The Gink Programming Language

Syntax, Semantics, and Design Rationale

Jayatheerth Kulkarni

First Draft

Contents

Part I

Foundations

1

Introduction

1.1 Purpose of This Language

Most programming languages lean toward extremes. Pick any language and it usually optimizes heavily for one thing: ease of learning, safety, or performance. Over the years, people have tried to combine two of these, performance with safety, or ease with productivity, and have seen partial success. Gink is my attempt to find a meaningful sweet spot between all three.

Gink is not trying to be the absolute best at performance, safety, or ease of use individually. Instead, it aims to provide all three in deliberate proportions, where each complements the others rather than competing with them.

Online programming debates overwhelmingly revolve around runtime speed and safety guarantees, while developer experience is often treated as an afterthought. Ironically, in the real world, especially in industry, what matters most is speed of development, not raw execution speed. Companies want to ship products faster than ever, and modern programming paradigms are evolving to reflect that reality.

Keeping this in mind, I chose to dedicate my time in college to building Gink as my major project.

There is no shortage of opinions online advocating for "blazingly fast" languages, yet many of the same companies behind those opinions continue to rely on Python for large-scale production systems. This is not because Python is fast, it clearly is not, but because it enables teams to move fast.

I am not claiming Gink will be as slow as Python. Rather, the point is this: a programming language does not exist solely to be the fastest; it exists to help people build and ship software efficiently.

With that idea clearly and intentionally conveyed, I introduce Gink.

Enjoy the read.

1.2 Design Philosophy

Gink is a compiled programming language that translates source code directly into machine code. It is designed to be easy to understand and practical to use, with readability as a core goal.

The syntax deliberately avoids symbols that are not easily accessible on a standard keyboard. A developer typing with eight fingers at 40 WPM or higher should be able to write Gink code comfortably using only common keyboard keys. That said, Gink does make use of symbols where the keyboard naturally allows them. For example, logical operators are written as `||` and `&&` instead of the keywords `or` and `and`. A complete overview of the symbols used by Gink is provided in later chapters.

The overarching goal of the language is to feel intuitive, consistent, and predictable.

Gink is a statically typed language. All variables must have explicitly defined data types, such as `i32` or `i64`. Data types and their semantics are explored in detail later in this book.

The syntax of Gink is inspired by C and Rust, while making a conscious effort to significantly reduce overall language complexity. Although Gink is distinct from both languages, familiarity with C-style syntax helps illustrate this influence. For instance, every statement in Gink must end with a semicolon, which explicitly terminates a line in the source code.

Because intuition and understanding are central design principles, Gink intentionally enforces a single, standardized way of writing code. To support this, the compiler includes built-in code formatting and an integrated package manager.

The compiler is designed to protect the user and therefore includes multiple safety features and guardrails. However, Gink also acknowledges that experienced developers may need to work with externally compiled or low-level code that cannot be fully verified. For these cases, the language provides an `unsafe` block. Code inside an `unsafe` block is compiled as long as it is syntactically valid, even if potential memory safety issues exist.

Since Gink operates close to low-level concepts, the language does not attempt to model everything as an object. Instead, each construct is explicitly defined with clear and distinct semantics.

As a result, Gink is a highly predictable language. Throughout this book, you will notice that regardless of where or how the code is written, its structure and style remain consistent.

1.3 Problems This Language Solves

The problems this language is trying to solve are actually very simple. Gink aims to find the best possible sweet spot where a small amount of performance is consciously traded for strong safety guarantees and extreme ease of use.

At first glance, this may sound like I am steering the language toward garbage collection. I want to be very clear: I am not. Gink does not rely on garbage collection.

A major source of inspiration for building this language comes from Terry A. Davis. He was undeniably brilliant, and part of my motivation was to experience, even in a small way, what it feels like to build something ambitious and deeply personal from the ground up as a college project.

That said, this is not just a personal exercise. The problem Gink addresses is very real. If you look across the programming landscape, you will quickly notice that no language maximizes performance, safety, and ease of use simultaneously, or even comes particularly close to doing so.

Gink deliberately avoids garbage collection, aims for Rust-level safety, and keeps a syntax that is closer to C in spirit and familiarity. The moment I say this, an obvious objection arises: "Is C really easy?"

Here is a simple thought experiment. If there were a million dollars on the line and you had one month to learn a language before being asked to write a complex program, which language would you choose? For most people, the honest answer would be C. You might say Rust, and that is a fair answer. Even if the entire world picked Rust, I would still build Gink because I personally find Rust syntax intentionally difficult without sufficient payoff.

And that, ultimately, is the point. These are the reasons I am building a programming language: not to win online debates, not to chase extremes, but to explore a more balanced and human-oriented design space.

This is Gink.

2 Getting Started

2.1 Hello World

Let us begin with Gink's first program: the classic Hello World example. Although small, this section stands on its own because it clearly demonstrates the language's overall style and structure.

First, here is the complete program. After that, each line is explained in detail.

```
import std;

fun main() {
    std.println("Hello World");
}
```

Line-by-Line Explanation

`fun main()` Defines the main function. This is the program's entry point: execution always begins here.

`{` Marks the beginning of the function body.

`std.println("Hello World");` Calls the standard library's `println` function. It prints the text *Hello World* followed by a newline.

`}` Marks the end of the function body.

That's it. When the program runs, execution enters `main`, prints Hello World, and then exits.

2.2 Compilation Model

The compilation model of Gink is inspired by GCC and follows a four-stage pipeline that transforms Gink source code into executable machine code. This entire pipeline is driven by a single CLI tool, also called `gink`.

To run a Gink program without storing the final binary:

```
gink run main.gink
```

This compiles the program, executes it, and discards the binary afterward. To build a production-ready executable:

```
gink build main.gink
```

As the language evolves, additional CLI commands and flags are provided. These will be discussed in detail later.

2.2.1 The Four Compilation Stages

1. **Preprocessing**: In the preprocessing stage, the compiler takes the entire project directory and produces a single, expanded Gink source file augmented with internal compiler helper comments. During this stage:

   - All imported modules and libraries are recursively inlined.
   - Only functions and symbols that are actually required by the program are included.
   - Unused functions are completely excluded, resulting in a smaller final binary.

   This stage can be invoked explicitly to inspect the preprocessed output instead of continuing to binary generation:

   ```
   gink build -p main.gink
   ```

   You will be prompted to provide a file name, and the output will be `<file_name>.pgink`. Here, `p` stands for preprocessed.

   Design note: This behaves similarly to a mix of preprocessing and tree-shaking at the source level.

2. **Compilation**: This is the core compilation stage. Here, the compiler performs syntax validation, type checking, ownership and memory safety checks, and semantic analysis. Any compilation errors, whether due to incorrect memory usage, invalid references, or simple spelling mistakes, are reported at this stage. If compilation succeeds, the output is architecture-specific assembly code. To generate assembly directly:

   ```
   gink build -asm main.gink
   ```

After providing a file name, the compiler emits an assembly (`.asm`) file for the target architecture.

3. **Assembling**: The assembler converts the generated assembly code into an object file (`.o`). Currently, Gink uses the GCC toolchain for this step. A native assembler is not yet implemented, but may be developed in the future. To stop the pipeline at the object file stage:

   ```
   gink build -o main.gink
   ```

   You will be prompted for an output name, and a `.o` file will be produced.

4. **Linking**: In the final stage, the linker combines one or more object files into a single executable binary. This step resolves external library symbols, cross-file references, and the final memory layout. Once linking is complete, you obtain a standalone executable ready to run.

2.3 Source Files and Project Layout

The primary file extension for the language is .gink, but any UTF-8 or ASCII file can be compiled. There are no special extensions required by the compiler. If special extensions were mandatory, compiling any UTF-8 or ASCII file regardless of extension would not be possible.

The only exception to this rule involves packages. The package manager holds elevated privileges: it can create default directories, execute code, and run or build code for both development and production environments.

2.3.1 The reqter File

Packages utilize a special file with no extension. Its fixed name is `reqter` and it cannot be used for any other purpose. If the compiler detects more than one `reqter` file under the same package name, it throws a fatal error. This check is mandatory and cannot be bypassed, even within unsafe blocks.

2.3.2 Project Initialization

To streamline development, the Gink toolchain provides commands to create the default project structure:

- `gink init`: Creates a project within the current directory. This command will fail and exit if an existing `reqter` file is already present in the directory.
- `gink new <folder_name>`: Creates a new directory with the specified name and initializes a fresh project structure inside it.

2.3.3 Formatting and Encoding

The default naming convention for files and variables is snake_case. While there are no hard restrictions, this style is strongly recommended for readability.

In terms of file encoding, ISO-8859-1 files are allowed but discouraged because the same content can be represented in UTF-8. Some systems and editors add a Byte Order Mark (BOM) at the start of files: the Gink compiler ignores the BOM and starts processing at the main program.

Line endings are handled flexibly to ensure cross-platform compatibility. The characters \r, \n, and \r\n are all treated as new lines. This removes ambiguity and relies on well defined behavior.

2.3.4 Character Restrictions

Unicode characters cannot be used in identifiers. They are strictly allowed in strings and string literals, including emojis. Variable names must follow strict rules required for machine code generation, which are defined in detail in the Identifiers section of this manual.

2.4 Comments and Documentation

Comments in Gink are ignored by the compiler, along with whitespace. They exist purely for human readability and documentation purposes. Gink supports three categories of comments: single line comments, multi line comments, and documentation comments used to generate documentation.

