Designing a Programming Language: From Concept to Implementation

Building an Interpreter Using Modern C++

Prepared by : Ayman Alheraki

First Edition

Designing Programming Language: From Concept to Implementation Building an Interpreter Using Modern C++

By Ayman Alheraki

ForgeVM.org

July 2025

Contents

Author's Introduction

This book, *Designing a New C-Style Programming Language: From Concept to Implementation*, was written to guide programmers through the complete journey of designing and building a working interpreted language using modern C++20 and C++23.

The core objective is **to demystify the internals of programming languages** by offering a hands-on, modular, and deeply technical approach. Rather than relying on high-level tools or code generation frameworks, the book takes a **low-level, system-oriented path** — implementing each component step by step: from lexical analysis and parsing, to scope management, expression evaluation, function calls, and performance optimizations.

My motivation stems from decades of experience in systems programming and my belief that many developers use languages daily without truly understanding their structure or behavior. By building a real interpreter that mimics the semantics of C-style languages, readers gain not only implementation skills but also deeper insight into the fundamental principles of language design.

I wrote this book to share that journey — clearly, practically, and professionally — to empower every capable programmer to go from user to creator.

Stay Connected

For more discussions and valuable content about **Designing a Programming**

Language: From Concept to Implementation : Building an Interpreter Using Modern C++.

I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Walaa Owier

Preface

Designing a New C-Style Programming Language: From Concept to Implementation Building an Interpreter Using Modern C++ (C++20/23)

Why Build a New Language?

The creation of a new programming language is a rare pursuit — one that demands curiosity, discipline, and a deep understanding of both syntax and semantics. Yet, throughout the last five decades, programming languages have evolved primarily through adaptation, extension, and reinterpretation of foundational designs — and few have been as enduring as the C family of languages.

C introduced a compact, low-level language model that influenced not only system software but also the conceptual structure of countless modern languages — from Java and JavaScript to C++, C#, Go, and even Rust. What made C—and later, C++—so successful is not just performance or proximity to hardware, but its clean, minimal syntax and expressive control over program structure.

This book is born from a simple yet bold idea: to **design and implement a new interpreted programming language** using modern C++ — a language that honors the heritage of C-style syntax, while embracing the readability, modularity, and safety features expected in contemporary development. Unlike existing interpreters or domain-specific languages designed for niche areas, this project aspires to be **general-purpose**,

educational, and **production-aware**.
It serves both as a hands-on implementation manual and as a case study in the architecture of real-world programming languages. The journey begins with tokenization and parsing, traverses deep into evaluation engines and function systems, and ascends toward embedding, debugging, and optimization — all constructed from scratch using C++20/23.
This book is not just about theory. It is about engineering.

Target Audience and Purpose

This book is written for a particular kind of reader — the one who wants to go beyond using languages and instead understand how they work. If you are interested in how source code becomes behavior, how interpreters track scope and type, and how a runtime stack is built and maintained — then this book is for you.

This book is especially relevant to:

- **Experienced C++ developers** wishing to broaden their design abilities into compilers and interpreters.
- **Educators** seeking a clear, practical resource for teaching language implementation through C++.
- **Systems programmers** designing DSLs or configuration languages for hardware, simulations, or compilers.
- **Language hackers and language design hobbyists** who want to make their own scripting or experimental languages.

- **Compiler researchers and tooling developers** exploring fast prototyping using modern C++ capabilities.

Prerequisites:

- Solid command of **C++ syntax and semantics**, especially C++17 and newer.
- Familiarity with programming language fundamentals: tokens, parsers, expressions, statements, and scoping.
- Interest in the **architecture of interpreters**, not just front-end parsing but full language execution stacks.
- Experience with **modular software development**, CMake, and optional familiarity with unit testing frameworks.

This book is designed to be **progressive and self-contained** — each chapter builds on the last, with increasingly sophisticated structures, while remaining grounded in practical implementation. By the end, you will not only understand how languages are built — you will have built one.

Book Structure and Roadmap

The book is divided into eight logically structured parts, followed by detailed appendices. Each part represents a clear phase in the construction of the language interpreter and runtime. Here's what each part offers:

Part 1: Foundation and Architecture

Sets up the base project using C++20/23 and CMake. Introduces the project layout, lexer/parser planning, and toolchain independence.

Part 2: Lexing and Parsing

Implements a hand-written lexer and recursive-descent parser with C-style grammar and operator precedence. This forms the syntactic core of the language.

Part 3: Syntax and Structure

Handles blocks, declarations, conditional statements, loops, and expression-based statements. You begin seeing your language behave like C.

Part 4: Evaluation Engine

Implements the interpreter runtime: value representations, type system, variable scoping, function evaluation, and expression calculation.

Part 5: Control Flow and Functions

Adds support for full function definitions, recursion, parameter passing, activation records, return values, and local scopes — the heart of C-style programming.

Part 6: Collections and Advanced Features

Introduces arrays, basic data structures, and standard library utilities (e.g., print, input, string functions), including file I/O and expression utility functions.

Part 7: Production-Quality Features

Presents error handling, debugging, REPL, stack traces, symbol resolution, command-line execution, and performance instrumentation.

Part 8: Optimization and Advanced Topics

Concludes with optional topics such as building a bytecode VM, optimizing interpreter performance, distribution, modular architecture, and embedding the language in other applications.
The book culminates with full appendices covering language grammar in EBNF, CMake integration for language projects, testing strategies, benchmarks, and future work.

Philosophy, Design Goals, and Methodology

Every system in this book is driven by clear and consistent design values:

Precision Over Abstraction

Wherever feasible, components are designed with minimal abstraction. Instead of relying on parser generators or heavy macro systems, all systems — from token matching to AST evaluation — are written manually and explained in detail.

Realistic C-style Modeling

The language mirrors the behavior of C-like languages: block scoping with braces, strong yet flexible typing, well-defined operator precedence, and side-effect-bearing statements. It is not a toy — it mimics production-worthy language behavior.

C++20/23 as the Implementation Language

Modern C++ is powerful and expressive. With `std::variant`, `std::optional`, ranges, concepts, lambdas, `constexpr`, smart pointers, and efficient containers, it provides the perfect toolkit for interpreter design while avoiding the pitfalls of earlier C++ versions.

Separation of Concerns

Each component of the language — the lexer, parser, AST, evaluation, runtime environment, and standard library — is clearly separated. This ensures that you can refactor, test, and extend with confidence and modularity.

Developer Empowerment

The goal is not just to write an interpreter but to build a foundation you can evolve — adding bytecode, building a compiler frontend, integrating with scripting tools, or repurposing components into embedded engines.
Each chapter concludes with a **Milestone**, ensuring you walk away not just with theory but a fully running feature. Hands-on coding, step-by-step diagnostics, and testable systems are embedded throughout the book.

Final Thoughts and What You Will Accomplish

By the end of this book, you will have accomplished what few developers ever do — created a working, extensible, modular, C-style interpreted language from scratch. You will be capable of:

- Designing and implementing a syntax that is readable, minimal, and flexible.
- Writing a full interpreter in C++ using variant-based evaluation and modern type safety.
- Building function dispatch, return stacks, and expression evaluation systems that reflect real-world compiler internals.
- Creating an extensible REPL and file execution engine.

- Embedding your language into larger systems.
- Measuring and optimizing interpreter performance.
- Packaging your language for use across platforms.
- Planning for future features such as object systems, foreign function interfaces, or bytecode JIT support.

The appendices serve as technical guides and reference implementations for everything you build, ensuring your language remains usable and relevant beyond the book.
This is more than just a technical book. It is a blueprint for ownership over your computing tools — a return to systems understanding in a world increasingly dependent on abstraction.
Whether your goal is academic, professional, or personal, I invite you to make this book your companion in mastering the art of language design. You will not only write code — you will define what code means.
Let's begin.

Part I

Foundation and Architecture

Chapter 1

Why Design a New Programming Language?

1.1 Motivation Behind Creating New Programming Languages

In the rapidly evolving domain of computer science and software engineering, the creation of a new programming language is not merely an academic exercise or a vanity project. It is often a response to real limitations in existing tools, a push for innovation, or a necessity dictated by emerging domains, hardware capabilities, or architectural patterns. Over the last five years, the resurgence in language design—particularly those inspired by the C tradition—has been fueled by a complex interplay of technological, practical, and philosophical motivations.

This section explores the concentrated motivations for designing a new programming language, particularly one that aligns with C-style syntax and semantics, while leveraging the expressiveness, safety, and abstraction capabilities introduced in **C++20** and **C++23**.

1.1.1 Addressing Limitations in Existing Languages

Despite the dominance of legacy languages like C, C++, and Java, modern software demands have exposed their deficiencies in various areas:

- **Safety and Undefined Behavior**: C and early C++ place performance above safety, leaving programmers vulnerable to undefined behavior, memory corruption, and data races. Designing a new language allows you to enforce compile-time guarantees without sacrificing system-level control.
- **Verbosity and Boilerplate**: Even with C++17 and prior standards, writing expressive, reusable code often involved excessive boilerplate. C++20 introduced *concepts*, *ranges*, and *coroutines* to combat this, inspiring newer languages and tools to reimagine simpler syntaxes and workflows.
- **Concurrency Models**: Traditional languages provide limited abstractions for safe and ergonomic multithreading. With the emergence of `std::jthread`, *atomic smart pointers*, and *semaphores* in C++20/23, we now have the vocabulary to reimagine concurrency as a first-class citizen in language design.

1.1.2 Integration of Modern Language Features with Lower-Level Control

Many recent systems-level languages (like Rust or Zig) aim to balance high-level abstraction with low-level control, but their syntax or compiler complexity might be off-putting for developers from C-style backgrounds.
Designing a new C-style language that leverages **modern C++ internals** (via modules, constexpr, template metaprogramming, and ranges) gives the designer the opportunity to:

- Provide **compile-time reflection and introspection**
- Enable **efficient deterministic memory handling** without garbage collection
- Build **powerful DSLs** using C++20 constexpr evaluators
- Enforce **user-defined safety contracts** via concepts and static assertions

A custom language interpreter written in modern C++ allows tight control over performance, parsing strategies, and the ability to evolve faster than general-purpose compilers.

1.1.3 Domain-Specific Needs and Embedded Use-Cases

Many new programming languages are born to serve specialized industries:

- **Embedded systems**
- **Robotics and IoT**
- **High-frequency trading**
- **Scientific simulation and GPU programming**

A general-purpose language may fail to offer concise constructs for these use-cases. By creating a new language, designers can provide:

- Custom data types for hardware representation
- Deterministic memory layouts and zero-cost abstractions
- Inline real-time constraints and timing models
- Platform-aware syntax that maps closely to hardware registers or signals

C++23 features like `constexpr dynamic allocation` and `std::span` now allow better memory views and compile-time memory modeling, which can be leveraged in a new language runtime or interpreter.

1.1.4 Pedagogical and Research Objectives

Another clear motivation is **educational clarity**. Language design exposes students and professionals to compilers, parsers, runtime models, and systems architecture in a unified, project-based framework. Creating a new language helps reinforce foundational CS concepts:

- **Lexical analysis and syntax trees**
- **Semantic resolution**
- **Intermediate representations**
- **Memory models and garbage collection algorithms**
- **Just-in-time (JIT) vs Ahead-of-Time (AOT) compilation**

Using modern C++ (especially ranges, variant, and coroutines), a full-featured interpreter can be implemented in a way that is modular, readable, and maintainable, turning abstract theory into concrete working code.

1.1.5 Experimentation and Language Innovation

Language design is often a space for experimentation—trying new models for:

- **Error handling** (optional types, monads, or result types)
- **Immutability and purity**

- **Pattern matching and algebraic data types**
- **Macros or compile-time metaprogramming**
- **Customizable syntax and infix operators**

While traditional C++ is bound by backward compatibility, a new language can embrace new paradigms without legacy constraints. C++20 and C++23 provide all necessary tools to build and test these concepts rapidly:

- `consteval` and `constinit` for compile-time execution
- Advanced template constraints via `requires` and `concepts`
- Lambda improvements and pipeline expressions with ranges
- Pattern-based traversal using `std::visit` on `std::variant`

1.1.6 Performance Transparency and Predictability

In modern computing, **predictable performance** is as important as raw speed.
A major reason behind new language creation is to avoid the opacity of runtime optimizations, GC pauses, or unexpected heap allocations.
A new language—especially when implemented in modern C++—can guarantee:

- Explicit stack vs heap allocation rules
- Fine-grained control over function inlining and tail calls
- Trivially destructible or relocatable types
- Custom ABI rules tailored for interop or hardware targets

Using C++20/23's `[[no_unique_address]]`, improved `constexpr` capabilities, and `std::bit_cast`, developers can build transparent, efficient data movement strategies directly into the language runtime.

1.1.7 Cultural and Ecosystem Reset

Some motivations are philosophical rather than technical. A new language offers a **cultural reset**, stripping away decades of accumulated legacy and forcing a rethinking of best practices.
By setting new defaults—immutability, safety, module-based encapsulation, strong typing—designers can influence coding culture from the ground up. Combined with C++ modules (introduced in C++20), this facilitates a modern build system and package manager, avoiding the traditional pitfalls of header-based dependency hell.

1.1.8 Conclusion

The motivation to design a new programming language stems from a deep desire to improve expressiveness, safety, and performance while removing the friction caused by legacy constraints. With the maturity of C++20 and C++23, developers are uniquely positioned to create performant, readable, and maintainable language tools, interpreters, and runtimes that not only demonstrate engineering precision but also serve as a launching pad for language innovation. By building on modern C++'s powerful metaprogramming, memory safety aids, and concurrency features, the creation of a new C-style language is no longer a monumental task—but a rewarding exploration of the very essence of software design.

1.2 Analysis of C-Style Languages: C, Go, Rust, Zig

In order to design a modern and competitive C-style programming language, it is essential to critically analyze the dominant and emerging players within this family. C-style languages are united by a common structural syntax—block delimiters using braces `{}`, statement terminations using semicolons `;`, infix expressions, and

relatively low-level control over memory and computation. However, the **design philosophies**, **target domains**, and **evolutionary trajectories** of these languages diverge significantly.
This section presents a focused analysis of **C**, **Go**, **Rust**, and **Zig**—each of which has carved out a strong position in systems programming, and each with implications that must inform any new language effort. The evaluation reflects key design principles, recent language developments (2019–2024), and what modern **C++20/23** can leverage from or improve upon.

1.2.1 C: The Root of the Tree

Overview:
C remains the foundational language of system software and low-level development. It is prized for its **predictable performance**, **direct memory manipulation**, and **minimal runtime overhead**. However, it suffers from lack of safety guarantees, absence of modern abstractions, and an outdated toolchain philosophy.
Strengths:

- Extremely small and portable runtime
- Full control over memory layout and hardware interaction
- ABI stability and mature compiler infrastructure (GCC, Clang)

Limitations:

- No concept of modules, generics, or strong typing beyond structs and unions
- High risk of undefined behavior and buffer overflows
- Poor support for concurrency abstraction and memory safety

Implications:
Any new C-style language must honor C's performance ethos while enforcing stricter **compile-time guarantees**. Using **C++20/23 features** like `concepts`, `consteval`, `constexpr allocation`, and strong typing via `variant` and `optional`, one can address safety and abstraction without sacrificing control. C remains a baseline reference, not a model to replicate uncritically.

1.2.2 Go: Simplicity Over Power

Overview:
Go (or Golang), developed at Google, emphasizes **simplicity, concurrency, and developer productivity** over feature-rich abstraction or low-level control. It targets backend systems, microservices, and cloud infrastructure.
Strengths:

- Minimalist syntax; easy learning curve
- Built-in garbage collection and CSP-style goroutines for concurrency
- Static binaries and cross-compilation ease

Limitations:

- No generics until 2022; limited metaprogramming support
- Garbage collection introduces unpredictable latency
- Weak abstraction capabilities for low-level system tasks

Recent Developments:

- Introduction of type parameters (generics) in Go 1.18

- Continued refinement of the module and toolchain ecosystem

Implications:
Go demonstrates that **developer ergonomics** and **consistent tooling** matter, especially for adoption. However, its lack of deterministic memory handling, compile-time evaluation, and low-level control makes it unsuitable as a foundation for embedded or real-time systems.
A modern C++ interpreter can adopt **Go's clean module system** and **error-handling discipline** (e.g., explicit error return values), but should reject its garbage-collected runtime in favor of **RAII** and **move semantics** offered by modern C++.

1.2.3 Rust: Safety with Zero-Cost Abstractions

Overview:
Rust is the most radical rethinking of C-style programming in the last decade. Its central feature is **ownership-based memory safety**, enabling deterministic memory management without garbage collection. Rust's goal is to eliminate entire classes of bugs at compile-time.
Strengths:

- Ownership and borrowing model enforce memory safety
- Zero-cost abstractions via monomorphization and inlining
- Strong concurrency model: Send, Sync, and thread safety by design
- Macro system and rich type inference

Limitations:

- Complex syntax and steep learning curve

- Slow compile times due to heavy monomorphization
- Lack of reflection or runtime code generation
- No stable ABI; challenging to integrate with other languages dynamically

Recent Developments:

- Stabilization of `const` generics and async improvements
- Ecosystem growth in embedded, web (via WebAssembly), and OS development
- Initiatives like `Rust-for-Linux` and `cargo` standardization

Implications:
Rust proves that **compile-time memory safety** is viable without garbage collection. However, its complexity, especially around lifetimes and trait resolution, often deters new users.
A C++20/23-based language can adopt **similar safety models** using `unique_ptr`, `span`, `std::optional`, and `concepts`, while offering **easier integration with C++ or C**. Moreover, with `constexpr` and `consteval`, C++ can now mimic many of Rust's compile-time guarantees in a more familiar syntax.

1.2.4 Zig: Pragmatism Meets Control

Overview:
Zig is a newer C-style language focused on **manual memory management, simplicity, and performance transparency**. It aims to replace C in domains like OS kernels, embedded systems, and performance-critical applications.
Strengths:

- No hidden control flow: no exceptions, macros, or GC

- Compile-time code execution with `comptime`
- Manual memory allocation with explicit ownership patterns
- Strong interop with C (drop-in C replacement)

Limitations:

- Limited ecosystem and tooling maturity
- Verbose allocation patterns compared to higher-level languages
- Fewer abstraction tools than Rust or C++

Recent Developments:

- Enhancements to the Zig build system and cross-compilation
- Growing use in game engines, graphics, and system bootloaders
- `comptime` as a language-native alternative to templates or macros

Implications:
Zig embraces **full transparency and user control**, rejecting language complexity and automation. It shows that a **well-designed compile-time execution model** (akin to `consteval` in C++20) can replace preprocessor macros and support meta-programming safely.
A new C-style language built in C++ can achieve similar goals using:

- `constexpr` classes and expressions
- `std::array`, `std::bit_cast`, and fixed-layout structs
- `concepts` and tagged unions (`std::variant`) for compile-time safety

Zig's design reinforces the idea that **simplicity and predictability** are as valuable as expressiveness in systems programming.

1.2.5 Comparative Summary

Comparison of Features Across C, Go, Rust, and Zig

Feature / Language	**C**	**Go**	**Rust**	**Zig**
Memory Safety	No	Partial (GC)	Yes (Ownership)	Manual, Explicit
Concurrency Model	Low-level	Goroutines	Compile-time Safe Threads	Manual
Compile-time Eval	Preprocessor	Limited	`const fn`, macros	`comptime`
Metaprogramming	None	Weak	Macros, Traits	`comptime`
Interop with C	Native	Good	Good but indirect	Excellent
Toolchain Simplicity	High	Very High	Moderate	Moderate
ABI Stability	Yes	Yes	No	Yes
Runtime Overhead	None	GC overhead	None	None

1.2.6 Conclusion

Each of the analyzed C-style languages provides distinct lessons for the design of a new programming language:

- From **C**, we learn the importance of **transparency and control**, but also the risks of underspecified behavior.
- From **Go**, we take inspiration in **simplicity and developer experience**, while avoiding its runtime abstraction costs.
- From **Rust**, we inherit **safety, concurrency, and expressive power**, while seeking a gentler learning curve and interoperability.
- From **Zig**, we embrace **manual control, predictability, and modern compile-time evaluation**, refined with better abstraction support via modern C++.

The modern C++20/23 standard is now powerful enough to serve as the **implementation platform** of a new interpreter or language runtime that integrates these lessons. With `concepts`, `modules`, `consteval`, `jthread`, `std::span`, and rich type algebra, C++ empowers developers to encode safety, performance, and modularity directly into their language infrastructure—delivering both **pragmatism and precision** in language design.

1.3 Designing Our New Language — Goals and Philosophy

The design of a new programming language is not merely a technical exercise, but an articulation of a clear philosophy—a vision that defines the language's purpose, its audience, and its boundaries. Informed by decades of evolution in system programming, the shortcomings of legacy tools, and the growing power of modern C++ (particularly C++20 and C++23), this section defines the **core goals and guiding philosophy** of our new C-style language.

Rather than reinventing syntax for novelty or copying features blindly, we aim to deliver a **cohesive, efficient, safe, and expressive language** tailored for system-level tasks, scripting, and educational use, implemented entirely using the capabilities of modern C++.

1.3.1 Goal 1: Simplicity without Sacrificing Power

One of the critical lessons from languages like Go and Zig is the value of simplicity: a language should be learnable, readable, and predictable. Yet, simplicity must not come at the cost of expressive power. Our language aims to:

- **Reduce cognitive overload** through clean, orthogonal syntax
- Eliminate unnecessary boilerplate while preserving semantic clarity
- Adopt a small, consistent set of core constructs that serve multiple roles

Using **C++20 concepts**, `constexpr`, and clean recursive descent parsing (enabled by `std::variant`, `std::visit`, and pattern matching constructs), we will design a grammar and interpreter that enable simplicity at both the syntax and runtime level.

1.3.2 Goal 2: Deterministic and Explicit Memory Model

Memory safety is a core concern of any new language. Instead of relying on garbage collection or complex borrow-checking like Rust, we embrace **explicit ownership with deterministic lifetime rules**. This approach is enabled by:

- Stack-first allocation model with scoped lifetime (RAII style)
- Optional heap allocation with explicit ownership transfer
- Value semantics by default; references must be opt-in and visible in syntax
- Built-in types for ownership: `ptr`, `ref`, `slice`, and `own`

These concepts map cleanly to **C++23 primitives** like `std::unique_ptr`, `std::span`, and `std::optional`, and can be mirrored in our language using similar idioms and internal representations.

1.3.3 Goal 3: Predictable and Safe Concurrency

Concurrency is a necessity in modern computing but remains error-prone in many languages. Our language will introduce **a simplified concurrency model** inspired by the advances in C++20/23:

- Concurrency primitives built into the language (`spawn`, `join`, `atomic`)
- Lightweight thread abstraction using `std::jthread` or coroutine-based dispatch
- No shared mutable state by default; message-passing and copy semantics preferred
- Race condition mitigation through compile-time ownership enforcement

By designing the interpreter using **`std::atomic`, `std::barrier`, and `std::latch`** where needed, concurrency can be tested and verified directly in the core runtime, ensuring safety without over-complexity.

1.3.4 Goal 4: Compile-Time Programming and Meta-Evaluation

Static correctness and performance optimization often require compile-time logic. We embrace this with **first-class compile-time evaluation**, inspired by both Zig's `comptime` and C++'s `consteval`:

- Functions and expressions can be marked to evaluate at compile time
- Constants are resolved and validated during parsing or AST building
- Type reflection and introspection tools are available in user space
- Simplified compile-time dispatch based on traits and argument types

Modern C++ provides robust compile-time execution using `constexpr`, `consteval`, and `template constraints`. Our language can abstract these through a type system capable of **static dispatch, constant folding**, and **error generation at parse time**, not runtime.

1.3.5 Goal 5: Strong but Flexible Type System

A powerful type system helps detect errors early without overwhelming the user. Our language will feature:

- Statically typed variables with optional type inference
- Algebraic data types and tagged unions
- Custom types, traits, and interfaces
- Structural typing for generic programming
- Optional nullability via explicit `maybe<T>` or `option<T>`

This design is deeply inspired by C++'s modern tools:

- `std::variant` and `std::monostate` for safe unions
- `std::optional` for nullable values
- `concepts` and `requires` for generic constraints

The type checker in our interpreter will be built using **C++20 type traits and `std::is_...` utilities**, extended by custom concepts to enforce trait conformance, making the language safe and predictable.

1.3.6 Goal 6: Minimal Runtime and High Portability

To serve embedded systems, education, and rapid tooling, our language must maintain a **minimal runtime**. That means:

- No dependency on external garbage collection or JIT engines
- No dynamic linking or external runtime initialization
- Self-contained interpreter with clear execution model
- Portability across platforms and architectures

Thanks to **C++ modules**, `constexpr allocation`, and modern ABI-stable components, we can design a runtime that is small, deterministic, and portable across operating systems and hardware targets without external dependencies.

1.3.7 Goal 7: Modularity, Encapsulation, and Package Discipline

A modern language must facilitate large-scale software construction. Our language will include:

- Module-based code organization from the ground up
- Namespaces or packages to prevent symbol collisions
- File-based compilation units with explicit imports and exports
- Optional inline modules for scripting and REPL usage

C++20's **module system** gives us the tools to build a modular interpreter backend. This will be reflected in the new language's compiler and runtime, allowing developers to **encapsulate logic** and share reusable components cleanly.

1.3.8 Goal 8: Syntax Familiarity with Semantic Rigor

The language will remain **C-style** in its surface syntax to lower the learning barrier and attract existing C/C++ developers, but it will diverge from legacy semantics by enforcing:

- No implicit type conversions between incompatible types
- No default integer promotion rules
- Explicit pointer dereferencing and address acquisition
- Clear expression of mutability and ownership in function parameters

In other words, **syntax will feel familiar**, but **behavior will be safer, stricter, and more predictable**, thanks to a rigorous parser and semantic checker built using C++20 constructs like `std::variant`, `std::monostate`, and `std::visit`.

1.3.9 Philosophy in Summary

The philosophy of our language can be summarized in the following core principles:

- **Clarity over cleverness**: No hidden magic; behavior is explicit
- **Safety by design**: Eliminate undefined behavior at compile time
- **Performance transparency**: Developers should understand the cost of every abstraction
- **Modern by default**: The language will use idioms and patterns enabled by C++20/23
- **Minimalism and completeness**: Few core features, but deeply composable
- **Practicality over perfection**: Designed for real-world use, not theoretical elegance

1.3.10 Conclusion

Our new language is not intended to replace C++ or compete with Rust—it is **a response to the challenges faced by developers today**, using the most powerful and refined tools C++ has introduced in the last five years. By embedding our interpreter in modern C++20/23, we gain unparalleled performance, abstraction, and safety in the implementation itself. At the same time, we produce a language that serves a purpose: clarity, correctness, and control in the hands of those who build systems.

1.4 Example Code in Our Target Language

To crystallize the goals and philosophy of our new C-style language, this section introduces **example source code** written in the proposed syntax and semantics of the language. This code is not only illustrative of its intended usage but also highlights the **motivations behind its design**, rooted in the practical capabilities of **C++20/23** which powers its interpreter.
These examples are deliberately designed to reflect the essential traits of the language:

- Simplicity and clarity in syntax
- Safety and deterministic behavior
- Explicit memory and ownership management
- Compile-time evaluation and static typing
- Concurrency primitives built into the language
- Minimal standard runtime abstractions

The language borrows its surface syntax from the C/C++ family but enforces modern principles such as **immutability by default**, **explicit mutability**, **explicit ownership transfer**, and **built-in safe types** like `Option`, `Result`, and `Slice`.

1.4.1 Hello World — Minimal Entry Point

```
fn main() -> int {
    print("Hello, World!");
    return 0;
}
```

Key Observations:

- `fn` is the keyword for function definition
- `main` returns an integer, explicitly
- `print()` is a built-in function in the minimal standard library
- Semicolon enforces clear statement boundaries
- No global side-effects or preprocessor use

1.4.2 Immutable and Mutable Variables

```
fn demo() {
    let x: int = 10;        // Immutable by default
    let mut y: int = 20;    // Mutable with 'mut'

    y = y + x;
    print(y);               // Output: 30
}
```

Design Notes:

- `let` declares a variable; `mut` marks it mutable
- No implicit type promotion
- All variables are block-scoped
- Encourages value-oriented programming, discourages shared mutability

1.4.3 Ownership and Option Type

```
fn safe_divide(a: int, b: int) -> Option<int> {
    if b == 0 {
        return none;
    }
    return some(a / b);
}

fn use_divide() {
    let result = safe_divide(10, 2);
    if result is some(val) {
        print("Result: ", val);
    } else {
        print("Division by zero");
    }
}
```

Conceptual Design:

- `Option<T>` is a built-in tagged union: `some(T)` or `none`
- Pattern matching is minimal but expressive
- Avoids nulls entirely by requiring explicit handling
- Internally backed by `std::optional` in the C++ interpreter

1.4.4 Compile-Time Evaluation

```
const fn factorial(n: int) -> int {
    if n <= 1 {
        return 1;
    }
    return n * factorial(n - 1);
}

fn main() {
    const fact_5: int = factorial(5); // Computed at compile time
    print(fact_5);                    // Output: 120
}
```

Engine Behavior:

- `const fn` marks functions evaluable at compile-time
- Can be folded by the interpreter before execution
- Uses `constexpr` or `consteval` under the hood in C++20/23
- Supports embedded and systems programming via deterministic computation

1.4.5 Struct and Pattern Matching

```
struct Point {
    x: float;
    y: float;
}

fn distance(p: Point) -> float {
    return sqrt(p.x * p.x + p.y * p.y);
}
```

```
fn example() {
    let origin = Point { x: 3.0, y: 4.0 };
    print("Distance: ", distance(origin)); // Output: 5.0
}
```

Design Insight:

- Structs have named fields and no implicit constructors
- No inheritance; instead, structural or trait-based behavior
- Internally modeled using `struct` and `std::variant` where polymorphism is required

1.4.6 Generics and Traits (Concepts)

```
trait Addable {
    fn add(self, other: Self) -> Self;
}

fn sum<T: Addable>(a: T, b: T) -> T {
    return a.add(b);
}
```

Semantics:

- Traits define required behavior (similar to C++ concepts or Rust traits)
- Generics with constraint `T: Addable`
- Enforced at compile-time; no runtime overhead
- Backed by `concepts` and `requires` clauses in C++20

1.4.7 Concurrency with Spawn and Join

```
fn compute() -> int {
    let mut result: int = 0;
    for i in 0..100 {
        result = result + i;
    }
    return result;
}

fn parallel_sum() {
    let t1 = spawn compute();
    let t2 = spawn compute();

    let r1 = join t1;
    let r2 = join t2;

    print("Total: ", r1 + r2);
}
```

Runtime Behavior:

- `spawn` creates a concurrent task (internally backed by `std::jthread`)
- `join` blocks and retrieves the result
- No shared mutable state; each function returns its own value
- Thread-safe by design with no race condition primitives exposed by default

1.4.8 Error Handling via Result

```
fn open_file(path: string) -> Result<File, string> {
    if path == "" {
        return err("Empty path");
    }
    return ok(File {});
}

fn main() {
    let file = open_file("log.txt");
    if file is ok(f) {
        print("Opened successfully");
    } else if file is err(e) {
        print("Error: ", e);
    }
}
```

Design Philosophy:

- No exceptions
- All errors must be handled explicitly
- `Result<T, E>` is modeled using a discriminated union
- Backed internally by `std::variant` or equivalent C++ construct

1.4.9 Slices and Bounds Safety

```
fn print_slice(data: slice<int>) {
    for i in 0..data.len {
        print(data[i]);
    }
}

fn use_slice() {
    let arr = [1, 2, 3, 4];
    print_slice(arr[1..3]); // Prints: 2, 3
}
```

Safety Model:

- `slice<T>` is a view, not an owning container
- All indexing is bounds-checked by default
- Internally uses `std::span<T>` with range validation in debug builds

1.4.10 Modules and Imports

```
module math;

fn square(x: int) -> int {
    return x * x;
}
cppCopyEditimport math;

fn main() {
    print(math::square(5)); // Output: 25
}
```

Module Philosophy:

- File = module; explicit import/export
- No preprocessor; no text-based inclusion
- Namespaces resolved statically; no symbol collisions
- Mirrors `C++20` modules and supports layered build architecture

1.4.11 Conclusion

The example code demonstrates a **realistic and coherent C-style language** that learns from the best practices of recent systems languages while avoiding their complexity and performance compromises. Each feature shown is **backed by real capabilities in C++20/23**, ensuring that both the interpreter and language design remain aligned with proven techniques in modern software engineering.
The examples also reflect our language's goals:

- Simplicity with clarity
- Explicit memory and concurrency handling
- Compile-time power without hidden magic
- Strong, safe typing with low abstraction overhead

1.5 Milestone — Initial Language Specification and Code Examples

After identifying the motivations, evaluating similar languages, and articulating the design philosophy, we arrive at a critical turning point: the **first formal milestone** in building our new C-style language—a concrete, **initial language specification** accompanied by **representative code examples**. This section sets the groundwork for the parsing engine, interpreter, and further compiler stages, all to be implemented in **modern C++20/23**.

Rather than attempting to deliver a complete language upfront, this milestone focuses on a **minimal but functional subset** that reflects the design intent, demonstrates language behavior, and validates the foundational choices through real examples. It serves as a practical prototype to test the syntax, type system, memory model, and runtime behavior using C++ as the host language for implementation.

1.5.1 Language Subset Goals for the First Milestone

The initial specification is deliberately compact, focused on proving the most essential aspects of the language:

- **Syntax familiarity** for C/C++ developers
- **Strong static typing** and early error detection
- **Deterministic and explicit variable lifetime**
- **Function calls and scope rules**
- **Minimal standard library** (print, input, math)
- **Compile-time constant evaluation**

- **Safe handling of optional and result values**

By delivering a working interpreter for this subset, we build a strong foundation for advanced features such as modules, traits, generics, and concurrency in later milestones.

1.5.2 Core Language Grammar (Minimal Specification)

Here is a simplified Backus-Naur Form (BNF)-like definition of the core syntax:

```
program      ::= { function | struct }*

function     ::= "fn" identifier "(" [ parameters ] ")" [ "->" type ] block
parameters   ::= parameter { "," parameter }*
parameter    ::= identifier ":" type
block        ::= "{" statement* "}"

statement    ::= variable_decl | assignment | if_stmt | while_stmt | return_stmt |
↪  expr_stmt
variable_decl ::= "let" ["mut"] identifier ":" type "=" expression ";"
assignment   ::= identifier "=" expression ";"
return_stmt  ::= "return" expression ";"
if_stmt      ::= "if" expression block [ "else" block ]
while_stmt   ::= "while" expression block
expr_stmt    ::= expression ";"

expression   ::= literal | identifier | call_expr | binary_expr | grouping
call_expr    ::= identifier "(" [ expression { "," expression }* ] ")"
binary_expr  ::= expression binary_op expression
grouping     ::= "(" expression ")"

type         ::= "int" | "float" | "bool" | "string" | identifier
literal      ::= integer | float | string | boolean
```

This grammar defines the basic structures required to build meaningful programs. The interpreter will tokenize, parse, and execute code written in this subset using C++20/23 features like `std::variant`, `std::visit`, and `std::monostate` to represent and handle expressions and types.

1.5.3 Built-in Types and Rules

- **Primitive Types**
  - `int`: 64-bit signed integer
  - `float`: 64-bit floating-point number
  - `bool`: Boolean true/false
  - `string`: UTF-8 encoded string
- **Composite Types**
  - `Option<T>`: Represents a value that may or may not exist
  - `Result<T, E>`: Represents success or failure of operations
- **Rules**
  - All variables are immutable unless marked with `mut`
  - No implicit type conversions (e.g., `int` to `float` requires explicit cast)
  - Arithmetic follows type safety rules; overflow and underflow behavior are defined
  - All function parameters are passed by value unless specified otherwise

1.5.4 Semantic Rules and Behaviors

- **Scope**: Variables and functions are scoped to their enclosing block
- **Typing**: All expressions and function return types must be explicitly typed
- **Control Flow**: Conditional and loop expressions work with boolean types only
- **Error Handling**: No exceptions; `Option` and `Result` used for recoverable errors
- **Ownership**: All values follow value semantics; reference types will be added in a later stage

These behaviors will be validated and enforced by the interpreter's semantic analysis layer, powered by modern C++'s type traits, templates, and strong typing mechanisms.

1.5.5 Example: Program Using the Initial Specification

```
fn factorial(n: int) -> int {
    if n <= 1 {
        return 1;
    }
    return n * factorial(n - 1);
}
```

```
fn main() -> int {
    let x: int = 5;
    let y: int = factorial(x);
    print("Factorial: ", y);
```

```
    return 0;
}
```

This code demonstrates:

- Function definition and recursion
- Immutable variable binding
- Typed function parameters and return type
- Print output using minimal standard I/O
- Basic integer arithmetic

1.5.6 Interpreter Architecture Preview (C++20/23)

The interpreter implementing this specification will include:

- **Tokenizer (Lexer)**: Converts source code into tokens using `std::regex` or custom matchers
- **Parser**: Generates an AST using recursive descent, backed by `std::variant` nodes
- **AST Types**:
  - `Expression`: base class with variants for literals, binary ops, function calls
  - `Statement`: variants for declaration, assignment, control flow, return
- **Type System**: Checked statically using template-like rules with `concepts` and type traits

- **Runtime**:
  - Evaluation of expressions using `std::visit`
  - Memory and variable tracking via scoped environments
  - Function call stack using managed frames

C++23 features like `constexpr virtual dispatch`, `std::expected`, and structured bindings will help us maintain readable, robust, and efficient interpreter code.

1.5.7 Development Plan for Next Phase

This milestone serves as both a deliverable and a foundation. With the initial grammar and runtime working, the following next steps will include:

- Expanding the parser to support user-defined structs
- Adding pattern matching to `Option` and `Result` values
- Introducing loop constructs and basic collection types (array, slice)
- Establishing a standard library interface
- Modularizing the interpreter using `C++20` modules and build system integration

1.5.8 Conclusion

The initial language specification outlined here is a **compact yet expressive subset** that embodies the language's core goals: simplicity, safety, clarity, and deterministic behavior. It is designed for immediate implementation using modern C++ tools and constructs, offering a solid launchpad for rapid iteration and refinement.

This milestone reflects a **balance between theoretical design and practical engineering**. With carefully selected features and minimal syntax, we now have a functional language core that can evolve into a full-fledged system through a staged development process—driven by C++20/23's expressive power, compile-time evaluation, and modular infrastructure.

Chapter 2

Language Implementation Project Structure

2.1 Project Structure for a Programming Language Interpreter

Designing a modern interpreter for a new programming language requires a well-thought-out and scalable project structure from the outset. A clean architecture not only enables faster iteration during development but also improves maintainability, testing, extensibility, and onboarding for contributors. With the capabilities introduced in **C++20 and C++23**, we can now structure our interpreter using powerful modern features such as **modules**, **concepts**, **`constexpr` evaluation**, and **`std::variant`-based ASTs**, avoiding legacy pitfalls like monolithic headers, excessive preprocessor usage, and rigid class hierarchies.

This section provides a focused exploration of a professional, modular project structure for implementing our language interpreter in **modern C++**, targeting clarity, correctness, and long-term sustainability.

2.1.1 Overview: Goals of a Good Project Structure

The primary goals behind a good interpreter project structure are:

- **Separation of concerns** between lexing, parsing, semantic analysis, runtime, and user interface
- **Incremental compilation** and improved build times using **C++20 modules**
- **Extensibility** to allow future components like a bytecode VM or JIT
- **Testability** at every level: lexer, parser, type checker, evaluator
- **Toolchain compatibility**, ensuring clean builds across platforms

Each part of the interpreter pipeline must be represented as an **isolated subsystem** communicating through well-defined interfaces and type-safe structures.

2.1.2 High-Level Project Layout

A recommended directory and module structure for our interpreter:

```
/ForgeLang

  /src
      /core          → Core utilities, error system, memory abstractions
      /lexer         → Tokenization and source preprocessing
      /parser        → AST construction and syntax grammar
      /semantics     → Type checking and static validation
      /runtime       → Execution engine and built-in standard library
      /stdlib        → Language-defined standard functions (print, math)
      /vm            → (optional) Future bytecode or stack machine backend
      /main          → CLI, REPL, and entry point
```

```
/include          → Public headers (if needed by tooling)
/tests            → Unit and integration tests for each subsystem
/examples         → Example programs written in our new language
/docs             → Auto-generated and written documentation
/build            → Out-of-source CMake build folder
```

This directory layout supports modular builds, isolated testing, and gradual expansion without cross-component contamination.

2.1.3 Core Interpreter Modules and Responsibilities

Each subdirectory represents a self-contained **C++ module or namespace**. Below is a breakdown of each major component:

- **a. `/core` — Foundation Utilities**

  Contains:

  - `SourceManager`: manages file loading, line/column mapping
  - `ErrorReporter`: structured diagnostics with error codes and hints
  - `ArenaAllocator`: optional custom allocator for AST and runtime
  - Utility types: string views, IDs, file paths, hash maps

  C++ Features:

  - `std::string_view`, `std::expected`, `std::source_location` (C++20/23)
  - Custom diagnostic formatting via `std::format`

- **b. `/lexer` — Lexical Analyzer**

  Contains:

- `TokenType`: enum class defining all tokens
- `Token`: structure with type, value, span
- `Lexer`: class/function that scans source and emits tokens

Design:

- Use `std::variant` for token value storage (e.g., string, int, float)
- Return `std::vector<Token>` or token stream iterator
- Support comments, whitespace control, and source span tracking

C++ Features:

- Ranges and iterators
- Unicode support via `char8_t` and `std::u8string_view` if required

- **c. `/parser` — Abstract Syntax Tree Construction**

  Contains:

  - AST Nodes: `Expr`, `Stmt`, `Decl`, etc. implemented via `std::variant`
  - Parser class: consumes tokens and produces typed AST nodes
  - Grammar implementation using recursive descent

  Design:

  - Each AST node as a `struct` with a unique tag
  - Use `std::visit` to process or transform AST nodes
  - Error recovery strategies for invalid syntax

C++ Features:

- `std::variant`, `std::optional`, and pattern matching (C++23)
- `concepts` to constrain node processing

- **d. `/semantics` — Type Checking and Analysis**

  Contains:

  - Symbol table, scopes, and identifier resolution
  - Static type checker and trait verifier
  - Compile-time evaluation of `const fn`

  Design:

  - Use `std::map` or custom hash maps for scope chaining
  - Rich error types returned via `std::expected`
  - Trait system modeled with `concepts` or structural typing rules

  C++ Features:

  - `constexpr` and `consteval` for internal simulation
  - Strong typing with `enum class` and tagged unions

- **e. `/runtime` — Evaluation Engine**

  Contains:

  - `Evaluator` or `Interpreter` class that runs AST nodes
  - Built-in runtime functions (print, input, arithmetic)

  - Memory stack and function call frames

  Design:

  - Separate call stack and value stack
  - Environment model for scoped variable binding
  - Optional tail-call optimization

  C++ Features:

  - Thread-local storage, lambdas for closures
  - Coroutines or continuations if advanced control flow is desired

- **f. `/stdlib` — Standard Library Functions**

  Contains:

  - Native functions implemented in C++ and exposed to the interpreter
  - Standard math, string, and I/O functions
  - Ability to register external functions to the language

  Design:

  - Internal registry of functions keyed by name or ID
  - C++ functions mapped to native call interface

2.1.4 Build System and Tooling

The project uses **CMake** with full C++20/23 support:

- Each module has its own `CMakeLists.txt`
- Compile flags enforce modern standards: `-std=c++23`, `-Wall`, `-Wextra`, `-Werror`
- Modular build using `add_library(MODULE_NAME MODULE)` and `target_link_libraries`
- Optional Clang modules support for dependency isolation
- Precompiled headers for faster build in larger projects

2.1.5 Modern Development Practices

- **Version Control**: Git repository with submodules (for third-party or future bytecode engine)
- **Testing**:
  - Unit tests for lexer, parser, and evaluator using `Catch2` or `doctest`
  - Integration tests using language snippets in `/examples`
  - Static analysis using Clang-Tidy and `-fsanitize=address`
- **CI/CD**:
  - GitHub Actions or similar pipeline to validate builds on Linux, Windows, macOS
  - Separate jobs for unit tests, style checks, and benchmarks

2.1.6 Advantages of Using Modern C++20/23

- **Modules** allow fast and scalable builds without excessive includes
- **Concepts** ensure that interpreter components (e.g., visitors, type checkers) are valid at compile time
- **`std::variant` and `std::visit`** enable safe and expressive AST traversal
- **`consteval/constexpr`** allow building parts of the interpreter that validate or simulate code at compile-time
- **Thread-safe execution** and `std::jthread` for future concurrency in the interpreter
- **Pattern matching** (C++23) simplifies AST and token processing

2.1.7 Conclusion

This initial structure is designed to support the full life cycle of a modern language interpreter, from early experimentation to production-grade tools. Built on **modern C++ standards**, it promotes **readability, modularity, type safety**, and **performance**. Each module is intentionally decoupled, allowing independent testing and iteration.

By establishing a scalable and maintainable structure now, we ensure that future chapters—on parsing, execution, error handling, and concurrency—are built on solid, modern foundations. This architecture reflects a 2020s-era mindset of system programming: precise, modular, and maintainable—without sacrificing performance or expressiveness.

2.2 CMake for Multi-Component Interpreter Projects

In building a modern interpreter in **C++20/23**, using a robust and modular build system is essential. The complexity of a language project increases rapidly with the addition of lexical analysis, parsing, semantic checking, runtime evaluation, testing, and tools. This complexity must be managed with a clean build configuration that supports:

- Multiple submodules or libraries
- Dependency isolation
- Modern C++ standards
- Easy testing and documentation integration

This section explains how to structure a **multi-component interpreter project** using **CMake**, leveraging the latest standards and practices introduced over the past five years, including **C++20 modules**, **target-based builds**, **toolchain abstraction**, and **cross-platform compatibility**.

2.2.1 Why CMake for Language Projects?

CMake has become the standard for large-scale C++ projects due to its flexibility, IDE integration, cross-platform support, and compatibility with modern build systems like Ninja, MSBuild, and Make. It supports:

- Out-of-source builds
- Fine-grained control over dependencies and targets
- Build type separation (Debug, Release, RelWithDebInfo)

- Test integration with `CTest` or external frameworks (Catch2, doctest)
- Native support for C++20/23 features, including modules and `std::format`

2.2.2 High-Level CMake Layout for the Interpreter

Assuming a project directory named `ForgeLang`, the structure would be:

```
/ForgeLang
  CMakeLists.txt
  /cmake                    → CMake helper modules
  /src
     /core
     /lexer
     /parser
     /semantics
     /runtime
     /stdlib
     /main
  /tests
  /examples
  /build                    → Out-of-source build directory
```

2.2.3 Root `CMakeLists.txt` (Top-Level Configuration)

```
cmake_minimum_required(VERSION 3.25)
project(ForgeLang VERSION 0.1 LANGUAGES CXX)

# Require C++23
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Use folders in IDEs like Visual Studio
set_property(GLOBAL PROPERTY USE_FOLDERS ON)

# Modules path
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

# Enable warnings and options
include(cmake/CompilerWarnings.cmake)
include(cmake/EnableSanitizers.cmake)

# Add source directories
add_subdirectory(src/core)
add_subdirectory(src/lexer)
add_subdirectory(src/parser)
add_subdirectory(src/semantics)
add_subdirectory(src/runtime)
add_subdirectory(src/main)

# Add tests
enable_testing()
add_subdirectory(tests)
```

This configuration enforces modern standards, organizes the project cleanly, and integrates useful features (warnings, sanitizers, tests).

2.2.4 Example Component CMake (e.g., `src/lexer/CMakeLists.txt`)

```
add_library(ForgeLexer STATIC
    Token.cpp
    Lexer.cpp
)

target_include_directories(ForgeLexer PUBLIC ${CMAKE_SOURCE_DIR}/src)
target_link_libraries(ForgeLexer PUBLIC ForgeCore)
target_compile_features(ForgeLexer PUBLIC cxx_std_23)

set_target_properties(ForgeLexer PROPERTIES FOLDER "ForgeLang/Lexer")
```

Each component is built as a static library (or module, where supported). This improves compilation time and enforces **separation of concerns**.

2.2.5 Shared Core Module (e.g., `src/core/CMakeLists.txt`)

```
add_library(ForgeCore STATIC
    SourceManager.cpp
    ErrorReporter.cpp
    Utility.cpp
)

target_include_directories(ForgeCore PUBLIC ${CMAKE_SOURCE_DIR}/src)
target_compile_features(ForgeCore PUBLIC cxx_std_23)

# Optional sanitizers and warnings
target_link_options(ForgeCore PRIVATE ${SANITIZER_FLAGS})
```

The core library provides foundational types and services for all modules. All other components (lexer, parser, runtime) depend on it.

2.2.6 Main Executable Entry Point (`src/main/CMakeLists.txt`)

```
add_executable(ForgeLang
    main.cpp
)

target_link_libraries(ForgeLang
    PRIVATE
    ForgeCore
    ForgeLexer
    ForgeParser
    ForgeSemantics
    ForgeRuntime
)

target_compile_features(ForgeLang PUBLIC cxx_std_23)
set_target_properties(ForgeLang PROPERTIES FOLDER "ForgeLang/Main")
```

This is the main interpreter binary. Additional REPL or debugging tools can be added as separate executables under `/tools` or `/cli`.

2.2.7 Using C++20 Modules (Experimental Support)

If your compiler supports it (Clang >= 16, MSVC >= 19.34, GCC >= 13):

```
target_sources(ForgeParser
  PRIVATE
    FILE_SET cxx_modules TYPE CXX_MODULES
```

```
    BASE_DIRS ${CMAKE_CURRENT_SOURCE_DIR}
    FILES
        Parser.ixx
)
```

Modules allow replacing header files, reduce compile time, and eliminate transitive dependencies. This is especially useful for high-level interpreter APIs.

2.2.8 Testing Subsystem (`tests/CMakeLists.txt`)

```
add_executable(ForgeTests
    LexerTests.cpp
    ParserTests.cpp
)

target_link_libraries(ForgeTests PRIVATE
    ForgeCore
    ForgeLexer
    ForgeParser
)

add_test(NAME LexerTests COMMAND ForgeTests)
```

Use modern test frameworks that support C++20, such as Catch2 v3, with test discovery integration for IDEs and CI.

2.2.9 Build Commands

Assuming you're using the terminal and Ninja:

```
mkdir build && cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
ctest --output-on-failure
```

This setup supports fast iteration, cross-platform builds, and integration with tools like Clang-Tidy, Valgrind, or sanitizers.

2.2.10 IDE Integration and Tooling

CMake integrates with major IDEs:

- **Visual Studio 2022** supports C++23 and modules
- **CLion** auto-detects `compile_commands.json`
- **VSCode** with CMake Tools and C++ extensions provides excellent workflow

You can configure per-target compiler flags, debugging info, or even cross-compilation profiles using CMake presets.

2.2.11 Conclusion

Using CMake effectively for a multi-component interpreter project enables **structured development**, **fast builds**, and **extensibility**. By dividing components into standalone modules and libraries, leveraging C++20/23 features, and enforcing modern build practices, the interpreter can evolve cleanly from prototype to production.
This project structure reflects modern systems programming expectations and aligns well with the goals of the language we are creating—**clarity, safety, and modularity**.

2.3 Dependency Management — Lexer, Parser, Runtime

In building a modular interpreter using modern **C++20/23**, one of the most important engineering tasks is to carefully define and manage **dependencies** between core components: the **lexer**, **parser**, and **runtime**. These three modules form the heart of any language engine, and poor coupling among them can result in architectural rigidity, untestable code, and difficult feature expansion.
This section describes how to **architect dependencies** between these components using clear boundaries, modern C++ idioms, and compiler-level guarantees. The goal is to achieve **modular cohesion**, **minimal coupling**, and **maximum clarity**, using idiomatic C++20/23 design principles.

2.3.1 Why Dependency Management Matters in a Language Project

In an interpreter or compiler, each phase of the pipeline must **consume only what it needs**, and **produce outputs suitable for the next stage**, without relying on runtime details of other phases. Poor separation typically manifests in:

- Lexer calling parser logic (bad design)
- Runtime logic embedded into AST nodes
- Cyclical dependencies between semantic analysis and evaluation
- Direct variable sharing or implicit globals

A clear dependency model eliminates these problems and enables **unit testing**, **parallel development**, and **future extensions**, such as JIT compilation, static

analysis, or embedded REPLs.

2.3.2 High-Level Component Boundaries

The architecture follows a layered structure:
Only downward dependencies are allowed. Each layer depends only on the layer(s) below it, never above.

2.3.3 Lexer: Input Tokenization Layer

Purpose:

- Accepts a source buffer (`std::string_view`)
- Produces a linear sequence of tokens (`TokenStream`)
- Does not depend on AST, parser, or runtime

Dependencies:

- `core/SourceManager.hpp` for span tracking
- `core/ErrorReporter.hpp` for structured errors
- Standard library only

Output:

- `Token` structure with type, lexeme, source location
- `TokenStream`: typically `std::vector<Token>` or an iterator view

Modern C++ Tools:

- `std::string_view`, `std::optional`, `std::variant` for token values
- `std::source_location` (C++20) for diagnostics
- `constexpr` lexers for testing and compile-time evaluation

2.3.4 Parser: Syntax Construction Layer

Purpose:

- Receives a stream of tokens from the lexer
- Produces an Abstract Syntax Tree (AST)
- Does not call or rely on runtime evaluation

Dependencies:

- `lexer/Token.hpp`
- `core/SourceManager.hpp`
- `core/ErrorReporter.hpp`
- Internal AST definitions (`ast/Expr.hpp`, `ast/Stmt.hpp`)

Output:

- AST nodes represented using `std::variant` or algebraic types
- For example:

```
using Expr = std::variant<BinaryExpr, LiteralExpr, CallExpr, VarExpr>;
```

Design Notes:

- Use **recursive descent** with backtracking or lookahead
- Report syntax errors gracefully via `std::expected` or error accumulation

Modern C++ Tools:

- `std::visit`, `std::monostate` for variant traversal
- Concepts and constraints for AST transformations

2.3.5 Runtime: Execution Layer

Purpose:

- Receives a fully parsed and semantically valid AST
- Executes the program in an environment model
- Provides built-in functions, memory, control flow, stack

Dependencies:

- AST definitions
- Symbol table or semantic results
- `core/Environment.hpp`, `core/Value.hpp`, and runtime data structures

Key Responsibilities:

- Interprets AST using a visitor or evaluation engine
- Manages a scoped environment (stack frames, variables, closures)
- Provides built-in I/O and standard library functions

Output:

- Result of program execution (`Value`)
- Can be used for REPL or scripting interface

Modern C++ Tools:

- `std::variant` to represent runtime values:

```
using Value = std::variant<IntValue, FloatValue, BoolValue, StringValue,
↪  FunctionValue>;
```

- `std::function` or lambdas for native function calls
- `std::jthread`, `std::future`, or coroutine-based async features for concurrency (optional)

2.3.6 Example of Dependency Direction (CMake and Code)

Let's say we define libraries like this in `CMakeLists.txt`:

```
add_library(ForgeCore ...)
add_library(ForgeLexer ...)
add_library(ForgeParser ...)
add_library(ForgeRuntime ...)
```

```
# Dependencies
target_link_libraries(ForgeLexer PUBLIC ForgeCore)
target_link_libraries(ForgeParser PUBLIC ForgeLexer ForgeCore)
target_link_libraries(ForgeRuntime PUBLIC ForgeParser ForgeCore)
```

This ensures that:

- Lexer is completely independent
- Parser depends only on Lexer and Core
- Runtime depends only on Parser and Core
- No module introduces upward or cyclic dependencies

2.3.7 AST and Value Boundary

A **key architectural boundary** lies between:

- **Parser output** (AST)
- **Runtime input** (Value system)

The AST must never carry runtime values. This enforces purity and makes the AST reusable for:

- Static analysis
- Code formatting
- Type checking
- Compilation or transpilation

All runtime data is generated and stored **after parsing**, within the evaluation engine.

2.3.8 Semantic Analysis as an Optional Intermediate Layer

To maintain decoupling between syntax and execution, a **semantic layer** can act as an intermediate verifier:

- Ensures type correctness
- Infers return types
- Validates function signatures
- Annotates AST nodes with resolved types or symbol bindings

This layer produces either:

- A **typed AST** (decorated with type info)
- Or a **symbol table** used by the runtime

It can optionally cache or transform parts of the AST for optimization.

2.3.9 Runtime Extensions and Isolation

To maintain a clean runtime interface:

- Built-in functions (e.g., `print`, `input`, `len`) are registered explicitly
- External native libraries can be loaded dynamically or statically linked
- Runtime APIs are defined through interfaces and `Value` conversions

In C++20/23, this is aided by:

- `std::function<Value(const std::vector<Value>&)>` for native function wrappers

- `std::span` for passing slices safely
- Optional use of modules to isolate standard library extensions

2.3.10 Testing Each Component in Isolation

Modular dependency design enables:

- Lexer unit tests using raw source strings
- Parser tests using token streams
- AST tests using manually constructed nodes
- Runtime tests using evaluation contexts and mock environments

Example: Test just the parser:

```
Lexer lexer(source);
Parser parser(lexer.tokenize());
AST ast = parser.parse_expression();
// Assertions on structure
```

2.3.11 Conclusion

Managing dependencies between the **lexer, parser, and runtime** is a foundational engineering principle when building a modern interpreter. With modern C++20/23, developers can enforce **type-safe boundaries**, minimize coupling, and structure their interpreter as a collection of focused, testable units.
By keeping components strictly layered and avoiding runtime dependencies in syntax and analysis phases, we enable a clean, composable, and scalable language infrastructure that supports both growth and correctness.

2.4 Organizing Target Language Files (.lang, .test)

In the design and implementation of a new programming language, managing the source files written in that language is as important as implementing its components. These source files—used for testing, demonstration, validation, and bootstrapping—should follow a clear structure and naming convention that reflects the language's evolution, helps automate test suites, and supports maintainability as the language expands.

This section explains how to organize, name, and integrate **target language files**, using extensions such as `.lang` for programs and `.test` for validation scripts, into the interpreter infrastructure. The approach follows modern software practices from the past five years and complements C++20/23-powered interpreter architecture.

