item, like functions or variables. However, when an instruction references one of these symbols, the assembler doesn't yet know its final memory address. So, it inserts a placeholder referencing the symbol. This is because the linker, not the assembler, determines the final memory layout. The assembler cannot assign final addresses itself.

To handle this, the assembler creates a relocation entry in the relocation table of the object file (`.o`). Each entry says something like: "At location `X` in the code or data, there is a reference to symbol `Y`. Linker, please calculate the final address for `Y` and patch location `X` with that address."

The Go assembler produces full object files (`.o`) that conform to the standard format expected by the linker. The layout varies slightly depending on architecture, but the object file typically includes these key sections:

- `TEXT` section (`.text`): Contains the actual machine instructions for executable code. This section may be split across multiple segments in some cases.
- `DATA` section (`.data`): Holds initialized global variables.
- `BSS` section (`.bss`): Holds uninitialized global variables. These variables do not have a starting value assigned in the source code.
- `RODATA` section (`.rodata`): Stores read-only data such as constants and string literals.
- Symbol tables: These contain all declared symbols (functions, global variables, etc.) along with their metadata. Each symbol is recorded with a relative address and its associated section (for example, `DATA`, `BSS`, or `RODATA`).
- Relocation tables: These keep track of positions in the code or data that require modification during linking. They mark references to symbols whose final locations were not known at assembly time.
- DWARF Debugging Information: Optional section for debugging support. It includes mappings from machine code to source lines, variable types, scopes, and more.

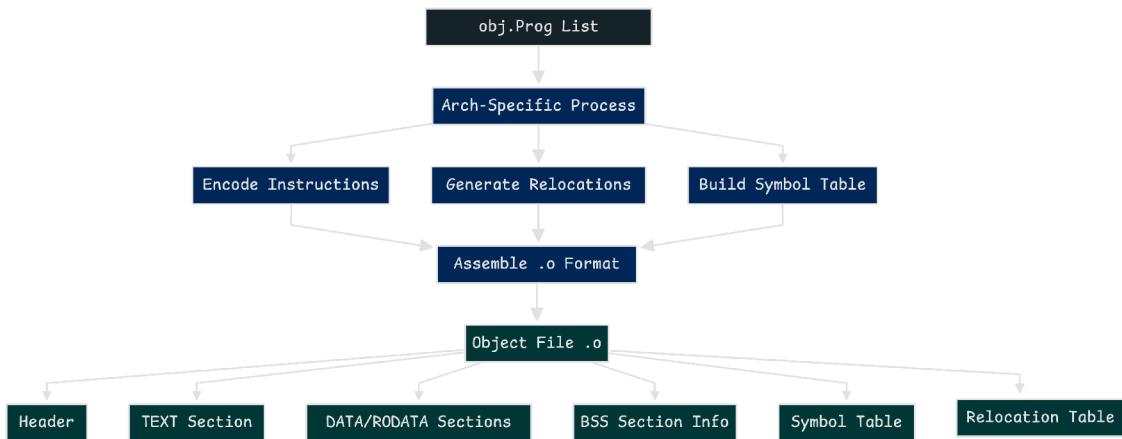


Illustration 176. From SSA to object file: How Go produces `.o` binaries

Once these sections are generated, the job of the assembler is complete. However, the Go compiler still has one more task: generating export data. You might remember export data from earlier, during IR node construction. Now, the compiler generates export data for the current package:

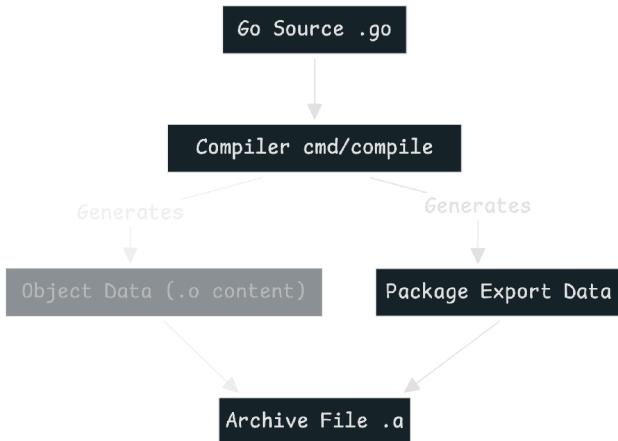


Illustration 177. Export data enables cross-package compilation

This export data includes type information, function signatures, and constants that other packages may need when importing this package. It enables separate compilation and avoids full re-compilation of dependencies.

Finally, Go combines the assembler output (`.o` file) and the export data into a single archive file, typically with a `.a` extension. This archive contains both pieces but in different sections:

- `___.PKGDEF` section: Stores the export data.
- `go.o` section: Contains the compiled object file with code and data.

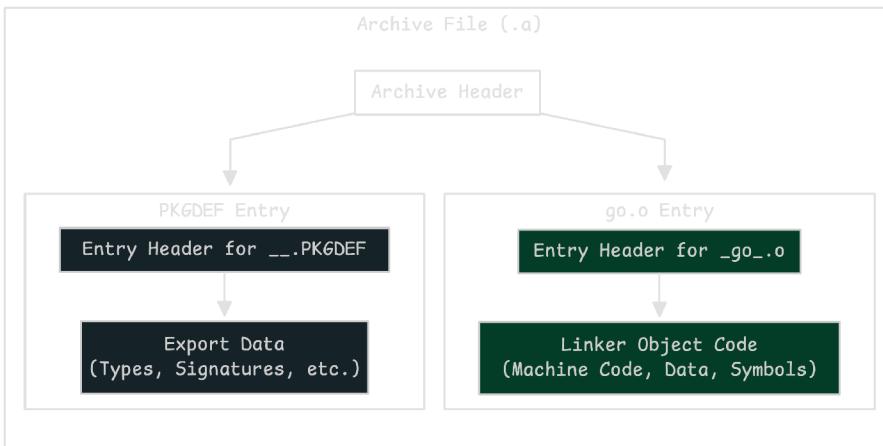


Illustration 178. Archive file holds code and export data

You can actually observe this behavior directly in the Go build cache:

```
$ ar -tv /theanatomyofgo/Library/Caches/go-
build/3e/3e6641643282eda2605f3727ac58a2eda99b146539a56b9f08e0
811a0d349a53-d
```

```

rw-r--r--      0/0      37542 Jan  1 08:00 1970 __.PKGDEF
rw-r--r--      0/0      1137368 Jan  1 08:00 1970 _go_.o

```

Here, `___.PKGDEF` contains the export data, and `_go_.o` is the object file with the compiled code and data. At this point, the linker takes over. It reads all input files, including archive files (`.a`) and possibly individual object files (`.o`). Its main task is to generate the final executable:

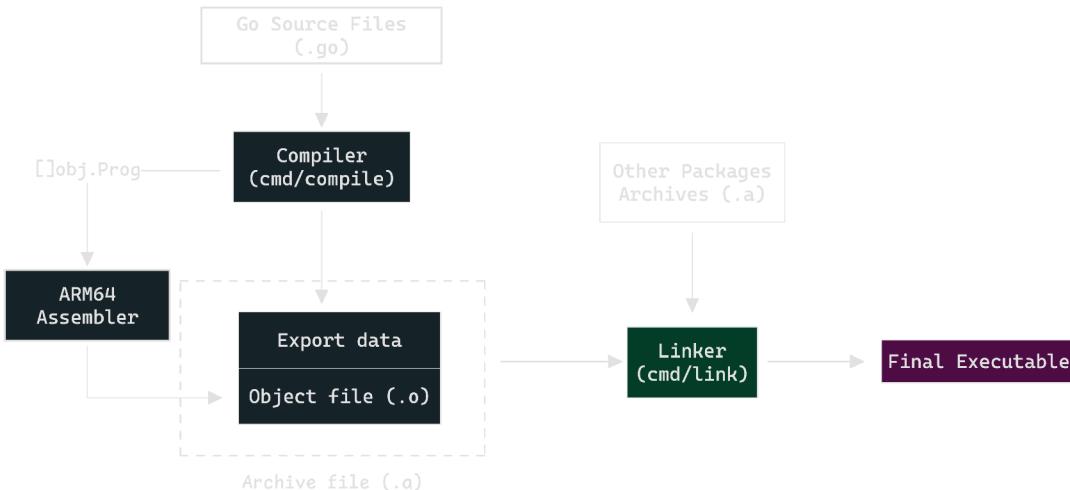


Illustration 179. Linking Go packages into binary

The linker focuses on the object content. It extracts code sections, data sections, symbol tables, and relocation entries from the input files. It then builds a combined symbol table by collecting all symbols (functions, global variables, etc.) from the inputs. It checks where each symbol is defined and validates that all referenced symbols are resolved. If any symbol is used but not defined in any input file, the linker reports an error:

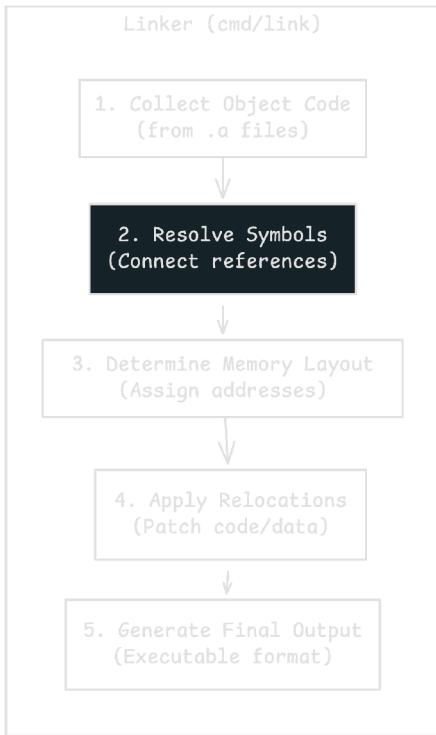


Illustration 180. How the Go linker operates

Next, the linker concatenates all the `.text`, `.data`, and other sections from the object files into unified sections in the output file. For example, it merges all `.text` sections into the final executable's code segment. Once this step is complete, the linker has a full view of the memory layout for the executable.

Then it processes the relocation entries created earlier during assembly. For each relocation entry, the linker uses the final memory address of the target symbol and calculates the correct value. It patches that value into the corresponding location in the code or data section.

This patching ensures that all function calls, data references, and jump targets use the correct absolute addresses in the final executable.

Now that we've covered the core mechanics, it's worth revisiting the first and second sections of this chapter. With the bigger picture in mind, those earlier steps —how Go builds and organizes a program—may now click into place in a way they didn't before.

## Wrapping Up

The Go compilation process transforms source code into executable machine instructions. Tools like `go build -n` reveal the sequence of steps within the Go build system.

The compiler begins by parsing source code into an Abstract Syntax Tree (AST), which structurally represents the program. Subsequently, the type-checking phase verifies code correctness and safety based on Go's language rules. This stage also calculates constant values and resolves types across package boundaries, often using export data loaded from imported packages.

Export data is fundamental to Go's separate compilation capability. Package information, types, and object details are encoded using relocations, enabling the compiler to link code units without requiring access to all source files simultaneously. This leads to the generation of an Intermediate Representation (IR), a lower-level format derived from the AST, suitable for subsequent analysis and optimization.

Optimization is a key part of the Go compiler's function. Techniques like devirtualization replace certain interface method calls with direct function calls, reducing runtime overhead when the concrete type is known at compile time. Inlining, another significant optimization, involves embedding a function's body directly at its call site. This requires evaluating function complexity and call context to balance performance benefits against potential code size increases.

The compilation pipeline includes the 'walk' or 'order' phase, responsible for lowering higher-level Go language constructs. Features like for-range loops (for arrays, slices, maps, strings) and `switch` statements are converted into simpler operational sequences, including conditional jumps, temporary variables, and runtime calls. This lowering simplifies the representation, ensures consistent behavior, and prepares the IR for the Static Single Assignment (SSA) transformation.

The introduction of an SSA backend significantly advanced the Go compiler. In SSA form, each variable is assigned a value exactly once, clarifying data flow and enabling effective optimizations. Passes based on SSA, such as bounds check elimination and nil check elimination, remove redundant safety checks, improving the performance of the generated code. The SSA stage itself influences compilation speed, creating a dynamic where its processing overhead can be partially offset by the performance gains it yields, sometimes even accelerating the compiler.

Following SSA optimizations, a further lowering phase translates generic SSA operations into instructions specific to the target architecture (e.g., AMD64,

ARM64). Architecture-specific rewrite rules map these portable IR operations to pseudo-operations representing the target machine's instruction set.

The final stages involve code generation and linking. The architecture-specific SSA form is translated into assembly language. The assembler converts this assembly into binary machine code, producing object files ( `.o` ). Symbols for global entities are created, but since final addresses are unknown, relocation entries serve as placeholders. The linker then processes these object files and archives. It resolves symbols across all compilation units, determines the final memory layout, calculates addresses for relocation entries, and patches the code and data sections. This procedure yields a single, executable binary file.

From initial source parsing to final binary linking, the Go compiler applies a structured sequence of transformations and optimizations. This process underpins Go's characteristics of simplicity, efficiency, and reliability. The complex internal steps result in a streamlined pathway from Go source code to performant native executables across diverse architectures.

## References

- [go117] Go 1.17 Release Notes: <https://go.dev/doc/go1.17>
- [go121] Go 1.21 Release Notes: <https://go.dev/doc/go1.21>
- [gpc2017] GopherCon 2017: Keith Randall - Generating Better Machine Code with SSA: <https://www.youtube.com/watch?v=uTMvKVma5ms>
- [ginl] Proposal: cmd/compile: add a go:inline directive: <https://github.com/golang/go/issues/21536>
- Go compiler internals: Adding a new statement to Go: <https://eli.thegreenplace.net/2019/go-compiler-internals-adding-a-new-statement-to-go-part-1/>
- Introduction to the Go compiler: <https://github.com/golang/go/tree/master/src/cmd/compile>
- Discussion - Meaning of SSA operation: <https://groups.google.com/g/golang-nuts/c/Sm8Smqhae58>

# Chapter 6: Functionality

## 1. Preliminaries

This chapter touches on some ideas that we will return to later, such as the stack frame (Chapter 7) and the MPG model (Chapter 8). We'll lay the groundwork here, before going into more depth in these later chapters. When we reach those later chapters, you will already have the basic concepts in mind.

### Prerequisite: MPG Model Overview

The MPG model is at the heart of how Go manages concurrency. It has three main parts:

- M: Machine, which is an OS thread that runs your code. This thread can also get blocked if it needs to wait for something from the operating system.
- P: Processor, but this is not a hardware CPU core. Instead, it is something that the Go runtime manages. It gives the thread what it needs to run goroutines.
- G: Goroutine, which is a lightweight function running alongside other goroutines.

A Machine (OS thread) cannot run a Goroutine by itself. It must have a processor. The processor gives the thread the right context to run goroutines. You can think of it like this: the thread carries the processor, and the processor carries the goroutine. If a thread needs to do a blocking operation, such as a system call, it gives up its processor. This lets another thread grab the processor and keep running other goroutines.

The number of threads and goroutines can change while your program is running. The number of processors, however, stays fixed and can be set manually by `runtime.GOMAXPROCS(n)`. If a thread does not have a processor, it just waits. It cannot run Go code unless it has one.

### Prerequisite: Stack Pointer, Program Counter, and Argument Pointer

This chapter also introduces the idea of the stack frame. The stack frame is the backbone for how the `defer` statement and the `panic` recovery mechanism work. It also sets the stage for understanding profiling and tracing at the runtime level.

Whenever your program calls a function, the processor sets aside a block of stack memory just for that function call. This block is called the stack frame. It contains everything the function needs to work and to give control back to its caller when finished:

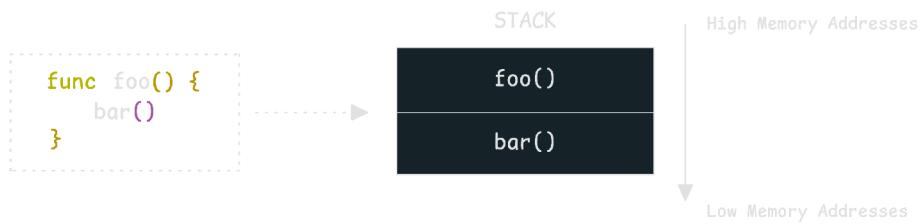


Illustration 181. Stack frame allocation during function calls explained

The stack in memory grows "downward". This means it goes from high memory addresses to low addresses. As you push data onto the stack, the stack pointer gets smaller. The stack frame usually contains the return address (so the CPU knows where to go after the function ends), the arguments passed to the function, space for the function's local variables, and more.

When a function finishes, its stack frame is removed. The program uses the saved information to restore the previous state. It moves the stack pointer back to where it was before the call, restores any saved values, and jumps back to the return address. This lets the program continue right where it left off.

This section focuses on three important ideas: the stack pointer (SP), the program counter (PC), and how function arguments are managed.

## Stack Pointer (SP)

Every function usually gets its own stack frame. The stack pointer always points to the top of the stack. In this context, "top" means the lowest memory address on the stack:



Illustration 182. Stack pointer indicating top of the stack

Each time a function is called and set up, the stack pointer moves downward to make room for that function's stack frame:

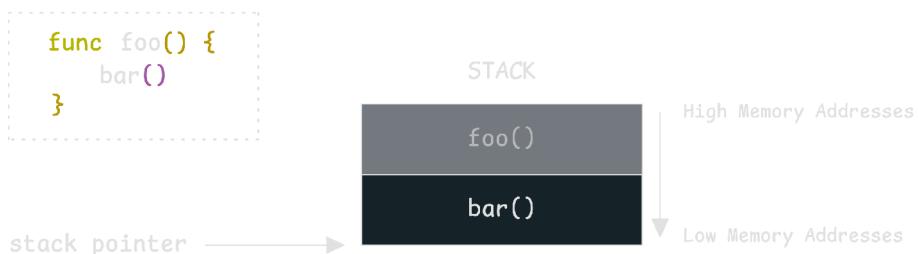


Illustration 183. Stack pointer moves down for each call

## Argument Pointer (argp)

If you call a function, the calling function has to pass data to the function being called. These are the arguments in your function call. So where does this data go in memory, and how does the called function find it?

In Go, function arguments can be passed in two ways. Some arguments go into processor registers. Registers are very fast storage spots inside the CPU. But there are only a few registers available. If you have more arguments than registers, or if an argument is too large, then the extra arguments are put into memory, usually on the stack. This was explained clearly in Chapter 5 in the register section.

The function being called (the callee) needs a way to find these arguments in memory. The argument pointer provides a reference point. It tells the callee exactly where its arguments begin.



Illustration 184. Argument pointer marks start of function arguments

Depending on the hardware, the layout might change. However, the argument pointer (`argp`) typically points to the same address as, or just above, the stack pointer. Note that the argument pointer is a runtime/compiler-internal concept and not a portable architectural construct.

## Program Counter (PC)

The program counter (PC), sometimes called the instruction pointer, is a special register in the CPU. It keeps track of the memory address for the next instruction your program will run. As your code runs, the CPU fetches instructions from memory, decodes them, and then executes them one by one.

After running each instruction, the program counter updates itself to point to the address of the next instruction.



Illustration 185. Program counter points to next instruction location

This diagram is a simple way to picture what happens, but it is not technically exact. The program counter operates at the instruction level, not the code line level. One line of code may become several instructions. Still, the diagram helps show what the program counter does as instructions run.

Whenever a function calls another function, the PC holds the address of the next instruction in the current function. Before jumping to the new function, this address is saved as the return address. That way, when the called function finishes, it knows where to come back and continue. The method for saving the return address depends on the CPU type:

- On AMD64, the `CALL` instruction automatically puts the return address (the address right after the call) on the stack.
- On ARM64, the `BL` (Branch with Link) instruction (equivalent of `CALL` in AMD64) stores the return address in a special link register (`R30`). Later, this value is saved in the called function's stack frame.

This difference is important for understanding how stack frames and function calls work at a low level.

## 2. Function Basics to Advanced

A function is a self-contained sequence of statements that performs a specific task and can be called from other parts of the program.

In Go, a normal function starts with the keyword `func`, followed by the function name, a list of parameters in parentheses, return type information, and the body inside curly braces. For example,

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

println(add(1, 2)) // 3
```

This code declares a function called `add`. It takes two integers and returns their sum. Go always makes return types clear. If your function needs to return more than one value, you can list the return types, separated by commas, in a single set of parentheses:

```
// a function that returns both quotient and remainder
func divMod(numerator, denominator int) (int, int) {
    q := numerator / denominator
    r := numerator % denominator
    return q, r
}

q, r := divMod(17, 5) // q == 3, r == 2
```

At the call site, you can capture both results at once by assigning them to variables in a single statement.

### Functions as First-Class Citizens

## Pass-by-Value Semantics

Functions are first-class citizens in Go. This means you can use functions as values, just like any other data type. You can assign a function to a variable, pass it as an argument to another function, return it from a function, or store it in a data structure:

```
// Function type declaration
var add func(int, int) int

// Function as a value in a map
var hammer32 = map[string]func(*uint32, int){
    ...
}

// Function type as parameter and return type
func OnceFunc(f func()) func() {
    ...
}
```

The default value of a function type is `nil`. Functions belong to the "uncomparable" type category, which means you cannot use operators like `==` to compare functions (except when comparing with `nil`).

The type of a function in Go is defined by its "signature." When we say two functions have the same signature, it means the types of their parameters and their return values match exactly and in the same order:

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

func sub(c, d int) int {
    return c - d
}

func main() {
    f := add
    println(f(1, 2))

    f = sub
    println(f(1, 2))
}
```

Here, `add(a, b int)` and `sub(c, d int)` have the same signature. The names of their parameters do not matter for the function signature.

Go does not support function overloading or operator overloading. You cannot define multiple functions with the same name and different parameters. You also cannot change the meaning of operators for different types.

The Go FAQ explains it this way: "*Method dispatch is simplified if it doesn't need to do type matching as well. Experience with other languages told us that having a variety of methods with the same name but different signatures was occasionally useful but that it could also be confusing and fragile in practice. Matching only by name and requiring consistency in the types was a major simplifying decision in Go's type system.*"

When it comes to the function signature, you can ignore the names of the parameters and return values. This applies to both interface definitions and function declarations:

```
func add(int, int) int {
    return 0
}

type Add func(int, int) int

type Adder interface {
    Add(int, int) int
}
```

Go also supports named return values, sometimes called named result parameters. With named return values, you can give each returned value a name. When you use named return values, Go automatically creates variables with those names. These variables start at their zero value as soon as the function begins:

```
func calculateRectangle(dim1, dim2 float64) (area, perimeter
float64) {
    area = dim1 * dim2
    perimeter = 2 * (dim1 + dim2)
    return
}
```

You can assign values to these named return variables anywhere in your function. When you use a bare `return` statement, Go returns the current values of those variables.

Although named return values can sometimes make code harder to read if used without care, they can also improve your code's documentation. They make it

clear in the function signature what each returned value means.

For example, here is a real example from Go's `internal/singleflight` package:

Figure 109. `internal/singleflight` Named Return Values Example  
(`src/internal/singleflight/singleflight.go`)

```
func (g *Group) Do(key string, fn func() (any, error)) (v  
any, err error, shared bool) {}
```

Without the named return values, it would not be clear what the `bool` return value represents.

Sometimes, you do not need all the return values from a function. Go lets you ignore any unwanted values by using the blank identifier `_`. You can use this as many times as needed:

```
area, _ := calculateRectangle(3, 4)  
  
// If the function returns more than two values  
area, _, _ := calculateRectangle(3, 4)
```

Go also provides a simple way to pass the multiple return values from one function straight into another function, as long as the numbers and types line up.

This feature allows you to chain functions without needing extra variables:

```
func dimensions() (int, int, int) {  
    return 5, 3, 2  
}  
  
func calculateVolume(length, width, height int) int {  
    return length * width * height  
}  
  
calculateVolume(dimensions())
```

In this example, `calculateVolume` can take the results of `dimensions` directly because they are all integers, and the numbers match. You can also write a function that simply returns the result of another function if their return types are the same:

```
func dimensionWrapper() (int, int, int) {  
    return dimensions()
```

```
}
```

Another important detail in Go is that function arguments are always passed by value. This means that when you pass an argument to a function, Go creates a copy of that argument. If you are working with "reference types" like slices, maps, or channels, the value being copied is actually a **descriptor**. This descriptor holds a pointer to the underlying data, along with other metadata.

This idea can be a bit confusing if you are used to languages like C# or Java. In those languages, "passing by reference" and "passing by value" have different rules. Take this example in Go:

```
func modifySlice(s []int) {
    s = []int{1, 2, 3}
}

func main() {
    s := []int{4, 5, 6}
    modifySlice(s)
    fmt.Println(s) // [4 5 6]
}
```

In this example, the function `modifySlice` assigns a new slice to `s`. Because the slice descriptor is passed by value, this reassignment only changes the local copy inside the function. It does not change the slice in the calling function.

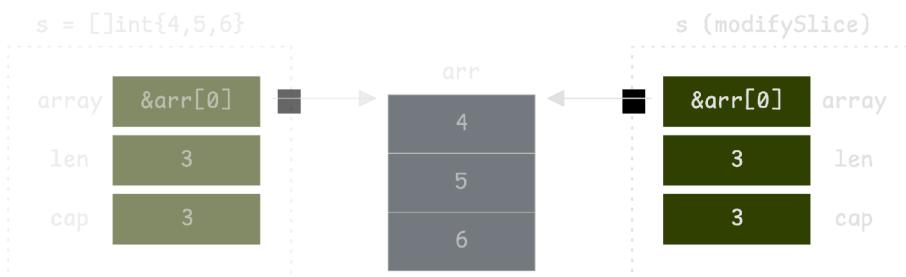


Illustration 186. Slice argument copies only the descriptor structure

That is why the original slice remains the same. This behavior is different from how things work in C#. To truly pass a reference by reference in C#, you use the `ref` or `out` keyword:

```
void ReplaceList(ref List<int> list) {
    list = new List<int> { 1, 2, 3 };
}

var myList = new List<int> { 4, 5, 6 };
```

```
ReplaceList(ref myList);
Console.WriteLine(string.Join(", ", myList)); // Output: 1,
2, 3
```

In this C# example, the reference itself is passed by reference. Any reassignment changes the variable in the caller as well. Some say Go does not have true "pass by reference." This is true, depending on whether you mean passing the descriptor or passing the real reference like in C#.

In Go, if you modify the elements inside a slice instead of reassigning it, your changes are visible to the caller:

```
func modifySlice(s []int) {
    s[0] = 1
}

func main() {
    s := []int{4, 5, 6}
    modifySlice(s)
    fmt.Println(s) // [1 5 6]
}
```

This works because, as explained in Chapter 3, a slice is actually a small struct. It contains a pointer to an array, the length of the slice, and its capacity. When you pass a slice to a function, Go copies this struct—the descriptor or wrapper.

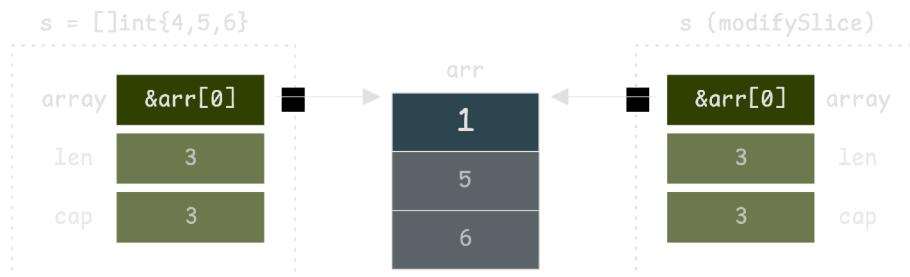


Illustration 187. Modifying slice elements updates underlying array data

So, you end up with two different slice descriptors, both pointing to the same array in memory. If you modify the elements of the slice, you are updating the shared underlying array, and the changes are seen in both places. This happens because you are working with an extra layer of indirection.

## Closures: Capturing and Lifetime

In Go, you cannot define a named function inside another function. All named functions have to be at the package level. They cannot be nested. However, because Go treats functions as first-class values, you can create anonymous functions (sometimes called function literals) inside other functions.

These do not need a name and can be assigned to variables, passed around, or even returned from other functions:

```
func main() {
    x := "Hello, Go!"

    f := func() {
        fmt.Println(x)
    }

    f()
}

// Hello, Go!
```

Here, we define an anonymous function with `func() { ... }` and assign it to the variable `f`. Later, we call `f()` just like any other function. The special thing about anonymous functions is that they can access variables that are in scope where the function is created. They "close over" their environment. That is why we call them closures.

Closures are powerful. An inner function can capture and keep access to the variables of the outer function, even after the outer function has returned:

```
func Incrementer() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {
    inc := Incrementer()
    fmt.Println(inc()) // 1
    fmt.Println(inc()) // 2
    fmt.Println(inc()) // 3
}
```

In this example, the `Incrementer` function returns another function. Inside `Incrementer`, there is a variable `i` that starts at `0`.

The function that `Incrementer` returns is anonymous. It adds `1` to `i` every time you call it, then returns the new value of `i`. When you call `Incrementer`, you get back this function along with its own `i` variable that sticks around between calls.

Go does not just copy the value of variables from the outer function at the time the closure is created. Instead, it builds a lasting connection to those variables. This allows the closure to read and change them as long as it lives. So, the closure and its environment share a state that sticks around as long as the closure itself does.

To really see where this shared state comes from and why closures work this way, we need to look at how the Go compiler manages function values behind the scenes.

## Treating Functions as Data: Function Values

When Go compiles functions, it creates several types of symbols. For every function, there is not only the function code itself, but also some related metadata and special function value symbols.

The function value symbol is especially important. It lets Go support first-class functions. That means you can pass functions around as values, store them in variables, and call them indirectly.

This symbol follows a specific naming rule:

Figure 110. Symbol Naming Convention  
(cmd/compile/internal/ir/func.go)

```
func FuncSymName(s *types.Sym) string {
    return s.Name + ".f"
}
```

So, if you have a function called `main.foo`, the compiler creates a symbol named `main.foo.f`. The middle dot (`.`) is important. It tells the function value symbols apart from regular function symbols. The function value symbol is not the actual code for the function. It is a data structure that holds a pointer to the real function code.

When you assign a function to a variable, as illustrated below:

```
func add(x, y int) int {
    return x + y
}

f := add
f(1, 2)
```

The Go compiler creates the function value symbol (`add·f`) and sets it up in your binary. You will not see this symbol in your Go code, but it is known to the Go compiler and runtime:

Figure 111. funcval Structure (src/runtime/runtime2.go)

```
type funcval struct {
    fn uintptr
    // variable-size, fn-specific data here
}
```

The function value symbol is actually a `funcval` structure. This structure stores a pointer to the function code. If the function is a closure and captures variables from its environment, then `funcval` will also store those captured variables, right after the `fn` field.

To see the address of this `add·f` symbol at runtime, you can use the `println` function as shown below:

```
//go:noinline
func add(x, y int) int {
    return x + y
}

func main() {
    f := add
    println(f)
}

// Output:
// 0x100bce7f0
```

Here, `f` is a pointer to the function value symbol `add·f`. In other words, `f` is of type `*funcval`:



Illustration 188. funcval structure holds pointer to function implementation

To see this in practice, look at this example and its Go assembly code:

```

func main() {
    f := add
    f(1, 2)

    // to avoid register optimization
    println(&f)
}

```

First, the function value symbol `add·f` is loaded, and then the function is called. The `println(&f)` line is a trick to prevent the Go compiler from optimizing the variable `f` into a register. This lets you see the variable `main.f` in the assembly output, instead of just registers like `R0` or `R1`. (This is discussed more in the register optimization section in Chapter 5.)

<code>f := add</code>	<code>0x0018 MOVD \$main.add·f(SB), R2</code>
	<code>0x0020 MOVD R2, main.f-8(SP)</code>
<code>f(1, 2)</code>	<code>0x0024 MOVD main.f-8(SP), R26</code>
	<code>0x0028 MOVD (R26), R2</code>
	<code>0x002c MOVD \$1, R0</code>
	<code>0x0030 MOVD \$2, R1</code>
	<code>0x0034 CALL (R2)</code>

Illustration 189. Go assembly: function assignment and call sequence

The first two instructions show how the function value is assigned (`f := add`):

- `MOVD $main.add·f(SB), R2` : loads the address of the `main.add·f` symbol into the `R2` register.
- `MOVD R2, main.f-8(SP)` : stores the `funcval` pointer into the local variable `f` on the stack at the address below the stack pointer by 8 bytes (`main.f-8(SP)`).

From this, you can see that the function value symbol `main.add·f` is a special symbol that the Go compiler creates ahead of time and knows how to use.

When the function value call `f(1, 2)` happens, the compiler follows Go's calling convention for function values:

- `MOVD main.f-8(SP), R26` loads the `funcval` pointer from the local variable `f` on the stack into `R26`. On ARM64, `R26` serves as the closure context register. This step is required by Go's ABI for all function value calls.
- `(R26), R2` : Because `R26` points to the `funcval` structure, and the first field in `funcval` is `fn` (which holds the function's address), this instruction loads the address of the actual function entry point into the `R2` register.
- `$1, R0` and `$2, R1` : These load the function arguments. The immediate values `1` and `2` are placed into registers `R0` and `R1`, following the ARM64 calling convention for integer arguments.
- `CALL (R2)` : Calls the function whose address was loaded from the `funcval` structure.

We can think of the process in this way using simple pseudo code:

```
var f = &main.add·f

R26 = f      // MOVD main.f-8(SP), R26
R2 = *R26 // MOVD (R26), R2
R0 = 1
R1 = 2
CALL (R2) // (*R2)(R0, R1) or f(1, 2)
```

You do not have to use assembly code to explore how function value calls work. With just a bit of knowledge about the `unsafe` package and how to bypass the type system, you can investigate this behavior using regular Go code. (This topic is explained in more detail in the bonus section of Chapter 2.)

```
type funcval struct {
    fn uintptr
}

func main() {
    f := add
    fValue := *(*funcval)(unsafe.Pointer(&f))
    println("funcval address:", fValue)

    fnCodePtr := unsafe.Pointer(fValue.fn)
    println("funcval.fn:", fnCodePtr)

    fn := runtime.FuncForPC(uintptr(fnCodePtr))
    if fn != nil {
```

```

        println("Function:", fn.Name())
        println("Entry point:", unsafe.Pointer(fn.Entry()))
    }
}

// Output:
// funcval address: 0x102e76970
// funcval.fn: 0x102e57c30
// Function: main.add
// Entry point: 0x102e57c30

```



Illustration 190. `funcval` pointer resolves to function code address

You can see that the address held in `funcval.fn` is actually the address of `main.add`. The `runtime.FuncForPC` function lets us verify this by returning information about the function at a given program counter value.

The `runtime.FuncForPC` function lets you get detailed information about a function if you have its program counter address. You pass it a `uintptr` value, and it returns a pointer to a `Func` struct that describes the function containing that address, or `nil` if no function is found.

If you have a program counter value from a stack trace, profiling, or debugging, `FuncForPC` is your link between a low-level address and high-level details such as the function name, source file, and entry point.

Digging a bit deeper, when you write `f := add`, the variable `f` points to `main.add.f` in the `RODATA` section. This section is for read-only data that never changes at runtime. In this case, the `funcval` structure is just 8 bytes, holding the address of `main.add` in the `TEXT` section, which is where the code lives.

To help visualize this concept, see the diagram below:

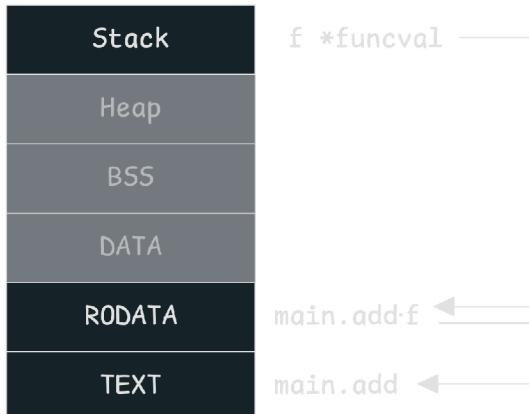


Illustration 191. RODATA holds function value symbol, TEXT holds code

Interestingly, Go does not create a function value symbol in `RODATA` for every function in your code. It only does this when you create a function value by assigning either a named function (such as `add`) or an anonymous function (such as `func() { ... }`) to a variable.

When that happens, the function value symbol is placed in the `RODATA` section:

```
func main() {
    // function value symbol: main.add·f
    f := add

    // function value symbol: main.func1·f
    f = func(x int, y int) int {
        return x + y
    }

    f(1, 2)
}
```

When the compiler sees an anonymous function, it creates an actual function definition for the code and gives it a unique generated name. Anonymous functions inside another function (for example, inside `main`) get names like `main.func1`, `main.func2`, and so on. The compiler keeps a counter to make sure each anonymous function gets a unique name.

Global anonymous functions get similar names, such as `main.init.func1`, `main.init.func2`, and so on. A global anonymous function is an anonymous function defined outside any function:

```
var glob = func() {
    fmt.Println("Hello, World!")
}
```

It's a common misunderstanding to think that defining an anonymous function inside a loop means the function is "recreated" on each iteration. That is not what actually happens in Go:

```
func main() {
    for i := 0; i < 1000000; i++ {
        func() {
            fmt.Println("Hello, World!")
        }()
    }
}
```

In reality, the compiler generates a single function with a unique name for this anonymous code just once, and that function exists as executable code in the binary's `TEXT` section. So, you can be sure that anonymous functions work just like named functions because of how the Go compiler handles them.

## How Closures Work

There is an important difference between an anonymous function that does not capture any variables from its surrounding environment (this is called a *trivial closure*) and one that does capture such variables (this is called a *non-trivial closure*).

A non-trivial closure does not have its function value symbol stored in the `RODATA` section. Instead, its function value symbol is created dynamically at runtime along with the captured variables. Here is an example of a non-trivial closure:

```
func main() {
    x := 1
    f := func(y int) int {
        return x + y
    }
    f(2)

    println(&f)
}
```

Below is its Go assembly code. You do not need to know every detail here, but pay attention to what is happening:

```
// f := func(y int) int {
//   return x + y
// }
0x0018 MOVD $main.main.func1(SB), R1
0x0020 MOVD R1, main..autotmp_3-24(SP)
0x0024 MOVD $1, R1
0x0028 MOVD R1, main..autotmp_3-16(SP)
0x002c MOVD $main..autotmp_3-24(SP), R1
0x0030 MOVD R1, main.f-8(SP)

// f(2)
0x0034 MOVD main.f-8(SP), R26
0x0038 MOVD (R26), R1
0x003c MOVD $2, R0
0x0040 CALL (R1)
```

What you should notice is that there is no function value symbol such as `main.func1.f`, and the `funcval` structure is created dynamically:

1. `MOVD $main.main.func1(SB), R1`: places the function address directly into the first field of the `funcval` structure.
2. `MOVD $1, R1`: loads the value `1` (which is the value of `x`) into `R1` and then stores it in the next position in the closure structure. This is how the closure "captures" the variable—it copies the value into the closure at the moment the closure is created. The calling convention for the function value works the same as already described.

The whole process for creating a non-trivial closure can be summarized in this pseudo code:

```
func main() {
    funcval := funcval{
        fn: $main.func1,
        arg1: 1,
    }

    f := &funcval
    f(2)
}
```

The main difference here is how the `funcval` structure is obtained. With a trivial closure, you get the `funcval` from a predefined symbol in `RODATA`. With a non-

trivial closure, the `funcval` structure is created on the fly, holding any captured variables.

Now, let's see what happens if you change the variable `x` before calling the closure:

```
func main() {
    x := 1
    f := func(y int) int {
        return x + y
    }

    x = 100
    f(2)

    println(&f)
}
```

In this case, `x` is captured by reference, the assembly code would look a bit different:

```
0x0020 MOVD $main.main.func1(SB), R1
0x0028 MOVD R1, main..autotmp_3-16(SP)
0x002c MOVD $main.x-32(SP), R1
0x0030 MOVD R1, main..autotmp_3-8(SP)
0x0034 MOVD $main..autotmp_3-16(SP), R1

0x0038 MOVD R1, main.f-24(SP)
0x003c MOVD $100, R1
0x0040 MOVD R1, main.x-32(SP)
```

Instead of storing the value `$1`, this stores the address of `x` (`$main.x-32(SP)`) in the closure structure. The closure function will then dereference this address whenever it needs to access the value.

You can observe this behavior by building your program with `go build` or by using `go tool compile`. To make the difference even clearer, let's adjust the closure to capture both by value and by reference:

```
func main() {
    x := 1
    y := 3
    f := func() int {
        return x + y
    }
```

```
x = 100
f()

    println(&f)
}
```

In this code, the value of `x` is changed to `100` after the closure is created, but the value of `y` stays the same during the function's execution.

Now, run the static analysis command to see how the compiler handles the closure and looking at the specific lines for closure:

```
$ go build -gcflags="-m=2" .

# theanatomyofgo
./main.go:3:6: cannot inline main: function too complex:
cost 95 exceeds budget 80
./main.go:6:7: can inline main.func1 with cost 4 as: func()
int { return x + y }
./main.go:4:2: main capturing by ref: x (addr=false
assign=true width=8)
./main.go:5:2: main capturing by value: y (addr=false
assign=false width=8)
./main.go:6:7: func literal does not escape
```

This static analysis output tells us that the `x` is captured by reference and `y` is captured by value. In other words, when we modify `x` after the closure is created, the changes are reflected in the closure's behavior.

There are three conditions under which the Go compiler captures by value rather than by reference:

- The variable's address is never taken.
- The variable is never reassigned after being captured by the closure (including inside the closure itself). If it is reassigned, it will be captured by reference.
- The variable's size is 128 bytes or less.

To conclude this section, we will look at another a bit more advanced example:

```
func returnAdder() func(int) int {
    x := 1
    return func(y int) int {
        x += y
    }
}
```

```

        return x
    }

func main() {
    adder := returnAdder()
    println(addr(1))
    println(addr(2))
    println(addr(3))
}

// Output:
// 2
// 4
// 7

```

Some may confusion because the variable `x` is the variable of function `returnAdder`. So when the function returns, why it's still can be accessed? Let's go through it step by step.