2.4.1 Single Line Comments

Everything following `//` on the same line is ignored.

```
// This line is ignored by the compiler.
```

2.4.2 Single Line Documentation Comments

Any single line comment that starts with `///` is treated as a documentation comment and is included in generated documentation.

```
/// This line is ignored and included in documentation.
//// Still a documentation comment as long as it starts with ///
```

2.4.3 Multi Line Comments

Everything between `/*` and `*/` is ignored. Nothing inside a comment affects program behavior.

```
/*
This entire block
is ignored by the compiler.
*/
```

2.4.4 Multi Line Documentation Comments

Blocks starting with `/**` are treated as documentation comments and are included in generated documentation.

```
/**
This block is ignored by the compiler
but included by the documentation generator.
*/
```

2.4.5 Comment Behavior

Anything inside any comment, including text, numbers, symbols, or emojis, is completely ignored by the compiler within its range. Comments exist only for humans and tooling.

Warning on Nesting

Multi line comments cannot be nested. In the case below, the comment ends at the first `*/`. Any additional `/*` inside the comment is ignored as plain text.

```
/* outer comment
   /* inner comment */
*/
```

2.4.6 Documentation Generation

To generate documentation for a file or directory, run the following command:

```
gink run --docs <file_name>
```

This command extracts all documentation comments and generates readable documentation. Note that Gink documentation does not support Markdown or formatting syntax: documentation output consists of plain text only. More details on documentation structure and organization are covered in the Project Organization section.

Part II

Types and Values

3

The Type System

3.1 Strong and Static Typing

Understanding why Gink is statically typed is important.

The core issue with dynamic typing is that it incurs a significant performance and maintainability cost for relatively little gain in ease of use, especially once projects grow beyond a certain size. This is evident in how languages like Python and JavaScript evolved. Both started without strong typing, yet their ecosystems eventually reintroduced types through external systems: Python gained tools like pydantic, and JavaScript effectively gained an entire parallel language in TypeScript.

This section is not an attack on any language. There are good reasons Python exists in the form it does. For example, in scientific computing, strict typing is often secondary. What matters more is whether a number fits within a variable, whether it overflows, and how efficiently mathematical operations can be expressed. In such domains, developer friction must be minimal.

However, Gink is not designed primarily for quick and dirty prototyping. While prototyping is still possible with the right libraries, Gink is built for developers who want a balanced trade-off between performance, safety, and ease of use, not the extreme optimization of just one.

With that focus in mind, Gink is statically typed. Every variable must have an explicitly defined type. There is no notion of “untyped objects” floating through the program. Every value has a concrete type, and identifiers are declared using that type.

This decision is driven by two major reasons:

1. **Long-term scalability:** Static typing is essential for maintaining large codebases over time.

2. **Code readability:** Types carry semantic meaning. When reading unfamiliar or complex code, types act as documentation and reduce cognitive load.

As software grows, code is read far more often than it is written. In the age of AI-assisted development, this becomes even more pronounced. Humans may write less code manually, but they still need to understand, review, debug, and reason about it. Clear, explicit types make that process dramatically easier.

So while Gink may require more intention at the point of writing code, it pays that cost forward, optimizing for clarity, maintainability, and performance over the lifetime of a project.

That trade-off is intentional.

3.2 Type Annotations

Before moving forward, there are a few fundamentals you should understand as a user of Gink. These concepts will be explored in much greater detail later, but for now, here is the foundation.

There are two primary constructs where types matter the most in Gink: functions and variables.

3.2.1 Functions

Functions are the primary units of execution. A function may take inputs, return a value, or do both. Consider the following declarations:

```
fun main()
fun i32: add(i32: a, i32: b)
```

3.2.2 Variables

A variable stores a value. Because functions and variables are the core building blocks of any program, type annotations are required for both.

3.2.3 The Annotation Syntax

Gink follows a single, consistent annotation pattern across the entire language:

<type>: <name>

This rule never changes.

3.2.4 Annotating Function Return Types

To declare a function that always returns 67, you write:

```
fun i32: six_seven() {
    return 67;
}
```

Notice the `i32:` before the function name. This is the return type annotation of the function. It explicitly states that `six_seven` returns a 32-bit integer.

3.2.5 Annotating Variables

The same annotation syntax applies to variables:

```
i32: gink = six_seven();
```

Here, `gink` is an `i32` and its value is 67, returned from `six_seven`. There are no surprises and no hidden rules.

3.2.6 Why This Matters

Because the same annotation syntax is used everywhere, including functions, variables, and parameters, the language remains predictable, readable, and easy to reason about. Once you understand type annotations in one place, you understand them everywhere.

3.3 Type Inference

Gink does not support type inference, and this is a deliberate design choice.

As we move deeper into the age of AI, code will increasingly be read more than it is written. In that world, hiding types provides little value. What matters is clarity, intent, and unambiguous meaning. Type inference actively works against that.

Type inference manages to combine the worst aspects of both typed and untyped systems. It removes the single biggest advantage of explicit types, readability, by replacing meaningful declarations like `i32` with vague placeholders like `var`. At the same time, it keeps the downsides of typed systems by forcing developers to constantly reason about what type a variable actually is.

Consider this:

```
var: x = 67;
```

What is x? Is it an integer? If yes, is it signed or unsigned? Is it 8-bit, 32-bit, or 64-bit? And more importantly, how is the language supposed to know what the developer intended?

If a compiler could reliably infer a user's intent in all cases, we would have solved problems far more complex than programming language design. We clearly have not. The result is ambiguity, pushed onto both the compiler and the reader.

Gink chooses a simpler and more honest tradeoff: spend two extra seconds writing a type, and save hours of debugging and mental overhead later. Explicit types answer critical questions immediately:

- Why does this variable exist?
- What can it hold?
- What guarantees does it provide?

There is also a practical consistency problem. Once type inference is allowed, uniformity collapses.

```
var: x = 67;
```

vs

```
i32: x = 67;
```

Which one is correct? Who decides when to use `var` and when to use an explicit type? Worse, both styles inevitably end up mixed inside the same codebase, destroying visual consistency and making code harder, not easier, to read.

To avoid ambiguity, inconsistency, and unnecessary cognitive load, Gink removes type inference entirely. The language makes a clear statement: types are part of the program's meaning, not optional hints to be guessed later.

That clarity is intentional.

3.4 Compile-Time Guarantees

Once a program is compiled, there is an implicit contract between the developer and the generated machine code. In Gink, the compiler enforces a clear and explicit version of this contract.

If a program successfully compiles, it is guaranteed to be memory-safe by design. This guarantee is rooted in compile-time analysis, and runtime checks. In other words, memory safety is a property proven before the program ever runs.

That said, there is an important exception.

3.4.1 The unsafe Block

Gink provides an explicit escape hatch for cases where low-level control or performance is required: the `unsafe` block.

```
unsafe {
    // unchecked operations
}
```

An `unsafe` block is simply a keyword followed by a scoped block. Any code inside this block is exempt from the compiler's memory-safety checks.

Once `unsafe` is used, the responsibility shifts entirely to the programmer. The compiler makes no guarantees about memory safety for code within this block, nor for any violations that originate from it.

Outside of `unsafe`, however, the rule is strict and simple: If a Gink program compiles and does not use `unsafe`, it is memory-safe, assuming a correct compiler implementation.

This separation keeps safety the default, while still allowing controlled access to unchecked behavior when absolutely necessary.

4 Primitive Types

4.1 Integer Types

In this chapter, we explore the most fundamental building blocks of data: numeric types.

In a low-level environment, numbers are not a single unified concept. To a computer, the integer 5 and the decimal 5.0 are fundamentally different representations. Understanding numeric types means understanding how physical states (electricity) are mapped to logical values.

Every numeric type you define corresponds to a fixed reservation in memory. The size of that reservation, measured in bits, determines both the memory the value occupies and the range of values it can safely represent.

Integers represent whole numbers and are divided into two categories: signed and unsigned.

4.1.1 Signed Integers

Signed integers (denoted by the `i` prefix) can represent positive values, negative values, and zero. They use a representation called two's complement, where the most significant bit acts as the sign indicator. This allows efficient arithmetic while preserving a symmetric range around zero.

4.1.2 Unsigned Integers

Unsigned integers (denoted by the `u` prefix) represent only non-negative values (zero and above). Because no bit is reserved for sign information, unsigned integers can represent values roughly twice as large as their signed counterparts using the same number of bits.

4.2 Bit Widths and Capacity

The numeric suffix (8 through 128) specifies exactly how many bits are allocated for the integer.

Size	Signed Type	Unsigned Type	Range (Approximate)
8-bit	`i8`	`u8`	-128 to 127 / 0 to 255
16-bit	`i16`	`u16`	-32,768 to 32,767 / 0 to 65,535
32-bit	`i32`	`u32`	±2.1 billion
64-bit	`i64`	`u64`	±9 quintillion
128-bit	`i128`	`u128`	Scientific-scale integers
Arch	`int`	`uint`	Native CPU word size

Table 4.1: Gink Integer Type Hierarchy

4.2.1 Architecture-Dependent Integers

The `isize` and `usize` types scale with the target CPU architecture:

- On a 64-bit system, they are 64-bit integers.
- On a 32-bit system, they are 32-bit integers.