2.4.1 Purpose of Organizing Language Files

When building a new language, you will be writing dozens to hundreds of test programs, examples, and behavioral specifications in your target language. These files serve multiple purposes:

- **Unit testing** the interpreter or compiler
- **Behavioral documentation** for language features
- **Regression tests** to prevent breaking existing features
- **End-user examples** for tutorials and learning
- **Interactive REPL experiments** or integration scripts

A disciplined file structure with dedicated extensions allows these files to be managed, loaded, interpreted, and tested automatically.

2.4.2 Suggested File Extensions

To differentiate target-language files from C++ implementation files:

- `.lang` — for source files written in the new language
  - Used for examples, programs, benchmarks, REPL inputs
- `.test` — for test scripts containing expected results
  - Paired with `.lang` files for output-based assertions
- `.fail` — for negative tests that must produce compile-time or runtime errors
  - Used to verify robustness and error diagnostics

This naming convention makes it possible to auto-discover and categorize files by purpose.

2.4.3 Directory Layout for Target Language Files

The following is a clean and scalable directory structure:

```
/ForgeLang

  /examples               → Simple programs for demonstration
     hello_world.lang
     factorial.lang
     file_io.lang

  /tests                  → Formal test cases
     /passing
```

```
        basic_arithmetic.test
        recursion.test
        variables.test
    /failing
        missing_semicolon.fail
        type_mismatch.fail
    /edge_cases
        shadowing.test
        large_loop.test

/benchmarks             → Performance and stress examples
    compute_pi.lang
    fib_iterative.lang
```

Each test or example can be read independently by tools or manually executed using the interpreter executable.

2.4.4 .lang File Design Conventions

A `.lang` file is a complete or partial program written in your custom C-style language. During early development, programs will likely demonstrate:

- Arithmetic and logical expressions
- Function definitions and calls
- Variable binding and scoping
- Conditionals and loops

Example — `factorial.lang`:

```
fn factorial(n: int) -> int {
    if n <= 1 {
        return 1;
    }
    return n * factorial(n - 1);
}

fn main() -> int {
    let value: int = 5;
    print("Result: ", factorial(value));
    return 0;
}
```

Conventions:

- Use consistent indentation and spacing
- Use comments (`//`) for expected behavior or notes
- Keep filenames and function names descriptive and lowercase with underscores

2.4.5 .test File Format and Structure

A `.test` file serves both as a program and an assertion document. It includes embedded expectations. A simple format could be:

```
// INPUT
fn main() -> int {
    print("Hello, world!");
    return 0;
}
```

```
// EXPECT
Hello, world!
```

A test harness in C++ parses this file, extracts the **input program** and the **expected output**, runs the interpreter, and compares actual output with expectations.
This format allows test automation without a separate database or metadata schema.
Features to support:

- `// EXPECT:` for single-line outputs
- `// ERROR:` for failure cases
- Optional `// EXIT:` for expected return code

2.4.6 .fail File Format

Negative tests ensure that invalid programs are rejected. These are useful for validating:

- Type system enforcement
- Syntax checking
- Name resolution
- Semantic constraints

Example — `missing_semicolon.fail`:

```
fn main() -> int {
    let x: int = 10
    return x;
}
```

```
// ERROR
Expected ';' after variable declaration
```

Interpreter test runners must capture the diagnostic and match it with the expected `// ERROR` section.

2.4.7 Integration with the Interpreter

To support test file automation in C++:

- Create a small test runner tool (`forge-test`) in C++ that:
  - Scans `.test` and `.fail` directories
  - Parses sections into input/output pairs
  - Executes the interpreter
  - Captures stdout/stderr and return codes
  - Compares against expectations
- Use `std::filesystem` (C++17/20) to traverse directories
- Use `std::ifstream`, `std::ostringstream` for file content management
- Use `std::regex` or simple line parsing for matching `//` markers

This keeps testing infrastructure entirely in modern C++ without requiring Python or scripting languages.

2.4.8 Benefits of Organized Language Files

Well-organized target language files provide:

- **Documentation**: Each file acts as a living specification of a language feature
- **Validation**: Ensures language changes do not regress existing behavior
- **Automation**: Enables integration with CI/CD and test runners
- **Onboarding**: New developers and contributors can explore language behavior easily

Moreover, the separation of `.lang`, `.test`, and `.fail` files enables modular testing and filtering.

2.4.9 Future Expansion

As the language matures, the file system can expand to support:

- `.mod.lang` — for module or package system definitions
- `.repl.test` — for interactive session scripts
- `.compile.test` — for checking bytecode or intermediate output
- `.doc.lang` — for showcasing documentation-driven development

Eventually, a package manager or module loader can use this same file structure for official standard library distribution.

2.4.10 Conclusion

Organizing target language files with dedicated extensions and a logical directory layout ensures that your interpreter can scale in both **features** and **testing capabilities**. Using `.lang`, `.test`, and `.fail` allows automated tooling, behavioral validation, and integration with the interpreter's evolution.

This strategy reflects modern C++ development practices where **infrastructure, source content, and testing are first-class citizens**, maintained in parallel with implementation.

2.5 Milestone — Project Structure with First Experimental Language File

The culmination of architectural planning and toolchain setup is reached with this milestone: successfully compiling and running the **first experimental source file** in our new C-style programming language. This step validates our modular project structure, build system, dependency boundaries, and runtime logic. It also serves as a checkpoint that demonstrates the **interpreter pipeline is connected end-to-end**—from reading a source file to executing its semantics.
This section documents the essential components and design insights required to produce the first executable `.lang` program using our interpreter, implemented with **modern C++20/23**.

2.5.1 Context: What This Milestone Proves

By this milestone, we aim to:

- Ensure the **entire interpreter pipeline** (Lexer → Parser → Evaluator) is functional
- Verify that the **project structure** supports running user code from a file
- Confirm the runtime can process basic constructs like function calls and print statements
- Execute a small `.lang` program with correct output
- Enable a basic **CLI interface or REPL** to interpret input files

This milestone serves as a working proof-of-concept interpreter, even if only for a tiny language subset.

2.5.2 Project Structure Recap

Let's revisit the **C++ project structure**, now populated with functional components:

```
/ForgeLang
   CMakeLists.txt
   /src
      /core           → SourceManager, diagnostics, utilities
      /lexer          → Tokenization logic
      /parser         → AST construction (Expr, Stmt)
      /runtime        → Evaluator, environment, built-ins
      /main           → Entry point, CLI handling
   /examples
      hello.lang      → First experimental language program
   /tests             → Interpreter and syntax tests (optional)
   /build             → CMake out-of-source directory
```

Each module compiles independently and links via modern **target-based CMake** using `target_link_libraries()`.

2.5.3 Minimal Feature Set for First Program

The first source file (`hello.lang`) and the interpreter must support a minimal set of language features:

- Function definition and execution
- Return value from `main()`
- Built-in `print()` function
- Basic arithmetic or string literals

Example `hello.lang`:

```
fn main() -> int {
    print("Hello, world!");
    return 0;
}
```

This program:

- Defines a `main` function
- Invokes a built-in print operation
- Returns an integer value to indicate successful termination

2.5.4 Interpreter Pipeline Overview

Each stage of the interpreter must now function with working integration.

- a. **Lexer Module**
  - Tokenizes input source string into a list of `Token` structures
  - Handles keywords, punctuation, literals, identifiers
  - Reports lexical errors (if any)
- b. **Parser Module**
  - Converts token stream into an abstract syntax tree (AST)
  - Recognizes function definitions, statements, expressions
  - Associates source locations for debugging and error reporting

- c. **Runtime Module**
  - Resolves and invokes `main()`
  - Registers built-in functions (like `print`)
  - Manages environment, call frames, and execution stack
- d. **Main Executable (`forge`)**
  - Reads `.lang` file
  - Passes it through lexer → parser → evaluator
  - Returns appropriate exit code and forwards any runtime output

2.5.5 Implementation Detail: First Language Features in C++

- **Token Definition (`Token.hpp`):**

```
enum class TokenType {
    Fn, Return, Identifier, StringLiteral,
    LParen, RParen, LBrace, RBrace,
    Arrow, Semicolon, Comma,
    IntegerLiteral, Print, EndOfFile
};

struct Token {
    TokenType type;
    std::string lexeme;
    SourceSpan span;
};
```

- **AST Nodes (`Expr.hpp`, `Stmt.hpp`):**

```
struct CallExpr {
    std::string callee;
    std::vector<Expr> arguments;
};

struct PrintStmt {
    Expr value;
};

using Stmt = std::variant<PrintStmt, ReturnStmt, FunctionDecl>;
```

- **Built-in Function Handling (`Runtime.hpp`):**

```
class Runtime {
public:
    void register_builtin(const std::string& name, NativeFunction fn);
    void run(const std::vector<Stmt>& program);
};

void builtin_print(const std::vector<Value>& args) {
    std::cout << args[0].as_string() << std::endl;
}
```

2.5.6 CLI Interface (`main.cpp`)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cerr << "Usage: forge <file.lang>" << std::endl;
        return 1;
```

```
    }

    std::string source = SourceManager::read_file(argv[1]);
    auto tokens = Lexer(source).tokenize();
    auto ast = Parser(tokens).parse_program();

    Runtime runtime;
    runtime.register_builtin("print", builtin_print);
    runtime.run(ast);

    return 0;
}
```

This minimal driver program enables direct interpretation from a file using the command:

```
./forge examples/hello.lang
```

2.5.7 Output and Success Criteria

When running the above command, you should see:

```
Hello, world!
```

And the program should exit with code `0`.
Success at this stage means:

- CMake builds all modules correctly
- Runtime system is functional for minimal execution
- Diagnostic system reports errors if the `.lang` file is malformed

- The pipeline operates in isolation and composes together seamlessly

2.5.8 Testing and Next Steps

After achieving this milestone, you can begin expanding:

- Add support for variable declarations and assignments
- Introduce expression evaluation: math, logic, comparisons
- Develop `.test` files for automation
- Implement control flow: `if`, `while`, `for`

This milestone validates the **project's architecture** and marks the **transition from design to iterative language development**.

2.5.9 Conclusion

This first milestone—executing a `.lang` file through a modern C++ interpreter—demonstrates the viability of the architecture, the correctness of module boundaries, and the effectiveness of C++20/23 features in building a clean, modular interpreter. It is not just symbolic, but foundational: every future feature builds atop this working skeleton.

The interpreter is now ready for feature expansion, testing infrastructure, error reporting refinement, and language evolution—all of which will be addressed in subsequent chapters.

Chapter 3

Development Environment for Language Implementation

3.1 Setting up C++20/23 for Building Interpreters

3.1.1 Introduction

In recent years, the evolution of the C++ language has provided language designers and systems developers with powerful tools for implementing interpreters with better maintainability, modularity, and performance. C++20 and C++23 introduce features such as concepts, ranges, modules, coroutines, constexpr enhancements, pattern matching proposals, and better compile-time diagnostics. For building interpreters—especially with a modern architecture involving lexers, parsers, ASTs, symbol tables, and execution runtimes—these language improvements offer practical advantages.

This section focuses on configuring a robust C++20/23 development environment tailored for designing interpreters and domain-specific languages. It addresses compiler choices, build systems, editor setups, and the adoption of modern C++ idioms, libraries,

and tooling.

3.1.2 Compiler Requirements and Setup

- **Choosing a Modern Compiler**

  To work with the most recent features of C++20 and C++23, ensure that your compiler supports them fully or at least substantially. As of the past five years, the most reliable compilers include:

  - **GCC (10 and above for C++20, 13+ for C++23 support)**
  - **Clang (12 and above for C++20, 16+ for C++23 previews)**
  - **MSVC (Visual Studio 2019 v16.10 and above for C++20, VS2022 for C++23)**

  For interpreter development, Clang is often preferred for its detailed diagnostics, while GCC offers broad cross-platform performance. MSVC is excellent for Windows-native tooling, especially if you target Windows APIs or use tools like Visual Studio or C++Builder.

  Recommended build flags for compiling with modern C++ features:

```
g++ -std=c++23 -Wall -Wextra -Wpedantic -O2 -g your_source.cpp -o interpreter
```

3.1.3 Build Systems and Project Organization

1. **CMake for Cross-Platform Builds**

   CMake remains the dominant build system for C++ projects. With C++20 modules and newer header units becoming more prevalent, CMake has introduced

better support in versions 3.20+. CMake allows scalable multi-target projects, separating lexer/parser generation, runtime engines, and test tools efficiently.

Use modern CMake practices:

```
cmake_minimum_required(VERSION 3.26)
project(MyLangInterpreter LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(interpreter src/main.cpp src/lexer.cpp src/parser.cpp)
target_include_directories(interpreter PRIVATE include)
```

2. **Logical Folder Structure**

   A typical interpreter should have a folder layout like:

```
/MyLangInterpreter
  /src
    main.cpp
    lexer.cpp
    parser.cpp
    ast.cpp
    runtime.cpp
  /include
    lexer.hpp
    parser.hpp
    ast.hpp
    runtime.hpp
  /tests
    test_lexer.cpp
    test_parser.cpp
  /tools
```

```
    codegen.cpp
CMakeLists.txt
```

Modularity supports testing, separation of concerns, and fast iteration.

3.1.4 IDE and Editor Configuration

To work efficiently with modern C++, consider tools that support:

- Real-time syntax checking
- Integration with clang-tidy, clang-format
- Code navigation, refactoring, and debugging support
- Git integration
- **Recommended IDEs:**
  - **CLion**: Deep CMake integration, built-in Clang engine, seamless support for C++20/23
  - **Visual Studio 2022**: Best for Windows-native development, IntelliSense, and debugging
  - **VSCode with C++ Extension**: Lightweight, customizable, and multiplatform
  - **Emacs/Vim** with LSP and `clangd` for experienced users

3.1.5 Compiler Tooling and Diagnostics

- **clang-tidy and clang-format**

  Use `clang-tidy` for code quality checks. It helps detect modern best practice violations, especially in template-heavy or constexpr-heavy components like parsers or evaluators.

  Enable C++20/23 checkers:

```
clang-tidy
↪  -checks="modernize-*,readability-*,performance-*,cppcoreguidelines-*" ...
```

  Use `.clang-format` with custom rules based on LLVM or Google style, adjusted for your project's clarity needs.

3.1.6 Testing Environment for Interpreters

Unit testing is essential, especially for language features. Integrate modern C++ testing frameworks:

- **doctest**: Header-only, fast compile time, perfect for TDD
- **Catch2**: More expressive syntax, very popular in open-source projects
- **GoogleTest**: Large-scale, stable, and powerful for integration testing

Organize tests in the `/tests` folder, and compile them as separate targets. Test lexers, parsers, AST walkers, and execution logic independently.

3.1.7 C++20/23 Specific Practices Beneficial to Interpreters

1. **Concepts**

   Use concepts to constrain template-based parser generators or AST visitors:

```
template<typename T>
concept TokenStream = requires(T stream) {
    { stream.next_token() } -> std::same_as<Token>;
};
```

2. **constexpr for Compile-Time Token Tables**

   C++20 allows complex structures at compile-time. Lexer tables and parser rules can be declared `constexpr`, speeding up runtime startup and improving testability.

```
constexpr std::array<TokenRule, N> token_rules = {
  TokenRule{"+", TokenType::Plus},
  TokenRule{"-", TokenType::Minus},
  ...
};
```

3. **std::ranges and Views**

   In interpreter pipelines—like transforming token streams into filtered views—`std::ranges` enhances readability and lazy computation:

```
auto filtered = tokens | std::views::filter([](const Token& t){ return t.type
↪  != TokenType::Whitespace; });
```

4. **Modules (Experimental)**

   Start modularizing if your compiler supports C++20 modules. Begin with internal tools like:

```
module; // global module fragment
#include <string>
#include <vector>

export module lexer;
export class Lexer {
  ...
};
```

   Use cautiously until module support becomes fully stable in your toolchain.

3.1.8 Optional Tools for Interpreter Development

- **Godbolt Compiler Explorer** for checking generated assembly of critical routines
- **Valgrind / AddressSanitizer** for memory leak detection
- **Fuzzing Tools** (like libFuzzer) for input testing of lexer/parser
- **Graphviz** integration to visualize ASTs or CFGs
- **REPL Shell** using C++ coroutines or threads to test runtime interactively

3.1.9 Conclusion

Setting up a solid, modern C++20/23 environment is foundational for building a future-proof, extensible interpreter. With proper tooling, modular project layout, and modern

language features, you can accelerate development and enforce clean design practices. This preparation phase ensures that upcoming chapters—focusing on lexical analysis, parsing strategies, semantic checks, and runtime behavior—are built on a stable, maintainable, and scalable foundation.

3.2 Language Development Tools: ANTLR, LLVM Comparison

3.2.1 Introduction

As the design and implementation of a programming language require both syntactic and semantic processing, language development tools play a vital role in automating and optimizing various phases of language construction. Among the most important tools that emerged or significantly evolved over the last five years are **ANTLR** (Another Tool for Language Recognition) and **LLVM** (Low-Level Virtual Machine). Each serves different purposes but can be effectively used together or independently in the process of interpreter or compiler development.
This section provides a focused comparison of ANTLR and LLVM within the context of building an interpreter using Modern C++ (C++20/23). We highlight their design philosophy, technical integration aspects, ecosystem maturity, and their suitability for specific stages such as lexical analysis, parsing, semantic checks, intermediate representations, and execution models.

3.2.2 ANTLR (Another Tool for Language Recognition)

1. **Overview**

   ANTLR is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. Over the past five years, ANTLR 4 has continued to receive active development, making it more robust, with increased grammar expressiveness, better error recovery, and improved code generation backends.

   ANTLR is most effective in the **front-end** development stages of language

implementation:

- **Lexical Analysis** (Lexer)
- **Syntax Analysis** (Parser)
- Optional: **Parse Tree Visitor / Listener pattern**

2. **Strengths of ANTLR**

   - **Grammar-Centric Development**: Languages are defined with clean EBNF-like grammars that are human-readable and modular.
   - **Tool Independence**: Although ANTLR's runtime is in Java, it generates code for several target languages, including C++, C#, Python, and Go.
   - **Error Recovery and Reporting**: Robust diagnostics, automatic error handling, and custom exception support help detect ambiguous or malformed syntax in user code.
   - **AST Construction Simplified**: Generates parse trees with optional visitor interfaces that work well in Modern C++ environments with concepts and variants.

3. **Integration with C++20/23**

   While ANTLR primarily generates code in C++14-style for the C++ runtime, modern interpreter projects can wrap or extend the generated parser classes using modern features:

   - Use `std::variant` or `std::any` in visitor result types
   - Combine with `std::ranges` and `std::string_view` for efficient token processing

- Implement semantic validation using `concepts` to constrain rule-specific checks
- Encapsulate the parse tree traversal within modular AST builder classes using `constexpr` utilities for transformation logic

4. **Limitations**

- **Dependency on Java Tools**: ANTLR grammar files (`.g4`) are compiled using Java-based tools, requiring a JDK and Gradle/Maven if integrated in a pipeline.
- **Heavyweight Runtime**: ANTLR runtime in C++ is large compared to hand-crafted lexers and parsers.
- **Limited Direct Support for Expression Optimization or IR**: ANTLR handles parsing but leaves semantic analysis and intermediate code generation to you.

3.2.3 LLVM (Low-Level Virtual Machine)

1. **Overview**

   LLVM is a modular compiler infrastructure designed to support compile-time, link-time, runtime, and "idle-time" optimization of programs written in arbitrary programming languages. It has become the industry standard backend for building compilers, JIT engines, and high-performance interpreters. Over the last five years, LLVM has matured its support for C++20/23 with stable APIs and tooling for modern language runtimes.

   LLVM is ideal for **back-end** development in language implementation:

- **Intermediate Representation (IR) Construction**

- **Type Checking and Verification**
- **Optimization Passes**
- **Machine Code Generation or JIT Execution**

2. **Strengths of LLVM**

   - **Flexible IR Design**: LLVM IR is both human-readable and strongly typed, making it excellent for targeting from a high-level language.
   - **Powerful Optimizations**: SSA-based optimization passes improve performance with techniques like constant folding, dead code elimination, and loop unrolling.
   - **Multi-Target Code Generation**: LLVM supports code generation for multiple architectures including x86, ARM, RISC-V.
   - **JIT Compilation with ORC (On Request Compilation)**: Enables interpreters to convert hot code paths to native code dynamically using LLVM's JIT APIs.

3. **Integration with C++20/23**

   LLVM APIs are written in C++, and Modern C++ can be fully leveraged to build high-performance, modular backend pipelines:

   - Use **RAII** to manage `LLVMContext`, `Module`, and `IRBuilder` lifecycles safely.
   - Implement AST to LLVM IR converters using **`std::variant` visitors** for expression trees.
   - Use **`constexpr` AST evaluators** to fold constant expressions before emitting IR.

- Apply **coroutines** to simulate async IR generation or deferred semantic checks.
- Embed **concepts** to constrain emitter functions and support static analysis at compile-time.

4. **Challenges**

- **Complex API and Documentation**: LLVM's API is extensive and has a steep learning curve, especially for developers new to compiler backends.
- **Build System Complexity**: Integrating LLVM as a library requires careful configuration of CMake targets and correct linking of LLVM components.
- **Error Reporting and Debugging**: Diagnosing LLVM IR bugs often requires deep knowledge of internal compiler structures and passes.

3.2.4 Comparative Summary

Comparison of ANTLR and LLVM in Compiler Toolchains

Feature	**ANTLR**	**LLVM**
Role in Toolchain	Front-End (Lexer/Parser)	Back-End (IR/Codegen)
Language Style	Grammar-Based	API-Based, SSA Representation
Modern C++ Compatibility	Partial (Generated C++14), Wrappable	Full C++20/23 Support via Native API
Integration Complexity	Moderate	High

Feature	ANTLR	LLVM
Compilation Model	Static Interpreter or Tree-Walker	JIT or AOT Compilation
Best Use Case	Rapid grammar prototyping	High-performance execution
Error Diagnostics	Built-in syntax error recovery	Requires manual error management
Tool Dependencies	Requires Java for grammar compilation	Requires LLVM build + linker integration
Extensibility	Add semantic rules externally	Full control over IR and optimizations

3.2.5 When to Use ANTLR or LLVM

- **Use ANTLR if:**
  - You are prototyping syntax quickly
  - You need clear separation between grammar and logic
  - You prefer declarative language design over hand-written parsers
- **Use LLVM if:**
  - You are building a performant backend with JIT or native compilation
  - You require code optimization or multi-architecture support
  - You need fine-grained control over IR and runtime behavior

- **Use Both if:**
  - You want a hybrid pipeline: parse with ANTLR, translate AST to LLVM IR, and compile or JIT execute it.
  - You are building a staged interpreter that starts as a tree-walker and later evolves to a compiled model.

3.2.6 Conclusion

ANTLR and LLVM represent two powerful but different approaches in the language development landscape. ANTLR empowers the grammar-oriented front-end, while LLVM gives unparalleled control and performance at the back-end. For C++20/23 interpreter developers, using both strategically provides a path from clean syntax parsing to high-performance execution. Carefully balancing both tools allows for a modern, extensible, and scalable language implementation pipeline.

3.3 IDE with Custom Language Grammar Support

3.3.1 Introduction

In the lifecycle of a modern programming language—particularly one built using Modern C++—the role of development tooling becomes critical. Beyond the command-line and compiler infrastructure, **an IDE that understands the grammar and semantics of your custom language** significantly boosts productivity, usability, and the adoption of your language. Over the last five years, IDE platforms have matured to support grammar-driven development through pluggable architecture, language servers, and custom tooling extensions.

This section focuses on practical approaches to integrating **custom language grammar support into modern IDEs**. It examines Language Server Protocol (LSP), grammar-aware plugins, syntax highlighting, code completion, and diagnostic features that can be integrated into IDEs to enable your language to compete with mainstream tools.

3.3.2 The Role of IDEs in Language Design

Developing a language involves more than parsing and executing code—it includes providing a friendly editing experience for the users of your language. This includes:

- Syntax highlighting based on token types
- Real-time syntax error diagnostics
- Code completion and context-aware suggestions
- Navigation (go to definition, find usages)
- Inline documentation

- Auto formatting and linting

These features are no longer optional. A well-designed IDE experience reflects the maturity and usability of a language.

3.3.3 Language Server Protocol (LSP): The Modern Backbone

Introduced and standardized in recent years, the **Language Server Protocol (LSP)** decouples the language logic (parsing, completion, linting) from the editor. This allows a single language backend (the **language server**) to provide IDE features across multiple editors like:

- **Visual Studio Code**
- **Neovim**
- **Sublime Text**
- **Eclipse Theia**
- **JetBrains IDEs (partial LSP support via plugins)**
- **Building an LSP Server for Your Language**

  When building an LSP server in C++20/23, you can leverage:

  - **`std::variant`, `std::optional`, and `std::map`** for robust JSON-RPC protocol handling
  - **`constexpr` for compile-time grammar rules**
  - **Threaded request handlers using `std::jthread` or `std::async`**
  - **AST and symbol table integration** for semantic operations like hover, completion, and reference search

An LSP server typically handles requests such as:

- `textDocument/didOpen`
- `textDocument/didChange`
- `textDocument/completion`
- `textDocument/definition`
- `textDocument/hover`
- `textDocument/publishDiagnostics`

By integrating your ANTLR parser or custom parser, you can use your AST and symbol data to populate these responses intelligently.

3.3.4 Editors and IDEs with Strong Grammar Plugin Support

1. **Visual Studio Code (VSCode)**

   VSCode is currently the most flexible and widely adopted IDE for custom language integration. You can create a VSCode **extension** to provide:

   - **TextMate grammar-based syntax highlighting** for fast integration
   - **Tree-sitter-based parsing** for advanced real-time syntax trees
   - **LSP client integration** to connect to your language server
   - **Code snippets, diagnostics, and refactorings**

   VSCode extensions are written in TypeScript, and the language server can be written in C++, communicating via stdio or sockets.

2. **JetBrains Platform**

   JetBrains IDEs (like CLion and IntelliJ IDEA) provide a powerful plugin architecture, but integration is more complex. You must write a **custom plugin** in Java or Kotlin, defining:

   - Grammar (via BNF or regex-based parser)
   - Lexer/Parser (usually generated via Grammar-Kit)
   - PSI (Program Structure Interface) tree structure
   - Syntax highlighting, code completion, intentions, and inspections

   While more complex than VSCode, JetBrains integration offers a deeply native feel and excellent performance for large codebases.

3. **Eclipse & Theia**

   Eclipse and Theia support LSP and custom grammar via **Xtext**, a framework for building domain-specific languages with editor support. Xtext can generate both parsers and IDE integration components based on a single grammar file.

   Although Xtext is Java-centric, it can work alongside a C++ backend if the language server is integrated properly.

3.3.5 Grammar Files and Syntax Highlighting

A crucial entry point to IDE support is the **definition of grammar and token-based styling**. This typically involves:

- Defining a `.tmLanguage.json` or `.plist` file for **TextMate-style grammars** (VSCode)
- Using **ANTLR grammar files** (`.g4`) to map syntax rules to color groups

- Registering language-specific comment, string, keyword, number, operator patterns
- Declaring indentation and brace matching rules

Example token types for highlighting:

```
"tokenColors": [
  {
    "scope": "keyword.control.mylang",
    "settings": { "foreground": "#569CD6", "fontStyle": "bold" }
  },
  {
    "scope": "string.quoted.double.mylang",
    "settings": { "foreground": "#CE9178" }
  },
  ...
]
```

3.3.6 Real-Time Diagnostics and Auto-Completion

Real-time diagnostics in IDEs are often driven by:

- **Incremental parsing**
- **On-the-fly semantic checks**
- **Custom linting rules**
- **Static analysis using AST walkers**

To implement this:

- Use your parser or visitor to walk the AST after each keystroke or file change
- Report issues as JSON diagnostics with line, column, severity, and message
- Store symbols in a context-aware scope table to support real-time suggestions

For code completion:

- Collect available symbols from the current scope
- Suggest keywords or context-specific tokens
- Include function signatures, variable types, and documentation where possible

3.3.7 Formatter and Style Tools

Implementing formatting rules that conform to your language's style guides:

- Define **formatting rules** (e.g., brace placement, indentation, spacing) using a rule engine or manual AST visitor
- Integrate with LSP's `textDocument/formatting` endpoint
- Use `std::ostringstream` or templated formatters in C++20 for clean formatting output

Optionally, provide CLI tools like `myfmt` or `mylint` that can also be used in CI pipelines.

3.3.8 Embedding Interpreter into IDE for Live Evaluation

In interactive environments (REPLs or script runners), your language interpreter can be embedded directly into the IDE extension or used over sockets:

- Build a REPL module using C++ coroutines for async command processing
- Provide a `Run` or `Evaluate Selection` command from the editor
- Return execution output, errors, or logs to a side panel in real time

3.3.9 Conclusion

Building an IDE with support for your custom language grammar is no longer a luxury—it is a core part of language design. Through LSP and modern IDE extension models, you can provide users with powerful development tools: syntax highlighting, error feedback, smart completion, and runtime interaction. When coupled with the expressive power of C++20/23, your interpreter project can offer a first-class editing and execution environment that inspires trust and productivity for developers adopting your language.

3.4 Testing and Debugging Interpreters

3.4.1 Introduction

Interpreter development, particularly for a new C-style language, requires rigorous **testing and debugging** strategies. An interpreter's correctness hinges on the consistent behavior of its core components: lexer, parser, AST generator, semantic analyzer, and runtime engine. In the last five years, Modern C++ (C++20 and C++23) has introduced advanced tools and language features that empower developers to write expressive, efficient, and testable interpreters.
This section explores comprehensive methodologies for **unit testing, integration testing, runtime evaluation, and debugging**, tailored to modern interpreter development using the latest features of the C++ language standard.

3.4.2 Foundations of Interpreter Testing

Testing an interpreter goes beyond checking outputs—it must ensure that:

- The lexer correctly tokenizes all valid and invalid inputs
- The parser produces valid ASTs for syntactically correct code
- The semantic analyzer identifies type errors, scope violations, and undefined behaviors
- The runtime engine executes all constructs accurately and consistently
- Errors are reported clearly and do not crash the interpreter

To achieve this, testing must be **layered**, modular, and automated.

3.4.3 Unit Testing with Modern C++ Frameworks

1. **Recommended Frameworks**

   Modern C++ unit testing is best supported by:

   - **doctest**: Header-only, minimal overhead, excellent for TDD
   - **Catch2**: Rich syntax, expressive assertions, powerful fixtures
   - **GoogleTest**: Mature and widely used in enterprise-grade systems

2. **Unit Testing Strategies**

   - **Lexer Tests**: Input a raw source string and assert the token stream

   ```
   CHECK(tokenize("var x = 5;") == std::vector<Token>{
       {TokenType::Keyword, "var"}, {TokenType::Identifier, "x"},
       {TokenType::Operator, "="}, {TokenType::Number, "5"},
       {TokenType::Semicolon, ";"}
   });
   ```

   - **Parser Tests**: Check that a valid token stream produces the correct AST nodes
   - **Expression Evaluator Tests**: Given AST nodes, ensure they evaluate to the correct result
   - **Semantic Checks**: Simulate errors (like undefined variables) and assert error detection

3. **Use of `constexpr` and `consteval` in Tests**

   With C++20/23, many internal components (e.g., grammar tables, type systems) can be validated at compile time using `constexpr` logic:

```
constexpr bool valid = validate_syntax_rules();
static_assert(valid, "Grammar validation failed at compile time");
```

This improves early detection of logic errors and reduces runtime testing overhead.

3.4.4 Integration and Regression Testing

1. **Interpreter Behavior Tests**

   Define `.lang` or `.test` files that contain sample programs and expected output or behavior. Use your interpreter to load, execute, and compare results.

   Example format:

   ```
   # test_basic_math.lang
   print(3 + 4 * 2);
   # Expected: 11
   ```

   Automated C++ test code:

   ```
   auto result = interpreter.run("test_basic_math.lang");
   CHECK(result.output == "11");
   ```

2. **Regression Testing Suite**

   Track previously fixed bugs and test them regularly to prevent reintroduction. Organize regression tests with unique identifiers and link them to bug IDs or Git commits.

3.4.5 Debugging Strategies for Interpreters

1. **Internal Debug Logging**

   Use structured logging tools instead of raw `std::cout`. Consider:

   - `std::format` (C++20) for clean, formatted logs
   - Custom logging macros with log levels (info, warn, error)
   - Toggle runtime logs via config flags or environment variables

   Example:

```
log_debug("Parsing function '{}' with {} parameters", function_name,
↪  param_count);
```

2. **AST Visualization**

   Generate DOT/Graphviz diagrams from AST nodes to visually debug structure:

```
digraph AST {
  node0 [label="BinaryExpr +"];
  node1 [label="Literal 3"];
  node2 [label="Literal 4"];
  node0 -> node1;
  node0 -> node2;
}
```

   Use this to verify parser behavior, nesting, and tree correctness.

3. **Value Tracing**

Introduce value traces in your evaluator. For each variable or expression, print evaluation steps:

```
Evaluating: x = 5 + 2;
Token: Identifier(x)
Token: Operator(=)
SubExpression: 5 + 2 = 7
Set x = 7
```

This helps debug runtime errors without attaching external debuggers.

3.4.6 Using Modern C++ Tools for Debugging

1. **Debugging Tools**

   - **GDB or LLDB**: For line-by-line inspection of C++ code
   - **Valgrind or AddressSanitizer**: To detect memory leaks, invalid accesses
   - **Clang Static Analyzer**: Helps catch logic errors at compile time
   - **Compiler Explorer** (Godbolt): Inspect generated assembly for hot paths in runtime

2. **Assertions and Contracts**

   C++20 introduces `[[expects]]` and `[[ensures]]` (in preparation for contracts), and assert-like patterns are still helpful:

```
void execute_statement(const Statement& stmt) {
    assert(stmt.is_valid());
    // proceed
}
```

This defensive approach traps logical errors early.

3.4.7 Testing Error Handling and Edge Cases

Robust interpreters must gracefully handle:

- Syntax errors
- Type mismatches
- Divide-by-zero
- Infinite recursion or loops
- Access to undefined variables
- Invalid function calls

For each scenario, develop tests that assert both error messages and absence of crashes.

```
CHECK_THROWS_WITH(interpreter.run("x = 10 / 0;"), "RuntimeError: Division by zero");
```

3.4.8 Test Automation and Continuous Integration

In modern C++ development, integrate test automation with:

- **CMake and CTest**: Build and run tests automatically
- **GitHub Actions or GitLab CI**: Trigger tests on every commit
- **Code coverage tools** (e.g., gcov, llvm-cov): Identify untested branches in your interpreter

Sample CMake integration:

```
enable_testing()
add_executable(run_tests tests/test_runner.cpp)
add_test(NAME InterpreterTests COMMAND run_tests)
```

3.4.9 Conclusion

Testing and debugging are not side concerns in interpreter development—they are foundational to its correctness, reliability, and future extensibility. By leveraging C++20/23 features like `constexpr`, `std::format`, `std::variant`, and modern testing frameworks, you can implement a full suite of precise, maintainable, and automated tests for your interpreter components. Combining static and dynamic analysis, layered testing, and visual debugging tools ensures that your interpreter behaves consistently under all conditions and is ready for scaling into larger applications or even compilation targets in the future.

3.5 Milestone – Development Environment Ready for Interpreter Building

3.5.1 Introduction

This milestone marks the **transition from environment preparation to actual interpreter implementation**. It confirms that the foundational tools, libraries, compilers, and design scaffolding are correctly assembled to support systematic development, testing, and extension of your custom C-style interpreted language.
In the context of the latest C++ standards, this milestone reflects not only the completion of tooling setup, but also **the integration of modern development principles**, including modular code organization, compile-time validation, test automation, diagnostics, and IDE support. This section consolidates all previous work into a verified state, enabling confident progression into the core phases of interpreter construction: lexical analysis, parsing, semantic evaluation, and execution.

3.5.2 Confirmed Compiler and Language Support

Ensure that the toolchain has been validated for:

- **C++20 and C++23 language features**
  - `constexpr`, `consteval`, `concepts`, `std::ranges`, `std::format`, `coroutines`, and improved `std::variant`/`std::optional`
- **Compilers configured and tested**
  - GCC (13+), Clang (16+), MSVC (Visual Studio 2022+)
- **Successful compilation and linking of core interpreter modules**

- Lexer
- Parser
- AST structures
- Token management
- Basic runtime shell

Checklist:

- `CMakeLists.txt` targets are defined and build successfully
- Compilation tested in debug and release configurations
- Compiler flags correctly enforce warnings (`-Wall -Wextra -pedantic`)

3.5.3 Project Structure Verified and Navigable

A consistent and modular project structure is in place. Key directories are organized with clearly scoped responsibilities:

```
/MyLangInterpreter/
  /src/           # Source files for interpreter logic
  /include/       # Public and private headers
  /tests/         # Unit and integration tests
  /grammar/       # ANTLR or custom grammar definitions
  /tools/         # Utilities: AST printers, symbol checkers, REPL
  /examples/      # Example programs in .lang or .test format
  /scripts/       # Build, test, deployment scripts
```

Checklist:

- Code navigation and IntelliSense functional in chosen IDE

- Headers are discoverable by the compiler and not duplicated
- Build system supports modular compilation (e.g., per component or per feature)

3.5.4 IDE and Editor Integration Complete

The editing environment supports efficient development through:

- Syntax highlighting for the custom language grammar
- Automatic formatting and linting for C++ interpreter code
- Real-time error diagnostics and navigation
- Integration of the Language Server Protocol (LSP) for `.lang` files (optional)
- Debugging support for the interpreter in C++

Checklist:

- Visual Studio Code or CLion configured with custom syntax files
- C++ LSP features enabled: Go to definition, Find references, Rename
- Breakpoints and variable watches functional during runtime evaluation

3.5.5 Core Testing Infrastructure Operational

The test framework is fully operational with unit, integration, and regression tests implemented for:

- Tokenization
- Parsing correctness

- AST construction
- Interpreter runtime shell
- Error handling and exception diagnostics

Checklist:

- Unit tests using Catch2 or doctest compile and run
- Sample programs (in `.test` or `.lang` files) pass automated evaluation
- Edge-case behavior validated (invalid syntax, divide by zero, undefined variable)

CMake test integration:

```
enable_testing()
add_test(NAME LexerTests COMMAND interpreter_tests --test lexer)
add_test(NAME ParserTests COMMAND interpreter_tests --test parser)
add_test(NAME RuntimeTests COMMAND interpreter_tests --test runtime)
```

3.5.6 Debugging and Diagnostics Functional

Core runtime features are observable and traceable:

- Internal logs for parsing, evaluation, and symbol resolution
- AST dump functionality
- Stack trace printing for runtime exceptions
- Optional DOT/Graphviz exports for AST visual debugging

Checklist:

- AST printer or visualizer integrated and produces readable output
- Internal log system toggleable (e.g., `INTERPRETER_LOG_LEVEL=debug`)
- Value tracing or runtime execution tracing enabled for test cases

3.5.7 Optional Tools and Enhancements Ready

Advanced tools are pre-configured for deeper testing and performance tuning:

- Static analysis via `clang-tidy`, `clang-analyzer`
- Memory validation with AddressSanitizer and Valgrind
- Performance benchmarks using `std::chrono` or lightweight profilers
- Integration of Godbolt (Compiler Explorer) to inspect generated code from tight evaluation loops

Checklist:

- Static analysis runs without critical errors or warnings
- Memory leak tests on runtime modules passed
- Performance-critical paths profiled and documented

3.5.8 Initial Interpreter Command-line Interface Bootstrapped

The interpreter's initial CLI interface has been implemented and is able to:

- Read and execute `.lang` files
- Display errors and evaluation results

- Optionally start a basic REPL loop
- Return exit codes indicating success/failure

Example usage:

```
./interpreter examples/hello_world.lang
cppCopyEditint main(int argc, char* argv[]) {
    if (argc < 2) {
        start_repl();
    } else {
        run_file(argv[1]);
    }
}
```

Checklist:

- Program executes and returns correct results for known examples
- REPL starts without crash and accepts input
- Error output is readable and traceable to source

3.5.9 Final Validation: Milestone Status

Table 5-2: Project Component Status Overview

Component	Status
Compiler toolchain setup	Ready
Project folder layout	Clean and modular

Component	Status
IDE/editor configuration	Operational
Unit and integration tests	Functional
Debugging and logs	Active
Interpreter CLI	Bootstrapped
Error diagnostics	Visible and traceable

3.5.10 Conclusion

This milestone signifies the full readiness of the development environment to begin serious interpreter construction. Every critical layer—toolchain, editor, source structure, grammar integration, testing, and debugging—is validated and functional. This foundation not only accelerates development of future chapters but also establishes a professional-grade workflow that reflects modern language implementation practices with C++20 and C++23. From this point onward, design efforts can be directed toward the interpreter core: tokenization, parsing logic, semantic validation, and code execution.

Part II

Lexical Foundation

Chapter 4

Designing Tokens for the New Language

4.1 Defining Language Tokens – Keywords, Operators, Literals

4.1.1 Introduction

Tokenization is the initial and foundational step in building any interpreter or compiler. It transforms a raw source code string into a stream of **tokens**, each representing a fundamental unit of the language: keywords, identifiers, operators, delimiters, literals, and more. These tokens are consumed by the parser and directly influence syntactic and semantic analysis.

In this section, we focus on the deliberate and structured **design of tokens**—specifically the **keywords**, **operators**, and **literals**—for a modern C-style language. We employ design techniques informed by interpreter architecture and enhanced with features from Modern C++ (C++20/23), such as `std::string_view`, `constexpr`, `enum`

`class`, `std::variant`, and efficient data structures.

4.1.2 Token Categories and Their Role

A **token** consists of:

- **Type** (from a fixed enum)
- **Lexeme** (text as found in the source)
- **Optional literal value** (used for literals like numbers and strings)
- **Source location** (line/column, for diagnostics)

These components help drive parsing decisions, semantic interpretation, and error messaging.

```
struct Token {
    TokenType type;
    std::string_view lexeme;
    std::variant<std::monostate, int, double, std::string> literal;
    SourceLocation location;
};
```

4.1.3 Defining Keywords

1. **Purpose of Keywords**

   Keywords represent **reserved words** in the language. They define its structure and control flow, and cannot be redefined or used as identifiers.

   Typical keywords in a C-style language include:

```
if, else, while, for, return, break, continue, true, false, let, const,
↪  function, import, export
```

2. **Keyword Representation in C++**

   Use `enum class` to strictly define token types:

```
enum class TokenType {
    // Keywords
    If, Else, While, For, Return, Break, Continue,
    Let, Const, Function, Import, Export,
    True, False,

    // Other categories (partial listing)
    Identifier,
    ...
};
```

   Map from lexeme to token type using `constexpr` containers:

```
constexpr std::pair<std::string_view, TokenType> keyword_map[] = {
    {"if", TokenType::If}, {"else", TokenType::Else}, {"while",
    ↪  TokenType::While},
    {"return", TokenType::Return}, {"const", TokenType::Const}, {"function",
    ↪  TokenType::Function}
};
```

   Use a hash table or binary search to check for keyword matches efficiently during lexing.

4.1.4 Designing Operators

1. **Operator Categories**

   Operators define expressions and logic in the language. They are divided into:

   - **Arithmetic**: `+`, `-`, `*`, `/`, `%`
   - **Assignment**: `=`, `+=`, `-=`, `*=`, `/=`
   - **Comparison**: `==`, `!=`, `<`, `>`, `<=`, `>=`
   - **Logical**: `&&`, `||`, `!`
   - **Bitwise**: `&`, `|`, `^`, `~`, `<<`, `>>`
   - **Other**: `++`, `--`, `->`, `.`

   Optional operators for modern languages:

   - **Safe navigation**: `?.`
   - **Null coalescing**: `??`

2. **Operator Token Mapping**

   Token types for operators are also defined in the same `enum class`:

```
enum class TokenType {
    // Operators
    Plus, Minus, Star, Slash, Percent,
    Equal, EqualEqual, NotEqual,
    Less, Greater, LessEqual, GreaterEqual,
    And, Or, Not,
    PlusEqual, MinusEqual, ...
};
```

Lexer logic needs **greedy matching**:

- Match `==` before `=`
- Match `<=` before `<`
- Handle multi-character operators with a lookahead system

4.1.5 Literal Tokens

1. **Types of Literals**

   Literals are values written directly in the source code and include:

   - **Integer literals**: `42`, `-10`, `0`
   - **Floating-point literals**: `3.14`, `0.001`, `1e10`
   - **String literals**: `"hello world"`, `'a'`
   - **Boolean literals**: `true`, `false`
   - **Null literal**: `null`

2. **Literal Parsing and Storage**

   Store literals in the `std::variant` of the token structure. During tokenization:

   - Convert text to numeric using `std::stoi`, `std::stod`
   - For performance, use `std::from_chars` in C++17+, updated in C++20
   - Store string literals by stripping delimiters and handling escape sequences

```
if (std::isdigit(current)) {
    auto [value, success] = parse_number();
    tokens.push_back({TokenType::NumberLiteral, lexeme, value, location});
```

```
} else if (current == '"') {
    auto value = parse_string();
    tokens.push_back({TokenType::StringLiteral, lexeme, value, location});
}
```

`std::variant` example:

```
std::variant<std::monostate, int, double, std::string> literal;
```

4.1.6 Literal and Identifier Differentiation

Identifiers follow specific rules:

- Start with a letter or underscore
- Followed by letters, digits, or underscores

To distinguish identifiers from keywords:

```
TokenType resolve_identifier(std::string_view text) {
    for (const auto& [kw, type] : keyword_map)
        if (kw == text) return type;
    return TokenType::Identifier;
}
```

This keeps keyword recognition fast and conflict-free.

4.1.7 Language Token Table (Summary View)

Table 1-1: Token Types in Language Lexical Analysis

Token Type	Examples	Category
Keyword	`if`, `return`, `let`	Reserved words
Operator	`+`, `!=`, `==`, `&&`	Expressions
Identifier	`myVar`, `sum`, `_temp`	User-defined
Literal: Integer	`42`, `0`, `-99`	Literal
Literal: Float	`3.14`, `1e-3`	Literal
Literal: String	`"hello"`, `'a'`	Literal
Literal: Bool	`true`, `false`	Literal
Literal: Null	`null`	Special value
Delimiter	`;`, `{`, `}`, `(`, `)`	Structural

4.1.8 Modern C++ Techniques for Token Design

1. **Use of `constexpr` and `consteval`**

   Define static token tables at compile-time:

```cpp
constexpr auto is_keyword(std::string_view word) -> std::optional<TokenType> {
    for (auto [kw, type] : keyword_map) {
        if (kw == word) return type;
    }
    return std::nullopt;
}
```

2. **Use of `std::string_view`**

   Avoid memory allocations by storing lexemes as views into the source buffer.

3. **Use of `std::variant`**

   Literals can now be type-safe using `std::variant`, allowing runtime dispatch with `std::visit`.

4.1.9 Conclusion

Designing language tokens is one of the most critical architectural steps in the creation of a programming language. The token set defines the grammar's flexibility, language features, and expressiveness. Using the latest C++20 and C++23 capabilities, we can build a highly efficient, readable, and extensible token system with low memory overhead, strong type safety, and high runtime performance. With keywords, operators, and literals defined and classified, the next step in building the interpreter is implementing the lexer that consumes source code and emits these structured tokens.

4.2 C-style Syntax Design: `{}`, `;`, `()`

4.2.1 Introduction

C-style syntax has endured as a dominant syntactic tradition in programming languages due to its **clarity, conciseness, and familiarity**. Languages such as C, C++, Java, JavaScript, and Rust adopt and adapt this syntax for defining program structure, expression grouping, and control flow. When designing a new C-style interpreted language, embracing tokens such as `{}`, `;`, and `()` is both a **design choice and a strategic decision** to improve readability, reduce learning curve, and enable precise parsing.

This section focuses on defining and formalizing these structural symbols within the token set, and explains their roles in lexical and syntactic design. It also explores how modern C++20 and C++23 features enable more precise and efficient handling of these tokens during lexing and parsing phases.

4.2.2 Braces `{}`: Code Block Delimiters

1. **Purpose and Role**

   Braces `{` and `}` mark **compound statements**, **function bodies**, **control flow blocks**, and **scopes**. They indicate that the enclosed group of statements is to be treated as a single unit.

   Common usages:

   - Function definitions
   - Conditional and loop blocks
   - Scope for variables and nested structures

   Example:

```
if (x > 0) {
    print("Positive");
}
```

2. **Token Definition**

   In the lexer, define these tokens clearly and distinctly:

```
enum class TokenType {
    LeftBrace,   // {
    RightBrace,  // }
    ...
};
```

The lexer must identify braces immediately upon reading them, with no lookahead required.

3. **Parsing Considerations**

   Braces affect:

   - **Scope depth tracking**: Push and pop scope stacks in the parser or semantic analyzer
   - **Block AST nodes**: A group of statements enclosed in `{}` forms a compound node

   Example parser logic (simplified):

```
if (current_token == TokenType::LeftBrace) {
    advance();
    std::vector<Stmt> statements = parse_block();
    expect(TokenType::RightBrace);
    return BlockStmt(statements);
}
```

4. **Compiler Diagnostics Integration**

   Using `std::format` (C++20), structured brace mismatch errors can be reported:

```
throw std::runtime_error(std::format("Expected '}}' to match '{{' opened at
↪  line {}", block_start.line));
```

4.2.3 Semicolon ;: Statement Terminator

1. **Purpose and Role**

   The semicolon ; is a **statement terminator**, marking the end of a logical instruction. This is a key syntactic feature of C-style languages that allows for unambiguous parsing of sequential code.

   Examples:

```
x = 5;
print(x);
```

2. **Token Representation**

```
enum class TokenType {
    Semicolon, // ;
    ...
};
```

   Semicolons are single-character tokens and should be handled as such in the lexer without requiring backtracking or lookahead.

3. **Design Decision: Required vs. Optional Semicolons**

   You must decide:

   - Should every statement require a semicolon?

- Will certain structures (e.g., function definitions, control blocks) allow omitting it?

Recommendation for early stages: enforce semicolons strictly for predictability and simpler parsing logic. Optional semicolon handling can be revisited in later stages with contextual lookahead parsing.

4. **Parser Integration**

```
Stmt parse_expression_stmt() {
    auto expr = parse_expression();
    expect(TokenType::Semicolon);  // Enforce required `;`
    return ExpressionStmt(expr);
}
```

4.2.4 Parentheses `()`: Grouping and Control Flow Syntax

1. **Purpose and Role**

   Parentheses serve two primary purposes:

   - **Expression grouping**: Control evaluation precedence
   - **Syntax structure**: Enclose conditions or parameters in control flows and function calls

   Examples:

```
(x + y) * z
if (x > 0)
function(x, y)
```

2. **Token Representation**

```
enum class TokenType {
    LeftParen,  // (
    RightParen, // )
    ...
};
```

3. **Parsing Rules**

- **Expression grouping**: Parse recursively inside `(` and `)`
- **Function calls**: Distinguish identifier followed by `(` as a potential call
- **Control flows**: Require conditions to be parenthesized for syntactic clarity

Example:

```
Expr parse_grouping_expr() {
    expect(TokenType::LeftParen);
    auto expr = parse_expression();
    expect(TokenType::RightParen);
    return expr;
}
```

4. **Optional Design Extensions**

- Consider supporting **tuple-like expressions** in the future: `(x, y)`
- Support for **default arguments** in function calls using parentheses

4.2.5 Modern C++ Integration for Delimiter Handling

1. **`std::string_view` for Lexeme Storage**

   Avoid copying brace, paren, or semicolon characters. Use `std::string_view` to reference lexemes directly from the source buffer.

2. **`constexpr` Sets for Delimiter Recognition**

   Define quick-lookup sets at compile time:

```
constexpr std::array<char, 6> delimiters = {'(', ')', '{', '}', ';', ','};
bool is_delimiter(char ch) {
    return std::ranges::find(delimiters, ch) != delimiters.end();
}
```

3. **AST Structural Use**

   Use braces and parentheses to build structured AST nodes:

   - `{}` → `BlockStmt`
   - `()` → `GroupExpr`, `CallExpr`, or `ConditionExpr`
   - `;` → Statement boundaries

   Use `std::variant` or polymorphic class hierarchies to express these in Modern C++.

4.2.6 Error Handling and Resynchronization

Proper handling of these delimiters is essential in error recovery:

- Unmatched `{` or `(` should trigger synchronizing strategies

- Recover by skipping tokens until `}` or `;` is found depending on context

Example:

```
void synchronize() {
    while (!is_at_end()) {
        if (previous().type == TokenType::Semicolon) return;
        if (current().type == TokenType::RightBrace) return;
        advance();
    }
}
```

This allows the parser to resume meaningful work after encountering an error inside a malformed block or expression.

4.2.7 Conclusion

The correct handling of `{}`, `;`, and `()` is essential in designing a C-style language. These tokens define the structural and sequential integrity of the source code. They are easy to tokenize but carry heavy syntactic and semantic implications. Using Modern C++ features such as `std::variant`, `constexpr`, `std::string_view`, and strong typing through `enum class`, you can define a robust token system that lays a solid foundation for parsing, interpretation, and diagnostics. These structural tokens will serve as the scaffolding for all higher-level constructs, including control flow, function definitions, and nested scopes in your new language.

4.3 Custom Tokens for Our Language – Additional Features

4.3.1 Introduction

A modern C-style language should not be restricted to the token sets inherited from classic C or C++. While foundational symbols like `{}`, `;`, and `()` form the structural core, **custom tokens** enable innovation, domain-specific enhancements, and expressiveness that distinguish your language from legacy designs. Over the past five years, language designers have embraced new token conventions to support features like safe access, pattern matching, optional chaining, string interpolation, and embedded metadata. These additions can improve ergonomics, reduce boilerplate, and open new syntax possibilities.

This section explores how to design, represent, and implement **custom tokens** in your language, using Modern C++ (C++20/23) to ensure high performance, maintainability, and compile-time validation where possible.

4.3.2 Motivation for Custom Tokens

Custom tokens serve several key goals:

- Extend expressiveness and syntax clarity
- Simplify common programming idioms
- Enable future language features such as reflection, optional types, or declarative structures
- Differentiate your language while maintaining familiar C-like aesthetics

Examples of modern syntax-enhancing tokens:

- `?.` for optional (safe) member access
- `??` for null coalescing

- `=>` for concise lambdas or match expressions
- `$""` for interpolated strings
- `@` for annotations or metadata
- `#` for preprocessor-style directives or compile-time blocks

4.3.3 Designing Custom Token Types

1. **Token Type Extension**

   Extend your `TokenType` enum to include custom features:

```
enum class TokenType {
    // Custom structural tokens
    Arrow,            // =>
    QuestionDot,      // ?.
    DoubleQuestion,   // ??
    DollarString,     // $"..."

    // Annotation or metadata
    AtSymbol,         // @

    // Extended literals or markers
    Hash,             // #
    InterpolationStart, // ${
    InterpolationEnd,   // }

    // Existing tokens...
    Identifier,
    StringLiteral,
    ...
};
```

2. **Custom Lexing Rules**

   Modern C++20/23 supports `constexpr`-driven lexers or table-driven dispatching. Custom tokens often require **multi-character lookahead**, handled by explicit lexer rules.

   Example: tokenizing `=>` or `?.`

```
if (match('=') && peek() == '>') {
    advance();
    return make_token(TokenType::Arrow);
}
if (match('?') && peek() == '.') {
    advance();
    return make_token(TokenType::QuestionDot);
}
```

   Maintain clear token boundaries using `std::string_view` and non-owning references to the source buffer.

4.3.4 Custom Tokens for Language Semantics

1. **Optional Access: `?.`**

   Supports safe member access or function calls, avoiding null reference errors:

```
user?.profile?.name
```

   Lexer emits `TokenType::QuestionDot`, and the parser constructs a safe-access expression node.

2. **Null Coalescing: `??`**

Provides fallback expressions:

```
name = input ?? "Guest";
```

Parser interprets `??` as a binary operator node with short-circuit evaluation logic.

3. **Arrow Expression: `=>`**

   Used for lightweight lambda or mapping:

   ```
   (x) => x * x
   ```

   Lexer identifies `=>`, parser recognizes it as a function or match expression.

4.3.5 Interpolated String Tokens

1. **Design Syntax**

   Support strings with embedded expressions:

   ```
   let name = "World";
   print($"Hello, ${name}!");
   ```

   Lexer must:

   - Emit `DollarString` when encountering `$"`
   - Recognize and tokenize `${` as the start of an interpolation block
   - Resume normal tokenization within the string
   - Track nested string boundaries

2. **Internal Lexer Strategy**

   Use a mode-switching mechanism:

```
enum class LexerMode { Normal, Interpolation };

LexerMode mode = LexerMode::Normal;

if (current == '$' && peek() == '"') {
    advance(); advance(); // skip $"
    mode = LexerMode::Interpolation;
    return make_token(TokenType::DollarString);
}
```

   During interpolation, return a token for each `${` and `}` and delegate embedded expressions to the parser.

4.3.6 Annotations and Metadata: `@`

Used for marking functions, types, or variables with compile-time metadata:

```
@export
function run() { ... }
```

- Lexer returns `TokenType::AtSymbol`
- Parser recognizes the annotation block and attaches metadata to AST nodes
- Later stages (semantic analysis or code generation) can inspect metadata for special behavior

You may extend this to support parameters:

```
@route("/login")
function loginHandler() { ... }
```

4.3.7 Preprocessor-style Extensions:

While not traditional in interpreters, # can introduce compile-time logic:

```
#define MAX 100
#debug
```

Depending on your language philosophy, this may:

- Trigger macro-like expansions
- Enable conditional interpretation
- Mark debug regions
  Treat it as a custom directive, not tied to standard preprocessing.

4.3.8 Parser Considerations for Custom Tokens

The parser must be able to:

- Recognize custom tokens in the grammar
- Support extended expression forms (e.g., ?., ??, interpolated strings)
- Handle errors gracefully for malformed custom constructs

Using `std::variant` for AST nodes:

```
std::variant<BinaryExpr, SafeAccessExpr, NullCoalesceExpr, InterpolatedStringExpr,
↪  ...> Expr;
```

And using `concepts` to constrain visitors in C++20:

```
template<typename T>
concept Expression = requires(T expr) {
    { expr.evaluate() } -> std::same_as<Value>;
};
```

4.3.9 Compile-Time Checks for Token Set

Validate the uniqueness of custom tokens using `constexpr` tests:

```
constexpr bool validate_token_set() {
    std::set<std::string_view> seen;
    for (auto& [lexeme, type] : token_map) {
        if (!seen.insert(lexeme).second) return false;
    }
    return true;
}
static_assert(validate_token_set(), "Duplicate token lexeme found");
```

4.3.10 Conclusion

Custom tokens are a powerful design element that differentiate your language, increase expressiveness, and modernize syntax. With careful planning and disciplined implementation, they can be seamlessly integrated into the lexer and parser without complexity. Modern C++20/23 provides elegant ways to represent, recognize, and

process these tokens efficiently using `enum class`, `std::string_view`, `std::variant`, `constexpr`, and parsing concepts. Whether introducing safe access, null coalescing, or interpolated strings, these custom tokens enrich your language's syntax and enable more concise, readable, and modern code.

4.4 Implementing Token System Using Modern C++

4.4.1 Introduction

Implementing a **token system** is the first concrete stage in building a programming language after designing its syntax. It is responsible for transforming the source code string into a structured sequence of **tokens** that drive parsing, AST generation, semantic checks, and execution. Modern C++ (C++20/23) offers powerful tools to implement a token system with high performance, type safety, low memory overhead, and rich diagnostics.

This section details the implementation of a complete, extensible, and efficient **token system** using advanced C++ features such as `enum class`, `std::string_view`, `std::variant`, `constexpr`, `concepts`, and `strong typing`. This system forms the core of the **lexer** (lexical analyzer) and plays a critical role throughout the interpreter pipeline.

4.4.2 Token Type Design Using `enum class`

Use a **strongly typed enumeration** to define all possible token types. This avoids name clashes and enables type-safe switch statements.