In the `returnAdder` function, the variable `x` is captured by reference because it's modified inside the closure (`x += y`). To keep the memory of `x` 'alive', the compiler allocates memory on the heap for `x` that will outlive the function's stack frame. Another heap allocation occurs for the closure structure itself:

```
struct { F uintptr; X0 *int }
```

This creates a closure object containing two fields: a function pointer (`F`) and a pointer to the captured variable (`X0 *int`).

In the `main` function, it has three separate calls to the closure. Each call operates on the same heap-allocated `x` variable through the pointer stored in the closure structure, which is why the value accumulates: first call returns `2` (1+1), second returns `4` (2+2), and third returns `7` (4+3).

To verify our assumption, let's run the static analysis:

```
# theanatomyofgo
...
./main.go:5:9: can inline main.returnAdder.func1 with cost 5
as: func(int) int { x += y; return x }
./main.go:13:15: inlining call to main.returnAdder.func1
./main.go:14:15: inlining call to main.returnAdder.func1
./main.go:15:15: inlining call to main.returnAdder.func1
./main.go:4:2: returnAdder capturing by ref: x (addr=false
assign=true width=8)
```

```
./main.go:5:9: func literal escapes to heap:  
...  
./main.go:4:2: x escapes to heap:  
...  
./main.go:4:2: moved to heap: x  
./main.go:5:9: func literal escapes to heap  
./main.go:12:22: main capturing by ref: x (addr=false  
assign=true width=8)  
./main.go:12:22: func literal does not escape
```

In addition to what we have discussed, the compiler can also inline closures in direct call scenarios where the target function is known at compile time. Exactly what we see in some first line of the output.

## Variadic Function Design

A function becomes "variadic" when its last parameter uses the `...` modifier. This tells the compiler to collect however many values are given in that spot (zero, one, or many) into a single slice.

Inside the function, you treat this parameter exactly like any other slice:

```
func sum(nums ...int) int {  
    total := 0  
    for _, v := range nums {  
        total += v  
    }  
    return total  
}  
  
sum(1, 2, 3, 4, 5)  
sum(1, 2)
```

In the `sum` function, `nums` is a slice of integers. You can use it like any other slice—iterate over it, access elements, and so on. The compiler creates the `nums` slice and passes it into the function.

Variadic functions are used widely in the standard library, such as `fmt.Println` or even in built-in functions like `append`:

```
func Println(a ...any) (n int, err error) {}  
  
func append(slice []Type, elems ...Type) []Type {}
```

If you already have a slice and want to pass it to a variadic function, you need to use a trailing ellipsis to "explode" the slice back into individual arguments:

```
var s = []int{1, 2, 3, 4, 5}
sum(s...)
```

The ellipsis is required here. If you leave it out, you pass the slice itself as a single argument of type `[]int`, which usually does not match the function's parameter list.

When you call a variadic function:

- The compiler packages any extra arguments into a new slice and passes that slice to the function. This means a new slice is created in memory and all arguments are copied into it.
- If you call the function with a slice and a trailing ellipsis, the compiler passes the slice directly into the function.

You might think the `func(slice...)` syntax unpacks or spreads out all the slice's elements, but that is not exactly true:

```
func sum(nums ...int) int {
    total := 0
    for _, v := range nums {
        total += v
    }

    nums[0] = 100 // Change the first element

    return total
}

func main() {
    s := []int{1, 2, 3, 4, 5}
    fmt.Println(sum(s...)) // 15
    fmt.Println(s) // [100 2 3 4 5]
}
```

Here, inside `sum`, we change the first element of the `nums` slice by setting `nums[0] = 100`. When we print the `s` slice in `main`, you see that the first element is now `100`. This shows that the `...` syntax does not just spread the elements; it passes the slice itself, so modifications are possible.

To understand this even better, look at the compiler code for how it checks and rewrites variadic function calls:

Figure 112. Variadic Function Call  
(cmd/compile/internal/typecheck/func.go)

```
// FixVariadicCall rewrites calls to variadic functions to
use an
// explicit ... argument if one is not already present.
func FixVariadicCall(call *ir.CallExpr) {
    fntype := call.Fun.Type()
    // If the function is not variadic, do nothing
    // If the call already has explicit ... syntax (IsDDD =
"is dot-dot-dot"), do nothing
    if !fntype.IsVariadic() || call.IsDDD {
        return
    }

    // vi = index of the variadic parameter (last parameter)
    // vt = type of the variadic parameter (e.g., []string
for func(args ...string)
    vi := fntype.NumParams() - 1
    vt := fntype.Param(vi).Type

    // `extra` holds all arguments that will go into the
variadic slice
    // For example, in fmt.Printf("Hello %s %s", "world",
"!"),
    // extra would be ["world", "!"]
    args := call.Args
    extra := args[vi:]

    // MakeDotArgs creates a slice literal for the `extra`
arguments
    // It creates []T{arg1, arg2, ...} where T is the
element type of the variadic parameter
    slice := MakeDotArgs(call.Pos(), vt, extra)
    for i := range extra {
        extra[i] = nil // allow GC
    }

    // Replace the extra arguments with the slice
    // Mark the call as having explicit ... syntax
    call.Args = append(args[:vi], slice)
    call.IsDDD = true
}
```

If the function being called is variadic and the arguments are already provided as a slice (with `call.IsDDD` set), the function does nothing. If not, it packages all extra arguments into a new slice using composite literals. When no arguments are passed, a `nil` slice is created—not an empty slice. This creation happens in the `MakeDotArgs` helper.

## The init Function

The `init` function is one of only two special functions in Go (the other is `main`). While it is often avoided in everyday code, understanding how `init` works is useful. You can have more than one `init` function in a package, or even in a single file. The details behind this are explained in Chapter 5:

```
package main

func init() {
    println("Init 1 called")
}

func init() {
    println("Init 2 called")
}

func main() {
    println("Main called")
}

// Init 1 called
// Init 2 called
// Main called
```

In this example, there are two `init()` functions, and both are called before `main()`. The process of collecting these `init()` functions and deciding the order in which they run is handled at compile time.

Each package in Go may depend on other packages. A package is only initialized after all its dependencies have been initialized. Also, before any `init()` function is called, all the package-level variables must be set up.

So, if you declare a package-level variable with an initial value, that variable is set before any function runs, including any `init()` functions in that package:

```
var x int = 10

func init() {
    println(x)
}

// 10
```

After all global variables are set up, the `init()` functions run. The main purpose of `init()` is to do any setup or preparation needed before the package is used. If

a package has more than one `init()` function, they are run in the order they appear in the source code.

This should give you a clear idea of how functions, including special ones like `init`, work in Go. More details about what happens with function stack frames and the inner mechanics of function calls will be covered in chapters that focus on memory.

## 3. Defer: Patterns and Performance

### The Role of Defer

The `defer` statement lets you schedule a function call to run right **before** the current function returns:

```
func main() {
    defer fmt.Println("World")
    fmt.Println("Hello")
}

// Hello
// World
```

The `defer` statement is useful for cleanup tasks, like closing files or releasing resources. These cleanup actions will run no matter how your function exits. Without `defer`, you would have to execute the cleanup code before every `return` statement:

#### Without defer

```
func do(b1, b2 bool) int
{
    if b1 {
        cleanup()
        return 1
    }

    if b2 {
        cleanup()
        return 2
    }

    cleanup()
}
```

#### With defer

```
func do(b1, b2 bool) int
{
    defer cleanup()

    if b1 {
        return 1
    }

    if b2 {
        return 2
    }

    return 0
}
```



The second version is much cleaner and easier to read. The `defer` statement also makes sure the cleanup runs even if the function exits early because of a panic:

```
func main() {
    defer fmt.Println("cleanup called")
    panic("panic")
}

// cleanup called
// panic: panic
```

`panic` and `recover` will be explained in the next section.

Under the hood, Go puts each `defer` call on a stack. When the function returns, deferred functions are called in reverse order. This is known as "last in, first out" (LIFO) order.

The last function you defer is the first one that runs:

```
func main() {
    defer fmt.Println("Hello,")
    defer fmt.Println("World!")
}

// World!
// Hello,
```

The deferred function runs later, but its arguments are evaluated right away, at the moment you write the `defer` statement. This can be surprising:

```
func main() {
    result := Result{}
    defer logBasedOnResult(result)

    result.IsSuccess = true
}

func logBasedOnResult(result Result) {
    if result.IsSuccess {
        println("Success")
    } else {
```

```

        println("Failure")
    }
}

// Output:
// Failure

```

The output will always be `"Failure"`, no matter what you set later. This is because the value of `result` is captured at the time of the `defer` statement.

This also happens with method receivers, which adds another layer of confusion:

```

func main() {
    result := Result{}
    defer result.Log()

    result.IsSuccess = true
}

func (r Result) Log() {
    if r.IsSuccess {
        println("Success")
    } else {
        println("Failure")
    }
}

// Output:
// Failure

```

This works the same way because the receiver is really just an argument that is copied at the time of the `defer` statement. In this case, you can fix the issue by using a pointer receiver:

```

func (r *Result) Log() {
    if r.IsSuccess {
        println("Success")
    } else {
        println("Failure")
    }
}

// Output:
// Success

```

There are two main ways to make sure your deferred function sees the latest value of a variable. Both require that you have access to the function you want to defer:

```

func print(i *int) {
    println(*i)
}

func main() {
    x := 10

    // Solution 1: Use a pointer
    defer print(&x)

    x = 20

    // Solution 2: Use a closure
    defer func() {
        print(&x)
    }()
}

x = 30
}

```

```

30
30

```

The second solution uses a closure, which captures `x` by reference. As a result, when `x` changes, the closure will see the new value. That is why it prints `30`.

The `defer` statement is always tied to the surrounding function, not to any smaller scope. A common mistake is putting a `defer` inside a loop and expecting it to run at the end of each loop iteration. Actually, all the deferred calls will run only when the function ends:

```

func main() {
    for i := 0; i < 3; i++ {
        defer fmt.Println("Inside loop", i)
    }

    {
        defer fmt.Println("Inside block")
    }

    fmt.Println("Done")
}

```

```

Done
Inside block
Inside loop 2
Inside loop 1
Inside loop 0

```

This example shows that all deferred calls in the loop and in the block run at the end of the `main` function, in reverse order.

This is important if you open many files in a loop and use `defer` to close them. If you use `defer` in a loop, all files will stay open until the whole function ends, which can be a problem if you have many files. A better way is to use a wrapper function or an anonymous function to handle opening and closing the file:

### Without Wrapper

```
func processFiles(files
[]string) error {
    for _, file :=
range files {
        f, err :=
os.Open(file)
        if err !=
nil {
            return err
        }
        defer
f.Close()
        //
Process the file
    }
    return nil
}
```

### With Wrapper

```
func processFiles(files
[]string) error {
    for _, file :=
range files {
        //
// Use an
anonymous function for
each file
        func(file
string) {
            f, err := os.Open(file)
            if err != nil {
                return
            }
            defer f.Close()
            //
Process the file
        }(file)
    }
    return nil
}
```

`f.Close()` can return an error, and you should handle it, for example by logging or returning it. Here we skip error handling to focus on the main idea.

The first example waits to close all files until the end of the function, which can use up many resources if there are many files. The second example closes each file as soon as you are done with it, which is safer and more efficient.

Now, if the `defer` statement runs right before the function returns, can it change the return value? The answer is absolutely yes, if you use a named return value:

```

func doSomething() (res int) {
    defer func() {
        res = 100
    }()
    return 0
}

func main() {
    result := doSomething()
    fmt.Println("After doSomething", result)
}

// Output:
// After doSomething 100

```

This works because the deferred function can update the named return value before the function actually returns. This trick can be useful, and you will see more in the section about panic recovery.

## How Defer Is Implemented

When you think about how `defer` works, it might seem like the Go compiler could just move the deferred function call to the end of your function, right before every `return` statement. It sounds like a simple compile-time trick with no extra cost, right?

### Go Source Code

```

func main() {
    defer println(1)
    defer println(2)
    defer println(3)
    println(4)
}

```

### Compiler Transformation

```

func main() {
    println(4)
    println(3)
    println(2)
    println(1)
}

```

For the simplest cases, such as this example with several `defer println(...)` calls at the top level, this mental model generally works for understanding the output. However, it is not how `defer` actually works, especially when things become more complex. Specifically, Go requires additional runtime support for `defer`.

One of the challenges is that the compiler often cannot know in advance which deferred calls need to run or how many times `defer` will be called. For example, consider this conditional `defer`:

```
func do(b1 bool) {
    if b1 {
        defer fmt.Println("b1")
    }

    defer fmt.Println("b2")
}
```

Here, `defer fmt.Println("b1")` only happens if `b1` is true at runtime. The compiler cannot know if that will happen. It cannot just paste `fmt.Println("b1")` at the end, since it might not be needed at all. But it also cannot ignore it, because if `b1` is true, it must run before the function returns.

This makes it impossible for the compiler to simply transform all defers at compile time. Go needs to manage defers at runtime. Another key point is that the arguments to a deferred function are always evaluated right away, as soon as the `defer` statement is hit. They are not delayed until the deferred function runs.

So, instead of only using the compiler, Go uses a runtime system to keep track of deferred calls. This mainly relies on the goroutine's stack and special runtime functions.

When the program hits a `defer` statement during execution, it does not schedule the function call like a timer would. Instead, it creates a record, called a `_defer` record, to remember what needs to be done later:

Figure 113. The Defer Record (src/runtime/runtime2.go)

```
type _defer struct {
    heap      bool // Stack or heap allocation flag.
    rangefunc bool // Special range-over-func defer
flag.
    sp       uintptr // Stack pointer when deferred.
    pc       uintptr // Program counter when deferred.
    fn       func() // Deferred function to execute.
    link     *_defer // Next defer in LIFO list.

    // If rangefunc is true, *head is the head of the
atomic linked list
    // during a range-over-func execution.
    head *atomic.Pointer[_defer] // Head of atomic
```

```
rangefunc list.  
}
```

This record keeps all the information needed to run the deferred function later and also tracks the order of execution. There are three important fields to notice:

`link`, `fn`, and `heap`. We will look at them one by one, starting with `link`. The `link` field points to the next `_defer` record, creating a simple linked list:

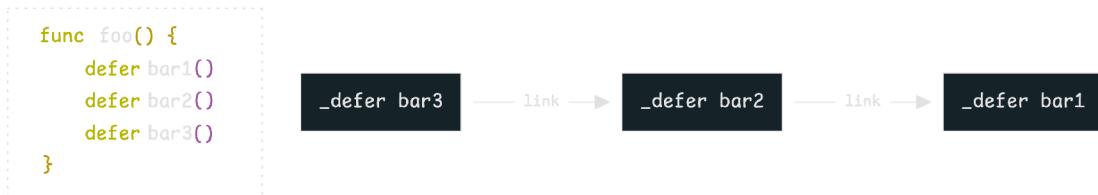


Illustration 192. Deferred calls managed as a linked list

Each goroutine holds the most recent `_defer` record for the current function frame:

Figure 114. Goroutine Representation (src/runtime/runtime2.go)

```
type g struct {  
    ...  
    _defer     *_defer // innermost defer  
}
```

It may seem odd that deferred functions are grouped by goroutine, as the defers for the current function frame are executed when a function returns. However, function return is not the only way defers get triggered. If a panic happens, all defers in the current goroutine stack are also executed.

This structure forms a singly linked list of `_defer` records for the entire call stack. The goroutine's `_defer` field acts as the head of this list:

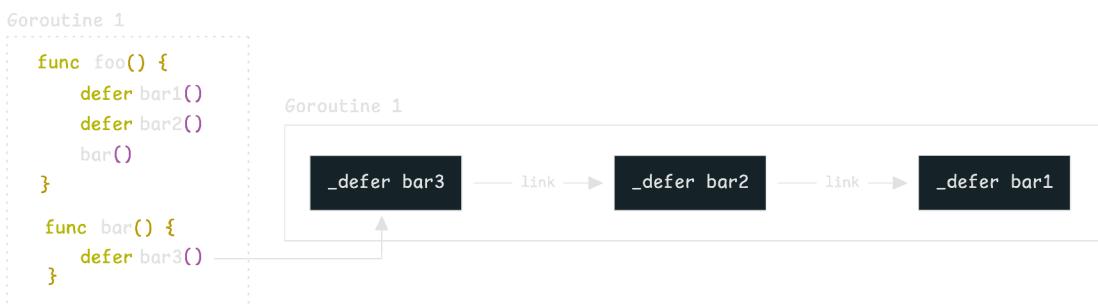


Illustration 193. All active defers linked in current goroutine

When a new `defer` statement is hit, a new `_defer` record is created. Its `link` field points to the current head (`g._defer`), and then `g._defer` is updated to point to the new record.

The next key part is the `fn func()` field. This holds the deferred function to call later. But you may notice that it has no arguments and no return values. How does this work if you defer a function that takes arguments?

```
func main() {
    defer func(a, b int) int {
        return a + b
    }(1, 2)
}
```

The answer is that the Go compiler uses closures and a process called "normalizing" the function.

When you write something like `defer myFunc(arg1, arg2)`, the compiler does not simply store the function and its arguments to use later. Instead, it does a transformation:

1. The Go compiler immediately evaluates the arguments `arg1` and `arg2` when the `defer` statement is hit. This is why deferred arguments get their values right away, not when the deferred function is finally run.
2. The compiler creates a new, hidden anonymous function (closure) on the fly. This closure does not take any arguments.
3. This closure "captures" the values of `arg1` and `arg2`, which were just evaluated.
4. The closure simply calls the original function using the captured values.

In simple terms, the compiler turns your defer call into something like this:

### Go

```
defer Hello(a, b)
```

### Pseudo Go

```
a1 := a
b1 := b
defer func() {
    Hello(a1, b1)
}()
```

You may have seen names like this closure in stack traces or when debugging Go code. The naming usually follows this pattern:

```
<OuterFunctionName>.deferwrap<N>
```

- `<OuterFunctionName>` is the function where the defer statement lives.
- `.deferwrap` shows it is a wrapper for a defer.
- `<N>` is a number, starting from 1, used if there are multiple defers.

You will not use these names yourself, but sometimes they show up in stack traces, profiling, or debugging tools that show symbols made by the compiler. Knowing what they mean can be useful.

There are three kinds of defer calls in Go: open-coded defer, stack defer, and heap defer. You can see this if you look at how Go handles the `ODEFER` node in the SSA phase when it turns IR nodes into SSA form:

Figure 115. Calling stmt `defer`  
(src/cmd/compile/internal/ssagen/ssa.go)

```
// stmt converts the statement n to SSA and adds it to s.
func (s *state) stmt(n ir.Node) {
    ...
    switch n.Op() {
    ...
    case ir.ODEFER:
        n := n.(*ir.GoDeferStmt)
        if base.Debug.Defer > 0 {
            var defertype string
            if s.hasOpenDefers {
                defertype = "open-coded"
            } else if n.Esc() == ir.EscNever {
                defertype = "stack-
allocated"
            } else {
                defertype = "heap-allocated"
            }
            base.WarnfAt(n.Pos(), "%s defer",
defertype)
        }
        if s.hasOpenDefers {
            s.openDeferRecord(n.Call.
(*ir.CallExpr))
        } else {
            d := callDefer
            if n.Esc() == ir.EscNever &&
n.DeferAt == nil {
```

```

                d = callDeferStack
            }
            s.call(n.Call.(*ir.CallExpr), d,
false, n.DeferAt)
}
...
}
...
}

```

The priority and performance of these `defer` types are clear: open-coded `defer` is the fastest, followed by stack-allocated `defer`, and finally heap-allocated `defer`. The main difference between stack-allocated and heap-allocated `defer` is where the `_defer` record is stored:

- Stack-allocated `defer` : The `_defer` record is stored right on the function's stack. This is faster because it avoids the extra cost of heap allocation and garbage collection.
- Heap-allocated `defer` : The `_defer` record is stored on the heap. This is used when the compiler cannot determine in advance how many `defer` statements will be created, such as when a `defer` appears inside a loop or a conditional.

## Heap-Allocated Defers

We will cover escape analysis in detail in the next chapter, but here is a preview of how the Go compiler decides whether a `defer` statement escapes to the heap or not:

Figure 116. goDeferStmt (src/cmd/compile/internal/escape/call.go)

```

func (e *escape) goDeferStmt(n *ir.GoDeferStmt) {
    k := e.heapHole()

    if n.Op() == ir.ODEFER && e.loopDepth == 1 && n.DeferAt
== nil {
        // Top-level defer arguments don't escape to the
        // heap,
        // but they do need to last until they're invoked.
        k = e.later(e.discardHole())

        // force stack allocation of defer record, unless
        // open-coded defers are used (see ssa.go)
        n.SetEsc(ir.EscNever)
    }
}

```

```
// ...
}
```

The key here is the condition `e.loopDepth == 1`. The compiler keeps a `loopDepth` counter, which starts at 1 because the `main` function body is considered depth 1, and increases when a loop is entered.

If a `defer` is at the top level of a function (when `loopDepth == 1`), the escape analysis sets its status to `EscNever`. This means the `_defer` record can be stack-allocated, which is more efficient. If a `defer` statement appears inside a loop, it is heap-allocated.

Here is an example of heap-allocated `defer`:

```
package main

func main() {
    for i := 0; i < 3; i++ {
        defer println("Defer in loop", i)
    }
}
```

To check if the `defer` is heap-allocated, you can build your program with a special debug flag:

```
go build -gcflags=-d=defer=1

# theanatomyofgo
./main.go:5:3: heap-allocated defer
```

Every heap-allocated `defer` results in a costly runtime call, `runtime.deferproc`. You can see this by looking at the assembly output:

```
0x0034 00052 MOVD      $type:noalg.struct { F uintptr; X0
int }(SB), R0
0x003C 00060 CALL      runtime.newobject(SB)
0x0040 00064 MOVD      $main.main.deferwrap1(SB), R1
0x0048 00072 MOVD      R1, (R0)
0x004C 00076 MOVD      main.i-8(SP), R2
0x0050 00080 MOVD      R2, 8(R0)
0x0054 00084 CALL      runtime.deferproc(SB)
0x0058 00088 CMP       $0, R0
0x005C 00092 BNE       100
0x0060 00096 JMP       32
0x0064 00100 CALL      runtime.deferreturn(SB)
```

```

0x0068 00104 LDP      -8(RSP), (R29, R30)
0x006C 00108 ADD      $48, RSP
0x0070 00112 RET      (R30)
...
0x0074 00116 CALL     runtime.deferreturn(SB)
0x0078 00120 LDP      -8(RSP), (R29, R30)
0x007C 00124 ADD      $48, RSP
0x0080 00128 RET      (R30)

```

The first step is to create a closure for the deferred function. As explained earlier, this means building a closure structure (called `funcval`). In the assembly code, this closure is represented by the register `R0`:

Assembly	Pseudo Code
<pre> MOVD \$type:naalg.struct{F uintptr;X0 int}(SB), R0 CALL runtime.newobject(SB) MOVD R1, (R0) MOVD main.i-8(SP), R2 MOVD R2, 8(R0) </pre>	<pre> obj = runtime.newObject(struct{F uintptr;X0 int}) obj.F = main.deferwrap1 obj.X = i </pre>

Illustration 194. Go closure created as heap-allocated object

This closure object holds a function pointer `F` that points to the wrapper function, and an argument field `X0` that stores the captured value of the loop variable `i` in this example. The Go runtime uses this information to create a closure object on the heap.

The highlighted lines in the assembly are two runtime calls: `CALL runtime.deferproc(SB)` and `CALL runtime.deferreturn(SB)`:

Assembly	Pseudo Code
<pre> CALL runtime.deferproc(SB) CMP \$0, R0 BNE 100 JMP 32 ► CALL runtime.deferreturn(SB) </pre>	<pre> r := runtime.deferproc(obj.F) if r != 0 {     runtime.deferreturn() } </pre>

Illustration 195. `runtime.deferproc` schedules a deferred function call

The `runtime.deferproc` function creates a `_defer` record that points to the wrapper function to run later when the function exits or if a panic happens. Here is a simplified version of how it works:

Figure 117. `deferproc` (src/runtime/panic.go)

```

func deferproc(fn func()) {
    gp := getg()
    if gp.m.curg != gp {
        throw("defer on system stack")
    }

    d := newdefer()
    d.link = gp._defer
    gp._defer = d
    d.fn = fn
    d.pc = getcallerpc()
    d.sp = getcallersp()

    return0()
}

```

First, the runtime gets the current goroutine (`getg()`). Then it makes a new `_defer` record (`newdefer()`). This new record is linked to the goroutine's defer chain using the `link` field. Each `_defer` points to the previous one, forming a linked list:

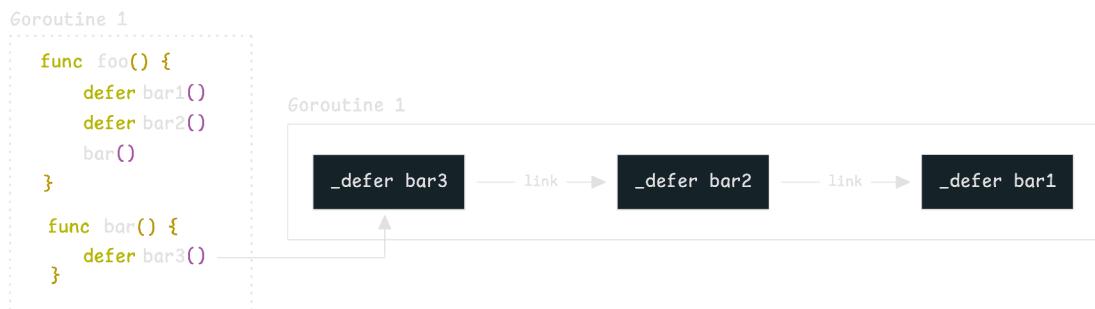


Illustration 196. All active defers linked in current goroutine

The `_defer` record also saves the program counter (`pc`) and stack pointer (`sp`) for the current function frame. These fields help the runtime handle deferred calls and panics correctly.

One subtle detail is that, even though `deferproc` is defined with no return value, the assembly code still checks a return value in register `R0` with `0` as the expected normal case:

Assembly	Pseudo Code
<pre> CALL  runtime.deferproc(SB) CMP   \$0, R0 BNE   100 JMP   32 → CALL runtime.deferreturn(SB) </pre>	<pre> r := runtime.deferproc(obj.F) if r != 0 {     runtime.deferreturn() } </pre>

Illustration 197. Assembly checks deferproc result for panic handling

If the return value is not `0`, this means the function is returning because of a panic, and the code jumps to `runtime.deferreturn`:

But how does this work if the Go signature for `deferproc` has no return value? This is where the special `return0()` function comes in.

Figure 118. deferproc — return0 (src/runtime/panic.go)

```

func deferproc(fn func()) {
    ...
    return0()
}

```

It is an assembly stub (defined in files like `runtime/asm_arm64.s`) that returns `0` in the machine register directly, without using the regular Go return system:

Figure 119. return0 (src/runtime/asm\_arm64.s)

```

TEXT runtime.return0(SB), NOSPLIT, $0
    MOVW   $0, R0
    RET

```

This trick makes the function seem to return a value at the assembly level while keeping a void signature at the Go source level. The compiler generates code to check this hidden return value. If `deferproc` returns `0`, execution continues as usual. If it returns `1` (which happens during panic recovery), execution jumps to the end of the function, where `runtime.deferreturn` is called, skipping the rest of the function body:

## Why doesn't `deferproc` just return a value in the normal Go way?

At the language level, a defer call should be invisible. It should look like a regular statement, not something that returns a value you could use or ignore. Internally, the runtime needs a way to communicate a single bit of information: did a panic get recovered? To do this, the runtime sets a value directly in the first return register using a small assembly stub (`return0`). This stub puts zero in the register and returns before any other code can change it:

```
TEXT runtime·return0(SB), NOSPLIT, $0
    MOVW    $0, R0
    RET
```

If a panic is recovered, the runtime later changes that register to one and resumes execution just after the original `CALL`. It looks like the call returned one, even though the body of `deferproc` did not run again. If `deferproc` had an int return in the Go signature, callers would have to handle the return value, and it would expose an implementation detail Go wants to keep hidden. Using the assembly stub keeps the language clean while still giving the runtime the control it needs.

Now it should be clear that `runtime.deferproc` is the function responsible for creating and linking a `_defer` record to the current goroutine's defer chain.

On the other side, `deferreturn` is the partner function that handles executing the `_defer` records in the reverse order they were created. This function is inserted into the code (when you have a `defer` statement) in two places:

1. At the normal exit points of a function, like the end of the function or after a `return` statement.
2. Right after every `runtime.deferproc` call, as explained earlier.

So, the example from before:

```
func main() {
    for i := 0; i < 3; i++ {
        defer println("Defer in loop", i)
    }
}
```

can be rewritten like this:

```
func main() {
    for i := 0; i < 3; i++ {
        obj := runtime.newobject(
            struct{ F uintptr; X0 int }{})
        obj.F = deferwrap1
```

```

        obj.X0 = i
        r := runtime.deferproc(obj.F)
        if r != 0 {
            runtime.deferreturn()
            return
        }
    }
    runtime.deferreturn()
}

```

You might wonder if this means there are two allocations each time: one for the closure object and one for the `_defer` record. Is this inefficient?

The answer is no. The Go runtime has a typical pattern for caching frequently used items. It uses a per-processor pool.

The MPG model was explained at the beginning of this section in the Prerequisites.

## Per-Processor Pooling (Defer Pool)

Each logical processor (`P`) has its own local cache of `_defer` structures, which is called a per-processor pool (`p.deferpool`).

```

type p struct {
    ...
    deferpool []*_defer
    deferpoolbuf [32]*_defer
    ...
}

```

When a new `_defer` is needed, the runtime first checks the local pool (`newdefer()` function).

This per-processor caching avoids synchronization overhead when allocating and freeing `_defer` records. Each `P` can access its own pool without needing locks. This is one of the main advantages of per-processor pools. There is no contention, since only one thread accesses each pool at a time, following the MPG model.

When a `P`'s local pool is empty, it fills up by pulling multiple `_defer` records from the global pool (`sched.deferpool`) in the runtime scheduler:

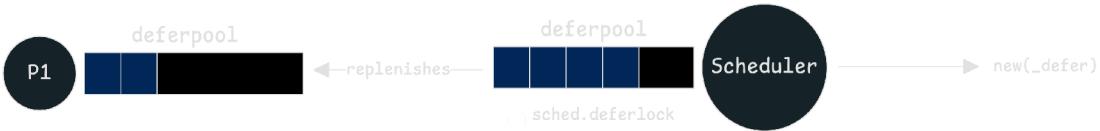


Illustration 198. Local pool replenishes from global pool

Pulling from the global pool does require a global lock, since multiple threads might try to access it at the same time. To keep lock contention low, the local pool grabs half of its capacity worth of defers (`cap(pp.deferpool)/2`), which is 16 elements in this Go version, from the global pool in one go, then quickly releases the lock:

```

func newdefer() *_defer {
    var d *_defer

    // Get the current processor for this goroutine
    mp := acquirem()
    pp := mp.p.ptr()

    // Check if the local pool is empty, and try to
    // refill from the global pool
    if len(pp.deferpool) == 0 && sched.deferpool != nil {
        ...
        // Move half the defers from global pool to
        // local pool
        for len(pp.deferpool) < cap(pp.deferpool)/2
        && sched.deferpool != nil {
            d := sched.deferpool
            sched.deferpool = d.link
            d.link = nil
            pp.deferpool = append(pp.deferpool,
d)
        }
        ...
    }

    // Pop a defer from the local pool
    if n := len(pp.deferpool); n > 0 {
        d = pp.deferpool[n-1]
        pp.deferpool[n-1] = nil
        pp.deferpool = pp.deferpool[:n-1]
    }

    // Release the M.
    releasem(mp)
    mp, pp = nil, nil

    // If the pool is still empty, allocate a brand new

```

```

defer from the heap
if d == nil {
    d = new(_defer)
}

// Mark it as heap allocated
d.heap = true
return d
}

```

If both the local and global pools are empty, then as a last resort, the runtime allocates a fresh `_defer` from the heap (`new(_defer)`). Heap allocation is more expensive and adds pressure to garbage collection, so Go only does this when there are no defers to reuse.

But who fills up the global pool? The answer is: the local pool gives back!

If a P's local pool gets too full (more than 32 elements), about half of its `_defer` records are moved back to the global pool. This keeps local pools from growing without limit but still leaves enough defers for fast reuse:

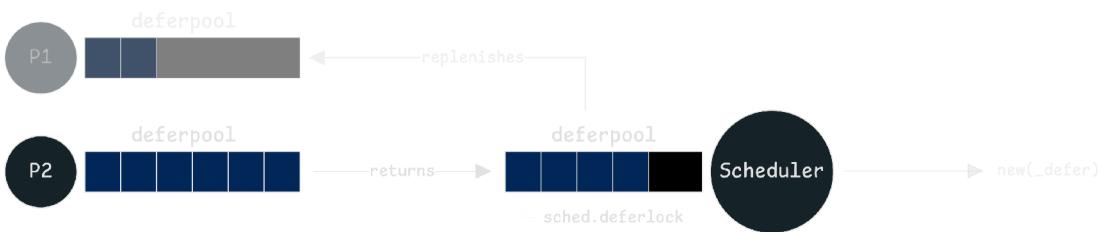


Illustration 199. Local defer pools replenish back to the global defer pool

If your program creates a lot of defers in a burst, and the local pools keep moving them back to the global pool, the global pool can also grow a lot. How does Go prevent the global pool from growing forever?

That is where the garbage collector comes in. At the start of every garbage collection cycle, it unlinks the whole global list, clears the `link` fields, and drops the head pointer to `nil`. From the garbage collector's perspective, the whole batch is now regular garbage and will be freed in the next sweep phase.

So, even if the global pool grows quickly in a single GC cycle (for example, if your code creates a ton of defers in a short time), it is guaranteed to be emptied once per GC cycle. This makes sure there is never unbounded retention of `_defer` objects.

## Executing Deferred Calls (Defer Return)

Now let's look at what happens when a function containing a `defer` statement returns. Each `deferproc` call has a matching `runtime.deferreturn` call. Right before the function finishes, `runtime.deferreturn` is called.

The purpose of `deferreturn` is to remove `_defer` structs from the goroutine's defer chain and execute them one by one, starting from the most recent (the head of the linked list) and continuing down:

Figure 120. `deferreturn` (`src/runtime/panic.go`)

```
func deferreturn() {
    var p _panic
    p.deferreturn = true

    p.start(getcallerpc(),
unsafe.Pointer(getcallersp()))
    for {
        fn, ok := p.nextDefer()
        if !ok {
            break
        }
        fn()
    }
}
```

But how does the runtime know which `_defer` records belong to the current function, especially since all `_defer` records are managed per goroutine?

Every time a `defer` is created, the runtime saves the stack pointer of the caller: `d.sp = getcallersp()`. Imagine a function `foo` calls a function `bar`, and there is a `defer` in `bar`:

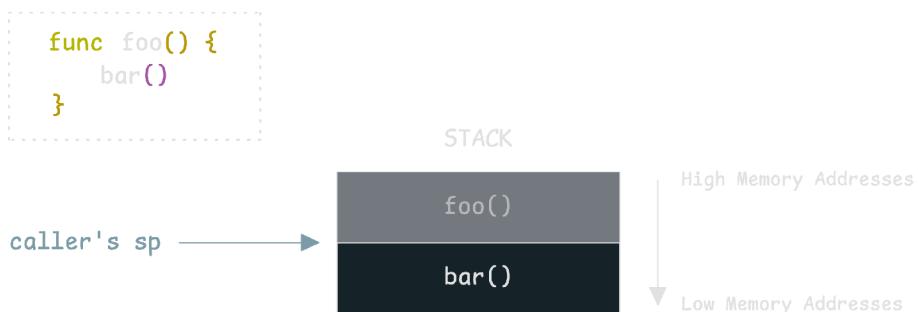


Illustration 200. Caller's stack pointer identifies defer ownership

The caller's stack pointer (`sp`) acts as a unique identifier for the callee frame (`bar`). When the function is about to return, the runtime checks which `_defer` records have an `sp` matching the current frame. Only these defers are executed.

The runtime also keeps a single `_panic` structure to manage all code paths that need to run deferred functions: this includes normal returns, explicit panic, and `Goexit`. The `_panic` struct tracks enough state (including whether the code is panicking or just doing a normal return via `deferreturn`) to allow the unwinder to walk the stack and invoke each frame's `_defer` in the right order.

The method `p.nextDefer()` (shown above) also recycles the `_defer` record back into the processor's pool after the deferred call is finished. It returns `false` when there are no more `_defer` records for the current stack frame.

## Stack-Allocated Defers

Heap-allocated defers have a cost: they may require heap allocation and add extra overhead for cache management.

But `defer` is built into Go, and we use it everywhere. In high-concurrency systems where thousands or millions of goroutines might be running, the extra cost of heap allocations adds up quickly. To solve this, starting with Go 1.13, the Go team introduced "stack-allocated defers" as an optimization.

How can you write code that produces a stack-allocated defer, without the compiler optimizing it away to an open-coded defer or falling back to heap allocation?

The most reliable way for testing is to use the `-N` compiler flag to turn off open-coded defer optimizations, and then write simple `defer` statements. But this is just for testing, not a real-world solution. In practice, there is a trick: if a function has at least one heap-allocated defer, then any other simple defer statements in that function will also not use open-coded defer. Instead, they will be stack-allocated.

So, you can force a stack-allocated defer by putting a simple `defer` before a heap-allocated defer in the same function:

Figure 121. Stack-Allocated Defer

```
//go:noinline
func printsmt() {
    println("Oh hey! I'm stack-allocated deferred!")
}

func main() {
    // Stack-allocated defer
    defer printsmt()

    // Heap-allocated defers
    for i := 0; i < 3; i++ {
        defer println(i)
    }
}
```

If your deferred function does not take arguments or return values (like `printsmt()`), this is the simplest case for the Go `defer` mechanism. This is actually the best-case scenario.

The Go compiler recognizes this situation. It does not need to create any wrapper function or closure to capture arguments. The runtime can call `deferproc` directly with the function pointer, with no extra work.

This lets us see the assembly code for stack-allocated defers very clearly, with no noise from extra closures or wrappers:

Assembly	Pseudo Code
<pre>MOVD \$main.printsm...f(SB), R1 MOVD R1, main..autotmp_3-24(SP) MOVD \$main..autotmp_3-48(SP), R0 CALL runtime.deferprocStack(SB) CMP \$0, R0 BNE 60 ... CALL runtime.deferreturn(SB)</pre>	<pre>R1 = printsm... temp.fn = R1 R0 = &amp;temp R0 = runtime.deferprocStack(R0) if R0 != 0 {     runtime.deferreturn }</pre>

Illustration 201. `deferprocStack` handles simple defers on stack

Here, the compiler creates the `_defer` record directly on the stack frame of the function, instead of allocating it on the heap. This stack-allocated `_defer` only needs the function pointer. To fill in the stack pointer, program counter, and other fields, the compiler calls the `runtime.deferprocStack` function, passing a pointer to the stack-allocated `_defer` record:

Figure 122. `deferprocStack` Function (src/runtime/panic.go)

```

func deferprocStack(d *_defer) {
    gp := getg()
    if gp.m.curg != gp {
        throw("defer on system stack")
    }

    d.heap = false
    d.rangefunc = false
    d.sp = getcallersp()
    d.pc = getcallerpc()

    (*(*uintptr)(unsafe.Pointer(&d.link)) =
uintptr(unsafe.Pointer(gp._defer))
    (*(*uintptr)(unsafe.Pointer(&d.head)) = 0
    (*(*uintptr)(unsafe.Pointer(&gp._defer)) =
uintptr(unsafe.Pointer(d))

    return0()
}

```

After this point, the process works just like with heap-allocated defer records, except that stack-allocated defers are not added to the defer pool. Instead, because they are created directly on the goroutine's stack, they are automatically cleaned up when the stack is unwound.

## Open-Coded Defers

Earlier we talked about the most basic way to implement a `defer` statement: just place the deferred function call right at the end of the function. Open-coded `defer` follows this idea very closely.

Open-coded defers are an optimization introduced in Go 1.14 that can remove the runtime overhead of defers in many cases. Instead of creating `_defer` records at runtime for each defer, the compiler puts the deferred function calls directly at every exit point in the function.

However, for a `defer` to be eligible for open-coded optimization, several rules must be met. These rules can change as Go evolves, but the main idea is that the defer situation must be simple and not involve extra complexity:

1. If you specify the `-N` flag in `go tool compile` or `go build`, most optimizations are turned off, including open-coded `defer`.
2. The function with `defer` statements must not have heap-allocated result parameters.

Caller and callee use the stack (or registers) to pass arguments and return results. If a function returns heap-allocated results, the result must be copied back to the stack slot (or register) at every function exit, which is similar to what open-coded defer does. This adds extra steps for each exit.

This is handled when the compiler generates SSA code from IR nodes:

Figure 123. open-coded defers are disabled when a function has heap-allocated result parameters  
(src/cmd/compile/internal/ssagen/ssa.go)

```
if s.hasOpenDefers {
    // Skip if there are any heap-allocated result
    // parameters that need to be copied back to their
    stack slots.
    for _, f := range s.curfn.Type().Results() {
        if !f.Nname.(*ir.Name).OnStack() {
            s.hasOpenDefers = false
            break
        }
    }
}
```

3. If the product of the number of return points and the number of defers in the function is greater than `15` (that is, `s.curfn.NumReturns * s.curfn.NumDefers > 15`), open-coded defers are skipped. This is because the compiler must inline the defer call at every exit, and too many combinations create bloated code.
4. The number of defers in the function (from the number of `ODEFER` nodes in the AST) cannot exceed 8.

Open-coded defers are for simple functions, so this limit is set at 8. But, why 8?