These types are mandatory for memory indexing, pointer arithmetic, and offsets, ensuring correctness and portability across architectures.

4.3 Floating-Point Types

Floating-point numbers represent values with fractional components (for example, 3.14159). Unlike integers, which are exact, floating-point values are approximations encoded using the IEEE 754 standard. This representation stores numbers as a significand (mantissa) and an exponent, enabling a wide range of magnitudes at the cost of precision.

Gink supports the following floating-point types:

4.4 Boolean Type

The Boolean data type represents logical values: true or false. The declaration syntax is:

```
bool: x = true;
```

Here, `bool` is the data type, and `true` or `false` is the value assigned to it.

A Boolean takes 1 byte of memory in RAM and is typically allocated on the stack.

If you come from a C programming background, you may be accustomed to treating `1` as true and `0` as false. This does not work the same way in Gink. You must explicitly specify

f16	**Half Precision:** A 16-bit floating-point type primarily used in machine-learning workloads and specialized GPU buffers, where reduced precision is acceptable in exchange for lower memory usage and higher bandwidth efficiency.
f32	**Single Precision:** A 32-bit floating-point type commonly used in graphics programming, game engines, and real-time simulations, offering a balance between performance and precision.
f64	**Double Precision:** A 64-bit floating-point type and the modern default for general-purpose numerical computation, providing high precision suitable for most mathematical and scientific tasks.
f128	**Quad Precision:** A 128-bit floating-point type intended for high-stakes scientific simulations and numerical analysis where rounding errors must be minimized.
float	**Architecture Precision:** A platform-dependent floating-point type that scales according to the native floating-point register width of the target architecture, allowing optimal use of available hardware precision.

whether a value is `true` or `false`.

As the name suggests, Boolean values support all commonly used logical operations.

There are multiple operators such as `OR`, `AND`, `XOR`, and `NOT` that can be used with the Boolean data type, and in some cases with numeric data types as well.

4.5 Character Type

A character in Gink occupies 4 bytes of memory. Unlike some languages where the size of a character can vary, Gink always uses exactly 4 bytes.

Each character represents a single Unicode character. Since Unicode characters can be up to 4 bytes in size, Gink's character type is designed to handle them directly.

This means you do not need to worry about multi-byte characters or partial characters. Each `char` always stores exactly one complete Unicode character.

```
char: x = 'a';
```

5 Variables and Initialization

5.1 Variable Declaration Syntax

To declare a variable in Gink, you must tell the compiler its identifier and its data type.

The general form is:

```
<type>: <name> = <value>;
```

For example, `i32` is a 32-bit integer type, so you can write:

```
i32: x = 88;
```

Here, `88` is the value stored in memory, and `x` owns that memory.

5.2 Initialization Rules

Gink is strict about initialization: a variable must either (1) own a concrete value, or (2) be explicitly marked as potentially absent (nullable).

Rule 1: Non-nullable variables must be initialized

A non-nullable variable must be created with a value. The following is invalid because `x` would contain an unknown (uninitialized) value:

```
i32: x;
```

Instead, initialize it immediately:

```
i32: x = 88;
```

Rule 2: Use `:?` for values that may be missing

If a variable is allowed to be absent, declare it with the `:?` modifier:

```
i32:? x;
```

`:?` does not introduce a `null` keyword; it represents an internal “no value” state tracked by the compiler.

Rule 3: Prove existence before using a nullable value

Before a nullable variable can be used as a normal value, you must check it inside a `check` block:

```
i32:? x;
std.scan(&x);

check {
  x;
}
```

Outside the `check` block, `x` is treated as non-nullable (as if it were declared with `:`). If no value exists within x then the program simply terminates.

But sometimes you might not want the program to terminate.

In this case, the program provides a function called `is_null`. It returns `true` if the value is internally null.

```
i32:? x;
std.scan(&x);

if(is_null(x) {
    // We handle x here
    // But we cannot access x
}

// To access x we still need the check block.
```

Multiple checks and loss of proof

A single `check` block can validate multiple nullable variables. However, the proof does not automatically carry through assignments from another nullable value. For example:

```
i32:? x;
std.scan(&x);

check {
  x;
}
// x is valid here

// You can now use x as if it is : and not :?
i32:? y;
x = y; // Illegal.
```

If x is owning y `x = y;`, y must be checked before unless the compiler can prove that y owns a value.

5.3 Assignment Semantics

Assignment in Gink is not just "copying a value". By default, assignment transfers ownership.

Ownership transfer with =

When you write:

```
i32: y = 10;
i32: x = y;
```

`x` becomes the owner of the value, and `y` is considered moved-from (it no longer owns the value). This rule applies regardless of whether the value is stack-allocated or backed by heap allocation.

If you want to use `y` again, you must give it a new value (i.e., make it own memory again):

```
i32: y = 10;
i32: x = y;
y = 10;
```

Keep it in your mind, x owns y but the scope of the variable x doesn't change.

For example if we do:

```
import std;

fun main() {
    i32: x = 10;
    {
        // Inner scope
        i32: y = 20;
        x = y;
        // y is useless here
    }
    // x is 20, even tho x owns y the scope of x is still to entire main block.
}
```

Copying with :=

Use := when you want to copy the data instead of transferring ownership:

```
i32: y = 10;
i32: x := y;
```

After this, both x and y remain valid and can be used normally.

Borrowing with pointers

To work with a value without taking ownership, you can borrow it through a pointer:

```
i32: x = 10;
*i32: y = &x;
```

Here, y does not own x; it borrows it.

Assigning between : and :?

A non-nullable variable (:) is "superior" to a nullable one (:?). Assigning a non-nullable value into a nullable variable is allowed:

```
i32: x = 10;
i32:? y;

y = x;
```

The other direction requires a proof of existence (a check):

```
i32:? x;
std.scan(&x);
```

```

check{
    x;
}

// Now x is superior too
```

Finally, note that you cannot "upgrade" a function's return type. If `std.scan` returns `i32:?`, a `i32:` variable cannot hold that result unless a `check` proves existence.

5.4 Reassignment and Mutability

Variables are mutable by default.

```
fun main() {
    i32: count = 0;
    count = count + 1; // Valid
}
```

To declare a value that cannot be changed, use the const keyword.

```
fun main() {
    const f64: PI = 3.14159;
    // PI = 3.14; // Compiler Error: Cannot assign to const variable
}
```

5.5 Scope of Variables

All variables are governed by their lexical scope (the block `{}` that contains them).

A variable is valid from its declaration until the end of the block it was declared in. When execution leaves that block, the variable is destroyed automatically.

This rule applies uniformly: whether a value lives on the stack or is backed by heap allocation, its lifetime is still bounded by scope, and cleanup happens deterministically at scope exit.

6 Arrays and Vectors

Arrays and vectors provide a structured way to work with collections of values, instead of declaring large numbers of individual variables.

6.1 Arrays

An **array** is a fixed-size, contiguous sequence of elements. Arrays are built into the language.

Declaration and Initialization

In Gink, arrays are **zero-initialized by default**. This design choice ensures that every element holds a valid, concrete value immediately upon creation. Consequently, there is no concept of a nullable array (:?); all arrays are guaranteed to be initialized.

The following declares an array of 10 integers, where every element is automatically set to `0`:

```
i32[10]: array;
```

Since the language guarantees initialization, you do not need to manually fill the array or perform existence checks on its elements.

Indexing and Bounds Checking

Array indexing in Gink is zero-based. For example, the 8th element is accessed with index `7`:

```
array[7];
```

Indexing is bounds-checked. If an index is out of range, the program terminates immediately to prevent memory safety violations.

6.2 Vectors

A **vector** is a dynamically sized array provided by the standard library (`std.vector`). Vectors can grow as elements are added.

Construction

A vector is created by simply specifying the element type.

```
std.vector(i32:): array;
```

Common Operations

Access an element:

```
i32: x = array.get(index);
```

If `index` is out of bounds, the program terminates.

Add an element:

```
array.add(32);
```

When the vector runs out of capacity, it grows (typically by allocating a larger backing buffer and moving elements).

Part III

Ownership and Memory

7

Ownership Model

7.1 What Ownership Means

Ownership is a fundamental concept that defines how data is managed in memory. At its core, ownership describes the relationship between a variable and the memory it controls. When a variable owns a piece of data, it is responsible for the lifetime of that data: its creation, access, and eventual release.

In Gink, ownership is explicit and literal.

```
i32: x = 10;
```

In this example, the integer value `10` is stored at a specific location in random access memory (RAM). The variable `x` owns that memory location. As long as `x` is valid, the memory remains allocated and accessible through `x`. When `x` goes out of scope, the memory it owns is automatically released.

Ownership is exclusive: at any given time, a block of memory has exactly one or zero owners. This rule prevents issues such as double frees, dangling pointers, and uncontrolled aliasing.

Ownership may be:

- **Transferred**, where control of the memory moves from one variable to another.
- **Borrowed**, where temporary access is granted through pointers without transferring ownership.

The following diagram illustrates how ownership maps a variable to a concrete region of RAM.