```
enum class TokenType {
    // Keywords
    If, Else, While, For, Return, Function, Let, Const, True, False,
```

```
    // Identifiers and literals
    Identifier,
    IntegerLiteral,
    FloatLiteral,
    StringLiteral,
    BoolLiteral,
    NullLiteral,

    // Operators
    Plus, Minus, Star, Slash, Percent,
    Equal, EqualEqual, NotEqual,
    Less, LessEqual, Greater, GreaterEqual,
    And, Or, Not,
    Arrow, QuestionDot, DoubleQuestion,

    // Delimiters
    LeftParen, RightParen,
    LeftBrace, RightBrace,
    Semicolon, Comma,

    // Special
    EndOfFile,
    Invalid
};
```

Using `enum class` provides **scoped values**, enhancing code readability and preventing collisions.

4.4.3 Representing Tokens with `std::string_view` and `std::variant`

Each token consists of:

- **Type** (`TokenType`)
- **Lexeme** (`std::string_view`)
- **Literal value** (`std::variant` of literal types)
- **Source location** (line/column)

```
struct SourceLocation {
    std::size_t line;
    std::size_t column;
};

using LiteralValue = std::variant<std::monostate, int, double, bool, std::string>;

struct Token {
    TokenType type;
    std::string_view lexeme;
    LiteralValue literal;
    SourceLocation location;
};
```

Advantages of `std::string_view`:

- Avoids unnecessary memory allocations
- References a slice of the original source buffer

- Ensures efficient memory and performance usage

`std::variant` ensures type safety for literal values, replacing unsafe unions or polymorphic pointers.

4.4.4 Compile-Time Maps Using `constexpr` for Keywords

Use `constexpr` containers to check if an identifier is a keyword during tokenization:

```
constexpr std::pair<std::string_view, TokenType> keyword_map[] = {
    {"if", TokenType::If},
    {"else", TokenType::Else},
    {"return", TokenType::Return},
    {"true", TokenType::True},
    {"false", TokenType::False},
    {"null", TokenType::NullLiteral},
    {"let", TokenType::Let},
    {"const", TokenType::Const},
};

constexpr std::optional<TokenType> resolve_keyword(std::string_view word) {
    for (auto [kw, type] : keyword_map) {
        if (kw == word) return type;
    }
    return std::nullopt;
}
```

This enables the lexer to quickly and efficiently resolve identifiers into keyword tokens at compile time.

4.4.5 Token Construction and Emission

Tokens are created by the **lexer** as it scans the source input character by character.

Sample token creation:

```
Token make_token(TokenType type, std::string_view lexeme, SourceLocation location) {
    return Token{type, lexeme, std::monostate{}, location};
}
```

For literals, capture and convert the value:

```
Token make_int_token(std::string_view lexeme, SourceLocation location) {
    int value = std::stoi(std::string(lexeme));
    return Token{TokenType::IntegerLiteral, lexeme, value, location};
}
```

For string literals:

- Strip surrounding quotes
- Handle escape sequences (e.g., `\n`, `\"`)

4.4.6 Diagnostics and Token Formatting

Use `std::format` (C++20) for clear and structured debug output or error diagnostics:

```
std::string token_to_string(const Token& token) {
    return std::format("Token(type = {}, lexeme = '{}', line = {}, col = {})",
        static_cast<int>(token.type), token.lexeme, token.location.line,
        ↪  token.location.column);
}
```

This facilitates error reporting during parsing and interpreter execution.
Example error:

```
throw std::runtime_error(std::format(
    "Unexpected token '{}' at line {}, column {}",
    token.lexeme, token.location.line, token.location.column
));
```

4.4.7 Modern Iteration and Token Filters with `std::ranges`

In token processing, use **ranges** and **views** (C++20) to process filtered or transformed token streams:

```
auto identifiers = tokens | std::views::filter([](const Token& t) {
    return t.type == TokenType::Identifier;
});
```

This allows elegant expression of patterns like:

- Finding specific token types
- Filtering out whitespace or comments
- Building matchers for syntax rules

4.4.8 Error Tokens and Resilience

Introduce `TokenType::Invalid` to represent unknown or malformed sequences:

```
Token make_invalid_token(std::string_view lexeme, SourceLocation location) {
    return Token{TokenType::Invalid, lexeme, std::monostate{}, location};
}
```

This allows the lexer to **recover** from errors and continue processing, enabling better diagnostics and partial parsing.

4.4.9 Extending the Token System

The token system is **extensible** and modular. Additions include:

- Custom structural tokens: `=>`, `?.`, `??`
- Template/string interpolation markers: `${`, `"}"`
- Directive markers: `@`, `#`

For each addition:

- Extend `TokenType`
- Update lexer logic to recognize new patterns
- Add test cases for token classification and formatting

4.4.10 Testing Tokenization

Automated unit tests validate lexer output against known source strings:

```
TEST_CASE("Lexer handles integer literals") {
    Lexer lexer("42;");
    auto token = lexer.next_token();
    CHECK(token.type == TokenType::IntegerLiteral);
    CHECK(std::get<int>(token.literal) == 42);
}
```

Use modern frameworks like `doctest`, `Catch2`, or `GoogleTest`.

4.4.11 Optional: Using `concepts` for Token Validation

C++20 `concepts` help constrain generic functions and validators:

```
template<typename T>
concept TokenWithLiteral = requires(T t) {
    { t.literal } -> std::convertible_to<LiteralValue>;
};
```

This helps write utility functions that apply only to tokens with embedded values.

4.4.12 Conclusion

The token system is a core building block in your interpreter, and with Modern C++ (C++20/23), it can be implemented with **clarity, safety, and performance**. Leveraging `enum class`, `std::variant`, `std::string_view`, `constexpr`, and structured diagnostics enables an architecture that is scalable, extensible, and well-suited for future evolution. A robust token system sets the stage for a modular lexer, a maintainable parser, and a powerful semantic engine, all essential to the success of a modern interpreted language.

4.5 Hands-on Complete Language Token Set

4.5.1 Introduction

At this stage of building a modern interpreter, having a **well-defined and complete token set** is essential for a successful parsing pipeline. Tokens are the smallest meaningful units of source code—each representing a structural, syntactic, or semantic role. This section presents a **comprehensive and practical token specification**,

capturing all core, extended, and custom language constructs in a form suitable for real implementation using C++20/23.
This hands-on token set forms the core of the lexer, parser, and semantic analyzer. The structure is influenced by successful C-style languages while embracing modern extensions that improve readability, safety, and expressiveness.

4.5.2 Token Structure Recap

Each token consists of:

- **Type**: A value from the strongly typed `TokenType` enum class
- **Lexeme**: A `std::string_view` pointing to the raw source text
- **Literal value**: Optional, typed using `std::variant`
- **Source location**: For diagnostics and debugging

```
enum class TokenType;
struct Token {
    TokenType type;
    std::string_view lexeme;
    std::variant<std::monostate, int, double, bool, std::string> literal;
    SourceLocation location;
};
```

4.5.3 Complete Token Type Enumeration

Here is the full token set, organized by category, and ready to be implemented in your Modern C++ lexer.

1. **Keywords**

   Reserved words with semantic significance. Cannot be redefined.

```
enum class TokenType {
    // Control Flow
    If, Else, While, For, Break, Continue, Return,

    // Declarations
    Let, Const, Var, Function,

    // Boolean and Null Literals
    True, False, Null,

    // Modules and Metadata
    Import, Export,

    // Visibility and Attributes
    Public, Private, Static,

    // Error Recovery and Special
    EndOfFile,
    Invalid,
```

2. **Identifiers**

   - **Identifier**: User-defined names for variables, functions, types, etc.

```
    Identifier,
```

3. **Operators**

Support arithmetic, logical, bitwise, and assignment expressions.

```
// Arithmetic
    Plus,         // +
    Minus,        // -
    Star,         // *
    Slash,        // /
    Percent,      // %

    // Assignment
    Equal,        // =
    PlusEqual,    // +=
    MinusEqual,   // -=
    StarEqual,    // *=
    SlashEqual,   // /=
    PercentEqual, // %=

    // Comparison
    EqualEqual,   // ==
    NotEqual,     // !=
    Less,         // <
    LessEqual,    // <=
    Greater,      // >
    GreaterEqual, // >=

    // Logical
    And,          // &&
    Or,           // ||
    Not,          // !

    // Bitwise
    BitAnd,       // &
    BitOr,        // |
```

```
    BitXor,       // ^
    BitNot,       // ~
    ShiftLeft,    // <<
    ShiftRight,   // >>
```

4. **Advanced and Custom Operators**

   Modern syntactic features adopted from newer languages.

```
    Arrow,          // =>
    QuestionDot,    // ?.
    DoubleQuestion, // ??
```

5. **Literals**

   Represent direct values embedded in source code.

```
    IntegerLiteral,
    FloatLiteral,
    StringLiteral,
    BoolLiteral,    // from `true` and `false`
    NullLiteral,    // from `null`
```

6. **Grouping and Structure Tokens**

   These delimit source code structures.

```
    // Delimiters
    LeftParen,    // (
    RightParen,   // )
    LeftBrace,    // {
```

```
    RightBrace,    // }
    LeftBracket,   // [
    RightBracket,  // ]

    // Separators
    Comma,         // ,
    Dot,           // .
    Semicolon,     // ;
    Colon,         // :
```

7. **String Interpolation and Embedded Constructs (Optional Feature)**

   For expressive, modern string formatting:

```
    DollarString,        // $"..." - interpolated string literal
    InterpolationStart,  // ${
    InterpolationEnd     // }
```

8. **Metadata, Directives, and Special Symbols**

   Used for annotations, compiler hints, and compile-time instructions.

```
    AtSymbol,      // @
    Hash,          // #
```

4.5.4 Practical Usage in Lexer

1. **Efficient Token Recognition**

   Using `std::string_view` and `constexpr` maps, the lexer can quickly identify keywords, operators, and custom tokens.

Example for keyword recognition:

```
constexpr std::pair<std::string_view, TokenType> keyword_map[] = {
    {"if", TokenType::If},
    {"else", TokenType::Else},
    {"return", TokenType::Return},
    {"true", TokenType::True},
    {"false", TokenType::False},
    {"null", TokenType::NullLiteral},
    {"function", TokenType::Function},
    {"let", TokenType::Let},
    {"const", TokenType::Const}
};
```

2. **Token Matching Strategy**

   Multi-character tokens (e.g., `==`, `<=`, `?.`, `??`) are handled using lookahead logic:

```
if (match('=') && peek() == '=') {
    advance();
    return make_token(TokenType::EqualEqual);
}
```

   Single-character tokens are handled directly:

```
switch (current_char) {
    case '(': return make_token(TokenType::LeftParen);
    case ')': return make_token(TokenType::RightParen);
    case ';': return make_token(TokenType::Semicolon);
    // ...
}
```

4.5.5 Testing and Validation Strategy

Each token must be testable. Use unit tests for:

- Token classification
- Keyword recognition
- Operator precedence parsing (via the parser)
- Literal parsing and value interpretation
- Error token generation

Example (Catch2 or doctest):

```
TEST_CASE("Lexer recognizes basic arithmetic tokens") {
    Lexer lexer("1 + 2;");
    auto tokens = lexer.tokenize();
    CHECK(tokens[0].type == TokenType::IntegerLiteral);
    CHECK(tokens[1].type == TokenType::Plus);
    CHECK(tokens[2].type == TokenType::IntegerLiteral);
    CHECK(tokens[3].type == TokenType::Semicolon);
}
```

4.5.6 Optional: Token Table Summary for Compiler Explorer or IDE Integration

To support IDE highlighting or syntax trees, emit a structured JSON or CSV table of all defined tokens:

```
{
  "type": "Plus",
  "lexeme": "+",
  "category": "Operator",
  "precedence": 10,
  "associativity": "Left"
}
```

This is useful for LSP server generation and formatting tools.

4.5.7 Conclusion

This complete token set represents the **concrete foundation** of your interpreter's lexical layer. It is compatible with C-style syntax while offering room for modern innovation. By carefully categorizing, implementing, and testing these tokens using C++20/23 techniques such as `std::variant`, `std::string_view`, `constexpr`, and scoped enumerations, your lexer becomes more performant, maintainable, and ready for advanced parsing phases. This hands-on definition is ready to be fed into your lexer, unit tested, and evolved as your language design expands.

Chapter 5

Lexical Analyzer for C-Style Language

5.1 Reading and Analyzing New Language Code

5.1.1 Introduction

Before the parser, semantic analyzer, or runtime can operate on user code, the source text must be scanned and segmented into fundamental units—**tokens**. This process begins with the **lexical analyzer (lexer)**, whose first responsibility is to **read, traverse, and analyze raw source code** in preparation for structured interpretation. In this section, we focus on the **reading and low-level analysis of source code**, leveraging the power of Modern C++ (C++20/23) for safe, fast, and efficient implementation.

Over the last five years, best practices for implementing language front-ends have evolved significantly with the addition of `std::string_view`, ranges, coroutines, `constexpr`, and concepts. These allow the lexer to be designed in a zero-allocation,

compile-time aware, and testable fashion.

5.1.2 Reading Source Code into Memory

The lexical analyzer operates on a **source buffer**, typically loaded from a file, editor, or REPL input. The goal is to process this content as a read-only character stream.

- **Using `std::string` or `std::string_view`**

  Modern C++ favors `std::string_view` for performance and safety:

```
std::string read_source_file(const std::filesystem::path& path) {
    std::ifstream file(path);
    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();  // Full ownership
}
```

  Internally, the lexer receives the buffer as:

```
Lexer lexer(std::string_view source);
```

  This keeps the lexer **non-owning**, allowing for shared or embedded sources without copies.

5.1.3 Managing Source Navigation

Lexing requires **character-by-character traversal** with support for peeking, backtracking, and location tracking (line/column).

- **Cursor-Based Navigation**

  Maintain a character index and location metadata:

```
class Lexer {
private:
    std::string_view source;
    std::size_t current = 0;
    std::size_t line = 1;
    std::size_t column = 1;
    std::size_t start = 0;

    char advance() {
        if (at_end()) return '\0';
        char ch = source[current++];
        if (ch == '\n') {
            ++line;
            column = 1;
        } else {
            ++column;
        }
        return ch;
    }

    char peek() const {
        return at_end() ? '\0' : source[current];
    }

    char peek_next() const {
        return (current + 1 < source.size()) ? source[current + 1] : '\0';
    }

    bool at_end() const {
```

```
        return current >= source.size();
    }
};
```

Use `advance()`, `peek()`, and `match(char expected)` for character control.

5.1.4 Identifying Token Boundaries

A core lexer operation is determining **where a token starts and ends**.

- **Tracking Lexeme Start**

  Record the start of the current token:

  ```
  start = current;
  ```

  Extract the lexeme at token creation:

  ```
  std::string_view lexeme() const {
      return source.substr(start, current - start);
  }
  ```

5.1.5 Skipping Whitespace and Comments

Whitespace and comments do not produce tokens but must be recognized and skipped.

1. **Whitespace Skipping**

   Support spaces, tabs, carriage returns, and newlines:

```
void skip_whitespace() {
    while (true) {
        char c = peek();
        switch (c) {
            case ' ':
            case '\t':
            case '\r':
                advance(); break;
            case '\n':
                advance(); break;  // handled in advance()
            default:
                return;
        }
    }
}
```

2. **Line and Block Comments**

- **Line comment**: starts with `//` and continues to end of line
- **Block comment**: starts with `/*` and ends with `*/`

```
void skip_comments() {
    if (peek() == '/' && peek_next() == '/') {
        while (peek() != '\n' && !at_end()) advance();
    } else if (peek() == '/' && peek_next() == '*') {
        advance(); advance();  // Skip /*
        while (!(peek() == '*' && peek_next() == '/') && at_end())
            advance();

        if (at_end()) {
            advance(); advance();  // Skip */
```

```
        }
    }
}
```

Use `skip_whitespace()` and `skip_comments()` together before each token analysis.

5.1.6 Character Classification for Tokens

Use C++ standard functions to classify input:

```
bool is_alpha(char c) {
    return std::isalpha(static_cast<unsigned char>(c)) || c == '_';
}

bool is_digit(char c) {
    return std::isdigit(static_cast<unsigned char>(c));
}

bool is_alphanumeric(char c) {
    return is_alpha(c) || is_digit(c);
}
```

These are used to recognize identifiers, keywords, and numeric literals.

5.1.7 Token Recognition Loop

Main driver function for token stream generation:

```
std::vector<Token> tokenize() {
    std::vector<Token> tokens;
```

```
    while (!at_end()) {
        skip_whitespace();
        skip_comments();
        start = current;
        Token token = scan_token();
        if (token.type != TokenType::Invalid) {
            tokens.push_back(token);
        }
    }
    tokens.push_back(make_token(TokenType::EndOfFile));
    return tokens;
}
```

Each call to `scan_token()` consumes one token.

5.1.8 Error Detection During Reading

Lexer errors include:

- Unterminated string literals
- Invalid characters
- Unexpected end-of-file in multi-line constructs

Example error reporting:

```
throw std::runtime_error(std::format(
    "Unexpected character '{}' at line {}, column {}",
    peek(), line, column
));
```

Modern C++20 enables detailed formatting of diagnostic messages with `std::format`.

5.1.9 Token Debugging and Tracing

During development, add internal logging using structured output:

```
void debug_token(const Token& t) {
    std::cout << std::format("Token: {:<15} Lexeme: '{}' at {}:{}\n",
        to_string(t.type), t.lexeme, t.location.line, t.location.column);
}
```

This helps during parser testing and AST debugging.

5.1.10 Conclusion

Reading and analyzing the source code is the first technical stage of interpretation. In Modern C++ (C++20/23), a robust lexical analyzer can be built efficiently with safe and expressive constructs like `std::string_view`, `constexpr` functions, `std::format`, and character-based navigation logic. Accurate source tracking, clean skipping of whitespace and comments, precise error diagnostics, and lexeme classification form the foundation for the next phase: **token recognition and classification**, which will be explored in the following sections of the lexer implementation.

5.2 Recognizing C Patterns: `int x = 5;`, `if (condition) {}`

5.2.1 Introduction

Recognizing familiar **C-style syntax patterns** such as variable declarations (`int x = 5;`) and control flow constructs (`if (condition) {}`) is a crucial responsibility of a lexer tailored to a C-style interpreted language. Although these patterns will be parsed

and semantically analyzed in later stages, the lexer must first **precisely identify and segment their building blocks into tokens**, using deterministic rules rooted in both lexical specification and source structure.
Modern C++20/23 provides the tools for building a fast and safe lexical engine that can recognize these constructs by scanning one character at a time, forming token sequences such as keywords, identifiers, literals, operators, and delimiters. This section describes how a lexer processes such patterns and prepares them for the parser.

5.2.2 Decomposing the Pattern: `int x = 5;`

1. **Token Sequence Breakdown**

   For the declaration `int x = 5;`, the lexer must produce:
   Lexeme and Corresponding Token Type

Lexeme	Token Type
`int`	Keyword/Identifier
`x`	Identifier
`=`	AssignmentOperator
`5`	IntegerLiteral
`;`	Semicolon

2. **Lexer Responsibilities**

   The lexer is responsible for:

   - Recognizing the keyword `int` (from a reserved word list)
   - Distinguishing the identifier `x` (based on character rules)

- Identifying `=` as a token (possibly distinguishing from `==`)
- Parsing numeric literal `5` as an integer
- Identifying `;` as a structural token

3. **Code Example: Lexing Simple Declarations**

```
if (is_alpha(current)) {
    std::string_view word = scan_identifier();
    if (auto keyword = resolve_keyword(word)) {
        return make_token(*keyword);
    } else {
        return make_token(TokenType::Identifier);
    }
} else if (std::isdigit(current)) {
    return scan_number_literal();
} else if (match('=')) {
    if (match('=')) return make_token(TokenType::EqualEqual);
    return make_token(TokenType::Equal);
} else if (match(';')) {
    return make_token(TokenType::Semicolon);
}
```

5.2.3 Decomposing the Pattern: `if (condition) {}`

1. **Token Sequence Breakdown**

   For the conditional statement `if (condition) {}`, the lexer must emit:

Table 2-2: Lexeme to Token Type Mapping

Lexeme	Token Type
`if`	Keyword
`(`	LeftParen
`condition`	Identifier
`)`	RightParen
`{`	LeftBrace
`}`	RightBrace

2. **Keyword Detection**

   `if` is matched as a keyword using `constexpr` keyword maps:

```
constexpr std::pair<std::string_view, TokenType> keywords[] = {
    {"if", TokenType::If},
    {"else", TokenType::Else},
    {"while", TokenType::While},
    {"return", TokenType::Return}
};
```

   Lookup during scanning:

```
std::optional<TokenType> resolve_keyword(std::string_view word) {
    for (auto [kw, type] : keywords) {
        if (word == kw) return type;
    }
```

```
    return std::nullopt;
}
```

5.2.4 Identifier and Literal Pattern Recognition

1. **Identifiers**

   Identifiers follow a C-style naming convention:

   - Start with a letter or underscore
   - Continue with letters, digits, or underscores

```
std::string_view scan_identifier() {
    while (is_alphanumeric(peek())) advance();
    return source.substr(start, current - start);
}
```

   If the scanned identifier matches a keyword, assign a keyword token. Otherwise, assign `TokenType::Identifier`.

2. **Numeric Literals**

   Numerical detection follows digit rules:

```
Token scan_number_literal() {
    while (std::isdigit(peek())) advance();
    if (peek() == '.' && std::isdigit(peek_next())) {
        advance(); // Consume '.'
        while (std::isdigit(peek())) advance();
        return make_token(TokenType::FloatLiteral);
```

```
    }
    return make_token(TokenType::IntegerLiteral);
}
```

This approach supports both integer and floating-point literals such as `5`, `3.14`.

5.2.5 Delimiters and Operators

The lexer must identify single-character tokens immediately:

```
switch (current) {
    case '(': return make_token(TokenType::LeftParen);
    case ')': return make_token(TokenType::RightParen);
    case '{': return make_token(TokenType::LeftBrace);
    case '}': return make_token(TokenType::RightBrace);
    case '=':
        return match('=') ? make_token(TokenType::EqualEqual) :
        ↪  make_token(TokenType::Equal);
    case ';': return make_token(TokenType::Semicolon);
}
```

These structural tokens are critical in recognizing syntax forms like:

- Grouped expressions
- Block statements
- Statement endings

5.2.6 Recognizing Common Code Patterns

The lexer, though not responsible for parsing structure, indirectly supports **pattern recognition** through the sequence and classification of tokens.

Examples:

- **Variable declarations**: Recognized by `Keyword Identifier Equal Literal Semicolon`
- **Function calls**: `Identifier LeftParen [args] RightParen`
- **Conditionals**: `If LeftParen Expression RightParen Block`

The lexer's accuracy ensures the parser receives meaningful, complete tokens to apply grammatical rules.

5.2.7 Diagnostic and Error Traps

A robust lexer should gracefully report malformed versions of these patterns:

- Missing semicolons
- Unterminated parentheses or braces
- Unexpected characters

Example diagnostic:

```
if (peek() == '{' && !match_closing_brace()) {
    throw std::runtime_error(std::format(
        "Unmatched opening brace at line {}, column {}", line, column
    ));
}
```

5.2.8 Efficient Token Stream Assembly

Tokens are collected in order and can be consumed by the parser:

```
std::vector<Token> tokenize() {
    std::vector<Token> tokens;
    while (!at_end()) {
        skip_whitespace();
        skip_comments();
        start = current;
        Token token = scan_token();
        tokens.push_back(token);
    }
    tokens.push_back(make_token(TokenType::EndOfFile));
    return tokens;
}
```

This process ensures sequences like `int x = 5;` and `if (x) {}` become fully analyzable token lists.

5.2.9 Conclusion

Recognizing core C-style patterns such as declarations (`int x = 5;`) and control flow constructs (`if (x) {}`) is fundamental to interpreting and parsing code. The lexer must classify each component precisely, tokenize structural characters, and resolve keywords using fast and safe Modern C++ mechanisms. Using features like `std::string_view`, `constexpr` keyword maps, and scoped enumerations ensures your lexer is not only correct but performant, ready to support the full language grammar in the parsing phase that follows.

5.3 Handling C-style Comments: `//` and `/* */`

5.3.1 Introduction

Comments are an essential part of any programming language, and a C-style language is expected to support both **single-line (`//`)** and **multi-line (`/\* \*/`)** comments. While comments are ignored by the compiler or interpreter during execution, they must be properly handled by the **lexer** to avoid corrupting the token stream.
Modern C++ (C++20/23) gives us advanced tools—such as `std::string_view`, `constexpr`, and clean class design patterns—to implement **efficient, safe, and expressive comment handling logic** in our lexical analyzer. In this section, we concentrate on implementing accurate and robust logic to **skip comments**, track line/column metadata, and optionally preserve comments for documentation or code generation if required.

5.3.2 Types of Comments in C-style Languages

1. **Single-Line Comments: `//`**
    - Starts with two forward slashes `//`
    - Terminates at the end of the line or end of file
    - Commonly used for brief inline annotations or debugging notes
2. **Multi-Line Comments: `/* */`**
    - Begins with `/*` and ends with `*/`
    - May span multiple lines
    - Can include line breaks, whitespace, symbols, and embedded text
    - Cannot be nested (in standard C-style behavior)

5.3.3 Design Principles for Comment Handling in the Lexer

1. **Comments Should Not Emit Tokens**

   Comments are not semantic content—they are meant to be **skipped entirely** during tokenization unless explicitly preserved for documentation.

2. **Accurate Line and Column Tracking**

   For multi-line comments, **line number advancement must be tracked** accurately to preserve correct error diagnostics for subsequent tokens.

3. **Robust Termination Detection**

   - Single-line comments: terminate on `\n` or EOF
   - Multi-line comments: must detect `*/` safely
   - Unterminated block comments should trigger a **clear and recoverable error**

5.3.4 Integration into the Lexical Scanning Loop

In the lexer's main `scan_token()` loop or `advance()` pipeline, comment detection is often embedded within a **`skip_whitespace_and_comments()`** method.

```
void Lexer::skip_whitespace_and_comments() {
    while (!at_end()) {
        char c = peek();

        switch (c) {
            case ' ':
            case '\t':
            case '\r':
```

```
                advance(); break;

            case '\n':
                advance();
                ++line;
                column = 1;
                break;

            case '/':
                if (peek_next() == '/') {
                    skip_single_line_comment();
                } else if (peek_next() == '*') {
                    skip_multi_line_comment();
                } else {
                    return;  // Not a comment
                }
                break;

            default:
                return;  // Not whitespace or comment
        }
    }
}
```

5.3.5 Implementing Single-Line Comments

```
void Lexer::skip_single_line_comment() {
    advance(); // consume first '/'
    advance(); // consume second '/'

    while (peek() != '\n' && !at_end()) {
```

```
        advance(); // consume characters until newline or EOF
    }
}
```

Notes:

- Do not advance past the newline; leave it for the main loop to handle line counting.
- `std::string_view` ensures no copying is required.
- Optional: store comment content for documentation extraction.

5.3.6 Implementing Multi-Line Comments

```
void Lexer::skip_multi_line_comment() {
    advance(); // consume '/'
    advance(); // consume '*'

    while (!at_end()) {
        if (peek() == '*' && peek_next() == '/') {
            advance(); advance(); // consume '*/'
            return;
        }

        if (peek() == '\n') {
            ++line;
            column = 1;
        }

        advance(); // consume comment content
```

```
    }

    throw std::runtime_error(std::format(
        "Unterminated multi-line comment starting at line {}, column {}",
        start_location.line, start_location.column
    ));
}
```

Key points:

- Line counting inside block comments must be accurate
- If end of file is reached without finding `*/`, emit an error
- `start_location` is recorded when `/*` is first encountered

5.3.7 Optional Enhancements

1. **Comment Collection for Documentation**

   Instead of discarding comment content entirely, consider buffering it for:

   - Syntax-aware editors or IDEs
   - Code documentation extraction (e.g., for `///` style comments)

   Example structure:

   ```
   struct Comment {
       std::string_view text;
       SourceLocation location;
   };
   ```

   Maintain a `std::vector<Comment>` if needed.

2. **Suppressing Certain Warnings**

   Use tagged comments to disable features:

```
// @no-lint
/* @ignore-coverage */
```

   Lexer can optionally emit **meta tokens** or attach flags to token streams.

5.3.8 Edge Case Handling

- **Nested Multi-Line Comments**

  Standard C does **not** support nested `/* ... */` blocks. Your lexer must reject nested patterns like:

```
/* Outer start
   /* Inner start */
   Outer end */
```

  To detect this:

  – Set a depth counter

  – If nesting is disallowed, throw an error when encountering a second `/*` before closing `*/`

  Alternatively, support nesting by maintaining a `depth++/--` counter—similar to Rust.

5.3.9 Unit Testing Comment Handling

Test the comment system independently:

```
TEST_CASE("Single-line comment is skipped") {
    Lexer lexer("// this is a comment\nlet x = 5;");
    auto tokens = lexer.tokenize();
    CHECK(tokens[0].type == TokenType::Let);
}

TEST_CASE("Multi-line comment is skipped") {
    Lexer lexer("/* comment */ const x = 10;");
    auto tokens = lexer.tokenize();
    CHECK(tokens[0].type == TokenType::Const);
}

TEST_CASE("Unterminated multi-line comment throws error") {
    Lexer lexer("/* unterminated...");
    CHECK_THROWS_AS(lexer.tokenize(), std::runtime_error);
}
```

5.3.10 Conclusion

C-style comment handling is a fundamental responsibility of a lexical analyzer. By implementing clean and efficient skipping logic for both `//` and `/* */` forms, while maintaining precise line tracking and optional comment preservation, you enhance the robustness of your interpreter's front-end. Using Modern C++ techniques—like `std::string_view`, `std::format`, and scoped logic with early returns—ensures performance and readability. Handling comments correctly not only avoids parsing errors but also supports future extensibility in tooling, documentation, and IDE integration.

5.4 Managing Syntax Errors in Source Code

5.4.1 Introduction

One of the most critical functions of a lexical analyzer is its ability to **detect, manage, and report syntax errors** during the early stages of language processing. Although full syntactic validation belongs to the parser, the **lexer plays a vital role in catching malformed constructs**, invalid characters, unterminated literals, or corrupted delimiters at the character level before they reach higher layers.
In this section, we explore **error detection and recovery mechanisms** within the lexer using Modern C++ (C++20/23), ensuring that the lexical analyzer not only fails gracefully but provides **actionable diagnostics** and **resilience** for continued parsing. We also discuss strategies to balance strict enforcement with developer-friendly behavior.

5.4.2 What is a Syntax Error at the Lexical Level?

Syntax errors in the lexer typically include:

- Unexpected or invalid characters
- Unterminated string or character literals
- Unterminated block comments
- Ill-formed numeric literals
- Invalid token sequences (`@#`, `!?=`, etc.)
- Unexpected end of input in middle of token

These errors prevent the lexer from producing a valid token and must be handled carefully to avoid corrupting the token stream.

5.4.3 Designing an Error Reporting System

To manage errors, the lexer should include:

- **Error reporting mechanism** (`report_error()`)
- **Location metadata** (`line`, `column`)
- **Custom exception types or error containers**
- Optional: **non-terminating recovery mode**

1. **Example: Error Reporting Function**

```
void report_error(const std::string_view message, SourceLocation loc) {
    std::cerr << std::format("[Lexer Error] Line {}, Column {}: {}\n",
        loc.line, loc.column, message);
}
```

2. **Exception Variant (for strict mode)**

```
struct LexerError : public std::runtime_error {
    SourceLocation location;
    LexerError(const std::string& msg, SourceLocation loc)
        : std::runtime_error(msg), location(loc) {}
};
```

5.4.4 Detecting Specific Lexical Errors

1. **Invalid Characters**

   Characters not part of the language character set should be flagged:

```
if (!is_valid_character(current)) {
    throw LexerError("Invalid character in source code", current_location());
}
```

   Define `is_valid_character()` to allow only letters, digits, punctuation, and allowed symbols.

2. **Unterminated String Literals**

```
Token Lexer::scan_string_literal() {
    while (peek() != '"' && !at_end()) {
        if (peek() == '\n') {
            ++line; column = 1;
        }
        advance();
    }

    if (at_end()) {
        throw LexerError("Unterminated string literal", start_location());
    }

    advance(); // Consume closing quote
    return make_token(TokenType::StringLiteral, extract_lexeme());
}
```

3. **Unterminated Multi-line Comments**

```
void Lexer::skip_multi_line_comment() {
    advance(); // consume '/'
    advance(); // consume '*'

    while (!at_end()) {
        if (peek() == '*' && peek_next() == '/') {
            advance(); advance();
            return;
        }

        if (peek() == '\n') {
            ++line; column = 1;
        }

        advance();
    }

    throw LexerError("Unterminated multi-line comment",
    ↪  comment_start_location);
}
```

4. **Ill-Formed Numbers**

```
Token Lexer::scan_number() {
    bool is_float = false;
    while (std::isdigit(peek())) advance();

    if (peek() == '.' && std::isdigit(peek_next())) {
        is_float = true;
        advance(); // consume '.'
        while (std::isdigit(peek())) advance();
    }
```

```
    std::string_view lexeme = extract_lexeme();
    try {
        if (is_float)
            return make_token(TokenType::FloatLiteral,
            ↪  std::stod(std::string(lexeme)));
        else
            return make_token(TokenType::IntegerLiteral,
            ↪  std::stoi(std::string(lexeme)));
    } catch (...) {
        throw LexerError("Malformed numeric literal", token_start_location);
    }
}
```

5.4.5 Error Token Strategy (Optional for Recovery)

Instead of halting the program, some interpreters emit an **invalid token** to allow continued processing:

```
Token Lexer::make_error_token(const std::string_view msg) {
    report_error(msg, current_location());
    return Token{TokenType::Invalid, extract_lexeme(), std::monostate{},
    ↪  current_location()};
}
```

The parser can skip over `TokenType::Invalid` and attempt recovery.

5.4.6 Line and Column Tracking for Error Reporting

Always maintain `line` and `column` counters to provide precise diagnostics:

```
char Lexer::advance() {
    char ch = source[current++];
    if (ch == '\n') {
        ++line;
        column = 1;
    } else {
        ++column;
    }
    return ch;
}
```

Attach `SourceLocation` to each token and error.

```
struct SourceLocation {
    std::size_t line;
    std::size_t column;
};
```

5.4.7 Modern C++ Features for Error Reporting

1. **`std::format` for Cleaner Error Messages**

   ```
   throw LexerError(std::format("Unexpected token '{}' at line {}, column {}",
       current_char, line, column), current_location());
   ```

2. **`std::expected` and `std::optional` (C++23)**

   In non-throwing lexers, return expected values:

```
std::expected<Token, LexerError> scan_token() {
    if (invalid_input()) {
        return std::unexpected(LexerError("Invalid", current_location()));
    }
    return valid_token;
}
```

5.4.8 Error Resynchronization Techniques

In a resilient lexer, consider skipping to the next semicolon or newline after an error:

```
void Lexer::synchronize_after_error() {
    while (!at_end()) {
        if (peek() == ';' || peek() == '\n') {
            advance();
            return;
        }
        advance();
    }
}
```

This allows the parser to continue from a known stable boundary.

5.4.9 Unit Testing Lexical Errors

Always write tests that:

- Expect errors with valid messages
- Catch unterminated strings, invalid symbols, and malformed numbers

```
TEST_CASE("Lexer detects unterminated string") {
    Lexer lexer("\"Hello");
    CHECK_THROWS_WITH(lexer.tokenize(), Catch::Contains("Unterminated string"));
}

TEST_CASE("Lexer detects invalid character") {
    Lexer lexer("@@@");
    CHECK_THROWS_WITH(lexer.tokenize(), Catch::Contains("Invalid character"));
}
```

5.4.10 Conclusion

Managing syntax errors at the lexical level is essential for building a reliable interpreter front-end. A lexer should **not only detect malformed code** but also provide meaningful, well-formatted diagnostics, and, where applicable, **attempt to recover** to continue analysis. Using C++20/23 features like `std::format`, scoped enums, `std::expected`, and structured error types, you can implement a **robust and modern error-handling system** that enhances language usability, simplifies debugging, and forms a stable base for higher-level grammar validation.

5.5 Milestone — Analyzer That Reads New Language Files

5.5.1 Introduction

This milestone represents a foundational achievement: the lexical analyzer can now **read, scan, tokenize, and validate entire source files written in the new language**, using a fully operational and error-aware lexer engine. This section

consolidates everything developed thus far—from token definitions and source navigation to comment handling and syntax error detection—into a robust, testable module.

With this functionality, the language implementation reaches a new level of maturity: it can **process any `.lang` file**, tokenize valid content, identify and report lexical errors, and produce structured output suitable for parsing and interpretation. This section describes the key capabilities and internal design of this lexical milestone using C++20/23 techniques, and verifies it with practical insights from recent compiler front-end development standards.

5.5.2 Objectives of This Milestone

The lexical analyzer must now:

- Load complete language source files (`.lang`, `.test`, etc.)
- Traverse and process input from start to finish
- Produce a **stream of valid tokens** conforming to the language specification
- Handle and report **comments, whitespace, and lexical errors**
- Track **source location metadata** (line/column/file)
- Output token diagnostics or JSON-compatible summaries (for toolchains or IDEs)

5.5.3 File-Based Input Handling

Using `std::ifstream` and `std::filesystem` to safely and portably read source code:

```
std::string read_file(const std::filesystem::path& path) {
    std::ifstream file(path);
    if (!file.is_open()) {
        throw std::runtime_error("Failed to open source file.");
    }

    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}
```

Pass the result as `std::string_view` to avoid unnecessary copies:

```
std::string source = read_file("example.lang");
Lexer lexer(source);
auto tokens = lexer.tokenize();
```

5.5.4 Tokenization Pipeline: Lexer Operational Flow

Finalized architecture of the lexer for file-based input:

```
std::vector<Token> Lexer::tokenize() {
    std::vector<Token> tokens;
    while (!at_end()) {
        skip_whitespace_and_comments();
        start = current;
        try {
            Token token = scan_token();
            tokens.push_back(token);
        } catch (const LexerError& err) {
            report_error(err.message, err.location);
```

```
            synchronize();  // optional for recovery
        }
    }
    tokens.push_back(make_token(TokenType::EndOfFile));
    return tokens;
}
```

This structure now includes:

- **Whitespace skipping**
- **Comment skipping** (single-line and block)
- **Token scanning dispatch** based on character class
- **Lexical error handling with location tracking**
- Optional recovery and resynchronization

5.5.5 Token Output: Readable and Structured

Output each token with details:

```
void print_token(const Token& token) {
    std::cout << std::format("Token({:<15}): '{:<10}' at line {}, column {}\n",
        to_string(token.type), token.lexeme, token.location.line,
        ↪  token.location.column);
}
```

Example output:

```
Token(Keyword        ): 'let'       at line 1, column 1
Token(Identifier     ): 'x'         at line 1, column 5
Token(Equal          ): '='         at line 1, column 7
Token(IntegerLiteral ): '10'        at line 1, column 9
Token(Semicolon      ): ';'         at line 1, column 11
```

This gives developers and tools immediate feedback about the token stream.

5.5.6 Source Location Tracking

Line and column information is attached to every token:

```
struct SourceLocation {
    std::size_t line;
    std::size_t column;
};

Token make_token(TokenType type) {
    return Token{type, current_lexeme(), {}, SourceLocation{line, column}};
}
```

Error messages include precise source locations using `std::format`.

5.5.7 Syntax Error Reporting During Tokenization

The lexer is now capable of reporting syntax errors such as:

- Invalid characters
- Unterminated string literals
- Unterminated block comments

- Invalid numeric formats

```
throw LexerError("Unterminated string", SourceLocation{line, column});
```

Example runtime output:

```
[Lexer Error] Line 4, Column 17: Unterminated string
```

5.5.8 Unit and Integration Testing

Create full `.lang` test files and validate the output:

```
// test_basic.lang
let x = 42;
if (x > 10) {
    print("High");
}
```

Test code using Catch2 or doctest:

```
TEST_CASE("Full source file tokenization") {
    std::string source = read_file("test_basic.lang");
    Lexer lexer(source);
    auto tokens = lexer.tokenize();
    REQUIRE(tokens.back().type == TokenType::EndOfFile);
}
```

5.5.9 Optional Output Format: JSON / Token Stream Dump

To support IDEs, syntax highlighters, or test runners, output can be serialized:

```
[
  { "type": "Let", "lexeme": "let", "line": 1, "column": 1 },
  { "type": "Identifier", "lexeme": "x", "line": 1, "column": 5 },
  ...
]
```

This can be generated using `std::format` or `nlohmann::json` if needed in a tooling module.

5.5.10 CLI Tool Integration: Language Analyzer

Compile the lexer as a command-line tool:

```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: analyzer <source.lang>\n";
        return 1;
    }

    try {
        std::string source = read_file(argv[1]);
        Lexer lexer(source);
        auto tokens = lexer.tokenize();

        for (const auto& t : tokens)
            print_token(t);
    } catch (const std::exception& ex) {
        std::cerr << "[Error] " << ex.what() << '\n';
        return 2;
    }

    return 0;
```

```
}
```

Usage:

```
./analyzer my_program.lang
```

5.5.11 Summary: What This Milestone Confirms

- The lexer accepts `.lang` files as input
- It can handle C-style syntax with structured tokens
- It skips comments and whitespace cleanly
- It tracks and reports line and column info for each token
- It detects and reports common lexical errors
- It outputs a stream of tokens usable by the parser
- It is fully testable and scriptable

5.5.12 Conclusion

This milestone marks the **completion of the lexical foundation** for your new C-style programming language. You now have a functional, extensible, and production-ready lexical analyzer that can process real code files, tokenize them accurately, and handle errors with precise diagnostics. The integration of Modern C++ features—such as `std::string_view`, `std::format`, and modular error types—ensures a high-performance and maintainable architecture. With this layer validated, the next phase is the construction of the **parser**, where the token stream will be transformed into a meaningful abstract syntax tree.

Chapter 6

REPL for the New Language – Version 1

6.1 Interactive Loop for Writing New Language Code

6.1.1 Introduction

A **REPL (Read-Eval-Print Loop)** is a core tool in the early development of any new interpreted language. It offers a fast and interactive environment for developers to write, test, and refine language features. In this first version of the REPL for your C-style language, the goal is to establish a **minimal, responsive, and testable loop** that allows users to input lines of code, tokenize them, and inspect the output in real-time.

This section focuses on building the REPL's **interactive loop** using Modern C++20/23, covering stream input handling, immediate feedback, integration with the lexer, and foundational design considerations for future extension into parsing and evaluation.

6.1.2 What is a REPL?

REPL stands for:

- **Read**: Accept user input (source code lines)
- **Eval**: Process the input (lexical analysis in this version)
- **Print**: Show the output (tokens or error messages)
- **Loop**: Return to reading more input

In early development phases, the REPL serves as a **lexical playground**. As the parser and runtime are added, it becomes the primary testing interface for language behavior.

6.1.3 REPL Architecture: Minimal Form

Functional steps of the REPL v1:

1. Prompt the user
2. Read a line from standard input
3. Pass it to the lexer
4. Display the resulting tokens
5. Repeat

6.1.4 C++20/23 Implementation: Basic REPL Loop

```
void start_repl() {
    std::string line;
    std::cout << "Welcome to LangREPL v1\nType 'exit' or Ctrl+D to quit.\n\n";

    while (true) {
        std::cout << ">>> ";
        if (!std::getline(std::cin, line)) break;

        if (line == "exit") break;
        if (line.empty()) continue;

        try {
            Lexer lexer(line);
            auto tokens = lexer.tokenize();
            for (const auto& token : tokens) {
                print_token(token);
            }
        } catch (const LexerError& err) {
            std::cerr << std::format("[Lexer Error] {} at line {}, column {}\n",
                err.what(), err.location.line, err.location.column);
        }
    }

    std::cout << "\nSession ended.\n";
}
```

6.1.5 Design Features and C++20/23 Enhancements

1. **`std::getline` for Safe Line Input**

   Modern, buffer-safe, and non-blocking input mechanism.

2. **`std::string_view` in Lexer**

   The line is passed into the lexer as a `std::string_view`, avoiding heap copies:

```
Lexer lexer(std::string_view source);
```

3. **Formatted Token Output with `std::format`**

```
void print_token(const Token& token) {
    std::cout << std::format("Token({:<15}): '{}'\n",
        to_string(token.type), token.lexeme);
}
```

6.1.6 Token Buffering and Line Tracking

Although each REPL line is standalone in version 1, internal structures are already designed to support **token sequences and source locations**:

```
struct Token {
    TokenType type;
    std::string_view lexeme;
    std::variant<std::monostate, int, double, bool, std::string> literal;
    SourceLocation location; // line = 1, column = calculated
};
```

In REPL mode, set the initial line to 1 and column tracking within each line.

6.1.7 Handling Syntax Errors in REPL

REPLs must recover from errors without terminating the session. The lexer should:

- Catch unterminated strings
- Detect invalid characters
- Report errors clearly

Errors should not exit the loop:

```
catch (const LexerError& err) {
    std::cerr << std::format("X Error: {} at line {}, col {}\n",
        err.what(), err.location.line, err.location.column);
}
```

6.1.8 Sample Interaction Output

```
Welcome to LangREPL v1
Type 'exit' or Ctrl+D to quit.

>>> let x = 5;
Token(Let           ): 'let'
Token(Identifier    ): 'x'
Token(Equal         ): '='
Token(IntegerLiteral): '5'
Token(Semicolon     ): ';'

>>> if (x > 0) { print("Hi"); }
Token(If            ): 'if'
Token(LeftParen     ): '('
Token(Identifier    ): 'x'
Token(Greater       ): '>'
Token(IntegerLiteral): '0'
```

```
Token(RightParen    ): ')'
Token(LeftBrace     ): '{'
Token(Identifier    ): 'print'
Token(LeftParen     ): '('
Token(StringLiteral ): '"Hi"'
Token(RightParen    ): ')'
Token(Semicolon     ): ';'
Token(RightBrace    ): '}'

>>> "unterminated
X Error: Unterminated string at line 1, col 1
```

6.1.9 Preparing for REPL Expansion

While REPL v1 only does lexical analysis, the following design decisions future-proof it:

- Token buffer is reusable by a parser
- Error handling already structured for semantic layers
- Integration with modules (e.g., import statements) possible
- Persistent state engine can be added to support variable definitions and execution

6.1.10 Optional: Line History and Scripting

Advanced REPLs often support:

- Command history using `readline` (optional)
- Multiline input for functions or blocks

- Saving and loading REPL sessions (`.repl` files)

These features can be added after full parser and evaluator integration.

6.1.11 Summary of Achievements in REPL v1

Table 1-1: Lexical System Capabilities

Capability	**Status**
Interactive line input	Ready
Real-time token generation	Functional
Structured output using `std::format`	T
Error detection and recovery	T
Integration with lexer module	T
Supports all token types	T
Extendable for parser/evaluator	T

6.1.12 Conclusion

The first version of your REPL forms the **interactive foundation** for experimenting with the language syntax and validating the token system in real time. It enables immediate feedback, streamlines development, and prepares the environment for the integration of parsing and execution. Built entirely with Modern C++ features, this REPL is both minimal and future-ready, embodying best practices for developing clean and extensible tooling for a new interpreted language.

6.2 Displaying Tokens Extracted from Code

6.2.1 Introduction

In the first version of a REPL (Read-Eval-Print Loop), the primary purpose is to **transform user input into a meaningful sequence of tokens** and then **display them clearly**. This provides developers and language designers with immediate feedback about how the lexer is interpreting each piece of input.
This section focuses on building a **structured, informative, and visually readable token display system**, showing not only the token types but also their raw text (lexemes), optional literal values, and source locations when applicable. We explore how to format this output using modern C++20/23 features like `std::format`, `enum class`, `std::variant`, and functional printing strategies to support both development and debugging.

6.2.2 Purpose of Token Display in REPL

Displaying extracted tokens in REPL helps:

- Verify correct tokenization of keywords, identifiers, literals, and symbols
- Detect early design flaws in the token set
- Spot lexical errors or ambiguities
- Understand token-level transformations during language experimentation
- Serve as a foundational diagnostic tool for parsing and interpretation phases

6.2.3 Token Data Model Recap

Each token has four main components:

```
enum class TokenType {
    Identifier, IntegerLiteral, FloatLiteral, StringLiteral, BoolLiteral,
    Let, Const, If, Else, Return, Function,
    Equal, Plus, Minus, Star, Slash, // and many more...
    Semicolon, LeftParen, RightParen,
    EndOfFile, Invalid
};

struct SourceLocation {
    std::size_t line;
    std::size_t column;
};

using LiteralValue = std::variant<std::monostate, int, double, bool, std::string>;

struct Token {
    TokenType type;
    std::string_view lexeme;
    LiteralValue literal;
    SourceLocation location;
};
```

6.2.4 Token Display Format Design

The display of each token must include:

- The token type name (aligned for readability)
- The lexeme (original text from the source)

- Literal value (if applicable)
- Source location (line and column)

Example:

```
Token(Let             ): 'let'        at line 1, col 1
Token(Identifier      ): 'x'          at line 1, col 5
Token(Equal           ): '='          at line 1, col 7
Token(IntegerLiteral  ): '42'         -> 42
Token(Semicolon       ): ';'          at line 1, col 9
```

6.2.5 Implementation of Token Display

1. **Mapping `TokenType` to String**

   Use a centralized function:

```
std::string to_string(TokenType type) {
    switch (type) {
        case TokenType::Let: return "Let";
        case TokenType::Const: return "Const";
        case TokenType::Identifier: return "Identifier";
        case TokenType::IntegerLiteral: return "IntegerLiteral";
        case TokenType::Equal: return "Equal";
        case TokenType::Semicolon: return "Semicolon";
        // Add all token types...
        default: return "Unknown";
    }
}
```

2. **Literal Value Conversion**

   Use a `std::visit` over the `std::variant`:

```
std::string literal_to_string(const LiteralValue& val) {
    return std::visit([](auto&& v) -> std::string {
        using T = std::decay_t<decltype(v)>;
        if constexpr (std::is_same_v<T, std::monostate>) {
            return "";
        } else if constexpr (std::is_same_v<T, std::string>) {
            return "\"" + v + "\"";
        } else if constexpr (std::is_same_v<T, bool>) {
            return v ? "true" : "false";
        } else {
            return std::to_string(v);
        }
    }, val);
}
```

3. **Display Function Using `std::format`**

```
void display_token(const Token& token) {
    std::string type_str = to_string(token.type);
    std::string literal_str = literal_to_string(token.literal);

    if (!literal_str.empty()) {
        std::cout << std::format("Token({:<16}): '{:<10}' -> {:<10} at line {},
        ↪  col {}\n",
            type_str, token.lexeme, literal_str, token.location.line,
            ↪  token.location.column);
    } else {
        std::cout << std::format("Token({:<16}): '{:<10}' at line {}, col
        ↪  {}\n",
```

```
            type_str, token.lexeme, token.location.line,
            ↪  token.location.column);
    }
}
```

6.2.6 Displaying All Tokens from REPL Input

From within the REPL loop:

```
auto tokens = lexer.tokenize();
for (const auto& token : tokens) {
    if (token.type != TokenType::EndOfFile)
        display_token(token);
}
```

You can optionally allow toggling of extra token data (e.g., `--verbose` flag or a special REPL command).

6.2.7 Error Tokens and Highlighting

Handle invalid tokens distinctly:

```
if (token.type == TokenType::Invalid) {
    std::cerr << std::format("X Invalid token: '{}' at line {}, col {}\n",
        token.lexeme, token.location.line, token.location.column);
}
```

This visually separates error tokens from the regular stream.

6.2.8 Supporting Minimal and Diagnostic Modes

For clean or debug views:

- **Minimal mode**: only print type and lexeme

  ```
  Identifier: 'x'
  IntegerLiteral: '42'
  ```

- **Diagnostic mode**: full info with location and literal

  ```
  Token(IntegerLiteral): '42' -> 42 at line 1, col 5
  ```

Can be toggled with a REPL setting.

6.2.9 Sample Session

```
>>> let pi = 3.14;
Token(Let             ): 'let'        at line 1, col 1
Token(Identifier      ): 'pi'         at line 1, col 5
Token(Equal           ): '='          at line 1, col 8
Token(FloatLiteral    ): '3.14'       -> 3.14 at line 1, col 10
Token(Semicolon       ): ';'          at line 1, col 14
python-replCopyEdit>>> "unterminated
X Error: Unterminated string at line 1, col 1
```

6.2.10 Extending Output for External Tools

Later, allow output as:

- JSON (for integration with editors or tools)
- Syntax-colored tokens (for GUI-based REPL or CLI color support)
- Streaming to a file log for testing

Example JSON format:

```
{
  "type": "IntegerLiteral",
  "lexeme": "42",
  "literal": 42,
  "line": 1,
  "column": 5
}
```

6.2.11 Summary of Capabilities

Table 2-2: Lexical Feature Status

Feature	**Status**
Display token type and lexeme	T
Support for literal values	T
Source location info (line/col)	T
Formatting with `std::format`	T
Handling invalid tokens separately	T
Ready for testing and parser integration	T

6.2.12 Conclusion

Displaying tokens in a REPL is a fundamental part of language development. It bridges raw user input and structured interpretation. Using modern C++20/23 tools such as `std::format`, `std::variant`, `std::string_view`, and scoped enums, you can create a clean and extensible display system that is both developer-friendly and toolchain-ready. This visualization reinforces confidence in the lexer, supports early debugging, and sets a visual standard for how your language "speaks back" to its users.

6.3 Testing Basic Language Constructs

6.3.1 Introduction

At this phase of the REPL (Read-Eval-Print Loop) development, the lexer is capable of processing live code inputs and producing a stream of tokens. The next critical step is to **test the tokenization of basic language constructs** in real-time. These tests validate that your language's core grammar—statements, expressions, blocks, and control structures—is being recognized correctly.
This section focuses on building a library of interactive REPL tests using various **C-style programming constructs**, verifying lexer robustness and token accuracy. These early-stage tests play a vital role in shaping the lexical model, ensuring consistency in token recognition, and preparing the path toward syntactic parsing and evaluation.

6.3.2 Purpose of Testing Constructs in REPL v1

Testing basic language constructs in the REPL allows you to:

- Confirm correct token generation for typical code patterns
- Identify misclassified lexemes or gaps in token definitions

- Refine the language grammar early using live examples
- Spot inconsistencies in whitespace, comment, or literal handling
- Validate lexer behavior across multi-token statements

These tests also serve as regression safeguards when modifying or optimizing lexer logic.

6.3.3 Scope of Basic Language Constructs to Test

Typical constructs you should evaluate:

- **Variable Declarations**: `let`, `const`, identifiers, and initialization
- **Assignments and Arithmetic**: `=`, `+`, `-`, `*`, `/`
- **Control Flow**: `if`, `else`, `while`
- **Block Syntax**: `{}`, `()`, `;`
- **Literals**: integer, float, string, boolean
- **Function Calls**: `print("Hello")`, `foo(x + 1)`
- **Comments**: `//`, `/* */`

6.3.4 Using the REPL to Test Constructs

For each construct, input the line, inspect the tokens, and validate formatting:

- **Example 1: Variable Declaration**

```
>>> let count = 10;
Token(Let             ): 'let'
Token(Identifier      ): 'count'
Token(Equal           ): '='
Token(IntegerLiteral  ): '10' -> 10
Token(Semicolon       ): ';'
```

- **Example 2: Expression Statement**

```
>>> x = (a + b) * 2;
Token(Identifier      ): 'x'
Token(Equal           ): '='
Token(LeftParen       ): '('
Token(Identifier      ): 'a'
Token(Plus            ): '+'
Token(Identifier      ): 'b'
Token(RightParen      ): ')'
Token(Star            ): '*'
Token(IntegerLiteral  ): '2' -> 2
Token(Semicolon       ): ';'
```

- **Example 3: Control Flow**

```
>>> if (x > 0) { print("positive"); }
Token(If              ): 'if'
Token(LeftParen       ): '('
Token(Identifier      ): 'x'
Token(Greater         ): '>'
Token(IntegerLiteral  ): '0' -> 0
Token(RightParen      ): ')'
```