If a function has many defers, Go can't know at compile time which ones will actually run:

```
func do() {
    if condition {
        defer func() {
            println("Defer 1")
        }()
    }

    defer func() {
        println("Defer 2")
    }
}
```

```
        }()
}
```

Go uses a single `uint8` bitmap, called `deferBits`, to keep track of all the `defer` statements in the function. Each defer gets a bit in the bitmap:

- The first `defer` uses bit `0` (least significant bit)
- The second `defer` uses bit `1`
- The third `defer` uses bit `2`, and so on up to bit `7` (the most significant bit)

A `uint8` only has 8 bits, so only 8 defers can be tracked with open-coded defers. If there are more, the compiler turns off open-coded defers and falls back to the normal runtime `defer` list.

When the function returns, Go checks `deferBits` and runs the appropriate deferred functions in reverse order.

5. If a function has even one heap-allocated `defer`, open-coded defers are not used for any other defers in the same function.

Figure 124. walkStmt ODEFER node  
(src/cmd/compile/internal/walk/stmt.go)

```
func walkStmt(n ir.Node) ir.Node {
    ...
    switch n.Op() {
    ...
    case ir.ODEFER:
        n := n.(*ir.GoDeferStmt)
        ir.CurFunc.SetHasDefer(true)
        ir.CurFunc.NumDefers++
        if ir.CurFunc.NumDefers > maxOpenDefers
        || n.DeferAt != nil {
            ir.CurFunc.SetOpenCodedDeferDisallowed(true)
        }
        if n.Esc() != ir.EscNever {
            ir.CurFunc.SetOpenCodedDeferDisallowed(true)
        }
        ...
    }
    ...
}
```

Now that we know each `defer` site is assigned a bit position at compile time, while the function is executing, each time control reaches a `defer` statement the compiler emits two kinds of instructions:

- One that stores the function pointer into a fixed stack slot reserved for that defer.
- One that sets the corresponding bit in `deferBits` with an `OR` operation.

Let's take an straightline example:

```
func main() {
    defer func() { println("defer 1") }()
    defer func() { println("defer 2") }()
    defer func() { println("defer 3") }()
    println("Hello, World!")
}
```

In this example, after the first `defer` runs the byte holds `0000 00012` (decimal 1). After the second `defer` runs it holds `0000 00112` (decimal 3). After the third `defer` runs it holds `0000 01112` (decimal 7). Look at the assembly code generated made this clearer:

### Go Assembly

```
00040 MOVD
$main.main.func1.f(SB),
R0
00048 MOVD      R0,
main..autotmp_1-24(SP)
00052 MOVD      $1, R0
00056 MOVB      R0,
main..autotmp_0-25(SP)
```

### Pseudo Go

```
R0 = &main.main.func1.f
main..autotmp_1-24 = R0
R0 = 1
main..autotmp_0-25 = R0
```

```
00060 MOVD
$main.main.func2.f(SB),
R1
00068 MOVD      R1,
main..autotmp_2-16(SP)
00072 MOVD      $3, R1
00076 MOVB      R1,
main..autotmp_0-25(SP)
```

```
R1 = &main.main.func2.f
main..autotmp_2-16 = R1
R1 = 3
main..autotmp_0-25 = R1
```

## Go Assembly

```
00080 MOVD  
$main.main.func3·f(SB),  
R2  
00088 MOVD      R2,  
main..autotmp_3-8(SP)  
00092 MOVD      $7, R2  
00096 MOVB      R2,  
main..autotmp_0-25(SP)
```

## Pseudo Go

```
R2 = &main.main.func3·f  
main..autotmp_3-8 = R2  
R2 = 7  
main..autotmp_0-25 = R2
```

The function pointer gets stored in the stack frame, while the bitmask keeps track of the deferred functions. You can see the bitmask values are set to 1, 3, and 7, corresponding to each function's `defer` slot. Each bit in the bitmask indicates whether a `defer` slot is occupied.

At every non-panic exit the compiler emits in-line code that walks the bits from high to low. For every bit that is still set it clears the bit and calls the stored function pointer. With three defers the epilogue looks like this (simplified):

```
# Call func3  
00124 MOVD      $3, R0 # clear bit 2: 7 &^ (1<<2) == 3  
00128 MOVB      R0, main..autotmp_0-25(SP)  
00132 CALL      main.main.func3(SB)  
  
# Call func2  
00136 MOVD      $1, R0 # clear bit 0: 3 &^ (1<<0) == 1  
00140 MOVB      R0, main..autotmp_0-25(SP)  
00144 CALL      main.main.func2(SB)  
  
# Call func1  
00148 MOVB      ZR, main..autotmp_0-25(SP)  
00152 CALL      main.main.func1(SB)  
  
# Ending  
00156 LDP      -8(RSP), (R29, R30)  
00160 ADD      $64, RSP  
00164 RET      (R30)
```

The high-to-low sweep enforces the usual LIFO order: the most recently registered defer runs first. However, this example is so straightforward that the compiler can hardcode the bits 1, 3 and 7 and call them directly.

In a more complex case, if a `defer` is hidden behind a branch the compiler cannot know at compile time whether its bit should be set or not. It uses different `if` look-alike instructions to set the bit.

## 4. Panic & Recover

### Usage Scenarios

`panic` and `recover` are two built-in functions in Go used for handling exceptional situations that cannot be managed through regular error handling.

Use `panic` when something goes unexpectedly wrong. These are problems you usually do not handle with normal error checks, such as dividing by zero, dereferencing a `nil` pointer, accessing an out-of-range index, or a bug you believe should never happen:

```
func main() {
    a := 0
    fmt.Println(1/a)
}

"runtime error: integer divide by zero"
```

As mentioned in Chapter 2, only integer division by zero causes a panic. Floating-point division by zero gives `+Inf` or `-Inf` depending on the sign.

When you call `panic()`, normal execution stops, and Go starts "unwinding" the stack. This means it goes back through the call stack and runs all deferred functions in the same goroutine. We saw this with `defer return`; a `_panic` record is created and the `defer` chain is walked.

Panicking continues running deferred functions in last-in-first-out order until it finds a deferred function that calls `recover()`:

```
func A() {
    println("A called")
}

func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
    }
}
```

```

        }()
        defer A()

        fmt.Println("Starting the program")
        panic("A severe error occurred")
        fmt.Println("This will not be printed")
    }

// Output:
// Starting the program
// A called
// Recovered: A severe error occurred

```

In this example, a deferred function is set up to call `recover`. After printing "*Starting the program*", a panic is triggered with the message "*A severe error occurred*".

Go immediately stops regular execution and starts to unwind the stack, running all defers. The deferred function containing `recover` is called, and it receives the panic value. Since the value is not `nil`, the program prints "*Recovered: A severe error occurred*". Execution continues after the `defer` block, so the last `fmt.Println` is skipped.

A key point: after the panic is recovered, the function completes by running any other deferred functions, then exits normally. The original flow does not continue from the panic point.

What about the value the function returns after a panic is recovered?

```

func add(a, b int) int {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()
    panic("oops")
    return a + b
}

```

Logically, when the panic is recovered, the function can't keep running because something failed at the panic point. The function stops and returns the default value of its type.

This can be risky: for example, `a + b` is never reached, so the function returns `0` (the default for `int`). While this example is simple, it shows a situation that can cause bugs in real programs. Some might want to use panic as an error to return to the caller, and this is possible with named return values:

```
func add(a, b int) (res int, err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("panic: %v", r)
        }
    }()
    res = a + b
    panic("oops")
    return
}
```

Now, the panic is both recovered and converted to an error, so the caller can handle the failure instead of the program failing silently.

Another interesting detail is that the value you pass to `panic(...)` and receive from `recover()` can be anything. This is because the argument is an empty interface (`interface{}`), so you are not limited to errors. Many people assume you always use an `error` value, but you can use strings, numbers, structs, or anything else.

For example, in our earlier code, the value inside the `panic(...)` call would look like `eface{ type: string, value: &"A severe error occurred" }` internally.

What happens if you pass `nil` to `panic(...)`?

```
defer func() {
    if r := recover(); r != nil {
        fmt.Println("Recovered:", r)
    }
}()
panic(nil)
```

Will the `if` condition `if r := recover(); r != nil` be true? The answer depends on your Go version.

Before Go 1.21, if you called `panic(nil)` and recovered from it, the value returned by `recover()` was also `nil`. This led to confusion, because there was

no way to tell the difference between:

- No panic at all (normal return from `recover`),
- A panic that was triggered with a `nil` value.

This made it tricky to detect if an actual panic had happened, or if `recover()` just returned normally.

Starting in Go 1.21, this changed. If you call `panic(nil)`, the runtime automatically wraps the `nil` value in a new type called `PanicNilError`. This error has a clear message: "*panic called with nil argument*".

## What the Runtime Does on Panic

Let's set up an example to illustrate how `panic` works in the rest of this section:

```
func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()
    defer func() {
        fmt.Println("defer but not recovered")
    }()
    panic("Unexpected wrong")
}
```

The call to `panic(any)` is translated to the `runtime.gopanic(any)` function. This function sets up the `panic` context and starts the stack unwinding process.

Figure 125. `panic(any)` or `gopanic(any)` (`src/runtime/panic.go`)

```
// The implementation of the predeclared function panic.
func gopanic(e any) {
    ... // if e == nil {

        // Setting up the panic context
        var p _panic
        p.arg = e
        ...

        p.start(getcallerpc(),
unsafe.Pointer(getcallersp()))}
```

```

        for {
            fn, ok := p.nextDefer()
            if !ok {
                break
            }
            fn()
        }

        preprintpanics(&p)

        fatalpanic(&p)
        *(*int)(nil) = 0 // not reached
    }
}

```

The Go runtime creates a `_panic` record (which we have seen before in `deferreturn`) and captures the program counter and stack pointer. The `_panic` record is then added to a list that tracks all active panics in the current goroutine —just like `_defer` records. This is handled in `_panic.start()`, and it allows for nested panics.

Skipping the finer details, the process is very similar to what we have already discussed with `deferreturn`. The function iterates over the `defer` chain using `nextDefer`, grabbing and executing each deferred function. After running through the chain, it calls `preprintpanics` to get all panic values ready for printing, and then calls `fatalpanic(&p)` to crash the program.

If you just look at the control flow, it looks like `fatalpanic(&p)` always runs and the program always crashes. But we know that is not actually true. If a deferred function calls `recover()` correctly, the panic can be stopped, and the program won't crash.

The runtime needs a way to intercept and change the flow in this case. This mechanism is found in the `p.nextDefer()` function, which finds the next deferred function to execute:

Figure 126. `nextDefer` (`src/runtime/panic.go`)

```

func (p *_panic) nextDefer() (func(), bool) {
    gp := getg()

    if !p.deferreturn {
        ...
        if p.recovered {
            mcall(recovery) // <- This is the
recovery process
            throw("recovery failed")
        }
    }
}

```

```

        }

    p.argp = add(p.startSP, sys.MinFrameSize)
    for {
        ...
        // Get the current _defer from the defer chain in
        // the current frame,
        // retrieve its function and free it.
        Recheck:
            if d := gp._defer; d != nil && d.sp ==
    uintptr(p.sp) {
        ...
        fn := d.fn
        d.fn = nil
        p.retpc = d.pc

        // Unlink and free the _defer.
        gp._defer = d.link
        freedefer(d)

        return fn, true
    }

    // If the current _defer is not in the
    // current frame,
    // move to the previous frame and check
    // again.
    if !p.nextFrame() {
        return nil, false
    }
}
}

```

The key part here is the `p.recovered` flag. If this flag is true, it means the panic has been recovered in a deferred function.

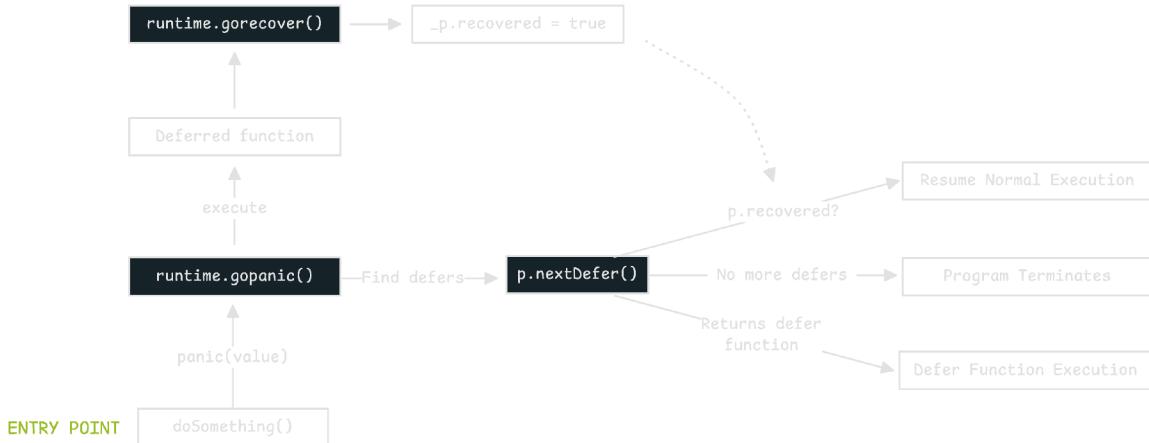


Illustration 202. Workflow of panic recovery

This function simply walks the defer chain and returns the next deferred function to be executed during stack unwinding.

When you call `recover()` inside a deferred function, it is translated to `runtime.gorecover(...)`, which sets the `recovered` flag of the `_panic` record to true. We will look at `gorecover` in more detail soon.

Now, assume the `_panic` record has already been recovered; how does this actually change the control flow? The key is the `mcall(recovery)` call. It changes control flow by manipulating the program's low-level state.

`mcall()` is a special assembly function that switches from the current goroutine's stack to the system stack. This is explained in detail in Chapter 7. The switch is important because it lets Go safely change the goroutine's state from the system stack, something too risky to do while still running on the same goroutine stack.

This function is responsible for running all deferred functions in the same frame, and then letting normal execution resume:

Figure 127. recovery (src/runtime/panic.go)

```

func recovery(gp *g) {
    ...
    // Make the deferproc for this d return again,
    // this time returning 1. The calling function will
    // jump to the standard return epilogue.
    gp.sched.sp = sp
    gp.sched.pc = pc
  
```

```

        gp.sched.lr = 0
        switch {
        case goarch.IsAmd64 != 0:
            gp.sched.bp = fp - 2*goarch.PtrSize
        case goarch.IsArm64 != 0:
            gp.sched.bp = sp - goarch.PtrSize
        }
        gp.sched.ret = 1 // <---
    }

    gogo(&gp.sched)
}

```

The Go runtime now takes over the control flow. Remember from the `deferproc` and `deferprocStack` discussion earlier:

Assembly	Pseudo Code
<pre> CALL  runtime.deferproc(SB) CMP   \$0, R0 BNE   100 JMP   32 → CALL runtime.deferreturn(SB) </pre>	<pre> r := runtime.deferproc(obj.F) if r != 0 {     runtime.deferreturn() } </pre>

Illustration 203. `runtime.deferproc` schedules a deferred function call

Normally, `deferproc` and `deferprocStack` return `0`, signaling regular control flow. During recovery, they return `1` instead and jump to the `deferreturn()` function to run any remaining deferred functions in that frame.

Recovery does the following:

- Sets up the stack pointer, program counter, and frame pointer for the caller, restoring the state from when `deferprocStack` was first called.
- Sets the return value to `1` (`gp.sched.ret = 1`) as if `deferproc` or `deferprocStack` is returning `1`.
- Calls `gogo(&gp.sched)` to actually resume the goroutine at the modified point.

The value in `gp.sched.ret` does not sit on the goroutine stack; it is copied directly into the architecture's return-value register (`AX` on amd64, `R0` on arm64, etc.) when `gogo` is executed.

So, when execution restarts at the `pc`, the calling code sees what appears to be a second return from `deferprocStack`, but this time with the value `1`:

Assembly	Pseudo Code
<pre> CALL  runtime.deferproc(SB) CMP   \$0, R0 BNE   100 JMP   32 → CALL runtime.deferreturn(SB) </pre>	<pre> r := runtime.deferproc(obj.F) if r != 0 {     runtime.deferreturn() } </pre>

Illustration 204. Recovery path forces deferproc to return one

The `deferreturn()` function is called immediately after, which executes any remaining deferred functions in that frame, assuming that the function is returning normally.

Now, what happens if we set a return value before the panic?

This is the same as manipulating the return values with `defer` statements we discussed earlier:

```

func doSomething() (res uint) {
    defer func() {
        fmt.Println("This remaining deferred
function will execute")
    }()

    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()
}

res = 100
panic("oops")
return 42
}

func main() {
    result := doSomething()
    fmt.Println("After doSomething, result is", result)
}

// Output:
// Recovered: oops
// This remaining deferred function will execute
// After doSomething 100

```

Since the panic is recovered, the function returns normally. The value of `res` is still `100` because of the earlier assignment, so that is what is returned.

Let's consider a case where a deferred function calls another function, and that helper calls `recover()`. Should this work? Should it be allowed to recover the panic?

```
func recoverHelper() {
    if r := recover(); r != nil {
        fmt.Println("Recovered:", r)
    }
}

func doSomething() (res int, err error) {
    defer func() {
        recoverHelper()
    }()
    panic("oops")
}
```

To understand what is allowed here, we need to look at how `recover()` works—more specifically, how `gorecover()` works.

## Recovery and Argument Pointer

Let's look at the condition required for a successful recovery:

Figure 128. `gorecover` (`src/runtime/panic.go`)

```
func gorecover(argp uintptr) any {
    gp := getg()
    p := gp._panic
    if p != nil && !p.goexit && !p.recovered && argp ==
    uintptr(p.argp) {
        p.recovered = true
        return p.arg
    }
    return nil
}
```

## The `p.goexit` flag

The `p.goexit` flag is not the main topic here, but it's useful to know. There is a special runtime function you can call, `runtime.Goexit()`.

This function terminates the current goroutine, similar to panic, but without causing a panic or affecting other goroutines. When you call it, all deferred functions in that goroutine are executed, but the program does not crash. You can think of it as a "return statement" for the whole goroutine.

The most important part is the argument pointer check (`argp == uintptr(p.argp)`). This check ensures that only the correct deferred function can recover from a panic.

If you forgot what the argument pointer is, look back at the start of this chapter. Here is a quick reminder:

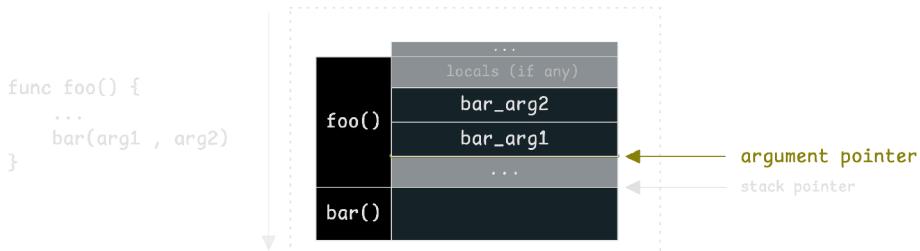


Illustration 205. Argument pointer marks start of function arguments

When a panic happens, the Go runtime (through `runtime.gopanic()`) stores the position of the outgoing argument area in `_panic.argp`:

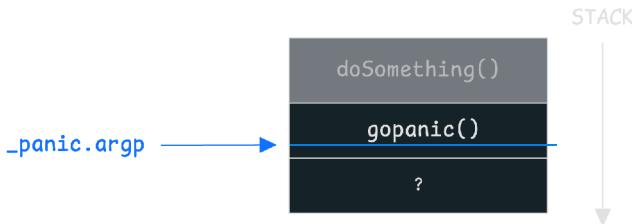


Illustration 206. Argument pointer uniquely identifies caller's frame

This is where `runtime.gopanic()` is preparing the arguments for the deferred function.

At this point, the runtime does not know which deferred function (if any) will call `recover()`. The key is that only the immediate callee has the correct argument pointer that matches what was prepared by `gopanic()`.

When a deferred function calls `recover()`, it passes its argument pointer to `runtime.gorecover(argp)`. If this deferred function is the one called directly by

the `gopanic()` frame, its argument pointer matches `_panic.argp`, so the check succeeds and the panic is recovered:



Illustration 207. Recover checks if argument pointer matches panic

This is a simplified explanation. If a deferred function has arguments, a closure (wrapper) with no arguments and no return value is created to wrap the deferred function.

The stack frame then looks a bit more complicated:



Illustration 208. Defer wrapper adjusts argp for recover check

The wrapper adjusts the argument pointer in the `_panic` record so that it matches the outgoing argument area for the wrapper itself.

If this is confusing, don't worry. From the user's point of view, what matters is: *"only the top-level deferred function containing `recover()` can actually recover the panic"*.

To make this clearer, let's look at a common mistake when using `recover()`:

```
func customRecover() error {
    if r := recover(); r != nil {
        recoverTime.Increase() // metric
        log.Printf("Recovered from panic: %v", r) //
    }
}
```

```

        return fmt.Errorf("panic: %v", r)
    }

    return nil
}

func doSomething() (err error) {
    // func 1
    defer func() {
        err = errors.Join(err, customRecover())
    }()
    panic("oops")
}

```

Here, the developer tries to combine any error that may have happened with an error from a panic. The idea is to collect metrics and log information in a custom function. But in this setup, the call stack looks like this:

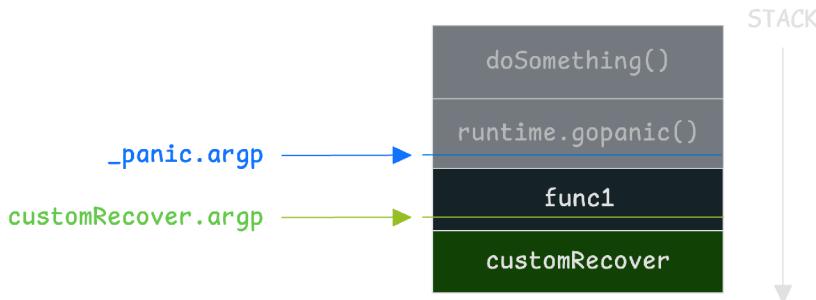


Illustration 209. `_panic.argp` does not match `customRecover.argp`; recover returns nil

The problem is that the argument pointer of the `_panic` record (`_panic.argp`) does not match the argument pointer at the call to `recover()` in `customRecover`. Because of this, the check `argp == uintptr(p.argp)` fails, and the panic cannot be recovered here.

Go 1.22 introduced a bug where, if `recover()` is called indirectly through another function, the panic will still be recovered:

```

func main() {
    defer func(i int) {
        recoverHelper()
    }(1)

    panic("oops")
}

```

```
func recoverHelper() {
    if r := recover(); r != nil {
        fmt.Println("Recovered:", r)
    }
}
```

This happens because the anonymous function is inlined in the wrapper, making the runtime treat the deferred function as `recoverHelper` and allowing `recover()` to succeed. This is a bug that is expected to be fixed in Go 1.26.

Another common mistake is calling `recover()` directly in a `defer` statement, instead of inside the deferred function:

```
func main() {
    defer recover()

    panic("Unexpected wrong")
}
```

This also won't work as expected. Consider the situation with two panics and one recover. Does one recover catch both panics, just the first, or only the last one?

```
func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()

    defer func() {
        panic("Second panic")
    }()

    panic("First panic")
}
```

The answer is that one `recover()` can only catch one panic at a time. In this case, the second `panic` will be caught, and the program will return normally without crashing. Panics do not stack up; `recover()` only gets the most recent panic.

```
Recovered: Second panic
```

That wraps up the `panic - recover` discussion. The current mechanism is not perfect, and it uses quite a few tricks to work reliably. Recovering from a panic means the runtime must unwind the stack very carefully, keep track of which

frames were created by real function calls versus those added by the runtime, and coordinate with deferred functions that might trigger more panics themselves.

A lot of this relies on conventions—how return program counters are stored, how different architectures manage their link registers, and where the compiler inserts `deferreturn`.

Since these conventions differ across architectures and change with new compiler optimizations, the Go runtime uses several heuristics and special-case tables to keep everything working as expected.

## 5. Profiling

`pprof` is a tool for profiling and performance analysis in Go. You use it to collect, process, and view runtime performance data. This helps you figure out how your app is running and find slow spots.

`pprof` has two main parts: the `runtime/pprof` package, which gathers profile data, and the `go tool pprof` command, which helps you analyze and visualize the data.

For example, a profile can keep track of all running goroutines and, for each one, record the stack trace to show what it is doing right now. Another profile can track memory allocations and keep the stack trace for where each allocation happened.

Go has some profiles built into the runtime. These are always ready to use when your program starts. You do not need to register them. They collect important performance data that helps with debugging and optimization.

There are four main groups of built-in profilers:

- Memory-sampling profilers (`heap` and `allocs`) are always active as long as `runtime.MemProfileRate` is not zero. By default, it is on. They report what their sampler already recorded.
- Concurrency profilers (`goroutine` and `threadcreate`) are instantaneous. The runtime keeps them up to date and you can look at them anytime, no setup needed.
- The CPU profiler works only while it is running. It gathers timed samples between calls to `pprof.StartCPUProfile` and `pprof.StopCPUProfile`.

- Event-driven contention profilers (`block` and `mutex`) start collecting only after you set `runtime.SetBlockProfileRate` or `runtime.SetMutexProfileFraction`. They track time spent waiting for synchronization.

Besides these, `runtime/pprof` lets you build your own user profiles with the following API:

```
func NewProfile(name string) *Profile {}
```

This lets you, as a library or app author, use the same reporting tools that the built-in profiles use for your own tracking needs.

## What is user-defined profile?

User-defined profiles are not the key focus of this book. But it is useful to know you can make your own. A user profile is basically a live set. Every time you call `Profile.Add`, you add a key and the stack trace of the code path that called it. Every time you call `Profile.Remove`, you delete that key.

Suppose your server creates temporary files, images, and reports, but sometimes does not close them. You can wrap the file object like this:

```
// openFiles is a custom profile to track files that are
// still open and not closed yet.
var openFiles = pprof.NewProfile("example.open-files")

// leakImage is a code path that makes an image file and
// forgets to close it.
func leakImage() {
    f, _ := os.CreateTemp("", "img-*tmp")
    openFiles.Add(f, 1)
    runtime.SetFinalizer(f, func(file *os.File) {
        openFiles.Remove(file)
    })
}

// leakReport is another code path that leaks report files.
func leakReport() {
    f, _ := os.CreateTemp("", "report-*tmp")
    openFiles.Add(f, 1)
    runtime.SetFinalizer(f, func(file *os.File) {
        openFiles.Remove(file)
    })
}
```

```
}
```

```
// Leak files every few seconds.
ticker := time.NewTicker(3 * time.Second)
for range ticker.C {
    leakImage()
    if time.Now().Unix()%2 == 0 {
        leakReport()
    }
}
```

Each time `leakImage` runs, it calls `os.CreateTemp` to make a temporary image file. Then it calls `openFiles.Add(f, 1)`. The `1` tells `pprof` to cut the stack so it starts at the caller of `leakImage`, which makes profiles easier to read. Because the file is left open, it stays in the profile. A finalizer is added with `runtime.SetFinalizer`. If the file ever gets closed, by you or by the garbage collector, the finalizer runs and calls `openFiles.Remove(file)`. This way the profile always matches reality.

At any time, you can snapshot the profile and use the usual `pprof` tools to see the exact call stacks:

```
$ go tool pprof http://host:6060/debug/pprof/myserver.open-
conns
```

User-defined profiles use the same `pprof` system. You can dump them to a file with `Profile.WriteTo`, serve them over `net/http/pprof`, and open them with the `go tool pprof` UI. You can also view them together with other profiles in one debugging session. This gives you insight into resources that matter to your app, without building your own reporting tools.

When you write the profile, `pprof` merges identical stacks and shows how many keys are still at each call site:

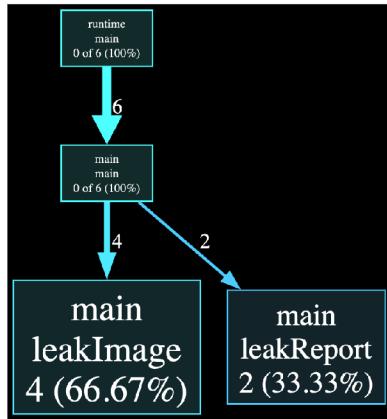


Illustration 210. Custom pprof profile of leaked file handles

This is very useful for finding resource leaks or odd retention patterns that built-in profiles do not catch. `pprof` will show which call chains are holding onto open files. You can use this pattern for open sockets, goroutines that should end, loaded plugins, objects in caches, or anything with a clear lifetime.

## Profile Collection & Access Methods

### Profiling with go test

The Go testing framework has strong profiling features you can turn on with command-line flags. When you run your tests or benchmarks using `go test`, you can enable different profiling types with these options:

```
$ go test -cpuprofile cpu.prof -memprofile mem.prof -bench .
```

If you look at the `testing/testing.go` file, you can see where these flags are set:

Figure 129. Benchmark Profile Flags (src/testing/testing.go)

```
memProfile = flag.String("test.memprofile", "", ...)
memProfileRate = flag.Int("test.memprofilerate", 0, ...)
cpuProfile = flag.String("test.cpuprofile", "", ...)
blockProfile = flag.String("test.blockprofile", "", ...)
blockProfileRate = flag.Int("test.blockprofilerate", 1, ...)
mutexProfile = flag.String("test.mutexprofile", "", ...)
mutexProfileFraction = flag.Int("test.mutexprofilefraction",
1, ...)
```

These flags control what kind of profile data gets collected and where it is saved. The actual profiling work is done by the `runtime/pprof` package. The `testing` package uses this at the right moments during your tests.

Before your tests start, the `M.before()` function in `testing` package runs and sets up the profiles based on the flags you passed in:

Figure 130. `M.before` (`src/testing/testing.go`)

```
package testing

// before runs before all testing.
func (m *M) before() {
    if *memProfileRate > 0 {
        runtime.MemProfileRate = *memProfileRate
    }
    if *cpuProfile != "" {
        f, err :=
os.Create(toOutputDir(*cpuProfile))
        if err != nil {
            fmt.Fprintf(os.Stderr, "testing:
%s\n", err)
            return
        }
        if err := m.deps.StartCPUProfile(f); err !=
nil {
            fmt.Fprintf(os.Stderr, "testing:
can't start cpu profile: %s\n", err)
            f.Close()
            return
        }
    }
    // Similar setup for other profiles...
}
```

What happens is `m.deps.StartCPUProfile()` calls the `runtime/pprof.StartCPUProfile()` function. This tells the Go runtime to start recording CPU profile data.

For memory profiles, `m.before()` changes `runtime.MemProfileRate`, which sets how often memory allocations are sampled. The other profiles, like `block` and `mutex`, have their own ways to set the sample rate. Soon we will see how to use the `pprof` package directly.

The `M.after()` function is called when all the tests are done. It stops the active profilers and writes the data to the files:

Figure 131. M.after (src/testing/testing.go)

```
// after runs after all testing.  
func (m *M) after() {  
    m.afterOnce.Do(func() {  
        m.writeProfiles()  
    })  
}
```

At this point, all your profiles have been collected and written to disk. You can now use the `go tool pprof` command to analyze them.

## Profiling via runtime/pprof API

The second way to use pprof is by calling the `runtime/pprof` package API directly.

### Lookup Standard Profiles

To get access to Go's standard profiles, such as heap, goroutine, or mutex, you use the `Lookup` method:

```
pprof.Lookup("name")
```

The argument must be one of the built-in profile names: `"heap"`, `"goroutine"`, `"allocs"`, `"threadcreate"`, `"block"`, or `"mutex"`. For example, to get the heap profile:

```
func main() {  
    runtime.GC() // make sure the heap profile shows the  
    // current memory state  
    prof := pprof.Lookup("heap")  
  
    var buf bytes.Buffer  
    if err := prof.WriteTo(&buf, 0); err != nil {  
        log.Fatal(err)  
    }  
  
    log.Println("Heap profile written to buffer, size:",  
    buf.Len())  
}
```

For profiles like `block` or `mutex`, you must turn them on first. If you do not enable them, the profile you get from `Lookup` will be empty.

### Write Profiles (`WriteTo`)

The `Profile` type also has the `WriteTo` method, which sends the profile data to any `io.Writer`:

```
func (p *Profile) WriteTo(w io.Writer, debug int) error {}
```

The `debug` value controls the format of the output:

- `debug=0` : Gives you a binary file in protocol buffer format, which is compact and best for tools.
- `debug=1` : Gives you a human-readable text file with function names and line numbers.
- `debug=2` : For some profiles, like goroutine, gives even more details.

With `Lookup` and `WriteTo`, you have the main tools for Go profiling. Here is a full example that exports the goroutine profile to a file:

```
func exportGoroutineProfile() error {
    f, _ := os.Create("goroutine.prof")
    defer f.Close()

    p := pprof.Lookup("goroutine")
    if p == nil {
        return errors.New("goroutine profile not found")
    }

    return p.WriteTo(f, 0)
}
```

### Count Entries

`Profile.Count` answers just one question: how many different execution stacks are held in the profile right now. It does not add up sample weights, memory usage, or time spent; it simply reports how many distinct call stacks have been recorded so far.

Use it exactly as you would call `len` on a slice:

```
func main() {
    prof := pprof.Lookup("goroutine")
    count := prof.Count()
    log.Println("Number of goroutines:", count)
}
```

The meaning of "how many stacks" depends on the kind of profile, because each profile records a different kind of event.

For the `goroutine` and `threadcreate` profiles the answer is a live snapshot. `goroutine` uses `runtime.NumGoroutine`, and `threadcreate` walks the runtime's list of machine structures (M). The number you see is the number that exist at the instant you ask, not the total ever created.

```
func countGoroutine() int {
    return runtime.NumGoroutine()
}

func countThreadCreate() int {
    n, _ := runtime.ThreadCreateProfile(nil)
    return n
}
```

Behind the scenes, this code walks through the runtime's list of machine structures (M) or processors (P) to count the current goroutines or threads.

For the sampling profiles `heap` and `allocs` the runtime returns the number of allocation sites that still have at least one sample after the runtime's sampling and grouping pass is finished.

```
func countHeap() int {
    n, _ := runtime.MemProfile(nil, true)
    return n
}
```

These two profiles share the same underlying data; only the default view in `pprof` is different.

For `block` and `mutex` profiles the count is the number of distinct call sites that have generated blocking or contention data so far.

Internally the runtime keeps a hash table keyed by the full stack trace where the event occurred. Each unique stack trace becomes a separate entry; repeated events at the same stack trace land in the same entry and increment its counters. When

you call `runtime.BlockProfile(nil)` or `runtime.MutexProfile(nil)` with a nil slice, the runtime walks that table, counts the entries, and returns the total:

```
func countBlock() int {
    n, _ := runtime.BlockProfile(nil)
    return n
}

func blockProfileInternal(size int, copyFn
func(profilerecord.BlockProfileRecord)) (n int, ok bool) {
    lock(&profBlockLock)
    head := (*bucket)(bbuckets.Load())
    for b := head; b != nil; b = b.allnext {
        n++
    }
    // ...
}
```

Count therefore tells you how many call-site entries exist, not how busy they are. To answer "how much?"—bytes, nanoseconds, objects—you must fetch the full records or write the profile out and inspect it with `go tool pprof`, which adds up the values for you.

### Start and Stop Profilers

You do not need to manually initialize most profiles in Go. For example, you do not have to say, "I want to start profiling goroutines now."

Some built-in profiles, like `goroutine` and `threadcreation` profiles, are always available and are kept up to date by the Go runtime. These profiles have almost no overhead because they simply give you access to data that the runtime already tracks. They do not require sampling or special code, so they are very lightweight. Memory profiling is also enabled by default and does add a small overhead.

Other profiles in Go such as CPU profiling, `block`, and `mutex`, are turned off by default. They add no overhead unless you turn them on. No data is collected until you enable them using their APIs.

For example, if you want to collect a CPU profile, you must start and stop profiling yourself:

```
func main() {
    var buf bytes.Buffer
```

```
if err := pprof.StartCPUProfile(&buf); err != nil {
    log.Fatal(err)
}

doHeavyComputation()

pprof.StopCPUProfile()
log.Println("CPU profile captured:", buf.Len(), "bytes")
}
```

`block` and `mutex` profiles use a different method. You enable or disable them by setting their rate:

```
func main() {
    runtime.SetBlockProfileRate(1)
    runtime.SetMutexProfileFraction(1)

    //...

    runtime.SetBlockProfileRate(0)
    runtime.SetMutexProfileFraction(0)
}
```

Under the hood, `pprof.StartCPUProfile` calls a runtime function like `runtime.SetCPUProfileRate`. This function gives low-level control over how often the CPU is sampled.

Most of the time, you should use `pprof.StartCPUProfile`, which sets a standard 100 Hz sampling rate and handles the profile output for you. The lower-level runtime call is mainly for special situations where you need a custom rate or want to use more advanced profiling tools.