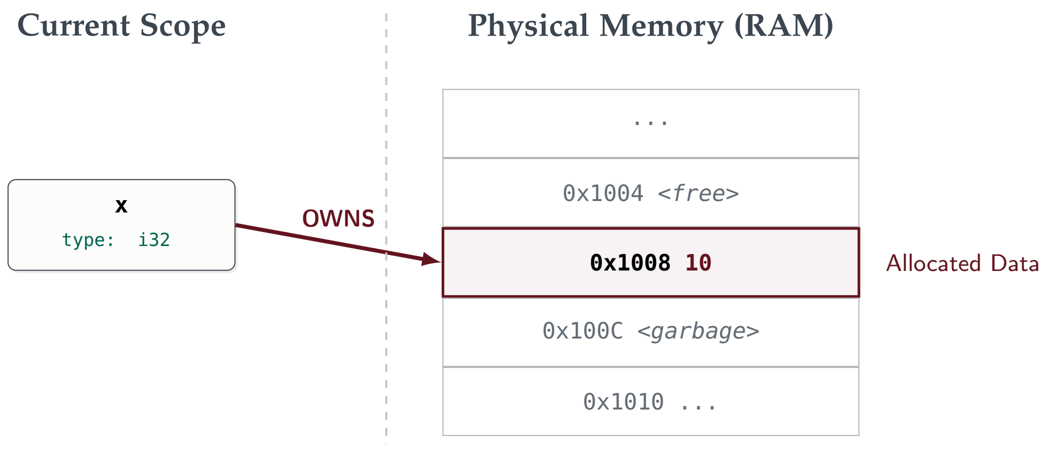

Figure 7.1: Visualizing Ownership: Variable `x` acts as the exclusive owner of the memory address `0x1008`.

7.2 Memory Layout and Ownership Mapping

In this diagram, the value `10` resides at memory address `0x1008`. The variable `x` owns this address, meaning it has exclusive rights to access and manage the data stored there. No other variable may claim ownership of this memory unless ownership is explicitly transferred.

This strict mapping between variables and memory is what enables Gink to provide strong safety guarantees while maintaining predictable and efficient memory management.

7.3 Single Ownership Rule

The most critical rule in Gink's memory model is the Single Ownership Rule: **at any given moment, exactly one variable owns a specific piece of memory.**

This rule ensures that the runtime always knows exactly who is responsible for freeing the memory. When the owner goes out of scope, the memory is freed. There is never any ambiguity about which variable triggers the cleanup.

7.3.1 Ownership vs. Pointing

While only one variable can *own* the data, multiple variables can *point* to it. This distinction is crucial.

- **Owner:** Controls the lifetime of the data.
- **Pointer:** Provides a view into the data for reading or writing.

Gink allows multiple pointers to reference the same memory address simultaneously. Because there is no `null` keyword in the language, these pointers are guaranteed to be valid as long as the owner is alive.

7.3.2 The Safety Constraint on Pointers

To maintain memory safety, Gink enforces a strict restriction on what can be pointed to: **a pointer may only reference a confirmed, non-nullable type (`:`).**

You cannot create a pointer to a nullable variable (`:?`). This prevents a class of errors where a pointer might try to access memory that does not exist.

If you have a nullable variable and wish to point to it, you must first prove its existence using a `check` block. Inside the block, the variable is treated as a confirmed type, allowing pointers to be created safely.

```
i32:? maybe_val;
std.scan(&maybe_val);

// *i32: ptr = &maybe_val; // Illegal: Cannot point to nullable

check {
    maybe_val; // proved to be safe.
}
// can point here

*i32: ptr = &maybe_val
```

7.4 Ownership Transfer

When you assign one variable to another, Gink does not copy the data by default. Instead, it performs a **move**. The ownership of the memory is transferred from the source to the destination.

```
i32: x = 10;
i32: y = x;  // Ownership moves from x to y
```

After this operation:

1. `y` is the new owner of the value `10`.
2. `x` is legally considered "empty" or "moved-from."

Any attempt to read from `x` immediately after this transfer will result in a compiler error.

7.4.1 Reviving a Variable

Unlike some languages where a moved variable is permanently dead, Gink allows you to **revive** a variable by assigning it a new value. The variable effectively becomes the owner of a new piece of memory.

```
i32: x = 10;
```

```
i32: y = x;

// x cannot be used here (it is empty)

x = 50;
// x is now alive again. It owns the value 50.
// y still owns the value 10.
```

This mechanism allows you to reuse variable names and memory slots efficiently without needing to redeclare them.

7.5 Ownership and Scope

Ownership is tightly bound to the scope in which a variable is declared. A scope is typically defined by a block of code enclosed in curly braces `{}`.

When a variable is declared, it owns its data for the duration of that scope. If the variable is moved to a new owner inside a function call or another variable, the original variable ceases to own the data. If it is never moved, it holds the data until the scope ends.

7.5.1 No Variable Shadowing

Gink enforces a strict "No Shadowing" rule similar to C.

If a variable named `x` exists in an outer scope, you cannot declare a new variable named `x` inside an inner scope (such as an `if` block or a loop). The inner scope always refers to the original `x` declared in the outer scope.

```
fun main() {
    i32: x = 10;

    if (true) {
        // i32: x = 20; // Error: Redeclaration of 'x'

        x = 20; // Valid: This modifies the outer 'x'
    }
}
```

This ensures that ownership is never ambiguous. When you see `x` in code, it always refers to the specific declaration visible at the top of the current function hierarchy.

7.6 Destruction and Resource Cleanup

Gink prioritizes simplicity and predictability in resource management.

7.6.1 Deterministic Destruction

Memory cleanup in Gink is deterministic. As soon as a variable goes out of scope (at the closing `}`), and provided it still owns its data, that memory is immediately released. There is no garbage collector running in the background, and there are no indeterminate pauses.

7.7 Destruction and Resource Cleanup

Gink takes a unique approach to resource management. It rejects both the hidden magic of C++ style RAII (where destructors run invisibly) and the manual error-prone cleanup of C (where you might forget to free a resource).

Instead, Gink enforces **Contract-Based Cleanup** using the `ensure` and `defer` keywords.

7.7.1 The `ensure` Contract

When a library author writes a function that allocates a resource (such as a file handle, a socket, or a lock), they can explicitly bind a cleanup requirement to that function using the `ensure` keyword.

This does not run the cleanup immediately. Instead, it creates a compile-time contract: any caller who invokes this function *must* schedule the specified cleanup action before the resource is used.

```
// The library author defines the allocator
fun i32: file(string: path) {
    // Internal logic to open file...
    i32: fd = 10;

    // Contract: The caller MUST defer a call to close()
    ensure(close());

    return fd;
}

fun void: close() {
    // Actual cleanup logic
    i32: a = 100;
}
```

Listing 7.1: Basic Pointer Usage

In the example above, the `ensure(close())` statement tells the compiler: "If anyone calls `file()`, they are legally required to defer `close()` immediately."

7.7.2 The `defer` Statement

The `defer` keyword allows the caller to schedule a block of code to execute deterministically when the current scope exits.

When a function tagged with `ensure` is called, the compiler checks the immediate surrounding code for a matching `defer` block. If the `defer` block is missing or does not contain the required cleanup call, the program will not compile.

```
import std;
1  fun main() {
2      // 1. We acquire the resource
3      i32: b = file("data.txt");
4
5      // 2. We MUST strictly defer the cleanup
6      defer {
7          close(); // Satisfies the ensure() contract
8      }
9
10     // 3. We can now safely use 'b'
11     std.println(b);
12
13 } // 'close()' is automatically executed here
```

Listing 7.2: Basic Pointer Usage

7.7.3 Compile-Time Verification

This system provides safety without runtime overhead.

- **Zero Cost:** The `ensure` statement in the library code is a compile-time marker. It is removed from the final binary and executes no machine code.
- **Safety:** It is impossible to forget to close a file or release a lock. The compiler will emit an error: *"Function 'file' requires ensure contract 'close', but it was not deferred."*
- **Clarity:** The cleanup logic is visible in the caller's code (inside `defer`), preserving the "Explicit is Better" philosophy of Gink.

8

Pointers and References

8.1 Pointer Types

Pointers in Gink are variables that store the memory address of another value. They act as the primary mechanism for borrowing data and referencing memory locations without transferring ownership.

8.1.1 Declaration and Syntax

A pointer type is declared by prefixing the target data type with an asterisk (∗). To obtain the address of a variable, use the ampersand (&) operator. To access the value stored at that address (dereference), use the asterisk (∗) operator on the pointer variable.

```
import std;

fun main() {
    // 1. Declare a standard integer
    i32: x = 10;

    // 2. Declare a pointer 'y' that holds the address of 'x'
    *i32: y = &x;

    // 3. Dereference 'y' to read the value of 'x'
    std.println(*y); // Prints 10
}
```

Listing 8.1: Basic Pointer Usage

8.2 Nullable Pointers

Gink allows the declaration of **Nullable Pointers** using the `:?` suffix on the type. This is essential for dynamic data structures like Linked Lists, where a pointer may explicitly point to "nothing" (e.g., the tail of a list).

8.2.1 Declaration

A nullable pointer (`*Type:?`) does not need to be initialized immediately. Its default internal state is `null`.

```
struct node {
    i32: data;
    *node:? next; // Can be null
}

fun main() {
    node:? n;
    *node:? p; // Default initialized to null

    // Valid assignment later
    p = &n;
}
```

Listing 8.2: Nullable Pointer Declaration

8.2.2 Pointing to Nullable Variables

You are permitted to take the address of a nullable variable, creating a nullable pointer. Because the underlying variable might not hold a value, you cannot dereference the pointer directly.

8.3 Pointer Validity and The Check Block

To ensure safety, you must verify the existence of a value behind a nullable pointer before accessing it. Gink provides the `check` block for this purpose.