```
Token(LeftBrace       ): '{'
Token(Identifier      ): 'print'
Token(LeftParen       ): '('
Token(StringLiteral   ): '"positive"' -> "positive"
Token(RightParen      ): ')'
Token(Semicolon       ): ';'
Token(RightBrace      ): '}'
```

6.3.5 Designing Automated REPL Test Sets (Internally or as Scripts)

In addition to manual testing, REPL behavior should be tested programmatically.

- **Internal Example Using Modern C++ and Catch2**

```
TEST_CASE("Basic statement tokenization") {
    std::string input = "let x = 5;";
    Lexer lexer(input);
    auto tokens = lexer.tokenize();

    REQUIRE(tokens[0].type == TokenType::Let);
    REQUIRE(tokens[1].type == TokenType::Identifier);
    REQUIRE(tokens[2].type == TokenType::Equal);
    REQUIRE(tokens[3].type == TokenType::IntegerLiteral);
    REQUIRE(std::get<int>(tokens[3].literal) == 5);
    REQUIRE(tokens[4].type == TokenType::Semicolon);
}
```

These tests are essential for maintaining correctness across iterations.

6.3.6 Testing Literals and Edge Cases

- **Boolean Literals**

```
>>> let flag = true;
Token(Let            ): 'let'
Token(Identifier     ): 'flag'
Token(Equal          ): '='
Token(BoolLiteral    ): 'true' -> true
Token(Semicolon      ): ';'
```

- **String Literal with Spaces**

```
>>> print("Hello World");
Token(Identifier     ): 'print'
Token(LeftParen      ): '('
Token(StringLiteral  ): '"Hello World"' -> "Hello World"
Token(RightParen     ): ')'
Token(Semicolon      ): ';'
```

- **Unterminated Strings (Error)**

```
>>> print("Hello
X Error: Unterminated string at line 1, col 7
```

6.3.7 Testing Multi-line Block Input (Future Extension)

Although REPL v1 reads single-line statements, multi-line constructs should be simulated and validated to prepare for REPL v2:

```
>>> if (n > 0) {
...     print("positive");
... }
```

Simulate this in REPL by joining lines internally before lexing.

6.3.8 Optional: Table Format for Token Summaries

For development tools and logging:

Table 3-3: Token Stream with Source Location Info

Token Type	Lexeme	Literal	Line	Column
Let	`let`		1	1
Identifier	`x`		1	5
Equal	`=`		1	7
IntegerLiteral	`42`	`42`	1	9
Semicolon	`;`		1	11

6.3.9 Confirming Grammar Design through Testing

Through these REPL tests, you indirectly confirm:

- Token priority and disambiguation rules
- Lexeme-to-token mapping
- Whitespace and newline behavior

- Token boundary recognition
- Support for language idioms

This is essential feedback before introducing parsing logic.

6.3.10 Summary of REPL Construct Testing Benefits

Table 3-4: Benefits of Lexical Analysis Validation

Benefit	Description
Ensures token type accuracy	Confirms lexer grammar alignment
Validates literal extraction	Shows `int`, `float`, `bool`, and `string` handling
Reveals lexical edge cases	Helps catch invalid sequences
Prepares for parsing	Ensures structural readiness
Enables early user feedback	Language designers can test ideas live
Simplifies debugging	Immediate insights into tokenization

6.3.11 Conclusion

Testing basic language constructs interactively through the REPL is more than just verification—it is a form of live specification. It sharpens the language grammar, builds confidence in the lexical system, and sets up a smooth transition into parsing and evaluation. Using modern C++20/23 features, your REPL becomes not just a development tool, but a real-time lens into the heart of your language.

6.4 Milestone — Interactive Explorer for the New Language

6.4.1 Introduction

This section marks a critical milestone in the design of the new C-style interpreted language: the creation of a **functional interactive explorer**, or **REPL-based language explorer**. At this stage, the system offers a structured, testable, and developer-friendly interface to input and analyze code in real time. The explorer is not yet a full interpreter—but it behaves as a **live lexical debugger**, revealing the token-level structure of each input, aiding experimentation, diagnostics, and grammar refinement.

The interactive explorer enables developers to **probe the language**: experiment with new syntax, test token recognition logic, study literal behavior, and confirm grammar rules—all within a tightly integrated REPL environment built using Modern C++20/23.

6.4.2 Purpose of This Milestone

This milestone confirms the successful integration of:

- **Live code input handling** through an interactive loop
- **Dynamic token generation** from raw user input
- **Real-time token inspection and feedback**
- **Consistent source location tracking**
- **User-friendly error reporting for invalid syntax**

- **Modular lexer integration with future parsing support**

This interactive explorer becomes the **primary development interface** in the early phases of language construction.

6.4.3 What the Explorer Does

At this stage, the explorer provides:

Table 4-5: Lexer Feature Overview

Feature	Description
Live tokenization	Accepts input line-by-line and returns structured tokens
Literal inspection	Displays token types, lexemes, and values
Syntax awareness	Detects comments, separators, keywords, symbols
Error diagnostics	Reports malformed tokens, unterminated strings, invalid characters
Clean formatting	Aligns tokens in readable tabular or formatted layout
Stateless analysis	Each input is evaluated independently
Source location display	Shows line and column position of each token

6.4.4 Internals of the Interactive Explorer

1. **Architecture Overview**

```
int main() {
    std::cout << "Lang Explorer v1 - Type `exit` to quit\n";

    std::string line;
    while (true) {
        std::cout << ">>> ";
        if (!std::getline(std::cin, line) || line == "exit") break;

        Lexer lexer(line);
        try {
            auto tokens = lexer.tokenize();
            for (const auto& t : tokens)
                if (t.type != TokenType::EndOfFile)
                    display_token(t);
        } catch (const LexerError& err) {
            std::cerr << std::format("X Error: {} at line {}, col {}\n",
                err.what(), err.location.line, err.location.column);
        }
    }
}
```

2. **Token Display and Formatting**

   Tokens are displayed with precision and clarity:

```
void display_token(const Token& token) {
    std::cout << std::format("Token({:<16}): '{:<10}'",
        to_string(token.type), token.lexeme);

    if (auto val = literal_to_string(token.literal); !val.empty())
        std::cout << std::format(" -> {}", val);
```

```
    std::cout << std::format(" at line {}, col {}\n",
        token.location.line, token.location.column);
}
```

This reinforces a user's understanding of how the language interprets each line.

6.4.5 Supported Input Patterns

The explorer is prepared to handle a wide range of basic constructs:

- Declarations: `let x = 10;`
- Control flow: `if (x > 0) { ... }`
- Arithmetic: `x = a * (b + 3);`
- Functions: `print("debug");`
- Boolean logic: `flag = true && !done;`
- Comments: `// example` and `/* block */`
- Strings: `"Hello, REPL!"`
- Edge cases: malformed or unterminated sequences

These are tokenized, validated, and printed without crashing or needing syntax parsing.

6.4.6 Use of Modern C++20/23 in the Explorer

The REPL explorer benefits from modern language features:

- `std::format` for clean token formatting

- `std::variant` and `std::visit` for handling literal values
- `std::string_view` for zero-copy lexeme processing
- `consteval` and `constexpr` to evaluate utility functions (optional)
- Strongly typed enums for `TokenType`
- `if constexpr` and concepts (in extension) for generic printing

These features boost clarity, type safety, and performance of the REPL system.

6.4.7 Example Interactive Session

```
>>> let version = 1.0;
Token(Let             ): 'let'        at line 1, col 1
Token(Identifier      ): 'version'    at line 1, col 5
Token(Equal           ): '='          at line 1, col 13
Token(FloatLiteral    ): '1.0'        -> 1.0 at line 1, col 15
Token(Semicolon       ): ';'          at line 1, col 18

>>> if (x < 10) { print("low"); }
Token(If              ): 'if'
Token(LeftParen       ): '('
Token(Identifier      ): 'x'
Token(Less            ): '<'
Token(IntegerLiteral  ): '10'         -> 10
Token(RightParen      ): ')'
Token(LeftBrace       ): '{'
Token(Identifier      ): 'print'
Token(LeftParen       ): '('
Token(StringLiteral   ): '"low"'      -> "low"
Token(RightParen      ): ')'
```

```
Token(Semicolon       ): ';'
Token(RightBrace      ): '}'
```

6.4.8 Benefits of the Explorer at This Milestone

Table 4-6: Benefits of Interactive Tokenization and REPL Lexing

Benefit	Impact
Encourages interactive experimentation	Try new syntax and see token effects instantly
Reduces development/debug time	Immediate feedback on lexical design
Confirms grammar decisions	Verifies correctness of language constructs
Provides error visibility	Users quickly learn token rules and boundaries
Serves as validation suite	Manual and automated REPL testing framework
Supports education and tutorials	Helps users understand the language structure

6.4.9 Future Path After This Milestone

After this milestone, the REPL will be extended to include:

- Token buffer persistence (across multiple REPL lines)

- Parser integration for expression/statement recognition
- Syntax tree visualization (AST output)
- Evaluation engine (expression results and variable tracking)
- Support for multiline constructs and user-defined functions

This explorer becomes the **frontend for live language growth**.

6.4.10 Conclusion

This milestone finalizes the construction of the **first interactive system for your new C-style programming language**. It stands as a solid lexical interface for language designers, implementers, and testers. Backed by Modern C++20/23 features, the interactive explorer is not just a playground—it is a **practical, extensible, and production-level component** that validates core design decisions and prepares the system for full parsing and evaluation in upcoming chapters.

Part III

Syntax and Structure

Chapter 7

AST Design for C-Style Constructs

7.1 Expression vs Statement Hierarchies for C-Style Syntax

7.1.1 Introduction

A key step in language implementation is designing the **Abstract Syntax Tree (AST)**, which represents the parsed structure of source code in a tree form. At the heart of this design lies a fundamental separation that most C-style languages follow: the **distinction between expressions and statements**. These two categories define the core structure of code and influence parsing, execution, and semantic analysis.

This section explains how to build a **robust and extensible hierarchy** of expressions and statements using **Modern C++20/23**, while adhering to the syntactic and semantic characteristics of C-style languages.

7.1.2 Understanding Expressions and Statements in C-style Languages

- **Expressions:**
  - Produce a value
  - Can be nested within other expressions
  - Can appear inside statements

  Examples:
  - `42`
  - `x + y`
  - `foo(a, b)`
  - `a > b && c == d`
- **Statements:**
  - Perform an action
  - Control flow or define structure
  - May contain expressions

  Examples:
  - `let x = 5;`
  - `if (a > b) { ... }`
  - `return x * 2;`
  - `while (x < 10) { ... }`

Understanding the **conceptual difference** between these helps to cleanly separate parsing logic, semantic validation, and AST design.

7.1.3 C++ Class Hierarchy Overview

We model the hierarchy using Modern C++ class structures and smart pointers. Expression and Statement become **abstract base classes**, and specific types derive from them.

```
struct Expr;
struct Stmt;

using ExprPtr = std::unique_ptr<Expr>;
using StmtPtr = std::unique_ptr<Stmt>;
```

7.1.4 Base Abstract Classes

```
struct Expr {
    virtual ~Expr() = default;
    virtual std::string debug() const = 0;
};

struct Stmt {
    virtual ~Stmt() = default;
    virtual std::string debug() const = 0;
};
```

The `debug()` function will help inspect trees during development.

7.1.5 Expression Types in a C-style Language

Typical expression node types:

```
struct LiteralExpr : Expr {
    std::variant<int, double, std::string, bool> value;
};

struct VariableExpr : Expr {
    std::string name;
};

struct BinaryExpr : Expr {
    ExprPtr left;
    std::string op;
    ExprPtr right;
};

struct CallExpr : Expr {
    std::string functionName;
    std::vector<ExprPtr> arguments;
};
```

All expression nodes are values or result in values. They can be composed into more complex trees.

7.1.6 Statement Types in a C-style Language

Typical statement node types:

```
struct ExprStmt : Stmt {
    ExprPtr expression;
```

```
};

struct VarDeclStmt : Stmt {
    std::string name;
    ExprPtr initializer;
};

struct BlockStmt : Stmt {
    std::vector<StmtPtr> statements;
};

struct IfStmt : Stmt {
    ExprPtr condition;
    StmtPtr thenBranch;
    std::optional<StmtPtr> elseBranch;
};

struct WhileStmt : Stmt {
    ExprPtr condition;
    StmtPtr body;
};

struct ReturnStmt : Stmt {
    std::optional<ExprPtr> value;
};
```

These structures control the flow and define the executable structure of the program.

7.1.7 Expression Inside Statement Context

In C-style syntax, expressions often occur **within** statements. For example:

```
let x = 5 + y;
```

This is represented as:

- `VarDeclStmt`
  - with initializer: `BinaryExpr` → `LiteralExpr` and `VariableExpr`

This compositional pattern is essential in recursive-descent or Pratt parsing strategies.

7.1.8 Ownership and Memory Management (C++20/23)

To model the AST with safety and performance, **use `std::unique_ptr` and move semantics**:

- `ExprPtr` and `StmtPtr` manage object lifetimes
- `std::make_unique` avoids raw pointers
- C++20's `[[nodiscard]]` ensures safety when constructing nodes

```
auto expr = std::make_unique<BinaryExpr>(
    std::make_unique<LiteralExpr>(5),
    "+",
    std::make_unique<VariableExpr>("y")
);
```

7.1.9 Variant-based AST Models (Optional Modern Alternative)

For projects preferring algebraic style over polymorphism, Modern C++20/23 allows:

```
using Expr = std::variant<
    LiteralExpr, VariableExpr, BinaryExpr, CallExpr
>;
```

But this sacrifices virtual dispatch and requires `std::visit`, which is less intuitive for recursive trees. Use only if your team is experienced with variant-heavy designs.

7.1.10 AST Debugging and Visualization

Using C++20's `std::format`:

```
std::string BinaryExpr::debug() const override {
    return std::format("({} {} {})", left->debug(), op, right->debug());
}
```

This recursive printer is used in REPL and test outputs.

7.1.11 Summary Table: Expressions vs Statements

Table 1-1: Comparison Between Expressions and Statements

Category	Expression	Statement
Purpose	Computes a value	Performs an action
Can Nest	Yes (inside other expressions/statements)	Yes (statements inside blocks or control)

Category	Expression	Statement
Examples	`a + b`, `true`, `foo(x)`	`let x = 10;`, `return a + b;`, `if (...)`
Output	Value	Side-effect or control result
AST Nodes	`BinaryExpr`, `LiteralExpr`, `CallExpr`	`VarDeclStmt`, `IfStmt`, `WhileStmt`

7.1.12 Conclusion

Building a robust AST for a C-style language starts with a clear separation between **expressions** and **statements**. This distinction shapes how you structure your parser, interpreter, and evaluator. By using Modern C++20/23 techniques—smart pointers, variants, scoped enums, and formatting—you build a maintainable, expressive, and powerful foundation that reflects the core design of your language while remaining extensible for future features like functions, classes, and modules.

7.2 Handling C-style Declarations: `int x;`, `float y = 3.14;`

7.2.1 Introduction

C-style variable declarations form one of the foundational building blocks in imperative languages. They introduce variables, optionally assign values, and associate each variable with a type and sometimes a mutability modifier like `const`. In this section, we construct the AST support for such declarations in our new interpreted language, adhering to the principles of Modern C++20/23, and ensuring extendibility, strong

typing, and clean integration with future semantic analysis.

7.2.2 Core Characteristics of C-style Declarations

C-style declarations typically follow this pattern:

```
<type> <identifier> [= <expression>] ;
```

Examples:

- `int x;`
- `float y = 3.14;`
- `bool flag = true;`
- `string name = "John";`

Key components:

- **Type**: Explicit and declared before the variable
- **Name**: Identifier of the variable
- **Initializer** (optional): An expression providing an initial value
- **Semicolon**: Statement terminator

These map directly to a *declaration statement* in the AST.

7.2.3 Designing the Declaration Node

We define a dedicated AST node type for variable declarations. It captures the declared type, the name, and an optional initializer expression.

```
struct VarDeclStmt : Stmt {
    std::string typeName;             // e.g., "int", "float", "bool"
    std::string variableName;         // e.g., "x", "y"
    std::optional<ExprPtr> initializer;

    std::string debug() const override {
        if (initializer)
            return std::format("declare {} {} = {}", typeName, variableName,
            ↪  (*initializer)->debug());
        else
            return std::format("declare {} {}", typeName, variableName);
    }
};
```

The `typeName` is kept as a string for now to simplify parsing. In future phases, it can be replaced with a `Type` object for semantic analysis and type checking.

7.2.4 Parsing C-style Declarations

- **Example: `float y = 3.14;`**

  The parser should:

  1. Recognize the type token (`float`)
  2. Read the variable name (`y`)
  3. Check for `=`, and if found, parse the initializer expression (`3.14`)
  4. Expect `;` to terminate the statement

Result:

```
auto decl = std::make_unique<VarDeclStmt>();
decl->typeName = "float";
decl->variableName = "y";
decl->initializer = parse_expression();  // Produces a LiteralExpr with value
↪  3.14
```

The resulting AST subtree looks like:

```
VarDeclStmt
    type: "float"
    name: "y"
    initializer:
        LiteralExpr: 3.14
```

7.2.5 Expression Types as Initializers

The initializer can be any expression—literal, binary operation, function call, etc. Examples:

- `int x = 10;` → `LiteralExpr`
- `int y = x + 5;` → `BinaryExpr`
- `float z = sin(angle);` → `CallExpr`

The design must support all these as valid expressions attached to `initializer`.

7.2.6 Optional Initialization Handling

Declarations like `int x;` are legal and produce a declaration without initialization. The `initializer` field is simply `std::nullopt`.
In evaluation phase (in later chapters), the interpreter can assign default values like `0`, `0.0`, `false`, or empty strings based on type.

7.2.7 Use of `std::optional` and `std::variant`

C++20's `std::optional` cleanly expresses the presence or absence of an initializer.

```
std::optional<ExprPtr> initializer;
```

The `ExprPtr` itself is a `std::unique_ptr<Expr>`, ensuring safe and non-copyable tree structures.

7.2.8 Supporting `const` and Mutability

If the language supports constants:

```
const float PI = 3.1415;
```

Extend the `VarDeclStmt`:

```
bool isConst = false;  // default false
```

During parsing:

```
if (match(TokenType::Const)) {
    stmt->isConst = true;
    consume_type();
```

```
    ...
}
```

Later in the interpreter, this flag helps enforce immutability.

7.2.9 Example Code and AST Output

- **Source:**

```
int x;
float y = 3.14;
const string name = "Alice";
```

- **AST Print (via `debug()`):**

```
declare int x
declare float y = 3.14
declare const string name = "Alice"
```

  Each statement is parsed into its own `VarDeclStmt`, allowing easy tree traversal, code generation, or interpretation.

7.2.10 Integration with the REPL

Users can type declarations directly in the REPL:

```
>>> int a = 100;
>>> string title = "Interpreter";
```

The REPL should parse and evaluate these, updating a runtime environment with the new variables.

7.2.11 Summary of AST Structure for Declarations

Table 2-2: Fields in Variable Declaration Representation

Field	Description
`typeName`	Declared type as string or enum
`variableName`	Name of the declared variable
`initializer`	Optional expression (`ExprPtr`)
`isConst`	Optional flag for immutability
`debug()`	Utility for AST visualization

7.2.12 Conclusion

C-style declarations form the structural backbone of variables in imperative languages. This section detailed the design and implementation of variable declarations in the AST using Modern C++20/23 features. By encapsulating type names, identifiers, and optional expressions inside a robust `VarDeclStmt` structure, we create a flexible and extensible foundation. This AST design cleanly separates parsing concerns from semantic analysis and will seamlessly support later interpreter stages like type checking and environment binding.

7.3 Block Structure and Scope Representation

7.3.1 Introduction

C-style languages rely heavily on **block structure** to define the boundaries of scopes, especially for functions, conditionals, and loops. In these languages, a block is a sequence of zero or more statements enclosed in `{}` braces. Understanding how to **model block structure and variable scope** in the Abstract Syntax Tree (AST) is essential for building a correct parser, interpreter, or compiler. This section provides a detailed blueprint for representing block structures and managing nested scopes using modern C++ features introduced in C++20 and C++23.

7.3.2 What is a Block in C-Style Syntax?

A block is a **compound statement** defined by:

```
{
    statement1;
    statement2;
    ...
}
```

Blocks:

- Group statements logically
- Introduce **local scopes**
- Allow shadowing of variables
- Are used in functions, conditionals, loops, and standalone code

Blocks are hierarchical, and every new block can introduce its own **symbol table** (or environment in interpreter terms) that overlays the outer scope.

7.3.3 AST Representation of a Block

A block is represented as an AST node that holds a list of child statements:

```
struct BlockStmt : Stmt {
    std::vector<StmtPtr> statements;

    std::string debug() const override {
        std::string result = "{\n";
        for (const auto& stmt : statements)
            result += "  " + stmt->debug() + "\n";
        result += "}";
        return result;
    }
};
```

Each `StmtPtr` inside `statements` could point to a declaration, expression, control flow, or even another nested `BlockStmt`, supporting arbitrary nesting.

7.3.4 Parsing a Block Structure

A recursive descent parser processes blocks by:

- Expecting a `{` token
- Repeatedly parsing valid statements until a matching `}` is found
- Constructing a `BlockStmt` with collected `StmtPtr` elements

Example:

```
BlockStmt parse_block() {
    consume(TokenType::LeftBrace); // expects `{`
    std::vector<StmtPtr> stmts;

    while (check(TokenType::RightBrace) && is_at_end()) {
        stmts.push_back(parse_statement());
    }

    consume(TokenType::RightBrace); // expects `}`
    return BlockStmt{ std::move(stmts) };
}
```

7.3.5 Scope Representation in Interpreter

Each block introduces a **new scope**. Scopes are implemented as **stacked symbol tables**, often using a `std::unordered_map<std::string, Value>` for each level.

```
class Environment {
    std::unordered_map<std::string, Value> variables;
    Environment* parent;

public:
    explicit Environment(Environment* enclosing = nullptr)
        : parent(enclosing) {}

    void define(const std::string& name, const Value& val) {
        variables[name] = val;
    }

    bool assign(const std::string& name, const Value& val) {
        if (variables.contains(name)) {
```

```
            variables[name] = val;
            return true;
        } else if (parent) {
            return parent->assign(name, val);
        }
        return false;
    }

    std::optional<Value> get(const std::string& name) {
        if (variables.contains(name))
            return variables[name];
        if (parent)
            return parent->get(name);
        return std::nullopt;
    }
};
```

On entering a block, a new `Environment` is pushed; on exit, it is popped.

7.3.6 Example: Nested Blocks and Shadowing

- **Source Code:**

```
{
    int x = 10;
    {
        int x = 20;
        print(x); // should print 20
    }
    print(x); // should print 10
}
```

- **AST Structure:**

```
BlockStmt
    VarDeclStmt (int x = 10)
    BlockStmt
        VarDeclStmt (int x = 20)
        ExprStmt (print(x))
    ExprStmt (print(x))
```

Each `BlockStmt` has its own scope. The second declaration of `x` **shadows** the first.

7.3.7 Block Statement Evaluation Workflow

In the interpreter:

```
Value evaluate(const BlockStmt& block, Environment* env) {
    Environment local(env); // new scope
    for (const auto& stmt : block.statements)
        evaluate(*stmt, &local);
    return {}; // or handle return/exit values
}
```

This ensures isolation of variables defined in blocks.

7.3.8 C++20/23 Enhancements in Scope Management

Recent C++ standards help write cleaner and more efficient scope-handling code:

- **`std::move` and `std::unique_ptr`**: For memory safety without leaks
- **Structured bindings**: For deconstructing assignments in environments

- **Ranges and views**: To iterate over scopes and simplify lookup chains
- **`std::optional` and `std::expected`**: For return type safety in lookups and assignments
- **Modules (C++20)**: For future integration of interpreter modules and symbol isolation

7.3.9 Real-World Usage of Block Structures

Block structures in your new language support:

- Nested control flow: `if`, `while`, `for`
- Function bodies
- Manual scope isolation for variables
- Future use in error-handling contexts (like `try`/`catch`)

They are the structural backbone of all non-trivial language behavior.

7.3.10 Summary of Block Scope Representation

Table 3-3: Block Scope Implementation Details

Feature	Implementation Detail
AST Node	`BlockStmt` with `std::vector<StmtPtr>`
Scope Layer	`Environment` with pointer to parent scope

Feature	Implementation Detail
Variable Lifetime	Defined and destroyed within block boundaries
Nested Scope Handling	Stack-like resolution using parent lookup
Shadowing Support	Allowed by checking current level first
Debugging	`debug()` method to visualize blocks

7.3.11 Conclusion

Block statements and nested scopes are central to the semantics of structured programming. In your interpreter design, `BlockStmt` becomes the primary unit for grouping logic and introducing isolated environments. C++20 and C++23 make scope management clearer and safer through modern memory ownership, structured error handling, and modular design. Proper implementation of block structure at the AST and interpreter level ensures correctness, maintainability, and extensibility as your language grows into a more powerful system.

7.4 Memory-Safe Tree Construction with Smart Pointers

7.4.1 Introduction

Constructing an Abstract Syntax Tree (AST) for a C-style interpreted language involves managing a hierarchical structure of nodes that represent expressions, statements, and control constructs. In the past, building such trees in C++ required tedious manual memory management, leading to leaks, dangling pointers, or ownership

confusion. With Modern C++ (C++20/23), we now have **safe, elegant, and efficient** tools to build and manage these trees using **smart pointers**.
This section explores the best practices and techniques for memory-safe tree construction in ASTs using `std::unique_ptr`, `std::shared_ptr`, and `std::make_unique` with a focus on clarity, correctness, and performance.

7.4.2 Why Smart Pointers for ASTs?

Smart pointers automate memory management and clearly express **ownership semantics**.

- **AST nodes form a tree**: Ownership is usually exclusive. A `BinaryExpr` owns its `left` and `right` expressions.
- **No shared cyclic references**: So `std::unique_ptr` is ideal.
- **Compiler errors on misuse**: Safer than raw pointers.

Using smart pointers:

- Prevents memory leaks
- Prevents premature deallocation
- Encourages correct lifetime and transfer semantics

7.4.3 Choosing the Right Smart Pointer

- `std::unique_ptr<T>`
  - **Best choice** for ASTs
  - Models **exclusive ownership**

  - Automatically deletes the owned object when destroyed

- **`std::shared_ptr<T>` (use sparingly)**
  - For **shared ownership** scenarios (rare in ASTs)
  - Slower due to atomic reference counting
  - Risk of cycles if not designed carefully

- **`std::weak_ptr<T>`**
  - Optional support to reference parent nodes (if needed), without ownership

  Conclusion: Use `std::unique_ptr<T>` as the default for all AST node relationships.

7.4.4 Defining AST Node Ownership

Create aliases for clarity:

```
struct Expr;
struct Stmt;

using ExprPtr = std::unique_ptr<Expr>;
using StmtPtr = std::unique_ptr<Stmt>;
```

Then use these in node definitions:

```
struct BinaryExpr : Expr {
    ExprPtr left;
    std::string op;
    ExprPtr right;
};
```

This clearly models the tree structure: `BinaryExpr` *owns* its children.

7.4.5 Constructing AST Nodes with `std::make_unique`

Use `std::make_unique` to safely construct nodes:

```
auto expr = std::make_unique<BinaryExpr>(
    std::make_unique<LiteralExpr>(5),
    "+",
    std::make_unique<VariableExpr>("x")
);
```

This is:

- Exception-safe
- Compact
- Clear in ownership

C++20 allows even cleaner syntax by leveraging CTAD (Class Template Argument Deduction) where applicable.

7.4.6 Moving Pointers: Avoiding Copy Mistakes

AST pointers must be **moved**, not copied:

```
ExprPtr left = std::make_unique<LiteralExpr>(5);
ExprPtr right = std::make_unique<LiteralExpr>(10);

auto node = std::make_unique<BinaryExpr>(std::move(left), "+", std::move(right));
```

Once moved, the original pointer is null. This ensures **single ownership** is preserved. Failing to `std::move` will result in compile-time errors, enforcing safety.

7.4.7 AST Destruction is Automatic

Once a parent node is destroyed, all its children are recursively destroyed:

```
void run() {
    auto tree = std::make_unique<BinaryExpr>(
        std::make_unique<LiteralExpr>(42),
        "*",
        std::make_unique<VariableExpr>("count")
    );

    // tree goes out of scope here → no leak, no manual delete
}
```

This simplifies cleanup and avoids manual memory management entirely.

7.4.8 Visitor and Pattern Matching with Smart Pointers

When implementing tree traversal (evaluation, printing, etc.), pass `const Expr&` or `const Expr*`:

```
void visit(const Expr& expr);
```

Never pass `unique_ptr` directly to visitors—it transfers ownership. Instead, dereference or use a raw pointer to observe, not own.

7.4.9 AST Nodes with Optional Members

C++20 introduced `std::optional`, perfect for optional children:

```
struct IfStmt : Stmt {
    ExprPtr condition;
    StmtPtr thenBranch;
    std::optional<StmtPtr> elseBranch; // optional else block
};
```

This makes ASTs more expressive and readable, while maintaining full safety.

7.4.10 Debugging AST with Smart Pointer Safety

Use utility functions for printing trees:

```
void print_ast(const ExprPtr& node) {
    if (!node) return;
    std::cout << node->debug() << "\n";
}
```

Or overload `debug()` to recursively call child `ExprPtr`s.

7.4.11 Summary: Best Practices

Table 4-4: Modern C++ Approach for AST Handling

Task	Modern C++ Approach
Create AST nodes	Use `std::make_unique<NodeType>()`
Represent child nodes	Use `std::unique_ptr<T>`
Optional children	Use `std::optional<std::unique_ptr<T>>`

Task	Modern C++ Approach
Traverse AST	Use references (`const T&`)
Destroy AST	Let smart pointers handle it
Avoid raw pointer bugs	Never use `new` or `delete` directly

7.4.12 Example: Full AST Construction Snippet

```
auto expr = std::make_unique<BinaryExpr>(
    std::make_unique<VariableExpr>("a"),
    "*",
    std::make_unique<BinaryExpr>(
        std::make_unique<LiteralExpr>(3),
        "+",
        std::make_unique<LiteralExpr>(5)
    )
);
```

This represents the code: `a * (3 + 5)`
Each node owns its children, and the entire structure is freed automatically.

7.4.13 Conclusion

The combination of **tree hierarchy design** and **Modern C++ smart pointers** provides an optimal and memory-safe strategy for AST construction. By exclusively using `std::unique_ptr` and embracing move semantics, you ensure correctness, performance, and maintainability without the overhead of manual memory handling. This approach is essential for large-scale interpreters where complex ASTs are frequently constructed, traversed, and discarded.

7.5 Hands-on – Core AST Nodes for C-Style Language

7.5.1 Introduction

A C-style language requires a well-organized and extensible **Abstract Syntax Tree (AST)** to represent expressions, statements, blocks, declarations, and control structures. In this section, we focus on **concrete implementation** of the **core AST nodes** that form the fundamental building blocks of the language syntax.
Using Modern C++ (C++20/23), we implement a safe, expressive, and maintainable hierarchy of AST classes leveraging smart pointers, structured enums, `std::optional`, and `std::variant` where necessary.
This hands-on section is the bridge between theoretical design and actual interpreter integration.

7.5.2 Base AST Node Interfaces

We start with two abstract base classes to distinguish **expressions** from **statements**:

```
struct Expr {
    virtual ~Expr() = default;
    virtual std::string debug() const = 0;
};

struct Stmt {
    virtual ~Stmt() = default;
    virtual std::string debug() const = 0;
};
```

These interfaces allow polymorphic dispatch and introspection for debugging and

evaluation.

7.5.3 Smart Pointer Aliases for Ownership

Use `std::unique_ptr` to model tree ownership relationships:

```
using ExprPtr = std::unique_ptr<Expr>;
using StmtPtr = std::unique_ptr<Stmt>;
```

All child nodes are stored and transferred using these types, ensuring memory safety and clarity.

7.5.4 Core Expression Nodes

- **Literal Expression**

  Represents primitive constant values (int, float, string, bool):

```
struct LiteralExpr : Expr {
    std::variant<int, double, std::string, bool> value;

    explicit LiteralExpr(auto v) : value(v) {}

    std::string debug() const override {
        return std::visit([](auto&& val) {
            return std::to_string(val);
        }, value);
    }
};
```

- **Variable Expression**

Represents a reference to a variable:

```
struct VariableExpr : Expr {
    std::string name;

    explicit VariableExpr(std::string n) : name(std::move(n)) {}

    std::string debug() const override {
        return name;
    }
};
```

- **Binary Expression**

  Represents binary operations (e.g., `a + b`):

```
struct BinaryExpr : Expr {
    ExprPtr left;
    std::string op;
    ExprPtr right;

    BinaryExpr(ExprPtr l, std::string o, ExprPtr r)
        : left(std::move(l)), op(std::move(o)), right(std::move(r)) {}

    std::string debug() const override {
        return "(" + left->debug() + " " + op + " " + right->debug() + ")";
    }
};
```

- **Unary Expression**

  Supports operators like `-x`, `!flag`:

```
struct UnaryExpr : Expr {
    std::string op;
    ExprPtr operand;

    UnaryExpr(std::string o, ExprPtr e)
        : op(std::move(o)), operand(std::move(e)) {}

    std::string debug() const override {
        return "(" + op + operand->debug() + ")";
    }
};
```

- **Function Call Expression**

  Used to represent function invocations:

```
struct CallExpr : Expr {
    std::string callee;
    std::vector<ExprPtr> arguments;

    CallExpr(std::string c, std::vector<ExprPtr> args)
        : callee(std::move(c)), arguments(std::move(args)) {}

    std::string debug() const override {
        std::string argList;
        for (const auto& arg : arguments)
            argList += arg->debug() + ", ";
        if (!argList.empty()) argList.pop_back(); // remove last comma

        return callee + "(" + argList + ")";
    }
};
```

7.5.5 Core Statement Nodes

- **Expression Statement**

  Wraps a pure expression as a standalone statement (e.g., `x + 2;`):

```
struct ExprStmt : Stmt {
    ExprPtr expression;

    explicit ExprStmt(ExprPtr expr) : expression(std::move(expr)) {}

    std::string debug() const override {
        return expression->debug() + ";";
    }
};
```

- **Variable Declaration Statement**

  Supports declarations like `int x = 5;`:

```
struct VarDeclStmt : Stmt {
    std::string typeName;
    std::string variableName;
    std::optional<ExprPtr> initializer;

    VarDeclStmt(std::string type, std::string name, std::optional<ExprPtr> init
    ↪  = std::nullopt)
        : typeName(std::move(type)), variableName(std::move(name)),
        ↪  initializer(std::move(init)) {}

    std::string debug() const override {
        if (initializer)
            return typeName + " " + variableName + " = " +
            ↪  (*initializer)->debug() + ";";
```

```
        return typeName + " " + variableName + ";";
    }
};
```

- **Block Statement**

  Represents `{ ... }` with scoped statements:

```
struct BlockStmt : Stmt {
    std::vector<StmtPtr> statements;

    explicit BlockStmt(std::vector<StmtPtr> stmts)
        : statements(std::move(stmts)) {}

    std::string debug() const override {
        std::string result = "{\n";
        for (const auto& stmt : statements)
            result += "  " + stmt->debug() + "\n";
        result += "}";
        return result;
    }
};
```

- **If Statement**

  Supports conditional execution:

```
struct IfStmt : Stmt {
    ExprPtr condition;
    StmtPtr thenBranch;
    std::optional<StmtPtr> elseBranch;
```

```
    IfStmt(ExprPtr cond, StmtPtr thenB, std::optional<StmtPtr> elseB =
    ↪  std::nullopt)
        : condition(std::move(cond)), thenBranch(std::move(thenB)),
        ↪  elseBranch(std::move(elseB)) {}

    std::string debug() const override {
        std::string result = "if (" + condition->debug() + ") " +
        ↪  thenBranch->debug();
        if (elseBranch)
            result += " else " + (*elseBranch)->debug();
        return result;
    }
};
```

- **While Statement**

  Used for looping constructs:

```
struct WhileStmt : Stmt {
    ExprPtr condition;
    StmtPtr body;

    WhileStmt(ExprPtr cond, StmtPtr b)
        : condition(std::move(cond)), body(std::move(b)) {}

    std::string debug() const override {
        return "while (" + condition->debug() + ") " + body->debug();
    }
};
```

- **Return Statement**

Handles function returns:

```
struct ReturnStmt : Stmt {
    std::optional<ExprPtr> value;

    explicit ReturnStmt(std::optional<ExprPtr> val = std::nullopt)
        : value(std::move(val)) {}

    std::string debug() const override {
        return value ? "return " + (*value)->debug() + ";" : "return;";
    }
};
```

7.5.6 Visual Summary of AST Nodes

Table 5-5: Common AST Node Types

Node Type	Kind	Example Code	AST Class
`5 + 3`	Expression	`5 + 3`	`BinaryExpr`
`x = 10;`	Statement	Assignment	`ExprStmt`
`int x = 0;`	Statement	Variable Declaration	`VarDeclStmt`
`{ ... }`	Statement	Block	`BlockStmt`
`if (...) {...}`	Statement	Conditional	`IfStmt`
`while (...) {}`	Statement	Loop	`WhileStmt`
`return x;`	Statement	Return	`ReturnStmt`

Node Type	Kind	Example Code	AST Class
`f(x, y)`	Expression	Function Call	`CallExpr`
`"hello"`	Expression	Literal	`LiteralExpr`

7.5.7 Conclusion

The hands-on AST construction using Modern C++ allows your interpreter to represent C-style syntax in a scalable and memory-safe way. Each node type is modeled using polymorphism and `std::unique_ptr`, ensuring correctness and clarity. With this core set of AST classes, you now have a solid foundation to support parsing, type checking, interpretation, optimization, and code generation for your language.

These nodes not only reflect the expressive structure of your language but also act as the runtime scaffolding upon which future features like functions, lambdas, classes, and modules can be elegantly built.

Chapter 8

Parsing C-Style Grammar

8.1 Grammar Design for C-Style Syntax

8.1.1 Introduction

Designing a grammar is the first fundamental step in defining the structure and meaning of a programming language. For a C-style language, the grammar needs to support familiar syntax constructs such as expressions, declarations, control flow, blocks, and function definitions, while also offering opportunities for extending the language in a structured and deterministic way.
This section outlines how to carefully craft a **recursive descent-compatible grammar** using modern parsing strategies, grammar best practices, and preparation for implementing it in C++20/23.

8.1.2 Objectives of C-Style Grammar Design

A well-designed grammar for a C-style syntax should:

- Represent the familiar structure of C-like languages (curly braces, semicolon-

terminated statements)

- Support left-to-right evaluation with clear precedence rules
- Enable unambiguous parsing of expressions and statements
- Be implementable using recursive descent parsers (LL(1) or operator-precedence driven)
- Be modular for extensions (such as types, user-defined functions, classes)

8.1.3 Lexical Considerations Before Parsing

Before designing grammar rules, ensure that the **token set** is well defined. Common token types for a C-style grammar include:

- **Keywords**: `if`, `else`, `while`, `return`, `int`, `float`, `bool`, `string`, `const`
- **Operators**: `+`, `-`, `*`, `/`, `%`, `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `!`
- **Delimiters**: `;`, `{`, `}`, `(`, `)`, `,`
- **Literals**: integer, floating-point, string, boolean
- **Identifiers**

These tokens must be clearly defined in your lexer to avoid ambiguity in the grammar.

8.1.4 Top-Level Grammar Rule: Translation Unit

At the highest level, a translation unit (file, REPL session, or input buffer) is a sequence of declarations or statements:

```
program        → declaration* EOF ;
```

8.1.5 Statements and Declarations

Grammar rules differentiate between **statements** (executed actions) and **declarations** (variable/function introduction).

```
declaration    → var_decl
               | function_decl
               | statement ;

var_decl       → type IDENTIFIER ( "=" expression )? ";" ;
function_decl  → type IDENTIFIER "(" parameters? ")" block ;

statement      → expr_stmt
               | if_stmt
               | while_stmt
               | return_stmt
               | block ;

expr_stmt      → expression ";" ;
block          → "{" declaration* "}" ;
return_stmt    → "return" expression? ";" ;
```

This supports flexible and expandable semantics.

8.1.6 Expressions and Operator Precedence

Expression parsing is best handled using **precedence-based rules**, moving from lower to higher precedence (e.g., assignment to unary).

```
expression     → assignment ;

assignment     → IDENTIFIER "=" assignment
               | logic_or ;

logic_or       → logic_and ( "||" logic_and )* ;
logic_and      → equality ( "&&" equality )* ;
equality       → comparison ( ( "==" | "!=" ) comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           → factor ( ( "+" | "-" ) factor )* ;
factor         → unary ( ( "*" | "/" | "%" ) unary )* ;
unary          → ( "!" | "-" ) unary
               | primary ;

primary        → INTEGER | FLOAT | STRING | "true" | "false"
               | IDENTIFIER
               | "(" expression ")" ;
```

Using this precedence ordering ensures correctness when parsing expressions like:

```
x = a + b * (c - d) / 2;
```

Each rule calls the higher-precedence rules, allowing for **left-associative parsing**.

8.1.7 Control Flow Statements

Support for `if`, `else`, and `while` can follow this structure:

```
if_stmt        → "if" "(" expression ")" statement ( "else" statement )? ;
while_stmt     → "while" "(" expression ")" statement ;
```

These rules integrate cleanly with `block` so that both single-line and block bodies are supported.

8.1.8 Function Parameters and Calls

Function declarations and calls require grammar support for parameter lists:

```
parameters     → type IDENTIFIER ( "," type IDENTIFIER )* ;

arguments      → expression ( "," expression )* ;

call_expr      → IDENTIFIER "(" arguments? ")" ;
```

Calls are integrated into `primary` via lookahead:

```
primary        → IDENTIFIER ( "(" arguments? ")" )?
               | ...
```

8.1.9 Grammar Error Recovery Patterns

While the parser should fail on invalid input, incorporating synchronization rules helps recover from common mistakes.
For example, after a parse error inside a block, skip to the next semicolon or closing brace:

```
void synchronize() {
    while (!is_at_end()) {
        if (previous().type == TokenType::Semicolon) return;
        switch (peek().type) {
            case TokenType::If:
```

```
            case TokenType::While:
            case TokenType::Return:
            case TokenType::Int:
            case TokenType::Float:
            case TokenType::Bool:
                return;
        }
        advance();
    }
}
```

This makes the parser more robust for both script input and REPL interaction.

8.1.10 C++20/23 Integration Ideas for Grammar Handling

Although parsing is mostly logic-driven, Modern C++ helps with:

- **`std::variant` for expression node representation**
- **`std::optional` for nullable rules like else-branch or initializer**
- **`std::format` or `std::ostringstream` for debug output**
- **Pattern matching (future in C++23+ or experimental)** for visitor-style traversal or transformation

8.1.11 Grammar Extensibility and Modularity

Each rule is designed to be modular and composable:

- Statements and declarations are independent
- Expressions are layered by precedence

- Function support can grow with new constructs like lambdas or closures
- New statement types (e.g., `switch`, `for`) can be plugged into the `statement` rule

This layered and modular approach is crucial for long-term language evolution.

8.1.12 Example: Full Grammar Snippet (Subset)

```
program        → declaration* EOF ;
declaration    → var_decl | function_decl | statement ;
var_decl       → type IDENTIFIER ( "=" expression )? ";" ;
function_decl  → type IDENTIFIER "(" parameters? ")" block ;
statement      → expr_stmt | if_stmt | while_stmt | return_stmt | block ;
expr_stmt      → expression ";" ;
if_stmt        → "if" "(" expression ")" statement ( "else" statement )? ;
expression     → IDENTIFIER "=" expression | logic_or ;
logic_or       → logic_and ( "||" logic_and )* ;
...
primary        → IDENTIFIER | NUMBER | STRING | "true" | "false" | "(" expression ")"
↪  ;
```

8.1.13 Conclusion

Grammar design is the architectural backbone of your language parser. A solid C-style grammar emphasizes clarity, precedence hierarchy, and modular rules that map directly to clean AST node construction. When combined with Modern C++ features like smart pointers, variants, and optionals, this design ensures your language is not only expressive and correct but also maintainable and extensible for future development.

8.2 Expression Parsing with C Operator Precedence

8.2.1 Introduction

Expression parsing in a C-style language is one of the most critical components in grammar implementation. It must correctly reflect **operator precedence**, **associativity**, and **expression nesting** in a way that aligns with the expectations of C/C++ developers. Misparsing expressions can break logic, cause subtle bugs, or produce incorrect AST structures.

In this section, we dive deep into constructing a precedence-based expression parser using recursive descent and Modern C++ techniques, adhering closely to the operator rules of C. The implementation is designed to support modular expansion and safe expression evaluation through AST node generation.

8.2.2 Why Precedence-Based Parsing Matters

In C and its derivatives, operators have well-defined **precedence** levels and **associativity** rules. For example:

```
int x = a + b * c - d / e;
```

must parse as:

```
x = ((a + (b * c)) - (d / e))
```

This parsing requires the parser to respect **multiplication and division** binding more tightly than **addition and subtraction**, and **assignment** binding less tightly than both.

8.2.3 Operator Precedence and Associativity in C

Here is a simplified table of common C operators in descending precedence:

Table 2-1: Operator Precedence and Associativity

Precedence	Operators	Associativity
1 (highest)	`()  []  .  ->`	Left-to-right
2	`++  --  +  -  !`	Right-to-left
3	`*  /  %`	Left-to-right
4	`+  -`	Left-to-right
5	`<  <=  >  >=`	Left-to-right
6	`==  !=`	Left-to-right
7	`&&`	Left-to-right
8		
9	`=  +=  -=  *=  /=`	Right-to-left

The parsing algorithm must honor both **precedence** and **associativity** for correctness.

8.2.4 Strategy: Recursive Descent with Precedence Climbing

There are multiple strategies for parsing expressions with precedence:

- **Operator-precedence parsing (shunting-yard)**
- **Pratt parsing**
- **Precedence climbing (recursive descent)**

For hand-written interpreters in Modern C++, **precedence climbing** using recursive descent is often the clearest and most maintainable. Each level of precedence is handled by a distinct function.

8.2.5 Layered Expression Grammar (EBNF)

Each expression level is structured from highest to lowest precedence:

```
expression      → assignment ;
assignment      → IDENTIFIER "=" assignment | logic_or ;
logic_or        → logic_and ( "||" logic_and )* ;
logic_and       → equality ( "&&" equality )* ;
equality        → comparison ( ( "==" | "!=" ) comparison )* ;
comparison      → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term            → factor ( ( "+" | "-" ) factor )* ;
factor          → unary ( ( "*" | "/" | "%" ) unary )* ;
unary           → ( "!" | "-" ) unary | primary ;
primary         → literal | IDENTIFIER | "(" expression ")" ;
```

This grammar structure is recursive and strictly enforces operator precedence via the calling order.

8.2.6 Implementing Precedence Parsing in C++20

Each function parses a specific level and delegates down for tighter binding. The result is an **AST node** (`ExprPtr`) returned upwards.

- **Example: Binary Expression Parsing**

```
ExprPtr parse_term() {
    ExprPtr expr = parse_factor();
```

```
    while (match(TokenType::Plus) || match(TokenType::Minus)) {
        Token op = previous();
        ExprPtr right = parse_factor();
        expr = std::make_unique<BinaryExpr>(std::move(expr), op.lexeme,
        ↪  std::move(right));
    }

    return expr;
}
```

- **Nested Composition**

  Each level nests lower levels:

```
ExprPtr parse_comparison() {
    ExprPtr expr = parse_term();

    while (match(TokenType::Greater, TokenType::LessEqual)) {
        Token op = previous();
        ExprPtr right = parse_term();
        expr = std::make_unique<BinaryExpr>(std::move(expr), op.lexeme,
        ↪  std::move(right));
    }

    return expr;
}
```

8.2.7 Unary Expressions

Unary operators (`-`, `!`) are **right-associative** and only bind to a single operand:

```
ExprPtr parse_unary() {
    if (match(TokenType::Minus, TokenType::Bang)) {
        Token op = previous();
        ExprPtr right = parse_unary();
        return std::make_unique<UnaryExpr>(op.lexeme, std::move(right));
    }

    return parse_primary();
}
```

8.2.8 Assignment and Right-Associativity

Assignment has the lowest precedence and is right-associative:

```
ExprPtr parse_assignment() {
    ExprPtr expr = parse_logical_or();

    if (match(TokenType::Equal)) {
        Token equals = previous();
        ExprPtr value = parse_assignment();

        if (auto var = dynamic_cast<VariableExpr*>(expr.get())) {
            return std::make_unique<AssignExpr>(var->name, std::move(value));
        } else {
            throw std::runtime_error("Invalid assignment target.");
        }
    }

    return expr;
}
```

This allows chaining like:

```
a = b = c = 42;
```

Which parses as: `a = (b = (c = 42))`

8.2.9 Using Modern C++ Features

C++20/23 offers improvements in parser implementation:

- `std::variant` for unified expression trees
- `std::optional` for nullable constructs
- `std::ranges` and `std::span` for streamlining token consumption
- `std::format` for debug-friendly output
- `concepts` to constrain parser utilities and error-checking

Example:

```
template<typename T>
concept IsExpr = std::derived_from<T, Expr>;
```

Used to enforce that helper functions only operate on valid AST types.

8.2.10 Debugging Expression AST

Each `Expr` node implements `debug()`:

```
std::string BinaryExpr::debug() const {
    return "(" + left->debug() + " " + op + " " + right->debug() + ")";
}
```

Helps visualize parsing correctness:
Input:

```
a + b * c
```

Output:

```
(a + (b * c))
```

8.2.11 Expression Parsing Example (From Tokens to AST)

Source:

```
x = 3 + 4 * (2 - 1);
```

Steps:

1. Parse assignment: `x = <expr>`
2. Parse addition: `3 + <expr>`
3. Parse multiplication: `4 * (2 - 1)`
4. Parse parentheses: `2 - 1`

AST result:

```
AssignExpr
   name: "x"
   BinaryExpr "+"
       LiteralExpr 3
       BinaryExpr "*"
```

```
        LiteralExpr 4
        BinaryExpr "-"
            LiteralExpr 2
            LiteralExpr 1
```

8.2.12 Conclusion

Expression parsing with C operator precedence is one of the most critical and error-prone areas in building a language parser. Using recursive descent and Modern C++ features, we achieve a clean, extensible parser that respects all precedence and associativity rules of C. This approach lays a stable foundation for a fully expressive and reliable interpreter.

8.3 Statement Parsing – Declarations, Blocks, and Control Flow

8.3.1 Introduction

Parsing statements is a cornerstone of any C-style language parser. Statements define how code executes: from variable declarations and scope blocks to flow-control instructions like `if`, `while`, and `return`. This section walks through the design and implementation of a **statement-level parser** using recursive descent, leveraging **Modern C++20/23** for safer and clearer abstraction.

Our parser assumes that the lexer has already transformed the raw source into tokens, and that we have successfully implemented expression parsing with proper precedence.

8.3.2 Statement Categories in C-Style Syntax

C-style languages typically define these categories of statements:

1. **Expression statements**
2. **Variable declarations**
3. **Block statements** (`{ ... }`)
4. **Control flow**: `if`, `else`, `while`, `return`
5. **Function definitions** (higher-level, handled separately)

We define a `parse_statement()` dispatcher that handles all the categories.

8.3.3 Core Parsing Function

```
StmtPtr parse_statement() {
    if (match(TokenType::If))       return parse_if_statement();
    if (match(TokenType::While))    return parse_while_statement();
    if (match(TokenType::Return))   return parse_return_statement();
    if (match(TokenType::LeftBrace))return parse_block();
    if (check_type())               return parse_variable_declaration();

    return parse_expression_statement();
}
```

Each `parse_..._statement()` is responsible for its own rule.

8.3.4 Variable Declarations

C-style variable declarations may include optional initializers:

```
int x;
float y = 3.14;
```

Grammar:

```
var_decl → type IDENTIFIER ( "=" expression )? ";" ;
```

Implementation:

```
StmtPtr parse_variable_declaration() {
    std::string type = parse_type(); // e.g., int, float
    consume(TokenType::Identifier, "Expected variable name");
    std::string name = previous().lexeme;

    std::optional<ExprPtr> initializer;
    if (match(TokenType::Equal)) {
        initializer = parse_expression();
    }

    consume(TokenType::Semicolon, "Expected ';' after declaration");
    return std::make_unique<VarDeclStmt>(type, name, std::move(initializer));
}
```

8.3.5 Block Statements

Blocks define scope and allow grouping multiple statements:

```
{
    int a = 10;
    print(a);
}
```

Grammar:

```
block → "{" declaration* "}" ;
```

Implementation:

```
StmtPtr parse_block() {
    std::vector<StmtPtr> statements;

    while (check(TokenType::RightBrace) && is_at_end()) {
        statements.push_back(parse_statement());
    }

    consume(TokenType::RightBrace, "Expected '}' after block");
    return std::make_unique<BlockStmt>(std::move(statements));
}
```

8.3.6 Expression Statements

Used for pure expressions, such as assignments or function calls:

```
x = 5;
log(x);
```

Grammar:

```
expr_stmt → expression ";" ;
```

Implementation:

```
StmtPtr parse_expression_statement() {
    ExprPtr expr = parse_expression();
    consume(TokenType::Semicolon, "Expected ';' after expression");
    return std::make_unique<ExprStmt>(std::move(expr));
}
```

8.3.7 If Statements

Handles conditional branching:

```
if (x > 0) print("positive");
else print("zero or negative");
```

Grammar:

```
if_stmt → "if" "(" expression ")" statement ("else" statement)? ;
```

Implementation:

```
StmtPtr parse_if_statement() {
    consume(TokenType::LeftParen, "Expected '(' after 'if'");
    ExprPtr condition = parse_expression();
    consume(TokenType::RightParen, "Expected ')' after condition");

    StmtPtr then_branch = parse_statement();
    std::optional<StmtPtr> else_branch;
```

```
    if (match(TokenType::Else)) {
        else_branch = parse_statement();
    }

    return std::make_unique<IfStmt>(std::move(condition), std::move(then_branch),
    ↪  std::move(else_branch));
}
```

8.3.8 While Loops

Used for repeating code while a condition holds:

```
while (i < 10) {
    i = i + 1;
}
```

Grammar:

```
while_stmt → "while" "(" expression ")" statement ;
```

Implementation:

```
StmtPtr parse_while_statement() {
    consume(TokenType::LeftParen, "Expected '(' after 'while'");
    ExprPtr condition = parse_expression();
    consume(TokenType::RightParen, "Expected ')' after condition");

    StmtPtr body = parse_statement();
    return std::make_unique<WhileStmt>(std::move(condition), std::move(body));
}
```

8.3.9 Return Statements

Used to return values from functions:

```
return 42;
```

Grammar:

```
return_stmt → "return" expression? ";" ;
```

Implementation:

```
StmtPtr parse_return_statement() {
    std::optional<ExprPtr> value;
    if (!check(TokenType::Semicolon)) {
        value = parse_expression();
    }
    consume(TokenType::Semicolon, "Expected ';' after return value");
    return std::make_unique<ReturnStmt>(std::move(value));
}
```

8.3.10 Error Recovery: Synchronization Strategy

If a parsing error occurs in the middle of a statement, we use a **synchronization** technique to avoid cascading errors.