## Profiling over HTTP (net/http/pprof)

The third method uses the `net/http/pprof` package to make profiling data available over HTTP. This is a very easy way to expose profiles, and it needs very little code change. Just add the following `import` statement:

```
import _ "net/http/pprof"
```

The package's `init` function will automatically register a set of HTTP handlers using the default HTTP server mux. These handlers use paths under the `/debug/pprof/` prefix. You can access them using any HTTP client or browser:

```
func init() {
    ...
    http.HandleFunc(prefix+"/debug/pprof/", Index)
    http.HandleFunc(prefix+"/debug/pprof/cmdline",
Cmdline)
    http.HandleFunc(prefix+"/debug/pprof/profile",
Profile)
    http.HandleFunc(prefix+"/debug/pprof/symbol",
Symbol)
    http.HandleFunc(prefix+"/debug/pprof/trace", Trace)
}
```

By default, `net/http/pprof` registers its handlers on `http.DefaultServeMux`. If your application uses the default server setup like this:

```
http.ListenAndServe(":8080", nil)
```

Then all the `/debug/pprof/` endpoints are automatically available. But if you use your own `http.ServeMux`, for example:

```
mux := http.NewServeMux()
http.ListenAndServe(":8080", mux)
```

then the pprof endpoints are not registered unless you add them yourself:

```
import "net/http/pprof"

mux.HandleFunc("/debug/pprof/", pprof.Index)
mux.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
mux.HandleFunc("/debug/pprof/profile", pprof.Profile)
mux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
mux.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

In production, you should protect these endpoints with proper access control, because they show a lot of internal information about your app. Many companies keep them private, only allowing internal access, using reverse proxies or network policies to block outsiders.

Under the hood, `net/http/pprof` uses the same manual instrumentation with `runtime/pprof` that we covered earlier. This package is a bridge between HTTP requests and the `runtime/pprof` API. For example, when you visit `/debug/pprof/profile`, the handler runs `pprof.StartCPUProfile()`, waits for the requested amount of time (30 seconds by default), and then calls `pprof.StopCPUProfile()`:

Figure 132. Profile (src/net/http/pprof/pprof.go)

```

func Profile(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("X-Content-Type-Options", "nosniff")
    sec, err := strconv.ParseInt(r.FormValue("seconds"),
10, 64)
    if sec <= 0 || err != nil {
        sec = 30
    }

    configureWriteDeadline(w, r, float64(sec))

    w.Header().Set("Content-Type", "application/octet-
stream")
    w.Header().Set("Content-Disposition", `attachment;
filename="profile"`)
    if err := pprof.StartCPUProfile(w); err != nil {
        serveError(w,
http.StatusInternalServerError,
fmt.Sprintf("Could not enable CPU
profiling: %s", err))
        return
    }
    sleep(r, time.Duration(sec)*time.Second)
    pprof.StopCPUProfile()
}

```

Here, the HTTP response writer is used as the output for the profile, so the client can download the profile data directly. Other types like heap, block, and mutex profiles work the same way. The handler calls the right function from the `runtime/pprof` package to collect the data.

You can also pass some query arguments to change how the HTTP endpoints behave:

- `debug=N` (all profiles): output format. `N=0` gives binary (default), `N>0` gives plain text.
- `gc=N` (heap profile): `N>0` runs garbage collection before collecting the profile.
- `seconds=N` (allocs, block, goroutine, heap, mutex, threadcreate profiles): returns a delta profile.
- `seconds=N` (cpu (profile), trace profiles): profiles for the specified number of seconds.

Next, you will see all the endpoints that `net/http/pprof` makes available.

[Index](#)

The base endpoint is `/debug/pprof/`. Visiting this URL in a browser returns an HTML page listing all the available profiles along with hyperlinks to their respective endpoints.

This includes `goroutines`, `heap`, `thread creation`, `block`, `mutex` and user-defined profiles. Most of the hyperlink has a query parameter `debug=1` to get the human-readable format, this debug value will be passed to the `Profile.WriteTo` method as we discussed earlier.



The screenshot shows a terminal window displaying the output of the Go pprof debug endpoint. The output is as follows:

```
/debug/pprof/
Set debug=1 as a query parameter to export in legacy text format

Types of profiles available:
Count Profile
11 allocs
0 block
0 cmdline
3 goroutine
11 heap
0 mutex
0 profile
0 symbol
8 threadcreate
0 trace
full goroutine stack dump

Profile Descriptions:
• allocs: A sampling of all past memory allocations
• block: Stack traces that led to blocking on synchronization primitives
• cmdline: The command line invocation of the current program
• goroutine: Stack traces of all current goroutines. Use debug=2 as a query parameter to export in the same format as an unrecovered panic.
• heap: A sampling of memory allocations of live objects. You can specify the gc GET parameter to run GC before taking the heap sample.
• mutex: Stack traces of holders of contended mutexes
• profile: CPU profile. You can specify the duration in the seconds GET parameter. After you get the profile file, use the go tool pprof command to investigate the profile.
• symbol: Maps given program counters to function names. Counters can be specified in a GET raw query or POST body, multiple counters are separated by '+'.
• threadcreate: Stack traces that led to the creation of new OS threads
• trace: A trace of execution of the current program. You can specify the duration in the seconds GET parameter. After you get the trace file, use the go tool trace command to investigate the trace.
```

Illustration 211. Go pprof debug endpoint showing available profiles

Now, the default call to `/debug/pprof/goroutine` returns a top summary of the most common goroutine stack traces, not every single one.

But the last line 'Full goroutine stack dump' runs the goroutine profile with `debug=2`. It returns the full stack trace of every live goroutine, similar to what you'd get from `runtime.Stack()` or from a `panic` output. This includes goroutine states, function calls, and blocking points, and is very helpful for diagnosing goroutine leaks, deadlocks, or unexpected concurrency patterns.

This behavior is hardcoded only in the goroutine profile endpoint. Other profiles either don't support `debug=2` or show no difference between `debug=1` and `debug=2`.

## Cmdline

The `/debug/pprof/cmdline` endpoint returns the command-line arguments passed to the running binary. This is helpful for understanding how the program was started, especially when working with binaries in production environments where you don't have access to how the process was launched.

The output is plain text, with arguments separated by null bytes (`\x00`). Assume the program was started with this command:

```
./myserver -config /etc/myserver/config.yaml -log-level  
debug
```

Then, the output of `/debug/pprof/cmdline` would be:

```
$ curl http://localhost:8080/debug/pprof/cmdline  
../myserver\x00-config\x00/etc/myserver/config.yaml\x00-log-  
level\x00debug
```

These are the exact contents of `os.Args`, but joined by null bytes (`\x00`) instead of spaces. This format makes it clear where each argument begins and ends, which avoids ambiguity in cases where arguments contain spaces.

## Profile (CPU)

The `/debug/pprof/profile` endpoint is the most commonly used for CPU profiling. When accessed, it triggers a CPU profiling session that runs for 30 seconds by default.

You can change the duration using the `seconds` query parameter, like `/debug/pprof/profile?seconds=10`. The result is a binary-encoded profile in the same format you'd get from `pprof.StartCPUProfile`. This data is suitable for analysis with `go tool pprof` or graphical frontends.

Unlike other profile types in Go's `pprof` package, CPU doesn't use `Profile.WriteTo` method. This fundamental architectural difference means that CPU profiles don't support the `debug` parameter that other profiles accept.

When you capture a CPU profile using `pprof.StartCPUProfile(writer)` and `pprof.StopCPUProfile()`, the profile is always written in a single binary format—the `proto` format, which is a compressed protobuf representation.

## Symbol

When a profile includes stack traces, those traces are made up of raw program counters. These need to be translated into function names and line numbers to make the profile readable and useful for people.

This is exactly what the `Symbol` endpoint does. You can send a `GET` or `POST` request to `/debug/pprof/symbol` with one or more hexadecimal addresses to look up.

For example, if you are looking at a heap profile (`/debug/pprof/heap`) and you see a stack trace like this:

Figure 133. VictoriaMetrics open-source heap profile sample

```
0: 0 [11: 18382848] @ 0x6e9ce8 0x6e9c92 0x81c09d 0x81b42a
 0x81b10a 0x81ae0a 0x47daa1
#      0x6e9ce7 /lib/slicesutil.SetLength[...]+0x87
/lib/slicesutil/slicesutil.go:20
#      0x6e9c91 /lib/prompbmarshal.(*WriteRequest).MarshalProtobuf+0x31
/lib/prompbmarshal/util.go:13
#      0x81c09c
/app/vmagent/remotewrite.tryPushWriteRequest+0x9c
/app/vmagent/remotewrite/pendingseries.go:252
#      0x81b429 /app/vmagent/remotewrite.(*writeRequest).tryFlush+0x89
/app/vmagent/remotewrite/pendingseries.go:155
#      0x81b109 /app/vmagent/remotewrite.(*pendingSeries).periodicFlusher+0x189
/app/vmagent/remotewrite/pendingseries.go:91
#      0x81ae09
/app/vmagent/remotewrite.newPendingSeries.func1+0x49
/app/vmagent/remotewrite/pendingseries.go:53
```

In the first line, the addresses like `0x6e9ce8` and `0x6e9c92` are program counters, or PCs. These are just memory addresses pointing to the exact instruction that was running. The profiler needs to resolve these PCs into readable information—like function names, source files, and line numbers, as shown in the lines that start with `# 0x6e9ce7`.

You can use the `Symbol` endpoint to resolve those addresses:

```
$ curl http://localhost:8080/debug/pprof/symbol?
address=0x6e9ce8+0x6e9c92+0x81c09d+0x81b42a+0x81b10a+0x81ae0a+0x47daa1
```

```
num_symbols: 1
0x6e9c92 /lib/prompbmarshal.(*WriteRequest).MarshalProtobuf
0x81c09d /app/vmagent/remotewrite.tryPushWriteRequest
0x81b42a /app/vmagent/remotewrite.(*writeRequest).tryFlush
0x81b10a /app/vmagent/remotewrite.
(*pendingSeries).periodicFlusher
0x81ae0a /app/vmagent/remotewrite.newPendingSeries.func1
0x47daa1 runtime.goexit
```

This tells you, for example, that address `0x81c09d` is inside `tryPushWriteRequest`, and `0x81b42a` is inside `tryFlush`. The same address can show up more than once in a stack trace if a function is re-entered or called more than once.

Even though you see six resolved symbols, the first line says `num_symbols: 1`. That number is not an actual count of how many symbols were resolved; it is a legacy part of the `pprof` protocol.

This header is essentially just a flag indicating that symbol resolution is working and results are being returned. The value (`1`) is not important—you could see `num_symbols: 2`, `3`, or any positive number, and it would not affect the rest of the output. The important data is in the lines below.

You may also notice that the first address (`0x6e9ce8`) is not resolved. In the heap profile, this address appears as `/lib/slicesutil.SetLength[...]`, but it is missing from the symbol endpoint response. This can happen if the address belongs to an inlined function or a location that the `runtime.FuncForPC` function cannot resolve directly.

## Trace

The execution trace from `/debug/pprof/trace` gives you a very detailed look at what your program is doing. It records many kinds of events, such as when goroutines are created, blocked, or unblocked, system calls, garbage collection events, CPU profile samples, and more. Each event has a nanosecond-precision timestamp and a full stack trace.

Because it records so much, execution tracing is very powerful for finding complex performance issues, especially those involving concurrency, scheduling, or interactions with the system.

The downside is clear: traces generate far more data than regular profiles. Even a short trace of just a few seconds can produce megabytes or even gigabytes of data. Traces are heavy to collect, store, and analyze, making them impractical for continuous monitoring or for tracking activity over long periods.

Tracing also has more overhead than regular profiling. When tracing is enabled, extra code runs to capture each event, which can slow down your program or change its behavior. In some cases, the act of tracing can hide or alter the very problems you are trying to investigate.

## Using go tool pprof

Now you know there are three main ways to collect profile data. But the last method, using `net/http/pprof`, does not save any file on disk by itself. It just makes the profile data available over HTTP.

You can fetch profiles using tools like `curl`, or you can use `go tool pprof` directly.

For a 30-second CPU profile:

```
$ go tool pprof http://localhost:6060/debug/pprof/profile?  
seconds=30
```

For a heap profile:

```
$ go tool pprof http://localhost:6060/debug/pprof/heap  
  
Saved profile in ~/pprof/pprof.vmstorage-  
prod.alloc_objects.alloc_space.inuse_objects.inuse_space.003  
.pb.gz  
File: vmstorage-prod  
Build ID: 27fc04955391ced6b280605139b81a2ebdaf78e0  
Type: inuse_space  
Time: 2024-11-21 14:41:29 +07  
Entering interactive mode (type "help" for commands, "o" for  
options)
```

The `pprof` tool first checks if the target you give is a file on disk by calling `os.Stat()`. If it is not a file, it treats the target as a URL. When you give it a URL, the tool makes an HTTP request to download the profile data. The profile is decoded and saved as a temporary file, with a name like `pprof.alloc_objects.alloc_space.inuse_objects.inuse_space.[pid].pb.gz` in your system's temp directory.

If you want to just save the profile data for later, you can use `curl`:

```
$ curl -o heap.pprof http://localhost:6060/debug/pprof/heap
```

## Interactive Shell

After launching the `pprof` tool, it opens an interactive shell. Here you can enter commands to analyze the profile. Below are the most important commands and how to use them.

`top` : Summary of Expensive Functions

This command shows you the functions using the most resources. For example:

```
$ (pprof) top

Showing nodes accounting for 1209.99MB, 97.23% of 1244.44MB
total
Dropped 99 nodes (cum <= 6.22MB)
Showing top 10 nodes out of 70
      flat  flat%   sum%      cum      cum%
  670.69MB 53.89% 53.89%  670.69MB 53.89%
/VictoriaMetrics/lib/bytesutil.ResizeNoCopyNoOverallocate
  224MB 18.00% 71.89%    224MB 18.00%
/VictoriaMetrics/lib/bytesutil.ResizeWithCopyMayOverallocate
  149.59MB 12.02% 83.92%  149.59MB 12.02% /fastcache.
(*bucket).cleanLocked
  40MB  3.21% 87.13%    40MB  3.21%
github.com/klauspost/compress/zstd.(*fastBase).ensureHist
  37.76MB 3.03% 90.16%  46.41MB 3.73%
/VictoriaMetrics/lib/mergeset.(*partSearch).readIndexBlock
  35.10MB 2.82% 92.99%  35.10MB 2.82%
/VictoriaMetrics/lib/slicesutil.SetLength[go.shape.struct {
Start uint32; End uint32 }]
  24MB  1.93% 94.91%    24MB  1.93%
/VictoriaMetrics/lib/storage.newRawRows
  12.40MB   1% 95.91%  12.40MB   1%
/VictoriaMetrics/lib/storage.unmarshalMetaindexRows
  8.64MB  0.69% 96.60%  8.64MB  0.69%
/VictoriaMetrics/lib/slicesutil.SetLength[go.shape.d228031a3
1c07344d6c67cbf9ac52645c33fdf1eac176dd5a5e7d72a21f318d5]
(inline)
  7.80MB  0.63% 97.23%  7.80MB  0.63%
/VictoriaMetrics/lib/blockcache.(*cache).cleanPerKeyMisses
```

The output columns can change based on the type of profile, but they usually include:

- `flat` : the exclusive number of samples, bytes, or resources used by this function alone.
- `flat%` : what percentage of the total resource was used by this function alone.
- `cum` : The total amount used by this function and everything it called.
- `cum%` : the total percentage used by this function and all its callees.
- `sum%` : the running total of `flat%` from the top of the list to this row, showing how much of the total resource is covered so far.

This means that the top function,

`/VictoriaMetrics/lib/bytesutil.ResizeNoCopyNoOverallocate`, directly used 670.69 MB of memory, which is 53.89% of the total memory in this profile.

Since the cumulative value is also 670.69 MB, this means the function did not call any other functions that allocated extra memory. The `sum%` of 53.89% tells you that this one function alone accounts for more than half of the total memory usage covered in the profile report. At the top, you see that the report includes nodes responsible for 1209.99 MB, which is 97.23% of the total 1244.44 MB. 99 nodes with very low impact are left out, since their combined usage is too small to be important.

This clear separation of flat and cumulative numbers helps you find expensive operations (`flat`) and track down responsibility (`cum`). Put simply:

- A function with a high flat value is doing heavy work by itself.
- A function with a low flat but high cumulative value is mostly delegating work to other functions, but it is still responsible for that work. For example, a dispatcher function may have a low flat cost but high cumulative cost, because it calls several costly functions.

By default, the report sorts functions by their exclusive usage, the `flat` column. If you want to sort by cumulative usage, which includes a function and all the functions it calls, use the `-cum` flag.

```
$ (pprof) top10 -cum
```

This command lists the top ten functions based on their total resource use. To see more entries, just adjust the number:

```
$ (pprof) top -cum          # top 10, sorted by cumulative  
use  
$ (pprof) top20 -cum         # top 20, sorted by cumulative  
use  
$ (pprof) top -cum 20        # same as the line above
```

These commands highlight functions that are at the top of the heaviest call trees; meaning they are responsible, through all their callees, for the largest share of resource use.

#### `list` : Detailed Function Listing

The `list` command shows source code with performance data, for any functions matching the name or pattern you provide (for example, `list main.expensiveFunction`). Here's an example:

```
$ (pprof) list bytesutil.ResizeNoCopyNoOverallocate  
  
Total: 1.22GB  
ROUTINE =====  
VictoriaMetrics/lib/bytesutil.ResizeNoCopyNoOverallocate in  
/VictoriaMetricsPlay/VM/lib/bytesutil/bytesutil.go  
  670.69MB  670.69MB (flat, cum) 53.89% of Total  
          . . . . .  
          48:func  
  ResizeNoCopyNoOverallocate(b []byte, n int) []byte {  
    . . . . .  
    49:    if n <= cap(b) {  
    . . . . .      50:        return b[:n]  
    . . . . .    51:    }  
  670.69MB  670.69MB  52:    return make([]byte, n)  
    . . . . .  
    53:}  
    . . . . .  
    54:
```

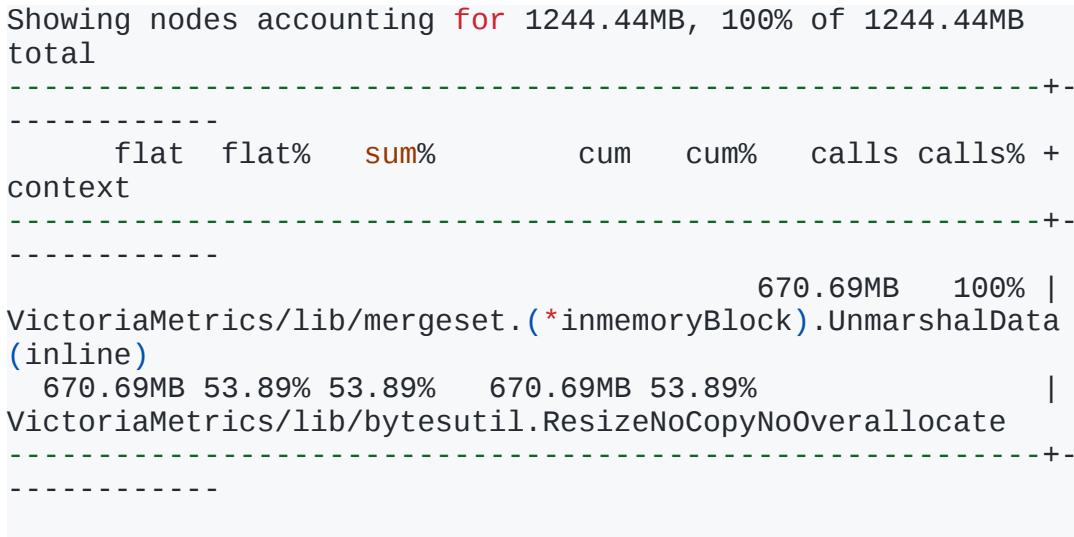
Now it is clear: nearly all of the memory used by `ResizeNoCopyNoOverallocate` (670.69 MB) comes from a single line: `return make([]byte, n)` at line 52.

#### `peek` : Call Path

The `peek` command shows you the immediate callers and callees of any function matching the given pattern. This lets you quickly see the context of a function—who is calling it, and what it calls in turn.

It is helpful for exploring the call graph without the clutter of a full web view:

```
$ (pprof) peek bytesutil.ResizeNoCopyNoOverallocate
```



In this example, the label (`inline`) after the caller function means the call from `UnmarshalData` to `ResizeNoCopyNoOverallocate` is inlined at the call site.

#### `disasm`: Disassemble Function

The `disasm` command in `pprof` displays the disassembly—actual assembly instructions—for any functions matching the pattern you give. It adds performance data from the profile, so you can see which instructions use the most CPU time or trigger other events.

```
$ (pprof) disasm bytesutil.ResizeNoCopyNoOverallocate
no matches found for regexp
bytesutil.ResizeNoCopyNoOverallocate
```

What happened here? Up until now, we have used `go tool pprof` with either HTTP endpoints or local profile files. But the `disasm` command needs the actual binary executable file that contains the compiled code.

This is necessary because disassembly depends on access to the real machine code in the binary. The profile file alone does not include the instructions—it only records profiling data.

To fix this, you need to re-run the `go tool pprof` command and provide the binary executable file:

```
$ go tool pprof ./vmstorage-prod
http://localhost:6060/debug/pprof/heap
```

Try running the command again:

```
$ (pprof) disasm bytesutil.ResizeNoCopyNoOverallocate  
no matches found for regexp  
bytesutil.ResizeNoCopyNoOverallocate
```

Still no result. In the previous section, we used `peek` to look up the immediate callers and callees, and we saw that `ResizeNoCopyNoOverallocate` is inlined at the call site inside `*inmemoryBlock.UnmarshalData`.

Inline functions are not disassembled because their code is compiled directly into the caller. To see the relevant assembly, you should disassemble the `UnmarshalData` function instead:

```
$ (pprof) disasm inmemoryBlock.*UnmarshalData  
  
Total: 1.22GB  
ROUTINE =====  
/VictoriaMetrics/lib/mergeset.(*inmemoryBlock).UnmarshalData  
    707.29MB  708.32MB (flat, cum) 56.92% of Total  
    . . .  
    . . . 7ac0cb: JA 0x7acada  
;bytesutil.go:50  . . . 7ac0d1: MOVQ 0x18(R8), R10  
;encoding.go:415  . . . 7ac0d5: JMP 0x7ac104  
;bytesutil.go:52  . . . 7ac0d7: LEAQ 0x13ae02(IP), AX  
    . . . 7ac0de: MOVQ CX, BX  
    670.69MB  670.69MB  7ac0e1: CALL runtime.makeslice(SB)  
;/VictoriaMetrics/lib/mergeset.  
(*inmemoryBlock).UnmarshalData bytesutil.go:52  
    . . . 7ac0e6: MOVQ 0x98(SP), CX  
;encoding.go:452  . . . 7ac0ee: MOVQ 0x178(SP), DX  
;encoding.go:424  . . . 7ac0f6: MOVQ 0x298(SP), R8  
;encoding.go:416  . . . 7ac0fe: MOVQ CX, R9  
;encoding.go:415  . . .
```

This output is detailed and shows a lot of assembly code, but what matters most here is the line for `bytesutil.go:52`, which is the call to the `runtime.makeslice` function. This should remind you of our earlier discussion in Chapter 3 about arrays, slices, strings, and maps.

## `web` : SVG in Web Browser

The `web` command creates a visual graph of your profile and opens it in a web browser. This command requires that you have Graphviz installed on your system:

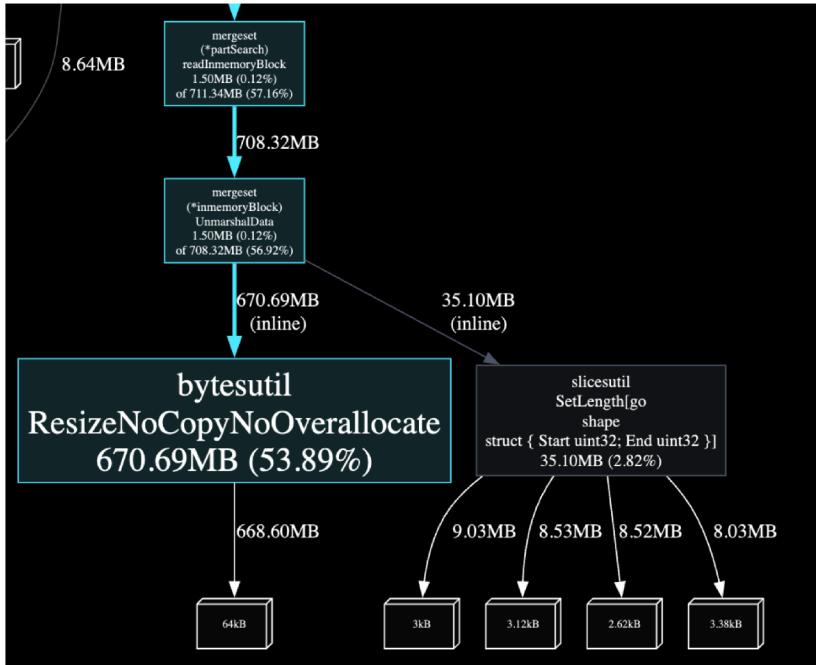


Illustration 212. Pprof web graph: function calls and allocations

In this graph, each function is shown as a node, and the calls between functions appear as edges. The size and color of each node reflect how much resource that function uses. This gives you a quick, visual way to see where your application is spending the most time or resources.

Using the web graph is especially helpful for spotting hotspots and understanding which functions are most expensive, all at a glance. It also helps you see how different functions are connected, without having to use text-based commands like `peek` or `tree`.

## HTTP UI Mode

Using `go tool pprof -http=[host]:[port]`, you can start a local web server that gives you an interactive interface for exploring profile data. This is an alternative to the regular command-line shell that `pprof` provides.

The web UI includes several useful visual tools:

- Interactive flamegraphs,
- Directed graphs showing call relationships,
- Top tables highlighting the most resource-intensive functions,
- Source code views with profiling data added right in,
- Disassembly views.

If you do not specify a port, `pprof` will pick a random free port. If you do not specify a host, it defaults to `localhost`.

```
$ go tool pprof -http=: cpu.prof
Serving web UI on http://localhost:64938
```

Usually, you will use a command like this:

```
go tool pprof -http=:8080 cpu.prof
```

This starts the web UI on port 8080, so you can open it at <http://localhost:8080>.

The web interface gives you much richer and clearer visualizations than the text-based shell. It is much easier to navigate large profiles and spot performance issues in your code. Everything we discussed about the command-line interactive shell becomes even more useful when you try it out in the HTTP-based `pprof` interface.

## Reading Call Graph Visualizations

Now you get a directed graph that visually shows how functions in your program are related and how much resource each one uses. Each function is a node in the graph. These nodes are connected by edges, which show who called whom and how much resource was used along those paths.

### Labels

Each node in the graph is labeled with several pieces of information. The function name is always shown, often with the full package path to avoid confusion.

Next to the name, you see the `flat`, `flat%`, `cum`, and `cum%` values:

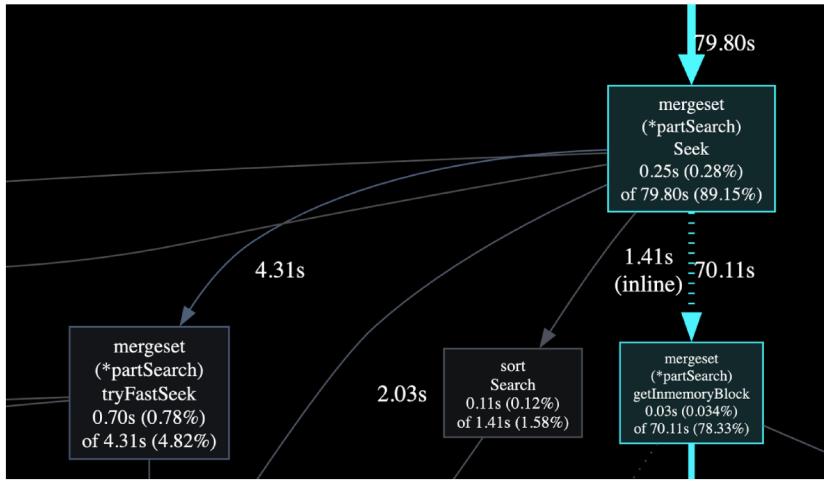


Illustration 213. Go pprof shows flat and cumulative CPU time

Let's look at the top node in a CPU profile graph:

- The node with the name `mergeset.(*partSearch).Seek` represents a function that was sampled many times.
- Next to the name, you see the `flat` value. This is the amount of CPU time spent only in this function. In this example, it is 0.25 seconds.
- The cumulative (`cum`) value is much larger, at 79.80 seconds. This total includes the time spent in this function and in all the functions it called, whether directly or indirectly.
- After the flat and cumulative values, you see their percentages. These numbers show what share of the total resource usage each value represents, across the whole profile.

#### Size & Color

The graph uses node size and color to help you spot important functions quickly:

- The size and font of a node are based on the flat value. If a function uses a lot of CPU or memory directly, its box is bigger and its label uses a larger font.
- Node color makes the graph easier to scan. The color goes from green to red. Green is for functions with low or negative resource use. Red is for functions that use the most resources. Gray is used for functions in the middle.

For example, the node `mergeset.(*partSearch).Seek` with 0.25s flat value is bigger than `mergeset.(*partSearch).getInMemoryBlock` with just 0.03s flat.

But it is smaller than `mergeset.(*partSearch).tryFastSeek`, which has 0.7s flat.

Even though `tryFastSeek` is bigger, its color is gray. This means it is not on the hot path. On the other hand, `Seek` has a low flat value but a high cumulative value, so it is on the hot path.

### Edges

The lines connecting the nodes are called edges. These edges show which functions call each other. Not all edges look the same. Some are solid, and some are dotted. Solid edges show a direct function call. Dotted edges show an indirect relationship—meaning the call goes through one or more middle functions, which are hidden from the graph to keep it readable.

The thickness of an edge shows how much resource flows along that path. Thicker edges mean more resource usage. The edge colors follow the same red-gray-green pattern as the nodes.

The color is based on how much resource flows through that edge, compared to the total.

### Nodelets

Nodelets are one of the most overlooked features in `pprof`. They are small box-shaped add-ons that attach to nodes in the graph, showing tag information.

They appear when a function has extra metadata, such as line numbers or memory allocation sizes. When there are many numeric tags, they are grouped into buckets (for example, "1MB..2MB") to keep things readable.

Let's look at a memory profile graph to make this clear:

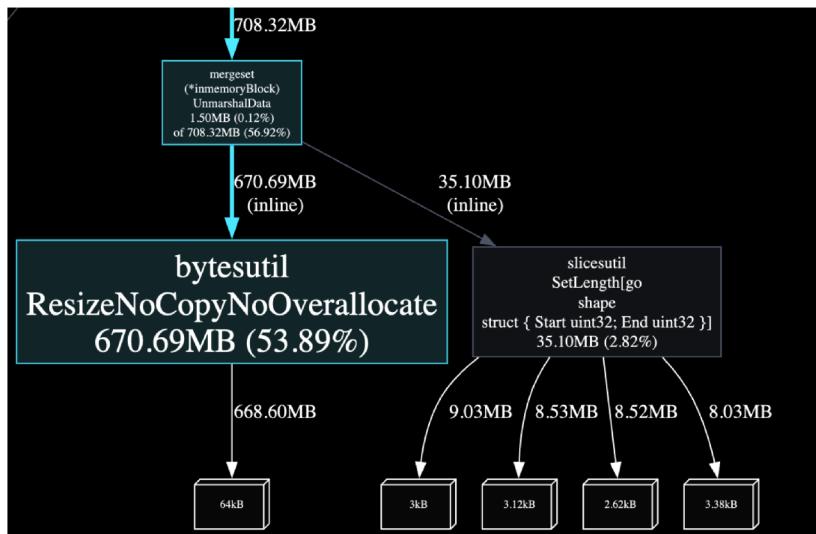


Illustration 214. Each nodelet represents a specific allocation bucket

Go's memory profiler does not just track which functions allocate memory; it also records the size of each allocation. These sizes are saved as numeric tags in the profile data, and each nodelet represents a specific allocation size linked to the function it is attached to.

For example, the 64kB nodelet attached to `ResizeNoCopyNoOverallocate` shows that this function is making many 64 KB allocations, which total 668.60 MB. The smaller nodelets (3kB, 3.12kB, 2.62kB, 3.38kB) attached to `slicesutil.SetLength` show different allocation sizes made by that function.

Nodelets are extremely helpful for memory optimization, because they let you see:

- The exact allocation sizes made by each function,
- Which allocation patterns are using the most memory,
- Potential inefficiencies, like making allocations just above a power-of-2 size.

After sorting, only the top four nodelets are shown for each node. This helps keep the graph simple and avoids clutter, especially for nodes with lots of tags or metrics.

## Nodelet values are not arbitrary

The values in the nodelets are not random. They match up with specific size classes. For example:

- 3kB is size class 41 (3072 bytes)

- 3.12kB is size class 42 (3200 bytes)
- 2.62kB is size class 40 (2688 bytes)

These numbers are the same size classes used by Go's memory allocator (explained in Chapter 7). When nodelets are shown in `pprof`, the raw byte counts are converted to human-readable units, but they directly show the size class used for the allocations.

Leaf nodes—functions that do not call any other functions—are often the last stop in the call stack before work is finished. If these leaf nodes have high flat values, it means they are doing most of the heavy work.

## Reading Flame Graphs

A flame graph in `pprof` is a way to visualize profiling data that makes it easy to see the execution stack of your program. Each function is shown as a horizontal bar, and the width of that bar represents how much resource (CPU time, memory, etc.) that function used.

You can view a flame graph in `pprof` by clicking the 'View' tab and choosing 'Flame Graph'.

Here's what a memory profile flame graph looks like:

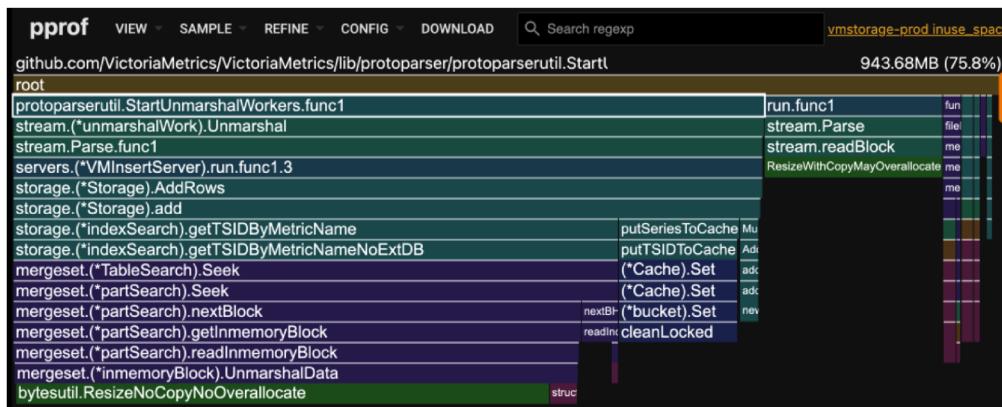


Illustration 215. Bottom-up flame graph reveals allocation hotspots

The image above is a "bottom-up" or "icicle" flame graph. In this view, the root of the program is at the top, and the call stack grows downward.

The x-axis of the graph shows the total memory allocated by your program during profiling. The width of each box is proportional to the number of bytes allocated by that function, either directly or by the functions it calls. A very wide box means that function (and its callees) is responsible for a large part of memory allocation, which makes it a good place to look for optimizations.

Each vertical column of boxes is a call stack. The root at the top is the function where all allocation starts. The top box shows the function that triggered the allocations.

Each box sums up all allocations for that function, both from its own code and from its callees (which are shown as child boxes).

So, how do you know how much memory is allocated by the function itself?

If the `UnmarshalData` function has a wide box, and its child `bytesutil.ResizeNoCopyNoOverallocate` is almost as wide, then most allocations are happening in the child. But if `UnmarshalData` is wider than its child, the extra width shows allocations made directly by `UnmarshalData`.

Each function is given a unique color based on its position in the source data. The colors are just to make different functions easy to tell apart visually. It does not represent performance or resource usage.

## Memory (Heap/Alloc) Profiling

The memory profiler in Go is a sampling-based system that watches memory allocations and deallocations while your application is running. Its main purpose is to help you see memory usage patterns, find leaks or inefficiencies, and improve performance.

Go does not record every single memory allocation. That would be too slow for most programs. Instead, it uses a statistical sampling approach. This sampling is controlled by the `MemProfileRate` variable found in `runtime/mprof.go`. By default, this value is set to 512 KB. This means the profiler records one sample for about every 512 KB of memory allocated.

Figure 134. MemProfileRate (`src/runtime/mprof.go`)

```
var MemProfileRate int = 512 * 1024
```

So, for every 512 KB of allocated memory, the profiler will record one sample. If you lower the `MemProfileRate`, the profiler will take more samples.

You can tune this value to balance accuracy and overhead. There are two special values for `MemProfileRate`:

- Setting it to `1` means the profiler samples every byte allocated.
- Setting it to `0` means no memory profiling at all.

Each processor `P` in Go has a counter called 'next sample' (`p.mcache.nextSample`). This counter tracks how many more bytes this processor can allocate before it must record the next sample for the heap profile.

When Go decides when to record the next heap profile sample, it does not just add a fixed 512 KB to the counter. Instead, it draws a new random number from an exponential distribution. The average of these numbers is `MemProfileRate`, which is 512 KB by default.

This means that the distance between samples is random. Sometimes Go might take a sample after only 120 KB, sometimes after 260 KB, or even after 1.3 MB. Most of the time, the interval will be smaller than 512 KB, but sometimes it will be larger. On average, the intervals work out to about 512 KB over time.

This randomness helps avoid predictable patterns and keeps the sampling rate close to the desired average.

When an allocation happens in `mallocgc` (the main allocation function), the runtime checks if this allocation should be sampled for profiling:

```
func mallocgc(size uintptr, typ *_type, needzero bool)
unsafe.Pointer {
    ...
    fullSize := span.elemsize
    if rate := MemProfileRate; rate > 0 {
        if rate != 1 && fullSize < c.nextSample {
            c.nextSample -= fullSize
        } else {
            profilealloc(mp, x, fullSize)
        }
    }
    ...
}
```

The sampling process uses `nextSample` like a countdown. Each allocation subtracts its size from `nextSample`. When it drops below zero, the allocation is sampled and `nextSample` is reset to a new random value.

Keep in mind that memory is managed in fixed-size slots. The profiler always uses the full slot size, not just the user-requested size.

Therefore, if you allocate 7 KB, Go's memory allocator will use an 8 KB slot. That is the smallest size class that fits 7 KB. When memory is freed, it is also freed at the slot size level (8 KB in this example), not the original size requested (7 KB). To be consistent, the profiler uses the same slot size for both allocation and deallocation events.

## Bucket

The memory profiler needs to track where each allocation comes from in your code. It does this by linking each allocation to a bucket.

A bucket represents a specific place in the code where an allocation happened, along with the size of that allocation. Let's look at an example to set the stage:

```
func main() {
    for i := range 1024 {
        alloc1(512)
        alloc1(1024)

        alloc2(2048)
    }

    runtime.GC()

    f, _ := os.Create("heap.prof")
    pprof.Lookup("allocs").WriteTo(f, 0)
}

//go:noinline
func alloc1(n int) []byte {
    return make([]byte, n)
}

//go:noinline
func alloc2(n int) []byte {
    return make([]byte, n)
}
```

Memory profiles show statistics as of the most recent garbage collection. By forcing a GC, you get a more accurate view of your app's typical memory use. For `net/http/pprof`, you can force a GC by adding `gc=1` to your query. Just remember, forcing GC comes with a performance cost.

In this example, there are three buckets: `alloc1(512)`, `alloc1(1024)`, and `alloc2(2048)`. The function `alloc1` is called twice in the loop, but with different sizes. So, you get three unique combinations of allocation site and size.

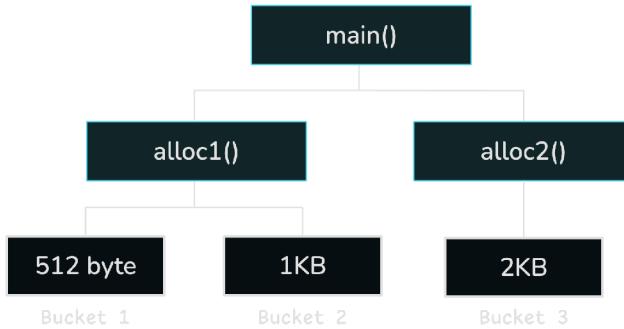


Illustration 216. Allocation buckets group by callsite and size

Imagine each bucket has a counter that tracks allocations and deallocations. Every time you allocate memory, the counter goes up. When memory is freed, the counter goes down. But this is not perfect.

The problem is timing. Allocations are counted right away, but frees only happen when garbage collection runs. This means the numbers can look wrong. It can seem like you are using more memory than you really are. Sometimes, it might even look like there is a memory leak, as the counter grows until a GC runs and suddenly drops.

There is another problem. If a garbage collection starts while you are getting profile data, you can get a mix of old and new information. Some memory might be freed, but not all, leading to a race condition.

To fix these problems, the Go memory profiler uses a three-cycle system. This gives accurate and steady profiling data for allocations and deallocations.

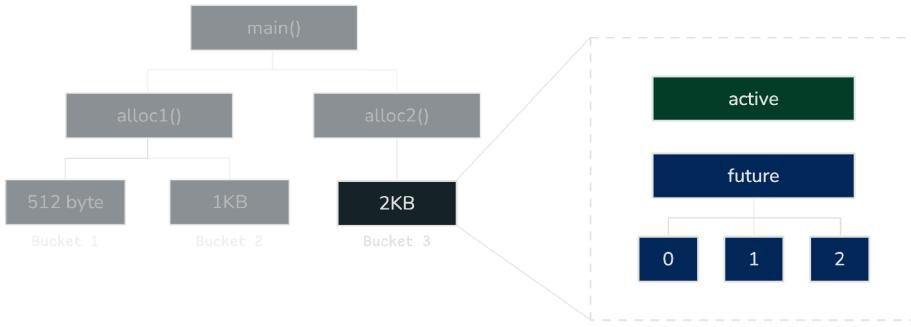


Illustration 217. Go memory profiler uses three-cycle allocation tracking

The `active` profile is the stable, published view of memory usage. This is what outside tools see. It contains memory stats that have been processed and checked for accuracy. When you request a memory profile, the data comes from the `active` profile.

The `future` cycles are where accounting happens. The profiler keeps a global cycle counter that advances during each garbage collection. Each allocation site has a ring buffer with three cycles. These are called `C`, `C+1`, and `C+2`, where `C` is the current cycle. These cycles temporarily hold allocation and deallocation events until they are moved into the `active` profile.

This is a rotating buffer. The current cycle number always increases (and wraps around when needed). To find the right slot for the current cycle, we use the following formula:

```
index := cycle % 3
```

Each cycle keeps track of four things:

- How many bytes were allocated
- How many bytes were deallocated
- How many allocation events occurred
- How many deallocation events occurred

```
type memRecordCycle struct {
    allocs, frees      uintptr
    alloc_bytes, free_bytes uintptr
}
```

These four numbers explain why, when you use `pprof` to look at memory profiles (`allocs` or `heap`), you will see four metrics in the output:

- Memory that is in use and not freed yet: `inuse_space = alloc_bytes - free_bytes`
- Number of objects still in use: `inuse_objects = allocs - frees`
- Total memory allocated during the program's lifetime, even if freed: `alloc_space = alloc_bytes`
- Total number of allocations during the program's lifetime: `alloc_objects = allocs`

Both `heap` and `allocs` profiles in Go use the same data, but show different default views. The `heap` profile defaults to `inuse_space`. The `allocs` profile shows `alloc_space` by default. You can see all four metrics from either profile.

So, how does this 3-cycle system actually work?

When memory is allocated, the event is recorded in cycle `C+2`. When a garbage collection happens and memory is freed, that deallocation is recorded in cycle `C+1` (which was the previous `C+2` before GC started).

Here's a simple example. Imagine your program starts at cycle `C=0`. The profiler tracks three cycles: `C=0`, `C+1=1`, and `C+2=2`. All are empty at the start. The `active` profile also shows zero allocations and zero bytes.

Now suppose the allocation `alloc2(2048)` gets sampled one time. This allocation goes into cycle `C+2` (cycle 2):



Illustration 218. New allocation tracked in profiler's C+2 buffer

Now cycle 2 shows: 1 allocation, 2048 allocated bytes, 0 frees, 0 freed bytes. Cycles 0 and 1 stay empty. The `active` profile is still all zeros.

Next, the program keeps running and triggers a garbage collection. The profiler increases the global cycle counter from 0 to 1. The cycles rotate: what was `C+2`

becomes  $C+1$ , and what was  $C+1$  becomes  $C$ .



Illustration 219. Cycle indices rotate as GC increments global counter

Suppose during the sweep phase, that 2 KB allocation is no longer reachable and is freed. This free is recorded in cycle  $C+1$  (which is now cycle 2):



Illustration 220. Freed memory is recorded in  $C+1$  cycle

After this, cycle 2 has: 1 allocation, 2048 allocated bytes, 1 free, 2048 freed bytes.

When the sweep phase is done, the profiler copies the data from cycle  $C+1$  (cycle 2) into the `active` profile. The `active` profile now has the data from cycle 2:

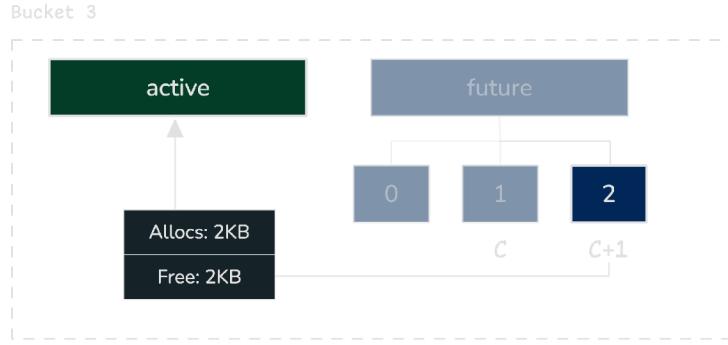


Illustration 221. Cycle C+1 data promoted to active profile

This is how the Go memory profiler uses the cycle system to avoid bias and provide smooth, reliable statistics. The cycle keeps rotating, and the `active` profile always gives you the most recent snapshot of memory usage. For best accuracy, call `runtime.GC()` before saving a profile.

## Sampling & Scaling

We also need to consider the scaling factor. If the Go memory profiler simply showed that `alloc2(2048)` only allocated 2048 bytes in the profile, that would be misleading. With a 2048-byte allocation, you would need about 256 allocations to reach the 512 KB sampling rate in the ideal scenario.

Earlier we explained that `MemProfileRate` (default 512 KB) controls how often allocations are sampled. Go uses randomized sampling with an exponential distribution.

This exponential distribution is important. It creates a process called a "Poisson process," which means random events that happen independently at a constant average rate. You do not need to know the math, but here is a simple way to think about it.

With the default `MemProfileRate` of 512 KB and a 2 KB allocation, the chance of sampling is about:

$$1 - \exp(-2048/512000) = 0.00399201065 \approx 0.004 \text{ (0.4%).}$$

So, 2 KB allocations have only a 0.4% chance of being sampled at a 512 KB sampling rate. When one of these allocations is sampled, Go assumes it represents about 256 other allocations of the same size that were not sampled. Go automatically scales the allocation size when showing the profile:

```

func scaleHeapSample(count, size, rate int64) (int64, int64)
{
    if count == 0 || size == 0 {
        return 0, 0
    }

    if rate <= 1 {
        return count, size
    }

    avgSize := float64(size) / float64(count)
    scale := 1 / (1 - math.Exp(-avgSize/float64(rate)))

    return int64(float64(count) * scale),
    int64(float64(size) * scale)
}

```

So for a 2 KB allocation in the 2 KB bucket and a 512 KB sampling rate, the scale is:

```
scale = 1 / (1 - exp(-2048/524288)) ≈ 256.5
```

This scaling applies to both `alloc_space` and `alloc_objects` metrics:

```
estimated allocations = 2048 * 256.5 ≈ 525312
estimated objects = 1 * 256.5 ≈ 256
```

This is what you will see in the memory profile output:

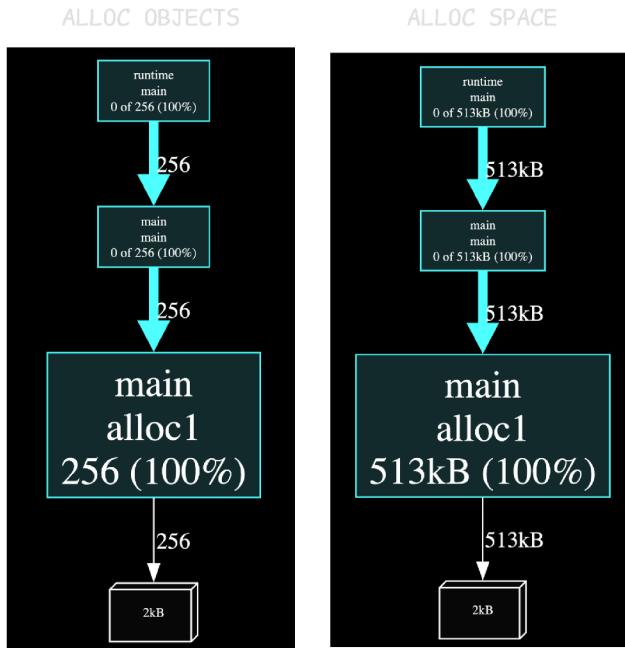


Illustration 222. Estimated allocation objects vs allocation space in profile

So, when you allocate 2048 bytes and it gets sampled, the Go memory profiler scales it up to about 513 KB in the report.

This statistical sampling approach is also why the memory profiler is always running from the start of the program, not just at the time you ask for a profile.

You don't need to worry about the memory profiler slowing down your program if you do not use it. The Go linker checks whether the function `runtime.memProfileInternal` is used. If it is not, Go sets `MemProfileRate` to 0 and disables the memory profiler.

The function `runtime.memProfileInternal` collects and creates memory profile data. It is called by the `net/http/pprof` package when you request a memory profile, or when you use `pprof` or `runtime` to write the profile to a file.

## CPU Profiling

### Process-Wide CPU Profiling

The CPU profiler works by quickly taking "snapshots" of what each thread in your program is doing. It sets up a periodic timer. On Unix systems, this uses the operating system feature called `ITIMER_PROF`. On other platforms, the Go runtime picks the closest alternative.

The timer is set so it goes off one hundred times per second by default. This means roughly every 10 milliseconds of CPU time that your process is running. When the timer expires, the kernel sends your process a special signal. On Unix, this signal is called `SIGPROF`. The Go runtime installs a small signal handler for that signal.

A signal handler is a regular function you register with the operating system. When the kernel sends that signal to your process, it pauses whatever your program is doing, runs your handler, then returns to normal execution after the handler is done.

Graceful shutdown is probably the most common use for a signal handler (see Graceful shutdown in Go - VitoriaMetrics [\[gcfsd\]](#)). Many programs install

handlers for `SIGINT` (from pressing `Ctrl-C`) or `SIGTERM` (from systemd or Docker). Inside those handlers, programs close sockets, flush logs, save state, and exit cleanly. Other handlers include `SIGHUP` to reload configuration and `SIGCHLD` to notice when a child process has finished.

The handler stops the thread that was running when the signal arrived and records a stack trace from the current program counter.

The 100 Hz (100 times per second) CPU profiling interval is a good default for most uses. It gives a balance of accuracy and low overhead. You can change the interval using `runtime.SetCPUProfileRate`. The `pprof.StartCPUProfile` function calls this internally.