8.3.1 The Check Assertion

The `check` block is an assertion mechanism. It takes a variable (or a pointer access) as input and asserts that it is not null.

- If the value exists, the check passes, and the compiler "promotes" the variable to a safe, non-nullable state for the current scope.
- If the value is null, the program terminates (or handles the failure, depending on configuration).

```
fun main() {
    i32:? input;
    std.scan(&input);

    *i32:? p = &input;

    // We cannot access *p yet.
    // std.println(*p); // Error: Unproven nullable access.

    // Prove it exists
    check {
        p; // Assertion
    }

    // Now 'p' is proven safe for this scope.
    std.println(*p);
}
```

Listing 8.3: Proof of Existence

8.4 Loop-Based Linking

Using nullable pointers and reassignment, you can construct dynamic structures like linked lists.

```
struct node {
    i32: data;
    *node:? next;
}

fun main() {
    // Head of the list
    node: head = { data = 10; }

    // 'temp' pointer tracks the tail.
    // It points to the 'head' initially.
    *node: temp = &head;

    // Staging variable for new nodes
    node:? nn;

    for(i32: i = 0; i < 5; i++) {
        // 1. Assign new data to 'nn'
        // Note the old value still exists, but nn doesn't own it.

        // the move operator takes the new struct.
        // And make sures nn owns this at nn's scope.
        nn = { data = i; }

        // 2. Link the current tail to the new node.
        (*temp).next = &nn;

        // 3. Advance the tail pointer.
        temp = &nn;
    }
}
```

Listing 8.4: Constructing a Linked List

8.5 Pointer Invalidation Rules

Gink tracks memory safety through ownership and movement. It is important to distinguish between **Moving** an owner and **Reassigning** an owner.

8.5.1 Invalidation on Move

If an owner variable is **Moved** (assigned to another variable), the original variable becomes invalid. Any pointers pointing to the original variable are immediately invalidated.

```
i32: x = 10;
*i32: p = &x;

i32: y = x; // MOVED. 'x' is now invalid.

// std.println(*p); // ERROR: 'p' points to 'x', which is invalid.
```

Listing 8.5: Move Invalidation

But something like this is funny.

```
fun main() {
    i32: x = 10;
    *i32: y = &x;

    x = 20;

    // y still points to the old 10 value.
    // x has been moved, keep that in mind.
}
```

Listing 8.6: Move Invalidation

I would say this is the only complex part about the entire language.

8.5.2 Validity on Reassignment

Reassigning a variable (giving it a new value) is a valid operation. Pointers tracking the variable simply see the new value

8.6 Ownership and Memory Persistence

8.6.1 The Move Rule (Section 7.4)

In the Gink language, when a variable's ownership is transferred, the original variable is marked as **invalid**. Any direct attempt to read from the moved-from variable will trigger a compiler error.

8.6.2 Memory Persistence and Pointers (Listing 8.6)

While the variable name itself becomes invalid after a move, the underlying memory segment may persist if an active pointer exists. This ensures memory safety without immediate deallocation of values that are still being referenced.

```
fun main() {
    i32: x = 10;
    i32: y = x;  // Ownership of the value '10' moves from x to y.
                 // x is now invalid.

    *i32: p = &y; // p points to the memory location currently held by y.

    y = 20;      // y is re-assigned to 20.
                 // Because the value '10' was not explicitly freed or moved
                 // again, p remains a valid pointer to the original value.
}
```

Listing 8.7: Example of Move Semantics and Pointer Persistence

Refined Logic: The contradiction is resolved by distinguishing between **variable validity** and **memory lifetime**. Moving from x invalidates the symbol x, but the value 10 remains live in the system as long as it is owned by another variable (y) or held by a pointer (p).

Part IV

Expressions and Control Flow

9

Expressions

9.1 Materiality of Expressions

In Gink, expressions are not just abstract calculations; they are material. Every expression, including a raw literal like `10` or an arithmetic operation like `a + b`, results in a concrete value residing at a specific memory location.

Variables in Gink are simply labels that take ownership of these existing memory locations.

9.2 Literals as Non-Addressable Values

9.2.1 R-Values and L-Values

In Gink, literals (such as `10`, `true`, or `'a'`) are strictly treated as "r-values". They represent a value to be used in computation, but they do not inherently occupy a fixed, addressable memory location (such as a stack slot) until they are assigned to a variable.

9.2.2 Address-of Restriction

Because a literal does not have a defined memory address or lifetime, you cannot take its address directly using the & operator.

To create a pointer to a literal value, you must first assign that value to a variable (an "l-value"). This step explicitly allocates memory for the value and defines its scope, ensuring that the pointer refers to valid storage.

```
fun main() {
    *i32: i = &(10);
    // This is invalid
}
```

Listing 9.1: Static Invalidation

This restriction prevents ambiguity regarding mutability and ensures that all pointers reference memory with a clear, deterministic lifetime.

9.3 Arithmetic and Ownership

The same logic applies to arithmetic expressions. An operation like `a + b` computes a result and places it in a temporary memory location.

When you assign this result to a variable, you are transferring ownership of that temporary result to the variable.

```
i32: a = 10;
i32: b = 20;

// (a + b) creates a value at a temporary location.
// 'result' takes ownership of that location.
i32: result = a + b;
```

Listing 9.2: Arithmetic Ownership

9.4 Assignment Semantics

Gink provides two distinct operators for assignment, allowing you to choose between moving ownership or copying data.

9.4.1 Move Semantics (=)

The standard assignment operator `=` transfers ownership. The variable on the left becomes the new owner of the memory location defined on the right.

```
// '10' is a location. 'x' owns it.
i32: x = 10;
```

Listing 9.3: Move Semantics

9.4.2 Copy Semantics (:=)

The copy assignment operator `:=` creates a *new* memory location and copies the *value* from the right-hand side into it.

```
// '10' is a location.
// 'x' allocates NEW memory and copies the value 10 into it.
```

```
i32: x := 10;
```

Listing 9.4: Copy Semantics

9.5 Return Values

While expressions produce addressable values, Gink is **statement-oriented** regarding control flow. Blocks, `if` statements, and loops do not implicitly return values.

To return a value from a function, you must use the explicit `return` keyword.

```
fun i32: add(i32: a, i32: b) {
    // a + b; // Error: Computed but discarded
    return a + b; // Valid
}
```

Listing 9.5: Explicit Return

9.6 Expression-Oriented vs. Statement-Oriented

Gink adopts a hybrid design philosophy that distinguishes clearly between calculating values and controlling program flow.

9.6.1 The Statement-Oriented Boundary

Unlike some modern languages where "everything is an expression" (allowing constructs like `let x = if condition {...}`), Gink remains strictly **Statement-Oriented** for control flow.

Control flow structures such as `if`, `while`, and code blocks `{ }` are statements, not expressions. They perform actions but do not evaluate to a data value. This design choice enforces clarity: a block of code exists to execute commands, not to implicitly return a result.

```
// INVALID in Gink (Expression-Oriented style)
// i32: x = if (valid) { 10 } else { 20 };

// VALID in Gink (Statement-Oriented style)
i32: x = 0;
if (valid) {
    x = 10;
} else {
    x = 20;
}
```

Listing 9.6: Statements vs Expressions

9.6.2 Materiality of Expressions

While control flow remains statement-based, Gink elevates data expressions to a higher status than C. As detailed in previous sections, any construct that produces a value (literals, arithmetic, function calls) produces a **material memory location** that can be addressed directly.

This creates a consistent mental model:

- **Expressions** produce addressable data (Values).
- **Statements** manipulate that data or alter execution flow (Actions).

9.7 Arithmetic Expressions

Arithmetic operations in Gink do not merely calculate transient values in a CPU register; they instantiate the result in a temporary, addressable memory location.

This behavior unifies arithmetic with the language's ownership model, allowing results to be owned or pointed to immediately upon evaluation.

9.7.1 Result Ownership vs. Copying

When the result of an arithmetic expression is assigned to a variable, the behavior depends on the assignment operator used:

- **Ownership Transfer (=):** The variable takes ownership of the temporary memory location created by the expression. No data copying occurs; the variable simply becomes the name for that memory slot.
- **Value Copy (:=):** The variable allocates distinct, new memory and copies the calculated value into it.

```
i32: a = 10;
i32: b = 20;

// 1. Efficient Ownership
// 'a + b' creates a temp slot with value 30.
// 'sum' takes ownership of that slot directly.
i32: sum = a + b;

// 2. Explicit Copy
// 'diff' allocates distinct memory.
// The result of 'a - b' is copied into 'diff'.
i32: diff := a - b;
```

Listing 9.7: Arithmetic Ownership

9.7.2 Pointing to Arithmetic Results

Because arithmetic expressions result in a literal again, you cannot take their address directly.

```
i32: x = 5;

*i32: p = &(x + 5);

// This is invalid too
// Even if the temporary memory is located
// There is owner to the new value x+5 so the pointer doesn't make sense
```

Listing 9.8: Pointers to Expressions

9.8 Logical Expressions

Logical expressions in Gink evaluate to a `bool` value: either `true` or `false`.

Just like arithmetic expressions, logical operations instantiate their result in a temporary, addressable memory location. This means the result of a comparison or logical check is a material value that can be owned or pointed to.

9.8.1 Short-Circuit Evaluation

Logical AND (&&) and OR (||) operators employ short-circuit evaluation.