```
void synchronize() {
    advance();
    while (!is_at_end()) {
        if (previous().type == TokenType::Semicolon) return;
```

```
        switch (peek().type) {
            case TokenType::If:
            case TokenType::While:
            case TokenType::Return:
            case TokenType::Int:
            case TokenType::Float:
            case TokenType::Bool:
                return;
        }

        advance();
    }
}
```

This helps recover from incomplete or invalid statements.

8.3.11 Using Modern C++20/23 Features

Modern C++ simplifies parser construction and memory safety:

- **`std::optional`** for optional branches (`else`, `return` values)
- **`std::move`** for efficient AST node transfer
- **`std::unique_ptr`** for owning AST memory safely
- **Structured bindings and `constexpr if`** for simplifying pattern matching (in newer C++23-compatible compilers)

8.3.12 Visual Summary of Statement Parsing

Table 3-2: Code Example to AST Node Mapping

Code Example	AST Node
`int x = 5;`	`VarDeclStmt`
`{ int y = 3; print(y); }`	`BlockStmt`
`if (x > 0) print(x);`	`IfStmt`
`while (i < 10) { i++; }`	`WhileStmt`
`return 42;`	`ReturnStmt`
`log("done");`	`ExprStmt`

8.3.13 Conclusion

Statement parsing in a C-style language is rich in structure, requiring cleanly organized parsing rules that support declarations, scoping blocks, and control flow. Using recursive descent and the strength of C++20/23, we achieve a parser that is modular, safe, and extensible.

By handling each statement type separately while maintaining a unified AST construction model, we prepare the groundwork for later interpreter stages like semantic analysis, type checking, and evaluation. This statement parser becomes the core skeleton that gives structure to any program written in your custom language.

8.4 Integrated Testing During Development

8.4.1 Introduction

Developing a new programming language requires tight feedback loops. Integrated testing during development plays a critical role in ensuring the correctness and reliability of the language parser, especially when handling grammar complexity, operator precedence, statement nesting, and error handling. This section explains how to build an **automated, incremental test system** for your parser using **Modern C++20/23**, with a focus on **AST integrity**, **error detection**, **token coverage**, and **grammar conformance**.

Over the past five years, integrated testing techniques in C++ projects have become more structured, modular, and expressive due to improvements in the language and the availability of lightweight testing frameworks.

8.4.2 Why Integrated Testing Matters for Parsing

Parsing is a foundational phase in language interpretation. A bug in the parser not only distorts program logic but often corrupts the internal representation (AST) silently. Therefore, real-time, automated tests offer benefits such as:

- Early detection of invalid grammar rules
- Guaranteed consistency of expression precedence and associativity
- Validation of block scoping and control structure parsing
- Continuous feedback during parser refactoring

Integrated tests are essential at every stage of the grammar implementation — from token sequence recognition to full AST comparison.

8.4.3 Categories of Tests to Implement

To properly verify the correctness of your parser, implement tests at multiple levels:

Table 4-3: Test Categories and Their Purpose

Test Category	Purpose
Lexer/token tests	Validate token recognition, position, and types
Expression parsing	Confirm correct AST structure with operator precedence
Statement parsing	Test blocks, `if`, `while`, `return`, declarations
Full program tests	Parse entire code snippets and validate AST
Error recovery	Check whether parser can recover from invalid syntax

Each test feeds real language input and compares the result with expected AST patterns or diagnostics.

8.4.4 Building a Parser Testing Framework with Modern C++

Use Modern C++ features to make testing expressive, modular, and maintainable.

- **Core Components:**
  - `ParserTest` class for test registration and execution
  - `TestCase` abstraction using `std::function<void()>`
  - AST comparison using debug serialization or node fingerprints

– Assertion macros with C++20 `consteval` checks if necessary

Example:

```
class ParserTest {
public:
    static void register_case(std::string name, std::function<void()> fn);
    static void run_all();
};
```

Tests are declared like:

```
ParserTest::register_case("Parse binary expression", []() {
    std::string input = "x = 1 + 2 * 3;";
    auto ast = parse(input);

    assert(ast->to_string() == "(x = (1 + (2 * 3)))");
});
```

8.4.5 Token Stream Validation

To ensure tokens are correctly extracted:

```
std::string code = "int a = 5;";
Lexer lexer(code);
auto tokens = lexer.tokenize();

assert(tokens[0].type == TokenType::Int);
assert(tokens[1].lexeme == "a");
assert(tokens[3].literal == "5");
```

This catches problems in the tokenizer before they reach the parser.

8.4.6 Expression Parser Unit Tests

These tests verify precedence and associativity:

```
std::string input = "1 + 2 * 3";
auto expr = parse_expression(input);

assert(expr->to_string() == "(1 + (2 * 3))");
```

Test all operator combinations:

- Binary: `+ - * / %`
- Logical: `&& || !`
- Comparisons: `== != < > <= >=`
- Assignments: `= += -=`

8.4.7 Statement and Block Tests

Example: testing `if-else` parsing

```
std::string input = "if (x < 10) return x; else return 0;";
auto stmt = parse_statement(input);

assert(stmt->to_string().find("IfStmt") != std::string::npos);
```

Include:

- Block nesting depth
- Variable declaration scope inside `{ }`
- Combined structures like `while-if` or `if-return`

8.4.8 Error Reporting and Recovery Tests

Inject errors and ensure the parser:

- Detects the error
- Does not crash
- Recovers to the next safe token (`;`, `}`)

Example:

```
std::string input = "if x > 0 return 1;";
Parser parser(input);
parser.parse();

assert(parser.has_error());
assert(parser.recovered_tokens().size() > 0);
```

8.4.9 Automating Test Execution

Integrate a test runner in your development cycle:

- Run after each commit using a CMake test target
- Print summary output per test
- Fail fast on parsing regressions

Use C++20 features like `consteval` or concepts to enforce compile-time constraints when validating AST types or node contracts.

8.4.10 Lightweight Testing Libraries in C++20/23

While you can build a custom framework, C++20/23 works well with modern test libraries:

- **doctest**: Single header, expressive assertions, easy to embed
- **Catch2**: Popular and powerful with support for sections and data generators
- **GoogleTest**: Rich API, but heavier; great for large-scale validation

Example with doctest:

```
DOCTEST_TEST_CASE("Variable declaration parsing") {
    auto stmt = parse_statement("int y = 42;");
    CHECK(stmt->kind() == StmtKind::VarDecl);
}
```

These tools allow you to focus on grammar correctness rather than infrastructure.

8.4.11 Building a Grammar Regression Suite

Maintain a folder of `.test` or `.lang` files representing core grammar constructs:

- Each file is parsed and optionally printed as AST
- Run the parser across all files and verify no errors
- Used to guard against accidental regressions

Command-line test harness:

```
./lang_test_runner tests/grammar/*.lang
```

8.4.12 Conclusion

Integrated testing during parser development is not optional — it is essential. With the rise of C++20/23, your test system can be more expressive, robust, and efficient. Testing from token streams to full blocks ensures that your grammar remains stable, extensible, and accurate across evolving language features.
The parser is the compiler's lens into user intent. With continuous, structured testing, that lens remains sharp.

8.5 Milestone — Parser Generating Valid ASTs for C-Style Code

8.5.1 Introduction

This section marks a major **milestone** in the implementation journey of your custom C-style language: the ability of your parser to **generate valid Abstract Syntax Trees (ASTs)** from real-world C-style code input. At this stage, the parser transforms a linear token stream into a hierarchical structure that accurately reflects the **syntactic and semantic organization** of the source code. This validates the correct design of your grammar, expression handling, block scoping, and statement parsing.
This achievement enables the next major phases: semantic analysis, runtime interpretation, or code generation. It is also the point where testing, REPL interaction, and file-based compilation become functional.

8.5.2 What Defines a "Valid AST" in a C-Style Language

An Abstract Syntax Tree is *valid* when it satisfies the following:

- **Matches the original program structure**, respecting operator precedence, associativity, and nesting
- **Preserves source code semantics** (e.g., scope, control flow)
- **Has no structural gaps** (e.g., empty or null branches)
- **Uses consistent and unique node types** with accurate ownership
- **Can be walked recursively without special cases**

Example:

Source code:

```
int x = 10;
if (x > 5) {
    x = x + 1;
}
```

AST:

```
BlockStmt
  VarDeclStmt: int x = 10
  IfStmt
      Condition: (x > 5)
      BlockStmt
          ExprStmt: x = x + 1
```

8.5.3 AST Node Structure and Ownership

Your AST classes are now concrete and organized. With Modern C++:

- Nodes use `std::unique_ptr` to manage tree memory
- Node types are separated into `Expr`, `Stmt`, and derived classes
- Use `std::variant` or tagged base classes to model polymorphism

Example:

```
struct Stmt {
    virtual ~Stmt() = default;
    virtual std::string debug() const = 0;
};

struct BlockStmt : public Stmt {
    std::vector<std::unique_ptr<Stmt>> body;
    std::string debug() const override {
        std::string result = "{\n";
        for (auto& stmt : body)
            result += "  " + stmt->debug() + "\n";
        return result + "}";
    }
};
```

With this foundation, ASTs can be **easily traversed**, **debugged**, and **executed** later.

8.5.4 Parser Responsibilities at This Milestone

The parser is now fully capable of:

- Parsing **complete C-style programs**

- Differentiating between **declarations**, **statements**, and **expressions**
- Respecting **operator precedence** and **scoping rules**
- Recovering from syntax errors and continuing parsing
- Generating **non-null, structurally sound ASTs**

Covered Grammar Rules:

```
program         → declaration* EOF ;
declaration     → var_decl | statement ;
var_decl        → type IDENTIFIER ("=" expression)? ";" ;
statement       → if_stmt | while_stmt | block | expr_stmt | return_stmt ;
expression      → assignment ;
assignment      → IDENTIFIER "=" assignment | logic_or ;
logic_or        → logic_and ("||" logic_and)* ;
...
primary         → literal | IDENTIFIER | "(" expression ")" ;
```

All of the above must now be **fully implemented** and **mapped to AST nodes**.

8.5.5 Visual and Debug Tools for AST Inspection

As part of validating this milestone, a debug representation of the AST should be developed.

- **Debug Printer:**

```
std::string ExprStmt::debug() const {
    return expr->debug();
}
```

```
std::string BinaryExpr::debug() const {
    return "(" + left->debug() + " " + op + " " + right->debug() + ")";
}
```

- **Input:**

```
x = 1 + 2 * 3;
```

- **Output:**

```
(x = (1 + (2 * 3)))
```

This proves:

  - AST hierarchy is correct
  - Precedence was respected
  - No memory leaks or dangling pointers exist

8.5.6 Verifying AST Validity through Testing

To lock in this milestone, tests should confirm:

- Token sequences yield correct AST structure
- AST traversals produce expected behavior
- Serialization from AST to debug format is stable

- No incorrect or partial node generation
- **Example Test:**

```
TEST_CASE("Parse variable declaration with expression") {
    auto stmt = parse("int x = 5 + 2;");
    CHECK(stmt->debug() == "int x = (5 + 2)");
}
```

Use `std::variant` or `std::visit` (C++17/20) to handle node dispatching safely.

8.5.7 Integrated Use of C++20/23 Features at Milestone

Modern C++ simplifies AST construction and parser safety:

Table 5-4: Modern C++ Feature Usage in AST and Parser Design

Feature	Usage
`std::unique_ptr`	Ownership of AST nodes and blocks
`std::optional`	Optional `else` branches or `return` expressions
`std::variant`	Unified node containers if needed
`concepts`	Enforcing parser constraints for generic AST walkers
`ranges` / `views`	Scanning tokens efficiently during parsing
`consteval`	Compile-time validation of AST node traits (optional)

This leads to robust, clear, and maintainable code structures.

8.5.8 Final Checklist for Milestone

To consider this milestone complete, the following must be true:

- Complete set of AST node classes implemented
- Parser recognizes and transforms all C-style grammar constructs
- ASTs are printed or inspected in a debug-friendly format
- Parser tested against valid and invalid input
- Memory usage is safe (no leaks, no shared mutable state)
- Nodes are modular and extensible for future semantic checks

8.5.9 Preparing for Next Stages

With valid AST generation achieved, your interpreter can now:

- Perform **semantic analysis** (type checks, symbol tables)
- Begin **evaluating expressions and statements**
- Enable **step-by-step execution in REPL**
- Offer **code generation targets** like bytecode or LLVM IR in future stages

This milestone is the true “birth” of your language’s syntactic engine.

8.5.10 Conclusion

The ability of the parser to produce valid ASTs for complex C-style code is a foundational achievement in building a programming language. It confirms that your grammar, lexer, and parser cooperate correctly, and that your AST design can express all meaningful constructs your language intends to support.

Moving forward, this AST will be the input for semantic validation and runtime evaluation. This milestone unlocks the power to test, execute, and visualize real programs written in your language — a true moment of transition from theory to functionality.

Chapter 9

Advanced Parsing and C-Style Error Handling

9.1 Error Recovery in C-Style Syntax

9.1.1 Introduction

Error recovery is a critical yet often overlooked feature in compiler and interpreter design. In a C-style language—where block scopes, control flow, semicolon-terminated statements, and nested expressions dominate—the ability to **detect**, **report**, and **recover** from syntax errors without halting parsing is vital for a smooth development experience. This section focuses on implementing robust error recovery mechanisms for your parser using structured techniques and **Modern C++20/23** practices.

Error recovery ensures that the parser can **continue processing the remaining code** after encountering invalid input, producing as many useful error messages as possible in a single pass. Without recovery, a single malformed token would prevent parsing an entire program, degrading usability for both developers and tools.

9.1.2 Common Sources of Syntax Errors in C-Style Languages

A parser must gracefully handle errors from multiple sources:

Table 1-1: Common Syntax Error Sources

Error Source	Example
Missing semicolons	`int x = 5`
Unmatched braces or parentheses	`if (x > 0 {`
Misused keywords or identifiers	`return 5 5;`
Invalid assignment targets	`5 = x + 1;`
Incomplete expressions or blocks	`while (x < 10)`

Understanding these patterns allows you to write targeted synchronization strategies.

9.1.3 Two-Phase Strategy: Detection and Recovery

Error handling in parsing typically follows this structure:

1. **Detection**: Throw an error (or trigger an error state) when unexpected tokens occur.

2. **Recovery**: Skip tokens until a ”safe” state is found (a sync point), then resume parsing.

Example in Recursive Descent Parser:

```
StmtPtr parse_statement() {
    try {
        if (match(TokenType::If)) return parse_if_statement();
        // ...
        return parse_expression_statement();
    } catch (const ParseError&) {
        synchronize();
        return nullptr; // Recover with dummy node
    }
}
```

9.1.4 Implementing `synchronize()` Function

Synchronization seeks known **structural tokens** to realign parsing. These are usually:

- Statement terminators: `;`
- Block boundaries: `{`, `}`
- Statement starters: `if`, `while`, `for`, `return`, variable types

```
void synchronize() {
    advance(); // Skip the error token

    while (!is_at_end()) {
        if (previous().type == TokenType::Semicolon) return;

        switch (peek().type) {
            case TokenType::If:
            case TokenType::While:
            case TokenType::Return:
```

```
            case TokenType::Int:
            case TokenType::Float:
            case TokenType::Bool:
            case TokenType::LeftBrace:
                return;
        }

        advance();
    }
}
```

This allows parsing to resume at logical code boundaries after an error.

9.1.5 ParseError Exception Handling

Use a lightweight exception type (or `std::optional` return type) for managing error propagation.

```
class ParseError : public std::exception {
public:
    explicit ParseError(const std::string& msg) : message(msg) {}
    const char* what() const noexcept override { return message.c_str(); }
private:
    std::string message;
};

[[noreturn]] ParseError error(const Token& token, const std::string& message) {
    report_error(token, message);
    throw ParseError(message);
}
```

Benefit with C++20:

Using `[[noreturn]]` makes intent clear to static analyzers and tools, reducing branching errors and improving maintainability.

9.1.6 AST Recovery Node for Invalid Statements

Instead of discarding a broken statement, optionally return a special **InvalidStmt** node to preserve AST shape:

```
class InvalidStmt : public Stmt {
public:
    std::string reason;
    InvalidStmt(std::string msg) : reason(std::move(msg)) {}
    std::string debug() const override { return "[InvalidStmt: " + reason + "]"; }
};
```

This is useful for editor-based tools, AST visualizers, or static analyzers.

9.1.7 Structured Error Messaging and Token Context

Generate helpful messages with context, including token text and position:

```
void report_error(const Token& token, const std::string& message) {
    std::cerr << "[line " << token.line << "] Error at '" << token.lexeme << "': " <<
    ↪  message << "\n";
}
```

Extend this with file names or columns when your language supports them.

9.1.8 Example: Recovery from Missing Semicolon

- **Input:**

```
int x = 10
x = x + 1;
```

- **Output:**

```
[line 1] Error at 'x': Expected ';' after variable declaration
[line 2] Successfully resumed parsing
```

- **Parser Reaction:**
  - Error is thrown at line 1
  - `synchronize()` skips until it sees the start of the next statement
  - Parsing resumes normally at line 2

9.1.9 Modern C++ Integration for Safer Error Control

Table 1-2: Modern C++ Features in Error Recovery

Feature	Use in Error Recovery
`std::optional`	Return empty result instead of exceptions (lightweight)
`constexpr`	Encode token types or patterns statically
`ranges`	Scan ahead to find recovery points (`views::drop_while`)

Feature	Use in Error Recovery
`concepts`	Type-check token categories (e.g., `is_control_keyword`)
`source_location` (C++20)	Report error origin at compile-time (for parser generator tools)

These help build robust, declarative, and testable parser components.

9.1.10 Testing Error Recovery Scenarios

Include invalid input cases in your test suite to ensure resilience:

```
TEST_CASE("Recovery from bad if-statement") {
    auto result = parse("if x > 0 { print(x); }");
    CHECK(result != nullptr);  // Parsing should not crash
    CHECK(contains_error(result));
}
```

Each test verifies:

- The parser does not halt
- AST is partially recoverable
- Errors are reported accurately

9.1.11 Summary and Best Practices

Table 1-3: Parser Error Handling Principles

Principle	Recommendation
Fail early	Detect syntax errors as soon as possible
Recover gracefully	Resume parsing using synchronization heuristics
Preserve structure	Use dummy AST nodes to avoid parser crashes
Report clearly	Inform user of error type, location, and fix
Test thoroughly	Include malformed inputs in every test cycle

By adopting these strategies, your parser becomes more professional, user-friendly, and production-grade.

9.1.12 Conclusion

Error recovery in C-style syntax is not an afterthought; it is a cornerstone of resilient language design. With structured synchronization, expressive error diagnostics, and AST continuity, your language can provide meaningful feedback even when programs are incomplete or incorrect.

This section establishes the robustness of your parser and prepares it for integration with interactive tools like editors, REPLs, and debuggers. As a result, your parser not only recognizes structure—it respects the developer's experience.

9.2 Rich Error Messages for Common C-Style Mistakes

9.2.1 Introduction

Clear and **rich error messages** are a vital part of a programming language's usability and learning curve. Developers—especially those using new languages—depend heavily on the **quality of diagnostics** when encountering syntax mistakes. In this section, we focus on designing **helpful, actionable, and context-aware error messages** for common mistakes in a C-style syntax language. The approach is fully grounded in **Modern C++20/23**, using its features to produce efficient and readable diagnostics with minimal runtime overhead.
A rich error message does more than report a problem. It tells the developer:

- **What went wrong**
- **Where it happened**
- **What the parser expected**
- **How to fix it**

9.2.2 Categories of Common C-Style Syntax Mistakes

Understanding the **most frequent syntax mistakes** allows you to tailor your error reporting system.

Table 2-4: Common Syntax Mistakes and Diagnostic Insights

Mistake Type	Example	Expected Message Insight
Missing semicolon	`int x = 5`	“Expected ‘;’ after variable declaration”

Mistake Type	Example	Expected Message Insight
Unmatched brackets	`if (x > 5 {`	"Expected ')' to close condition expression"
Invalid assignment target	`3 = x + 2;`	"Invalid assignment target; must be identifier"
Confused operator	`x === 3`	"Unknown operator '==='; did you mean '=='?"
Unterminated block	`while (x < 5) {`	"Expected '}' to close block starting here"
Incomplete control flow syntax	`if x > 5 return x;`	"Expected '(' after 'if' keyword"

These types represent at least **80% of early parsing errors** in C-style languages.

9.2.3 Designing Human-Friendly Messages

Good error messages use **natural language** and **specific terminology**. They also follow these guidelines:

- **Describe what was expected**: "Expected `)` but found `{`"
- **Show what was found**: Include the actual token
- **Display location**: Show line number and optionally column or snippet
- **Provide suggestions**: Where applicable, show what might fix it

Example:

```
[line 12] Error: Expected ';' after expression
    --> x = 5
            ^ Unexpected end of line. Did you forget a semicolon?
```

9.2.4 Implementation in C++20/23: Error Reporter Design

Use a structured `ErrorReporter` class to encapsulate logic.

```
class ErrorReporter {
public:
    void syntax_error(const Token& token, std::string_view message);
    void suggestion(std::string_view hint);
};
```

Use `std::format` (C++20) for precise string formatting:

```
void ErrorReporter::syntax_error(const Token& token, std::string_view message) {
    std::cerr << std::format("[line {}] Error at '{}': {}\n",
                             token.line, token.lexeme, message);
}
```

To improve developer guidance, attach possible **code snippets** and **quick fixes** to the message:

```
error_reporter.syntax_error(token, "Expected ')' after condition");
error_reporter.suggestion("Example: if (x > 5) {...}");
```

9.2.5 Detecting Specific Patterns for Enhanced Diagnostics

In the parser, tailor each rule to include enhanced error messages:

- **Example 1: Missing Semicolon**

```
if (!match(TokenType::Semicolon)) {
    error(token, "Expected ';' after statement");
    suggestion("Insert a semicolon at the end of this line.");
}
```

- **Example 2: Invalid Expression as Assignment Target**

```
ExprPtr expr = parse_primary();
if (!is_assignable(expr)) {
    error(expr_token, "Invalid assignment target");
    suggestion("Only variables can appear on the left-hand side of '='.");
}
```

9.2.6 Enabling Context-Aware Suggestions

Use `source_location` (C++20) and `std::string_view` for context tracking.

- **Example:**

```
void report_mismatched_token(const Token& token, TokenType expected) {
    std::cerr << std::format(
        "[line {}] Error at '{}': Expected '{}'\n",
        token.line, token.lexeme, token_type_to_string(expected));
}
```

Create a mapping function `token_type_to_string(TokenType)` for readable output.

9.2.7 Providing Fix-It Hints (Optional for IDE Integration)

Design your diagnostics to support **automated fix suggestions**, useful for future editor integrations:

```
struct FixItHint {
    std::string insert;
    int position;
};

report_error_with_fix("Missing ';'", token, FixItHint{";", token.end_pos});
```

This enables integration with custom language servers or code editors in the future.

9.2.8 Example Message Enhancements

Table 2-5: Improving Diagnostic Messages with Suggestions

Problem	Weak Message	Rich Message + Suggestion
`int x = ;`	"Syntax error"	"Expected expression after '='. Did you forget a value?"
`if x > 5 {`	"Unexpected token"	"Expected '(' after 'if'. Example: if (x > 5) { … }"
`return`	"Invalid syntax"	"Return statement requires a value or semicolon"

Problem	Weak Message	Rich Message + Suggestion
== vs = confusion	"Unexpected operator"	"Did you mean '==' for comparison instead of '='?"

9.2.9 Building a Diagnostic Table for All Grammar Rules

As you expand your grammar, build a **diagnostic table** per rule:

```
struct DiagnosticEntry {
    std::string rule;
    std::string error_message;
    std::string fix_hint;
};
```

Populate this table manually or generate it from parser metadata. It provides centralized, maintainable error message control.

9.2.10 C++20 Features That Improve Diagnostics

Table 2-6: C++20/23 Features Benefiting Error Messaging

C++20/23 Feature	Benefit in Error Messaging
`std::format`	Clean formatting of messages and token data
`source_location`	Compiler-time code tracing for internal tools
`std::string_view`	Efficient manipulation of source slices

C++20/23 Feature	Benefit in Error Messaging
`concepts`	Improve compile-time error messages in metaprogramming
`[[nodiscard]]`	Alert developers about ignored error results

These features support writing diagnostics that are efficient and scalable.

9.2.11 Testing Diagnostic Accuracy

Create a **diagnostic testing suite** where malformed source lines produce expected error outputs.

- **Example:**

```
std::string input = "int x = ;";
auto result = parser.parse(input);
CHECK(result.has_error());
CHECK(result.error_message().contains("Expected expression after ‘=’"));
```

  Include all error categories in test coverage:

  - Statement terminators
  - Control flow heads (`if`, `while`)
  - Block matching
  - Operator misuse
  - Literal misuse

9.2.12 Conclusion

Designing rich error messages is not a cosmetic feature—it is a **core aspect of language usability and learning**. With clear messages, precise context, and helpful suggestions, developers using your language will be guided through their mistakes confidently. Rich diagnostics built with Modern C++20/23 not only improve user experience but also enable seamless integration with tooling ecosystems such as IDEs, REPLs, and language servers.

9.3 Parser Testing with C-Style Code Patterns

9.3.1 Introduction

A parser is the backbone of any compiler or interpreter. It must be reliable, consistent, and capable of handling all expected (and many unexpected) code patterns. For a C-style programming language, parser testing must cover both **syntactically valid constructs** and **invalid patterns** that mimic real-world mistakes. This section describes how to design a comprehensive, **automated parser test suite** for a modern interpreter project using **C++20/23**. It focuses on test structure, code coverage, assertion strategies, and integration with modern C++ testing libraries.

9.3.2 Purpose of Parser Testing

The parser's job is to transform a **linear token stream** into a **structured AST**. Testing ensures:

- Correct parsing of all language constructs
- Accurate precedence and associativity handling

- Robust error detection and recovery
- Structural integrity of the resulting AST
- Compatibility with future grammar expansions

9.3.3 C-Style Syntax Patterns to Cover

Parser tests must reflect realistic code usage and edge cases. These are grouped into:

- **a. Declarations**

```
int x;
float y = 3.14;
bool flag = true;
```

- **b. Expressions**

```
x + y * z;
(x + 3) / (z - 1);
x == 10 && y != 20;
```

- **c. Statements and Blocks**

```
{
    int x = 5;
    x = x + 1;
}
```

- **d. Control Flow**

```
if (x > 0) {
    x--;
} else {
    x++;
}

while (x < 10) {
    x += 2;
}
```

- **e. Errors and Incomplete Syntax**

```
int x = ;            // Missing expression
if x > 0             // Missing parentheses
{ return 1;          // Unmatched braces
```

9.3.4 Test Architecture in Modern C++

Use modern C++ testing libraries such as **doctest**, **Catch2**, or **GoogleTest**. Prefer modern features like:

- `std::string_view` for input slices
- `std::unique_ptr` and `std::variant` for AST integrity
- `constexpr` matchers (C++20)
- Compile-time grammar helpers

- **xample using Catch2:**

```
TEST_CASE("Parse simple variable declaration") {
    std::string input = "int x = 5;";
    Parser parser(input);
    auto ast = parser.parse();
    REQUIRE(ast != nullptr);
    CHECK(ast->debug() == "int x = 5;");
}
```

9.3.5 AST Structure Verification

Beyond parsing success, you must **validate the structure** of the generated AST.

- **Pattern Matching via `std::variant` or Visitor:**

```
CHECK(std::holds_alternative<VarDeclStmt>(*ast));
auto& decl = std::get<VarDeclStmt>(*ast);
CHECK(decl.name == "x");
CHECK(decl.type == VarType::Int);
CHECK(decl.initializer->debug() == "5");
```

9.3.6 Token Snapshot Tests

To ensure that parsing is triggered correctly, write **token stream snapshot tests** as a pre-parser check:

```
Lexer lexer("if (x > 1) { x = 2; }");
auto tokens = lexer.tokenize();
CHECK(tokens.size() == 11);
```

```
CHECK(tokens[0].type == TokenType::If);
CHECK(tokens[3].lexeme == ">");
```

This helps isolate lexer/token bugs from parser issues.

9.3.7 Expression Parsing with Precedence

You must test all major operator categories in combinations to verify precedence and associativity:

```
TEST_CASE("Operator precedence: addition and multiplication") {
    auto ast = parser.parse("1 + 2 * 3;");
    CHECK(ast->debug() == "(1 + (2 * 3))");
}
```

Build tests for:

- Logical: `&&`, `||`
- Comparison: `==`, `!=`, `<`, `>`
- Arithmetic: `+`, `-`, `*`, `/`
- Unary: `-x`, `!x`
- Grouping: `(x + y)`

9.3.8 Error Pattern Testing

Test not only valid input but also **faulty** and **ambiguous** code. Each test must:

- Trigger the correct error

- Ensure the parser **recovers** and parses subsequent code
- Verify the number and kind of errors reported
- **Example:**

```
TEST_CASE("Missing semicolon in declaration") {
    Parser parser("int x = 10");
    auto ast = parser.parse();
    CHECK(ast == nullptr);
    CHECK(error_reporter.count() == 1);
    CHECK(error_reporter.latest_message().contains("Expected ';'"));
}
```

9.3.9 Parser Test Utility Functions

Design test helpers for parsing and AST validation:

```
auto parse_stmt(std::string_view code) -> std::unique_ptr<Stmt>;
auto parse_expr(std::string_view code) -> std::unique_ptr<Expr>;

void assert_expr_debug(const std::string& code, const std::string& expected_debug);
```

These make your test cases **cleaner and more maintainable**.

9.3.10 Using C++20 Features in Parser Tests

Table 3-7: C++20/23 Features Supporting Testing and Verification

C++20/23 Feature	Use Case in Testing
`std::format`	Formatted debug output and assertion error printing
`std::ranges`	Analyzing tokens, validating AST shape
`std::span`	Slicing tokens for inline test assertions
`consteval`	Build-time grammar assertions (advanced use)
`concepts`	Enforce AST node types during test construction

These improve clarity, performance, and error visibility during testing.

9.3.11 Milestone: Confidence in Parser Robustness

Once you implement tests for:

- Declarations, expressions, and control structures
- Operator precedence and associativity
- Syntax error handling and recovery
- Edge cases and partial input

You can consider your parser **verified for correctness**. The test suite becomes your safety net as the language evolves, ensuring that even with future feature additions or optimizations, your parser remains predictable, reliable, and structurally sound.

9.3.12 Conclusion

Parser testing is more than a quality assurance task—it is a form of **language specification in code**. The coverage and clarity of these tests reflect the depth of your language's grammar design. By building comprehensive tests around real-world C-style code patterns and incorporating modern C++20/23 capabilities, you ensure the long-term integrity of your language's core: the parser. This lays a solid foundation for semantic analysis, interpretation, and toolchain integration in the next development phases.

9.4 Milestone – Robust Parser with Excellent C-Style Error Reporting

9.4.1 Introduction

This section marks a major milestone in the language implementation journey: **a fully functional, fault-tolerant parser** that can process realistic C-style code and provide **clear, rich, and actionable error diagnostics**. This parser becomes the backbone for the upcoming phases—semantic analysis, interpretation, and optimization. Achieving this milestone means more than passing syntax trees through tests. It requires precision in grammar parsing, **robust error detection and recovery**, and integration of **modern C++20/23 features** for expressive diagnostics and reliable performance.

9.4.2 Defining "Robustness" in a Parser

A **robust parser** is characterized by:

- The ability to **accurately parse** valid C-style code into an AST

- Handling malformed or partial input **gracefully**
- **Continuing** parsing after encountering errors (not halting at first issue)
- Providing **clear, context-aware** diagnostics
- **Maintaining structural integrity** of the AST despite recoveries
- Enabling **automated testing** of parser behavior under normal and error scenarios

Robustness is the quality that transforms a basic parser into a tool ready for integration into IDEs, linters, compilers, and interpreters.

9.4.3 Key Features of the Final Parser at This Stage

At this milestone, the parser should support:

Table 4-8: Key Features of C-Style Parser Implementation

Feature	Description
Full grammar for C-style constructs	Declarations, blocks, control flow, expressions
Precedence-correct expression parsing	Operators grouped with correct associativity
Braced block parsing	`{}`-scoped statements with nested structures
Statement terminators	Detection of missing `;` and block delimiters
Synchronization recovery	Continue parsing after error via token skipping

Feature	Description
Rich diagnostics	Specific error messages with line/token context
AST construction	Structure generated with `unique_ptr` or `variant`
Token feedback loop	Integration with lexer and token stream navigation

9.4.4 Unified Error Reporting Infrastructure

The parser now includes a **centralized error reporting mechanism** that distinguishes:

- **Syntax errors** (unexpected tokens, missing parts)
- **Semantic hints** (suggested corrections)
- **Recovery notes** (warnings about skipped regions)
- **Sample error output:**

```
[line 7] Error: Expected ';' after variable declaration
    --> int x = 5
                ^

Note: Insert a semicolon to terminate the statement
```

This is enabled using `std::format` (C++20), structured `ErrorReporter` classes, and clear token metadata.

9.4.5 Error Resilience with Synchronization Techniques

Using synchronization points (like `;`, `{`, `}`, `if`, `while`, `for`) ensures the parser skips over malformed input and resumes parsing cleanly.

- **Example Code:**

```
void synchronize() {
    while (!is_at_end()) {
        if (previous().type == TokenType::Semicolon) return;

        switch (peek().type) {
            case TokenType::If:
            case TokenType::While:
            case TokenType::For:
            case TokenType::Return:
            case TokenType::LeftBrace:
            case TokenType::RightBrace:
                return;
        }
        advance();
    }
}
```

This technique is now **integrated in all statement-level parsing rules**, ensuring graceful recovery.

9.4.6 AST Structural Integrity

In error cases, rather than returning `nullptr`, the parser now returns **dummy nodes** (`InvalidStmt`, `ErrorExpr`) to maintain AST continuity. This is essential for tools that analyze or visualize the AST even in broken code.

```
class InvalidStmt : public Stmt {
public:
    std::string message;
    InvalidStmt(std::string m) : message(std::move(m)) {}
    std::string debug() const override { return "[InvalidStmt: " + message + "]"; }
};
```

9.4.7 Parser Test Coverage at Milestone

Your test suite should now include:

- Valid C-style declarations
- Complex nested expressions and blocks
- All major control flows (`if`, `while`, `for`, `return`)
- Error conditions: missing tokens, unmatched braces, invalid assignments
- Recovery tests: continuing after a failure
- AST validation checks

Each test case confirms **both correct output and correct error reporting**.

9.4.8 Using C++20/23 to Improve Parser Quality

Table 4-9: C++20/23 Features Beneficial for Parser Implementation

Feature	Benefit in Parser Implementation
`std::variant`	Represent AST node alternatives cleanly
`std::format`	Elegant and readable error messages
`std::source_location`	Track source of internal parser errors (debug/dev mode)
`std::ranges`	Cleanly manipulate tokens when scanning for recovery points
`[[nodiscard]]`	Prevent silent error ignoring in parse return values

These features lead to readable, maintainable, and type-safe parser design.

9.4.9 Debug Mode with Verbose Token and Parse Logs

At this stage, a **debug mode** can be toggled (via CLI or config) to print:

- Token stream
- AST hierarchy
- Errors with their recovery actions

This allows you and contributors to diagnose parser behavior in fine detail.

```
if (debug_mode) {
    std::cout << "Parsed Expression Node: " << expr->debug() << "\n";
}
```

9.4.10 Code Snapshot: Final Parser API Example

```
class Parser {
public:
    explicit Parser(std::vector<Token> tokens);
    std::vector<std::unique_ptr<Stmt>> parse();
    bool has_errors() const;
    const std::vector<ParseError>& errors() const;
};
```

The parser now exposes its diagnostics externally, enabling integration with REPL, editors, and test systems.

9.4.11 Milestone Summary Checklist

Table 4-10: Parser Feature Implementation Status

Feature or Goal	Status
Full C-style statement parsing	True
Expression precedence and associativity	True
Error recovery and synchronization	True
AST construction (even after errors)	True

Feature or Goal	Status
Specific and rich error messages	True
Integration-ready parser API	True
Modular and testable parser design	True

This milestone establishes the **core reliability** of your language's front-end.

9.4.12 Conclusion

Reaching this parser milestone is a turning point. From this foundation, your language becomes *usable*, *educational*, and *extendable*. Thanks to Modern C++ features and structured design, you now have a production-ready parser that balances grammar accuracy, error clarity, and resilience.

The next stages—type checking, evaluation, and runtime management—can now build upon this solid parsing layer with full confidence in the integrity and clarity of your program representation.

Part IV

Evaluation Engine

Chapter 10

Value System for C-Style Types

10.1 Type System – `int`, `float`, `bool`, `string`, `arrays`

Designing a reliable and efficient type system is the foundational step in implementing a value system for a C-style programming language. This section discusses how to define, manage, and evaluate primitive and compound types (`int`, `float`, `bool`, `string`, and `arrays`) using idiomatic and modern C++20/23 techniques. The focus is on establishing an internal representation that balances runtime flexibility with type safety and extensibility.

10.1.1 Core Goals of the Type System

- **Clarity**: Simple, well-understood primitives familiar to C/C++ programmers.
- **Extensibility**: Ability to easily expand to support custom structs, enums, and more complex user-defined types.
- **Performance**: Efficient memory and runtime behavior with optional optimization paths.

- **Compatibility**: Interoperability with modern C++ constructs like `std::variant`, `constexpr`, `concepts`, and `modules`.

10.1.2 Defining the `Value` Type

All types must be encapsulated in a single runtime-dispatchable `Value` class, commonly implemented via `std::variant` in C++17+, but more effectively enhanced using `std::variant`, `std::visit`, and `constexpr if` with C++20/23.

```
using IntType    = int64_t;
using FloatType  = double;
using BoolType   = bool;
using StringType = std::string;
using ArrayType  = std::vector<Value>; // Recursive definition

struct Value {
    using VariantType = std::variant<
        IntType,
        FloatType,
        BoolType,
        StringType,
        ArrayType
    >;

    VariantType data;

    // Constructors for implicit conversion
    Value(IntType v)    : data(v) {}
    Value(FloatType v)  : data(v) {}
    Value(BoolType v)   : data(v) {}
    Value(const StringType& v) : data(v) {}
    Value(const ArrayType& v)  : data(v) {}
```

```
    template<typename T>
    bool is() const {
        return std::holds_alternative<T>(data);
    }

    template<typename T>
    T& get() {
        return std::get<T>(data);
    }
};
```

10.1.3 Primitive Types Implementation

1. `int`
    - 64-bit signed integers (`int64_t`) ensure uniformity across platforms.
    - Operations: addition, subtraction, multiplication, division, modulus, comparisons.
    - Cast support to `float` and `bool`.
2. `float`
    - Use `double` precision (`double` is preferred over `float` due to improved numerical stability).
    - Implicit widening from `int`.
    - Comparisons and arithmetic must handle IEEE floating-point quirks (e.g., NaN, $\pm\infty$).
    - Use `std::isnan`, `std::isinf` where needed.

3. **`bool`**

    - Stored as `bool`, but in evaluation:
        - `0` and `0.0` become `false`.
        - Non-zero values become `true`.
    - Boolean short-circuiting must be built at the AST evaluation level, not the `Value` level.

4. **`string`**

    - Internally represented using `std::string` for UTF-8 support and STL compatibility.
    - Support concatenation, equality, indexing, and slicing.
    - Literal support via string interning for performance optimization (optionally using `unordered_map`).

10.1.4 Arrays

Arrays are homogenous in C-style syntax but for interpreter flexibility, can be **runtime-homogenous or heterogeneous**.

- **Internal Representation:**

```
using ArrayType = std::vector<Value>;
```

- **Key Features:**
    - Dynamic sizing (`push_back`, `resize`, etc.)

  - Index-based access and bounds checking
  - Element-wise operations (copy, map, filter)

- **Optional Enhancements:**
  - Type-checked arrays: Add metadata for static-like type checking.
  - Slicing: Provide subviews or shallow-copies via `std::span` or custom slice class.

10.1.5 Type Promotion and Compatibility Rules

```
Value promote(const Value& lhs, const Value& rhs) {
    if (lhs.is<FloatType>() || rhs.is<FloatType>())
        return FloatType{ lhs.toFloat() + rhs.toFloat() };
    else if (lhs.is<IntType>() && rhs.is<IntType>())
        return IntType{ lhs.get<IntType>() + rhs.get<IntType>() };
    // Add rules for string, bool, etc.
}
```

Implement type promotion rules to allow natural C-like expression resolution:

Table 1-1: Type Promotion Rules

LHS	RHS	Result
int	int	int
int	float	float
float	int	float
bool	int	int

LHS	RHS	Result
bool	float	float
string	any	string (concat)

Use `std::visit` with lambdas and `if constexpr` to write compile-time dispatch logic efficiently.

10.1.6 Using Concepts for Type-Safe Operations (C++20/23)

```
template<typename T>
concept Numeric = std::is_same_v<T, IntType> || std::is_same_v<T, FloatType>;

template<Numeric T>
T add(T lhs, T rhs) {
    return lhs + rhs;
}
```

This enforces compile-time restrictions when extending the language interpreter's core operators.

10.1.7 Type Inspection and Debugging Utilities

```
std::string typeName(const Value& v) {
    return std::visit([](auto&& arg) -> std::string {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, IntType>) return "int";
        else if constexpr (std::is_same_v<T, FloatType>) return "float";
        else if constexpr (std::is_same_v<T, BoolType>) return "bool";
```

```
        else if constexpr (std::is_same_v<T, StringType>) return "string";
        else if constexpr (std::is_same_v<T, ArrayType>) return "array";
        else return "unknown";
    }, v.data);
}
```

10.1.8 Interfacing with AST Evaluation

Each node of the AST will return a `Value`, and expression nodes will perform type-checked operations on these. Control flow structures (`if`, `while`, etc.) expect `bool` values, and failure to evaluate to `bool` must trigger runtime errors with helpful diagnostics.
Example:

```
if (!cond.is<BoolType>())
    throw std::runtime_error("Condition must evaluate to a boolean.");
```

10.1.9 Future-Proofing and Extensibility

- Add support for:
  - Structs (`std::unordered_map<std::string, Value>`)
  - Enums (backed by `int` or strings)
  - Nullable types (`std::optional<Value>`)
- Support C++23's `std::expected` for error-aware evaluations.
- C++23 `deducing this` and `explicit object parameters` can simplify fluent API for `Value`.

10.1.10 Conclusion

This section establishes a flexible yet powerful `Value` type to model the foundational types (`int`, `float`, `bool`, `string`, `array`) of a new C-style language. Using modern C++20/23 constructs such as `std::variant`, `concepts`, and compile-time checks ensures that the interpreter core remains maintainable, fast, and safe, while offering room to scale the type system toward more complex language features in future chapters.

10.2 Type Checking and Conversion in C-Style Context

C-style languages like C, C++, and their derivatives are known for their flexible but error-prone type system. In a modern interpreter inspired by such languages, designing **a robust and predictable type checking and conversion mechanism** is essential to support dynamic evaluation, meaningful diagnostics, and behavior faithful to the original C-style semantics. In this section, we explore **compile-time and runtime type analysis**, **implicit and explicit conversions**, and how to implement these mechanisms effectively using C++20 and C++23 features.

10.2.0.1 2.1 Overview of C-Style Type Semantics

In C-style languages:

- **Implicit type conversions** (also called *coercions*) are common in expressions involving mixed types.
- **Type promotion** occurs in arithmetic (e.g., int to float).

- **Boolean context** uses zero/non-zero or null/non-null rules rather than strict `bool` types.
- **Weak typing** allows risky or silent conversions unless restricted by language design.

Your interpreter must choose:

- **Strict type checking** with explicit casts only, or
- **Relaxed C-style typing** with implicit promotions and conversions.

This section assumes the latter, following familiar C/C++ behavior.

10.2.0.2 2.2 Internal Type System for the Interpreter

All values are wrapped in a `Value` object. This object is built using `std::variant` to represent one of the defined types (`int`, `float`, `bool`, `string`, `array`).

```
using ValueType = std::variant<int64_t, double, bool, std::string,
↪  std::vector<Value>>;
```

A `TypeKind` enum may be maintained to simplify type comparison without triggering variant visits:

```
enum class TypeKind { Int, Float, Bool, String, Array };

TypeKind getTypeKind(const Value& v);
```

This allows quick dispatching during type checking.

10.2.0.3 2.3 Implicit Type Conversion Rules

Following C-style rules, your interpreter should define **priority levels** for types in expression evaluation:

Table 2-2: Type Promotion Rank

Rank	Type
1	float
2	int
3	bool

When two types are involved:

- Promote `bool` → `int`
- Promote `int` → `float`
- Never promote `string` implicitly
- `string` participates only in concatenation or explicit casts

Example Conversion Logic:

```
Value promoteForArithmetic(const Value& lhs, const Value& rhs) {
    if (lhs.is<double>() || rhs.is<double>())
        return Value(lhs.toFloat() + rhs.toFloat());
    else if (lhs.is<int64_t>() && rhs.is<int64_t>())
        return Value(lhs.get<int64_t>() + rhs.get<int64_t>());
    else if (lhs.is<bool>() && rhs.is<bool>())
        return Value(static_cast<int64_t>(lhs.get<bool>()) +
        ↪  static_cast<int64_t>(rhs.get<bool>()));
```

```
    else
        throw std::runtime_error("Unsupported types in arithmetic expression.");
}
```

10.2.0.4 2.4 Explicit Conversion and Casting

Support explicit casting through a built-in `cast()` operation in the language syntax:

```
cast(x, "float")
```

In the interpreter:

```
Value castTo(const Value& v, TypeKind target) {
    switch (target) {
        case TypeKind::Int:
            if (v.is<int64_t>()) return v;
            if (v.is<double>()) return static_cast<int64_t>(v.get<double>());
            if (v.is<bool>()) return static_cast<int64_t>(v.get<bool>());
            break;

        case TypeKind::Float:
            if (v.is<double>()) return v;
            if (v.is<int64_t>()) return static_cast<double>(v.get<int64_t>());
            if (v.is<bool>()) return static_cast<double>(v.get<bool>());
            break;

        case TypeKind::Bool:
            if (v.is<bool>()) return v;
            if (v.is<int64_t>()) return static_cast<bool>(v.get<int64_t>());
            if (v.is<double>()) return static_cast<bool>(v.get<double>());
            break;
```

```
        case TypeKind::String:
            return Value(to_string(v)); // defined separately

        default:
            throw std::runtime_error("Invalid cast.");
    }
    throw std::runtime_error("Unsupported cast operation.");
}
```

Modern C++ enables this conversion logic to be simplified with `std::visit`, lambdas, and `if constexpr`.

10.2.0.5 2.5 Type Checking in Expressions

The interpreter should perform **runtime type checking** for:

- Binary operations (e.g., `+`, `-`, `*`, `/`)
- Comparison operators (`==`, `!=`, `<`, `>`)
- Logical operators (`&&`, `||`)
- Function arguments
- Conditional expressions

Use this strategy:

1. Validate expected types.
2. Promote types if needed.
3. Execute operation.

Example for addition:

```
Value evaluateAdd(const Value& lhs, const Value& rhs) {
    if (lhs.is<std::string>() || rhs.is<std::string>())
        return Value(to_string(lhs) + to_string(rhs));

    if (lhs.is<double>() || rhs.is<double>())
        return Value(lhs.toFloat() + rhs.toFloat());

    if (lhs.is<int64_t>() && rhs.is<int64_t>())
        return Value(lhs.get<int64_t>() + rhs.get<int64_t>());

    throw std::runtime_error("Addition not supported for these types.");
}
```

10.2.0.6 2.6 Boolean Context Evaluation

In C-style languages, non-zero and non-null values are treated as `true`. To implement this:

```
bool evaluateAsBool(const Value& v) {
    return std::visit([](auto&& val) -> bool {
        using T = std::decay_t<decltype(val)>;
        if constexpr (std::is_same_v<T, bool>) return val;
        else if constexpr (std::is_same_v<T, int64_t>) return val = 0;
        else if constexpr (std::is_same_v<T, double>)return val=0.0;
        else if constexpr (std::is_same_v<T, std::string>) return val.empty();
        else if constexpr (std::is_same_v<T, std::vector<Value>>)return val.empty();
        else return false;
    }, v.data);
}
```

This enables evaluating conditions in `if`, `while`, and logical expressions.

10.2.0.7 2.7 Error Diagnostics and Type Mismatch Handling

For better error reporting, use type introspection:

```
void expectType(const Value& v, TypeKind expected, const std::string& context) {
    if (getTypeKind(v) != expected) {
        throw std::runtime_error("Type error in " + context + ": expected " +
                                 to_string(expected) + ", got " + typeName(v));
    }
}
```

Helpful diagnostics are key for a better developer experience.

10.2.0.8 2.8 Leveraging C++20/23 Features

Modern C++ aids type checking and conversion logic:

- `consteval` and `constexpr` to optimize type logic at compile-time for literals and constant folding.
- `std::visit` with lambdas and `if constexpr` for clean and efficient dispatch.
- `concepts` to restrict template operations during compile-time type-safe transformations.
- `std::expected` (C++23) to return value or error when casting or evaluating.

Example using `std::expected`:

```
std::expected<Value, std::string> safeCastToInt(const Value& v) {
    if (v.is<int64_t>()) return v;
    if (v.is<double>()) return static_cast<int64_t>(v.get<double>());
    return std::unexpected("Cannot cast to int.");
}
```

10.2.0.9 Conclusion

C-style type systems are permissive but demand a clear, internally consistent interpretation strategy when reimplemented. With C++20/23, we can build a value system that reflects the flexibility of C-style conversions, while adding the safety and diagnostics of modern interpreter design. This section formalizes runtime type checking, implicit and explicit conversions, type promotions, and contextual evaluations in conditions or expressions — forming the backbone of a realistic evaluation engine for your new language.

10.3 Value Operations Matching C Semantics

To faithfully replicate the behavior of C-style languages in a modern interpreter, **value operations**—including arithmetic, logical, relational, and assignment semantics—must mirror those of the C language standard. This requires handling type promotions, operator precedence, overflow behavior, short-circuit logic, and comparison logic with precision. This section presents a detailed, modern C++20/23-based implementation of such value operations using `std::variant`, `std::visit`, and advanced dispatching logic.

10.3.1 Core Objectives

- **Match C's behavior** for mixed-type expressions, boolean evaluations, and operator effects.
- **Support overload dispatch** for binary and unary operators using clean and efficient code.
- **Enable diagnostics and error reporting** for invalid or undefined operations.
- **Preserve precision and overflow behavior** consistent with C's runtime model.

10.3.2 Unified Operator Dispatcher Using `std::variant` and Lambdas

Each operator (e.g., `+`, `-`, `*`, `/`, `==`, `<`, `!`, `&&`, `||`) is implemented as a standalone function that accepts `Value` instances and internally performs:

- **Type inspection**
- **Promotion (if needed)**
- **Evaluation**
- **Result construction**

```
Value applyAdd(const Value& lhs, const Value& rhs) {
    return std::visit([](auto&& a, auto&& b) -> Value {
        using A = std::decay_t<decltype(a)>;
        using B = std::decay_t<decltype(b)>;
```

```
        if constexpr (std::is_same_v<A, std::string> || std::is_same_v<B,
        ↪  std::string>) {
            return std::string(to_string(a)) + std::string(to_string(b));
        } else if constexpr ((std::is_arithmetic_v<A> || std::is_same_v<A, bool>) &&
                             (std::is_arithmetic_v<B> || std::is_same_v<B, bool>)) {
            if constexpr (std::is_same_v<A, double> || std::is_same_v<B, double>)
                return static_cast<double>(a) + static_cast<double>(b);
            else
                return static_cast<int64_t>(a) + static_cast<int64_t>(b);
        } else {
            throw std::runtime_error("Invalid operands for +");
        }
    }, lhs.data, rhs.data);
}
```

Use this pattern for every C-style operator. It allows:

- Proper handling of `string + anything` as string concatenation.
- `int + float`, `bool + int`, etc., promoted correctly.
- Early failure for unsupported operations.

10.3.3 Arithmetic Operators: `+`, `-`, `*`, `/`, `%`

Key Rules:

- Promote `bool` → `int` → `float`.
- `%` is valid only for integer types.
- Division by zero detection must mirror C's undefined behavior by throwing runtime exceptions.

```
Value applyMod(const Value& lhs, const Value& rhs) {
    if (lhs.is<int64_t>()||rhs.is<int64_t>())
        throw std::runtime_error("Modulo requires integer operands");
    auto b = rhs.get<int64_t>();
    if (b == 0)
        throw std::runtime_error("Modulo by zero");
    return lhs.get<int64_t>() % b;
}
```

Modern C++ enables custom overload sets using concepts if extended to static dispatch.

10.3.4 Comparison Operators: `==`, `!=`, `<`, `>`, `<=`, `>=`

These operations **must return `bool`**, not `int`. Behavior must match:

- **Integers** compared numerically.
- **Floats** follow IEEE comparison semantics.
- **Strings** compare lexicographically.
- **Arrays** may compare by size or content (optionally unsupported for simplicity).

```
Value applyEqual(const Value& lhs, const Value& rhs) {
    return std::visit([](auto&& a, auto&& b) -> bool {
        using A = std::decay_t<decltype(a)>;
        using B = std::decay_t<decltype(b)>;

        if constexpr (std::is_same_v<A, B>) {
            return a == b;
        } else if constexpr (std::is_arithmetic_v<A> && std::is_arithmetic_v<B>) {
```

```
            return static_cast<double>(a) == static_cast<double>(b);
        } else {
            return false; // Different types not equal
        }
    }, lhs.data, rhs.data);
}
```

10.3.5 Logical Operators: `&&`, `||`, `!`

C-style logical operators require **short-circuit evaluation**, which must be handled at the **AST evaluation level**, not inside `Value`.

```
Value evalLogicalAnd(const Value& lhsExpr, std::function<Value()> rhsEval) {
    if (!evaluateAsBool(lhsExpr)) return Value(false);
    return Value(evaluateAsBool(rhsEval()));
}
```

This ensures `rhsEval()` is not evaluated if `lhsExpr` is false, mimicking:

```
if (a && b) // b is not evaluated if a is false
```

10.3.6 Unary Operators: `-`, `+`, `!`

Unary operators act on a single operand. Implement them using `std::visit`.

```
Value applyUnaryNegate(const Value& v) {
    return std::visit([](auto&& val) -> Value {
        using T = std::decay_t<decltype(val)>;
        if constexpr (std::is_same_v<T, int64_t>)
            return -val;
```

```
        else if constexpr (std::is_same_v<T, double>)
            return -val;
        else
            throw std::runtime_error("Invalid operand for unary minus");
    }, v.data);
}
```

10.3.7 Assignment and Compound Assignment: =, +=, -=, etc.

The interpreter must support assignment expressions that:

- Store a computed `Value` in a symbol table.
- Respect C-like l-value rules (e.g., only assignable variables or dereferenced arrays).

```
Value assign(SymbolTable& symbols, const std::string& name, const Value& rhs) {
    symbols[name] = rhs;
    return rhs;
}
```

Compound assignments combine binary logic and assignment:

```
Value applyPlusEquals(SymbolTable& symbols, const std::string& name, const Value&
↪   rhs) {
    auto& lhs = symbols[name];
    lhs = applyAdd(lhs, rhs);
    return lhs;
}
```

You can optimize these using references and in-place mutation.

10.3.8 String and Array Specific Operations

- **String + string** → Concatenation.
- **String[n]** → Character access (return as `string` or `int`).
- **Array[n]** → Element access, with bounds checking.
- **Array.length** → Expose `.length` or `size()` via built-in syntax.

10.3.9 Operator Precedence and Evaluation Order

Although not handled directly in the value system, it must be mentioned:

- Operator precedence must be enforced during AST construction.
- Evaluation order (left-to-right, short-circuit) is defined in the parser/evaluator logic.

10.3.10 Overflow and NaN Behavior

- **Integer overflow**: Can use C++ behavior (two's complement wraparound).
- **Floating-point**: Leverage `std::isnan`, `std::isinf` from `<cmath>` to detect special cases.
- Interpreter can be extended with runtime flags for "strict mode" to throw on overflow.

10.3.11 Leveraging Modern C++ Features

C++20/23 tools that strengthen the implementation:

- `std::visit` and `if constexpr`: Type-safe dispatch.
- `concepts`: Restrict operator templates.
- `constexpr` functions: For constant folding in expressions.
- `consteval` (C++20): For compile-time evaluation of constant expressions.
- `std::expected` (C++23): For safe operator evaluation with error propagation.

10.3.12 Conclusion

This section establishes a robust, clean, and modular set of **C-style value operations** implemented using the full expressive power of modern C++20/23. Arithmetic, comparison, logic, and assignment are handled through type-aware dispatch, safe promotion rules, and behavioral mirroring of C's well-understood semantics. This allows your interpreter to behave as a predictable C-style runtime while being powered by the safety, clarity, and modularity of modern C++.

10.4 Hands-on — Value System with C-Style Type Behavior

10.4.0.1 Introduction

This hands-on section translates theory into practice by building a **fully operational value system** that captures the dynamic runtime behavior of a typical C-style language. You will use **modern C++20/23 features** to implement a flexible, type-safe, and extensible runtime type system capable of handling arithmetic, logical, relational, and assignment operations in an interpreter context. The code is designed

to be integrated directly into the evaluation engine and AST traversal logic of your language.

10.4.1 Defining the `Value` System

The `Value` type must support these runtime types:

- `int` (64-bit signed)
- `float` (`double`)
- `bool`
- `string` (`std::string`)
- `array` (`std::vector<Value>`)

Core Representation

```
#include <variant>
#include <string>
#include <vector>
#include <iostream>
#include <stdexcept>
#include <cmath>

struct Value;

using IntType    = int64_t;
using FloatType  = double;
using BoolType   = bool;
using StringType = std::string;
using ArrayType  = std::vector<Value>;
```

```
using ValueVariant = std::variant<IntType, FloatType, BoolType, StringType,
↪  ArrayType>;

struct Value {
    ValueVariant data;

    Value(IntType v) : data(v) {}
    Value(FloatType v) : data(v) {}
    Value(BoolType v) : data(v) {}
    Value(const StringType& v) : data(v) {}
    Value(const ArrayType& v) : data(v) {}

    template<typename T>
    bool is() const {
        return std::holds_alternative<T>(data);
    }

    template<typename T>
    T& get() {
        return std::get<T>(data);
    }

    template<typename T>
    const T& get() const {
        return std::get<T>(data);
    }
};
```

10.4.2 Boolean Context Evaluation

```
bool toBool(const Value& v) {
    return std::visit([](auto&& val) -> bool {
        using T = std::decay_t<decltype(val)>;
        if constexpr (std::is_same_v<T, BoolType>) return val;
        else if constexpr (std::is_same_v<T, IntType>) return val = 0;
        else if constexpr (std::is_same_v<T, FloatType>) return val = 0.0;
        else if constexpr (std::is_same_v<T, StringType>) return val.empty();
        else if constexpr (std::is_same_v<T, ArrayType>) return val.empty();
        else return false;
    }, v.data);
}
```

Used in `if`, `while`, logical operations.

10.4.3 Arithmetic Operations (Example: Addition)

```
Value add(const Value& lhs, const Value& rhs) {
    return std::visit([](auto&& a, auto&& b) -> Value {
        using A = std::decay_t<decltype(a)>;
        using B = std::decay_t<decltype(b)>;

        if constexpr (std::is_same_v<A, StringType> || std::is_same_v<B, StringType>)
        ↪ {
            return Value(std::string(a) + std::string(b));
        } else if constexpr ((std::is_arithmetic_v<A> || std::is_same_v<A, BoolType>)
        ↪ &&
                             (std::is_arithmetic_v<B> || std::is_same_v<B,
                             ↪ BoolType>)) {
            if constexpr (std::is_same_v<A, FloatType> || std::is_same_v<B,
            ↪ FloatType>)
```

```
                return static_cast<FloatType>(a) + static_cast<FloatType>(b);
            else
                return static_cast<IntType>(a) + static_cast<IntType>(b);
        } else {
            throw std::runtime_error("Invalid operands for '+'");
        }
    }, lhs.data, rhs.data);
}
```

You can replicate this pattern for `sub`, `mul`, `div`, and `mod`.

10.4.4 Relational Comparison

```
Value equal(const Value& lhs, const Value& rhs) {
    return std::visit([](auto&& a, auto&& b) -> BoolType {
        using A = std::decay_t<decltype(a)>;
        using B = std::decay_t<decltype(b)>;
        if constexpr (std::is_same_v<A, B>) return a == b;
        else if constexpr (std::is_arithmetic_v<A> && std::is_arithmetic_v<B>)
            return static_cast<FloatType>(a) == static_cast<FloatType>(b);
        else return false;
    }, lhs.data, rhs.data);
}
```

This gives consistent behavior for numeric and string types. Extend this for `!=`, `<`, `>`, `<=`, `>=`.

10.4.5 Logical Operations

C-style logic uses short-circuit evaluation, so the interpreter must delay evaluating the right-hand expression using function wrapping.