```
func StartCPUProfile(w io.Writer) error {
    const hz = 100

    cpu.Lock()
    defer cpu.Unlock()
    if cpu.done == nil {
        cpu.done = make(chan bool)
    }
    // Double-check.
    if cpu.profiling {
        return fmt.Errorf("cpu profiling already in
use")
    }
    cpu.profiling = true
    runtime.SetCPUProfileRate(hz)
    go profileWriter(w)
    return nil
}
```

If you set the rate to `0`, the timer is disabled:

```
func StopCPUProfile() {
    cpu.Lock()
    defer cpu.Unlock()

    if !cpu.profiling {
        return
    }
    cpu.profiling = false
    runtime.SetCPUProfileRate(0)
    <-cpu.done
}
```

When a `SIGPROF` signal is pending for your process, the kernel must decide which thread should handle it. For most signals, any thread that does not have the signal blocked can handle it. The kernel prefers to pick threads that are already running, to avoid extra scheduling work.

This selected thread is temporarily paused by the kernel (the kernel decides on a safe point for the thread). The kernel saves the thread's execution context, such as the registers, program counter, and stack pointer, then sends control to the registered signal handler. On Unix-like systems, the signal handler's main job is to record the stack trace of the code that was running on the interrupted thread. As different threads run and use CPU time, each one gets interrupted by `SIGPROF` from time to time.

On Windows, Go's CPU profiler works differently. It uses a dedicated profiler thread to capture the stack trace from other threads that are eligible. This is a different design than the signal-based method used on Unix-like systems.

The CPU profiler has a hard limit for how many call stack frames it can capture. In the Go runtime code, this constant is set to 64. This means the profiler will only record up to 64 levels of nested function calls.

Figure 135. maxCPUProfStack (runtime/cpuprof.go)

```
const (
    maxCPUProfStack = 64

    profBufWordCount = 1 << 17 // 131072

    profBufTagCount = 1 << 14 // 16384
)
```

So, if a function call stack is deeper than 64, only the most recent 64 calls are shown in the profiler output. The oldest calls in the stack will be left out. In practice, this is almost never a problem. Most real-world programs do not have such deep call stacks, and performance bottlenecks usually happen near the most recent calls.

Each processor (P) keeps a small buffer in memory to hold the raw samples. When the Go runtime's signal handler is triggered, it walks the stack of the thread that was interrupted and adds the stack trace to the buffer for the P that thread is running on:

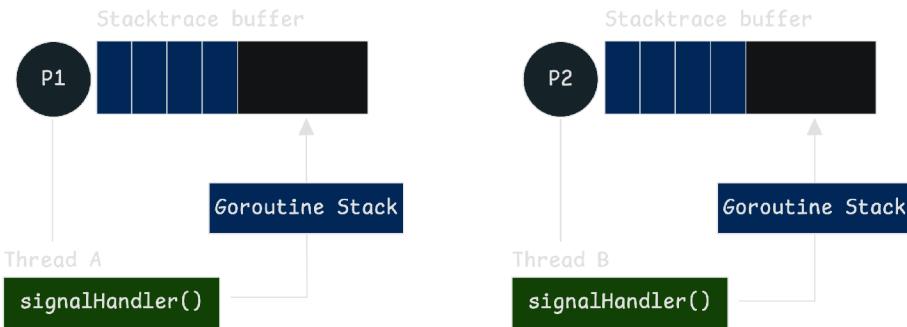


Illustration 223. Each processor maintains its own stacktrace buffer

Then who reads from these buffers? If you look at the `pprof.StartCPUProfile` function, you will see that it starts a goroutine called `profileWriter` to read from these buffers and write the data to the given `io.Writer`:

Figure 136. profileWriter (runtime/pprof/pprof.go)

```
func profileWriter(w io.Writer) {
    b := newProfileBuilder(w)
    var err error
    for {
        time.Sleep(100 * time.Millisecond)
        data, tags, eof := readProfile()
        if e := b.addCPUData(data, tags); e != nil
&& err == nil {
            err = e
        }
        if eof {
            break
        }
    }
    if err != nil {
        panic("runtime/pprof: converting profile: "+
err.Error())
    }
    b.build()
    cpu.done <- true
}
```

The writer goroutine wakes up about every 100 milliseconds (one-tenth of a second) and gets a batch of samples that have built up in the per-P buffers since the last read:



Illustration 224. Profile writer merges stack traces from all buffers

It passes the batch to the profile builder, which merges and compresses the data in memory. The builder writes to the output only after all sampling is finished.

For every sample, the profile builder checks the sample's stack trace (and tag, if there is one) in a hash table. If a sample with the same trace is already in the table, it just increments that entry's count. If not, it creates a new entry. This way, while the writer goroutine is still in memory, before any bytes are written out, it combines all identical stack traces into one entry with a counter to show how many times it happened.

## Thread-Wide CPU Profiling

With process-wide CPU profiling, a single timer sends `SIGPROF` signals to random eligible threads in the process. This can make the results less accurate because threads that use a lot of CPU time might not get their fair share of signals. Some threads, like those used for cgo calls or system threads, might not show up in the profile as much as they should.

To solve these problems, Go adds thread-specific CPU profiling on platforms that support it (mainly Linux). In this model, each thread gets its own timer, which sends `SIGPROF` signals only to that thread. Go supports both process-wide and thread-specific profiling on systems where it is possible. The global timer still works for backward compatibility and for threads that cannot use their own timers.

When a thread has its own timer (`mp.profileTimerValid == true`), it only accepts `SIGPROF` signals from that timer (created using `timer_create`). It ignores signals from the process-wide timer (`setitimer`).

This helps avoid double-counting CPU usage:

```

// decide if a SIGPROF signal should be used for profiling
//go:nosplit
func validSIGPROF(mp *m, c *sigctxt) bool {
    code := int32(c.sigcode()) // get the reason why the
signal was sent
    setitimer := code == _SI_KERNEL // signal came from
a process-wide timer
    timer_create := code == _SI_TIMER // signal came
from a thread-specific timer

    if !(setitimer || timer_create) {
        return true // allow signals from unknown or
external sources
    }

    if mp == nil {
        return setitimer // if we don't know the
thread, only trust process-wide signals
    }

    if mp.profileTimerValid.Load() {
        return timer_create // if the thread has its
own timer, only trust that
    }

    return setitimer // otherwise, only trust the
process-wide timer
}

```

Windows does not support thread-specific CPU timers like Linux. Instead, a single central thread samples all the other threads. The benefit of this method is its simplicity: only one timer is needed, no matter how many threads you are profiling.

## Interpreting CPU Profiles

Let's look at a simple example to show how CPU profiling works in Go:

```

func hot() {
    // Simulate CPU-bound work
    x := 0
    for i := range 100000000 {
        x += i
    }
    fmt.Println(x)
}

func cold() {

```

```

        time.Sleep(100 * time.Millisecond)
    }

func indirectHot() {
    hot()
}

func main() {
    f, err := os.Create("cpu.prof")
    if err != nil {
        panic(err)
    }
    defer f.Close()

    if err := pprof.StartCPUProfile(f); err != nil {
        panic(err)
    }
    defer pprof.StopCPUProfile()

    for range 5 {
        hot()
        cold()
        indirectHot()
    }
}

```

This example defines three functions:

- `hot` : a CPU-heavy function that sums integers in a tight loop.
- `cold` : sleeps for 100 milliseconds to represent waiting or I/O.
- `indirectHot` : just calls `hot` .

In `main` , we loop five times and alternate between the CPU-intensive work (`hot` and `indirectHot` ) and the sleep (`cold` ). The program collects a CPU profile while running, which you can later inspect with tools like `go tool pprof` . When you analyze `cpu.prof` , you will see two main measurements:

- **Samples count:** Shown as "samples" (unit: count) in the profile data. This is just the number of times a particular stack trace was observed. It tells you how many times the profiler saw each function or code path.
- **CPU time:** Shown as "cpu" (unit: nanoseconds). This is the sample count multiplied by the sampling rate.

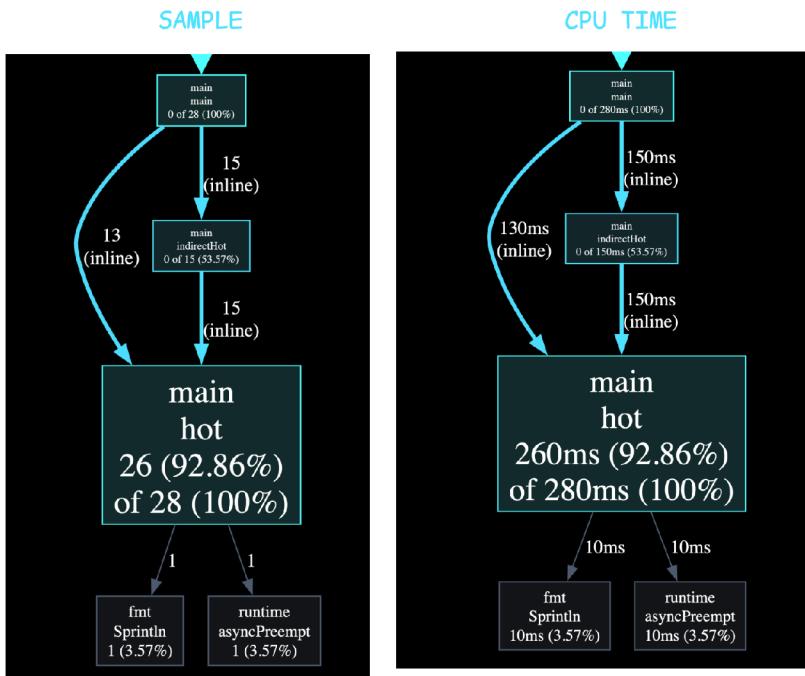


Illustration 225. Most CPU samples are from the hot function

By default, `pprof` shows CPU time in nanoseconds, but you can view raw sample counts with the `-sample_index=samples` flag.

Looking at the profile output, it shows the stack trace where the sampling signal (`SIGPROF`) was delivered. The trace walks up from the leaf function (the function actively running) to the root. Every function in the stack trace is part of the call stack, but only the leaf function is using CPU at that exact moment.

If you see a sample count like "0 of 28" for `main.main`, this means zero samples where `main.main` was the leaf function. However, it is present in all 28 call stacks, so it appears in 100% of the samples. This highlights the difference between self-time and total-time (also called inclusive time). Self-time is only when that function was the leaf. Total-time is all time spent in that function and everything it calls.

There are two ways to reach `main.hot`: one is a direct call from `main`, the other is through `indirectHot`.

In the sample, `main.hot` is the leaf for 26 out of 28 samples, which is 260ms out of 280ms in the time view (sampling at 100Hz means each sample is about 10ms). Sometimes the leaf function is `fmt.Sprintln` (which we added on

purpose), or `runtime.asyncPreempt` (a Go helper that can preempt long-running goroutines).

In short, the graph from the profile matches what our loop is doing: most time spent in `hot`, very little in `cold`, and some calls reaching `hot` through `indirectHot`, all running under `main.main`.

## Mutex (Lock Contention) Profiling

Mutex profiling works by watching when goroutines are blocked while trying to acquire a mutex. This includes `sync.Mutex`, `sync.RWMutex`, and internal runtime locks. Both `mutex` and `block` profiling are disabled by default, even if you use the `net/http/pprof` package. You need to turn them on manually by calling `runtime.SetMutexProfileFraction` and `runtime.SetBlockProfileRate`.

### Understanding Contention Time

When a goroutine tries to lock a `sync.Mutex` that is already locked, it has to wait. The profiler tracks how long it waits.

Once the mutex is released and the waiting goroutine acquires the lock, the profiler records a sample. Importantly, the profiler records the stack trace of the goroutine that released the lock, not the one that was waiting. In other words, mutex profiling assigns the blame to the locker, not the waiter. This is intentional, because the profiler aims to answer the question, "*Who is holding the lock long enough to block others?*" The code that holds the mutex while others are waiting is the source of contention.

Each sample includes:

- The stack trace of the code that unlocked the mutex.
- The number of times that stack trace caused contention.
- The total wait time experienced by other goroutines while the lock was held. This wait time is credited to the unlock point, since that is where contention ends and the blame is assigned.

This can be confusing when you first look at the profile output. Here is an example:

```

var mtx sync.Mutex
var wg sync.WaitGroup

func acquire(n time.Duration) {
    mtx.Lock()
    defer mtx.Unlock()

    time.Sleep(n)
    wg.Done()
}

func main() {
    runtime.SetMutexProfileFraction(1)

    wg.Add(5)
    go acquire(1 * time.Second) // gowrap1
    go acquire(2 * time.Second) // gowrap2
    go acquire(3 * time.Second) // gowrap3
    go acquire(4 * time.Second) // gowrap4
    go acquire(5 * time.Second) // gowrap5

    wg.Wait()

    // Write mutex profile to file
    f, err := os.Create("mutex.prof")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    if err := pprof.Lookup("mutex").WriteTo(f, 0); err
    != nil {
        log.Fatal(err)
    }
}

```

Assume the goroutines start in this order: `gowrap1`, `gowrap5`, `gowrap2`, `gowrap3`, `gowrap4`. The profile shows two views based on the samples: `contention` and `delay`:

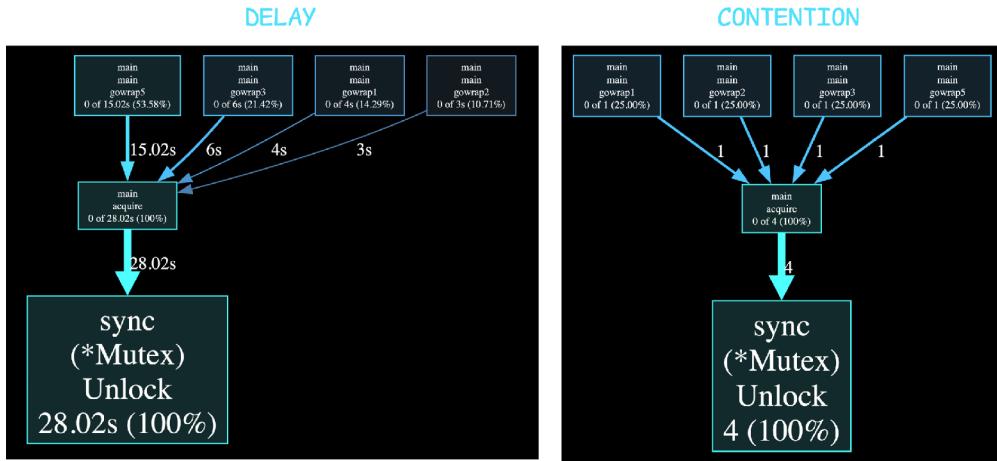


Illustration 226. Delays accumulate as goroutines hold the lock

`gowrap1` starts first, gets the mutex right away, and releases it after one second. But the graph shows 4 seconds of contention for `gowrap1`. Why?

As explained above, the mutex profile shows blame, not the waiting time for `gowrap1` itself. The 4 seconds is the sum of the time that `gowrap2`, `gowrap3`, `gowrap4`, and `gowrap5` had to wait while `gowrap1` was holding the lock. Each of these waited one second, so the total is 4 seconds. This blame is placed on `gowrap1` for making others wait.

`gowrap5` starts second, grabs the mutex, and holds it for 5 seconds. While `gowrap5` is holding the lock, `gowrap2`, `gowrap3`, and `gowrap4` are all blocked for 5 seconds each. This adds up to 15 seconds of contention, all attributed to `gowrap5`.

Note that, as of Go 1.23, mutex profiling can sometimes show odd values for other goroutines in this kind of simple example. This is due to a bug in the profiler, so you may see some jitter or unexpected results.

## Sampling Fraction

The best way to enable mutex profiling is to call `runtime.SetMutexProfileFraction(1)` at the start of your `main` function.

This gives you high-fidelity mutex profiling, which is especially helpful when you are debugging or tuning performance:

```

func main() {
    runtime.SetMutexProfileFraction(1)

    // ...
}

```

Internally, every time a goroutine unlocks a `sync.Mutex`, the runtime decides whether to record a profile sample. The argument to `SetMutexProfileFraction(n)` controls the probability: on average, one out of every `n` mutex contention events is recorded.

- If you use `1`, the profiler captures every contention event, giving you complete data, but with more runtime overhead.
- If you use `0`, mutex profiling is turned off.
- If you use a larger number, the profiler runs with less overhead but may miss some events.

The function where this decision is made is called `mutexevent`. It runs after the lock has been released and a waiting goroutine wakes up:

```

func mutexevent(cycles int64, skip int) {
    if cycles < 0 {
        cycles = 0
    }
    rate := int64(atomic.Load64(&mutexprofilerate))
    if rate > 0 && cheaprrand64()%rate == 0 {
        saveblockevent(cycles, rate, skip+1,
                      mutexProfile)
    }
}

```

For example, if you call `SetMutexProfileFraction(100)`, Go will record, on average, 1 out of every 100 mutex contention events. This is simple random sampling: each event has a 1% chance of being picked. The same rule applies for any value: if you use `n`, each contention event has a `1/n` chance to be recorded.

For most internal profiling like `mutex` and `block` profiles, Go does not use the real-time clock (`time.Now` or system nanoseconds). Instead, it uses CPU cycle counters, measured with `cputicks()`. A cycle is not a fixed time unit. It is the number of CPU clock cycles, and the length of a cycle depends on your processor's speed.

For example, on a 2.5 GHz CPU, one cycle is about 0.4 nanoseconds (1 / 2.5 billion seconds). So, on faster CPUs, the value from `cputicks()` grows faster.

Go converts these cycle counts into time units for your reports (like those from `go tool pprof`) using the `ticksPerSecond()` value, which Go estimates when the program starts. So when you see blocking times shown as nanoseconds or milliseconds in a profile, Go has already translated those numbers from cycles to time.

## Defer Interactions

What happens if you use two different mutexes in the same function? Does the profiler aggregate the time for that function or separate it?

Go's mutex profiler does separate the mutexes internally, but in the graph it shows the total aggregated time for the function. If you check the `source` view, it will show the blocked time for each `mtx.Unlock()` call.

However, if you use `defer mtx.Unlock()`, then the times will be combined. Instead of showing how much time is spent on each individual `mtx1.Unlock()` or `mtx2.Unlock()`, the profiler will report the total time at the exit point of the function.

Here's an example using two mutexes in the same function, with unlocks using deferred calls:

```
Total:          0      40.02s (flat, cum)   100%
    ...
    17           .      .
    func acquire(n
time.Duration) {
    18           .      1.24ms
    if rand.Intn(2) == 0 {
    19           .
    mtx.Lock()
    20           .
    defer mtx.Unlock()
    21           .
    } else {
    22           .
    mtx2.Lock()
    23           .
    defer mtx2.Unlock()
    24           .
    }
    25           .
    26           .
    time.Sleep(n)
    27           .
    wg.Done()
    28           .      40.01s
    29           .
```

The profiler does not keep track of which specific mutex was locked. It only records the stack trace at the time of unlock. Because both `mtx.Unlock()` and `mtx2.Unlock()` are called from the same location (the deferred function runs at the end of `acquire()`), the profiler groups them under the same stack trace. Their contention times are mixed together in the profile. That is why the total contention is shown only once, even if you have two separate critical sections with two different mutexes.

If you want to tell the difference between mutexes in your profiles, you should unlock them at different points in your code, so that the stack traces are not the same. For example, avoid using `defer` for unlocking and instead manually unlock each mutex in separate code branches. This will make the profiler show contention for each unlock location separately.

## Block (Wait) Profiling

The block profile in Go records how long goroutines are blocked while waiting on synchronization primitives like channels, `select` statements, and many other mechanisms that use the runtime semaphore (`runtime.sema`) behind the scenes. This includes things like `sync.Mutex`, `sync.RWMutex`, `WaitGroup`, and even I/O operations on network connections or files.

Block profiling helps you see where goroutines are getting stuck. The profile shows the call site where the block happens and how long it typically lasts. Importantly, it shows this from the perspective of the waiter—the goroutine that is waiting.

Let's revisit our previous example:

```
var mtx sync.Mutex
var wg sync.WaitGroup

func acquire(n time.Duration) {
    mtx.Lock()
    defer mtx.Unlock()

    time.Sleep(n)
    wg.Done()
}
```

```

func main() {
    runtime.SetBlockProfileRate(1)

    wg.Add(5)
    go acquire(1*time.Second) // gowrap1
    go acquire(2*time.Second) // gowrap2
    go acquire(3*time.Second) // gowrap3
    go acquire(4*time.Second) // gowrap4
    go acquire(5*time.Second) // gowrap5
    wg.Wait()

    // Write block profile to file
    f, err := os.Create("block.prof")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    if err := pprof.Lookup("block").WriteTo(f, 0); err
!= nil {
        log.Fatal(err)
    }
}

```

The result is now easy to understand. Suppose the goroutines execute in this order: `gowrap1`, `gowrap5`, `gowrap2`, `gowrap3`, `gowrap4`:

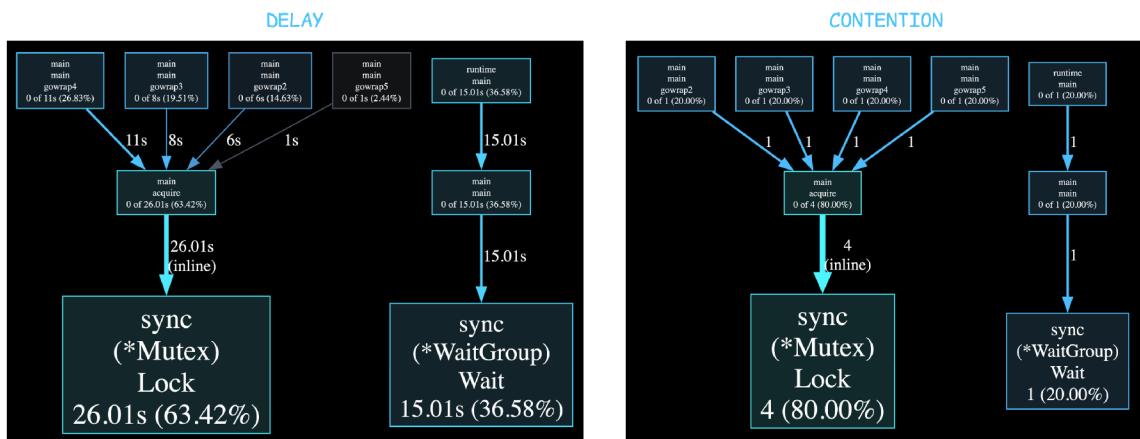


Illustration 227. Block profile shows where goroutines wait longest

Here is how the profile output breaks down for each goroutine:

- `gowrap1` acquires the lock right away and does not block, so it does not show up in the block profile.
- `gowrap5` starts at the same time as the others, but must wait for `gowrap1` to finish its 1-second hold. This 1-second wait is recorded in the block profile.

- `gowrap2` starts blocked and must wait for both `gowrap1` and `gowrap5` to finish, totaling 6 seconds of waiting before it gets the lock.
- `gowrap3` waits even longer, behind `gowrap2`, and ends up with 8 seconds of waiting.
- `gowrap4` is last in line and must wait behind everyone else, so it records 11 seconds of blocked time.

All of this waiting is measured from the waiter's perspective. When all the blocked times are added up, you get 26.01 seconds of waiting time, all attributed to `sync.(*Mutex).Lock`.

On the right side of the graph, you see `WaitGroup.Wait`, showing 15.01 seconds. This is the main goroutine blocking on `wg.Wait()` as it waits for all five `acquire` goroutines to finish. The last one (`gowrap4`) finishes at 15.01 seconds, so that is how long the main goroutine was waiting.

In summary, block profiles focus on how long each goroutine was stuck waiting. The mutex profile, in contrast, focuses on the perspective of the lock holder—how much contention its lock caused. The mutex profiler also tries to estimate the effect of multiple waiters and average the contention impact, but block profiling is always one-to-one: one goroutine, one amount of time spent waiting.

## Sampling Rate

The sampling rate for block profiling works differently from mutex profiling. If you look closely, mutex profiling uses the wording "Sampling fraction," while block profiling uses "Sampling rate."

The function `SetMutexProfileFraction` is named this way because it does event-based sampling. Every mutex contention event has a 1-in-N chance of being recorded, so sampling happens by fraction of events.

In contrast, block profiling uses `SetBlockProfileRate`, which is time-based and works on a threshold:

```
func blockevent(cycles int64, skip int) {
    if cycles <= 0 {
        cycles = 1
    }

    rate := int64(atomic.Load64(&blockprofilerate))
    if blocksampled(cycles, rate) {
```

```

        saveblockevent(cycles, rate, skip+1,
blockProfile)
    }
}

func blocksampled(cycles, rate int64) bool {
    if rate <= 0 || (rate > cycles && cheaprnd64()%rate
> cycles) {
        return false
    }
    return true
}

```

For example, let's say `blockprofilerate` is set to 1000 cycles. If a blocking event lasts longer than 1000 cycles, it will always be recorded. If it lasts less than 1000 cycles, it still has a chance of being recorded, depending on how big `cycles` is compared to `rate`. The chance is about `cycles / rate`:

- For a 100-cycle block, there is a 10% chance it will be recorded.
- For a 500-cycle block, the chance is about 50%.

This is controlled by this line:

```
cheaprnd64()%rate > cycles
```

If this condition is `true`, the sample is skipped. So this is a randomized filter that favors longer blocking events.

Why is this important? Because you don't want the profiler to record every tiny blocking event. That would just fill the profile with noise and unimportant details. But you also do not want to completely miss short blocks. If many of them happen, they could still be meaningful.

This sampling approach gives you a set of samples that is statistically representative, with a bias toward longer and more important blocking events.

## Goroutine Profiling

The Go goroutine profiler is a helpful tool built into the Go runtime. It lets you inspect the state of all active goroutines in your program. This inspection mostly means capturing the call stack of each goroutine, giving you a snapshot of what each one was doing at a specific moment.

There is a runtime call to enable goroutine profiling:

```
runtime.GoroutineProfile .
```

Figure 137. GoroutineProfile on runtime (src/runtime/mprof.go)

```
func GoroutineProfile(p []StackRecord) (n int, ok bool) {
    records := make([]profilerecord.StackRecord, len(p))
    n, ok = goroutineProfileInternal(records)
    if !ok {
        return
    }
    for i, mr := range records[0:n] {
        l := copy(p[i].Stack0[:], mr.Stack)
        clear(p[i].Stack0[1:l])
    }
    return
}
```

You usually should not call this directly. Instead, use the `runtime/pprof` package or the `net/http/pprof` package.

The reason is that the goroutine profiler needs a pre-allocated buffer (`p` `[]StackRecord`) to store the stack traces. If your buffer is smaller than the current number of goroutines, the function just returns the number of goroutines and a `false` signal to indicate it did not write the traces.

You might think you can first get the count and then allocate your buffer. However, there is always a timing gap between getting the number and calling the profiler, so the number of goroutines may have changed.

The `net/http/pprof` package does not handle this buffer sizing logic. Instead, it calls `runtime/pprof`'s ``Lookup("goroutine").WriteTo`` method.

Here is how you can capture the goroutine profile with the `Lookup("goroutine")` method:

```
var mtx sync.Mutex
var wg sync.WaitGroup

func acquire(n time.Duration) {
    mtx.Lock()
    defer mtx.Unlock()

    time.Sleep(n)
    wg.Done()
}
```

```

func main() {
    wg.Add(5)
    go acquire(1*time.Second)
    go acquire(2*time.Second)
    go acquire(3*time.Second)
    go acquire(4*time.Second)
    go acquire(5*time.Second)

    // Write goroutine profile to file
    f, err := os.Create("goroutine.prof")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    if err := pprof.Lookup("goroutine").WriteTo(f, 0);
err != nil {
        log.Fatal(err)
    }

    wg.Wait()
}

```

In this example, we create 5 goroutines, each trying to acquire the same mutex one after another. When the goroutine profile is captured (right after starting the goroutines, before waiting for them to finish), the first goroutine has probably acquired the lock and is sleeping, while the other four are blocked in `sync.Mutex.Lock`, waiting for the mutex.

These blocked goroutines are all waiting using the same mechanism. They are stuck in `runtime.gopark`, which is the low-level function where goroutines "park" themselves when they have to wait:

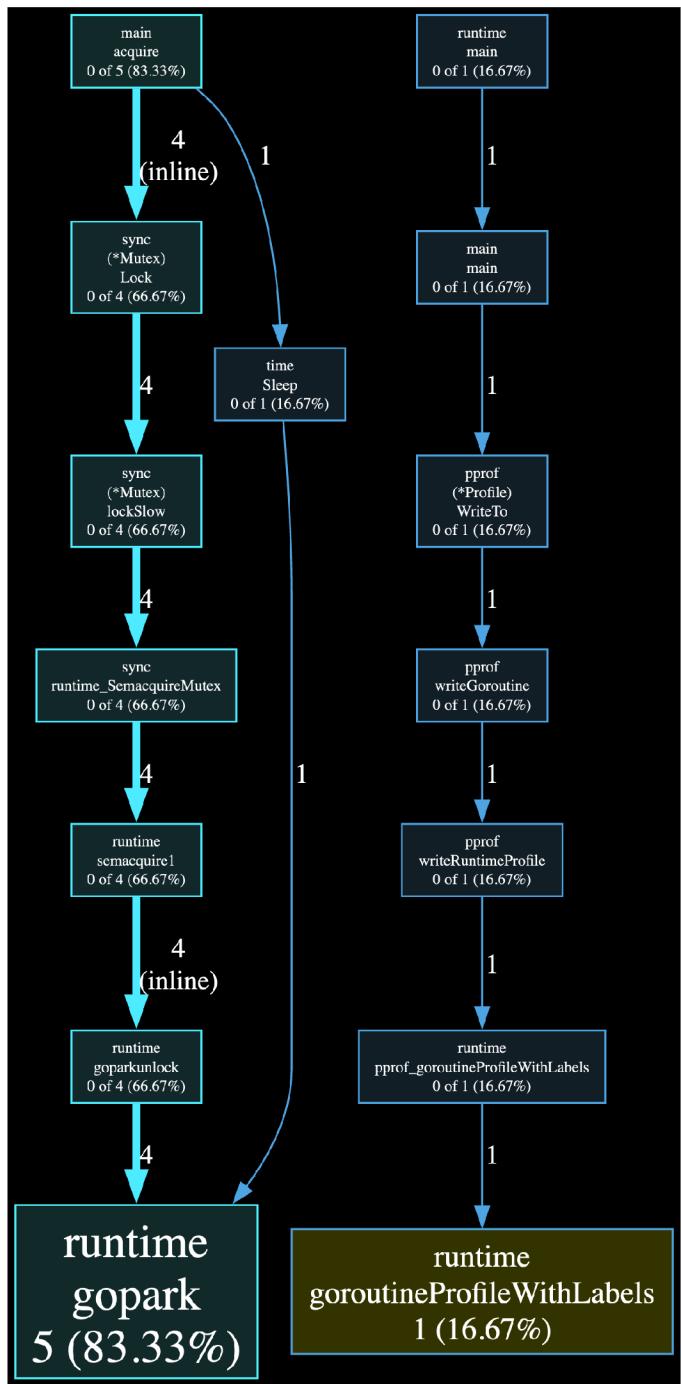


Illustration 228. Four goroutines blocked, one is sleeping

The sixth goroutine is the `main` goroutine. At the moment the profile is captured, it is running the call to `pprof.Lookup("goroutine").WriteTo`. Its stack trace appears in the profile under the `pprof` and

`runtime.goroutineProfileWithLabels` functions, showing the profiler itself in action.

Now, how does `pprof.Lookup("goroutine").WriteTo` actually work behind the scenes?

When you pass an empty stack trace slice to the `runtime.GoroutineProfile` function (which, as we said, you should not call directly), it just returns the number of goroutines and does nothing else. This lets callers find out how much space is needed for the stack traces before allocating a buffer. This is exactly how the `runtime/pprof` package works.

The `runtime/pprof` package first calls `runtime.goroutineProfileWithLabels` to get the number of goroutines. Then, it allocates a buffer a bit larger than needed (`n+10`) to make room for any new goroutines that might appear before the actual profile is collected.

Figure 138. `writeRuntimeProfile` in `runtime/pprof`  
(`src/runtime/pprof/pprof.go`)

```
func writeRuntimeProfile(w io.Writer, debug int, name
string, fetch func([]profilerecord.StackRecord,
[]unsafe.Pointer) (int, bool)) error {
    var p []profilerecord.StackRecord
    var labels []unsafe.Pointer
    n, ok := fetch(nil, nil)

    for {
        p = make([]profilerecord.StackRecord, n+10)
        labels = make([]unsafe.Pointer, n+10)
        n, ok = fetch(p, labels)
        if ok {
            p = p[0:n]
            break
        }
        // Profile grew; try again.
    }

    return printCountProfile(w, debug, name,
    &runtimeProfile{p, labels})
}
```

If the profile still does not fit (if `fetch` returns `ok=false`), it repeats the process with a bigger buffer. This loop continues until the profile is captured fully. This

retry is rare, but if it happens, it can be expensive because collecting the goroutine profile requires a stop-the-world pause.

Here is how the profiler works step by step:

```
func goroutineProfileWithLabelsConcurrent(p
    []profilerrecord.StackRecord, labels []unsafe.Pointer) (n
    int, ok bool) {
    if len(p) == 0 {
        return int(gcount()), false
    }

    ...
}
```

First, if the caller passes `nil` or an empty slice, the profiler simply returns an approximate count of goroutines, without stopping the world. This call to `gcount()` happens while the program is running, so the number may not be exact.

In the next phase, the profiler needs to get an exact count and all the stack traces. To do this, it stops the world to get a consistent view:

```
func goroutineProfileWithLabelsConcurrent(p
    []profilerrecord.StackRecord, labels []unsafe.Pointer) (n
    int, ok bool) {
    ...

    stw := stopTheWorld(stwGoroutineProfile)
    n = int(gcount())
    if n > len(p) {
        startTheWorld(stw)
        return n, false
    }

    ...
}
```

This count is now reliable because no goroutines can start or finish while the world is paused. The profiler checks if the buffer size is big enough. If not, it restarts the world and aborts profiling, returning the count so the caller can allocate a larger buffer. This is why the retry in the `runtime/pprof` package is considered expensive.

If the buffer is large enough, the profiler sets up for collection:

```

func goroutineProfileWithLabelsConcurrent(p
[]profilerecord.StackRecord, labels []unsafe.Pointer) (n
int, ok bool) {
    ...

    ourg.goroutineProfiled.Store(goroutineProfileSatisfied)
        goroutineProfile.offset.Store(1)
        goroutineProfile.active = true
        goroutineProfile.records = p
        goroutineProfile.labels = labels
    ...

    startTheWorld(stw)
}

```

After setup, the profiler "starts the world" again, and collection begins. It scans all goroutines and captures their stack traces. Note that before collecting, the world is restarted, so the program is running at the same time as the profiling:

```

func goroutineProfileWithLabelsConcurrent(p
[]profilerecord.StackRecord, labels []unsafe.Pointer) (n
int, ok bool) {
    ...
        // Collect stack traces
        forEachGRace(func(gp1 *g) {
            tryRecordGoroutineProfile(gp1, pcbuf,
Gosched)
        })

        // Cleanup
        stw = stopTheWorld(stwGoroutineProfileCleanup)
        endOffset := goroutineProfile.offset.Swap(0)
        goroutineProfile.active = false
        goroutineProfile.records = nil
        goroutineProfile.labels = nil
        startTheWorld(stw)
    ...
}

```

Finally, the world is stopped one more time for cleanup. By this point, the goroutine profile has been collected into the `pcbuf` buffer.

The collection step happens after the world is started. This is good for performance, since gathering stack traces for all goroutines can be costly, especially in programs with many goroutines.

However, this introduces a chance for the number of goroutines to change between the initial counting and the actual collection. The author of this code was aware of this race condition—so the profiler just proceeds with whatever data is actually collected, even if the number of stack traces doesn't exactly match the initial count `n`. The main reason for the first count is to make sure the buffer is big enough.

One side effect: any new goroutines created during the collection phase will not be included in the profile:

```
func newproc1(fn *funcval, callergp *g, callerpc uintptr,
parked bool, waitreason waitReason) *g {
    if goroutineProfile.active {
        // A concurrent goroutine profile is
running. It should include
        // exactly the set of goroutines that were
alive when the goroutine
        // profiler first stopped the world. That
does not include newg, so
        // mark it as not needing a profile before
transitioning it from
        // _Gdead.

newg.goroutineProfiled.Store(goroutineProfileSatisfied)
    }
}
```

The Go runtime marks any new goroutine as already captured (even though it is not) to make sure these are not included in the profile. Only the goroutines that existed at the start of profiling are captured.

To sum up, this design aims to reduce the impact on the running program while still providing accurate and useful profiling data. The stop-the-world phases are as short as possible, and the main profiling work is done while the program keeps running.

## Threadcreate Profiling

The thread creation profile tracks when and where OS threads are created in a Go program. This is different from goroutine profiling, it only deals with actual operating system threads.

In Go's runtime, the `m` structure represents an OS thread. Each thread has a field called `createtestack` which stores the stack trace at the point where the thread

was created by `runtime.newm`:

Figure 139. `createstack` on `m` (`src/runtime/runtime2.go`)