- **AND (&&):** If the left operand is `false`, the right operand is not evaluated.
- **OR (||):** If the left operand is `true`, the right operand is not evaluated.

9.8.2 Operator Precedence

Gink follows the standard C operator precedence and associativity rules. This ensures that expressions behave exactly as a systems programmer would expect.

The following table lists all operators from highest precedence (evaluated first) to lowest precedence.

Prec	**Operator**	**Description**	**Associativity**
1	`() [] .`	Function call, Array subscript, Member access	Left-to-Right
2	`! ~`	Logical NOT, Bitwise NOT	Right-to-Left
	`+ -`	Unary Plus, Unary Minus	Right-to-Left
	`* &`	Dereference, Address-of	Right-to-Left
	`(type) sizeof`	Cast, Size of	Right-to-Left
3	`* / %`	Multiplication, Division, Modulo	Left-to-Right
4	`+ -`	Addition, Subtraction	Left-to-Right
5	« »	Bitwise Left Shift, Right Shift	Left-to-Right
6	< <=	Relational Less than, Less than or equal	Left-to-Right
	> >=	Relational Greater than, Greater than or equal	Left-to-Right
7	`== !=`	Equality, Inequality	Left-to-Right
8	&	Bitwise AND	Left-to-Right
9	ˆ	Bitwise XOR	Left-to-Right
10	|	Bitwise OR	Left-to-Right
11	&&	Logical AND	Left-to-Right
12	`||`	Logical OR	Left-to-Right
13	`?:`	Ternary Conditional	Right-to-Left
14	`= :=`	Assignment, Copy Assignment	Right-to-Left
	`+= -=` ...	Compound Assignment	Right-to-Left
15	,	Comma	Left-to-Right

Table 9.1: Gink Operator Precedence (Standard C Model)

10 Control Flow

10.1 Assignment as an Expression

In Gink, assignment is an expression. It evaluates to the variable that was just assigned (the left-hand side).

However, because the standard assignment operator (=) transfers ownership, chaining it is destructive. If you were to write `a = b = 10`, the value would move into `b`, and then immediately move into `a`, leaving `b` empty (moved-from).

10.1.1 Chaining with Copy (:=)

To initialize multiple variables to the same value, you must use the **Copy Assignment** operator (`:=`). This ensures that the value is replicated at each step of the chain, rather than moved.

```
i32: a = 0;
i32: b = 0;
i32: c = 0;

// Correct: Using := creates copies down the chain.
// 1. c := 10 (c gets a copy of 10)
// 2. b := c  (b gets a copy of c's value)
// 3. a := b  (a gets a copy of b's value)
a := b := c := 10;

// Result: a, b, and c all hold the value 10.
```

Listing 10.1: Correct Assignment Chaining

10.1.2 Embedded Assignments

One important note that you have to keep in mind is this:

```
fun main() {
    i32: x = 0; // x is a variable in the scope of main

    // Say I do this

    if((x = 10) == 10) {
        // x is 10 here
    }
    // x is 10 here too
    // Cause x now owns 10 in the if
    // But as we cleared x's scope remains the same.
}
```

Listing 10.2: Embedded Assignment

10.2 Evaluation Order

To ensure deterministic behavior, Gink enforces a strict evaluation order for all expressions.

10.2.1 Left-to-Right Evaluation

In any expression involving multiple operands or function calls, the compiler guarantees that sub-expressions are evaluated strictly from Left to Right.

```
// Gink guarantees the following order:
// 1. func_a() is executed.
// 2. func_b() is executed.
// 3. func_c() is executed.
process(func_a(), func_b(), func_c());
```

Listing 10.3: Deterministic Evaluation

10.2.2 Associativity vs. Evaluation

While evaluation of terms happens left-to-right, the associativity of operators determines how they group. Assignment is Right-Associative.

In the expression `a := b := c`, the compiler:

1. Evaluates the terms `a`, `b`, and `c` (finding their addresses).
2. Groups the operations from the right: `a := (b := c)`.
3. Executes the assignments from the inside out (right to left).

10.3 Conditional Statements

Conditional statements allow the program to execute different code paths based on the evaluation of a boolean expression.

10.3.1 The if Statement

The `if` statement evaluates a condition enclosed in parentheses. If the condition evaluates to `true`, the subsequent statement or block is executed.

Gink follows C-style syntax rules for conditionals:

- Parentheses around the condition are mandatory.
- Braces `{ }` are optional if the body consists of a single statement, though using them is recommended for consistency and safety.

```
i32: x = 10;

if (x > 5) {
    std.println("x is greater than 5");
}

// Single statement without braces (C-style)
if (x == 10) std.println("x is 10");
```

Listing 10.4: Basic if Statement

10.3.2 Else and Else If

To handle alternative cases, use the `else` keyword. You can chain multiple conditions using `else if`.

```
if (x > 10) {
    std.println("Greater than 10");
} else if (x < 10) {
    std.println("Less than 10");
} else {
    std.println("Equal to 10");
}
```

Listing 10.5: Else If Chain

10.3.3 Scoped Declarations in Conditions

Gink allows you to declare and initialize a variable strictly within the scope of the conditional block.

Because a declaration assignment (e.g., `i32: x = 0`) evaluates to the assigned value, it can be embedded directly inside the condition. This is particularly useful for limiting the scope of a temporary variable to the block where it is needed.

```
fun main() {
    // 'x' is declared, initialized to 0, and compared to 0.
    // If the result is true, the block executes.
    if ((i32: x = 0) == 0) {
        // 'x' is valid here.
        // You can modify, move, or read 'x'.
        x = x + 1;
    }

    // 'x' is NOT valid here. It has gone out of scope.
}
```

Listing 10.6: Scoped Declaration in Condition

This pattern is cleaner than declaring the variable outside the `if` statement, as it prevents the variable from leaking into the surrounding scope.

10.4 Loops

Loops allow for the repeated execution of a block of code. Gink supports two primary loop constructs: `while` loops and `for` loops.

Consistent with the language's design, loops are statements, not expressions. They do not evaluate to a value.

10.4.1 While Loops

The `while` loop repeatedly executes a block of code as long as a specified condition remains `true`. The condition is evaluated before every iteration.

```
fun main() {
    i32: i = 0;
    while(i < 5) {
        i = i+1;
        // i := i+1 works too
        std.println(i);
    }
}
```

Listing 10.7: While Loop

Infinite Loops

To create an infinite loop, explicitly use the boolean literal `true` as the condition.

```
while (true) {
    // Runs forever
}
for(;;) {
    // Runs forever as well
}
```

Listing 10.8: Infinite Loop

10.4.2 For Loops

The `for` loop provides a compact syntax for iterating over a range of values. It follows the standard C structure with three components: initialization, condition, and update.

```
// for (initialization; condition; update)
for (i32: i = 0; i < 10; i += 1) {
    std.println(i);
}
```

Listing 10.9: For Loop Structure

Scope of Loop Variables

Variables declared in the initialization statement (like `i` in the example above) are scoped strictly to the loop block. They are created when the loop begins and destroyed when the loop terminates.

```
for (i32: i = 0; i < 5; i += 1) {
    // 'i' is valid here
}
// 'i' is not valid here
```

Listing 10.10: Loop Scope

10.5 Break and Continue

Gink provides standard mechanisms to alter the flow of a loop: `break` to terminate the loop entirely, and `continue` to skip the current iteration.

10.5.1 Basic Usage

Used without labels, `break` and `continue` apply to the innermost loop currently being executed.

```
i32: i = 0;
```

```
while (true) {
    if (i == 5) {
        break; // Exits the loop immediately
    }
    i += 1;
}
```

Listing 10.11: Basic Break

10.5.2 Labeled Loops

To break out of nested loops, Gink allows you to label a loop directly in its declaration. The syntax appends the label name to the loop keyword, separated by a colon.

Syntax: `while:label_name (condition)` or `for:label_name (...)`

```
fun main() {
    i32: i = 0;

    // 'outer' is the label for this while loop
    while:outer (i < 5) {
        i32: j = 0;
        while (j < 5) {
            if (i * j > 10) {
                // Breaks out of the 'outer' loop, terminating both
                break outer;
            }
            j += 1;
        }
        i += 1;
    }
}
```

Listing 10.12: Labeled Break

10.5.3 Continue with Labels

The `continue` statement also supports labels, allowing you to skip the remainder of a nested inner loop and proceed directly to the next iteration of a specific outer loop.

```
for:main_loop (i32: x = 0; x < 3; x += 1) {
    for (i32: y = 0; y < 3; y += 1) {
        if (x == 1 && y == 1) {
            // Skips to the next iteration of 'main_loop'
            continue main_loop;
        }
        std.println(x + y);
    }
}
```

Listing 10.13: Labeled Continue

10.6 Early Returns

The `return` keyword allows a function to cease execution immediately and pass control back to the caller. This is frequently used for "guard clauses" to handle invalid states early, avoiding deep nesting of `if-else` blocks.

10.6.1 Returning Values

In functions declared with a specific return type, the `return` statement is mandatory on all code paths and must be followed by an expression matching the declared type.

```
fun i32: max(i32: a, i32: b) {
    // Early return if 'a' is larger
    if (a > b) {
        return a;
    }

    // Fallback return
    return b;
}
```

Listing 10.14: Early Return with Value

10.6.2 Returning from Void Functions

In functions with no return type (void), the `return` statement can be used without a value to exit the function execution early.