```
Value logicalAnd(const Value& lhs, std::function<Value()> rhs) {
    if (!toBool(lhs)) return Value(false);
    return Value(toBool(rhs()));
}

Value logicalOr(const Value& lhs, std::function<Value()> rhs) {
    if (toBool(lhs)) return Value(true);
    return Value(toBool(rhs()));
}

Value logicalNot(const Value& v) {
    // return Value(!toBool(v)); // just for pass overleaf compiler
}
```

This supports constructs like:

```
if (a && b)  // b evaluated only if a is true
```

10.4.6 String Indexing and Array Access

```
Value index(const Value& container, const Value& idx) {
    if (!idx.is<IntType>())
        throw std::runtime_error("Index must be an integer");

    IntType i = idx.get<IntType>();

    if (container.is<StringType>()) {
        const auto& s = container.get<StringType>();
        if (i < 0 || i >= static_cast<IntType>(s.size()))
            throw std::runtime_error("String index out of bounds");
```

```
        return Value(std::string(1, s[i]));
    } else if (container.is<ArrayType>()) {
        const auto& arr = container.get<ArrayType>();
        if (i < 0 || i >= static_cast<IntType>(arr.size()))
            throw std::runtime_error("Array index out of bounds");
        return arr[i];
    } else {
        throw std::runtime_error("Cannot index this type");
    }
}
```

10.4.7 Assignment Simulation

You can simulate assignment via a variable map:

```
#include <unordered_map>

using SymbolTable = std::unordered_map<std::string, Value>;

void assign(SymbolTable& table, const std::string& name, const Value& value) {
    table[name] = value;
}

Value get(SymbolTable& table, const std::string& name) {
    if (table.find(name) == table.end())
        throw std::runtime_error("Undefined variable: " + name);
    return table[name];
}
```

10.4.8 Debugging and Type Introspection

```
std::string typeOf(const Value& v) {
    return std::visit([](auto&& val) -> std::string {
        using T = std::decay_t<decltype(val)>;
        if constexpr (std::is_same_v<T, IntType>) return "int";
        else if constexpr (std::is_same_v<T, FloatType>) return "float";
        else if constexpr (std::is_same_v<T, BoolType>) return "bool";
        else if constexpr (std::is_same_v<T, StringType>) return "string";
        else if constexpr (std::is_same_v<T, ArrayType>) return "array";
        else return "unknown";
    }, v.data);
}
```

10.4.9 Example Program Evaluation

Sample pseudocode using the above system:

```
SymbolTable vars;

assign(vars, "x", Value(42));
assign(vars, "y", Value(3.14));
assign(vars, "z", add(get(vars, "x"), get(vars, "y")));

std::cout << "z = " << get(vars, "z").get<FloatType>() << "\n";
std::cout << "type(z) = " << typeOf(get(vars, "z")) << "\n";
```

Expected output:

```
z = 45.14
type(z) = float
```

10.4.10 Optional Enhancements

- Use `std::expected<Value, std::string>` instead of `throw` to allow graceful error propagation.
- Add `.length` property access support via member system.
- Introduce `consteval` evaluations for constant-folding in the AST.
- Hook into a `Visitor`-based evaluation model for full interpreter support.

10.4.11 Conclusion

This hands-on section forms the backbone of your interpreter's **dynamic evaluation engine**, reflecting the core of C-style behavior with **type-safe and modern** C++20/23 implementation. You now have a practical system for managing and evaluating values of various types in expressions, conditions, and function calls—mirroring the semantics of the C language while leveraging the expressive power of contemporary C++.

Chapter 11

Environment and C-Style Scoping

11.1 Symbol Table Design with C-Style Block Scoping

11.1.0.1 Introduction

In C-style programming languages, variables are scoped to blocks (`{ ... }`) and follow a *lexical scoping* model. A variable declared inside a block is visible only within that block and its sub-blocks. Redeclaring a variable in an inner block shadows the outer variable. This section explains how to design a **symbol table system** that reflects **block-level scoping semantics** and supports nested environments using **modern C++20/23 idioms** such as smart pointers, `std::unordered_map`, and concepts.

11.1.1 Goals of the Symbol Table

The symbol table must:

- Store and retrieve variables by name.

- Respect C-style block scoping rules.
- Allow variable shadowing in inner blocks.
- Support dynamic evaluation with efficient lookup.
- Facilitate future extension to function scopes and closures.

11.1.2 Lexical Scoping Model Recap

C-style scoping uses **static lexical blocks**, not dynamic runtime stack frames for local scope resolution.
Example:

```
int x = 10;
{
   int x = 20; // shadows outer x
   {
       int y = x; // refers to inner x
   }
}
```

Variables are resolved **from innermost to outermost scope**.

11.1.3 Core Structure: Chained Symbol Tables

We implement block scoping using **chained environments**, where each block has a symbol table that points to its enclosing scope.

Core Types

```
#include <unordered_map>
#include <memory>
#include <string>
#include <stdexcept>

struct Value; // Defined in the Value System

class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;

    Environment(Ptr parent = nullptr) : parent(parent) {}

    void declare(const std::string& name, const Value& value) {
        if (symbols.contains(name))
            throw std::runtime_error("Variable already declared in this block: " +
            ↪  name);
        symbols[name] = value;
    }

    void assign(const std::string& name, const Value& value) {
        if (symbols.contains(name)) {
            symbols[name] = value;
        } else if (parent) {
            parent->assign(name, value);
        } else {
            throw std::runtime_error("Undeclared variable: " + name);
        }
    }

    Value get(const std::string& name) const {
        if (symbols.contains(name)) {
```

```
            return symbols.at(name);
        } else if (parent) {
            return parent->get(name);
        } else {
            throw std::runtime_error("Undefined variable: " + name);
        }
    }

    bool isDeclaredInCurrentScope(const std::string& name) const {
        return symbols.contains(name);
    }

private:
    std::unordered_map<std::string, Value> symbols;
    Ptr parent;
};
```

This supports:

- Block-local declaration (`declare`)
- Variable assignment with resolution (`assign`)
- Lookup from nested to global (`get`)
- Shadowing check (`isDeclaredInCurrentScope`)

11.1.4 Creating and Disposing Scopes

At runtime, when entering a block (e.g. `{ ... }`), a **new environment is created** with a pointer to the outer (enclosing) one. When exiting the block, the inner environment is discarded.

```
void evaluateBlock(const std::vector<Statement>& stmts, Environment::Ptr outerEnv) {
    auto blockEnv = std::make_shared<Environment>(outerEnv);
    for (const auto& stmt : stmts) {
        evaluate(stmt, blockEnv);
    }
}
```

This naturally reflects the **lifetime and visibility** of local variables.

11.1.5 Handling Shadowing

The `declare()` method ensures that shadowing is **allowed only across scopes**, not within the same block.

```
blockEnv->declare("x", Value(10)); // OK
blockEnv->declare("x", Value(20)); // Error: already declared
```

In contrast, inner blocks may declare the same variable:

```
globalEnv->declare("x", Value(1));

auto inner = std::make_shared<Environment>(globalEnv);
inner->declare("x", Value(2)); // Shadows outer x

inner->get("x"); // returns 2
inner->get("y"); // fallback to outer environment
```

11.1.6 Using Concepts for Type Constraints (Optional)

To restrict environment use to `std::string` keys and `Value`-like entries:

```
template<typename T>
concept ValueLike = requires(T v) {
    { v.toString() } -> std::convertible_to<std::string>;
};

template<ValueLike V>
class Environment { ... };
```

This strengthens type safety and may be used for extending support to other symbol types such as functions, constants, or types.

11.1.7 Scope Levels and Debugging

To trace or debug block-level symbol states, you can instrument the environment:

```
void printScope(const Environment::Ptr& env, int level = 0) {
    if (!env) return;
    for (const auto& [key, val] : env->symbols)
        std::cout << std::string(level * 2, ' ') << key << " = " << val.toString() <<
        ↪  "\n";
    printScope(env->parent, level + 1);
}
```

This can help visualize variable resolution across blocks.

11.1.8 Global vs Local Environments

You may separate **global constants/functions** from dynamic block scopes:

```
class GlobalEnvironment : public Environment {
public:
```

```
    void defineNative(const std::string& name, const Value& v) {
        symbols[name] = v;
    }

    // prevent redefinition
    void declare(const std::string&, const Value&) = delete;
};
```

This is useful when building **REPLs** or scripting environments that retain global state across commands.

11.1.9 Integration with AST Evaluation

Each node in the AST should receive the current environment pointer:

```
Value evaluate(const Expression& expr, Environment::Ptr env);
void evaluate(const Statement& stmt, Environment::Ptr env);
```

Statements like `if`, `while`, and `block` introduce new environments; assignments and expressions refer to the current one.

11.1.10 Memory and Performance Considerations

- Use `std::shared_ptr` for shared scope graphs (preferred for interpreter lifecycle).
- Consider `std::unordered_map` with `reserve()` to avoid hash reallocation.
- You may optionally use `Arena` allocators for high-performance interpreter loops.

11.1.11 Conclusion

C-style block scoping can be implemented cleanly using **chained symbol tables** modeled with modern C++ features. This design supports nested blocks, variable shadowing, precise error reporting, and integrates directly with runtime evaluation. By managing environments with smart pointers and hash tables, the interpreter achieves both **accuracy in scoping semantics** and **runtime flexibility**, preparing the foundation for more advanced scoping structures like function closures and lexical environments in future chapters.

11.2 Variable Resolution Following C Rules

11.2.0.1 Introduction

In C-style programming languages, **variable resolution** adheres to well-defined lexical scoping rules. A variable is always resolved within the nearest enclosing scope where it is declared. If not found locally, the search proceeds up through the enclosing blocks, all the way to the global scope. This section focuses on implementing **variable name resolution** in a C-style interpreter using **modern C++20/23** principles, ensuring correctness, performance, and extensibility.

11.2.1 Overview of C Variable Resolution Semantics

In C and C-like languages:

- Variable declarations are **block-scoped**.
- Inner scopes **can shadow** outer variables.
- Variables must be **declared before use**, except in forward declarations (not relevant in expression-oriented interpreters).

- **No hoisting** (unlike JavaScript); the declaration must precede usage within the same scope.

Example in C:

```
int x = 10;
{
    int x = 20; // shadows outer x
    printf("%d", x); // prints 20
}
printf("%d", x); // prints 10
```

In your interpreter, the goal is to replicate this behavior exactly during **evaluation**.

11.2.2 Environment Model Recap

The interpreter uses a **chain of environments** to simulate the C-style scoping model. Each block creates a new `Environment` object that references its parent.

```
class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;

    Value get(const std::string& name) const;
    void assign(const std::string& name, const Value& value);
    void declare(const std::string& name, const Value& value);

private:
    std::unordered_map<std::string, Value> symbols;
    Ptr parent;
};
```

11.2.3 Variable Lookup (`get` Resolution)

When an expression refers to a variable by name, the interpreter must **resolve that name** according to the nearest enclosing scope rule.

```
Value Environment::get(const std::string& name) const {
    if (symbols.contains(name)) {
        return symbols.at(name);
    } else if (parent) {
        return parent->get(name);
    } else {
        throw std::runtime_error("Undefined variable: " + name);
    }
}
```

This ensures that:

- Local variables are accessed first.
- Outer variables are only considered if no local variable is found.
- An error is thrown if no matching declaration exists.

11.2.4 Variable Assignment (`assign` Resolution)

When assigning to a variable (e.g., `x = 42`), the interpreter must find the **nearest declared variable** with that name and update its value. If not found, it should throw an error (or define it globally, if the language permits).

```
void Environment::assign(const std::string& name, const Value& value) {
    if (symbols.contains(name)) {
        symbols[name] = value;
```

```
    } else if (parent) {
        parent->assign(name, value);
    } else {
        throw std::runtime_error("Assignment to undeclared variable: " + name);
    }
}
```

This enforces C's rule: **You cannot assign to a variable that hasn't been declared in any accessible scope.**

11.2.5 Declaring Variables (`declare`)

C does not allow redeclaring variables within the same block scope. The interpreter must reject re-declaration in the current block but allow shadowing across scopes.

```
void Environment::declare(const std::string& name, const Value& value) {
    if (symbols.contains(name)) {
        throw std::runtime_error("Variable already declared in this block: " + name);
    }
    symbols[name] = value;
}
```

Example:

```
global->declare("x", Value(10));     // OK
block->declare("x", Value(20));      // Shadows outer x
block->declare("x", Value(30));      // Error: duplicate in same block
```

11.2.6 Applying Resolution in AST Evaluation

Every variable reference (`x`, `y`, `counter`, etc.) in the AST must be evaluated with access to the current environment pointer.

- **Variable Expression Evaluation**

```
Value evalVariableExpr(const VariableExpr& expr, Environment::Ptr env) {
    return env->get(expr.name);
}
```

- **Variable Assignment Evaluation**

```
Value evalAssignmentExpr(const AssignmentExpr& expr, Environment::Ptr env) {
    Value val = evaluate(expr.value, env);
    env->assign(expr.name, val);
    return val;
}
```

11.2.7 Example: Block Scope Simulation

```
void simulateBlockScope() {
    auto global = std::make_shared<Environment>();

    global->declare("x", Value(1));  // global x = 1

    auto block1 = std::make_shared<Environment>(global);
    block1->declare("y", Value(2));  // block1 y = 2
```

```
    auto block2 = std::make_shared<Environment>(block1);
    block2->declare("x", Value(3));  // block2 x shadows global x

    std::cout << block2->get("x").get<IntType>() << "\n"; // prints 3
    std::cout << block2->get("y").get<IntType>() << "\n"; // prints 2
    std::cout << block2->get("z").get<IntType>() << "\n"; // error: undefined
}
```

11.2.8 Enhancing Performance: Optional Variable Resolution Caching

For future performance tuning:

- Add **identifier resolution caching** by tagging each identifier during parsing with a pointer to its resolved scope level.
- This avoids repeated traversal during evaluation.
- Especially beneficial in loops and large blocks.

11.2.9 Scoped Constants (Optional Extension)

In C, `const` variables cannot be reassigned. This can be supported by tracking a flag in the environment:

```
struct VariableBinding {
    Value value;
    bool isConst;
};

std::unordered_map<std::string, VariableBinding> symbols;
```

Then `assign()` must check `isConst` and reject modification if true.

11.2.10 Error Reporting and Diagnostics

When resolution fails:

- Report the missing identifier.
- Provide scope hierarchy (optional).
- Suggest closest identifiers (Levenshtein distance or simple prefix match).

```
throw std::runtime_error("Undefined variable: '" + name + "'.");
```

This encourages correct usage and assists debugging.

11.2.11 Conclusion

Variable resolution is a fundamental pillar of the interpreter's correctness. Using a **chained `Environment` structure** with C++20 idioms allows the interpreter to replicate the exact behavior of C's lexical scoping model, including shadowing, assignment propagation, and access to global declarations. This section formalizes the resolution logic that allows every variable reference in your interpreted language to behave like its C counterpart, ensuring clarity, consistency, and predictability in every program executed.

11.3 Lexical Scoping Implementation

11.3.0.1 Introduction

Lexical scoping, also called **static scoping**, is the foundation of most C-style programming languages. In this model, the scope of a variable is determined by its **position in the source code**, and not by the call stack or execution flow. Every block defines a new **lexical environment**, and variable resolution is performed by inspecting enclosing blocks at compile or runtime in a predictable and fixed manner. This section focuses on **how to implement lexical scoping** using modern **C++20/23 idioms**, including block environment chaining, controlled variable visibility, shadowing, and integration with expression evaluation.

11.3.1 What is Lexical Scoping?

In lexical scoping:

- The structure of the source code defines visibility.
- Every `{ ... }` block introduces a **new scope**.
- Variables declared in an outer block are visible in inner blocks.
- Inner variables can **shadow** outer ones, hiding them within their local scope.
- Lookup begins in the **current block** and proceeds outward until found or fails.

Example:

```
int x = 5;
{
    int x = 10;
    printf("%d", x); // prints 10
}
printf("%d", x);     // prints 5
```

11.3.2 Environment as Lexical Scope

Lexical scoping in an interpreter is naturally implemented by a **linked chain of environments**—each representing a scope.

```
class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;

    Environment(Ptr parent = nullptr) : parent(parent) {}

    void declare(const std::string& name, const Value& val);
    void assign(const std::string& name, const Value& val);
    Value get(const std::string& name) const;

private:
    std::unordered_map<std::string, Value> symbols;
    Ptr parent;
};
```

Each block creates a new `Environment` pointing to its enclosing scope. This forms a **lexical scope chain**, similar to how C compilers analyze symbol tables.

11.3.3 Entering and Exiting Lexical Scopes

Every time the interpreter enters a block (`{}`), it must create a new environment:

```
void evaluateBlock(const std::vector<Statement>& stmts, Environment::Ptr outerEnv) {
    auto innerEnv = std::make_shared<Environment>(outerEnv);
    for (const auto& stmt : stmts) {
        evaluate(stmt, innerEnv);
    }
```

```
}
```

When execution leaves the block, the `innerEnv` is discarded. Because it is a `shared_ptr`, no manual deallocation is needed.

11.3.4 Visualizing the Scope Chain

Consider this code:

```
int x = 1;
{
    int y = x + 1;
    {
        int x = y + 2;
    }
}
```

The environment chain at the deepest block looks like:

```
Env3: { x=4 }       --> innermost
Env2: { y=2 }
Env1: { x=1 }       --> global
```

Variable `x` in `Env3` shadows `x` in `Env1`, but `y` from `Env2` is still visible.

11.3.5 Scope Chain Traversal in `get` and `assign`

- **Variable Lookup (`get`):**

```
Value Environment::get(const std::string& name) const {
    if (symbols.contains(name)) return symbols.at(name);
```

```
    if (parent) return parent->get(name);
    throw std::runtime_error("Undefined variable: " + name);
}
```

- **Variable Assignment (`assign`):**

```
void Environment::assign(const std::string& name, const Value& val) {
    if (symbols.contains(name)) {
        symbols[name] = val;
    } else if (parent) {
        parent->assign(name, val);
    } else {
        throw std::runtime_error("Assignment to undeclared variable: " + name);
    }
}
```

11.3.6 Shadowing and Uniqueness

Lexical scoping permits **shadowing**, but not **duplicate declarations** in the same scope.

```
void Environment::declare(const std::string& name, const Value& val) {
    if (symbols.contains(name))
        throw std::runtime_error("Variable already declared in this scope: " + name);
    symbols[name] = val;
}
```

This matches C's behavior—multiple declarations with the same name are allowed **only in different scopes**.

11.3.7 Managing the Global Scope

The **global environment** is the outermost scope. All top-level declarations are stored here.

```
auto globalEnv = std::make_shared<Environment>();
```

You can inject built-in functions or constants here:

```
globalEnv->declare("PI", Value(3.14159265));
globalEnv->declare("print", BuiltinFunction(printHandler));
```

Global variables are always reachable unless shadowed in a block.

11.3.8 Integration with Control Structures

Each control structure that introduces a block (`if`, `while`, `for`, etc.) must create a scope:

- **Example: `if` block**

```
if (toBool(evaluate(cond, env))) {
    auto newEnv = std::make_shared<Environment>(env);
    evaluateBlock(thenBranch, newEnv);
} else {
    auto newEnv = std::make_shared<Environment>(env);
    evaluateBlock(elseBranch, newEnv);
}
```

11.3.9 Optional Optimizations

- **a) Scope Depth Tracking**

Use integer levels to track how deep a scope is (for debugging, performance tuning):

```
class Environment {
    int depth;
};
```

- **b) Static Resolution Caching**

  During parsing, tag each identifier with the exact environment depth it belongs to. At runtime, you access it directly. This is similar to how closures are optimized in other interpreters.

11.3.10 Future-Proofing

The same lexical environment system supports future constructs:

- **Closures**: Functions capturing outer scopes.
- **Modules**: Isolated global scopes per file.
- **Block expressions**: Scopes that return a value.
- **Static analysis**: Using the lexical chain to validate unused or unreachable variables.

11.3.11 Conclusion

Lexical scoping is at the heart of variable visibility and evaluation order in C-style languages. Using a chain of `Environment` objects with modern C++20/23 features like `shared_ptr`, `unordered_map`, and structured error handling, you can build a powerful

and extensible interpreter that exactly models how scopes behave in C. This design supports nested blocks, variable shadowing, and dynamic expression evaluation, while providing a clean path forward to advanced features like closures, modules, and type inference.

11.4 Milestone — Working C-Style Variable System

11.4.0.1 Overview

This milestone marks the **successful implementation of a fully functional C-style variable system** in your interpreter. With this system in place, your language now supports **block-level variable declaration**, **lexical scoping**, **variable shadowing**, **value assignment**, and **resolution behavior** that mirrors the semantics of C. In this section, you will review the structure, usage, test cases, and integration of the variable system using **modern C++20/23** capabilities, ensuring correctness and extensibility.

11.4.1 Goals of the Variable System Milestone

- Support for runtime variable declaration and lookup.
- Lexical (block-based) scope chain.
- Shadowing of outer variables in nested scopes.
- Accurate variable resolution and assignment propagation.
- Safe redeclaration prevention in the same block.
- Clear error reporting for undeclared or redefined variables.
- Seamless integration with value evaluation and control flow.

11.4.2 Architecture Summary

- **Components**

  1. **Value**: Represents runtime data (`int`, `float`, `bool`, `string`, `array`).
  2. **Environment**: Represents a single lexical scope.
  3. **Environment Chain**: Linked via `std::shared_ptr` to parent environments.
  4. **Variable System API**: `declare`, `assign`, `get`.

- **Value System Reference**

```
using ValueVariant = std::variant<int64_t, double, bool, std::string,
↪  std::vector<Value>>;

struct Value {
    ValueVariant data;
    // Constructor overloads and helper methods...
};
```

- **Environment System Reference**

```
class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;

    Environment(Ptr parent = nullptr) : parent(parent) {}

    void declare(const std::string& name, const Value& value);
    void assign(const std::string& name, const Value& value);
```

```
    Value get(const std::string& name) const;

private:
    std::unordered_map<std::string, Value> symbols;
    Ptr parent;
};
```

11.4.3 Key Functional Behavior

- **Declaration**
  - Ensures no redeclaration in the same scope.
  - Declares new variable only in current block.

```
void Environment::declare(const std::string& name, const Value& value) {
    if (symbols.contains(name))
        throw std::runtime_error("Variable already declared in this block: " +
        ↪  name);
    symbols[name] = value;
}
```

- **Assignment**
  - Updates nearest enclosing variable with matching name.
  - Throws on assignment to undeclared name.

```
void Environment::assign(const std::string& name, const Value& value) {
    if (symbols.contains(name))
        symbols[name] = value;
    else if (parent)
        parent->assign(name, value);
    else
        throw std::runtime_error("Assignment to undeclared variable: " + name);
}
```

- **Lookup**
  - Retrieves from current or outer scopes.
  - Reports error if name is not found.

```
Value Environment::get(const std::string& name) const {
    if (symbols.contains(name)) return symbols.at(name);
    if (parent) return parent->get(name);
    throw std::runtime_error("Undefined variable: " + name);
}
```

11.4.4 Evaluation Integration Example

Assume the AST provides variable expressions and assignments:

```
struct VariableExpr {
    std::string name;
};

struct AssignmentExpr {
```

```
    std::string name;
    ExprPtr value;
};
```

- **Variable Expression**

```
Value evaluate(const VariableExpr& expr, Environment::Ptr env) {
    return env->get(expr.name);
}
```

- **Assignment Expression**

```
Value evaluate(const AssignmentExpr& expr, Environment::Ptr env) {
    Value val = evaluate(*expr.value, env);
    env->assign(expr.name, val);
    return val;
}
```

11.4.5 Demonstration and Test Case

- **Source Program (in your language):**

```
int x = 5;
{
    int x = 10;
    print(x); // should output 10
}
print(x); // should output 5
```

- **Interpreter Evaluation**

```
auto global = std::make_shared<Environment>();
global->declare("x", Value(5));

auto block = std::make_shared<Environment>(global);
block->declare("x", Value(10));
std::cout << block->get("x").get<int64_t>() << "\n"; // 10

std::cout << global->get("x").get<int64_t>() << "\n"; // 5
```

11.4.6 Error Handling Examples

- **Duplicate Declaration**

```
env->declare("a", Value(1));
env->declare("a", Value(2)); // Throws error
```

- **Use of Undeclared Variable**

```
env->get("undefined"); // Throws error
```

- **Assignment Without Declaration**

```
env->assign("a", Value(10)); // Throws unless 'a' was declared
```

11.4.7 Design Conformance to C Semantics

Table 4-1: Scope and Variable Resolution Features

Feature	Behavior	Status
Block scoping	Introduced per `{}`	Implemented
Shadowing	Inner blocks can shadow outer	Implemented
Resolution order	Innermost to outermost	Implemented
Redeclaration in block	Disallowed	Implemented
Reassignment	Allowed after declaration	Implemented
Global environment	Root scope, persistent	Implemented

11.4.8 Modern C++ Enhancements

- **`std::shared_ptr<Environment>`**: Simplifies memory and scope management.
- **`std::unordered_map`**: Efficient name-value lookup.
- **C++20 `ranges`/`concepts` (future)**: Can enforce constraints on Value or identifiers.
- **Optional**: Use `std::expected<Value, std::string>` (C++23) instead of exceptions for assign/get.

11.4.9 Diagnostic and Debugging Tools

Optional logging and inspection utilities:

```
void debugScope(const Environment::Ptr& env, int level = 0) {
    for (const auto& [name, val] : env->symbols)
        std::cout << std::string(level * 2, ' ') << name << " = " << val.toString()
        ↪  << "\n";
    if (env->parent) debugScope(env->parent, level + 1);
}
```

11.4.10 Summary

This milestone finalizes the **C-style variable system** in your interpreter, enabling all standard scoping behaviors known in C and similar languages. By utilizing chained `Environment` instances, clean `Value` abstraction, and scoped operations like `declare`, `assign`, and `get`, you now possess a solid and extensible infrastructure for variable management. This system is the foundation for future chapters on functions, closures, modules, and advanced static analysis, making it a critical checkpoint in your interpreter's design.

Chapter 12

Expression Evaluation in C-Style

12.1 Binary and Unary Operations with C Precedence

12.1.0.1 Introduction

C-style languages rely on a well-defined **operator precedence and associativity model** to determine how expressions are grouped and evaluated. A correct interpreter must respect these rules in its **abstract syntax tree (AST) construction** and **runtime evaluation**. This section explores how to implement **unary and binary operations** with full support for **C's operator precedence hierarchy**, leveraging **modern C++20/23 features** such as `std::variant`, `std::visit`, and `constexpr`-based operator mapping.

12.1.1 Operator Categories in C

C-classifies operators into **unary** and **binary**, and assigns each a **precedence level** and **associativity** (left-to-right or right-to-left). You must mirror these rules in both:

1. **Parser:** To construct the correct AST shape.
2. **Evaluator:** To process the nodes in the correct order.

Simplified Operator Precedence Table (High to Low)

Table 1-1: C-style Operator Precedence and Associativity

Precedence	Operators	Type	Associativity
15	`()` `[]` `.`	postfix	left-to-right
14	`!` `-` `+` `~`	unary	right-to-left
13	`*` `/` `%`	binary	left-to-right
12	`+` `-`	binary	left-to-right
11	`<` `>` `<=` `>=`	binary	left-to-right
10	`==` `!=`	binary	left-to-right
9	`&&`	binary	left-to-right
8	`||`	binary	left-to-right
3	`=` `+=` `-=`	binary	right-to-left

12.1.2 Expression Representation in AST

Expressions are modeled as trees, with each node representing an operator and its operand(s).

AST Node Types

```
struct Expr;
using ExprPtr = std::unique_ptr<Expr>;

struct BinaryExpr {
    std::string op;
    ExprPtr left;
    ExprPtr right;
};

struct UnaryExpr {
    std::string op;
    ExprPtr operand;
};

struct LiteralExpr {
    Value value;
};

struct VariableExpr {
    std::string name;
};
```

The parser must use **precedence climbing** or **recursive descent with precedence levels** to build these nodes according to C rules.

12.1.3 Evaluating Binary Expressions

Binary evaluation follows type-aware dispatch and promotion (already implemented in the Value System). Here's how the evaluator might look:

```
Value evaluate(const BinaryExpr& expr, Environment::Ptr env) {
    Value left = evaluate(*expr.left, env);
    Value right = evaluate(*expr.right, env);

    if (expr.op == "+")      return applyAdd(left, right);
    else if (expr.op == "-") return applySub(left, right);
    else if (expr.op == "*") return applyMul(left, right);
    else if (expr.op == "/") return applyDiv(left, right);
    else if (expr.op == "==")return applyEqual(left, right);
    else if (expr.op == "&&")return Value(toBool(left) && toBool(right));
    else if (expr.op == "||")return Value(toBool(left) || toBool(right));

    throw std::runtime_error("Unknown binary operator: " + expr.op);
}
```

Each `apply*` function internally handles:

- Type promotion (`int → float`)
- Type dispatch using `std::visit`
- Error on invalid type combinations

12.1.4 Evaluating Unary Expressions

Unary operators are parsed with **right-to-left** associativity and high precedence. Evaluation is straightforward:

```
Value evaluate(const UnaryExpr& expr, Environment::Ptr env) {
    Value operand = evaluate(*expr.operand, env);

    if (expr.op == "-") {
```

```
        return std::visit([](auto&& val) -> Value {
            using T = std::decay_t<decltype(val)>;
            if constexpr (std::is_same_v<T, int64_t> || std::is_same_v<T, double>)
                return -val;
            else
                throw std::runtime_error("Invalid unary '-' operand");
        }, operand.data);
    }
    else if (expr.op == "!") {
        return Value(!toBool(operand));
    }

    throw std::runtime_error("Unknown unary operator: " + expr.op);
}
```

12.1.5 Parser: Operator Precedence Parsing (Shunting Yard or Precedence Climbing)

During parsing, your grammar should handle binary operator precedence using either:

- **Pratt Parsing / Precedence Climbing**
- **Shunting Yard Algorithm**

Example: Precedence Climbing

```
ExprPtr parseExpression(int minPrecedence) {
    ExprPtr left = parsePrimary();

    while (hasNextOperator() && getPrecedence(currentOp()) >= minPrecedence) {
```

```
        std::string op = currentOp();
        int precedence = getPrecedence(op);
        bool rightAssoc = isRightAssociative(op);
        int nextMinPrecedence = rightAssoc ? precedence : precedence + 1;

        advance(); // consume operator
        ExprPtr right = parseExpression(nextMinPrecedence);

        left = std::make_unique<BinaryExpr>(op, std::move(left), std::move(right));
    }

    return left;
}
```

12.1.6 Operator Table and Precedence Metadata

Use a static map or constexpr table to define operator precedence and associativity.

```
struct OperatorInfo {
    int precedence;
    bool rightAssociative;
};

inline const std::unordered_map<std::string, OperatorInfo> operatorTable = {
    {"+",  {12, false}},
    {"-",  {12, false}},
    {"*",  {13, false}},
    {"/",  {13, false}},
    {"%",  {13, false}},
    {"==", {10, false}},
    {"!=", {10, false}},
```

```
    {"&&", {9, false}},
    {"||", {8, false}},
    {"=",  {3, true}},
};
```

Use this in your parser to drive precedence decisions.

12.1.7 Short-Circuit Evaluation

Operators like `&&` and `||` require **short-circuit semantics**. That means only evaluating the right-hand side **if needed**.

Example:

```
Value evaluate(const BinaryExpr& expr, Environment::Ptr env) {
    if (expr.op == "&&") {
        Value left = evaluate(*expr.left, env);
        if (!toBool(left)) return Value(false);
        Value right = evaluate(*expr.right, env);
        return Value(toBool(right));
    }
    if (expr.op == "||") {
        Value left = evaluate(*expr.left, env);
        if (toBool(left)) return Value(true);
        Value right = evaluate(*expr.right, env);
        return Value(toBool(right));
    }
    // Other cases...
}
```

12.1.8 C++20/23-Specific Enhancements

- **a) `if constexpr` in Operation Dispatch**

  Use in `apply*` functions to statically resolve type combinations in `std::visit`.

- **b) `consteval` or `constexpr` for Constant Folding**

  Enable compile-time expression reduction during parsing (optional optimization).

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

- **c) Concepts (optional)**

  Enforce constraints on operand types if template-based dispatch is used:

```
template<typename T>
concept Numeric = std::is_same_v<T, int64_t> || std::is_same_v<T, double>;
```

12.1.9 Validation and Diagnostics

Ensure:

- Division by zero is detected.
- Invalid operand combinations are reported with full source info.
- Associativity and precedence are verified by test cases like:

```
int result = 1 + 2 * 3;  // should evaluate as 1 + (2 * 3) = 7
int a = 5;
int b = 0;
int c = a  b++; // b++ should not execute
```

12.1.10 Conclusion

Binary and unary operator evaluation forms the **semantic core of expression handling** in a C-style interpreter. By respecting the **precedence and associativity rules** defined in C, and implementing them both in the parser and evaluator using **modern C++20/23 techniques**, this section ensures that your language expressions behave intuitively and predictably. With this system complete, your interpreter can now execute complex mathematical and logical expressions with full fidelity to C's expression semantics.

12.2 Assignment Operations and Side Effects

12.2.0.1 Introduction

Assignment expressions in C-style languages are **evaluated for their side effects** and often used as full expressions in larger compound statements. This includes simple assignments (`=`), compound assignments (`+=`, `-=`, `*=`, etc.), and postfix increment/decrement (`x++`, `x--`). Properly supporting these requires your interpreter to handle **side effects**, **l-value resolution**, **evaluation order**, and **value propagation** in compliance with C semantics.
This section explains how to implement **assignment operations** and manage their effects on the runtime environment using modern C++20/23 practices.

12.2.1 Assignment as an Expression in C

In C, assignment is not just a statement but also an **expression** with a resulting value:

```
int x;
x = 5;        // evaluates to 5
```

```
int y = (x = 10) + 2;  // y becomes 12
```

This behavior must be preserved in your interpreter, ensuring:

- The left-hand side (LHS) is an **l-value** (i.e., a resolvable variable reference).
- The right-hand side (RHS) is **evaluated first**.
- The result of the entire expression is the value **assigned**, not the LHS or a boolean.

12.2.2 AST Representation of Assignments

To enable generality and side-effect tracking, assignment is represented in your AST as a distinct expression type.

```
struct AssignmentExpr {
    std::string name;     // variable to assign
    std::string op;       // =, +=, -=, etc.
    ExprPtr value;        // expression to evaluate and assign
};
```

This supports both simple and compound assignments by storing the operator as a string.

12.2.3 Evaluation Strategy

When evaluating an assignment expression:

1. Evaluate the RHS expression.

2. If the operation is a **compound assignment** (`+=`, `*=`, etc.), fetch the current value of the LHS and apply the binary operation.

3. Assign the result to the LHS in the current environment.

4. Return the assigned value.

Evaluation Logic

```
Value evaluate(const AssignmentExpr& expr, Environment::Ptr env) {
    Value rhs = evaluate(*expr.value, env);

    if (expr.op == "=") {
        env->assign(expr.name, rhs);
        return rhs;
    }

    Value lhs = env->get(expr.name);

    if (expr.op == "+=") {
        Value result = applyAdd(lhs, rhs);
        env->assign(expr.name, result);
        return result;
    } else if (expr.op == "-=") {
        Value result = applySub(lhs, rhs);
        env->assign(expr.name, result);
        return result;
    }
    // Repeat for *=, /=, %=, etc.

    throw std::runtime_error("Unknown assignment operator: " + expr.op);
}
```

Use the `apply*` functions from the value system that support proper type promotion and operation dispatch.

12.2.4 Side Effects and Evaluation Order

In C, **side effects must be sequenced correctly**, especially when assignments are nested or mixed with increments:

```
int y = (x += 3) * 2;  // x becomes 4, y becomes 8
```

To replicate this:

- Always evaluate RHS **before** any change to the LHS.
- Capture and return the **new value** as the result of the expression.

Assignment expressions must never introduce **undefined behavior**, so protect against:

- Reassigning undeclared variables.
- Assigning to constants (if your language includes them).
- Type mismatches between LHS and RHS.

12.2.5 Variable Tracking for Side Effects

Your interpreter's environment (symbol table) already tracks variable values. For side effects, ensure that:

- The `assign()` function updates the **existing symbol** in the correct enclosing scope.
- Assignments propagate **only if the variable is found**, otherwise throw an error.

```
void Environment::assign(const std::string& name, const Value& value) {
    if (symbols.contains(name)) {
        symbols[name] = value;
    } else if (parent) {
        parent->assign(name, value);
    } else {
        throw std::runtime_error("Assignment to undeclared variable: " + name);
    }
}
```

12.2.6 Compound Assignment as Syntax Sugar

Each compound assignment is syntactic sugar for a binary operation + assignment:

```
x += 5;   // becomes: x = x + 5;
```

Handle this transformation during parsing or directly during evaluation.

To simplify, map operators:

```
const std::unordered_map<std::string, std::function<Value(const Value&, const
↪  Value&)>> compoundOps = {
    {"+=", applyAdd},
    {"-=", applySub},
    {"*=", applyMul},
    {"/=", applyDiv},
    {"%=", applyMod}
};
```

Then:

```
if (auto it = compoundOps.find(expr.op); it != compoundOps.end()) {
    Value lhs = env->get(expr.name);
    Value result = it->second(lhs, rhs);
    env->assign(expr.name, result);
    return result;
}
```

12.2.7 Postfix and Prefix Increment/Decrement (Optional)

C supports `x++`, `++x`, `x--`, `--x`. These are **expressions with side effects**, and behave differently in value-returning context.

```
// x++: return current value, then increment
// ++x: increment, then return new value
```

- **AST Representation**

```
struct UnaryUpdateExpr {
    std::string name;
    std::string op; // "++", "--"
    bool prefix;
};
```

- **Evaluation Logic**

```
Value evaluate(const UnaryUpdateExpr& expr, Environment::Ptr env) {
    Value current = env->get(expr.name);

    if (!current.is<int64_t>())
```

```
        throw std::runtime_error("Increment/decrement only allowed on
        ↪  integers");

    int64_t value = current.get<int64_t>();

    if (expr.op == "++") {
        if (expr.prefix) {
            env->assign(expr.name, Value(value + 1));
            return Value(value + 1);
        } else {
            env->assign(expr.name, Value(value + 1));
            return Value(value);
        }
    } else if (expr.op == "--") {
        if (expr.prefix) {
            env->assign(expr.name, Value(value - 1));
            return Value(value - 1);
        } else {
            env->assign(expr.name, Value(value - 1));
            return Value(value);
        }
    }

    throw std::runtime_error("Unknown unary update operator");
}
```

12.2.8 Constant Assignments and Immutability (Optional)

For future extension, add support for `const` declarations and prevent reassignment:

```
struct VariableBinding {
    Value value;
    bool isConst;
};

std::unordered_map<std::string, VariableBinding> symbols;

void assign(...) {
    if (symbols[name].isConst)
        throw std::runtime_error("Cannot reassign to const variable");
}
```

This supports safer programming practices and enables static analysis.

12.2.9 Expression Sequencing and Return Value

Ensure assignment expressions **return the final assigned value**, not the variable name or RHS:

```
int x;
int y = (x = 3) + 4; // x becomes 3, y becomes 7
```

The assignment expression evaluates to `3`, which becomes part of a larger expression. Your AST and evaluator must support nesting and sequencing of assignments.

12.2.10 Test Case Examples

- **Simple Assignment**

  ```
  int x = 10;
  x = x + 5;    // x = 15
  ```

- **Compound Assignment**

```
int a = 3;
a *= 4 + 1;   // a = a * (4 + 1) = 15
```

- **Nested Assignment**

```
int x;
int y;
x = y = 20;   // x and y both 20
```

- **Side Effect Order**

```
int x = 1;
int y = (x += 3) * 2; // x = 4, y = 8
```

12.2.11 Conclusion

This section completes your interpreter's support for **assignment semantics and side effects**, a cornerstone of C-style evaluation. By treating assignments as expressions, preserving correct order, supporting compound operations, and managing scope-aware updates in your environment, your interpreter now mirrors the behavior of C in how it modifies program state. Using modern C++20/23 features, this system is designed for safety, extensibility, and fidelity to C's operational logic — forming a robust base for function return values, loop control variables, and expression-based evaluation in all forms.

12.3 Variable Lookup Following C Scoping Rules

12.3.0.1 Introduction

In C and other C-style languages, **variable lookup** follows a strict **lexical scoping** model. A variable name refers to the closest enclosing declaration visible at the point of use, and this rule must be strictly preserved at both compile-time (for static analyzers or compilers) and runtime (for interpreters).
In an interpreter written in modern C++20/23, the goal is to model C's **block-structured resolution strategy**, implement variable visibility through **environment chaining**, and ensure **deterministic and efficient lookup** of variable values.
This section focuses on the design and implementation of variable resolution in the expression evaluator using up-to-date C++ practices.

12.3.1 C Variable Lookup Semantics

In C:

- **Scopes** are introduced by `{}` blocks.
- A **name is resolved** to the nearest declaration visible from the point of use.
- Inner scopes **can shadow** variables from outer scopes.
- **Global scope** is visible unless shadowed.
- Variables must be **declared before use** in the same block (though not enforced strictly in all C compilers, your language can).

Example:

```
int x = 5;
{
    int x = 10;   // shadows outer x
    printf("%d", x); // prints 10
}
printf("%d", x); // prints 5
```

Your interpreter must resolve `x` correctly based on where it is **lexically defined**, not based on execution flow.

12.3.2 Environment Chain Overview

To support lexical lookup, each **execution context** (block, function, etc.) is associated with an `Environment` object. Each environment stores variable bindings and points to its **enclosing (parent) environment**.

```
class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;

    Environment(Ptr parent = nullptr)
        : parent(std::move(parent)) {}

    void declare(const std::string& name, const Value& value);
    void assign(const std::string& name, const Value& value);
    Value get(const std::string& name) const;

private:
    std::unordered_map<std::string, Value> symbols;
    Ptr parent;
};
```

- `symbols`: Holds local variables.
- `parent`: Points to the outer scope (lexically enclosing block).

This forms a **scope chain** similar to symbol tables in compilers.

12.3.3 Lookup Algorithm: C Rules in Practice

To resolve a variable:

1. Search in the **current scope**.
2. If not found, search in the **immediate parent**.
3. Continue until the variable is found or the **global scope** is reached.
4. If the name does not exist, throw a resolution error.

Implementation

```
Value Environment::get(const std::string& name) const {
    if (symbols.contains(name)) {
        return symbols.at(name);
    } else if (parent) {
        return parent->get(name);
    } else {
        throw std::runtime_error("Undefined variable: " + name);
    }
}
```

This implementation is:

- **Deterministic** (mirrors lexical structure).

- **Safe** (throws error on undefined variable).
- **Extensible** (can support closures, modules, etc.).

12.3.4 Expression Evaluation Integration

Every **variable expression** node must trigger lookup via the current environment:

- **AST Node**

```
struct VariableExpr {
    std::string name;
};
```

- **Evaluation Logic**

```
Value evaluate(const VariableExpr& expr, Environment::Ptr env) {
    return env->get(expr.name);
}
```

This ensures expressions like `x + 1` evaluate `x` using the nearest visible binding.

12.3.5 Handling Shadowing

C allows variables in inner scopes to **shadow outer ones**.

```
int x = 1;
{
    int x = 2;  // shadows outer x
    printf("%d", x); // prints 2
}
```

This behavior is automatically respected due to `Environment::declare()` placing the variable only in the **current** environment:

```
void Environment::declare(const std::string& name, const Value& value) {
    if (symbols.contains(name))
        throw std::runtime_error("Variable already declared in this block: " + name);
    symbols[name] = value;
}
```

Thus:

- Inner `x` hides outer `x`.
- Access to `x` in the inner block resolves to the one in `symbols`.
- The `get()` logic never reaches the outer scope when a shadow exists.

12.3.6 Example Execution Chain

- **Source Program:**

```
int x = 3;
{
    int y = x + 2;   // uses outer x
    {
        int x = 10;  // shadows outer x
        print(x);    // prints 10
        print(y);    // prints 5
    }
}
```

- **Scope Chain Illustration at `print(x);`:**

```
Env3: { x = 10 }
Env2: { y = 5 }
Env1: { x = 3 }
```

Call to `get("x")` starts from `Env3`, finds `x`, and returns `10`.

Call to `get("y")` searches `Env3` → not found
Then `Env2` → found → returns `5`.

12.3.7 Lookup Failure Handling

When a variable is not declared anywhere in the scope chain:

```
Value v = env->get("not_defined"); // throws runtime_error
```

This mimics C's behavior (e.g., undefined identifier in source results in compiler error).
You can enhance this with better diagnostics:

```
throw std::runtime_error("Undefined variable: '" + name + "'. Did you mean ...?");
```

Optional: Use Levenshtein distance to suggest possible matches.

12.3.8 Optional Optimization: Static Resolution Hints

For performance, you can **tag variable expressions** during parsing with their **resolved scope depth** or pointer to the defining environment.
At evaluation time, instead of recursive lookup, you jump directly to the scope:

```
struct ResolvedVariableExpr {
    std::string name;
```

```
    Environment::Ptr resolvedEnv;
};
```

This mimics **compiler symbol resolution**, increasing performance at the cost of dynamic flexibility.

12.3.9 Modern C++ Enhancements

- **Use of `std::shared_ptr`**

  ```
  Environment::Ptr parent = std::make_shared<Environment>();
  ```

  - Manages scope lifetime automatically.
  - Avoids manual cleanup or memory leaks.
  - Can be replaced with `std::unique_ptr` if scopes are strictly nested and not shared.

- **Use of `std::unordered_map`**

  - Provides fast O(1) average lookup for variable names.
  - Can be customized with string interning for faster performance.

12.3.10 Summary of Lookup Behavior

Table 3-2: Scope Features and Implementation Status

Feature	Implementation Status
Lexical scope chaining	via `Environment::parent`
Block-level variable visibility	`symbols` are local per scope
Shadowing detection	`declare()` enforces per-block uniqueness
Recursive lookup	`get()` searches parent chain
Error on undefined name	throws with message
Optional optimization (static)	can tag resolved scopes at parse time

12.3.11 Conclusion

Variable lookup is a central part of C-style expression evaluation. In your interpreter, it is best achieved through **lexical scope chaining**, **shadowing-respecting search**, and **block-based symbol maps**. With modern C++20/23 idioms like `shared_ptr`, `unordered_map`, and error-safe logic, your variable resolution system is now faithful to C's behavior, efficient, and ready to scale to support closures, modules, and static checking in future chapters.

12.4 Hands-on — Calculator Supporting C-Style Expressions

12.4.0.1 Introduction

As a practical culmination of this chapter, we implement a **mini calculator interpreter** that evaluates C-style expressions. This includes:

- Arithmetic expressions with operator precedence.
- Unary and binary operations.
- Parentheses for grouping.
- Variable assignments.
- Variable usage with scoping.
- Full expression evaluation respecting C semantics.

This section demonstrates how to design and build this interactive calculator using modern **C++20/23**, encapsulating all evaluation logic, scoping, and expression parsing from earlier sections.

12.4.1 Key Features

- **Arithmetic operators**: `+`, `-`, `*`, `/`, `%`
- **Unary operators**: `-`, `+`, `!`
- **Parentheses**
- **Variables**: `x = 3 + 2`
- **Chained assignments**: `a = b = 5`
- **Error handling**
- **Evaluation with correct precedence**

12.4.2 Value System

Value Type

```
using IntType = int64_t;
using FloatType = double;
using BoolType = bool;
using StringType = std::string;

using ValueVariant = std::variant<IntType, FloatType, BoolType, StringType>;

struct Value {
    ValueVariant data;

    template<typename T>
    bool is() const { return std::holds_alternative<T>(data); }

    template<typename T>
    T& get() { return std::get<T>(data); }

    std::string toString() const {
        return std::visit([](auto&& val) {
            std::ostringstream oss;
            oss << val;
            return oss.str();
        }, data);
    }
};
```

12.4.3 Environment for Variable Support

```
class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;

    Environment(Ptr parent = nullptr) : parent(std::move(parent)) {}

    void declare(const std::string& name, const Value& val) {
        symbols[name] = val;
    }

    void assign(const std::string& name, const Value& val) {
        if (symbols.contains(name))
            symbols[name] = val;
        else if (parent)
            parent->assign(name, val);
        else
            throw std::runtime_error("Undeclared variable: " + name);
    }

    Value get(const std::string& name) const {
        if (symbols.contains(name))
            return symbols.at(name);
        if (parent)
            return parent->get(name);
        throw std::runtime_error("Undefined variable: " + name);
    }

private:
    std::unordered_map<std::string, Value> symbols;
    Ptr parent;
```

```
};
```

12.4.4 Expression AST Design

```
struct Expr;
using ExprPtr = std::unique_ptr<Expr>;

struct Expr {
    virtual Value eval(Environment::Ptr env) const = 0;
    virtual ~Expr() = default;
};

struct LiteralExpr : Expr {
    Value value;
    LiteralExpr(Value v) : value(std::move(v)) {}
    Value eval(Environment::Ptr) const override { return value; }
};

struct VariableExpr : Expr {
    std::string name;
    VariableExpr(std::string n) : name(std::move(n)) {}
    Value eval(Environment::Ptr env) const override {
        return env->get(name);
    }
};

struct UnaryExpr : Expr {
    std::string op;
    ExprPtr operand;
    UnaryExpr(std::string o, ExprPtr e) : op(std::move(o)), operand(std::move(e)) {}
    Value eval(Environment::Ptr env) const override {
```

```
        Value v = operand->eval(env);
        if (op == "-") return Value(-v.get<IntType>());
        if (op == "!") return Value(!static_cast<bool>(v.get<IntType>()));
        return v;
    }
};

struct BinaryExpr : Expr {
    std::string op;
    ExprPtr left, right;

    BinaryExpr(std::string o, ExprPtr l, ExprPtr r)
        : op(std::move(o)), left(std::move(l)), right(std::move(r)) {}

    Value eval(Environment::Ptr env) const override {
        Value l = left->eval(env);
        Value r = right->eval(env);

        if (op == "+") return Value(l.get<IntType>() + r.get<IntType>());
        if (op == "-") return Value(l.get<IntType>() - r.get<IntType>());
        if (op == "*") return Value(l.get<IntType>() * r.get<IntType>());
        if (op == "/") return Value(l.get<IntType>() / r.get<IntType>());
        if (op == "%") return Value(l.get<IntType>() % r.get<IntType>());
        if (op == "==") return Value(l.get<IntType>() == r.get<IntType>());
        if (op == "!=") return Value(l.get<IntType>() != r.get<IntType>());
        if (op == "<") return Value(l.get<IntType>() < r.get<IntType>());
        if (op == "<=") return Value(l.get<IntType>() <= r.get<IntType>());
        if (op == ">") return Value(l.get<IntType>() > r.get<IntType>());
        if (op == ">=") return Value(l.get<IntType>() >= r.get<IntType>());
        if (op == "&&") return Value(static_cast<bool>(l.get<IntType>()) &&
        ↪  static_cast<bool>(r.get<IntType>()));
        if (op == "||") return Value(static_cast<bool>(l.get<IntType>()) ||
        ↪  static_cast<bool>(r.get<IntType>()));
```

```
        throw std::runtime_error("Unknown binary operator: " + op);
    }
};

struct AssignmentExpr : Expr {
    std::string name;
    ExprPtr value;
    AssignmentExpr(std::string n, ExprPtr v) : name(std::move(n)),
    ↪  value(std::move(v)) {}
    Value eval(Environment::Ptr env) const override {
        Value val = value->eval(env);
        env->assign(name, val);
        return val;
    }
};
```

12.4.5 Recursive Descent Parser with Precedence

Simplified parser using recursive descent to respect operator precedence:

```
ExprPtr parseExpression();       // Handles lowest precedence
ExprPtr parseTerm();             // +, -
ExprPtr parseFactor();           // *, /
ExprPtr parsePrimary();          // literals, variables, ()
```

Each function reduces a tighter subset of operations. Operators are left-associative by default (like C).

12.4.6 Main Loop (REPL-like)

```
int main() {
    Environment::Ptr globalEnv = std::make_shared<Environment>();

    std::string line;
    while (std::getline(std::cin, line)) {
        try {
            ExprPtr expr = parse(line); // Your parser implementation
            Value result = expr->eval(globalEnv);
            std::cout << "=> " << result.toString() << "\n";
        } catch (const std::exception& ex) {
            std::cerr << "Error: " << ex.what() << "\n";
        }
    }
}
```

12.4.7 Example Interactions

- **Input**

```
x = 5 + 2
y = x * 10
y + 3
```

- **Output**

```
=> 7
=> 70
=> 73
```

12.4.8 Future Extensions

- Add `float`, `bool`, and `string` types.
- Support for `if`, `while`, `for` expressions.
- Function calls and user-defined functions.
- Pre/post-increment operators (`x++`, `--x`).
- Expression folding and constant evaluation (`consteval`).

12.4.9 Conclusion

This hands-on calculator validates the interpreter's capability to evaluate C-style expressions with full precedence, assignment, and variable handling. It combines all major components of your evaluation engine—lexical scoping, value system, expression hierarchy, and runtime environment—into a practical tool. It also provides a solid testbed for extending the language toward full scripting or systems language capabilities using modern C++20/23.

Chapter 13

Enhanced REPL – Version 2

13.1 Expression Evaluation in Interactive Mode

13.1.0.1 Introduction

A robust interpreter requires a **responsive and reliable REPL** (Read-Eval-Print Loop). The REPL is not just a convenience—it's a dynamic testing ground for your parser, evaluation engine, and variable system. In Version 2 of your REPL, the focus is on enabling **real-time expression evaluation** with complete support for C-style semantics: operator precedence, variable assignments, side effects, lexical scoping, and immediate feedback.

This section details the implementation of interactive **expression evaluation**, from input parsing to value resolution and result display, all using **modern C++20/23** principles.

13.1.1 Goals of Interactive Expression Evaluation

- Parse and evaluate expressions line-by-line.

- Support variable declaration, use, and reassignment.
- Preserve state across expressions (global environment).
- Respect C-style operator precedence and scoping.
- Show evaluation results immediately.
- Handle invalid inputs gracefully with diagnostic feedback.

13.1.2 Essential Architecture Components

1. **Environment**

   A persistent global environment holds user-defined variables.

   ```
   Environment::Ptr globalEnv = std::make_shared<Environment>();
   ```

   This environment is passed into every evaluation call, ensuring that variables retain their values across REPL inputs.

2. **Parser**

   A recursive descent parser or Pratt parser processes the user input and builds an abstract syntax tree (AST) representing expressions.

3. **Evaluator**

   Each AST node implements `eval(Environment::Ptr env)`, returning a `Value` object.

13.1.3 Read-Eval-Print Loop Implementation

```
void runREPL(Environment::Ptr globalEnv) {
    std::string line;
    while (true) {
        std::cout << ">>> ";
        if (!std::getline(std::cin, line)) break;

        try {
            ExprPtr expr = parseExpression(line);
            Value result = expr->eval(globalEnv);
            std::cout << "=> " << result.toString() << "\n";
        } catch (const std::exception& e) {
            std::cerr << "Error: " << e.what() << "\n";
        }
    }
}
```

- **Input** is read from standard input.
- **Parsing** converts the line into an AST expression.
- **Evaluation** occurs with access to the shared global environment.
- **Results** are printed using `Value::toString()`.

13.1.4 Supported Expression Types

1. **Literals**

```
42
3.14
true
```

2. **Arithmetic**

```
1 + 2 * 3   // 7
(4 + 2) / 3 // 2
```

3. **Unary Operators**

```
-5
!0
```

4. **Variables**

```
x = 10
y = x * 2
```

5. **Compound Assignments**

```
x += 5
x *= 2
```

6. **Comparisons and Logic**

```
x > 5 && x < 15
x == 10
```

13.1.5 Persistent Evaluation Context

Each evaluation modifies the environment:

```
Value AssignmentExpr::eval(Environment::Ptr env) const {
    Value rhs = value->eval(env);
    env->assign(name, rhs);
    return rhs;
}
```

Subsequent expressions reflect changes:

```
x = 3
x + 1 // returns 4
```

13.1.6 Error Detection and Recovery

Errors are caught at the top-level REPL loop. Example error cases:

- Use of undefined variable:

```
>>> z + 1
Error: Undefined variable: z
```

- Division by zero:

```
>>> 10 / 0
Error: Division by zero
```

These exceptions are surfaced using runtime checks in the evaluation phase.

13.1.7 C++20/23 Features in Use

- **std::variant** and `std::visit`

  Used in the `Value` system to represent and evaluate runtime types.

- **`std::shared_ptr`**

  Used to manage scoped environments and AST nodes with safe automatic memory control.

- **`if constexpr`**

  Used inside visitor dispatches for type-safe evaluation logic.

- **`constexpr` operator precedence table**

  Used during parsing to manage correct construction of binary expressions.

- **`std::string_view` (optional)**

  Can be used in the tokenizer to reduce string copying overhead.

13.1.8 Example REPL Session

```
>>> x = 10
=> 10
>>> y = x * 2
```

```
=> 20
>>> y + 5
=> 25
>>> x == 10
=> true
>>> x = x + 1
=> 11
>>> x
=> 11
```

This reflects full dynamic interaction with stateful variable management and expression chaining.

13.1.9 Test Cases for Validation

Each of the following should pass:

- `a = 1 + 2 * 3` → `a = 7`
- `b = a + (4 * 2)` → `b = 15`
- `b -= 5` → `b = 10`
- `b == 10` → `true`
- `c` → error: undefined variable

13.1.10 Future Extensions

- Allow declarations with types (e.g., `int x = 5`)
- Include functions and scoping inside blocks

- Add control flow expressions (e.g., `if`, `while`)
- Add multi-line input (REPL buffer and parser state)
- Add history and command editing

13.1.11 Conclusion

This section delivers a fully working REPL that supports **expression evaluation in C-style syntax**, enabling users to interactively test expressions, define and use variables, and see real-time results. The interpreter now supports a stateful, user-friendly interactive mode, forming a crucial milestone in language usability. With modern C++20/23 features, the architecture is clean, modular, and scalable — ready for further enhancements in block execution, type declarations, and function calls.

13.2 Variable Persistence with C-Style Scoping

13.2.0.1 Introduction

A reliable Read-Eval-Print Loop (REPL) in any C-style language must support **persistent variable storage** across expressions and sessions, while preserving **C-style block scoping rules**. In Version 2 of the REPL, users should be able to define variables, reuse them later, and create **nested scopes** where outer variables can be shadowed without being lost. This section presents how to implement persistent and block-aware variable resolution using modern **C++20/23** constructs.

13.2.1 C-Style Scoping Rules Recap

C-style scoping follows these principles:

- A variable declared in a block `{ ... }` is visible only within that block and its nested scopes.
- Inner scopes may **shadow** outer variables with the same name.
- Variables declared at the global level are accessible unless shadowed.
- Once a block ends, its variables **cease to exist** and are not accessible outside.

In REPL mode, users simulate these scopes through statements and expressions. Your interpreter must ensure:

- Variable **values persist** in the global scope.
- Block scopes are **created and destroyed dynamically** during evaluation.
- Variable **lookups always follow lexical (not dynamic) scope resolution**.

13.2.2 Persistent Global Environment

The global environment is the **root symbol table** where all top-level variables reside. It must live **as long as the REPL session runs**, and it must be **shared with all evaluated expressions**.