```
type m struct {
    ...
    createstack [32]uintptr // stack that created this
    thread
}
```

This array holds up to 32 pointers that represent the call stack when the thread was created. This lets the profiler report where in the code each thread started.

This data is always available inside the Go runtime and does not have any sampling rate or filtering. When you request the thread creation profile, the profiler simply walks through the list of all threads (the `allm` list) and collects the `m.createstack` for each one.

However, `threadcreate` profiling is rarely useful for most developers. The problem is that most threads in Go are created by the scheduler or the system monitor thread. When threads are created this way, the stack trace comes from the system or scheduler code, not your application code—even if your app indirectly triggered the need for a new thread.

The problem with the `threadcreate` profile is well known and is even documented in the Go code. There is a long-standing issue (#6104) about it: `runtime: threadcreate profile is broken` [[threadcreate](#)]. This issue points out two possible solutions:

- Remove the `threadcreate` profile completely, since it is not very useful,
- Or capture the stack trace from the blocking cgo or syscall that actually caused the thread to be created.

This issue has been open since 2013, and the "release-none" label means it was never given priority to fix.

## Delta (Diff) Profiling

Delta profiling is a technique where you compare two snapshots of a profile and focus only on the difference between them. In Go, you do this by collecting a "before" profile, letting your workload run, then collecting an "after" profile. You can then use the `pprof` tool to subtract the first profile from the second.

This method is very useful for understanding your application's behavior. For example: What happens if we increase the workload—where does the bottleneck move? How does the performance differ between the `master` (or `main`) branch and your branch? Or if you change an algorithm, how much performance improvement do you get?

The typical workflow looks like this:

```
go test -run=Warmup -cpuprofile=base.out
go test -run=Benchmark -cpuprofile=after.out
go tool pprof -base base.out after.out
```

When you take two snapshots, one at time A (earlier) and one at time B (later), `pprof` can compare them. For every unique stack trace (the "sample key") that appears in either snapshot, `pprof` puts the two sample values side by side. For example, a stack might show:

- A: 50 KB live on the heap
- B: 120 KB live on the heap

The diff is  $120 \text{ KB} - 50 \text{ KB} = +70 \text{ KB}$ . These differences make up a new profile:

- Positive numbers mean the metric grew from A to B (such as new allocations, more CPU time, or longer blocking).
- Negative numbers mean it shrank (such as memory being freed, code running less, or contention clearing up).
- Anything that is the same in both snapshots becomes zero and disappears from the diff view.

What you see is only what changed between the two times. Because the result is still a normal `pprof` file, you can use all the usual commands (`top`, `web`, `list`, and so on) to explore just the deltas.

Let's look at an example. Here is a diff between two heap profiles of VictoriaLogs:

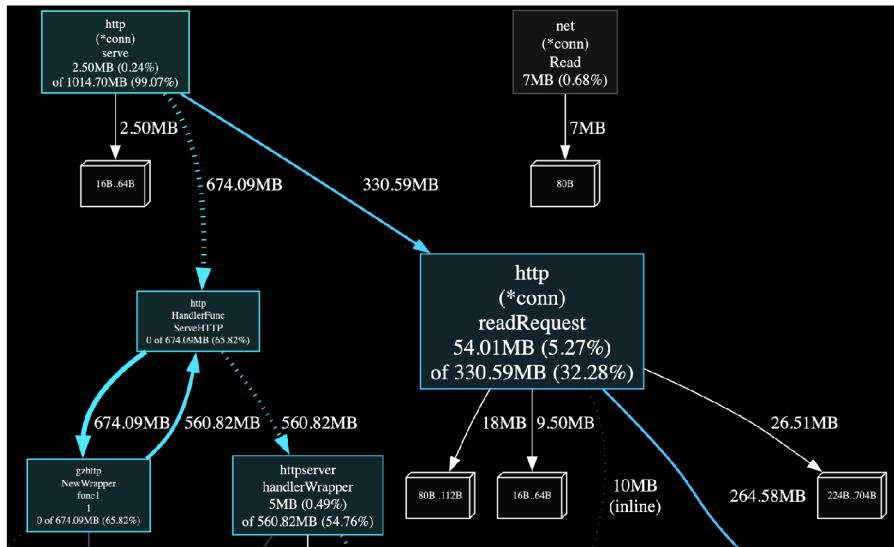


Illustration 229. Heap profile diff: memory growth in HTTP handlers

The diagram is a "call-graph diff" that `pprof` created by subtracting the first heap profile (time A) from the second (time B).

Everything in red shows memory that grew between the two snapshots. Grey boxes are areas where the change was so small it is shown as `~0`. Thick, solid red arrows show the call paths that contributed the most to memory growth. Dotted red arrows mean `pprof` trimmed out some intermediate frames, but the path still mainly represents positive growth.

Let's read the diagram:

- `http.(*conn).serve` is the root, and now holds about 1.01 GB more live memory than before.
- Most of that increase (about 674 MB) flows through `http.HandlerFunc.ServeHTTP` into the middleware's `gzhttp.NewWrapper`, suggesting the wrapper is holding on to large buffers or objects that were not present in the base profile.
- Another branch shows an extra 330 MB building up while `readRequest` parses incoming HTTP requests, with smaller extra allocations of 18 MB and 9.5 MB for helper buffers.

So the diff shows that almost all the new memory came from the request-handling code, especially the gzip wrapper chain. Investigating how `gzhttp.NewWrapper` allocates memory and when it releases buffers is likely the next step to find the root cause of the memory growth.

Now, when you capture profiles while scaling up the workload, you usually see memory growth but not much shrinkage. However, you can reverse the order of the profiles to see what changes when the workload is decreased:

```
go tool pprof -base after.out base.out
```

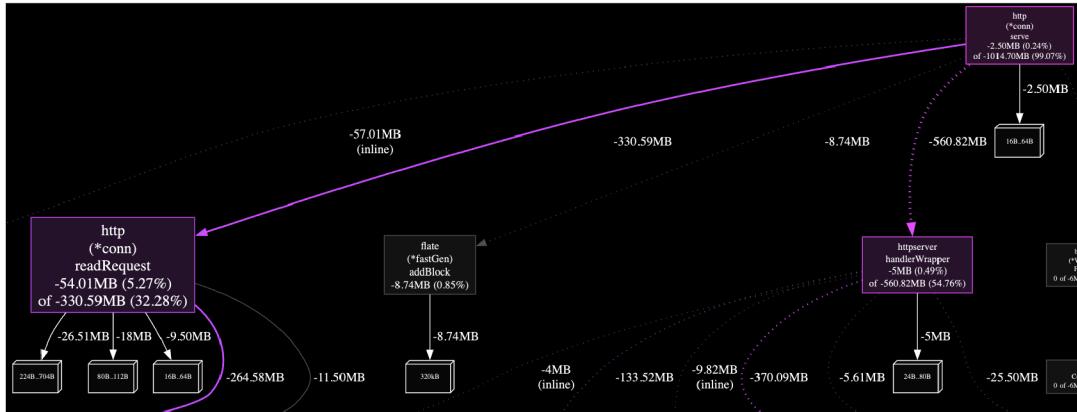


Illustration 230. Green arrows indicate significant reductions in memory

This is the same view, but all sample values that were positive (showing growth) now become negative, and all values that were negative (showing shrinkage) become positive. The user interface changes the colors based on the sign, so areas that were red now appear green. This signals that these areas represent memory that was released or disappeared between the later snapshot and the earlier one.

Go offers a shortcut for taking delta profiles through its HTTP debug endpoints. If you add a `seconds=N` query parameter to any snapshot-style profile endpoint—such as `/debug/pprof/heap`, `/allocs`, `/block`, `/goroutine`, `/mutex`, or `/threadcreate`—the runtime will automatically collect a before-and-after pair and return only their difference.

The handler first records an initial profile, called `p0`. It then sleeps for `N` seconds, letting the program run normally. After the wait it captures a second profile, `p1`, of the same type. Both snapshots hold cumulative counters up to their respective moments. To isolate activity that occurred only during the waiting period, the handler subtracts every counter in `p0` from the matching counter in `p1`, stack trace by stack trace. Any stack with no new activity subtracts to zero and is dropped. The handler finally sends this difference profile back to your browser or tool.

## 6. Trace: Timeline & Internals

Built-in profilers give you lots of runtime data. Still, these tools mostly show either a summary or a static snapshot. This can make it hard for a developer to really see how a Go program changes and moves *over time*. For example, a profiler might say:

90% of allocations happen at line 35 in server.go

This tells you where the memory pressure starts, but it does not reveal which goroutine performed the allocation, when it happened, or what else was occurring in the program at the same time. What Go lacked was a tool for a dynamic and detailed view of execution, including the precise timing of events as they occur.

That is what the Go execution tracer is designed for. The tracer provides a timeline of events as your program runs, helping you answer in-depth questions about goroutine behavior, such as:

- When did this goroutine actually run?
- How long did it run before it became blocked?
- What caused it to block?
- Which other goroutine or event unblocked it?
- How did garbage collection affect the execution flow of this goroutine?

With the execution tracer, you can follow the full lifecycle of a goroutine. For example, imagine your web server starts a new goroutine for every request:

```
go handleConnection(conn)
```

With tracing, you can see when `handleConnection` started, how long it spent waiting on I/O, how often the scheduler stopped it, and if it was ever blocked on a mutex or a channel. This level of detail is extremely helpful for understanding concurrency, finding bottlenecks, and debugging deadlocks.

Tracing is built into every Go binary and can be turned on during testing, at runtime using an HTTP endpoint, or directly in your code. When tracing is off, it has almost no effect on performance.

The main idea is simple. When you turn on tracing, the Go runtime starts to record a series of events as your program runs. These events show key runtime activities, and each one is timestamped very precisely, down to nanoseconds.

Each event also records details like what type of event it is, the OS thread ID, the logical processor ID, the goroutine ID, a stack trace, and other fields based on the context. In Go, the word "processor" means a logical unit set by `GOMAXPROCS`. This is the maximum number of goroutines that can run at the same time, not the number of physical CPU cores.

The tracer records different types of events, such as:

- Scheduling events, which show when goroutines are started, blocked, unblocked, or ended,
- Network activity, which logs when goroutines block or resume on network I/O,
- System calls, which record when goroutines enter or leave a syscall,
- Garbage collection, which tracks when GC starts and ends, including steps like sweeping,
- User-defined events and logs, which you can add to the trace for custom tracking.

After you collect a trace with the `runtime` package, you can look at it with the built-in `trace` tool. This tool opens up a browser-based viewer that shows a timeline for all goroutines, logical processors, and system threads. By exploring this timeline, you can see the life of each goroutine, spot bottlenecks, and find the causes of slowdowns.

## Turning On Tracing (go test, HTTP, code)

You can start tracing using the `go test` tool with the `-trace` flag. This way is best when you want to analyze performance in a controlled test run.

For example, you can run a specific benchmark and record a trace with this command:

```
go test -trace=trace.out -bench=. -benchtime=10s -cpu=16
```

This saves the trace to a file called `trace.out`. Later, you can analyze it using `go tool trace`.

For applications that are already running, such as servers, you can trigger tracing through an HTTP handler at `/debug/pprof/trace`. You pass in a `seconds` parameter to control how long the trace should last. This uses the same runtime

tracing APIs and lets you trace remotely without changing or rebuilding your code.

```
curl http://localhost:6060/debug/pprof/trace?seconds=5 >
trace.out
```

The most flexible way to use tracing is through the programmatic API in the `runtime/trace` package:

```
func main() {
    f, err := os.Create("trace.out")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    if err := trace.Start(f); err != nil {
        log.Fatal(err)
    }
    defer trace.Stop()

    // Your logic here
    ...
}
```

Tracing by default gives you lots of information about memory, such as when garbage collection runs, how the heap size changes, and more. If you want even more detail about memory, you can set the environment variable

`GODEBUG=traceallocfree=1`. When you do this, the trace includes three extra groups of events: heap span events, heap object events, and goroutine stack events.

- **Heap span events:** Memory spans are how the runtime manages memory pages. Each span usually covers several pages, often 8KB each. These events show when the runtime allocates or frees memory regions. You also see all current spans at the start of tracing.
- **Heap object events:** These show every time your program allocates or frees a heap object. You see when an object is allocated (`malloc`), freed (`free`), and all heap objects at the start of tracing.
- **Goroutine stack events:** These events show stack growth for goroutines. You see when a stack grows or is freed.

You should not use this detailed tracing for normal profiling, because it slows down your program a lot. Regular traces and heap profiles are enough for most

performance work and have much lower cost. This detailed memory trace feature is still experimental. It creates a huge amount of data and can slow your program down a lot. But it lets you see exactly how Go manages memory in ways that were never possible before.

## Visualizing Traces

### Proc-Oriented View (per-P lanes & flows)

The "proc view" is one of the main ways to visualize data in Go's trace tool. This view shows a timeline of events, sorted by logical processors (P), so you can see exactly what was happening in your Go program during the trace period.

When you open the trace viewer and use the proc view, you will see a complex layout with several sections stacked vertically.

At the very top is the **STATS** section, which holds three important time series graphs:

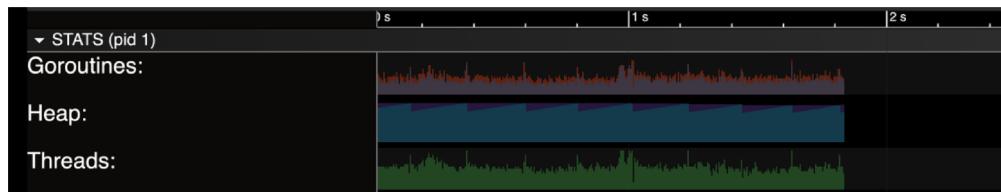


Illustration 231. Goroutines, heap, and thread statistics

Here is what each graph shows:

- The "Goroutines" graph displays the number of active goroutines over time. If you click any point on the timeline, you will get a breakdown of the goroutine states at that moment. You can see how many were running, how many were runnable (ready to be scheduled), and how many were waiting (blocked). This graph helps you understand your app's concurrency and spot possible scheduling problems.
- The "Heap" graph shows memory usage. The orange line is total heap allocation in bytes. The green line is the GC threshold, which marks when the next garbage collection cycle will start. This view helps you see memory patterns and how garbage collection affects your program.
- The "Threads" graph shows the number of OS threads Go sees as active (running or in a system call). If you move your mouse along the timeline,

you will notice that the count goes up when threads start working (when their Ps are running) and goes down as soon as they are parked.

Below the `STATS` section are several more timelines, each focused on key runtime events:



Illustration 232. GC, network, and syscall events timeline

These show garbage collection events and runtime events that cause goroutines to wake up:

- The "GC" timeline shows when garbage collection is happening and what phase it is in, such as mark or sweep. GC activity appears as colored blocks, and you can click them to see details about each GC event.
- The "Network" lane shows unblock events when goroutines are woken up because of network activity, such as when data arrives on a socket. These events have reasons like "network" and are handled by the NetpollP special resource.
- The "Timers" lane shows when goroutines are woken up by timers, like after `time.Sleep()` or `time.After()` finishes. These events are handled by the TimerP special resource.
- The "Syscalls" lane shows when goroutines come back from system calls that blocked. These events are handled by the SyscallP special resource.

If your app does not use any network operations, timer functions, or blocking syscalls during the trace, these special lanes will not appear.

## What are `TimerP` and `SyscallP`?

In the execution trace viewer, every vertical "lane" is labeled with a numeric P-ID. Real P-IDs are numbered from 0 up to GOMAXPROCS-1 and correspond to the actual processors that run Go code. Sometimes, however, the runtime needs to record an event that is not linked to any real P.

To keep the timeline organized, the runtime creates a few special, or "pseudo," processor IDs. These events get their own lanes:

- `TimerP` is a pseudo-P for timer-related wake-ups. When a sleeping goroutine's timer expires, the timer thread wakes it up. That wake-up is shown on the `TimerP` lane because no real processor was involved at that moment.
- `SyscallP` is a pseudo-P for wake-ups that occur when a goroutine returns from a blocking system call. When a goroutine enters a long syscall, its P is returned to the scheduler. When the syscall finishes, the goroutine becomes runnable again, but at that moment, it does not yet have a P.

The main part of the trace viewer shows timelines labeled "Proc 0", "Proc 1", and so on. Each one represents a logical processor (based on your `GOMAXPROCS` setting). Each processor can run only one goroutine at a time.



Illustration 233. Goroutine execution spans across multiple processors

Let's break down how to read these timelines:

- Colored horizontal bars show when goroutines are running. Each goroutine has its own color and label, such as "G1" or "G2". The label usually shows the main function the goroutine is running—for example, "G1 runtime.main" means goroutine 1 is running the `runtime.main` function.
- When a goroutine moves to a different processor, its colored bar continues but appears in a new processor lane. This happens when goroutines are parked or preempted by the scheduler.
- Under each execution bar, you may see small markers or vertical lines that indicate events in that goroutine's lifecycle, such as:
  - When the goroutine was created
  - When a function call or return occurred
  - Synchronization events (mutex locks and unlocks, channel operations)

- System calls
- Garbage collection activity

Each colored bar in a processor row means that goroutine was running on that processor at that time. The color for each goroutine stays the same, even when it moves between processors, so you can easily follow its path.

You can click on any execution bar to see a popup with more details, including:

- How long the span lasted
- The stack trace when the goroutine stopped or yielded
- What made the goroutine stop running, like a channel operation or locking a mutex

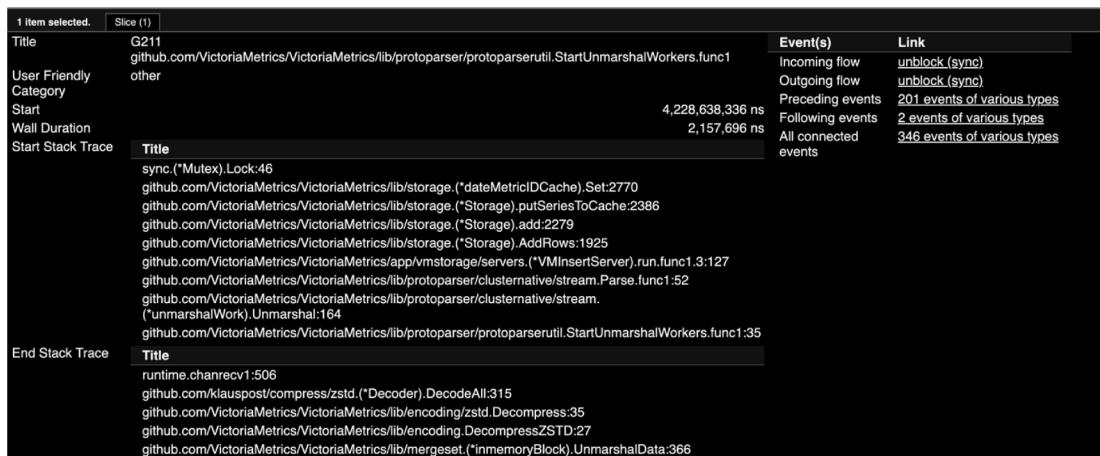


Illustration 234. Execution slice reveals goroutine duration and context

When you click on a slice, you get a detail panel. This panel shows complete info about that goroutine's run period. For example, in the image, goroutine `G211` is running the function `StartUnmarshalWorkers.func1` from the VictoriaMetrics code.

The span for this goroutine starts at "4,228,638,336 ns" (about 4.23 seconds) after the trace begins. The "Wall Duration" value of 2,157,696 ns means this execution span lasted about 2.16 milliseconds.

When you look at the stack trace, you will see two parts:

- "Start Stack Trace": This shows which functions were being called when the goroutine started running. The most recent function call is listed at the top.

- "End Stack Trace": This shows the functions being called when the goroutine stopped running, again with the most recent call at the top.

In this example, we can see that goroutine `G211` started its execution by locking a mutex with `sync.(*Mutex).Lock()`. It stopped when it was blocked while waiting to receive from a channel, shown by `runtime.chanrecv`.

Now, look at the event on the right side. The "unblock (sync)" incoming flow means this `G211` goroutine was blocked on a synchronization primitive, and a different goroutine did something that unblocked it.

If we hover over the "unblock (sync)" link in the properties panel, the trace viewer highlights a flow arrow between the two goroutines.

This makes it much easier to see which operations are directly related:

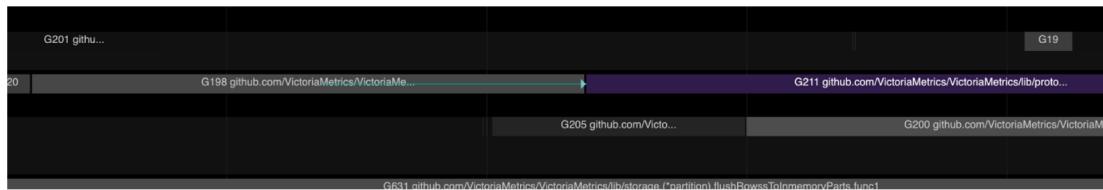


Illustration 235. Flow (red) arrow connects blocking and unblocking goroutines

We can also click on the "unblock (sync)" link. This opens more details about the flow event, such as the event ID, how long the flow arrow lasted, and more.

The "outgoing flow" section will show which other goroutines were unblocked by the current goroutine.

There is also a "Related Events" section. This area shows events that are causally linked in the Go execution trace. When you pick a goroutine execution span in the viewer, it figures out which other events are related to that span. These related events are grouped in three ways:

- Preceding Events: These are events that happened before your chosen goroutine started running. They came earlier and might have caused your goroutine to start.



Illustration 236. Visualizing causality: curved arrows show goroutine flow events

- Following Events: These are events that happened after your goroutine finished. Often these are the next goroutine that started running or any goroutines that were unblocked by yours.
- All Connected Events: This group gives you the full "conversation" your goroutine was involved in. It includes both the preceding and following events, plus any others that belong to the same chain of causes and effects.

There is another view in the trace viewer that is a bit hidden, called the "trace breakdown". To see it, look to the far right of the trace viewer and click the "File size stats" button.

Group by: <input checked="" type="checkbox"/> Event Type ►   ◀ <input checked="" type="checkbox"/> Title   <input type="checkbox"/> Category	
Title ▾	Num Events ▾
▼ counters	357250
Goroutines	273413
Heap	70465
Threads	13372
▼ flows	164124
unblock (chan receive)	162450
unblock (sync)	940
unblock (system goroutine wait)	454
unblock (select)	164
unblock (preempted)	42
unblock (sleep)	16
go	14
unblock (GC background sweeper wait)	12
unblock (network)	12
exit blocked syscall	10
unblock	6
unblock (chan send)	4
► begin_end (compact)	93748
▼ instant	13383
proc start	6690
proc stop	6682
unblock (network)	6
exit blocked syscall	5
▼ metadata	44
thread_name	20
thread_sort_index	20
process_name	2
process_sort_index	2

Illustration 237. Trace event summary: see where time and space go

This feature analyzes your trace and gives you a statistical breakdown by event type. You will see counts and categories for each event in the trace:

1. **counter** (357,250 events): These events track numeric values that change over time and are used to make time-series graphs.
  - Goroutines (273,413 events): Each event records how many goroutines are in each state (running, runnable, or waiting) at a certain moment. A high count means a lot of goroutine state changes.
  - Heap (70,465 events): Each event shows a change in heap memory allocation or in the garbage collection target.
  - Threads (13,372 events): These show how many times the thread state counter was updated during the trace. You will see lots of these if threads start, stop, or enter/exit syscalls often.
2. **flows** (164,124 events): These show causal relationships, like when one goroutine unblocks another. These are what create the arrows you see in the trace view.

- `unblock (chan receive)` (162,450 events): Shows when a goroutine is unblocked by getting data from a channel.
  - `unblock (sync)` (940 events): Shows when a goroutine is unblocked by a sync operation, such as a mutex or WaitGroup.
  - `unblock (system goroutine wait)` (454 events) and other smaller types: These track other causes that unblock goroutines, each representing a different way for a goroutine to resume work.
3. `begin_end` (93,748 events): These mark the start and end of an activity, so you can see how long something took.
  4. `instant` (13,383 events): These are single point-in-time events with no duration. They show up as dots in the trace.
    - `proc start` (6,690 events): Shows when a processor (P) started to do work.
    - `proc stop` (6,682 events): Shows when a processor (P) stopped doing work.
    - There are also a few other instant event types with lower counts.

These stats are useful because they show exactly what your trace data is made of. For example, if you see a lot of channel operation events (like the 162,450 unblock events from channel receives), it means your application relies heavily on Go's channels for communication.

## **Thread-Oriented View (per-thread lanes)**

OS threads are the real execution units that the operating system manages. The thread-oriented view in the trace viewer lets you see how these OS threads are used during your program's run.

If you are not familiar with the MPG model, or if you need a refresher, please go back to the start of this chapter.

When you open the trace viewer at `http://[::]:<port>/trace?view=thread`, each row now stands for an OS thread instead of a logical processor.

Threads are labeled by their thread ID, such as "Thread 1" or "Thread 2". These are the actual thread IDs from the operating system, not logical processors or goroutines, and not any internal runtime thread ID.



Illustration 238. Visualizing Go program execution by OS thread

Usually, the number of threads is about the same as, or just above, the number of logical processors you have set with `GOMAXPROCS`. If your program creates too many threads, your performance will get much worse.

This view works almost the same as the "proc view". All the explanations and details about the "proc view" also apply to the thread view.

## Goroutine Analysis Tools (summary, breakdown, ranges)

The goroutine analysis view in Go's trace viewer is available at [/goroutines](#) (e.g., [http://\[::\]:8086/goroutines](http://[::]:8086/goroutines)). This view shows a breakdown of goroutine behavior during your program's execution. It is a very useful tool, though often overlooked, for spotting performance bottlenecks and understanding patterns in your Go programs.

You will see a table that groups goroutines by their start location, which is usually the function name where the goroutine was created. Each row shows:

1. **Start location:** The function where the goroutine started, which acts as the group's identifier
2. **Count:** The total number of goroutines in this group
3. **Total execution time:** The total time all goroutines in this group spent running code

## Goroutines

Below is a table of all goroutines in the trace grouped by start location and sorted by the total execution time of the group.

Click a start location to view more details about that group.

Start location	Count	Total execution time
github.com/VictoriaMetrics/VictoriaMetrics/lib/protoparser/protoparserutil.StartUnmarshalWorkers.func1	16	30.016175595s
runtime.gcBgMarkWorker	13	902.409355ms
github.com/VictoriaMetrics/VictoriaMetrics/lib/storage.(*partition).mustMergeInmemoryParts.func1	3	633.017544ms
github.com/VictoriaMetrics/VictoriaMetrics/lib/storage.(*partition).flushRowssToInmemoryParts.func1	8	598.981438ms
github.com/VictoriaMetrics/VictoriaMetrics/lib/storage.(*partition).mergePartsToFiles.func1	1	491.250053ms
github.com/VictoriaMetrics/VictoriaMetrics/app/vmstorage/servers.(*VMInsertServer).run.func1	1	346.967046ms
runtime.bgsweep	1	316.00135ms
github.com/klauspost/compress/zstd.(*Decoder).startStreamDecoder.func2	1	93.28186ms
github.com/klauspost/compress/zstd.(*Decoder).startStreamDecoder.func1	1	86.25728ms
github.com/VictoriaMetrics/VictoriaMetrics/lib/storage.(*partition).startPendingRowsFlusher.func1	1	57.354753ms
runtime.bgscavenge	1	55.284289ms
github.com/klauspost/compress/zstd.(*Decoder).startStreamDecoder	1	34.554867ms
runtime.(*traceAdvancerState).start.func1	1	29.93984ms

Illustration 239. Which goroutines are costing you the most time?

By default, the table is sorted by total execution time. The groups at the top are the ones that used the most CPU time. This helps you quickly see which types of goroutines are costing you the most.

If you click on a goroutine group, you will go to a detailed view at [/goroutine?name=\[function\\_name\]](#). Here, each goroutine in the group is shown in detail. This page has three main sections: summary, breakdown, and special ranges.

The summary section gives you high-level details about the group:

- The function where the goroutines were created,
- How many goroutines are in the group,
- What percentage of total program execution time is taken by this group.

Summary									
Goroutine start location:	<code>github.com/VictoriaMetrics/VictoriaMetrics/lib/protoparser/protoparserutil.StartUnmarshalWorkers.func1</code>								
Count:	16								
Execution Time:	89.14% of total program execution time								
Network wait profile:	<a href="#">graph (download)</a>								
Sync block profile:	<a href="#">graph (download)</a>								
Syscall profile:	<a href="#">graph (download)</a>								
Scheduler wait profile:	<a href="#">graph (download)</a>								

Illustration 240. Goroutine group summary: creation site, count, and execution share

The summary also includes links to related profiles, such as network wait, sync block, syscall, and scheduler wait. These are explained in the following sections.

The breakdown section is the main part of the goroutine analysis view. It is shown as a table with columns that use different colors for different states. These states show where a goroutine spent its time, such as executing code, blocked on synchronization, or waiting on the scheduler.

Goroutine	Total	Execution time	Block time ()	Block time (GC mark assist wait for work)	Block time (chan receive)	Block time (preempted)	Block time (sync)	Block time (syscall)	Sched wait time	Syscall execution time	Unknown time
199	5.000442305s	1.927895291s	0s	44.608μs	2.272274332s	53.952μs	10.848769ms	0s	789.044905ms	0s	280.448μs
204	5.000442305s	1.831512315s	0s	0s	2.339069026s	162.496μs	15.352705ms	0s	813.804963ms	0s	540.8μs
206	5.000442305s	1.859500936s	2.048μs	0s	2.318066351s	150.208μs	11.51693ms	0s	811.010376ms	0s	195.456μs
208	5.000442305s	1.905031948s	0s	0s	2.276217225s	153.984μs	14.642113ms	0s	803.566763ms	0s	830.272μs
213	5.000442305s	1.873888573s	0s	0s	2.288361252s	222.72μs	18.192961ms	0s	819.537567ms	0s	239.232μs
205	5.000442305s	1.884659148s	0s	0s	2.285262995s	253.184μs	15.756606ms	0s	813.636196ms	0s	874.176μs
212	5.000442305s	1.88742026s	0s	0s	2.28941636s	97.088μs	14.652994ms	0s	808.769523ms	0s	86.08μs
203	5.000442305s	1.872104s	0s	0s	2.310279713s	170.496μs	20.408ms	0s	797.380128ms	0s	99.968μs
202	5.000442305s	1.908462627s	0s	0s	2.278525444s	177.856μs	14.579841ms	0s	797.969625ms	0s	726.912μs
200	5.000442305s	1.857027982s	1.664μs	0s	2.317563084s	87.681μs	16.657279ms	0s	808.904807ms	0s	199.008μs
201	5.000442305s	1.835264914s	0s	0s	2.33090131s	185.024μs	14.681791ms	0s	818.546802ms	0s	862.464μs
209	5.000442305s	1.931479053s	0s	0s	2.253073076s	118.08μs	15.891201ms	0s	799.108582ms	0s	314.624μs
207	5.000442305s	1.900898353s	0s	0s	2.279276891s	175.168μs	12.426239ms	0s	807.481014ms	0s	184.64μs
210	5.000442305s	1.823826705s	0s	0s	2.32607299s	212.608μs	18.8304ms	0s	831.147794ms	0s	351.808μs
198	5.000442305s	1.888480537s	0s	0s	2.29075308s	163.264μs	13.097342ms	0s	807.496818ms	0s	451.264μs
211	5.000442305s	1.828722953s	0s	0s	2.318141776s	197.504μs	16.687167ms	0s	836.062633ms	0s	630.272μs

Illustration 241. Goroutine state breakdown: execution, blocking, and scheduler waits

Each row in the breakdown table stands for one goroutine, identified by its unique ID.

The columns in the breakdown table show different time categories. These are mutually exclusive, which means a goroutine can only be in one state at any one

time. The main time categories in the breakdown section are:

- **Execution time** (red): This is the useful time when the goroutine is actually running on a processor and doing its assigned work. The trace records every time a goroutine enters the `GoRunning` state and measures how long it stays active before blocking, yielding, or ending.
- **Block time by reason** (purple): This tracks how long the goroutine is blocked, and it breaks down the blocking time by reason. For example, it shows when a goroutine is blocked on a channel receive, channel send, select, mutex, or something else. If the reason is empty, it means the trace did not capture what caused that blocking period.
  - **Block time (syscall)**: This is the time a goroutine had to give up its processor (P) because a system call was blocking and waiting for something outside the program. Not all syscalls are blocking, but this counts only the time for blocking ones.
- **Scheduler wait time** (blue): This is the time a goroutine spends waiting for a processor because there are more goroutines ready to run than processors available. This helps you spot if your program has scheduler bottlenecks, where goroutines are ready but have to wait for CPU resources.
- **Syscall execution time** (dark purple): This is the time a goroutine is actually running system call code while still owning a processor. This is when the syscall is running but has not yet released the processor back to the scheduler.
- **Unknown time** (gray): This is time that does not fit into any other category. It is calculated as whatever time is left after adding up all other categories. If you see a lot of unknown time, it might mean there are gaps in the trace or some runtime activity that is not tracked by the trace system.

One of the most helpful visuals in the breakdown section is the stacked bar graph. This is a compact horizontal bar that shows how the time for each goroutine is split across the different states.

Each part of the bar is color-coded to match its category. The width of each segment shows the proportion of total lifetime the goroutine spent in that state. This makes it easy to see at a glance where the time went for each goroutine.

There is a small but useful feature: you can sort the breakdown table by clicking any column header. By default, it sorts by total time in descending order.

The last section is called "Special Ranges". This section shows periods where goroutines were part of specific runtime activities, mostly related to garbage

collection.

Only goroutines that spent time in at least one special range will appear in this table.

Goroutine	Total	GC incremental sweep	GC mark assist	stop-the-world (GC sweep termination)
199	5.000442305s	743.042µs	1.997696ms	567.808µs
204	5.000442305s	299.776µs	3.501952ms	625.28µs
206	5.000442305s	543.166µs	1.667904ms	593.536µs
208	5.000442305s	662.4µs	838.4µs	523.072µs
213	5.000442305s	637.438µs	1.7712ms	180.928µs
205	5.000442305s	492.798µs	1.351744ms	0s
212	5.000442305s	606.398µs	1.473152ms	201.152µs
203	5.000442305s	501.244µs	1.658112ms	455.936µs
202	5.000442305s	688.192µs	1.230784ms	275.584µs
200	5.000442305s	758.274µs	1.144ms	492.096µs
201	5.000442305s	868.286µs	1.310208ms	217.792µs
209	5.000442305s	945.024µs	1.098752ms	605.376µs
207	5.000442305s	824.575µs	1.345152ms	0s
210	5.000442305s	789.885µs	2.00096ms	0s
198	5.000442305s	472.766µs	2.815232ms	413.312µs
211	5.000442305s	578.046µs	1.788545ms	194.88µs

Illustration 242. Per-goroutine GC participation times (incremental sweep, mark assist, stop-the-world)

The "Special Ranges" section is different from the "Breakdown" section above it in an important way. The "Breakdown" section shows time categories that never overlap (like execution time or blocking time), but the "Special Ranges" section shows periods that can overlap with regular execution statistics.

This means the time shown in special ranges is not separate from execution time. Instead, it gives you another view of how a goroutine's time was used.

The Go runtime tracks these special periods with events called `RangeBegin`, `RangeActive`, and `RangeEnd`. These events show when a goroutine enters, is active in, or leaves a special runtime activity. The trace viewer uses these to add up the time each goroutine spent in these activities.

The most common special ranges you'll see include:

- **GC mark assist:** This is the time a goroutine spent helping the garbage collector mark objects. Sometimes, when garbage collection is running and needs extra help, the runtime makes regular goroutines help with marking. This can slow down their main work for a while.

- **GC incremental sweep:** This is the time spent cleaning up memory after the mark phase. Like mark assist, it can take time away from the goroutine's main task.
- **Stop-the-world phases** (several kinds): These are periods when the goroutine was stopped because of "stop-the-world" phases during garbage collection or other runtime events.
- **GC concurrent mark phase:** This shows time when the goroutine was running while the garbage collector was marking objects at the same time as regular execution.

Because these activities can overlap, a goroutine might be both running code (counted as "Execution time" in the Breakdown) and helping with garbage collection (counted as "GC mark assist" in Special Ranges) at the same moment.

## Profiles Generated From Traces (scheduler, sync, syscall, net)

The Go execution trace supports several special profile types that you can generate from trace data. These profiles are different from the usual `pprof` profiles because they are created by analyzing state changes and events inside the trace, not by runtime sampling.

There are 4 main profile types supported by the execution trace: block profiles, scheduler latency profiles, syscall profiles, and I/O profiles.

- **Scheduler latency profile:** This tracks the delay between when a goroutine becomes runnable (ready to run) and when it actually starts running on a processor (P). For example, if a goroutine is waiting for data from a channel, when the data arrives, it is unblocked and becomes runnable. But it still has to wait for the scheduler to pick it up and actually run it. The time between being unblocked and starting execution is called scheduler latency.
- **Synchronization blocking profile:** This shows the time goroutines spend waiting for synchronization reasons. It tracks time spent waiting on channels, mutexes, wait groups, and other sync primitives. The profile adds up all the time goroutines wait at each stack trace location. For example, if five goroutines each wait 100 ms on a mutex at the same code location, the profile will show 500 ms of contention at that spot.
- **Syscall profiles:** These profiles work the same way as the sync blocking profile, but focus on time spent waiting for system calls. The trace can tell the difference between blocking and non-blocking syscalls.

- **Network blocking profiles:** This profile is different from the block profile in standard pprof profiling. When a goroutine calls a blocking network operation, the runtime parks the goroutine and lets others run. The block profile itself ignores I/O waits and does not report anything about sockets. It only focuses on sync objects like channels, mutexes, `sync.Cond`, and select.

These trace-based profiles can be downloaded and used with `go tool pprof` like other profiles.

For example, let's look at a scheduler latency profile:

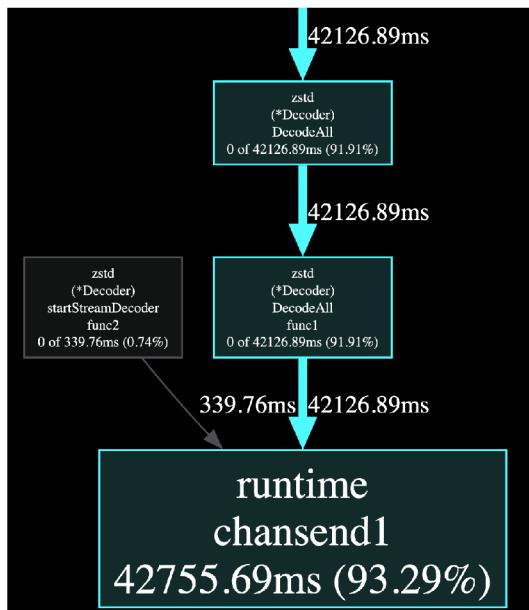


Illustration 243. Channel send is top scheduler latency hotspot

The `runtime.chansend1` function is used by the Go runtime when a channel send must block. For example, if your code does:

```
ch <- v // sending on an unbuffered or full channel
```

If there is no receiver ready, the sending goroutine is parked and cannot continue. When a receiver arrives, the runtime wakes up the sender and puts it on the run queue.

From that moment until a processor finally picks it up, the goroutine is "Runnable" but not running yet.

In this trace, all goroutines that were unblocked after waiting in `chansend1` spent a combined total of 42.7 seconds waiting in the run queue before they could run again. This was 93% of all runnable-queue time for the whole program.

It is important to note that this is not one goroutine stalled for 42 seconds. Instead, it is thousands or even millions of very short stalls (often microseconds or milliseconds each). The trace adds up all these tiny stalls because they all happened at the same channel send location.

This shows that the channel send at this code location is a major hotspot for contention. Many goroutines block here and then wake up together, causing a lot of time to be wasted waiting to run. If you can reduce this contention (for example, by making the channel buffer larger, speeding up the consumer, sharding, or batching), you can directly cut down that 42 seconds and make your program faster.

## Adding Semantics: Tasks, Regions, Logs

Tracing collects a huge amount of low-level runtime information, such as goroutine state changes, system calls, garbage collection, and scheduler activity, as we have seen in earlier sections. But there is still a big gap between what the runtime can see and what you, as a developer, actually care about.

For example, if you look at a typical trace of a web server, you might find thousands of goroutine creations, network I/O events, memory allocations, and scheduler preemptions. Yet it is almost impossible to answer simple questions like:

- Which goroutines handled the same HTTP request?
- How long did it take to process a specific user's order?
- Which database queries were part of the same transaction?
- Why was one request slower than another, even if they used the same code?

The trace might tell you that goroutine 147 blocked on a channel at a certain time, but it will not tell you that this was actually the payment validation step for customer 12345's order. The runtime knows what happened, but the meaning of each action is lost.

User annotations add a semantic layer that we, as developers, can use to connect our application logic to the runtime trace. This lets us add business meaning to the trace data. Instead of just seeing a list of runtime events, we get a clear story

about what our app is doing at the business level. We can add three types of semantic information: `task`, `region`, and `log`.

Tasks represent logical units of work that may include many goroutines and are meaningful in our application. For example, we can create a task for "process HTTP request" or "execute database transaction". By marking a task, we tell the tracer: "all activity within this context belongs to this logical operation".

The task system supports a hierarchy. Tasks can have parent and child relationships, letting us model complex operations with many steps:

```
ctx, task := trace.NewTask(parentCtx, "processRequest")
defer task.End()
```

When you create a new task, the tracer looks for a parent task in the context using a special key.