While a `return` is not required at the end of a void function (it returns implicitly), it is the only way to stop execution in the middle of a block.

```
fun process_positive(i32: x) {
    // Guard clause: Exit immediately if input is invalid
    if (x < 0) {
        return;
    }

    // This code only runs if x >= 0
    std.println(x);
}
```

Listing 10.15: Early Return in Void Function

Part V

Functions and Abstraction

11

Functions

11.1 Function Declarations

Functions are the building blocks of Gink programs. They are defined using the `fun` keyword. Function declarations are order-independent, meaning a function can be called before it is defined in the source file.

11.1.1 Basic Declaration

A function signature consists of the return type(s), the function name, and the parameter list.

```
// Returns an i32
fun i32: add(i32: a, i32: b) {
    return a + b;
}

// Returns nothing (void)
fun log_message(string: msg) {
    std.println(msg);
}
```

Listing 11.1: Basic Function

11.2 Parameters

Parameters are declared as a comma-separated list of `type: name` pairs. Gink follows C-style rules for parameters: there are no named arguments at the call site, and no default parameter values.

11.2.1 Variadic Parameters

Gink supports variadic functions, allowing a function to accept a variable number of arguments of a specific type. This is denoted by appending ... to the parameter name.

Inside the function, the variadic parameter is treated as an array/collection that can be iterated over.

```
// Accepts zero or more string arguments
fun print_all(string: messages...) {
    // Iterate over the variadic arguments
    for (string: msg in messages) {
        std.println(msg);
    }
}

fun main() {
    print_all("Hello", "World", "Gink");
}
```

Listing 11.2: Variadic Function

11.2.2 The For-Each Loop

As seen in the variadic example, Gink provides a specialized loop syntax for iterating over arrays and collections. This `for..in` loop simplifies retrieving data without managing indices.

Syntax: `for (type:  item in collection) { ...  }`

11.3 Return Values

Gink supports returning values from functions using the `return` keyword.

11.3.1 Multiple Return Values

Unlike C, Gink treats multiple return values as a first-class feature. This is particularly useful for returning a result alongside an error or success flag, a pattern common in Go.

To declare multiple return values, list the types separated by commas before the function name.

```
// Declares that the function returns an i32 AND a bool
fun i32, bool: find_index(i32: target) {
    if (target == 200) {
        // Return both values
        return 200, true;
    }

    return -1, false;
}
```

Listing 11.3: Multiple Return Values

11.4 Function Overloading

Gink enforces a strict rule regarding function names: Every function must have a unique name. Unlike C++ or Java, you cannot define multiple functions with the same name but different parameter lists. This decision prioritizes clarity and prevents ambiguity during compilation.

```
// Valid
fun i32: add_i32(i32: a, i32: b) { ... }

// INVALID: Name collision, even if arguments differ
fun f32: add_i32(f32: a, f32: b) { ... }
```

Listing 11.4: Overloading is Invalid

11.5 Type Passing (Generics)

To compensate for the lack of overloading and to support generic programming, Gink allows Types to be passed as function arguments.

This feature allows you to write a single function that operates on any type, provided the operations inside (like +) are valid for that type.

11.5.1 Syntax

The syntax uses the `type` keyword in a special manner:

- **Return Type:** `type::` (double colon) indicates that the return type is determined by the type argument passed to the function.
- **Parameter List:** The first parameter is declared as `type:`, acting as a placeholder for the type being passed.
- **Variables:** Subsequent parameters can use `type:` to declare that they are of that passed type.

```
// 'type::' - Return type depends on the passed type
// The double colon looks at the parameters and checks if there are any types
    passes with name type:
// 'type:'  - First argument is the type itself
// 'type: a, b' - 'a' and 'b' are variables of that passed type
fun type:: add(type:, type: a, type: b) {
    return a + b;
}

fun main() {
    // Pass 'i32' as the type argument
    i32: sum_int = add(i32:, 10, 20);

    // Pass 'f32' as the type argument
    f32: sum_float = add(f32:, 10.5, 20.5);

    std.println(sum_int);   // Prints 30
    std.println(sum_float); // Prints 31.0
}
```

Listing 11.5: Type Passing

11.6 Recursion

Gink supports standard recursion, allowing functions to call themselves.

As with C, recursion in Gink relies on the call stack. Each function call pushes a new frame onto the stack. While powerful, deep recursion without base cases can lead to a Stack Overflow.

```
fun i32: factorial(i32: n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Listing 11.6: Recursive Function

11.7 Inlining

To optimize performance for small, frequently called functions, Gink supports the `inline` keyword. This serves as a hint to the compiler to replace the function call directly with the function body, eliminating the overhead of a function call.

```
inline fun i32: max(i32: a, i32: b) {
    return (a > b) ? a : b;
}
```

Listing 11.7: Inline Function

11.8 Ownership in Function Calls

Gink imposes strict ownership rules on function arguments. Unlike C, where arguments are copied by value, or Java, where references are shared, Gink uses Move Semantics by default for all variables.

11.8.1 The Move Rule

When a variable is passed to a function as an argument, its ownership is transferred (moved) to the function's parameter.

The original variable in the caller becomes **uninitialized** (moved-from) and cannot be used again in the current scope. This ensures that every piece of data has exactly one active owner, preventing double-free errors.

```
fun take_ownership(i32: val) {
    // 'val' now owns the memory.
    // It will be freed when this function returns.
}

fun main() {
    i32: x = 10;

    // Ownership of 'x' moves to 'take_ownership'.
    take_ownership(x);

    // COMPILER ERROR: 'x' is invalid here because it was moved.
    // std.println(x);
}
```

Listing 11.8: Argument Ownership Transfer

11.8.2 Borrowing (Pass by Reference)

If a function needs to read or modify a value without taking ownership of it, you must use pointers. This is known as "borrowing."

The function declares a pointer parameter (e.g., `*i32`), and the caller passes the address of the variable (e.g., &x).

```
// Accepts a pointer (borrow)
fun peek(const *i32: ptr) {
    // Read the value at the address
    std.println(*ptr);
}

fun main() {
    i32: x = 10;

    // Pass the address. Ownership stays with 'main'.
    peek(&x);

    // 'x' is still valid here.
    std.println(x);
}
```

Listing 11.9: Borrowing via Pointers

11.8.3 Explicit Copying

If you need to pass ownership to a function but also retain the original variable for later use, you must create an explicit copy at the call site.

Using the copy assignment operator (`:=`), you can create a temporary copy that is moved into the function, leaving the original variable intact.

```
fun i32: add(i32: a, b) {
    return a + b;
}


fun main() {
    i32: a = 10;
    i32: b = 10;

    std.println(add(:a, :b));
}

: before a and b refers to copy
```

Listing 11.10: Passing a Copy

11.9 Borrowed Parameters

While the **Copy Prefix** (`:`) allows you to duplicate values to retain ownership, it can be inefficient for large data structures. Additionally, copies do not allow a function to modify the original variable.

To solve this, Gink uses **Borrowing**. A borrowed parameter accepts a pointer to the data rather than the data itself. The caller retains ownership, and the function receives temporary access via an address.

11.9.1 Syntax

Borrowing is implemented using standard pointer syntax:

- **Parameter:** Declare the parameter as a pointer type (e.g., `*i32: ptr`).

- **Argument:** Pass the address of the variable using the address-of operator (e.g., `&variable`).

11.9.2 Mutable Borrows

If a function needs to modify the caller's variable, use a standard pointer.

```
// Accepts two pointers to i32
fun swap(*i32: a, *i32: b) {
    // Dereference to access and swap values
    i32: temp := *a;
    *a := *b;
    *b := temp;
}

fun main() {
    i32: x = 10;
    i32: y = 20;

    // Pass addresses to borrow 'x' and 'y'
    swap(&x, &y);

    // x is now 20, y is now 10
}
```

Listing 11.11: Mutable Borrow (Swap)

11.9.3 Immutable Borrows (Read-Only)

To prevent a function from modifying the borrowed data, use the `const` keyword. This enforces read-only access at the compiler level.

```
// Accepts a read-only pointer
fun print_value(const *i32: ptr) {
    // Reading is allowed
    std.println(*ptr);

    // COMPILER ERROR: Cannot assign to const pointer
    // *ptr = 100;
}

fun main() {
    i32: data = 42;
    print_value(&data);
}
```

Listing 11.12: Immutable Borrow

11.10 Inlining

Function calls incur a small amount of overhead due to stack frame creation and context switching. For small, frequently executed functions, this overhead can be significant.

Gink provides the `inline` keyword to suggest that the compiler replace the function call directly with the function's body code.

11.10.1 Usage

To inline a function, prefix the definition with the `inline` keyword. Note that this is a hint; the compiler may choose to ignore it if inlining would negatively impact performance (e.g., by causing excessive code bloat).

```
// Suggests inlining for performance
inline fun i32: max(i32: a, i32: b) {
    return (a > b) ? a : b;
}

fun main() {
    // The compiler will likely replace this call with:
    // i32: m = (10 > 20) ? 10 : 20;
    i32: m = max(10, 20);
}
```

Listing 11.13: Inline Function

11.11 Recursion

Gink supports standard recursion, allowing functions to call themselves.

Recursion is implemented using the call stack. Each recursive step pushes a new stack frame. While powerful for algorithms like tree traversals, deep recursion without a base case will lead to a Stack Overflow.