```
Environment::Ptr globalEnv = std::make_shared<Environment>();
```

Every evaluation in the REPL passes this `globalEnv` to the evaluator. It holds:

- All user-defined variables (`int x = 10`)
- Their most recent values
- Type-safe bindings in a `std::unordered_map`

```
std::unordered_map<std::string, Value> symbols;
```

13.2.3 Environment Chaining for Block Scoping

To support **nested block scoping** in REPL expressions (e.g., during `if`, `while`, `{ ... }`), each block or context evaluation must **create a new environment** that points to the parent:

```
auto innerEnv = std::make_shared<Environment>(outerEnv);
```

When the block ends, `innerEnv` is destroyed. Lookups always go from **inner to outer**:

```
Value Environment::get(const std::string& name) const {
    if (symbols.contains(name)) return symbols.at(name);
    if (parent) return parent->get(name);
    throw std::runtime_error("Undefined variable: " + name);
}
```

This maintains **lexical visibility** while allowing **temporary variables** to exist only in certain scopes.

13.2.4 Variable Assignment and Update

Assignments must follow C's behavior: update the **nearest** declared variable. If none is found, throw an error (or implicitly declare in global if the language permits).

```
void Environment::assign(const std::string& name, const Value& value) {
    if (symbols.contains(name)) {
        symbols[name] = value;
    } else if (parent) {
```

```
        parent->assign(name, value);
    } else {
        throw std::runtime_error("Assignment to undeclared variable: " + name);
    }
}
```

This supports persistent updates like:

```
x = 5
x = x + 1  // persists across REPL inputs
```

13.2.5 Shadowing and Restoration

Users may define temporary variables that shadow outer variables. After leaving the block, the outer variable is visible again.

Example

```
x = 10
{
    int x = 20;  // shadows global x
    x = x + 5;   // x becomes 25 inside block
}
x              // should still be 10
```

This is achieved by:

- Creating a new `Environment` for the block
- Evaluating statements inside the block with the new environment
- Discarding it after block completion

```
void evaluateBlock(const std::vector<Statement>& stmts, Environment::Ptr outerEnv) {
    auto localEnv = std::make_shared<Environment>(outerEnv);
    for (auto& stmt : stmts) {
        evaluate(stmt, localEnv);
    }
}
```

13.2.6 REPL Use Case

When a user types:

```
>>> a = 5
>>> b = a + 2
>>> {
>>>     int a = 100
>>>     b = a + b
>>> }
>>> a
>>> b
```

The output must be:

```
=> 5
=> 7
=>
=> 5
=> 107
```

Explanation:

- `a = 5` in global

- `b = 7`
- Inside block: shadowed `a = 100`; `b` updated to $100 + 7 = 107$
- Global `a` remains `5`
- Updated `b` visible globally

13.2.7 Value Consistency and Type Safety

To preserve variable integrity:

- Ensure consistent types for updates (optional stricter mode).
- Consider using a `VariableBinding` structure for `const`, `type`, or `lifetime` support.

```
struct VariableBinding {
    Value value;
    bool isConst;
    // optional: std::string typeName;
};
```

Modify assignment logic to respect immutability and type constraints.

13.2.8 Error Reporting and Shadow Awareness

When shadowing occurs, your interpreter can provide helpful feedback in REPL:

```
>>> x = 10
>>> {
>>>     int x = 20
```

```
>>>     x
>>> }
```

Optional message:

```
[info] Variable 'x' shadows one from outer scope
```

This is helpful for debugging but not required for correctness.

13.2.9 C++20/23 Features Used

- `std::shared_ptr` for scope chaining and lifetime management.
- `std::unordered_map` for fast symbol resolution.
- `std::variant` for `Value` abstraction.
- `if constexpr` for type-based dispatch inside evaluation.
- `constinit` or `consteval` (in future) for compile-time constants (optional).

13.2.10 Summary Table

Table 2-1: REPL Variable and Scope Behavior

Feature	Behavior
Global variables	Persist across REPL expressions
Block variables	Exist temporarily within block scope
Shadowing	Inner blocks can shadow outer vars

Feature	Behavior
Assignment	Modifies nearest visible variable
Lookup	Searches lexical chain
Shared REPL environment	Passed to every evaluation call

13.2.11 Conclusion

This section implements full **C-style variable persistence with lexical scoping** in an interactive REPL environment. By combining a **persistent global environment** with **temporary block environments**, and enforcing consistent **variable resolution and shadowing behavior**, your interpreter now mimics the exact scoping rules of C. Using modern C++20/23 capabilities, this system is clean, memory-safe, and extensible—ready for the addition of functions, closures, modules, and more advanced scope-dependent semantics.

13.3 Enhanced Debugging Output for Language Constructs

13.3.0.1 Introduction

As an interpreter matures and supports more advanced C-style constructs, the ability to **trace, inspect, and debug** language behavior becomes essential for both users and implementers. In Version 2 of the REPL, enhanced debugging output bridges the gap between internal interpreter logic and user understanding, especially during live coding sessions. This section describes how to design and implement structured, informative, and context-aware debugging feedback using **modern C++20/23**, tailored to reflect

C-style semantics faithfully.

13.3.1 Objectives of Enhanced Debugging

- **Visualize evaluation steps** clearly.
- **Expose environment and scope transitions**.
- **Highlight variable bindings and shadowing**.
- **Trace expression trees and operator evaluations**.
- **Report runtime errors with context**.
- **Assist language designers and REPL users alike**.

13.3.2 Core Debug Output Components

To support structured debug output, your interpreter must:

1. Annotate each major language construct with traceable information.
2. Output data consistently (e.g., indentation, prefixing).
3. Allow **debugging to be toggled** without recompilation (e.g., via a global flag).

Example: Debug Options

```
struct DebugOptions {
    bool traceExpressions = false;
    bool traceScopes = false;
    bool traceAssignments = false;
    bool traceErrors = true;
};
```

Used globally during the REPL session:

```
DebugOptions debug;
```

13.3.3 Tracing Variable Access and Mutation

- **Variable Lookup**

```
Value Environment::get(const std::string& name) const {
    if (symbols.contains(name)) {
        if (debug.traceAssignments)
            std::cout << "[lookup] " << name << " => " <<
            ↪  symbols.at(name).toString() << "\n";
        return symbols.at(name);
    } else if (parent) {
        return parent->get(name);
    } else {
        if (debug.traceErrors)
            std::cerr << "[error] Undefined variable: " << name << "\n";
        throw std::runtime_error("Undefined variable: " + name);
    }
}
```

- **Assignment**

```
void Environment::assign(const std::string& name, const Value& value) {
    if (symbols.contains(name)) {
        if (debug.traceAssignments)
            std::cout << "[assign] " << name << " := " << value.toString() <<
            ↪  "\n";
        symbols[name] = value;
```

```
    } else if (parent) {
        parent->assign(name, value);
    } else {
        if (debug.traceErrors)
            std::cerr << "[error] Assignment to undeclared variable: " << name
            ↪  << "\n";
        throw std::runtime_error("Assignment to undeclared variable: " + name);
    }
}
```

13.3.4 Tracing Expression Evaluation

Each expression type should report its action if enabled:

- **Binary Expression**

```
Value BinaryExpr::eval(Environment::Ptr env) const {
    Value l = left->eval(env);
    Value r = right->eval(env);
    Value result = applyOp(op, l, r);

    if (debug.traceExpressions) {
        std::cout << "[expr] (" << l.toString() << " " << op << " " <<
        ↪  r.toString()
                  << ") => " << result.toString() << "\n";
    }

    return result;
}
```

- **Unary Expression**

```
Value UnaryExpr::eval(Environment::Ptr env) const {
    Value operandVal = operand->eval(env);
    Value result = applyUnaryOp(op, operandVal);

    if (debug.traceExpressions) {
        std::cout << "[expr] (" << op << operandVal.toString() << ") => "
                  << result.toString() << "\n";
    }

    return result;
}
```

13.3.5 Tracing Scope Creation and Destruction

Scopes are crucial for block evaluations, and debugging them helps visualize variable lifetimes.

```
void evaluateBlock(const std::vector<Statement>& stmts, Environment::Ptr outerEnv) {
    if (debug.traceScopes)
        std::cout << "[scope] Entering new block scope\n";

    auto localEnv = std::make_shared<Environment>(outerEnv);
    for (auto& stmt : stmts)
        evaluate(stmt, localEnv);

    if (debug.traceScopes)
        std::cout << "[scope] Exiting block scope\n";
}
```

13.3.6 Optional: AST Structure Visualization

Add a debug-printing method to all AST node types to visualize parsed expressions.

Example Output

```
[ast]
AssignmentExpr {
    name = x
    value = BinaryExpr {
        left = LiteralExpr(3)
        op = +
        right = LiteralExpr(4)
    }
}
```

This requires recursive AST inspection methods like:

```
void BinaryExpr::printTree(int indent = 0) const {
    std::string pad(indent, ' ');
    std::cout << pad << "BinaryExpr(" << op << ")\n";
    left->printTree(indent + 2);
    right->printTree(indent + 2);
}
```

Call `.printTree()` before evaluation when debug is enabled.

13.3.7 Enhanced Error Context and Reporting

All runtime errors should include:

- **Expression type or source**

- **Variable name**
- **Operator**
- **Suggestions**, if possible

```
try {
    Value val = expr->eval(env);
} catch (const std::exception& ex) {
    if (debug.traceErrors) {
        std::cerr << "[error] While evaluating expression: " << expr->describe() <<
        ↪  "\n";
        std::cerr << "[error] " << ex.what() << "\n";
    } else {
        throw;
    }
}
```

Implement `Expr::describe()` in each node type for meaningful output.

13.3.8 Toggleable Debug Mode

Allow enabling/disabling debug options dynamically from the REPL:

```
>>> #debug traceExpressions on
>>> #debug traceAssignments on
>>> x = 5 + 3
[expr] (5 + 3) => 8
[assign] x := 8
```

This command system can be implemented with a parser hook for `#debug`.

13.3.9 Summary of Debugging Features

Table 3-2: Interpreter Debug Output Features

Feature	Output Example
Expression trace	`[expr] (a + 3) => 7`
Variable lookup	`[lookup] a => 4`
Assignment	`[assign] x := 10`
Undefined variable error	`[error] Undefined variable: y`
Scope enter/exit	`[scope] Entering new block scope`
AST visualizer (optional)	Indented tree of expression structure

13.3.10 Conclusion

This section equips your REPL and evaluation engine with powerful, developer-friendly **debugging features** that trace the behavior of all major language constructs. By leveraging **modern C++20/23**, structured logging, and optional introspection utilities, users and developers can visualize how the interpreter evaluates expressions, resolves variables, processes blocks, and handles errors. These debugging tools not only improve development workflow but also serve as an educational layer for users experimenting with your C-style language interactively.

13.4 Milestone — Interactive C-Style Expression Evaluator

13.4.1 Introduction

This section represents a **critical milestone** in the development of your new C-style programming language: a **fully operational interactive expression evaluator**, embedded within an enhanced REPL. At this stage, the evaluator supports real-time user interaction, expression parsing, variable scoping, operator precedence, side effects, and informative debugging. The evaluator provides a **live, self-contained, and persistent environment**—mirroring the behavior of C-like expressions, while utilizing **modern C++20/23** constructs.

13.4.2 Milestone Objectives

- **Evaluate user input expressions interactively**
- **Support full C-style semantics**: precedence, associativity, scoping
- **Maintain persistent global environment**
- **Respect block scoping and variable shadowing**
- **Enable debugging output for clarity and introspection**
- **Ensure type-safe expression handling using C++20/23 features**

13.4.3 Components Realized in This Milestone

- **a) Expression Parser**

A complete parser transforms user-entered expressions into a structured **Abstract Syntax Tree (AST)**, respecting C operator precedence and associativity.

- Handles arithmetic: `1 + 2 * 3`
- Recognizes unary operations: `-x`, `!true`
- Supports assignment and compound assignment: `x = 5`, `x += 2`
- Interprets parentheses: `(3 + 4) * 2`
- Optional support for chained assignments: `a = b = 10`

- **b) AST Node Structure**

  Each AST node implements a unified `eval()` method using polymorphism:

```
struct Expr {
    virtual Value eval(Environment::Ptr env) const = 0;
    virtual ~Expr() = default;
};
```

  Expression types implemented:

  - `LiteralExpr`
  - `VariableExpr`
  - `UnaryExpr`
  - `BinaryExpr`
  - `AssignmentExpr`
  - (Optional: `BlockExpr`, `UpdateExpr` for future control structures)

13.4.4 Persistent Evaluation Environment

A **persistent global environment** tracks variable state across expressions.

```
auto globalEnv = std::make_shared<Environment>();
```

- Variables declared once remain accessible:

```
>>> x = 10
>>> x + 5
=> 15
```

- Assignments update existing variables:

```
>>> x += 1
=> 11
```

Environment chaining supports **C-style block scoping**, useful when block expressions or future control flow constructs are introduced.

13.4.5 Expression Evaluation Lifecycle

Each input line in REPL passes through:

1. **Tokenization**
2. **Parsing into AST**
3. **Evaluation** using shared environment

4. **Result printing**

Example

```
>>> a = 3
>>> b = a * (2 + 1)
>>> b - 1
```

Yields:

```
=> 3
=> 9
=> 8
```

13.4.6 Modern C++ Integration

Key C++20/23 features used:

Table 4-3: C++ Feature Usage in Evaluator Design

Feature	Purpose
`std::variant`	Runtime type abstraction (`Value`)
`std::visit`	Type-safe operation dispatch
`std::shared_ptr`	AST and environment lifetime control
`if constexpr`	Compile-time branching in evaluators
`ranges/string_view`	(optional) Efficient input processing

Feature	Purpose
`concepts`	(optional) Restrict evaluator templates

13.4.7 Debugging and Trace Output (Optional)

Enabled through a global `DebugOptions` flag:

```
>>> x = 4 + 1
[expr] (4 + 1) => 5
[assign] x := 5
=> 5
```

Helpful for:

- Verifying precedence evaluation
- Inspecting variable updates
- Tracking scope transitions

13.4.8 User Experience

A responsive and safe REPL interface:

- **Immediate feedback**: every line outputs result or error
- **Error reporting**:

  ```
  >>> y + 1
  Error: Undefined variable: y
  ```

- **No crashes**: exceptions are caught and reported
- **Stateful interaction**: full program history is respected

13.4.9 Example Session

```
>>> x = 10
=> 10
>>> y = x + 2
=> 12
>>> y += 5
=> 17
>>> {
>>>     int x = 100
>>>     x + y
>>> }
=> 117
>>> x
=> 10
```

This illustrates:

- Assignment and compound assignment
- Expression nesting and arithmetic
- Block scoping and variable shadowing
- Persistent outer variable state

13.4.10 Summary Table of Supported Features

Table 4-4: Feature Implementation Status

Feature	**Status**
Integer and float literals	Implemented
Arithmetic operations	Implemented
Unary operators	Implemented
Operator precedence	Implemented
Variable assignment	Implemented
Compound assignment	Implemented
Variable persistence	Implemented
Scoping with shadowing	Implemented
Debug output (optional)	Implemented
Safe runtime error handling	Implemented

13.4.11 Roadmap Beyond This Milestone

This milestone completes the **core REPL interaction** model. Next steps include:

- Multi-statement block execution
- Conditional and loop control structures (`if`, `while`)
- Function declaration and calls
- Built-in functions (e.g., `print`, `len`)
- Type declarations (`int`, `bool`)

- Expression folding with `consteval`
- Module or file-based scope handling

13.4.12 Conclusion

This milestone marks the **transition from a language prototype to a functioning interactive engine**. It enables users to explore expressions, understand evaluation semantics, and manipulate state in real time—all using a system that mirrors the C expression model but with modern architecture. Built with **C++20/23 idioms**, the evaluator is scalable, safe, and expressive, forming the core of the execution model for future language features.

Part V

Control Flow and Functions

Chapter 14

C-Style Statement Execution Engine

14.1 Block Scoping with {} Delimiters

14.1.0.1 Introduction

In C-style languages, **block scoping** is one of the most fundamental concepts that governs visibility, lifetime, and resolution of variables and control statements. Blocks are defined by **`{}` braces**, and everything enclosed within them operates under an **isolated lexical environment** that may inherit from its outer context but also allows for local overrides (shadowing). This section details how to **design and implement block scoping in the statement execution engine**, using **modern C++20/23 constructs**, and how `{}`-delimited blocks influence control flow and environment lifetimes.

14.1.1 Purpose of Block Scoping

Block scoping provides:

- **Lexical visibility** control

- **Safe reuse of variable names**
- **Isolation of intermediate computations**
- **Foundation for conditional (`if`, `else`) and loop (`while`, `for`) statements**
- **Predictable variable lifetime and destruction semantics**

In REPL or script-based evaluation, `{}` blocks create **temporary environments** that can nest and shadow outer scopes, consistent with C semantics.

14.1.2 Conceptual Model

Each block `{}` in the AST is treated as a **new scope level**. The interpreter must:

1. **Create a new environment** for the block.
2. **Link it to its outer environment**.
3. **Evaluate contained statements in this new environment**.
4. **Discard the environment** when the block ends.

This supports nested scopes, variable shadowing, and isolation.

Environment Chaining:

```
GlobalEnv
   ↓
Block1Env
   ↓
Block2Env
```

Each environment has a pointer to its parent:

```
class Environment {
public:
    using Ptr = std::shared_ptr<Environment>;
    explicit Environment(Ptr parent = nullptr);
    Ptr parent;
    std::unordered_map<std::string, Value> symbols;
};
```

14.1.3 Parser Recognition of Block Statements

The parser must recognize `{}` as the start and end of a block and collect a sequence of statements in between.

Example:

```
{
    int x = 5;
    y = x + 2;
}
```

This is parsed into a `BlockStmt` node:

```
struct BlockStmt : Stmt {
    std::vector<std::unique_ptr<Stmt>> statements;
};
```

14.1.4 Executing a Block Statement

Evaluation of a block statement creates a **new environment**, executes each inner statement, and then **discards the local environment** after execution ends.

Interpreter Implementation:

```
Value Interpreter::executeBlock(const BlockStmt& block, Environment::Ptr parentEnv) {
    auto localEnv = std::make_shared<Environment>(parentEnv);

    for (const auto& stmt : block.statements) {
        execute(stmt.get(), localEnv);
    }

    return Value::None(); // or result of last evaluated expression if desired
}
```

The scope ends when the block ends. All local variables declared inside disappear and do not leak.

14.1.5 Shadowing and Lifetime Behavior

Block scoping enables **variable shadowing**:

```
int x = 10;
{
    int x = 20; // shadows outer x
    print(x);   // prints 20
}
print(x);       // prints 10
```

This behavior is enforced by:

- Checking variable existence **only in current environment** during declaration.
- Allowing lookup through parent chain during access.

Declaration Logic:

```
void Environment::declare(const std::string& name, const Value& val) {
    if (symbols.contains(name)) {
        throw std::runtime_error("Variable redeclared in same scope: " + name);
    }
    symbols[name] = val;
}
```

14.1.6 Integration with Control Flow

Blocks are core to:

- `if`, `else`
- `while`, `for`
- Functions (body is a block)
- Nested blocks in compound statements

Each control structure will internally call `executeBlock(...)` for its body.

Example: `if` statement

```
if (cond) {
    // true block
} else {
    // false block
}
```

Internally:

```
if (evaluate(condExpr, env).asBool()) {
    executeBlock(ifStmt.trueBlock, env);
} else {
    executeBlock(ifStmt.falseBlock, env);
}
```

14.1.7 Debug Output for Block Scope

To aid in debugging, the REPL or interpreter can provide scope tracking:

```
if (debug.traceScopes) {
    std::cout << "[scope] Entering block\n";
}
auto localEnv = std::make_shared<Environment>(parentEnv);
...
if (debug.traceScopes) {
    std::cout << "[scope] Exiting block\n";
}
```

14.1.8 C++20/23 Enhancements

Table 1-1: Modern C++ Features and Their Application

Feature	Application
`std::shared_ptr`	Scope environment chaining and lifetime
`std::unordered_map`	Efficient symbol lookup in environments

Feature	Application
`std::variant`	Used in `Value` system for runtime type support
`if constexpr`	For evaluating specific types at compile-time
`concepts`	Optional: enforce valid AST/evaluator types

14.1.9 Example Execution Flow

Input Code:

```
int x = 1;
{
    int x = 2;
    x = x + 1;
}
x = x + 3;
```

Internal Flow:

1. Global `x = 1`
2. Enter block → local `x = 2`
3. Modify local `x → 3`
4. Exit block → local `x` discarded
5. Global `x` now updated: `1 + 3 = 4`

14.1.10 Summary

Table 1-2: Scope Behavior in Modern Interpreter Design

Feature	Supported Behavior
`{}` as scope boundary	Introduced new environment
Nested scopes	Chain to outer environments
Shadowing	Allowed inner declaration of same name
Scope destruction	Local environment discarded on exit
Debug output	Enter/exit messages (optional)

14.1.11 Conclusion

Block scoping with `{}` is a **core execution model** in all C-style languages. This section establishes how to parse, interpret, and manage scope lifetimes, enabling robust, predictable behavior for variable visibility and isolation. Backed by **C++20/23 memory-safe features**, the statement engine now honors standard C scoping semantics and prepares the foundation for introducing control structures, nested blocks, and function definitions in upcoming sections.

14.2 Conditional Statements — `if`, `else if`, `else`

14.2.0.1 Introduction

Conditional execution is the **foundation of control flow** in all C-style languages. The `if`, `else if`, and `else` statements determine the **dynamic path** a program follows at runtime. Implementing them requires a precise **evaluation engine** that respects **C semantics**—including condition evaluation, short-circuit execution, and scoped block execution.

This section presents a focused implementation of conditional constructs using modern C++20/23, leveraging shared environments, short-circuit logic, and structured AST evaluation.

14.2.1 Role of Conditional Statements

Conditional statements in a C-style language provide:

- **Branching logic**: decisions made based on evaluated expressions
- **Scoped execution**: each branch executes in its own block
- **Structured flow control**: deterministic program behavior

Example syntax supported:

```
if (x > 10) {
    print("x is large");
} else if (x > 5) {
    print("x is medium");
} else {
    print("x is small");
}
```

14.2.2 Grammar and AST Representation

Grammar (simplified):

```
if_statement:
    "if" "(" expression ")" block
    [ "else if" "(" expression ")" block ]*
    [ "else" block ]?
```

AST Node Structure:

```
struct IfStmt : Stmt {
    std::vector<std::pair<ExprPtr, BlockStmt>> branches;
    std::optional<BlockStmt> elseBranch;
};
```

Each branch includes:

- A condition (expression)
- A body (block of statements)

14.2.3 Parsing Strategy

When the parser encounters an `if` token:

1. Parse the condition inside `(...)`
2. Parse the `{...}` block that follows
3. Look ahead for zero or more `else if (...) {...}`

4. Optionally capture a final `else {...}` block

Each part is stored in `branches` and `elseBranch`.

14.2.4 Execution Logic

Evaluation of `IfStmt` follows a linear priority:

- Evaluate each condition in order
- On the first `true` condition, execute its corresponding block
- Skip the rest
- If none match and `else` exists, execute it

Execution Example in C++:

```
Value Interpreter::execute(const IfStmt& stmt, Environment::Ptr env) {
    for (const auto& [condition, block] : stmt.branches) {
        Value condResult = condition->eval(env);
        if (!condResult.isBool())
            throw std::runtime_error("Condition must evaluate to bool");

        if (condResult.asBool()) {
            executeBlock(block, env);
            return Value::None();
        }
    }

    if (stmt.elseBranch.has_value()) {
        executeBlock(stmt.elseBranch.value(), env);
    }
```

```
    return Value::None();
}
```

14.2.5 Block Isolation and Scope

Each block in a conditional (including `else`) must:

- Execute in its **own lexical environment**
- Link to the parent scope
- Allow variable declarations without leaking to outer scope

```
executeBlock(block, env); // creates a nested environment
```

14.2.6 Error Handling

Your interpreter must detect:

- **Missing parentheses**
- **Non-boolean conditions**
- **Improper block formatting**

Additionally, a runtime error occurs if a condition does not resolve to a boolean:

```
if (x) { ... } // valid if x is boolean
if (3)  // should error unless implicit conversion is defined
```

Optionally, allow numeric-to-boolean conversion if your language design supports it, as C does (0 is false, non-zero is true). Implement this via a conversion function:

```
bool asBool(const Value& val) {
    if (val.isBool()) return val.asBool();
    if (val.isInt()) return val.asInt() = 0;
    if (val.isFloat()) return val.asFloat() = 0.0;
    throw std::runtime_error("Invalid condition type");
}
```

14.2.7 Debug Output (Optional)

To support tracing during development, print conditional evaluation steps:

```
if (debug.traceConditions) {
    std::cout << "[if] Condition: " << condResult.toString()
              << " → " << (condResult.asBool() ? "true" : "false") << "\n";
}
```

14.2.8 Example Use Case

Input:

```
x = 8;
if (x > 10) {
    y = 1;
```

```
} else if (x > 5) {
    y = 2;
} else {
    y = 3;
}
```

Execution Trace:

```
[if] Condition x > 10 → false
[if] Condition x > 5 → true
[block] Executing second branch
```

Result: `y = 2`

14.2.9 C++20/23 Features in Use

Table 2-3: C++ Features for Conditional Handling

Feature	Role in Conditional Handling
`std::optional`	Optional `else` block
`std::vector`	Maintain ordered list of `if`/`else if` branches
`std::shared_ptr`	Pass environments safely between blocks
`std::variant`	Represent condition values and enable type-checking
`if constexpr`	Optional: Used in type-based condition evaluation

14.2.10 Future Extensions

Once conditionals are stable:

- Support **expression-level `if`** (ternary operator `?:`)
- Introduce **short-circuit evaluation** for logical `&&` and `||`
- Add support for **pattern matching**, if language requires
- Extend to **compile-time evaluation** via `consteval` in future phases

14.2.11 Summary Table

Table 2-4: Conditional Components and Status

Component	Status
`if` block	Implemented
`else if` chains	Implemented
`else` fallback	Implemented
Boolean evaluation	Required
Block scoping	Isolated
Debug trace	Optional

14.2.12 Conclusion

The implementation of `if`, `else if`, and `else` conditional statements brings your interpreter in line with **standard C-style control flow**. These statements are the

building blocks of logic-driven programming, and their execution model—paired with scoped environments and robust expression evaluation—establishes a strong control system. Using **modern C++20/23** tools ensures that this implementation is clean, modular, and ready for expansion into loops, functions, and more advanced flow-control mechanisms in the upcoming chapters.

14.3 Loop Constructs — `while`, `for`, `do-while`

14.3.0.1 Introduction

Looping constructs are a core part of any C-style language, enabling repeated execution of code blocks based on evaluated conditions. C offers three primary loop forms—`while`, `for`, and `do-while`—each with specific semantics around condition evaluation, scoping, and body execution. In this section, we will deeply analyze how to implement these constructs in a modern interpreter using C++20/23, maintaining C-style behavior, type safety, and environment control.

14.3.1 Overview of Loop Constructs

Table 3-5: Loop Types and Behavior Summary

Loop Type	Condition Check Time	Guaranteed Execution?	Control Variables
`while`	Before body	No	External or declared before
`for`	Before body	No	Init, condition, update
`do-while`	After body	Yes (once)	External or internal

14.3.2 Common Interpreter Requirements

All loop constructs must:

- Support **environment isolation**
- Handle **break** and **continue** correctly
- Optionally support **debug trace**
- Use **typed condition evaluation**
- Be interruptible in future async/VMs

A shared method for evaluating boolean expressions should be used:

```
bool evalCondition(const ExprPtr& condition, Environment::Ptr env) {
    Value val = condition->eval(env);
    if (val.isBool()) return val.asBool();
    if (val.isInt()) return val.asInt() = 0;
    if (val.isFloat()) return val.asFloat() = 0.0;
    throw std::runtime_error("Loop condition must evaluate to a boolean-compatible
    ↪  value.");
}
```

14.3.3 `while` Loop

3.3 while Loop

Syntax:

```
while (condition) {
    // body
}
```

AST Node:

```
struct WhileStmt : Stmt {
    ExprPtr condition;
    BlockStmt body;
};
```

Execution:

```
Value Interpreter::execute(const WhileStmt& stmt, Environment::Ptr env) {
    while (evalCondition(stmt.condition, env)) {
        try {
            executeBlock(stmt.body, std::make_shared<Environment>(env));
        } catch (BreakSignal&) {
            break;
        } catch (ContinueSignal&) {
            continue;
        }
    }
    return Value::None();
}
```

Each iteration creates a new nested environment for isolation, consistent with scoped variables declared within the loop body.

14.3.4 `do-while` Loop

Syntax:

```
do {
    // body
} while (condition);
```

AST Node:

```
struct DoWhileStmt : Stmt {
    BlockStmt body;
    ExprPtr condition;
};
```

Execution:

```
Value Interpreter::execute(const DoWhileStmt& stmt, Environment::Ptr env) {
    do {
        try {
            executeBlock(stmt.body, std::make_shared<Environment>(env));
        } catch (BreakSignal&) {
            break;
        } catch (ContinueSignal&) {
            // do nothing, proceed to condition check
        }
    } while (evalCondition(stmt.condition, env));

    return Value::None();
}
```

This structure ensures **body is executed at least once**, regardless of the condition.

14.3.5 `for` Loop

Syntax:

```
for (init; condition; update) {
    // body
}
```

AST Node:

```
struct ForStmt : Stmt {
    std::unique_ptr<Stmt> initializer;
    ExprPtr condition;
    ExprPtr update;
    BlockStmt body;
};
```

Execution:

```
Value Interpreter::execute(const ForStmt& stmt, Environment::Ptr env) {
    auto localEnv = std::make_shared<Environment>(env);

    if (stmt.initializer)
        execute(stmt.initializer.get(), localEnv);

    while (!stmt.condition || evalCondition(stmt.condition, localEnv)) {
        try {
            executeBlock(stmt.body, std::make_shared<Environment>(localEnv));
        } catch (BreakSignal&) {
```

```
            break;
        } catch (ContinueSignal&) {
            // continue to update
        }

        if (stmt.update)
            stmt.update->eval(localEnv);
    }

    return Value::None();
}
```

- **Initializer** runs once at the start
- **Condition** is evaluated before each iteration (default `true` if null)
- **Update** is executed at the end of each iteration
- **Environment** can preserve loop state across iterations

14.3.6 Break and Continue Handling

To support `break` and `continue`:

Signals:

```
struct BreakSignal : public std::exception {};
struct ContinueSignal : public std::exception {};
```

Triggered by:

```
Value Interpreter::execute(const BreakStmt&, Environment::Ptr) {
    throw BreakSignal();
}

Value Interpreter::execute(const ContinueStmt&, Environment::Ptr) {
    throw ContinueSignal();
}
```

Handled in loop body:

Each loop handler must `try-catch` these exceptions appropriately.

14.3.7 Debugging Trace Support

Optional debug output for loop activity:

```
if (debug.traceLoops) {
    std::cout << "[loop] Starting new iteration\n";
}
```

Can be toggled from REPL using `#debug traceLoops on`.

14.3.8 C++20/23 Modernization Techniques

Table 3-6: C++ Feature Usage in Loop Constructs and Control Logic

Feature	Usage
`std::shared_ptr`	For safe environment management

Feature	Usage
`std::variant`	In `Value` system for condition logic
`if constexpr`	Compile-time branching for type safety
`std::optional`	Nullable `condition` in `for` loops
`ranges` (optional)	May apply for future iterable loops

14.3.9 Example Execution

```
for (int i = 0; i < 3; i = i + 1) {
    print(i);
}
```

Output:

```
0
1
2
```

Scoping Behavior:

- `i` exists only inside the `for` loop
- After the loop, `i` is undefined

14.3.10 Summary

Table 3-7: Loop Types and Scoping Characteristics

Loop Type	Scoping	Repetition Control	Guaranteed Execution	Environment
`while`	Yes	Condition before	No	Nested
`do-while`	Yes	Condition after	Yes	Nested
`for`	Yes	Init → Cond → Body → Update	No	Nested

14.3.11 Conclusion

Implementing `while`, `for`, and `do-while` loops gives your interpreter full support for iterative control flow consistent with C semantics. By using **modern C++20/23 idioms**, the interpreter becomes safer, cleaner, and easier to maintain. These constructs not only empower users to write expressive logic but also prepare the ground for more advanced structures like iterators, lambdas, and user-defined control mechanisms in the future chapters.

14.4 Milestone — Full C-Style Statement Interpreter

14.4.0.1 Introduction

This section marks a significant achievement: the completion of a **full-featured C-style statement interpreter**, capable of parsing, executing, and managing the complete range of control flow constructs—`if`, `else`, `while`, `for`, `do-while`, and blocks `{}`—with proper **scoping, variable lifetime control, and side-effect behavior**, all faithfully aligned with C semantics. The interpreter is now able to **execute**

structured programs, enforce type correctness, and manage scope transitions, all while using clean, modular, and modern C++20/23 idioms.

14.4.1 Milestone Goals

At this point, your interpreter should:

- Evaluate complete program blocks composed of:
  - `if`, `else if`, `else`
  - `while`, `for`, `do-while` loops
  - `{}`-delimited scoped blocks
  - Assignments, compound operations, and value expressions
- Handle:
  - Lexical scoping and shadowing
  - Runtime type evaluation
  - Structured environment management
  - Interruptions via `break`, `continue`
- Be integrated into a REPL or script execution engine

14.4.2 Structural Overview

All statement types in your AST should now conform to a unified evaluation interface:

```
struct Stmt {
    virtual Value execute(Environment::Ptr env) const = 0;
    virtual ~Stmt() = default;
};
```

Each derived class (e.g., `IfStmt`, `ForStmt`, `BlockStmt`, `WhileStmt`) handles:

- Scoped execution
- Result propagation (if needed)
- Side-effect integration
- Control flow signaling (`Break`, `Continue`)

14.4.3 Statement Execution Engine

The **statement dispatch** function should be capable of dynamically evaluating any supported statement:

```
Value Interpreter::execute(const Stmt* stmt, Environment::Ptr env) {
    switch (stmt->type()) {
        case StmtType::Block: return executeBlock(*static_cast<const
        ↪  BlockStmt*>(stmt), env);
        case StmtType::If: return executeIf(*static_cast<const IfStmt*>(stmt), env);
        case StmtType::While: return executeWhile(*static_cast<const
        ↪  WhileStmt*>(stmt), env);
        case StmtType::DoWhile: return executeDoWhile(*static_cast<const
        ↪  DoWhileStmt*>(stmt), env);
        case StmtType::For: return executeFor(*static_cast<const ForStmt*>(stmt),
        ↪  env);
        case StmtType::Assignment: return executeAssignment(*static_cast<const
        ↪  AssignmentStmt*>(stmt), env);
```

```
        case StmtType::Break: throw BreakSignal();
        case StmtType::Continue: throw ContinueSignal();
        default: throw std::runtime_error("Unknown statement type");
    }
}
```

Alternatively, you can use **`std::variant` and `std::visit`** for cleaner type-safe dispatching if your AST is implemented as tagged unions.

14.4.4 Scope and Environment Integration

Scope management is enforced via nested `Environment` objects:

```
auto childEnv = std::make_shared<Environment>(parentEnv);
```

Each block creates its own environment, preserving:

- Local variables
- Lifetime isolation
- Shadowing behavior
- Parent environment access for outer symbols

You ensure **predictable memory and type behavior**, aligned with C semantics.

14.4.5 Supported Language Features at This Stage

Table 4-8: Language Feature Support Overview

Language Feature	Status
Block `{}` statement	Fully supported
`if`, `else if`, `else`	Fully supported
`while` loop	Fully supported
`do-while` loop	Fully supported
`for` loop	Fully supported
`break`, `continue`	Fully supported
Variable scoping	Fully implemented via `Environment`
Expression-based evaluation	Integrated with statement engine
Side effects	Supported in expression and control flows

14.4.6 Example Program Execution

Sample source:

```
int x = 0;
for (int i = 0; i < 5; i = i + 1) {
    if (i % 2 == 0) {
        x = x + i;
    } else {
        continue;
    }
}
```

Expected behavior:

- `x` accumulates even numbers: $0 + 2 + 4 = 6$
- Variables `i` and `x` follow scoped rules
- Loop control obeys C-style evaluation flow

14.4.7 Modern C++ Practices in Use

Table 4-9: Modern C++ Feature Usage in Language Implementation

C++ Feature	Usage Context
`std::variant`	Statement and expression representation
`std::visit`	Statement evaluation dispatch
`std::shared_ptr`	Environment and AST lifetime management
`std::optional`	For optional `else` branches or loop conditions
`if constexpr`	Compile-time control in type-specific operations
`constexpr`	Optional: constant folding and future optimizations
`ranges/views`	Optional: iteration and future dataflow constructs

These features allow cleaner, safer, and more modular interpreter design, and prepare the base for future compiler optimizations, VM integration, or JIT expansions.

14.4.8 Debug and Tracing Infrastructure

Ensure that debug capabilities exist:

```
if (debug.traceStatements) {
    std::cout << "[stmt] Executing: " << stmt->toString() << "\n";
}
```

Add fine-grained tracing for:

- Condition evaluation
- Scope entry/exit
- Loop iteration counts
- Variable updates

14.4.9 Stability and Error Handling

The interpreter must handle:

- Type mismatches in conditions
- Invalid operations (e.g., assigning to undeclared variable)
- Uninitialized variables
- Unexpected control flow (e.g., `break` outside of loop)

All errors must throw descriptive `std::runtime_error` exceptions and be catchable in the REPL or execution environment.

14.4.10 Preparing for Next Stage

With the statement engine complete, the interpreter can now evolve to:

- Add **function declarations and calls**
- Implement **return statements and scopes**
- Support **early exit mechanisms**
- Introduce **user-defined types and arrays**
- Begin **module and file-level execution**

This milestone enables full program execution in scripts or the REPL, and forms the **core foundation** for procedural programming in your C-style language.

14.4.11 Conclusion

Achieving a full C-style statement interpreter is a transformative milestone. You've now implemented the **entire execution pipeline** for core C-style control flow using safe, composable, and extensible C++20/23 architecture. This unlocks your language's capability to run structured logic-driven programs with full scope management and flow control.

The interpreter is no longer experimental—it now behaves like a foundational tool ready to support real users, script execution, and future function and object constructs.

Chapter 15

Function Implementation in C-Style

15.1 Function Declarations — `int func(int x, float y)`

15.1.0.1 Introduction

Function declarations form the foundation of modular, reusable logic in C-style languages. Declaring functions in the form of `int func(int x, float y)` not only introduces a name-bound block of code but also establishes parameterized execution with type constraints, scoped environments, and return value semantics. Implementing this in a modern C++ interpreter requires careful coordination between **AST construction**, **symbol table management**, **scoping behavior**, and **runtime invocation**.

This section details how to design and implement **function declarations** using modern C++20/23 idioms, enabling the language to behave consistently with classic C syntax while leveraging a robust value and type system.

15.1.1 Purpose and Scope

Function declarations in C-style languages:

- Define **named reusable routines**
- Specify **typed input parameters**
- Have an explicit **return type**
- Allow **type checking** during calls
- Enable **block-level lexical scope** within the function body

C-style functions must:

- Be declared before use (in interpreted mode: registered in a global symbol table)
- Store metadata: name, parameter types, body, and return type
- Support recursive and nested calls

15.1.2 Syntax Definition

Example:

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

Grammar representation:

```
function_decl:
    type identifier '(' parameter_list ')' block

parameter_list:
    [ parameter (',' parameter)* ]

parameter:
    type identifier
```

15.1.3 AST Representation

```
struct FunctionDecl : Stmt {
    std::string name;
    Type returnType;
    std::vector<std::pair<Type, std::string>> parameters;
    BlockStmt body;
};
```

Each function includes:

- A unique name
- A return type
- A parameter list with names and types
- A body (block of statements)

15.1.4 Symbol Table Registration

Functions must be stored in a **global function table** upon declaration:

```
class FunctionTable {
public:
    void registerFunction(const FunctionDecl& func);
    const FunctionDecl& getFunction(const std::string& name) const;
private:
    std::unordered_map<std::string, FunctionDecl> functions;
};
```

This table is separate from variable symbol tables to distinguish variable bindings from callable symbols.

15.1.5 Environment Preparation for Calls

Function calls must create a **new local environment** where:

- Parameters are initialized with argument values
- Local variables are declared
- Return values are tracked

Upon calling:

```
Value Interpreter::callFunction(const FunctionDecl& func, const std::vector<Value>&
↪  args) {
    auto localEnv = std::make_shared<Environment>(globalEnv);

    // Bind parameters
    for (size_t i = 0; i < func.parameters.size(); ++i) {
        const auto& [type, name] = func.parameters[i];
        if (!typeMatches(type, args[i]))
            throw std::runtime_error("Function argument type mismatch for parameter:
            ↪  " + name);
```

```
        localEnv->declare(name, args[i]);
    }

    // Execute function body
    try {
        executeBlock(func.body, localEnv);
    } catch (ReturnSignal& ret) {
        if (!typeMatches(func.returnType, ret.value))
            throw std::runtime_error("Return type mismatch in function: " +
            ↪  func.name);
        return ret.value;
    }

    if (func.returnType == Type::Void)
        return Value::None();

    throw std::runtime_error("Missing return statement in function: " + func.name);
}
```

15.1.6 Return Mechanism

Use a runtime signal to handle `return` cleanly:

```
struct ReturnSignal {
    Value value;
    explicit ReturnSignal(Value val) : value(std::move(val)) {}
};

Value Interpreter::execute(const ReturnStmt& stmt, Environment::Ptr env) {
    Value val = stmt.expr->eval(env);
    throw ReturnSignal(val);
```

```
}
```

15.1.7 Type Safety and C++20/23 Usage

The type system should support:

- `int`, `float`, `bool`, `string`, etc.
- Matching by exact type or through limited implicit conversion
- Type declarations enforced both at declaration and invocation

C++20/23 tools used:

Table 1-1: Selected C++ Features and Their Usage

Feature	Usage
`std::variant`	Representing value and type combinations
`std::optional`	Handling return type existence
`concepts` (optional)	Enforcing type compatibility rules
`consteval` (future)	Precomputed functions

15.1.8 Example

Code:

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

Internal flow:

- Registered as `FunctionDecl` in the function table
- Accepts two `int` values
- Creates a local environment on call
- Evaluates the body
- Returns an `int` result

15.1.9 Error Cases

1. Duplicate function declaration:
    - Detected in registration step
2. Mismatched return type:
    - Checked after `return`
3. Too many or too few arguments:
    - Checked before binding parameters
4. Type mismatch in arguments:
    - Enforced during call

15.1.10 Summary

Table 1-2: Function Support Components Status

Component	Implemented
Function registration	True
Typed parameters	True
Return value control	True via exception
Local scope creation	True
Type safety	True strict

15.1.11 Conclusion

Function declaration and integration is a major milestone in building a real-world, reusable C-style programming language. With proper scoping, type-checked parameters, and structured return semantics, your language now supports structured, modular programming and prepares for the implementation of **function calls, recursion, lambdas, and closures**. Modern C++20/23 features ensure your architecture is clean, maintainable, and forward-compatible for optimizations and native compilation in future stages.

15.2 Call Stack and Activation Records

15.2.0.1 Introduction

To support **function calls**, recursion, nested scopes, and return value propagation, your interpreter must simulate a **call stack**—a dynamic structure that mimics how C and other compiled languages track function execution. Each function invocation generates a **new activation record** (also known as a stack frame), which contains critical execution state such as parameter values, local variables, the return address (in compiled code), and any control information like the caller environment.
In this section, we design a fully functional **interpreter-level call stack** using modern C++20/23 facilities, tailored to dynamic interpretation and C-style semantics.

15.2.1 What Is the Call Stack?

The **call stack** is a runtime structure that:

- Records each function call as a new **frame**
- Holds the context for that invocation
- Enables **recursive and nested** function support
- Unwinds automatically on return
- Preserves **lexical scoping**, **parameter bindings**, and **execution isolation**

In interpreted systems, it does not manipulate real memory addresses but emulates this behavior using C++ classes.

15.2.2 Structure of an Activation Record

Each activation record holds the following:

```
struct ActivationRecord {
    std::string functionName;
    Environment::Ptr localEnv;
    std::optional<Value> returnValue;
};
```

Components:

- `functionName`: for diagnostics and debugging
- `localEnv`: the scoped environment for local variables and parameters
- `returnValue`: optional result of the function (set during return)

15.2.3 The Call Stack Implementation

A stack structure is required to manage active function calls:

```
class CallStack {
public:
    void push(const ActivationRecord& record);
    void pop();
    ActivationRecord& top();
    const ActivationRecord& top() const;
    size_t depth() const;

private:
    std::vector<ActivationRecord> stack;
};
```

This class mimics the CPU call stack using a `std::vector` where the top of the stack is the current executing function context.

15.2.4 Function Call Flow with the Stack

When a function is called:

1. The interpreter evaluates arguments
2. A **new environment** is created for the function
3. An **activation record** is pushed to the `CallStack`
4. The function body is executed in that environment
5. On `return`, the return value is stored and the frame is **popped**

Interpreter Example:

```
Value Interpreter::callFunction(const FunctionDecl& func, const std::vector<Value>&
↪  args) {
    auto localEnv = std::make_shared<Environment>(globalEnv);

    for (size_t i = 0; i < func.parameters.size(); ++i) {
        const auto& [type, name] = func.parameters[i];
        if (!typeMatches(type, args[i]))
            throw std::runtime_error("Argument type mismatch");
        localEnv->declare(name, args[i]);
    }

    ActivationRecord record{func.name, localEnv, std::nullopt};
    callStack.push(record);

    try {
        executeBlock(func.body, localEnv);
    } catch (ReturnSignal& signal) {
```

```
        callStack.top().returnValue = signal.value;
        callStack.pop();
        return signal.value;
    }

    callStack.pop();

    if (func.returnType == Type::Void)
        return Value::None();

    throw std::runtime_error("Missing return in function: " + func.name);
}
```

15.2.5 Benefits of a Proper Call Stack

- **Recursion Support**: Each call has isolated scope
- **Stack Tracing**: Track current and previous execution points
- **Environment Chaining**: Allows nested environments to be tied to stack levels
- **Debugging**: Print stack trace when error occurs

15.2.6 Recursive Example

Code:

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Each call to `factorial(n)` creates:

- A new activation record
- An isolated environment with `n`
- A deferred computation awaiting the next return

Upon reaching `n == 1`, the stack starts **unwinding**, returning each result upward.

15.2.7 Debugging Support

To visualize the call stack, implement:

```
void CallStack::print() const {
    std::cout << "Call Stack:\n";
    for (auto it = stack.rbegin(); it != stack.rend(); ++it) {
        std::cout << " > " << it->functionName << "\n";
    }
}
```

This aids users in tracing deep recursion or call chains during interpretation.

15.2.8 C++20/23 Modern Practices

Table 2-3: Modern C++ Features for Function Runtime and Scope Management

Feature	Purpose
`std::optional`	Store optional return values

Feature	Purpose
`std::shared_ptr`	Reference-counted environments
`std::vector`	Stack structure for activation records
`structured bindings`	Cleaner unpacking of parameters
`constexpr` / `concepts` (optional)	Enforce compile-time type safety

C++20 and C++23 features provide **clean abstraction**, **strong type guarantees**, and **simplified code design** for recursive and stack-based logic.

15.2.9 Future Extensions

Once the base call stack is stable:

- Add **stack overflow detection** (max recursion depth)
- Support **tail-call optimization** (optional)
- Capture **stack traces** for runtime error reporting
- Integrate **closures** and **lambda environments**

15.2.10 Summary Table

Table 2-4: Function Runtime Features and Support Status

Feature	Status
Function context isolation	True
Parameter binding per call	True
Recursion support	True
Stack unwinding on return	True
Return value propagation	True
Debugging / tracing support	True

15.2.11 Conclusion

The interpreter's **call stack and activation record mechanism** form the structural core that enables deep function logic, scoped variables, and recursion. This simulation aligns closely with compiled C behavior while allowing introspection and control at the interpretation level. With this in place, your interpreter now behaves like a real execution engine, capable of running user-defined functions with reliability, scope control, and advanced error tracing, all backed by modern C++20/23 features.

15.3 Parameter Passing and Type Checking

15.3.0.1 Introduction

Parameter passing and type checking are central to **safe and correct function execution** in any statically typed C-style language. This section describes the

implementation of **call-time argument binding**, **type conformity validation**, and the infrastructure to support these operations using **C++20/23 techniques**. The goal is to reproduce behavior familiar to C programmers, where function arguments must match both in **number** and **type**, and where runtime type mismatches raise descriptive errors.

15.3.1 Fundamentals of C-Style Parameter Passing

In traditional C-style syntax:

```
int add(int a, float b);
```

- The function `add` accepts an `int` and a `float`
- These are **passed by value**, unless otherwise specified
- Type matching is strict: implicit conversions (e.g., `int → float`) are limited and controlled

The interpreter must emulate this process, supporting:

- Strict **arity check** (argument count)
- Per-parameter **type compatibility check**
- Runtime **environment binding**

15.3.2 Internal Representation of Parameters

Function declarations carry parameter information as part of the AST node:

```
struct Parameter {
    Type type;
    std::string name;
};

struct FunctionDecl : Stmt {
    std::string name;
    Type returnType;
    std::vector<Parameter> parameters;
    BlockStmt body;
};
```

The `Type` enum (or class) models primitive types (`int`, `float`, `bool`, etc.), and optionally supports complex types (`array`, `pointer`, `user-defined`) in future stages.

15.3.3 Argument Evaluation and Matching

Upon a function call, the interpreter must:

1. **Evaluate each argument expression**
2. **Compare each argument type against the expected parameter type**
3. **Bind values to parameter names in a new function-local environment**

Example implementation:

```
Value Interpreter::callFunction(const FunctionDecl& func, const std::vector<ExprPtr>&
↪  argsExpr) {
    if (argsExpr.size() != func.parameters.size())
        throw std::runtime_error("Function call argument count mismatch for '" +
        ↪  func.name + "'");
```

```
    std::vector<Value> args;
    for (const auto& expr : argsExpr)
        args.push_back(expr->eval(currentEnv));

    auto localEnv = std::make_shared<Environment>(globalEnv);

    for (size_t i = 0; i < args.size(); ++i) {
        const Parameter& param = func.parameters[i];
        const Value& arg = args[i];

        if (!typeMatches(param.type, arg))
            throw std::runtime_error("Type mismatch for parameter '" + param.name +
            ↪  "' in function '" + func.name + "'");

        localEnv->declare(param.name, arg);
    }

    // Now execute the body
    // ...
}
```

15.3.4 `typeMatches` Implementation

The `typeMatches` function ensures values conform to expected types:

```
bool typeMatches(const Type& expected, const Value& actual) {
    switch (expected) {
        case Type::Int: return actual.isInt();
        case Type::Float: return actual.isFloat() || actual.isInt(); // allow
        ↪  widening
        case Type::Bool: return actual.isBool();
```

```
        case Type::String: return actual.isString();
        default: return false;
    }
}
```

Optional implicit promotions (e.g., `int → float`) should be **explicitly handled**, and narrowing conversions (e.g., `float → int`) should be disallowed unless cast is explicitly used.

15.3.5 Value Binding in Local Scope

Each parameter is bound in a **new activation environment**, separate from the caller. This is critical for recursion and lexical scoping:

```
localEnv->declare(param.name, arg);
```

This ensures that:

- Parameters are **isolated per call**
- Shadowing does not affect the caller
- Nested calls do not overwrite outer values

15.3.6 Error Handling

Common error cases and their responses:

Table 3-5: Function Argument Error Handling Cases

Error Case	Action Taken
Too few or too many arguments	Throw arity mismatch error
Type mismatch in argument	Throw with parameter name and expected type
Invalid type conversion attempt	Throw explicit cast required error
Use of undeclared parameter	Compile-time catch or runtime binding error

Example message:

```
Error: Type mismatch for parameter 'b' in call to 'add'. Expected 'float', got 'int'.
```

15.3.7 Debug Support

For tracing, you can log each parameter binding:

```
if (debug.traceFunctionCalls) {
    std::cout << "[call] Binding parameter '" << param.name << "' = " <<
    ↪  arg.toString() << "\n";
}
```

This is useful in recursive calls and debugging interpreter logic.

15.3.8 C++20/23 Usage

Modern C++ improves expressiveness and safety:

Table 3-6: Modern C++ Features for Value and Helper Abstractions

Feature	Usage
`std::variant`	Encapsulate `Value` types cleanly
`std::optional`	Represent absence of values safely
`constexpr` / `consteval`	Optimize type definitions at compile time
`concepts` (optional)	Type constraints on helper functions
`structured bindings`	Clean parameter unpacking

These features reduce boilerplate, improve diagnostics, and enable future optimizations like pre-validated compiled call graphs.

15.3.9 Realistic Example

Function declaration:

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

Interpreter behavior:

1. Registers `add` with parameters: `[int a, int b]`
2. On call: `add(5, 2)`
    - `5` and `2` evaluated as `int`

- Type match confirmed
- Values bound to `a`, `b`
- Body executed in isolated scope

15.3.10 Preparing for Next Stage

This system forms the base for:

- **Return type validation**
- **Variadic functions**
- **Function overloading (if supported)**
- **Default arguments (optional)**
- **Pass-by-reference semantics (later stage)**

15.3.11 Summary Table

Table 3-7: Function Parameter System: Feature Support

Feature	Status
Parameter declaration syntax	True
Arity checking	True
Type enforcement (static/dynamic)	True
Runtime binding	True

Feature	Status
Scoped isolation	True
Debug tracing support	True

15.3.12 Conclusion

By implementing strict **parameter passing and type checking**, your language interpreter reaches a level of reliability, predictability, and safety that mirrors real-world statically typed C-style systems. Users are protected from silent bugs due to type coercion, and your interpreter lays a firm foundation for **function recursion**, **higher-order functions**, and eventually **type-safe functional abstractions**. This precise handling of function parameters ensures correctness and enhances language trustworthiness—backed fully by modern C++20/23 capabilities.

15.4 Return Statement Handling

15.4.0.1 Introduction

Handling the `return` statement is a fundamental feature in any C-style language. It enables a function to send a value back to its caller and exit immediately. This behavior introduces **non-linear control flow**, **function-result propagation**, and **early termination** of a function block.

In an interpreter, especially one built using Modern C++ (C++20/23), `return` must be **explicitly modeled** in both the parser and runtime engine, with **type-check enforcement** and a mechanism to **exit execution flow mid-block** without executing remaining statements.

15.4.1 Purpose and Behavior of `return`

In C-style semantics:

- `return <expr>;` terminates the function immediately and returns the evaluated result
- `return;` is valid for `void` functions
- Code after a `return` is unreachable
- The type of the return expression must match the declared return type of the function

The interpreter must support:

- Immediate control exit from the function body
- Value propagation to the caller
- Type validation between returned value and function declaration

15.4.2 AST Representation of Return Statements

The `return` statement is represented in the AST as:

```
struct ReturnStmt : Stmt {
    std::optional<ExprPtr> expr;
    ReturnStmt() = default;
    ReturnStmt(ExprPtr e) : expr(std::move(e)) {}
};
```

This allows support for both `return;` and `return <expr>;` in typed and void contexts.

15.4.3 Runtime Handling via Exception-like Signal

Since C++ lacks native non-local jumps like `goto` across function contexts, **interpreter-level return handling is implemented via custom control flow signals**. The most robust approach is to use a specific exception-like mechanism:

```
struct ReturnSignal {
    Value value;
    explicit ReturnSignal(Value v) : value(std::move(v)) {}
};
```

The `execute` function for `ReturnStmt` throws this signal:

```
Value Interpreter::execute(const ReturnStmt& stmt, Environment::Ptr env) {
    if (stmt.expr) {
        Value val = stmt.expr.value()->eval(env);
        throw ReturnSignal(val);
    } else {
        throw ReturnSignal(Value::None());
    }
}
```

15.4.4 Handling in the Caller Context

The function call mechanism must **catch** the `ReturnSignal`, validate the return type, and extract the returned value:

```
Value Interpreter::callFunction(const FunctionDecl& func, const std::vector<Value>&
↪  args) {
    auto localEnv = std::make_shared<Environment>(globalEnv);
```

```
    // Bind parameters
    for (size_t i = 0; i < args.size(); ++i)
        localEnv->declare(func.parameters[i].name, args[i]);

    Value returnValue = Value::None();

    try {
        executeBlock(func.body, localEnv);
    } catch (ReturnSignal& signal) {
        returnValue = signal.value;
    }

    if (!typeMatches(func.returnType, returnValue)) {
        throw std::runtime_error("Return type mismatch in function '" + func.name +
        ↪  "'");
    }

    return returnValue;
}
```

If the function expects a `void` return (`Type::Void`), `returnValue` must be `None()`.

15.4.5 Type Checking Logic

Type matching is enforced as part of the function return validation phase. Depending on the language's strictness:

- Exact match: `int` returned for `int`
- Safe implicit conversions: `int` to `float` allowed if supported
- Mismatched types result in runtime error

```
bool typeMatches(Type expected, const Value& actual) {
    if (expected == actual.type()) return true;
    if (expected == Type::Float && actual.type() == Type::Int) return true; //
    ↪  implicit promotion
    return false;
}
```

15.4.6 Early Return in Nested Blocks

`return` must function even if placed inside nested blocks or loops:

```
int compute() {
    for (int i = 0; i < 10; ++i) {
        if (i == 5)
            return i;
    }
    return -1;
}
```

In this case:

- Loop is exited via `return`
- Any remaining statements are skipped
- Execution jumps back to the function caller

15.4.7 Debugging and Diagnostics

For development and REPL environments, log return behavior:

```
if (debug.traceReturns) {
    std::cout << "[return] Value: " << val.toString() << "\n";
}
```

Include stack trace support if the `CallStack` object is available, to trace the origin of the return.

15.4.8 C++20/23 Techniques

Table 4-8: Key C++ Features Used in Return Expression Handling

Feature	Usage
`std::optional`	Represent optional expression in `return`
`std::variant`	Represent multiple `Value` types safely
`structured bindings`	For clean unpacking of signals/values
`concepts` (optional)	Type-checked utilities
`consteval` (optional)	Used in static analysis (future stage)

These tools help keep the interpreter modular, safe, and easily extensible.

15.4.9 Sample Scenario

Function:

```
int sign(int x) {
    if (x > 0) return 1;
    if (x < 0) return -1;
    return 0;
}
```

Interpreter flow:

- `x > 0`: return 1 → signal caught, execution ends
- No further code runs
- Return value passed back to caller and validated

15.4.10 Error Scenarios

Table 4-9: Return Statement Scenarios and Behaviors

Scenario	Behavior
Return in non-function context	Runtime error (disallowed)
Missing return in non-void func	Runtime error: no value
Return type mismatch	Runtime error with details
Return unreachable	Optional warning (not runtime)

Future enhancements can support static analysis to detect missing or unreachable returns during compile or interpretation setup.

15.4.11 Summary Table

Table 4-10: Return Statement Feature Status

Feature	Status
`return <expr>` support	True
`return;` (void function)	True
Early termination of execution	True
Runtime return value propagation	True
Type-checked return enforcement	True
Error messages on invalid return	True

15.4.12 Conclusion

With complete `return` statement handling, your interpreter now mirrors real-world function execution logic, providing **safe value returns**, **control flow shortcuts**, and **runtime type enforcement**. This system ensures that function behavior is reliable and predictable, and that violations (such as type mismatches or missing returns) are immediately and clearly reported. The design is extendable and prepares your interpreter for **higher-level features** such as **first-class functions**, **lambdas**, and **exceptions**, while remaining grounded in C-style discipline and driven by Modern C++ practices.

15.5 Hands-on — C-style Function Support with Recursion

15.5.0.1 Introduction

This hands-on section demonstrates how to implement full **C-style function support with recursion** in your interpreter, including syntax, AST representation, symbol resolution, environment creation, stack handling, and type-checked return propagation. C-style recursion mirrors standard C semantics: each call has an isolated environment and local variables, but shares access to global functions and constants.
The implementation uses modern C++20/23 constructs like `std::shared_ptr`, `std::variant`, `std::optional`, and structured environments to manage call stack depth and isolation.

15.5.1 Step-by-Step Overview

We'll walk through implementing the recursive evaluation of this classic example:

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}
```

The interpreter must:

- Register the function in the global symbol table
- Evaluate calls with argument binding
- Create a new scoped environment per call

- Manage call stack and return values
- Detect base case and unwind the recursion

15.5.2 Parsing and AST Construction

Parser produces a node of type `FunctionDecl`:

```
struct FunctionDecl : Stmt {
    std::string name;
    Type returnType;
    std::vector<Parameter> parameters;
    BlockStmt body;
};
```

Call expressions produce `CallExpr` nodes:

```
struct CallExpr : Expr {
    std::string functionName;
    std::vector<ExprPtr> arguments;
};
```

15.5.3 Registering the Function

On function declaration:

```
void Interpreter::defineFunction(const FunctionDecl& func) {
    functionTable[func.name] = func;
}
```

15.5.4 Calling the Function

Function call runtime logic:

```
Value Interpreter::evalCall(const CallExpr& expr) {
    const auto& func = functionTable.at(expr.functionName);
    if (expr.arguments.size() != func.parameters.size())
        throw std::runtime_error("Argument count mismatch");

    std::vector<Value> evaluatedArgs;
    for (const auto& argExpr : expr.arguments)
        evaluatedArgs.push_back(argExpr->eval(currentEnv));

    auto newEnv = std::make_shared<Environment>(globalEnv);

    for (size_t i = 0; i < func.parameters.size(); ++i) {
        if (!typeMatches(func.parameters[i].type, evaluatedArgs[i]))
            throw std::runtime_error("Type mismatch for parameter '" +
            ↪  func.parameters[i].name + "'");
        newEnv->declare(func.parameters[i].name, evaluatedArgs[i]);
    }

    callStack.push({func.name, newEnv});

    try {
        executeBlock(func.body, newEnv);
    } catch (ReturnSignal& r) {
        callStack.pop();
        if (!typeMatches(func.returnType, r.value))
            throw std::runtime_error("Return type mismatch in function '" + func.name
            ↪  + "'");
        return r.value;
    }
```

```
    callStack.pop();

    if (func.returnType == Type::Void)
        return Value::None();

    throw std::runtime_error("Missing return in function '" + func.name + "'");
}
```

15.5.5 Handling Recursion

Since function declarations are globally registered, recursive calls resolve normally:

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}
```

This call chain:

1. Binds `n`
2. Checks base case
3. If `n > 1`, calls `factorial(n - 1)`
4. Each call stacks new environment and awaits result
5. On `n == 1`, returns 1
6. Stack unwinds with cumulative multiplication

The interpreter's call stack holds:

- Scoped variables for each `n`
- Return values per frame
- Error recovery per call

15.5.6 Runtime Value Definitions

The `Value` type stores runtime values:

```
using ValueVariant = std::variant<int, float, bool, std::string>;
struct Value {
    Type type;
    ValueVariant data;

    static Value Int(int v) { return { Type::Int, v }; }
    static Value Float(float v) { return { Type::Float, v }; }
    static Value None() { return { Type::Void, 0 }; }
};
```

Result of each recursive call is wrapped in `Value` and passed back.

15.5.7 Call Stack Visualization

Enable optional logging:

```
void Interpreter::printCallStack() const {
    std::cout << "Stack:\n";
    for (const auto& frame : callStack.frames())
        std::cout << " > " << frame.functionName << "\n";
}
```

During recursion, the stack should show:

```
 > factorial
 > factorial
 > factorial
...
```

This is essential for debugging infinite recursion and ensuring termination conditions.

15.5.8 Test Case: Recursion in REPL

User inputs:

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

factorial(5);
```

Expected output: `120`
Internal steps:

- $5 \to 4 \to 3 \to 2 \to 1$
- $1 \to$ return 1
- unwind: $2{\times}1 \to 3{\times}2 \to 4{\times}6 \to 5{\times}24 \to$ return 120

15.5.9 C++20/23 Highlights

Table 5-11: C++ Features and Their Use Cases in Runtime Evaluation

Feature	Use Case
`std::optional`	Return statements (e.g., `return;` without value)
`std::variant`	Handling multi-type runtime `Value` representations
`std::shared_ptr`	Persistent/shared environments across function calls
`structured bindings`	Clean unpacking of parameters or results during evaluation
`consteval` (future)	Optimizing precomputed recursive cases at compile-time

15.5.10 Summary Table

Table 5-12: Function Call Components Implementation Status

Component	Implemented
AST for function and call	True
Global function registry	True
Scoped environment per call	True

Component	Implemented
Call stack management	True
Recursion support	True
Return value propagation	True

15.5.11 Conclusion

With full **recursive function support**, your interpreter now mirrors one of the most important features of C-style languages: the ability to define **self-referential**, **purely mathematical**, or **control-driven logic**. Backed by strong typing and scoped environments, this hands-on addition confirms your interpreter's maturity in control flow and stack management, while leveraging modern C++ design patterns to maintain clarity, safety, and performance. This milestone sets the stage for higher-level features such as closures, first-class functions, and tail-call optimization in future expansions.