```
processRequest          (top-level)
└── readFromDB          (child #1)
    └── event: query=SELECT ...
└── renderResponse       (child #2)
    └── event: tmpl=home.html
```

If the context does not have a parent, the new task is a top-level task. This inheritance means that any work done inside the task context is automatically grouped under that task.

One of the most useful features of the task system is automatic latency measurement. The tracer tracks the time between when a task is created and when `End` is called. You get latency measurements for every type of task, with no extra work needed in your application code. This happens entirely inside the runtime.

Next, regions give you a more detailed way to annotate your code. They are designed to mark specific time intervals inside a single goroutine's execution.

Unlike tasks, regions must always start and end within the same goroutine. They must also follow strict nesting rules: a region started later must end before an earlier region can end. There are two ways to create regions, and each serves a different purpose with its own trade-offs:

```
trace.WithRegion(ctx, "databaseQuery", func() {
    result, err := db.Query("SELECT * FROM users")
    // ... process result
})
```

`WithRegion` is the simple option. It takes a function, starts the region, runs your function, and ends the region when your function returns. This cleanup is automatic and happens even if the function panics. Internally, it is similar to:

```
// What WithRegion does internally
id := fromContext(ctx).id
userRegion(id, regionStartCode, regionType)
defer userRegion(id, regionEndCode, regionType)

fn()
```

If you need more control, use `StartRegion`. This lets you decide exactly when the region starts and ends:

```
region := trace.StartRegion(ctx, "fileProcessing")
defer region.End()

// Your code here
file, err := os.Open("data.txt")
if err != nil {
    return err
}

// ... process file
```

While `WithRegion` is easy and safe, it can feel limited for more complex code. It also uses closures, which sometimes hold on to variables longer than needed and might affect garbage collection. `StartRegion` gives you more flexibility, but you have to be careful to always call `End` so the region is finished properly.

Regions are always linked to whatever task is active in the current context. If there is no active task, they are attached to the special background task (ID 0). This setup creates a natural hierarchy: tasks contain regions. You get both a high-level view (tasks) and a detailed, internal structure (regions).

The rule that later regions must end before earlier ones comes from how regions are managed in the runtime. The runtime uses a stack for each goroutine, so starting a region pushes it onto the stack, and ending a region pops the latest one off.

Logs are different from tasks and regions. Logs record a point-in-time event and give context during execution. They do not track duration, just moments or state information. There are two log functions: `Log` for simple messages, and `Logf` for formatted messages, much like `fmt.Sprintf`:

```
trace.Log(ctx, "cache", "cache miss for key: " + key)
trace.Logf(ctx, "performance", "query took %v milliseconds",
duration.Milliseconds())
```

Logs always include both a category and a message. The category is used for filtering and grouping by trace analysis tools. The system expects only a limited number of unique categories, which helps with trace performance and memory usage.

Up to this point, one of the most important aspects of the user annotation system is how it works with Go's `context` package. Every annotation API takes a `context.Context` parameter, and task information is carried through the context with a special context key. This means that the task context moves automatically through your application's call stack. Any annotation you make with a derived context is always connected to the correct task.

This integration with `context` also handles cross-goroutine cases very well. If you pass a context with task info into a new goroutine, that goroutine's annotations are linked to the same task. This is very useful for tracking work that happens across multiple goroutines, such as in fan-out patterns or producer-consumer designs.

User annotations in Go tracing are very different from microservices tracing tools like OpenTelemetry. Go tracing shows connections between your tasks and low-level runtime events that OpenTelemetry cannot see. OpenTelemetry is good for showing request flows between services, but Go tracing shows you what is happening inside the Go runtime.

For example, if a database call is slow, Go tracing can tell you if the delay came from a syscall block, slow scheduling, or network lag. OpenTelemetry would only report that the database call was slow, without showing why.

To see this all come together, let's build a realistic example using tasks, regions, and logs. We will start with the main function and build from there. First, we set up tracing by creating an output file and starting the tracer.

This will record both runtime events and our user annotations:

```
func main() {
    f, err := os.Create("trace.out")
    if err != nil {
        log.Fatalf("Failed to create trace file:
%v", err)
```

```

        }
        defer f.Close()

        if err := trace.Start(f); err != nil {
            log.Fatalf("Failed to start trace: %v", err)
        }
        defer trace.Stop()

        ctx := context.Background()
        ctx, rootTask := trace.NewTask(ctx, "rootTask")
        defer rootTask.End()
    }
}

```

Here we create a root task called `rootTask` that is the parent for everything else. This gives us a single top-level operation for the whole program run.

Let's simulate 10 concurrent requests:

```

func main() {
    ...
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(reqID int) {
            defer wg.Done()
            handleRequest(ctx, reqID)
        }(i)
        time.Sleep(time.Millisecond * 10)
    }

    wg.Wait()
    trace.Stop()
}

```

Each goroutine receives the same context, so they all inherit the root task. The short sleep between launches helps us see the requests staggered out in the trace. Next, we simulate a handler function:

```

func handleRequest(ctx context.Context, requestID int) {
    ctx, task := trace.NewTask(ctx, "handleRequest")
    defer task.End()

    trace.Log(ctx, "request", fmt.Sprintf("started
request %d", requestID))
}

```

Every request creates its own task, which becomes a child of the root task. The log records a message with a timestamp for when this request started.

The next step is validation, which we wrap in a region to measure exactly how long it takes:

```
func handleRequest(ctx context.Context, requestID int) {
    ...
    trace.WithRegion(ctx, "requestValidation", func() {
        time.Sleep(time.Millisecond *
time.Duration(rand.Intn(5)+1))
    })
}
```

This runs synchronously and will always appear first in each request's timeline.

Here's where it gets more interesting. We now fan out to three parallel database queries, but all of them use the same context:

```
func handleRequest(ctx context.Context, requestID int) {
    ...
    var wg sync.WaitGroup
    results := make(chan string, 3)

    for i := range 3 {
        wg.Add(1)
        go func(queryID int) {
            defer wg.Done()

            trace.WithRegion(ctx,
fmt.Sprintf("query-%d", queryID), func() {
                result := dbQuery(ctx,
requestID*10+queryID)
                results <- result
                trace.Log(ctx, "result",
result)
            })
        }(i)
    }
}
```

This setup means all three goroutines will show up in the trace as working on the same request task. Each one gets its own region with a name like "query-0", "query-1", or "query-2".

After the queries, we process the results in a new region:

```
func handleRequest(ctx context.Context, requestID int) {
    ...
    trace.WithRegion(ctx, "processResults", func() {
        for result := range results {
```

```
        time.Sleep(time.Millisecond *  
time.Duration(rand.Intn(5)+1))  
        trace.Logf(ctx, "processing",  
"processed %s", result)  
    }  
}  
}
```

The logs here show exactly which results are being processed and at what time. Now let's look at the database function:

```
func dbQuery(ctx context.Context, id int) string {  
    defer trace.StartRegion(ctx, "dbQuery").End()  
  
    trace.Log(ctx, "query", fmt.Sprintf("id=%d", id))  
    trace.WithRegion(ctx, "dbConnect", func() {  
        time.Sleep(time.Millisecond *  
time.Duration(rand.Intn(300)))  
    })  
  
    time.Sleep(time.Millisecond *  
time.Duration(rand.Intn(200)))  
    return fmt.Sprintf("result-%d", id)  
}
```

The whole database call is wrapped in a `"dbQuery"` region using a `defer` to ensure it always ends. Right away, we log the ID we are querying. Inside, there is a nested `"dbConnect"` region for connecting, followed by extra time for the actual query work.

This setup gives you a detailed view of all the database timing. When you run this and open the trace with

```
go tool trace trace.out
```

you will see a clear hierarchy in the viewer. You can drill down from the root task, to individual requests, to each database operation. Logs provide helpful context at every level.

To see the full hierarchy, open `go tool trace trace.out` and go to the "User-defined tasks and regions" section:

## User-defined tasks and regions

The trace API allows a target program to annotate a [region](#) of code within a goroutine, such as a key function, so that its performance can be analyzed. [Log events](#) may be associated with a region to record progress and relevant values. The API also allows annotation of higher-level [tasks](#), which may involve work across many routines.

The links below display, for each region and task, a histogram of its execution times. Each histogram bucket contains a sample trace that records the sequence of events such as goroutine creations, log events, and subregion start/end times. For each task, you can click through to a logical-processor or goroutine-oriented view showing the tasks and regions on the timeline. Such information may help uncover which steps in a region are unexpectedly slow, or reveal relationships between the data values logged in a request and its running time.

- [User-defined tasks](#)
- [User-defined regions](#)

Illustration 244. Drill down into trace regions for bottleneck analysis

Click on [/usertasks](#) (the [User-defined tasks](#) link) to see a summary table of all task types in your trace. In this example, you will see entries for ["rootTask"](#) and ["handleRequest"](#) tasks.

Each row in the summary shows the task type, how many tasks of that type were created, and a distribution of durations:

Duration distribution (complete tasks)		
Task type	Count	
handleRequest	10	158.489319ms
		251.188643ms
		398.10717ms
		630.957344ms
rootTask	1	

Illustration 245. Task summary showing count and duration buckets

Keep in mind, these timing statistics only include tasks that have finished. The [rootTask](#) usually does not show a distribution because it is probably still running when the trace ends. For the [handleRequest](#) tasks, you can see all 10 requests we created. The duration buckets break down as follows:

- [158.489319ms](#) : 1 task finished between 158.489319ms and 251.188643ms
- [251.188643ms](#) : 8 tasks finished between 251.188643ms and 398.10717ms
- [398.10717ms](#) : 1 task finished between 398.10717ms and 630.957344ms
- [630.957344ms](#) : This bucket shows up even though there are no tasks in it. The Go trace system uses a logarithmic histogram, so every bucket from minimum to maximum is shown, even if some have zero tasks.

This means most tasks (8 out of 10) took between 251ms and 398ms to complete. This matches what we would expect, given the random sleep ranges in our

example code.

To see the exact sequence of events for a specific task duration, click on the time range or the count in the summary. This takes you to the `/usertask?type=handleRequest` view, which shows a chronological breakdown for each individual `handleRequest` task:

User Task: "handleRequest"			
When	Elapsed	Goroutine	Events
0.043904384s	214.18112ms		<u>Task 6 (goroutine view)</u> (complete)
0.043904384	. 43904384	7	task "handleRequest" (D 6, parent 1) begin
0.043908096	. 3712	7	log "started request 4"
0.043908672	. 576	7	region "requestValidation" begin
0.049555904	. 5647232	7	region "requestValidation" end
0.049563264	. 7360	7	region "processResults" begin
0.049566592	. 3328	8	region "query-0" begin
0.049567168	. 576	8	region "dbQuery" begin
0.049568128	. 960	8	log "id=40"
0.049568448	. 320	8	region "dbConnect" begin
0.049569088	. 640	9	region "query-1" begin
0.049569344	. 256	9	region "dbQuery" begin
0.049570304	. 960	9	log "id=41"
0.049570496	. 192	9	region "dbConnect" begin
0.049580096	. 9600	10	region "query-2" begin
0.049580416	. 320	10	region "dbQuery" begin
0.049580992	. 576	10	log "id=42"
0.049581184	. 192	10	region "dbConnect" begin
0.082761024	. 33179840	10	region "dbConnect" end
0.136847168	. 54086144	8	region "dbConnect" end
0.151650752	. 14803584	9	region "dbConnect" end
0.169120384	. 17469632	10	region "dbQuery" end
0.169132608	. 12224	10	log "result-42"
0.169134144	. 1536	10	region "query-2" end
0.171022272	. 1888128	8	region "dbQuery" end
0.171023168	. 896	8	log "result-40"

Illustration 246. Detailed event timeline for handleRequest user task.

For every event, the trace viewer shows the absolute timestamp since the start of the trace, the time elapsed since the previous event in that task, and the goroutine where the event happened. For example, starting at timestamp 0.043904384s, Task 6 began with the following sequence:

- Initial setup (first 5ms): The task starts, logs "started request 4", and runs the `requestValidation` region, which takes about 5.6ms. You can see this region start at 0.043908672 and end at 0.049555904.
- Fan-out to parallel queries: After validation, it immediately starts the `processResults` region and then launches the three database query goroutines. All of them begin almost at the same time:
  - `query-0` starts at 0.049566592 on goroutine 8
  - `query-1` starts at 0.049569088 on goroutine 9

- `query-2` starts at 0.049580096 on goroutine 10
- The bottleneck: Each query has nested `dbQuery` and `dbConnect` regions.  
Look at the `dbConnect` durations:
  - `query-2's` ``dbConnect`` (goroutine 10): starts at 0.049581184, ends at 0.082761024 = 33ms
  - `query-0's` ``dbConnect`` (goroutine 8): starts at 0.049568448, ends at 0.136847168 = 87ms
  - `query-1's` ``dbConnect`` (goroutine 9): starts at 0.049570496, ends at 0.151650752 = 102ms

This shows that the random sleep in our `dbConnect` region (from 0 to 300ms) is causing the main bottleneck. `query-1` took the longest at 102ms for just the connection phase. This determined the overall request latency, since all queries must finish before the request can complete.

If you prefer a visual representation instead of the text or tabular view, you can click on [Task 6 \(goroutine view\)](#). This view shows the same information but in a much more visual format:

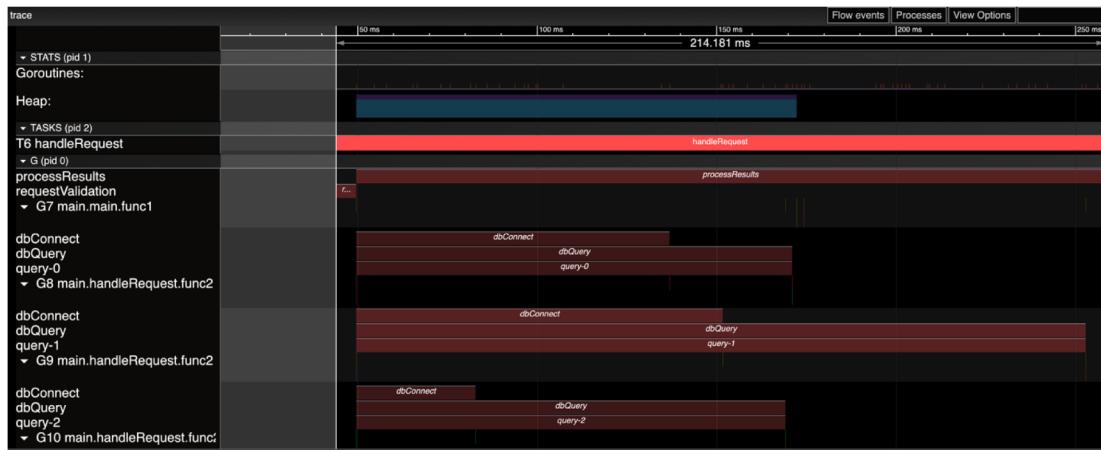


Illustration 247. Visual trace of handleRequest task timeline

There is also a region summary view, which is similar to [/usertask?type=handleRequest](#) and offers the same features. It shows each region instance along with a distribution of durations:

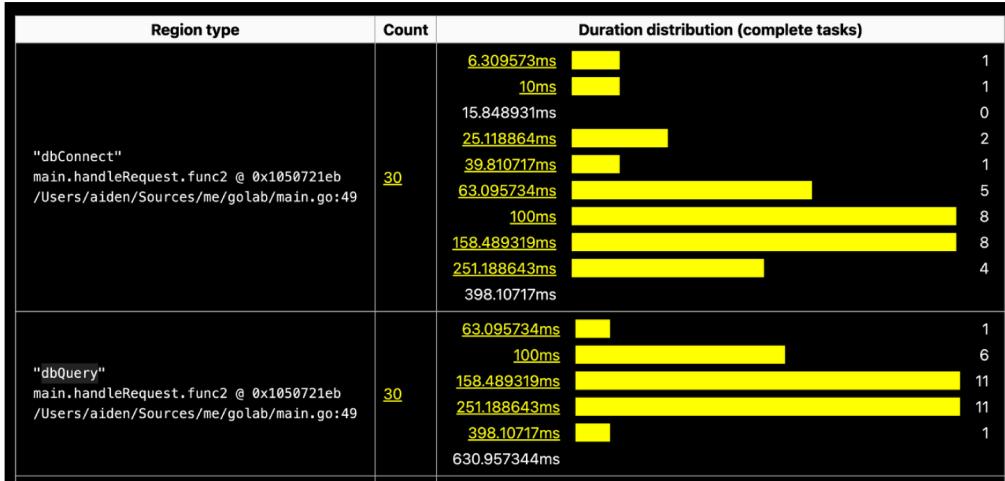


Illustration 248. Region summary: dbConnect and dbQuery durations

The only difference is in the detail view for each region. While it does not display logs, it does show a breakdown similar to what we saw in the goroutine analysis view earlier:

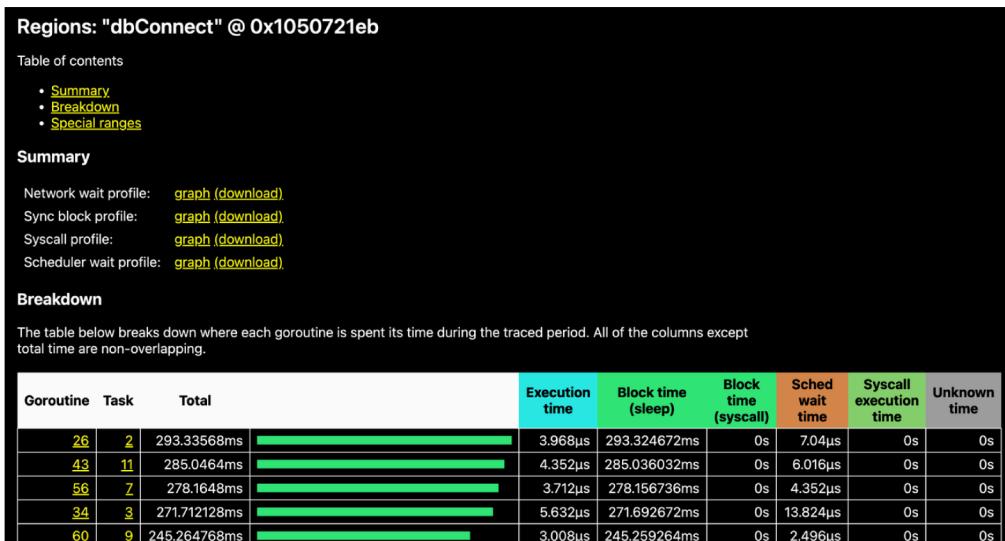


Illustration 249. Detailed breakdown of dbConnect region timing

This kind of visual analysis is really valuable for learning Go. You can actually see how goroutines, channels, and the scheduler interact. For example, you can see how `go func()` creates new goroutines and when they yield to others.

It is important to keep performance in mind when using user annotations. The tracing system is built to be lightweight when tracing is turned off, but it still allocates task objects just in case tracing gets turned on during the task's lifetime.

## Under the Hood

*Understanding how trace works is not required, so feel free to skip this section if you want.*

### Buffering & Generations (full/empty queues)

At the center of Go's tracing system is a global `trace` object that manages everything. This central piece coordinates the whole tracing operation:

```
1 var trace struct {
2     ...
3
4     reading      *traceBuf
5     empty        *traceBuf
6     full         [2]traceBufQueue
7     workAvailable atomic.Bool
8
9     readerGen    atomic.Uintptr // the generation the
10    reader is currently reading for
11
12    flushedGen   atomic.Uintptr // the last
13    completed generation
14
15    ...
16    gen          atomic.Uintptr // the current
17    generation
18 }
19
20 type traceBufQueue struct {
21     head, tail *traceBuf
22 }
```

During program execution, the tracer collects a lot of information—like which goroutines are running, when they block, and when garbage collection occurs. All

of this information is recorded as events in memory. These events need a place to go, and that place is called the trace buffer (`tracebuf`).

These buffers are global, meaning they are shared across the system and not tied to a single goroutine or thread. However, writing directly to these shared buffers would create too much contention because many goroutines could try to write at once. Instead, each thread keeps two private 64 KB trace buffers:

```
type mTraceState struct {
    seqlock atomic.Uintptr
    buf     [2]*traceBuf // Two buffers for
alternating generations
    link   *m
}

type traceBuf struct {
    _ sys.NotInHeap
    traceBufHeader
    arr [64<<10 - unsafe.Sizeof(traceBufHeader{})]byte
}
```

So, when a goroutine running on a thread needs to record a trace event, it writes into its own thread's active buffer for the current generation. Since the buffer is private to the thread, and a thread only runs one goroutine at a time, there is no need for extra locking. The global buffers in `trace` are used when data from these local buffers is ready to be read.

Why are there two buffers (`[2]*traceBuf`) per thread? The tracing system is organized into **generations**. Each generation keeps its trace data separate.

When many threads write events at the same time, but only one goroutine reads them later, they need to avoid getting in each other's way. With only one buffer per thread, there are two main problems:

- If a thread is still writing, the reader cannot safely read the buffer.
- If there is no clear stopping point for writing, memory cannot be safely reused, and memory use could grow forever.

Generations solve both issues.

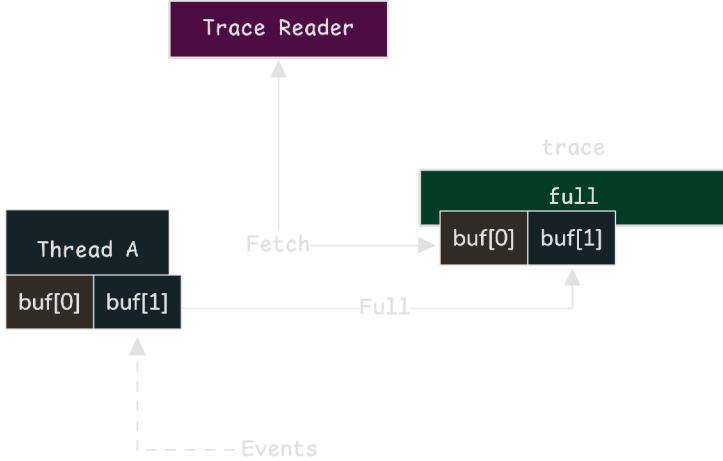


Illustration 250. Buffers ensure threads and readers don't clash

Threads write to their buffer for the current generation. After some time, usually about a second, Go's tracer ends that generation (for example, generation 5). All threads are told to stop writing to the generation 5 buffer (`buf[1]`) and switch to a new buffer for generation 6 (`buf[0]`). Now, the reader can safely read the data from generation 5, knowing that no more events are being written there. When reading is done, the memory for that generation can be used again.

This setup gives us three major benefits:

1. Writers and the reader never access the same data at the same time, so everything stays safe.
2. We can always tell when it's safe to reuse or free memory.
3. The reader always gets a complete and finished set of trace events for each generation, making processing simple.

Advancing to the next generation happens regularly—usually once every second. This time window is long enough to collect useful trace data, but short enough that no single generation becomes too large. The process of moving to the next generation is handled in a background goroutine, so it does not block or slow down your main program.

Other optimizations, like reusing string tables, stack tables, and type tables, also benefit from generations, but we will not cover those here.

## Trace Locker

After a short period, the runtime decides it is time to advance to a new generation. This step is called "generation advancement". Before switching generations, the runtime has to be sure that no thread is still writing to the old generation's buffer. Once the switch happens, the old buffers might get flushed, processed, or freed. To make sure this is safe, the runtime uses a system called the trace locker.

Each thread has a sequence lock (`seqlock`), which is simply an atomic counter:

```
type m struct {
    ...
    trace mTraceState
}

type mTraceState struct {
    seqlock atomic.Uintptr // seqlock indicates this
    thread is writing to a trace buffer.
    buf     [2]*traceBuf  // Per-thread traceBuf for
    writing. Indexed by trace.gen%2.
    link   *m             // Snapshot of alllink or
    frealink.
}
```

When a thread wants to write trace events, it calls `traceAcquire()`, which increases its sequence lock (`seqlock`) by 1.

```
seq := mp.trace.seqlock.Add(1)
```

If the sequence lock is an odd number, it means "this thread is writing trace events now." When the thread finishes, it calls `traceRelease()`, which adds 1 again, making the sequence lock even. So, an even sequence lock means "this thread is not writing right now."

When it's time to move to a new generation, the tracer increases the generation number, for example to 5, and all threads with a new trace locker start writing to the generation 5 buffer. The tracer can then safely process or reuse the generation 4 buffers for all threads, but only if the sequence lock is even.

Why? Because there could be a race where a thread is still writing to the generation 4 buffer (`buf[0]`) when the tracer switches to generation 5:

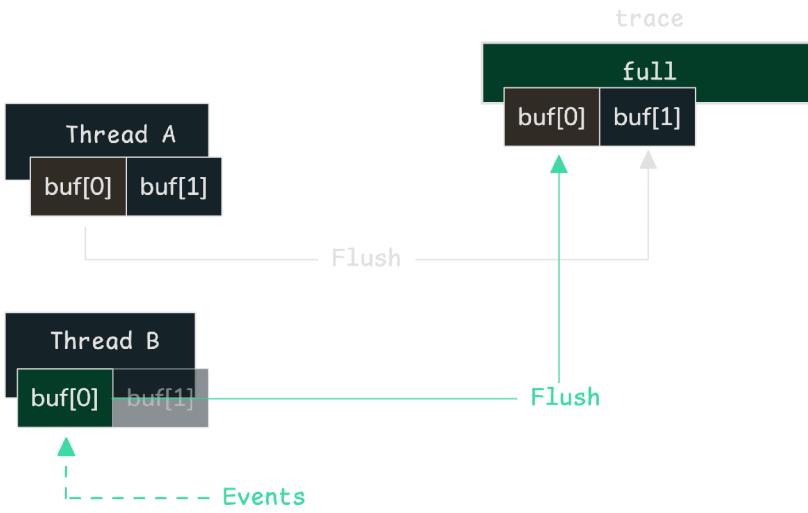


Illustration 251. Thread B hasn't finished with previous generation.

The next time thread B grabs the trace locker, it will write to the generation 5 buffer ( `buf[1]` ). The tracer checks the sequence lock of each thread, and if the sequence lock is even, it is safe to say that the thread has switched to the new generation.

If any thread has an odd sequence lock, that means it is still writing trace data.

In that case, the tracer waits until the thread is done:

```

for mToFlush != nil {
    prev := &mToFlush
    for mp := *prev; mp != nil; {
        if mp.trace.seqlock.Load()%2 != 0 {
            // The thread is writing. Come back
            to it later.
            ...
            continue
        }
        ...
    }

    // Yield only if we're going around the loop again.
    if mToFlush != nil {
        osyield()
    }
}

```

The thread might be writing to the generation 5 buffer ( `buf[1]` ) or the generation 6 buffer ( `buf[0]` ), and that is fine. The tracer skips over it, processes

other threads, and comes back later to check the sequence lock again.

In theory, the tracer could do a generation-aware check. It could flush the generation 5 buffer if it knows the thread is already working with the generation 6 buffer. However, this would need extra state. It would require reading the writer's current generation (which means another atomic read and a memory fence to keep everything in sync).

If the thread is not writing, then the tracer flushes its trace buffer to the global container (`trace.full`). A thread that has had its buffer flushed does not need to wait for other threads. It can immediately keep collecting trace data in the new generation's buffer.

The key part of this system is the `traceLocker` structure. It gives exclusive access to a thread's trace resources when emitting events:

- Any part of the runtime that wants to write a trace event must first acquire a locker (`traceAcquire()`), which makes the sequence lock odd.
- When done, it releases the lock (`traceRelease()`), which makes the sequence lock even again.

## Trace Writer

The trace writer acts as the link between the trace locker and the buffer management system. It lets you write trace events while taking care of all the buffer allocation, flushing, and synchronization details:

```
type traceWriter struct {
    traceLocker
    *traceBuf
}
```

The trace writer uses a fluent API pattern, where every method returns a new trace writer instance. This lets you chain methods together. For example, to flush and end a trace writer, you can do:

```
statusWriter.flush().end()
```

Now that you have seen all the main parts of the tracer, it is time to see how they work together.

## Event Pipeline

When tracing starts using `StartTrace()`, the system goes through several initialization steps to reset its state and record the initial runtime state. However, this initialization requires stopping the world (STW).

### STW Bootstrap

Once tracing is enabled, goroutines that emit events will automatically write status events for themselves and also for other goroutines they interact with, if those have not been traced yet. To understand this better, let's see how a goroutine writes status events:

```
trace := traceAcquire()  
... PROBLEM WINDOW  
  
casgstatus(gp, _Gwaiting, _Grunnable)  
if trace.ok() {  
    trace.GoUnpark(gp, 2)  
    traceRelease(trace)  
}
```

This code should look familiar after reading about the trace locker. To recap: when a goroutine wants to emit trace events, it first acquires a trace locker. If tracing is enabled, `traceAcquire()` returns a valid trace locker—think of it as permission to emit events.

If tracing is not enabled, `traceAcquire()` returns an invalid trace locker. The goroutine then performs its state change and, if needed, emits the trace event (like `GoUnpark` above). After emitting, it must call `traceRelease()` to clean up the tracing state.

A risky scenario appears when a goroutine calls `traceAcquire()` before tracing is enabled, and then tracing is enabled while that goroutine is still running. The sequence goes like this:

1. Goroutine A calls `traceAcquire()` while tracing is still off and gets an invalid locker.
2. Trace initialization begins and tracing is turned on.
3. Goroutine B calls `traceAcquire()` and gets a valid trace locker because tracing is now on.
4. Goroutine B reads goroutine C's status, sees it as waiting, and emits a waiting event for goroutine C.

5. Goroutine A changes goroutine C's status from waiting to runnable. But since it had an invalid locker from before, it does not emit the new state for C.

Now the trace is inconsistent: it shows goroutine C as waiting, but it is actually runnable. Later events will expect that state change, but the transition was never recorded.

Stopping the world removes this problem. When the world restarts, every goroutine's next call to `traceAcquire()` will return a valid trace locker. No goroutine will be left holding a stale, invalid locker.

While the world is stopped, the runtime also records an initial, explicit status event for every goroutine and every P (processor):

Figure 140. StartTrace STW (src/runtime/trace.go)

```
func StartTrace() error {
    ...
    stw := stopTheWorld(stwStartTrace)
    ...

    // Make sure a ProcStatus is emitted for every P,
    while we're here.
    for _, pp := range allp {
        tl.writer().writeProcStatusForP(pp, pp ==
t1.mp.p.ptr()).end()
    }
    ...
}

    startTheWorld(stw)
}
```

This way, the trace has a clear baseline for all goroutines and processors, even if nothing happens immediately after tracing starts.

#### Event Writing (proc, goroutine, STW, GC, user)

The execution tracer acquires a trace locker at many important points throughout the Go runtime to capture execution events. Let's quickly walk through some of the key types of events.

Processor events track the lifecycle and state changes of Go's logical processors (Ps), which are the execution contexts for running goroutines:

```

const (
    // Procs.
    traceEvProcsChange // current value of GOMAXPROCS
[timestamp, GOMAXPROCS, stack ID]
    traceEvProcStart // start of P [timestamp, P ID, P
seq]
    traceEvProcStop // stop of P [timestamp]
    traceEvProcSteal // P was stolen [timestamp, P ID,
P seq, M ID]
    traceEvProcStatus // P status at the start of a
generation [timestamp, P ID, status]
)

```

Goroutine lifecycle events provide a full picture of goroutine behavior, covering creation, execution, blocking, unblocking, and destruction:

```

const (
    // Goroutines.
    traceEvGoCreate // goroutine creation
[timestamp, new goroutine ID, new stack ID, stack ID]
    traceEvGoCreateSyscall // goroutine appears in
syscall (cgo callback) [timestamp, new goroutine ID]
    traceEvGoStart // goroutine starts
running [timestamp, goroutine ID, goroutine seq]
    traceEvGoDestroy // goroutine ends
[timestamp]
    traceEvGoDestroySyscall // goroutine ends in
syscall (cgo callback) [timestamp]
    traceEvGoStop // goroutine yields its
time, but is runnable [timestamp, reason, stack ID]
    traceEvGoBlock // goroutine blocks
[timestamp, reason, stack ID]
    traceEvGoUnblock // goroutine is unblocked
[timestamp, goroutine ID, goroutine seq, stack ID]
    traceEvGoSyscallBegin // syscall enter
[timestamp, P seq, stack ID]
    traceEvGoSyscallEnd // syscall exit
[timestamp]
    traceEvGoSyscallEndBlocked // syscall exit and it
blocked at some point [timestamp]
    traceEvGoStatus // goroutine status at
the start of a generation [timestamp, goroutine ID, M ID,
status]
)

```

Next, STW events capture global runtime pauses that affect all goroutines. These events mark the start and end of stop-the-world (STW) phases, recording when all goroutines are paused and when they are allowed to resume. The "kind" parameter shows the reason for the STW pause.

```

const (
    // STW.
    traceEvSTWBegin // STW start [timestamp, kind]
    traceEvSTWEnd   // STW done [timestamp]
)

```

GC events track all phases of garbage collection, from high-level GC cycles to detailed sweep and mark assist operations. Heap allocation and heap goal events track memory usage changes that drive GC decisions. Mark assist events show when user goroutines help with GC marking work:

```

const (
    // GC events.
    traceEvGCActive           // GC active [timestamp,
seq]      traceEvGCBegin        // GC start [timestamp,
seq, stack ID]      traceEvGCEnd         // GC done [timestamp,
seq]      traceEvGCSweepActive // GC sweep active
[timestamp, P ID]      traceEvGCSweepBegin // GC sweep start
[timestamp, stack ID]      traceEvGCSweepEnd   // GC sweep done
[timestamp, swept bytes, reclaimed bytes]
    traceEvGCMarkAssistActive // GC mark assist active
[timestamp, goroutine ID]
    traceEvGCMarkAssistBegin // GC mark assist start
[timestamp, stack ID]
    traceEvGCMarkAssistEnd  // GC mark assist done
[timestamp]
    traceEvHeapAlloc          // gcController.heapLive
change [timestamp, heap alloc in bytes]
    traceEvHeapGoal           // gcController.heapGoal()
change [timestamp, heap goal in bytes]
)

```

Annotation events capture user-defined trace information that developers can add for debugging and profiling. These match the user-facing trace API and let you add custom instrumentation to your code.

Labels add metadata to goroutines, tasks provide hierarchical grouping for related work, regions mark spans of interest, and logs record key-value data:

```

const (
    // Annotations.
    traceEvGoLabel        // apply string label to
current running goroutine [timestamp, label string ID]

```

```

        traceEvUserTaskBegin // trace.NewTask [timestamp,
internal task ID, internal parent task ID, name string ID,
stack ID]
        traceEvUserTaskEnd   // end of a task [timestamp,
internal task ID, stack ID]
        traceEvUserRegionBegin // trace.{Start,With}Region
[timestamp, internal task ID, name string ID, stack ID]
        traceEvUserRegionEnd   // trace.{End,With}Region
[timestamp, internal task ID, name string ID, stack ID]
        traceEvUserLog        // trace.Log [timestamp,
internal task ID, key string ID, stack, value string ID]
)

```

Each event is represented by an `int` constant. The comments next to each constant describe what the event means and what arguments it includes. These events are collected throughout the runtime using the `traceAcquire()` and `traceRelease()` pattern. For example, GC events are emitted during garbage collection as in this snippet:

```

trace := traceAcquire()
if trace.ok() {
    trace.GCStart()
    traceRelease(trace)
}

```

When a goroutine acquires the trace locker, it prevents preemption by binding itself to its thread (`M`). Any stop-the-world pause by the garbage collector or other runtime operations will be delayed until the trace locker is released. The goroutine then acquires the sequence lock (`seqlock`) on its thread's trace state by incrementing it to an odd value. This marks that the thread is actively writing trace events.

The trace locker now holds two things: the thread (`M`) and the current generation number:

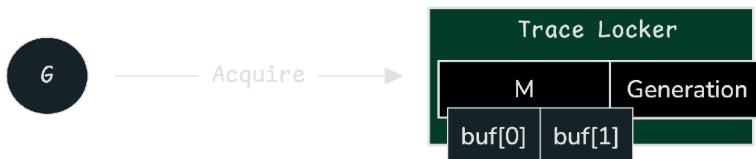


Illustration 252. Trace locker binds goroutine to thread and generation

If the tracer is not enabled (for example, the current generation is `0` or the global `trace.enabled` is `false`), it returns an invalid trace locker with generation set to `0`:

Figure 141. traceLocker.ok() (src/runtime/traceruntime.go)

```
// ok returns true if the traceLocker is valid (i.e. tracing
// is enabled).
//
// nosplit because it's called on the syscall path when
// stack movement is forbidden.
//
//go:nosplit
func (tl traceLocker) ok() bool {
    return tl.gen != 0
}
```

When a goroutine emits an event, it creates a trace writer:

```
func (tl traceLocker) writer() traceWriter {
    return traceWriter{traceLocker: tl, traceBuf:
        tl.mp.trace.buf[tl.gen%2]}
}
```

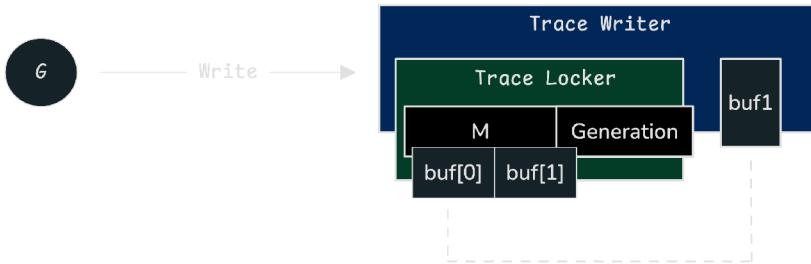


Illustration 253. Writer connects trace locker to correct buffer

For example, if we are on generation 1, `tl.gen%2` equals 1, so `tl.mp.trace.buf[1]` points to the second buffer. The trace writer now acts as the bridge that holds both the locking logic and the actual buffer where events will be written.

The trace system follows an important rule: every active scheduling resource (goroutines and processors) must have their status explicitly recorded at least once in each generation. This status event acts as the baseline state for the resource, so all later events in that generation can be interpreted correctly. If a goroutine participates in any traced event during a generation, the tracer automatically emits a status event for it if none has been recorded yet:

```
func (tl traceLocker) eventWriter(goStatus traceGoStatus,
    procStatus traceProcStatus) traceEventWriter {
    // Get the underlying trace writer that will write
    // to the trace buffer
```

```

        w := tl.writer()

        // Check if we need to emit a processor status event
        // Write a ProcStatus event to establish the
processor's baseline state
        if pp := tl.mp.p.ptr(); pp != nil &&
!pp.trace.statusWasTraced(tl.gen) &&
pp.trace.acquireStatus(tl.gen) {
            w = w.writeProcStatus(uint64(pp.id),
procStatus, pp.trace.inSweep)
        }

        // Check if we need to emit a goroutine status event
        // Write a GoStatus event to establish the
goroutine's baseline state
        if gp := tl.mp.curg; gp != nil &&
!gp.trace.statusWasTraced(tl.gen) &&
gp.trace.acquireStatus(tl.gen) {
            w = w.writeGoStatus(uint64(gp.goid),
int64(tl.mp.procId), goStatus, gp.inMarkAssist, 0 /* no
stack */)
        }
        return traceEventWriter{w}
    }
}

```

However, the trace writer does not write the event immediately. It first checks that the buffer has enough free space for the event:

```

// ensure makes sure that at least maxSize bytes are
available to write.
//
// Returns whether the buffer was flushed.
func (w traceWriter) ensure(maxSize int) (traceWriter, bool)
{
    refill := w.traceBuf == nil || !w.available(maxSize)
    if refill {
        w = w.refill(traceNoExperiment)
    }
    return w, refill
}

```

If the buffer does not have enough space, the writer will flush the current buffer to the `trace` system's `full` buffer queue, which holds all completed and immutable buffers. It then refills itself with a new empty buffer. The full process of flushing and getting a new buffer is handled in the `refill` method.

Let's first look at how a buffer is flushed to the global trace system:

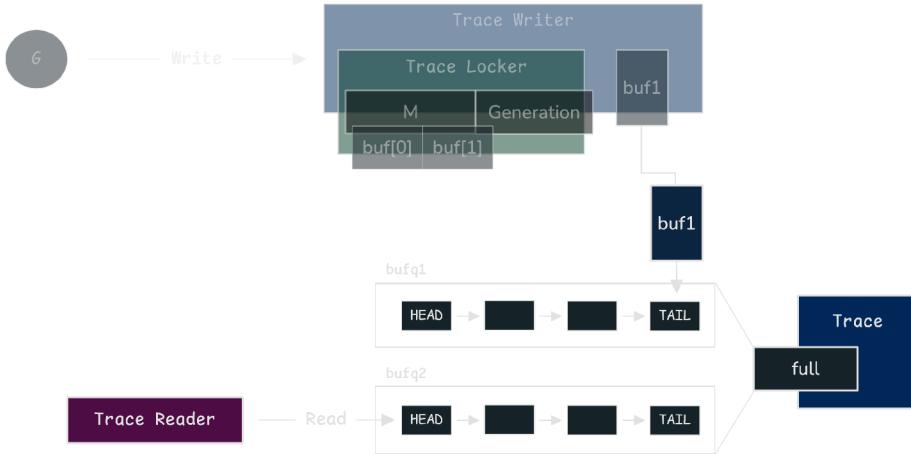


Illustration 254. Buffers are flushed into full queue

The `trace.full` structure is made of two linked lists—one for each generation. Each list uses a simple queue with `head` and `tail` pointers. When a buffer is finished (flushed), it is added to the end of the queue for its generation. This is done by updating the current tail's link and moving the `tail` pointer to the new buffer.

When the trace reader needs to read, it removes buffers from the front of the queue by following the `head` pointer and moving it along the link chain.

That covers flushing, but where does the new buffer come from for the thread's trace writer?

There are two ways to refill a buffer. The writer can either reuse a buffer from the tracer's `empty` list, or, if there are no empty buffers available, allocate a new 64 KB buffer from the operating system:

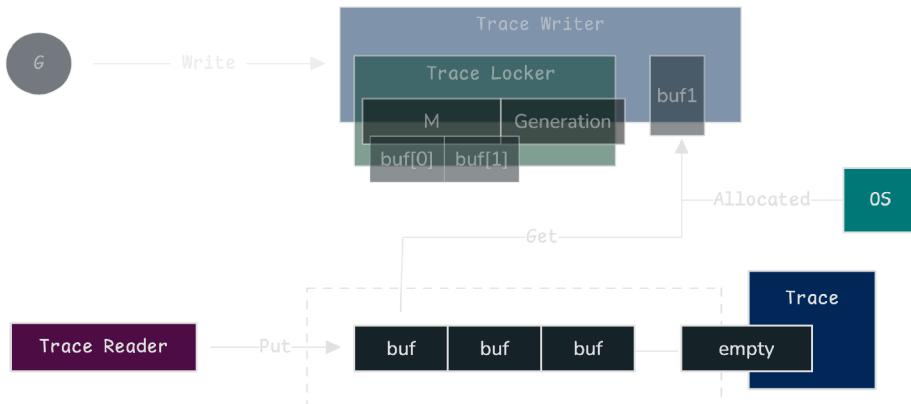


Illustration 255. Trace writer requests buffer from empty list

The `empty` list is another linked list, separate from the `full` list. It holds buffers that are no longer in use—buffers that were previously full, have been read completely by the trace reader, and then returned for reuse.

The reader plays a major role in this system by returning buffers to the `empty` list after they are finished:

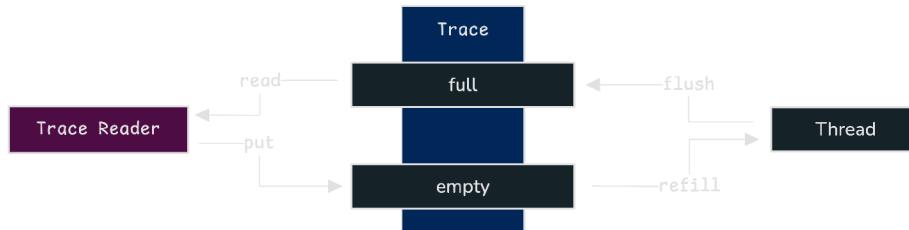


Illustration 256. Full and empty buffers coordinate tracing

Here is the source code for the refill function:

Figure 142. refill (src/runtime/tracebuf.go)