```
fun i32: factorial(i32: n) {
    // Base case
    if (n <= 1) {
        return 1;
    }
    // Recursive step
    return n * factorial(n - 1);
}
```

Listing 11.14: Recursive Function

11.11.1 Tail Recursion

While Gink supports recursion, it follows standard C-style compilation. Guaranteed Tail Call Optimization (TCO) is not strictly enforced by the language specification, though the underlying optimizer may apply it in simple cases. Developers should prefer iterative loops (`while` / `for`) for unbound recursion depth to ensure safety.

12

Error Handling

12.1 Error Handling Philosophy

Gink deliberately omits dedicated syntax for error handling. There are no `try`, `catch`, `throw`, or `raise` keywords in the language.

Most programming languages treat errors as "exceptional" events that require a separate, parallel control flow mechanism. This introduces hidden complexity: a function call might return a value, or it might silently jump to a catch block three stack frames up. This violates Gink's core design goal of explicit and predictable control flow.

12.1.1 Errors are Values

In Gink, an error is not a special state; it is simply a value, identical in nature to an integer, a boolean, or a string.

Because errors are treated as standard values, they are handled using standard control flow tools: `if` statements and early `return`s. This removes the need for developers to learn a separate sub-language just to handle failure states.

12.2 Using Multiple Returns

Since Gink supports multiple return values as a first-class feature, the idiomatic way to handle errors is to return the result alongside a status indicator.

```
// Returns the value AND a success boolean
fun i32, bool: parse_positive(i32: input) {
    if (input < 0) {
        // Failure case: return dummy value and false
        return 0, false;
    }
    // Success case: return value and true
    return input, true;
}

fun main() {
    i32: val = 0;
    bool: ok = true;

    // implicit unpacking
    val, ok := parse_positive(-5);

    if (!ok) {
        std.println("Parsing failed");
        return;
    }

    std.println(val);
}
```

Listing 12.1: Idiomatic Error Handling

This pattern ensures that error handling is visible. The reader can see exactly where the error comes from and how it is handled, with no invisible jumps in execution.

12.3 The Check Block vs Error Handling

It is important to distinguish between *Safety Checks* and *Logic Errors*.

- **Safety Checks (check blocks):** These are used for `:?` (nullable) types where the *existence* of data is unproven. This is a memory safety feature enforced by the compiler.
- **Error Handling (if/else):** This is used for logic errors (e.g., "File not found", "Permission denied"). These are handled at runtime using standard boolean logic.

Part VI

Modules and Organization

13

Modules and Packages

13.1 The Package System

Gink adopts a strict directory-based package system similar to Go. In this model, the file system structure dictates the logical organization of the code.

13.1.1 Directories are Packages

A package is defined as a directory containing one or more `.gink` source files. All files residing in the same directory belong to the same package and share the same namespace.

Unlike file-based module systems, you do not import individual files. You import the directory (the package), and all public symbols defined across the files in that directory become available.

13.1.2 Package Declaration

Every source file must begin with a package declaration. This statement identifies the logical group to which the file belongs.

```
// File: /math/algebra.gink
package math;

// File: /math/geometry.gink
package math;
```

Listing 13.1: Package Declaration

The package name defined in the source file generally matches the name of the directory containing it.

13.1.3 The Main Package

An executable program must have a defined entry point. This is managed by the `main` package. The directory containing the entry point (usually the project root) must be declared as `package main`.

```
// File: /src/main.gink
package main;

import std;

fun main() {
    std.println("System Start");
}
```

Listing 13.2: Entry Point

13.2 Visibility

Gink enforces strict encapsulation at the package level.

13.2.1 Private by Default

By default, all identifiers (functions, structures, variables, constants) are private. They are visible only to other files within the same package (directory). They cannot be accessed by code in other directories.

13.2.2 Public Export

To expose an identifier to external packages, you must explicitly mark it with the `pub` keyword.

```
package math;

// Accessible to importers
pub fun i32: add(i32: a, i32: b) {
    return a + b;
}

// Only visible inside the 'math' package
fun i32: internal_calc() {
    return 0;
}
```

Listing 13.3: Exporting Symbols

13.3 Imports

To use code from another package, you use the `import` statement. The import path corresponds to the project's directory structure or the aliases defined in the dependency file.

13.3.1 Local Package Imports

If your project contains subdirectories, you import them using their path relative to the module root.

Consider the following project structure:

```
/project
  |-- reqter
  |-- main.gink
  |-- /utils
       |-- strings.gink  (package utils)
       |-- numbers.gink  (package utils)
```

To use the code in `/utils` from `main.gink`:

```
package main;

use utils;

// use is for sub-directories
// we use import for reqter modules

fun main() {
    // Access symbols via the package name
    utils.format_string("Hello");
}
```

13.4 Dependency Management

Gink uses a decentralized dependency management system driven by the `reqter` file. This file resides at the root of your project and functions similarly to `Cargo.toml` in Rust.

13.4.1 The reqter File Configuration

The `reqter` file uses TOML syntax to define project metadata and map external dependencies to local package aliases.

```
[project]
name = "my_server"
version = "0.1.0"
```

```
authors = ["Jayatheerth Kulkarni"]

[dependencies]
// Format: alias = { git = "url", tag = "version" }
json = { git = "https://github.com/gink/json.git", tag = "v1.2" }
http = { git = "https://github.com/gink/http.git", branch = "main" }
```

Listing 13.4: reqter Configuration

13.4.2 Using External Dependencies

Once a dependency is defined in `reqter`, it is available to your code as a first-class package. You import it using the alias defined in the `[dependencies]` section.

```
package main;

import json;
import http;

fun main() {
    json.parse("{}");
}
```

13.4.3 The gink pull Command

To simplify adding dependencies, the CLI provides the `pull` command. This fetches the remote repository and automatically appends the correct entry to your `reqter` file.

```
$ gink pull https://github.com/auth/auth.git
> Fetching repository...
> Enter package alias [default: auth]: my_auth
> Added 'my_auth' to reqter.
```

13.5 Building Libraries

In Gink, a library is structurally identical to an application, with the exception that it does not contain a `main` package or a `main()` function.

To create a library: 1. Initialize a new project directory. 2. Create a `reqter` file defining the library name and version. 3. Write your code in the root directory or subdirectories, ensuring you add `pub` to any function or type you wish to expose. 4. Push the directory to a Git hosting provider.

Users can then import your library directly via its Git URL, aliasing it however they prefer in their own projects.

13.6 Local Packages (The use Keyword)

To access code organized in subdirectories within your project (local modules), you must use the `use` keyword. The path provided to `use` is strictly relative to your project root.

13.6.1 Deeply Nested Packages

For larger projects, packages are often nested several layers deep. Gink supports navigating this hierarchy using dot notation.

Consider the following project structure where the logic for a game engine is nested two levels deep:

```
/my_game
  |-- main.gink
  |-- /physics          (Level 1 Directory)
       |-- /mechanics    (Level 2 Directory)
            |-- rigid.gink  (Declares: package mechanics)
```

To import the `mechanics` package into `main.gink`, you chain the directory names using dots.

```
package main;

// 'use' drills down the directory structure: physics -> mechanics
use physics.mechanics;

fun main() {
    // Access symbols via the package name (the last segment)
    mechanics.simulate_gravity();
}
```

Listing 13.5: Using Nested Packages

Important Rule: The identifier used in your code is always the **package name** (usually the last segment of the path), not the full path. In this example, you call `mechanics.simulate()`, not `physics.mechanics.simulate()`.

13.7 Handling Naming Conflicts

A strict design rule in Gink is that every package identifier in a source file must be unique. The compiler will not attempt to guess which package you mean if two imports share the same name.

13.7.1 The Conflict Scenario

Consider a scenario where you are building a web service. You have a local package for database utilities (`/db/sql`) and an external dependency for SQL formatting (also named `sql`).

```
package main;

// Imports external dependency 'sql'
import sql;

// Imports local package '/db/sql'
use db.sql;

fun main() {
    // COMPILER ERROR: Ambiguous reference 'sql'.
    // Does this refer to the external lib or the local package?
    sql.query();
}
```

Listing 13.6: Conflict Error

13.7.2 Resolution: The 'as' Keyword

To resolve this, you must explicitly rename one or both of the imports using the `as` keyword. This creates a local alias valid only within the current file.

```
package main;

// Keep the external lib as 'sql'
import sql;
// You can only use as keyword with the use keyword

// Rename the local package to 'db_sql'
use db.sql as db_sql;

fun main() {
    // Explicitly uses the external library
    sql.format("SELECT *");

    // Explicitly uses the local package
    db_sql.connect();
}
```

Listing 13.7: Resolving Conflicts

13.7.3 Reqter Alias vs. Import Alias

It is important to distinguish between the two types of aliasing in Gink:

1. **Project-Level Alias (reqter):** This renames a dependency for the *entire project*. If you define `my_json = "..."` in `reqter`, you must strictly import it as `import my_json` everywhere.

2. **File-Level Alias (as):** This renames a package for the *current file only*. This is purely for avoiding name collisions in a specific context and does not affect other files.

14 Standard Library Overview

Work to be done here

15 The End

Hey, thanks for reading this book. This represents a personal project of mine. There may be contradictions or some errors, or maybe even typos, please email

» jayatheerthkulkarni2005@gmail.com

I will fix them as I release newer drafts.