```
func (w traceWriter) refill(exp traceExperiment) traceWriter {
    systemstack(func() {
        lock(&trace.lock)
        if w.traceBuf != nil {
            traceBufFlush(w.traceBuf, w.gen) // flush
        }

        // refill
        if trace.empty != nil {
            w.traceBuf = trace.empty
            trace.empty = w.traceBuf.link
            unlock(&trace.lock)
        } else {
            unlock(&trace.lock)
            w.traceBuf = (*traceBuf)
            (sysAlloc(unsafe.Sizeof(traceBuf{}), &memstats.other_sys))
            if w.traceBuf == nil {
                throw("trace: out of memory")
            }
        }
    })
}
```

Each event is written along with its "timestamp delta". Each trace buffer keeps a `lastTime` field to track the timestamp of the most recent event written to that buffer. When writing a new event, the system calculates the difference between the current time and `lastTime`, and stores only this delta in the trace data:

```
// traceEventWrite is the part of traceEvent that actually
writes the event.
func (w traceWriter) event(ev traceEv, args ...traceArg)
traceWriter {
    // Make sure we have room.
    w, _ = w.ensure(1 +
(len(args)+1)*traceBytesPerNumber)

    // Compute the timestamp diff that we'll put in the
trace.
    ts := traceClockNow()
    if ts <= w.traceBuf.lastTime {
        ts = w.traceBuf.lastTime + 1
    }
    tsDiff := uint64(ts - w.traceBuf.lastTime)
    w.traceBuf.lastTime = ts

    // Write out event.
    w.byte(byte(ev))
    w.varint(tsDiff)
    for _, arg := range args {
        w.varint(uint64(arg))
    }
    return w
}
```

This delta-based method reduces the space required to store timestamps, since most deltas are much smaller than full timestamps. Smaller numbers are more efficient to encode using variable-length encoding, so this approach saves significant space in the trace data.

Each trace event in Go's execution trace is stored in a binary and compact format that follows a specific layout:

```
[event type][timestamp delta][arg0][arg1][arg2]...
```

After all events are written, the goroutine calls `traceWriter.end()` to complete the operation. This method updates the trace buffer pointer in the M's trace state at `mp.trace.buf[gen%2]`, making sure any changes (like acquiring a new buffer during writing) are saved for future writes by this thread:

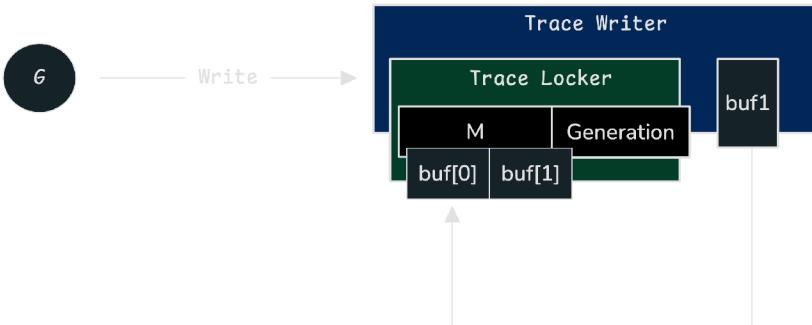


Illustration 257. Trace writer updates buffer reference after writing

Finally, the goroutine releases the trace locker, increments the sequence lock again (making it even), and also releases the M so that preemption can happen.

This is how the trace mechanism works with careful design:

- Each thread writes its trace events into its own private buffer, so there is no contention over shared resources or unnecessary waiting.
- The runtime occasionally creates a new generation, making sure all the old data is sealed and safe to read by using the sequence lock, while new events are written into a fresh buffer.
- Delta-based timestamp encoding reduces the space needed to store timestamps, making the trace more efficient.

All of these are practical design choices and can be applied to other high performance applications.

## 7. Profile-Guided Optimization (PGO)

Profile-Guided Optimization (PGO), sometimes called Feedback-Directed Optimization (FDO), is a compiler optimization technique. It uses real data from running your application to help the compiler make better decisions the next time it builds your code.

With PGO in Go, the process is a loop: you build and run your app under real conditions and collect CPU profiles. These profiles show which parts of your code run most often (called "hot" paths) and how often functions are called. You then give these profiles back to the Go compiler for the next build. The compiler uses this feedback to make smarter choices and tries to build a faster program:

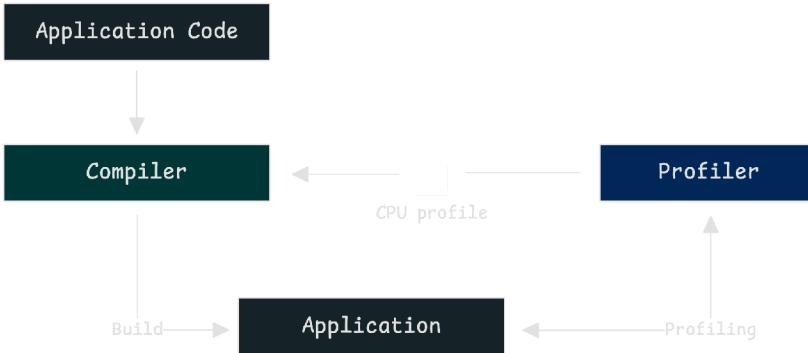


Illustration 258. Profile-guided optimization loop in Go compilation

PGO was first added to Go in version 1.20, released in February 2023. The initial work started in September 2022, with a commit by Raj Barik from Uber (commit [99862cd57d](#)) [[g9986](#)]. This made it possible for the Go compiler to use profile-guided inlining.

The first version of PGO focused on inlining, meaning the compiler uses real runtime data to decide which functions to inline instead of relying solely on rules or guesses. In later versions, such as Go 1.21 and Go 1.22, PGO added support for devirtualization. This optimization allows the compiler to turn certain indirect calls (such as calls through interfaces or function values) into direct calls if the profile shows that a specific concrete type is used most of the time. This can reduce CPU usage, and many programs have seen CPU time decrease by 2 to 14% after using PGO in Go 1.21 or 1.22.

PGO also makes small improvements to SSA code generation, mostly with loop optimizations and code layout. These details give a little extra performance, but they do not change how your code works in a major way. We focus on the main PGO features, inlining and devirtualization, since these have the biggest impact and are most useful.

## Using PGO in Practice

Let's look at how to use PGO with a simple example. We will use the same code as in the CPU profiling section:

```

func hot() {
    x := 0
    for i := range 1000000000 {
        x += i
    }
    fmt.Sprintln("prevent inlining")
}

```

```

        fmt.Sprintln("prevent inlining")
    }

func cold() {
    time.Sleep(5 * time.Millisecond)
    fmt.Sprintln("prevent inlining")
}

func indirectHot() {
    hot()
    fmt.Sprintln("prevent inlining")
}

func main() {
    f, err := os.Create("cpu.prof")
    if err != nil {
        panic(err)
    }
    defer f.Close()

    if err := pprof.StartCPUProfile(f); err != nil {
        panic(err)
    }
    defer pprof.StopCPUProfile()

    indirectHot()
    for range 150 {
        hot()
        cold()
    }
}

```

In this example, we add more "useless" `fmt.Sprintln` calls to make sure the compiler does not inline these functions by default. This works because calling two functions inside is enough to stop the inliner due to the cost. Also, we moved `indirectHot` outside the loop.

The first step is to build your Go program the usual way (without PGO) and run it in a real environment. This should be as close to production as possible:

```
$ go build -o main main.go
$ ./main
```

Next, use Go's profiling tools to find out where your program uses the most CPU **while it is running real workloads**. In our example, the program automatically saves the CPU profile data to a file called `cpu.prof`, so you do not need to do anything extra. Here is how to view the CPU profile graph:

```
$ go tool pprof -http=:8080 main cpu.prof
```

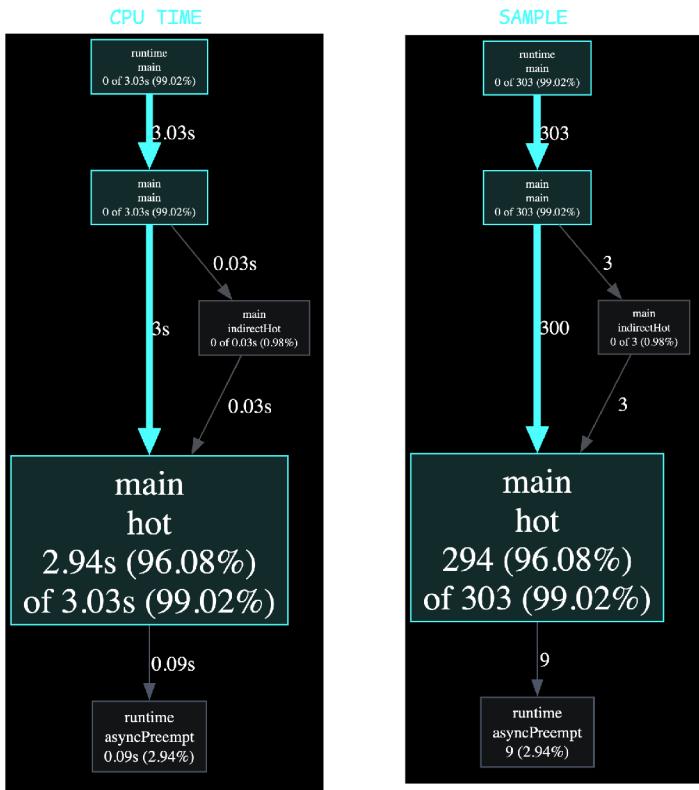


Illustration 259. pprof output shows the main execution path

For the next build, turn on PGO by giving the collected profile to the Go compiler. Go makes this easy: if you rename your profile to `default.pgo` and put it in your main module's directory, running `go build` will automatically find the profile and use PGO optimizations:

```
$ mv cpu.prof default.pgo
$ go build -o main main.go
```

This automatic PGO feature makes it simple to add PGO to your usual build process. You can commit a `default.pgo` file into your repository alongside your code, and all future builds will automatically benefit from the performance improvements.

With Go 1.21 and newer, auto-detection is always on. In Go 1.20, however, you had to enable PGO manually.

If you want more control, use the `-pgo` flag with the build or test commands:

```

# Build the binary with a specific profile file.
$ go build -pgo=cpu.pprof -o main main.go

# Auto-detects a default.pgo file in the package if present.
$ go build -pgo=auto -o main main.go

# Turn off PGO.
$ go build -pgo=off -o main main.go

```

This explicit mode is helpful when you have several profiles for different scenarios, or when your profiles are in non-standard places, or when you are trying different optimization strategies. The new binary should run faster under the same workload because the compiler will inline more code.

Let's run the optimized binary and check the profile:

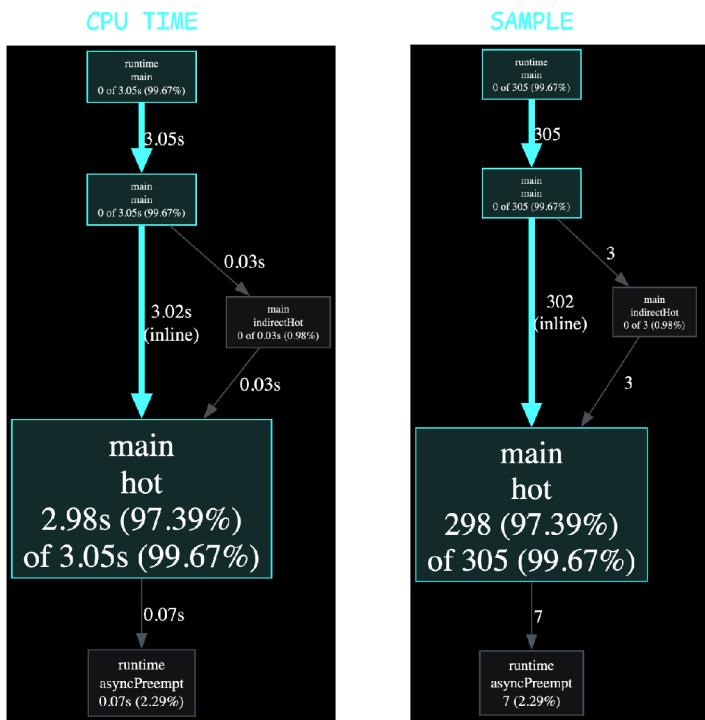


Illustration 260. Profiling shows most time spent in inlined code

The arrow from `main` to `main.hot` is marked as inlined because `main.hot` is a hot path. However, the arrow from `main.indirectHot` to `main.hot` is not marked as inlined. This is the power of PGO: it considers not only hot functions but also hot call paths. As your application changes or receives new workloads, you can continue collecting new profiles and rebuild with PGO. This creates a cycle: profile, optimize, deploy, and repeat.

Keep in mind that the result is not always the same. Sometimes, the call to `hot` inside `indirectHot` may also be inlined. This depends on factors we will discuss in the next section.

## PGO Mechanics

You already know how CPU profiling works from the profiling section. The profiler samples the program's execution at regular time intervals, for example, every 10 milliseconds, and records the call stack at each point.

Over time, this builds up a statistical view of which parts of your program are "hot." A "hot" area is one that shows up often in the samples, meaning the program spends a lot of time there.

The stack traces show the full context—not just which function is running, but the whole chain of calls that led to it. So the profiler does not just say `hot()` is hot. It can see that `hot()` is hot when called from `main`, but not when called from `indirectHot`.

Here is a detail to keep in mind: a function that runs very quickly but is called a lot may not show up as hot, while a function that runs slowly and takes up a lot of CPU time will appear more in the samples.

## Build Process (what the toolchain does)

Let's see what happens during the build process when you use PGO. We use the same method as in Chapter 5 to look inside the build steps.

```
$ go build -pgo=default.pgo -n main.go
```

You can skip `-pgo=default.pgo` because Go will find this file in the application's root directory automatically. Here, we make it explicit just for the example.

The first step in the build is to create the artifact directory:

```
$ mkdir -p $WORK/b007/
```

A quick reminder: `$WORK/b007/` is just a directory the Go build system makes for the current build action. The name `b007` is a unique ID for that action, like `b001`, `b002`, `b007`, and so on. In this case, `b007` is the build action for PGO

preprocessing. The number does not matter, it just means it is the seventh action in this build.

So, Go creates a dedicated directory for PGO preprocessing. Next, it runs the `preprofile` tool:

```
$ /tools/preprofile -o $WORK/b007/pgo.preprofile -i  
~/theanatomyofgo/default.pgo
```

What is the `preprofile` tool, and why do we need it?

The PGO system can use two main types of profile files. The first is the standard `pprof` format, which is Go's built-in profiling format. This format contains extensive information, including stack traces, sample counts, timing data, and additional metadata. In our example, we are using this standard `pprof` format.

The second format is a custom serialized intermediate representation created specifically for PGO workflows. It uses a simple text-based structure. The file begins with the header `GO PROFILE V1`, followed by a list of caller names, callee names, and information about call sites with their associated weights:

```
GO PROFILE V1  
caller_name  
callee_name  
<call site offset> <call edge weight>  
...
```

If you use the first format (the standard `pprof`), the Go build system will automatically convert it to this second format with the `preprofile` tool.

You can also run this tool yourself to do the conversion:

```
$ go tool preprofile -i default.pgo -o pgo.preprofile
```

Here is what the `pgo.preprofile` file looks like in our example:

```
GO PROFILE V1  
main.main  
main.hot  
14 293  
main.hot  
runtime.asyncPreempt  
2 9  
runtime.wakeNetpoll  
runtime.kevent
```

```

7 2
main.indirectHot
main.hot
1 1
runtime.semasleep
runtime.pthread_cond_wait
32 1

```

To keep things simple, we do not focus on most runtime functions that show up in the call graph, such as `runtime.wakeNetpoll` or `runtime.semasleep`. We make an exception for `runtime.asyncPreempt`, since it can affect the user code call path from `main.main`.

The profile format always starts with the header `GO PREPROFILE V1`. After that, the file is organized in groups of three lines: the caller function name, the callee function name, and then two numbers. The numbers show the call site offset and the call edge weight. Entries are sorted by weight, with the highest weights at the top. This is why `main.main → main.hot` is first, with a weight of 293.

To understand what this weight means, let's look back at the CPU profile graph before optimization:

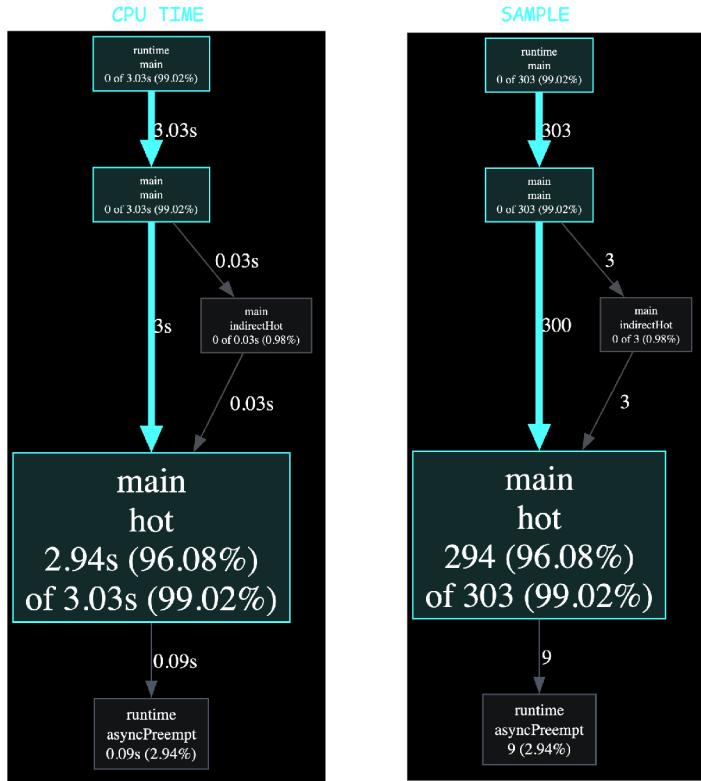


Illustration 261. pprof output shows the dominant execution path

The call path from `main.main` to `main.hot` had 300 samples when profiling ran. This means there were 300 stack traces showing `main.main` as the caller and `main.hot` as the callee. Out of those, 293 samples ended with `main.hot` at the bottom of the stack. The other 7 samples ended with `runtime.asyncPreempt` as the last function on the stack.

The preprocessing step happens only once per build, even though the profile will be used by many different package compilations. The Go build system saves the preprocessed profile:

```
$ cp $WORK/b007/pgo.preprofile /go-build/8f/8f75b55a6905032fe8125e6858df1d85edd5ea636685e288d1b0c7b68bfba807-d # internal
```

If you build again later, the system uses this long-named preprofile file instead of running the `preprofile` tool again.

After preprocessing, every `compile` command in the build now uses the same preprocessed profile:

```
$ /tool/compile -o $WORK/b030/_pkg_.a ... -goversion go1.23.9 -pgoprofile=$WORK/b007/pgo.preprofile ... -pack /go/src/internal/race/doc.go /go/src/internal/race/norace.go  
$ /tool/compile -o $WORK/b009/_pkg_.a ... -goversion go1.23.9 -pgoprofile=$WORK/b007/pgo.preprofile ... -pack /go/src/internal/unsafeheader/unsafeheader.go
```

This means every package compilation (`internal/race`, `internal/unsafeheader`, `math`, `runtime`, and so on) has access to the same PGO data.

## Preprocess: GO PREPROFILE V1

Parsing `pprof` profiles takes a lot of compute power. Without preprocessing, every single call to the compiler would need to parse the same profile over and over, building call graphs, finding function names, and computing edge weights each time. This is wasteful, especially in large builds where many packages use the same profile.

When the build system looks at your profile and does not see the `GO PROFILE V1` header, it knows preprocessing is needed and will generate the call graphs:

Figure 143. PGO IsSerialized (src/cmd/internal/pgo/deserialize.go)

```
const serializationHeader = "GO PREPROFILE V1\n"

func IsSerialized(r *bufio.Reader) (bool, error) {
    hdr, err := r.Peek(len(serializationHeader))
    if err == io.EOF {
        // Empty file.
        return false, nil
    } else if err != nil {
        return false, fmt.Errorf("error reading
profile header: %w", err)
    }

    return string(hdr) == serializationHeader, nil
}
```

The graph-building process works on each sample, and each sample is a captured stack trace. For every frame in the stack, it creates a node for the function and an edge for the call between them.

The system only looks at the last two frames in each stack trace to define the actual call edge. There are a few reasons for this:

- The higher frames in the stack often show initialization code rather than real hot paths in your own code. For example, every Go program has the call chain `runtime.main → main.main → user functions`. If included, these would dominate the profile, but they are not locations where meaningful optimization occurs.
- The higher frames might be called only once but still appear in many samples. This is because they remain on the stack while deeper functions execute.

So, the system finds caller-callee pairs from the end of the stack traces. For each pair of adjacent frames, it creates a call edge that records the caller name, callee name, the call site offset inside the caller, and the weight (how many samples saw this call relationship).

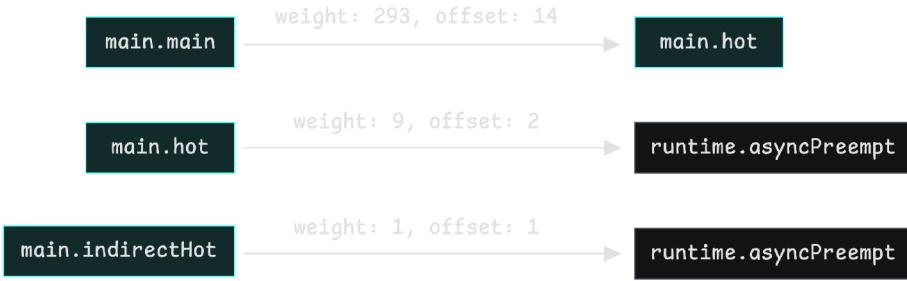


Illustration 262. Profiled call edges between Go functions

These edges are sorted by weight, highest to lowest. You can see the full output from processing in the previous section:

```

GO PREPROFILE V1
main.main
main.hot
14 293
main.hot
runtime.asyncPreempt
2 9
runtime.wakeNetpoll
runtime.kevent
7 2
main.indirectHot
main.hot
1 1
runtime.semasleep
runtime.pthread_cond_wait
32 1

```

The profile holds call edge information based on the source code from an earlier build. Between builds, things can change in the code that affect how the profile data matches up with the current source. Because of this, the Go compiler needs a way to match old profile data to the new code. Let's look at how this matching works and why it can sometimes break.

## Matching: Call Edges & Offsets

When you compile with PGO, the compiler has to bring together two different versions of your program. The profile is made from old execution data. It includes function names and call site locations from a previous build. The current build has the actual IR (Intermediate Representation) of your functions as they are now. The job is to match up the profile's call edges to the call sites in the new code.

To do this, the compiler needs to understand the source code layout. Matching must happen before optimizations like inlining or devirtualization are applied:



Illustration 263. PGO profile matching during IR construction phase

The most detailed part of this process is matching call sites between the old profile and the new build. The system uses a line offset method to help it handle small changes in the code. To identify a call site, the matcher uses a composite ID. This ID includes the linker symbol name of the caller (like `main.main`), the linker symbol name of the callee (like `main.hot`), and the call site offset from the start of the function (for example, `43 - 29 = 14`).

```
29 func main() {
30     f, err := os.Create("cpu.prof")
31     if err != nil {
32         panic(err)
33     }
34     defer f.Close()
35
36     if err := pprof.StartCPUProfile(f); err != nil {
37         panic(err)
38     }
39     defer pprof.StopCPUProfile()
40
41     indirectHot()
42     for range 150 {
43         hot()
44         cold()
45     }
46 }
```

Illustration 264. Offset is distance from function's start line

The Go compiler looks at every function and its body in the current build and creates edges for direct calls it finds in the IR. For each call site, it figures out the relative line number (line offset) of the callee from the caller and checks the profile to find the weight for that call site. If it cannot find a matching call site in the profile, the compiler sets the weight to `0`.

Line numbers are part of the call site identifier. Because of this, changes in line numbers are a big practical problem:

- **Line number shifts:** Imagine a function with a call site at line offset 5 from the start. If someone adds or removes code before that point, the offset might move to 6 or 4. The profile still has the old offset 5, so the call site will not match. It will be seen as cold even if it used to be hot.
- **Stale hotness:** Suppose a call site was hot because it was inside a loop that ran many times. If the loop changes to run only once, the call site is cold now, but the profile may still mark it as hot if the line offset stays the same.

The matching process then creates a graph of call edges between functions and their call sites. However, if the call site is an indirect call, the Go compiler does not create any edges for it at this stage. This is because the compiler cannot know which concrete function is called at compile time.

Here is an example with a `Worker` interface:

```
type Worker interface {
    Do(int) int
}

func Process(worker Worker) {
    worker.Do(1)
}
```

The `Process` function takes a `Worker` interface as its argument. At compile time, the `Process` function does not know which concrete type `worker` will have. Another common example is using a function pointer:

```
func Process(do func(int) int) {
    do(1)
}
```

Again, the compiler does not know which function will be passed in at compile time. All of these indirect calls are handled in the second phase of PGO profile mapping.

The second phase fills in the gaps left by the first phase. It handles edges that cannot be found just by looking at the code. In this phase, the build system goes through all edges in the profile data and tries to map them to the current code. The CPU profile only contains direct, concrete calls. But at an indirect call site,

the profile tells the compiler which concrete functions were called during execution.

Let's look at an example with the `Process` function:

```
var developer = Developer{}
var tester = Tester{}

func main() {
    result := 0
    var worker Worker
    for i := range 1000 {
        if i%200 == 0 {
            worker = tester
        } else {
            worker = developer
        }

        result += worker.Do(i)
    }

    println(result)
}
```

If you run this function, about 0.5% of the time, `worker` is a `Tester`, and 99.5% of the time, it is a `Developer`. So the call site `worker.Do(i)` will have at most two concrete calls in the CPU profile data.

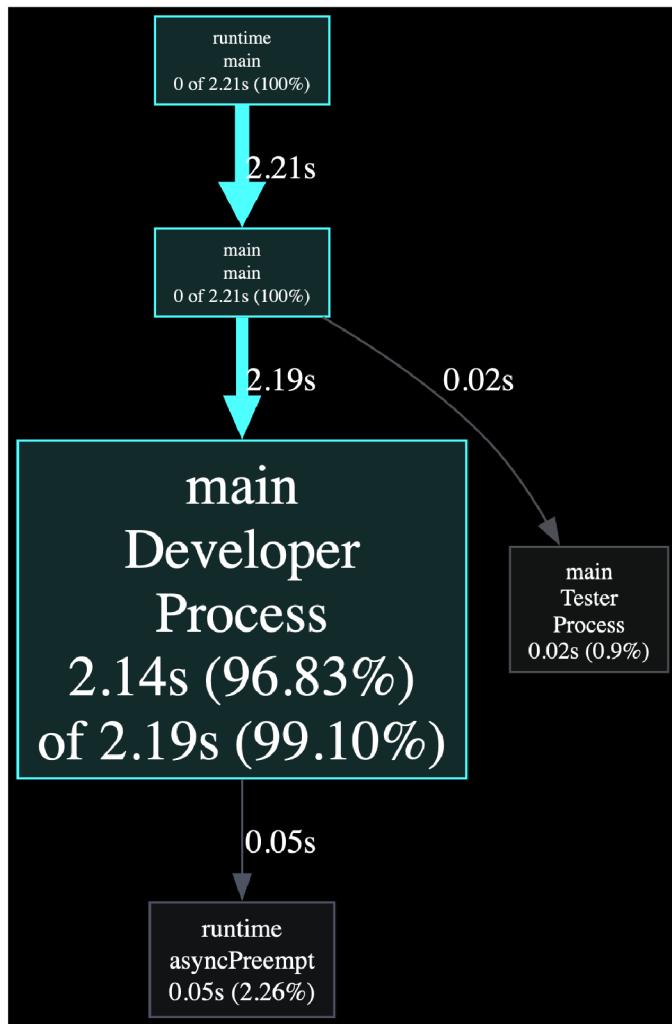


Illustration 265. Developer dominates Do call site performance profile

The Go compiler reads your PGO profile data from top to bottom, sorting the profile edges by weight from highest to lowest. It will see the `Developer.Do(i)` call as the heaviest edge at the `worker.Do(i)` call site and create an edge for it first. When it processes the `Tester.Do(i)` call at the same call site, it will also create a separate edge for it.

When the profiler sees two different concrete callees at the same indirect call site, it treats them as separate keys. This means both will appear in the call graph.

## Hot Callsite Policy & Inline Budget

To decide which call sites are hot, the Go compiler uses a cumulative distribution method. The idea is to select the most performance-critical calls—the ones that together account for 99 percent of all activity in your program.

Here is how it works with a concrete example. Imagine your program has these call sites with these weights:

```
call site A: 5000 weight
call site B: 1000 weight
call site C: 500 weight
call site D: 40 weight
call site E: 10 weight
Total: 6550 weight
```

The 99% threshold means we want the smallest set of call sites that add up to at least 99% of 6550, which is 6484.5. The preprocessed PGO data is already sorted from highest to lowest weight, so it goes through them in order A, B, C, D, E.

Here is the step-by-step:

- Step 1 (call site A): Running total is 5000. This is  $5000/6550 = 76.3\%$ . That is less than 99%, so keep going.
- Step 2 (call site B): Running total is 6000. That is  $6000/6550 = 91.6\%$ . Still less than 99%, so keep going.
- Step 3 (call site C): Running total is 6500. That is  $6500/6550 = 99.2\%$ . Now we pass the 99% mark, so we stop.

So call sites A, B, and C are "hot" and get special inlining treatment. Call sites D and E are "cold" and get normal treatment.

But what is that special treatment?

The biggest effect of PGO on inlining is the budget change. Normally, the inlining budget is 80 (or 160 with some context-aware inlining). For hot call sites, the budget goes up to 2000:

Figure 144. PGO Inline Hot Max Budget  
(src/cmd/compile/internal/inline/inline.go)

```
var inlineHotMaxBudget int32 = 2000
```

Let's look again at how inlining works. A hot call site can be inlined if it passes two checks:

- Inlinable function: The function itself must be inlinable. This means it does not contain anything that blocks inlining, such as `defer` statements, `go` statements, or a size that exceeds the budget. For hot functions, the budget is now 2000, so more functions will be inlinable.
- Inlinable call site: Each call site also has its own budget. Hot call sites have their budget increased to 2000 (unless the function is too large). A function that is inlinable with the 2000 hot budget could still have some call sites limited to the normal 80 if those calls are not hot.

That is why in the earlier example, the `hot` call site in `main.indirectHot` was not inlined even though the `hot` function itself was marked as hot.

## Devirtualization (guided by profiles)

Devirtualization is an optimization that happens when the compiler finds an indirect call and knows ahead of time which concrete function will be called at runtime. With normal static analysis, this is hard because the compiler cannot always tell which type will be used. The static logic is limited and usually cannot make the decision on its own.

What PGO adds is real evidence, gathered from profiling, about which concrete types actually show up at specific interface call sites during real execution. Instead of guessing, the compiler uses data from actual program runs. For example, the compiler might see that an interface method is almost always implemented by one concrete type.

Let's use the previous `Process` function example:

```
// func (Tester/Developer) Process(x int) int {
//     for i := range 10000000 {
//         x += i
//     }
//     return x
// }

var developer Developer
var tester Tester

func main() {
    result := 0
    var worker Worker
    for i := range 1000 {
        if i%100 == 0 {
            worker = tester
        }
    }
}
```

```

        } else {
            worker = developer
        }

        result += worker.Process(i)
    }

    println(result)
}

```

After running this program and collecting the profile with pprof, we get a preprocessed PGO profile like this:

```

GO PREPROFILE V1
main.main
main.Developer.Process
14 214
main.Developer.Process
runtime.asyncPreempt
1 5
main.main
main.Tester.Process
14 2

```

Here, the call to `Developer.Process` is much hotter than the call to `Tester.Process`. The most important transformation happens at `worker.Process(i)`. This is an interface call, so it could go to either `Developer.Process` or `Tester.Process` at runtime, depending on what `worker` is.

Normally, the compiler cannot inline interface calls because they could be any concrete type. Both `Developer.Process` and `Tester.Process` could be called at that point.

With PGO devirtualization, the compiler looks at the profile data and sees that `Developer.Process` is called much more often (weight 214) than `Tester.Process` (weight 2). Because almost every call goes to `Developer.Process`, the compiler can change the interface call into something like:

```

if dev, ok := worker.(Developer); ok {
    dev.Process(i) // Direct call - can be inlined now!
} else {
    worker.Process(i) // Fallback to interface call
}

```

This devirtualization makes a direct call to `dev.Process(i)`, which the inliner can now optimize. Two optimizations happen here: the interface dispatch overhead goes away, and the function call overhead also goes away for the common case (about 99%). The rare case, when `tester.Process(i)` is called, still works but will be a little slower.

What if, in a newer version of the code, someone removes the `Developer` type and only keeps the `Tester` type? This does not affect the compiler's decision at this stage. So, if the hottest callee (`Developer.Process`) is deleted in the new version:

- The profile still says `Developer.Process` is hottest.
- The compiler cannot find the symbol in the new code, so devirtualization is skipped.
- The rare `Tester.Process` edge is ignored because it was never the hottest in the profile.

## Summary

Go's runtime schedules goroutines using what is known as the MPG triad. An operating system thread (M) runs code only when it holds a logical processor token (P). A goroutine (G) executes on that thread. If the thread blocks because of a (blocked) syscall, it releases its P so another thread can pick it up and continue running other goroutines. Stack frames grow toward lower memory addresses. A stack pointer marks the top of the frame, while an argument pointer helps a callee find parameters that spill past the registers. The program counter provides the next instruction address, and on a call the return address is saved either on the stack for amd64 or in the link register for ARM64.

Functions in Go are represented as ordinary values called function values (`funcvals`). A `funcval` starts with a code pointer followed by any captured variables. Assigning a named function creates a pre-compiled `funcval` symbol in read-only memory:

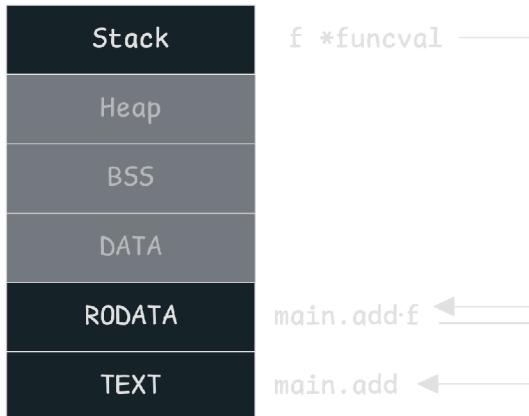


Illustration 266. RODATA holds function value symbol, TEXT holds code

When a closure captures state, the `funcval` is allocated at runtime. The compiler captures variables by value if their address is never taken, if they are not reassigned after capture, and if the size is under 128 bytes. Otherwise, the closure stores a pointer so later changes are visible. Function parameters always pass by value. For slices, maps, and channels the copied header still points to the original data, which means element mutations are visible but reassigning the header is not.

A `defer` statement records a function call on a LIFO chain tied to the current frame:

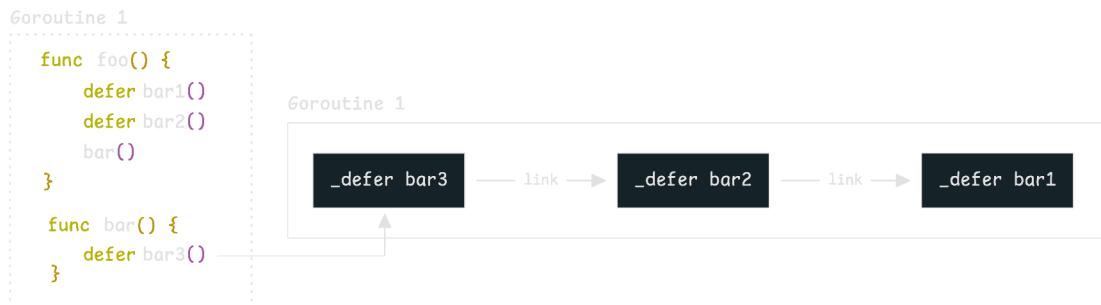


Illustration 267. All active defers linked in current goroutine

Go evaluates the arguments immediately but runs the deferred call right before the surrounding function returns. Named result parameters allow a deferred closure to modify the return value. For simple patterns the compiler emits direct calls and uses a bitmap instead of actual records. More complex or dynamic cases allocate `defer` records either on the stack or from a pooled heap allocator that cleans up when the function ends.

When panic is called, a `_panic` object is created and stack unwinding begins. Deferred calls are run in reverse order until one of them calls recover. If recover returns a non-nil value, the panic is cleared and normal return continues. If not, the runtime prints a stack trace and the program terminates. From Go 1.21 a panic with `nil` creates a `PanicNilError` so recover can tell the difference between a silent return and a panic with a nil argument.

Profiling is supported through the `runtime/pprof` package. CPU profiling samples the program counter at fixed intervals, while `heap` and `alloc` profiles sample memory allocations based on size. `mutex`, `block`, and `goroutine` profiles collect data on contention, blocking, and snapshots of goroutine states. Profiles can be enabled using `go test` flags, HTTP endpoints, or programmatic calls, and are written in protobuf format for analysis with `go tool pprof`. Trace recording captures scheduler, garbage collector, network, and syscall events. Each thread writes to its own buffer protected by a sequence lock, and timestamps use delta encoding to save space. The browser viewer shows global counters, per-P timelines, flow arrows connecting blocking and unblocking, a view of operating system threads, and an event breakdown page that counts all event types. Enabling `GODEBUG=traceallocfree=1` adds detailed information about memory spans, heap objects, and stack growth events for in-depth memory analysis.

Profile-guided optimisation (PGO) takes a sampled CPU profile and feeds it back into the compiler:

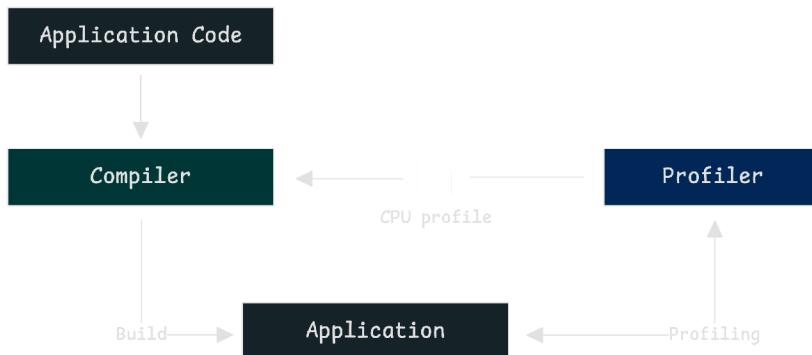


Illustration 268. Profile-guided optimization loop in Go compilation

A `default.pgo` file in the module root or a file passed with `-pgo` is processed by the preprofile tool into a compact map of caller and callee pairs with weighted call-site offsets. On the next build the compiler uses these weights to prefer inlining on hot call edges. Starting in Go 1.21 it can also devirtualise interface or function-value calls that mostly target one concrete callee. This often leads to

CPU time savings of several percent. Developers can repeat the cycle by collecting fresh profiles under real workloads, rebuilding with PGO, and redeploying for continuous improvement.

Knowing how MPG scheduling works, how calls use registers and stack frames, how `funcvals` are laid out, how `defer` and `panic` are implemented, and how profiling and tracing tools gather data gives a developer the ability to reason about performance, debug runtime behavior, and guide the compiler toward producing faster binaries.

## References

- [gcfsd] Graceful Shutdown in Go: Practical Patterns:  
<https://victoriametrics.com/blog/go-graceful-shutdown/>
- [threadcreate] runtime: threadcreate profile is broken:  
<https://github.com/golang/go/issues/6104>
- [g9986] cmd/compile: Enables PGO in Go and performs profile-guided inlining: <https://github.com/golang/go/commit/99862cd57d>
- More powerful Go execution traces - Michael Knyszek:  
<https://go.dev/blog/execution-traces-2024>
- Go Execution Tracer - Dmitry Vyukov:  
<https://docs.google.com/document/u/1/d/1FP5apqzBgr7ahCCgFO-yoVhk4YZrNIDNf9RybngBc14/pub>
- Profile-guided optimization: <https://go.dev/doc/pgo>