Chapter 16

Advanced C-Style Function Features

16.1 Function Pointers and First-Class Functions

16.1.0.1 Introduction

Function pointers and first-class functions extend your language's flexibility and power, enabling dynamic behavior such as passing functions as arguments, storing functions in variables, and returning functions from other functions. In traditional C, function pointers serve as a low-level mechanism to reference functions. In higher-level interpreted languages, this capability is abstracted through first-class functions, closures, and environments.
This section focuses on how to support **C-style function pointers** in syntax and semantics, and how to **internally model first-class functions** within your interpreter using modern C++20/23 features.

16.1.1 Understanding the Concept

- **Function pointer** in C: A variable holding the address of a function.

- **First-class function**: A function that can be assigned to variables, passed as arguments, and returned from other functions.
- C doesn't support closures or lexical environments, but function pointers simulate some of this flexibility.

Your language will support:

- Declaring function-type variables
- Assigning functions to these variables
- Calling functions via variables (indirect call)
- Type-checking for function signatures

16.1.2 Function Pointer Syntax Design

A C-style declaration:

```
int (*fptr)(int, int);
```

This indicates `fptr` is a pointer to a function taking two `int`s and returning `int`.
Your language can simplify this, using a clearer form:

```
function<int(int, int)> fptr;
fptr = add;
int result = fptr(3, 4);
```

Or:

```
auto fptr = add;
```

In the parser:

- Recognize function types
- Allow assignment of declared functions to function-type variables
- Ensure call expressions resolve to the correct signature

16.1.3 Internal Representation: Function Value Type

Extend your `Value` type to include a function reference:

```
using FunctionPtr = std::shared_ptr<FunctionDecl>;

using ValueVariant = std::variant<
    int, float, bool, std::string,
    FunctionPtr // function as value
>;

struct Value {
    Type type;
    ValueVariant data;

    static Value Function(FunctionPtr ptr) {
        return { Type::Function, ptr };
    }

    bool isFunction() const { return std::holds_alternative<FunctionPtr>(data); }
};
```

16.1.4 Storing and Resolving Function Values

When a function is declared:

```
functions["add"] = std::make_shared<FunctionDecl>(...);
```

When assigned:

```
Value fnVal = Value::Function(functions["add"]);
env->declare("fptr", fnVal);
```

At runtime, this `Value` behaves like a variable containing a function, and can be invoked with arguments.

16.1.5 Calling via Function Variable

To support calls like `fptr(3, 4)`, your interpreter must:

- Lookup `fptr` in the environment
- Validate it holds a `FunctionPtr`
- Extract and execute the target function

```
Value Interpreter::evalCallExpr(const CallExpr& expr) {
    Value callee = expr.target->eval(currentEnv);

    if (!callee.isFunction())
        throw std::runtime_error("Attempted to call non-function");

    FunctionPtr fn = std::get<FunctionPtr>(callee.data);
    return callFunction(*fn, evaluateArguments(expr.args));
}
```

This supports both direct calls and indirect calls via function variables.

16.1.6 Type Checking Function Signatures

Each `FunctionDecl` includes parameter types:

```
struct FunctionDecl {
    std::string name;
    Type returnType;
    std::vector<Parameter> parameters;
    BlockStmt body;
};
```

During call:

- Match number of arguments
- Match types of each argument to the corresponding parameter
- Match return type if needed

Mismatch results in runtime error.
You may enhance safety with a type descriptor:

```
struct FunctionSignature {
    Type returnType;
    std::vector<Type> paramTypes;

    bool matches(const std::vector<Value>& args) const;
};
```

This can be stored inside `Value` alongside `FunctionPtr`.

16.1.7 First-Class Function Operations

Once treated as values, functions support:

- Passing to another function

```
void applyTwice(function<int(int)> f) {
    print(f(f(2)));
}
```

- Returning from a function

```
function<int(int)> getAdder(int x) {
    function<int(int)> addX = function(y) { return x + y; };
    return addX;
}
```

To support closures, bind the environment at definition time:

```
struct Closure {
    FunctionPtr fn;
    Environment::Ptr closureEnv;
};
```

Then `ValueVariant` stores `Closure` instead of raw `FunctionPtr`.

16.1.8 Modern C++ Support

Table 1-1: Modern C++ Features Used in Function Implementation

Feature	Role
`std::variant`	Unified `Value` representation (e.g., `int`, `float`, functions, etc.)
`std::shared_ptr`	Safe and automatic ownership of user-defined functions and environments
`std::function`	Optionally used to wrap native C++ functions as callable objects
`structured bindings`	Simplifies unpacking of parameters and return values
`concepts` (optional)	Enables compile-time constraints for function signature enforcement

Modern C++ tools make dynamic function representation more type-safe and expressive.

16.1.9 Debugging and Tracing

For traceability:

```
if (debug.traceCalls)
    std::cout << "[call] Calling function via pointer: " << fn->name << "\n";
```

Also log mismatch details:

```
if (!signature.matches(args))
    throw std::runtime_error("Function signature mismatch at call site");
```

16.1.10 Summary Table

Table 1-2: Function Capabilities in the Interpreter

Capability	Supported
Function-type variables	True
Assignment of declared functions	True
Indirect function calls	True
Function signature type-checking	True
First-class function values	True
Closure (optional / future step)	True

16.1.11 Conclusion

With support for **function pointers and first-class functions**, your interpreter becomes vastly more expressive. It now handles a critical feature that underlies callback systems, functional patterns, higher-order abstractions, and dynamic behavior in modern programming. Backed by **C++20/23's advanced type and memory control**, you have a solid infrastructure to expand toward **closures, lambdas**, and even **native host-bound functions** in later chapters. This feature marks a major leap toward building a modern and capable C-style language.

16.2 Local Function Declarations

16.2.0.1 Introduction

In traditional C, function declarations are global and must appear at file scope. However, many modern languages allow **local function declarations** — functions defined **within another function's body**, scoped only to that outer function. This feature enhances encapsulation, avoids name collisions, and enables better structure in recursive or helper-based logic.
In a C-style interpreter with modern design goals, supporting local function declarations allows:

- Nested lexical scoping for functions
- Full support for recursion inside a function
- Cleaner code organization

This section explains how to extend your interpreter with local function declaration capabilities using the latest features of C++20 and C++23.

16.2.1 Local Function Use Case

Example in your language:

```
int wrapper(int x) {
    int inner(int y) {
        return y * y;
    }

    return inner(x) + inner(x + 1);
}
```

Here:

- `inner` is declared *inside* `wrapper`
- It is **not visible** outside of `wrapper`
- `inner` can recursively call itself
- `inner` must be resolved correctly at runtime

16.2.2 Parsing Local Functions

Your parser must detect and register `FunctionDecl` nodes even inside blocks:

```
stmt:
    | function_decl_stmt
    | expression_stmt
    | if_stmt
    | block_stmt
    | ...
```

Add:

```
FunctionDecl parseFunctionDeclaration() {
    expect(Token::Function);
    auto returnType = parseType();
    std::string name = parseIdentifier();
    auto params = parseParameterList();
    auto body = parseBlock();
    return FunctionDecl{name, returnType, params, body};
}
```

When parsing statements within a function block, the parser should allow these declarations and treat them like scoped variables, except their type is `function`.

16.2.3 Environment and Scope Design

Each function call creates a new `Environment`. This environment should support:

- Binding of variables (like `int x = 5`)
- Binding of functions (like `int f(int y) { ... }`)

You must modify the environment to store **local function declarations** just like it stores variables:

```
struct Environment {
    std::unordered_map<std::string, Value> values;
    std::shared_ptr<Environment> parent;

    void declare(const std::string& name, const Value& value);
    bool existsInCurrent(const std::string& name) const;
    Value get(const std::string& name) const;
};
```

On function declaration (local):

```
Value fnVal = Value::Function(std::make_shared<FunctionDecl>(...));
env->declare("inner", fnVal);
```

Now, `inner` exists only in the current function's scope.

16.2.4 Resolving Function References

When calling `inner(...)`, you must:

- Resolve `inner` in the current local environment

- Confirm it is a function type
- Use it as a `FunctionPtr` value

This reuses your general `evalCallExpr` logic:

```
Value Interpreter::evalCallExpr(const CallExpr& expr) {
    Value callee = expr.target->eval(currentEnv);

    if (!callee.isFunction())
        throw std::runtime_error("Cannot call non-function");

    return callFunction(*std::get<FunctionPtr>(callee.data), evaluatedArgs);
}
```

16.2.5 Scoping Behavior

Function declarations **shadow outer scopes**, including globals.
Example:

```
int compute() {
    int helper(int x) { return x + 1; }
    {
        int helper(int x) { return x * 2; }
        return helper(4); // calls second helper, returns 8
    }
}
```

Your interpreter must:

- Always look for the **nearest scoped definition**
- Properly destroy local functions after their block ends

16.2.6 Recursive Support in Locals

A local function can call itself:

```
int outer(int x) {
    int factorial(int n) {
        if (n <= 1) return 1;
        return n * factorial(n - 1);
    }

    return factorial(x);
}
```

In this case:

- `factorial` is bound to the current environment
- Calls resolve within the same block via `env->get("factorial")`

To avoid lookup errors, define the function in the environment *before* executing its body:

```
auto localFunc = std::make_shared<FunctionDecl>(...);
env->declare("factorial", Value::Function(localFunc));
```

16.2.7 Return Scoping and Error Prevention

Ensure that:

- `return` inside the local function returns from the *local function*, not the outer one
- The local function's body is executed in its own `Environment`

16.2.8 Diagnostics and Debug Support

To aid in development:

```
if (debug.traceFunctionDefinitions) {
    std::cout << "[define] Local function: " << name << " in current block\n";
}

if (debug.traceFunctionCalls) {
    std::cout << "[call] Local function: " << name << "\n";
}
```

Allow REPL inspection of block-local functions:

```
print typeof(wrapper.inner); // should print: function<int(int)>
```

16.2.9 C++20/23 Techniques

Table 2-3: Modern C++ Features for Function Handling

Feature	Use Case
`std::shared_ptr`	Persistent function representation
`std::optional`	Optional expression in `return`
`std::variant`	Function as runtime value
`constinit` / `consteval`	Static environment pre-fill

Feature	Use Case
`concepts` (future)	Safe constraint for callable objects

16.2.10 Summary Table

Table 2-4: Function Scope Features – Implementation Status

Feature	Implemented
Function declaration inside block	True
Lexical scoping	True
Shadowing outer/global declarations	True
Local recursion	True
Isolation of function environments	True
Type-checked function calls	True

16.2.11 Conclusion

Supporting **local function declarations** marks a significant improvement in the expressiveness and structure of your interpreter. With this feature, users can encapsulate logic within specific scopes, avoid polluting the global namespace, and write recursive helper functions that are invisible outside their defining context. This design promotes modularity, correctness, and cleanliness — qualities critical in modern interpreted languages. Backed by the precision and safety features of C++20/23,

your interpreter moves one step closer to offering advanced functional and procedural capabilities in a C-style syntax.

16.3 Built-in Function Integration

16.3.0.1 Introduction

A mature interpreted language must support not only user-defined functions but also **built-in functions**, which provide direct access to essential capabilities like input/output, mathematical operations, and system utilities. Built-in functions are often implemented in the host language (C++20/23 in this case) for performance, safety, and access to system-level features.
This section focuses on how to **define**, **register**, **store**, and **execute** built-in functions in your interpreter, while maintaining compatibility with user-defined functions and leveraging modern C++ techniques to ensure type safety and extensibility.

16.3.1 What Are Built-in Functions?

Built-in functions are:

- Predefined functions implemented in the interpreter's host language
- Available by default without user declaration
- Typically provide low-level or utility operations

Examples:

```
print("Hello");
sqrt(9.0);
```

```
clock();
strlen("hello");
```

These functions:

- May not have user-visible source code
- Can support polymorphic argument types
- Must be accessible via the same `call` mechanism as user-defined functions

16.3.2 Unified Function Representation

To treat built-in and user-defined functions uniformly, we define a new variant:

```
struct UserFunction {
    std::shared_ptr<FunctionDecl> decl;
    std::shared_ptr<Environment> closure;
};

using BuiltinFunction = std::function<Value(const std::vector<Value>&)>;

using FunctionValue = std::variant<UserFunction, BuiltinFunction>;

struct Value {
    Type type;
    std::variant<...other types..., FunctionValue> data;

    static Value Builtin(BuiltinFunction fn) {
        return Value{ Type::Function, fn };
    }
};
```

This allows `Value` to hold both built-in and user-defined functions under the same `FunctionValue` tag.

16.3.3 Defining Built-in Functions

Example: `print(...)`

```
Value builtin_print(const std::vector<Value>& args) {
    for (const auto& val : args)
        std::cout << val.toString() << " ";
    std::cout << std::endl;
    return Value::None(); // void
}
```

Built-in functions return a `Value`, accept `std::vector<Value>`, and are stored in the global environment at startup:

```
globalEnv->declare("print", Value::Builtin(builtin_print));
```

Similarly:

```
globalEnv->declare("sqrt", Value::Builtin([](const std::vector<Value>& args) -> Value
↪ {
    if (args.size() = 1 || args[0].type = Type::Float)
        throw std::runtime_error("sqrt expects one float argument");
    return Value::Float(std::sqrt(std::get<float>(args[0].data)));
}));
```

16.3.4 Calling Built-in Functions at Runtime

In the call evaluation logic, distinguish between user-defined and built-in:

```
Value Interpreter::callFunction(const Value& fnVal, const std::vector<Value>& args) {
    if (!std::holds_alternative<FunctionValue>(fnVal.data))
        throw std::runtime_error("Attempted to call non-function");

    const auto& func = std::get<FunctionValue>(fnVal.data);

    if (std::holds_alternative<UserFunction>(func)) {
        const auto& userFn = std::get<UserFunction>(func);
        return callUserFunction(userFn, args);
    } else {
        const auto& builtin = std::get<BuiltinFunction>(func);
        return builtin(args);
    }
}
```

16.3.5 Type-Checking and Safety

You must implement **defensive checks** inside built-in functions because they bypass the parser's static type checking.
Each built-in function must:

- Check argument count
- Validate argument types
- Return appropriate `Value`

For example:

```
Value builtin_strlen(const std::vector<Value>& args) {
    if (args.size() = 1 || args[0].type = Type::String)
```

```
        throw std::runtime_error("strlen expects one string argument");
    return
    ↪  Value::Int(static_cast<int>(std::get<std::string>(args[0].data).length()));
}
```

16.3.6 Registering Built-in Functions

You can encapsulate all built-ins in a single function called at startup:

```
void Interpreter::registerBuiltins() {
    globalEnv->declare("print", Value::Builtin(builtin_print));
    globalEnv->declare("strlen", Value::Builtin(builtin_strlen));
    globalEnv->declare("sqrt", Value::Builtin(builtin_sqrt));
    globalEnv->declare("clock", Value::Builtin(builtin_clock));
    // Extend with more as needed
}
```

This simplifies interpreter initialization and encourages modularity.

16.3.7 Reflection and Debugging Support

To enable user introspection:

```
Value builtin_typeof(const std::vector<Value>& args) {
    if (args.size() != 1)
        throw std::runtime_error("typeof expects 1 argument");
    return Value::String(typeToString(args[0].type));
}
```

And optionally:

```
Value builtin_is_function(const std::vector<Value>& args) {
    if (args.size() != 1)
        throw std::runtime_error("is_function expects 1 argument");
    return Value::Bool(std::holds_alternative<FunctionValue>(args[0].data));
}
```

16.3.8 Modern C++ Integration

Table 3-5: Builtin Function Feature Roles

Feature	Role
`std::function`	Builtin function signature abstraction
`std::variant`	Unified function representation
`std::optional`	Optional handling inside builtin logic
Lambdas	Fast definition of built-ins
Concepts (optional)	Validate function argument types statically

You may optionally use `constexpr` to define functions that must run at compile time (in `consteval` contexts for later extensions).

16.3.9 Example: Advanced Built-in

```
Value builtin_map(const std::vector<Value>& args) {
    if (args.size() = 2 || args[1].isFunction())
        throw std::runtime_error("map expects an array and a function");
```

```
    const auto& array = std::get<std::vector<Value>>(args[0].data);
    const auto& func = std::get<FunctionValue>(args[1].data);

    std::vector<Value> result;
    for (const auto& elem : array)
        result.push_back(callFunction(func, { elem }));

    return Value::Array(result);
}
```

16.3.10 Summary Table

Table 3-6: Builtin Function Capabilities

Capability	Supported
Built-in function registration	True
Unified calling system	True
Type-checked runtime validation	True
Custom host-implemented logic	True
Lambda-based quick definitions	True
Support for higher-order built-ins	True

16.3.11 Conclusion

Built-in function integration is a key part of building a powerful and usable interpreter. It bridges the gap between language-level abstraction and host-level efficiency. By treating built-ins as first-class function values, and using `std::function`, `std::variant`, and runtime safety checks, your language can provide rich system-level capabilities without compromising clarity or modularity. This chapter builds the foundation for adding native modules, host extensions, and system APIs later — making your C-style language practical, dynamic, and scalable.

16.4 Milestone — Complete C-Style Function System

16.4.0.1 Introduction

This milestone marks the completion of a **fully operational C-style function system** within your interpreter. The function subsystem now supports:

- Global and local user-defined functions
- Recursion and nested calls with lexical environments
- Function pointers and first-class function values
- Type checking and call validation
- Built-in function integration using native host-side logic

This architecture now mirrors the expressiveness and performance philosophy of C, while modernizing it for safe and dynamic interpretation using C++20/23.

16.4.1 Architectural Overview

The function subsystem is designed as a unified abstraction, capable of handling both:

- **Statically defined user functions** (with source code)
- **Dynamically defined host functions** (via C++ lambdas or callable objects)

Key Components:

Table 4-7: Function Call Components and Roles

Component	Role
`FunctionDecl`	AST node representing function definitions.
`FunctionValue`	Runtime wrapper around either user or built-in functions.
`Environment`	Scoped symbol table tracking function context.
`CallExpr`	AST node representing a function invocation.
`Interpreter::callFunction`	Dispatcher based on runtime type (built-in vs user-defined).

16.4.2 Execution Pipeline Summary

1. **Declaration**
   Function declarations are parsed into `FunctionDecl` nodes and stored in the symbol table with type information.

2. **Invocation**
   A call expression is evaluated, resolving the callee either as a variable or global symbol.

3. **Type Check and Dispatch**
   At runtime, the interpreter distinguishes between user-defined and built-in functions using `std::variant`.

4. **User Function Handling**

   - Creates a new environment
   - Binds parameters with evaluated arguments
   - Executes the function body
   - Catches and returns `ReturnSignal` with validated return type

5. **Built-in Function Handling**

   - Host-side lambda receives `std::vector<Value>` as arguments
   - Returns a valid `Value`
   - Implements own error handling and coercion

6. **Recursion and Closure Support**

   - Functions can call themselves
   - Lexical context is preserved for local functions
   - Recursive calls reuse the same pipeline

16.4.3 Language Capabilities Achieved

Table 4-8: Function System Feature Support

Feature	Status
Global function declarations	True
Local (nested) function declarations	True
Recursion (direct and mutual)	True
Lexical scoping for functions	True
First-class function variables	True
Function pointers and indirect calls	True
Call stack with isolated environments	True
Type-checked parameter binding	True
Type-checked return validation	True
Support for void functions	True
Built-in host-side function support	True

16.4.4 Modern C++ Implementation Features

The function system takes advantage of modern C++ features from C++20 and C++23:

Table 4-9: Modern C++ Features for Function Representation and Evaluation

Feature	Usage Example
`std::variant`	Store different function types inside `Value`
`std::shared_ptr`	Reference-counted function and environment ownership
`std::optional`	Represent optional return values
`std::function`	Abstract native built-in functions with lambda wrappers
Structured bindings	Deconstruct parameters and function return objects cleanly
Designated initializers	Clear and readable initialization of AST and runtime objects
Ranges (optional use)	Simplify iteration and filtering of arguments or locals

These tools increase expressiveness while reducing boilerplate and preventing ownership issues in scoped environments.

16.4.5 Testing and Validation Cases

To confirm a complete and correct implementation, test the following cases:

- **Simple Function**

```
int add(int x, int y) { return x + y; }
print(add(2, 3)); // 5
```

- **Recursive Function**

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
print(fib(5)); // 5
```

- **Local Function and Shadowing**

```
int run(int x) {
    int square(int a) { return a * a; }
    {
        int square(int b) { return b + b; }
        return square(x); // uses inner square
    }
}
```

- **Function Pointer**

```
function<int(int)> f = fib;
print(f(6)); // 8
```

- **Built-in Function**

```
print(strlen("hello")); // 5
```

- **Type Errors (Invalid Call)**

```
add("5", "3"); // Error: type mismatch
```

16.4.6 Developer-Focused Extensions

With this system in place, it becomes trivial to add:

- Anonymous functions (`lambda`)
- Partial application or currying
- Closures with captured outer variables
- Native host integrations (file IO, networking)
- Foreign Function Interfaces (FFI) to C or C++

16.4.7 Diagnostic and Debugging Support

Enable function tracing for debugging:

```
if (debug.traceFunctionCalls)
    std::cout << "[call] Function: " << name << "(" << argCount << " args)\n";

if (debug.traceFunctionReturns)
    std::cout << "[return] " << returnValue.toString() << "\n";
```

Also provide `typeof()`, `is_function()`, and `function_signature()` introspection in the REPL for advanced usage.

16.4.8 Final Architecture Snapshot

```
[ Parser ]
   |
[ FunctionDecl AST ]
   |
[ Symbol Table ] --+--> Global/User
                   |
               [ Environment ]
                   |
              [ FunctionValue ]
          /                      \
 [UserFunction + Closure]   [Builtin Lambda]
                   |
            [Call Dispatch]
                   |
              [Type Checking]
                   |
               [Execution]
```

16.4.9 Conclusion

This milestone confirms that your interpreter now offers a **complete, safe, and expressive function system**, rivaling mature scripting languages but staying close to C-style semantics. By combining static guarantees, lexical scoping, and dynamic dispatching, this function subsystem becomes the backbone of the language's control flow, abstraction mechanisms, and modularity.
It also opens the way to build:

- Functional programming support

- Event/callback systems
- Interpreter-hosted libraries
- Embedded APIs

Future chapters can now safely build on this robust foundation to implement modules, object systems, and concurrency — knowing that your function system is solid, modern, and extensible.

Part VI

Collections and Advanced Features

Chapter 17

Arrays and C-Style Data Structures

17.1 Static and dynamic array implementation

17.1.1 Introduction

Arrays are foundational in any C-style language, serving as contiguous data structures that map directly to memory models. Your interpreter should support both **statically-sized arrays** and **dynamically-sized arrays** to mirror the duality in C between `int a[10];` and `int* a = malloc(n * sizeof(int));`.
In this section, we build a flexible system for representing, initializing, accessing, and manipulating arrays using modern C++20/23 features. The goal is to support:

- Fixed-size arrays (`int[5]`)
- Dynamic arrays with runtime resizing
- Type checking and index validation
- Interoperability with other types like `int`, `float`, etc.

17.1.2 Conceptual Design: Static vs Dynamic

Table 1-1: Comparison of Static vs Dynamic Arrays

Feature	Static Array	Dynamic Array
Size Known at Compile	Yes	No
Allocation	Fixed during declaration	Heap-like, grows/shrinks
Syntax Example	`int a[4];`	`int[] a = new int[n];`
Lifetime	Scope-based	Managed at runtime

17.1.3 Type Representation

In your interpreter, both array types can be unified under an internal `Array` type while preserving either static or dynamic semantics:

```
struct ArrayValue {
    std::vector<Value> elements;
    std::optional<size_t> fixedSize; // nullopt if dynamic
    Type elementType;
};
```

The presence of `fixedSize` allows you to distinguish between static and dynamic arrays during validation and resizing.

17.1.4 Declaration Syntax and Semantics

Static:

```
int nums[3] = {1, 2, 3};
```

- Size known at declaration
- Initializer must match size
- No reallocation allowed

Dynamic:

```
int[] list = new int[capacity];
```

- Size determined at runtime
- May be resized or extended
- Similar to `std::vector` behavior internally

17.1.5 Runtime Value Model

Define `Value::Array` variant:

```
using ArrayPtr = std::shared\_ptr<ArrayValue>;

struct Value {
    Type type;
```

```
    std::variant<..., ArrayPtr> data;

    static Value Array(ArrayPtr arr) {
        return Value{Type::Array, arr};
    }
};
```

This enables passing arrays by reference, resizing in place, and copying if needed.

17.1.6 Array Initialization

Parser must distinguish static and dynamic:

- For static:

```
Type type = Type::Int;
size_t size = 3;
std::vector<Value> init = {Value::Int(1), Value::Int(2), Value::Int(3)};
```

- For dynamic:

```
auto dynamicArray = std::make_shared<ArrayValue>();
dynamicArray->elementType = Type::Int;
dynamicArray->elements.resize(capacity, Value::Int(0)); // default init
```

This aligns with C++'s zero-initialization or value-initialization policies.

17.1.7 Access and Bounds Checking

Implement `evalArrayAccess`:

```
Value Interpreter::evalArrayAccess(const Value& arrayVal, size_t index) {
    auto arr = std::get<ArrayPtr>(arrayVal.data);
    if (index >= arr->elements.size())
        throw std::runtime_error("Array index out of bounds");
    return arr->elements[index];
}
```

Set access:

```
arr->elements[index] = newValue;
```

17.1.8 Array Type Inference and Consistency

Type inference ensures:

- Homogeneity: all elements must be same type
- Operation safety: array of `float` cannot be assigned `string`

Enforced at:

- Declaration time (for static arrays)
- Mutation time (for dynamic arrays)

17.1.9 Array Utilities and Built-ins

Built-in functions for dynamic arrays:

```
// push_back
Value builtin_push(const std::vector<Value>& args) {
    if (args.size() != 2)
        throw std::runtime\_error("push expects array and value");

    auto arr = std::get<ArrayPtr>(args[0].data);
    if (args[1].type != arr->elementType)
        throw std::runtime\_error("Type mismatch in push");

    arr->elements.push\_back(args[1]);
    return Value::None();
}

// length
Value builtin\_length(const std::vector<Value>& args) {
    auto arr = std::get<ArrayPtr>(args[0].data);
    return Value::Int(arr->elements.size());
}
```

17.1.10 C++20/23 Techniques

Table 1-2: C++ Features for Array Handling and Debugging

Feature	Usage
`std::shared_ptr`	Shared ownership of dynamic arrays
`std::optional`	Represent static vs dynamic distinction
`std::vector`	Dynamic growth and indexing
`constinit`	Static arrays with compile-time size
`structured bindings`	Element iteration in REPL/debugging

You can also use `ranges` to filter, map, or slice arrays efficiently in REPL:

```
for (auto [i, v] : views::enumerate(arr->elements)) { ... }
```

17.1.11 Error Handling

Ensure that:

- Assigning to static arrays beyond bounds triggers error
- Dynamic arrays support `push`, `resize`, `clear`
- Accessing out of bounds gives meaningful feedback
- Type mismatches are caught early and reported precisely

17.1.12 Summary

Feature	Supported
Static array declaration	True
Dynamic array allocation	True
Element access with bounds check	True
Push and resize operations	True
Type-safe element assignments	True
Array introspection (`length()`)	True
Built-in support for array ops	True

17.1.13 Conclusion

With static and dynamic arrays in place, your interpreter now supports foundational C-style data structures, enabling indexed collection processing, batch computations, and memory-efficient access patterns. The design closely mirrors traditional C behavior while embracing dynamic capabilities, flexible memory safety, and modern C++ practices. This prepares the ground for more advanced structures like structs, maps, and user-defined types, all grounded in a robust, type-safe array model.

17.2 Array Indexing with Bounds Checking

17.2.1 Introduction

In C-style programming, array indexing is a powerful yet dangerous operation. While C allows unchecked indexing (`a[i]`), leading to undefined behavior when out of bounds, modern language design demands **runtime safety**. In your interpreter, especially using C++20/23, array indexing should retain the low-level feel of C, but with **explicit bounds checking** to detect invalid memory access early.
This section covers:

- Index evaluation
- Type validation
- Bounds checking strategy
- Error diagnostics
- Read/write operations
- How modern C++ helps you implement all this cleanly

17.2.2 Understanding Indexing Semantics

Array access in C-like syntax:

```
int x = arr[2];     // Read
arr[4] = 10;        // Write
```

Your interpreter must support:

- Safe evaluation of the `arr[index]` pattern
- Correct indexing from zero

- Validation that `index` is an integer
- Protection against out-of-bounds access
- Differentiation between read and write context

17.2.3 Internal Value Representation

Assuming the `Value` type supports arrays:

```
using ArrayPtr = std::shared_ptr<ArrayValue>;

struct ArrayValue {
    std::vector<Value> elements;
    std::optional<size_t> fixedSize;
    Type elementType;
};
```

And:

```
struct Value {
    Type type;
    std::variant<..., ArrayPtr> data;
};
```

17.2.4 Bounds Checking Logic

- **Read Access**

```
Value Interpreter::readArrayElement(const Value& arrayVal, const Value&
↪  indexVal) {
    if (arrayVal.type != Type::Array)
```

```
        throw std::runtime_error("Attempt to index a non-array value");

    if (indexVal.type != Type::Int)
        throw std::runtime_error("Array index must be of type int");

    const auto& array = std::get<ArrayPtr>(arrayVal.data);
    int idx = std::get<int>(indexVal.data);

    if (idx < 0 || static_cast<size_t>(idx) >= array->elements.size())
        throw std::runtime_error("Array index out of bounds: " +
        ↪  std::to_string(idx));

    return array->elements[idx];
}
```

- **Write Access**

```
void Interpreter::writeArrayElement(Value& arrayVal, const Value& indexVal,
↪  const Value& rhsVal) {
    if (arrayVal.type != Type::Array)
        throw std::runtime_error("Attempt to index a non-array value");

    if (indexVal.type != Type::Int)
        throw std::runtime_error("Array index must be of type int");

    auto& array = std::get<ArrayPtr>(arrayVal.data);
    int idx = std::get<int>(indexVal.data);

    if (idx < 0 || static_cast<size_t>(idx) >= array->elements.size())
        throw std::runtime_error("Array index out of bounds: " +
        ↪  std::to_string(idx));
```

```
    if (rhsVal.type != array->elementType)
        throw std::runtime_error("Type mismatch in array assignment");

    array->elements[idx] = rhsVal;
}
```

17.2.5 Performance Consideration

Even though bounds checking adds overhead, it:

- Prevents critical interpreter crashes
- Aids debugging by catching errors early
- Can be optionally disabled in production (advanced mode)

Using `[[likely]]` and `[[unlikely]]` C++23 attributes can help optimize common safe access:

```
if (idx >= 0 && static_cast<size_t>(idx) < array->elements.size()) [[likely]] {
    return array->elements[idx];
} else [[unlikely]] {
    throw std::runtime_error("Array index out of bounds");
}
```

17.2.6 Compiler-Level Optimizations and C++ Tools

Leverage:

Feature	Purpose
`std::vector`	Dynamic array backend
`std::variant`	Type-safe container for values
`std::optional`	Optional fixed size for static array tracking
`structured bindings`	Cleaner code in loops
`ranges::views::enumerate` (optional)	For debugging purposes

17.2.7 Error Diagnostics and User Feedback

When a bounds error occurs, show clear messages:

```
Error: Array index 5 out of bounds (size = 3)
At line: 12, expression: data[5]
```

If possible, include:

- Current array size
- Index attempted
- Line or source info (if AST tracks positions)

This improves trust and usability for beginners and professionals alike.

17.2.8 Extended Feature: Runtime Safe Mode

You can optionally support a ”safe mode” where bounds checks are enforced, and a ”fast mode” where they are optionally skipped for performance:

```
bool Interpreter::safeMode = true;

if (safeMode && (idx < 0 || idx >= array->size()))
    throw std::runtime_error("Out-of-bounds");
```

17.2.9 Integration into AST and REPL

Grammar:

```
array_access_expr:
    identifier '[' expression ']'
```

AST Node:

```
struct ArrayAccessExpr : Expr {
    std::unique_ptr<Expr> array;
    std::unique_ptr<Expr> index;
};
```

Evaluation:

```
Value eval(const ArrayAccessExpr& expr) {
    Value arrayVal = eval(expr.array);
    Value indexVal = eval(expr.index);
    return readArrayElement(arrayVal, indexVal);
}
```

17.2.10 Test Cases

- **Valid Access**

```
int[] a = new int[3];
a[0] = 1;
a[1] = 2;
print(a[1]); // 2
```

- **Out of Bounds**

```
print(a[5]); // Error: index out of bounds
```

- **Type Mismatch**

```
a[0] = "hello"; // Error: expected int
```

17.2.11 Summary

Capability	Status
Indexing for static arrays	True
Indexing for dynamic arrays	True
Type checking for index (must be int)	True
Type checking for assignment value	True
Bounds checking with diagnostics	True
Optional safe/unsafe mode	True

17.2.12 Conclusion

Array indexing with bounds checking ensures correctness, safety, and professionalism in your language runtime. Unlike traditional C that exposes raw memory, your C-style language introduces safe array operations without compromising expressiveness. Backed by C++20/23, you now offer the user confidence in memory usage and structure manipulation, preparing for higher abstractions like slices, views, and even multidimensional arrays in future extensions.

17.3 Pointer-like Operations (Optional)

17.3.1 Introduction

Pointer manipulation is at the core of low-level C programming, providing direct memory control, array traversal, function call indirection, and dynamic memory access. While pointers are often omitted in modern interpreted languages due to their complexity and safety risks, supporting **pointer-like operations** in your interpreter can offer great power, especially for teaching, systems simulation, or embedded-level experimentation.
This section explores how to simulate pointer behavior in a safe and controlled environment using modern C++20/23 techniques while preserving the essence of C-style pointer operations.

17.3.2 Design Philosophy

Instead of exposing raw memory addresses, simulate pointer semantics:

- **Dereferencing (`\*ptr`)**
- **Address-of (`&var`)**

- **Pointer arithmetic (`ptr + 1`)**
- **Array decay to pointer (`int[]` to `int\*`)**

Your design must:

- Avoid exposing actual host machine memory
- Simulate referencing and dereferencing using internal references or identifiers
- Keep operations well-typed and bounds-checked

17.3.3 Value Representation for Pointers

Define a pointer-like runtime object using indirection:

```
struct PointerValue {
    Value* pointee;  // Runtime reference to another value
    Type baseType;
};
```

Update `Value` variant to include:

```
std::variant<..., std::shared_ptr<PointerValue>> data;
```

Then wrap creation in:

```
static Value Pointer(Value* target, Type base) {
    return Value{Type::Pointer, std::make_shared<PointerValue>(PointerValue{target,
    ↪  base})};
}
```

17.3.4 Implementing `&` (Address-of)

When evaluating an address-of expression like `&x`, the interpreter must:

- Find the memory location (reference) of variable `x` in the current environment
- Wrap that in a `PointerValue`

```
Value Interpreter::evalAddressOf(const std::string& name, const Environment::Ptr&
↪  env) {
    Value* varPtr = env->lookupReference(name); // returns pointer to variable
    ↪  storage
    if (!varPtr) throw std::runtime_error("Undefined variable: " + name);
    return Value::Pointer(varPtr, varPtr->type);
}
```

17.3.5 Implementing `*` (Dereference)

Given a pointer value, extract the actual value:

```
Value Interpreter::evalDereference(const Value& ptrVal) {
    if (ptrVal.type != Type::Pointer)
        throw std::runtime_error("Cannot dereference non-pointer");

    auto ptr = std::get<std::shared_ptr<PointerValue>>(ptrVal.data);
    return *ptr->pointee;
}
```

For assignment via pointer:

```
void Interpreter::assignViaPointer(Value& ptrVal, const Value& newVal) {
    if (ptrVal.type != Type::Pointer)
        throw std::runtime_error("Cannot assign via non-pointer");

    auto ptr = std::get<std::shared_ptr<PointerValue>>(ptrVal.data);
    if (ptr->baseType != newVal.type)
        throw std::runtime_error("Type mismatch in pointer assignment");

    *ptr->pointee = newVal;
}
```

17.3.6 Pointer Arithmetic (Simulated)

Optional implementation (simplified):

- Allow pointer to array elements
- Move pointer index using offset:

```
struct ArrayPointer {
    std::shared_ptr<ArrayValue> array;
    size_t offset;
    Type baseType;
};
```

Extend dereference to:

```
Value Interpreter::dereferenceArrayPointer(const Value& arrPtrVal) {
    auto arrPtr = std::get<std::shared_ptr<ArrayPointer>>(arrPtrVal.data);
    if (arrPtr->offset >= arrPtr->array->elements.size())
        throw std::runtime_error("Pointer offset out of bounds");
```

```
    return arrPtr->array->elements[arrPtr->offset];
}
```

17.3.7 Simulating Array Decay to Pointer

In C, an array name in an expression (except with `sizeof`, `&`, or as `type`) decays to a pointer to the first element.
Implement similar behavior:

```
Value Interpreter::arrayToPointer(const Value& arrVal) {
    if (arrVal.type != Type::Array)
        throw std::runtime_error("Cannot convert non-array to pointer");

    auto arr = std::get<ArrayPtr>(arrVal.data);
    if (arr->elements.empty())
        throw std::runtime_error("Cannot decay empty array to pointer");

    return Value::Pointer(&arr->elements[0], arr->elementType);
}
```

This operation is implicit during expression evaluation unless explicitly overridden.

17.3.8 Debug and Trace Output

For educational use, trace pointer behavior:

```
std::cout << "[ptr] Address-of " << name << " -> simulated pointer\n";
std::cout << "[ptr] Dereference value: " << value.toString() << "\n";
```

Allow inspection in REPL:

```
print_ptr(ptr); // print address, base type
```

17.3.9 C++20/23 Enhancements

Feature	Usage
`std::shared_ptr`	Safe and tracked simulated pointer
`std::variant`	Type-safe pointer and value storage
`std::optional`	Optional offsets for pointer arithmetic
`concepts` (optional)	Enforce pointer-valid operations

Use `[[nodiscard]]` on critical pointer return values to avoid discarding them silently.

17.3.10 Test Cases

- **Address-of and Dereference**

```
int x = 5;
int* px = &x;
print(*px); // 5
```

- **Assignment via Pointer**

```
*px = 10;
print(x); // 10
```

- **Array Decay**

```
int[] a = {1, 2, 3};
int* p = a;
print(*(p + 1)); // 2
```

17.3.11 Summary

Pointer Operation	Supported
Address-of (&)	True
Dereference (*)	True
Pointer assignment	True
Pointer to array element	True
Optional pointer arithmetic	True
Type checking	True

17.3.12 Conclusion

By simulating pointer behavior safely and selectively, you bridge the gap between low-level C-style access and modern interpreter constraints. The pointer model built here is rich enough to express indirection, array traversal, and function pointers while remaining memory-safe. This forms the foundation for advanced systems programming features in your language such as memory manipulation, data structure traversal, and even interfacing with C libraries in the future.

17.4 Hands-on — C-Style Array Manipulation

17.4.1 Introduction

This hands-on section offers a practical implementation of C-style array manipulation in your interpreter, covering both static and dynamic arrays. Using the evaluation model you've already built, this exercise demonstrates:

- Array declarations
- Index-based read/write
- Boundary protection
- Built-in functions for array operations
- Integration with REPL using modern C++20/23 constructs

The aim is to ensure a smooth user experience while staying close to C syntax and semantics, enriched by type safety and memory protection using modern C++.

17.4.2 Step-by-Step Implementation

1. **Array Declaration Syntax**

   Interpreter grammar supports:

```
int a[3] = {10, 20, 30};     // Static
int[] b = new int[5];        // Dynamic
```

   Parser translates to an AST node:

```
struct ArrayDecl : Statement {
    std::string name;
    Type elementType;
    std::optional<size_t> fixedSize;
    std::vector<std::unique_ptr<Expr>> initializer;
};
```

Evaluator implementation:

```
void Interpreter::evalArrayDecl(const ArrayDecl& decl) {
    ArrayValue array;
    array.elementType = decl.elementType;

    if (decl.fixedSize.has_value()) {
        array.elements.resize(decl.fixedSize.value(),
        ↪  defaultValueForType(decl.elementType));
        for (size_t i = 0; i < decl.initializer.size(); ++i) {
            Value v = eval(*decl.initializer[i]);
            checkTypeMatch(decl.elementType, v.type);
            array.elements[i] = v;
        }
        array.fixedSize = decl.fixedSize;
    } else {
        // Dynamic with initializer
        for (auto& expr : decl.initializer) {
            Value v = eval(*expr);
            checkTypeMatch(decl.elementType, v.type);
            array.elements.push_back(v);
        }
    }

    env->define(decl.name, Value::Array(std::make_shared<ArrayValue>(array)));
```

```
}
```

2. **Indexing and Assignment**

   Example input:

```
a[1] = 100;
print(a[1]);
```

   Evaluator logic:

```
Value Interpreter::evalArrayAccess(const ArrayAccessExpr& expr) {
    Value arrayVal = eval(*expr.array);
    Value indexVal = eval(*expr.index);

    if (arrayVal.type = Type::Array || indexVal.type = Type::Int)
        throw std::runtime_error("Invalid array access");

    auto arr = std::get<ArrayPtr>(arrayVal.data);
    int idx = std::get<int>(indexVal.data);

    if (idx < 0 || static_cast<size_t>(idx) >= arr->elements.size())
        throw std::runtime_error("Index out of bounds");

    return arr->elements[idx];
}
```

   Assignment:

```
void Interpreter::evalArrayAssign(const ArrayAccessExpr& target, const Value&
↪  rhs) {
    Value arrayVal = eval(*target.array);
    Value indexVal = eval(*target.index);

    auto arr = std::get<ArrayPtr>(arrayVal.data);
    int idx = std::get<int>(indexVal.data);

    if (idx < 0 || static_cast<size_t>(idx) >= arr->elements.size())
        throw std::runtime_error("Index out of bounds");

    if (rhs.type != arr->elementType)
        throw std::runtime_error("Type mismatch in array assignment");

    arr->elements[idx] = rhs;
}
```

3. **Built-in Array Functions**

   Add functions like:

```
Value builtin_length(const std::vector<Value>& args) {
    if (args.size() = 1 || args[0].type = Type::Array)
        throw std::runtime_error("length(array) expects one array argument");
    auto arr = std::get<ArrayPtr>(args[0].data);
    return Value::Int(static_cast<int>(arr->elements.size()));
}

Value builtin_push(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::Array)
        throw std::runtime_error("push(array, value)");
```

```
    auto arr = std::get<ArrayPtr>(args[0].data);
    const Value& newElem = args[1];

    if (arr->fixedSize.has_value())
        throw std::runtime_error("Cannot push to static array");

    if (newElem.type != arr->elementType)
        throw std::runtime_error("push: type mismatch");

    arr->elements.push_back(newElem);
    return Value::None();
}
```

17.4.3 Test Programs

- **Static Array Access**

```
int a[3] = {1, 2, 3};
a[0] = 10;
print(a[0]);  // Output: 10
print(length(a)); // Error: static array - length not allowed
```

- **Dynamic Array Mutation**

```
int[] b = new int[2];
b[0] = 5;
b[1] = 6;
push(b, 7);
print(b[2]);     // Output: 7
print(length(b));// Output: 3
```

17.4.4 Debug and REPL Tracing

Enable detailed tracing:

```
std::cout << "[array] Write a[" << index << "] = " << rhs.toString() << "\n";
std::cout << "[array] Read a[" << index << "] = " << result.toString() << "\n";
```

REPL support:

```
> int[] nums = new int[2];
> nums[0] = 42;
> nums[1] = nums[0] + 3;
> print(nums[1]);   // Output: 45
```

17.4.5 C++20/23 Features in Use

Feature	Role
`std::variant`	Multi-type support for value representation
`std::shared_ptr`	Dynamic ownership for array objects
`std::optional`	Track static vs dynamic array sizing
`ranges` (if used)	Useful for looping/debug in advanced REPL versions
`constexpr if`	Simplify generic printing and debugging of array types
`[[likely]]` / `[[unlikely]]`	Optimize index checking branches

17.4.6 Summary

Feature	Implemented
Static array declaration with initializer	True
Dynamic array allocation with runtime size	True
Index access and value assignment	True
Type-safe read/write	True
Built-in `length`, `push` for dynamic use	True
Error messages for invalid access/types	True
REPL integration	True

17.4.7 Conclusion

This hands-on section turns the abstract model of arrays into a concrete, working subsystem that supports C-style syntax with modern safety and diagnostics. Your interpreter now handles not just storage and access but also expressive manipulation of data in array form. This is essential for future constructs like multidimensional arrays, slices, or data buffers. With array support complete, your language moves one step closer to being a viable C-style interpreted system, built on clean C++20/23 design.

Chapter 18

Standard Library for C-Style Language

18.1 Built-in Functions — `printf`, `scanf` Equivalents

18.1.1 Introduction

In every C-style language, the `printf` and `scanf` functions serve as foundational I/O primitives. They provide a flexible, low-level interface to interact with the user via formatted text input/output. Your interpreter must include safe, structured equivalents that follow the core semantics of these functions while embracing modern runtime safety and extensibility. This section focuses on building the **`print`** and **`input`** families of functions, mimicking `printf` and `scanf` respectively.
We will explore:

- Argument handling
- Format string parsing

- Type safety enforcement
- Runtime I/O behavior
- Integration with the interpreter core using modern C++20/23 facilities

18.1.2 Objective: Design Goals

- Emulate `printf`/`scanf` functionality
- Use safe value formats and type checks
- Support variable-length argument lists
- Enable easy extensions (e.g., file I/O later)
- Integrate with REPL and script I/O seamlessly

18.1.3 Built-in `print()` Equivalent (`printf`-like)

- **Syntax Examples:**

```
print("Hello %s, your score is %d\n", name, score);
print("Pi   %.2f", pi);
```

- **Interpreter Signature:**

```
Value builtin_print(const std::vector<Value>& args);
```

- **Format String Parsing**

  We handle simple `%` placeholders:

- `%d` → integer
- `%f` → float
- `%s` → string
- `%%` → literal %

```
std::string format_string(const std::string& fmt, const std::vector<Value>&
↪  values) {
    std::ostringstream out;
    size_t argIndex = 0;

    for (size_t i = 0; i < fmt.size(); ++i) {
        if (fmt[i] == '%' && i + 1 < fmt.size()) {
            char next = fmt[++i];
            if (next == '%') {
                out << '%';
                continue;
            }

            if (argIndex >= values.size())
                throw std::runtime_error("Not enough arguments for format
                ↪  string");

            const Value& val = values[argIndex++];
            switch (next) {
                case 'd':
                    if (val.type != Type::Int)
                        throw std::runtime_error("Expected int for %d");
                    out << std::get<int>(val.data);
                    break;
                case 'f':
                    if (val.type != Type::Float)
```

```
                        throw std::runtime_error("Expected float for %f");
                    out << std::get<double>(val.data);
                    break;
                case 's':
                    if (val.type != Type::String)
                        throw std::runtime_error("Expected string for %s");
                    out << std::get<std::string>(val.data);
                    break;
                default:
                    throw std::runtime_error("Unknown format specifier: %" +
                    ↪  std::string(1, next));
            }
        } else {
            out << fmt[i];
        }
    }

    return out.str();
}
```

- **Final Print Function**

```
Value builtin_print(const std::vector<Value>& args) {
    if (args.empty() || args[0].type != Type::String)
        throw std::runtime_error("print() requires format string as first
        ↪  argument");

    std::string fmt = std::get<std::string>(args[0].data);
    std::vector<Value> formatArgs(args.begin() + 1, args.end());

    std::string output = format_string(fmt, formatArgs);
    std::cout << output;
```

```
    return Value::None();
}
```

18.1.4 Built-in `input()` Equivalent (`scanf`-like)

- **Syntax Examples:**

```
int x;
input("%d", &x);

string name;
input("%s", &name);
```

This requires passing pointers to variables (simulated), which were defined earlier in the pointer system.

- **Interpreter Signature:**

```
Value builtin_input(const std::vector<Value>& args);
```

- **Parsing and Input Conversion**

```
void parse_input(const std::string& fmt, const std::vector<Value>& args) {
    size_t fmtIndex = 0;
    size_t argIndex = 0;

    for (; fmtIndex < fmt.size(); ++fmtIndex) {
        if (fmt[fmtIndex] == '%' && fmtIndex + 1 < fmt.size()) {
```

```
            char next = fmt[++fmtIndex];

            if (argIndex >= args.size())
                throw std::runtime_error("Too few arguments for input format");

            Value& arg = const_cast<Value&>(args[argIndex++]);

            if (arg.type != Type::Pointer)
                throw std::runtime_error("input() requires pointer arguments");

            auto ptr = std::get<std::shared_ptr<PointerValue>>(arg.data);
            Value temp;

            switch (next) {
                case 'd': {
                    int v;
                    std::cin >> v;
                    temp = Value::Int(v);
                    break;
                }
                case 'f': {
                    double v;
                    std::cin >> v;
                    temp = Value::Float(v);
                    break;
                }
                case 's': {
                    std::string v;
                    std::cin >> v;
                    temp = Value::String(v);
                    break;
                }
```

```
                default:
                    throw std::runtime_error("Unknown input format specifier");
            }

            if (ptr->baseType != temp.type)
                throw std::runtime_error("Type mismatch in input assignment");

            *ptr->pointee = temp;
        }
    }
}
```

- **Final Input Function**

```
Value builtin_input(const std::vector<Value>& args) {
    if (args.empty() || args[0].type != Type::String)
        throw std::runtime_error("input() requires format string as first
        ↪ argument");

    std::string fmt = std::get<std::string>(args[0].data);
    std::vector<Value> argList(args.begin() + 1, args.end());

    parse_input(fmt, argList);
    return Value::None();
}
```

18.1.5 REPL Integration

Make these built-ins available to user code:

```
globalEnv->define("print", Value::BuiltinFunction(builtin_print));
globalEnv->define("input", Value::BuiltinFunction(builtin_input));
```

18.1.6 Test Scenarios

```
int age;
string name;

print("Enter your name: ");
input("%s", &name);
print("Enter your age: ");
input("%d", &age);

print("Hello %s, you are %d years old\n", name, age);
```

18.1.7 C++20/23 Features Used

Feature	Usage
`std::vector<Value>`	Flexible argument lists
`std::variant`	Multi-type runtime container
`std::ostringstream`	String formatting
`std::shared_ptr`	Pointer simulation
`const_cast` (carefully)	Mutable input pointer references
`std::string_view` (optional)	Format parsing, for performance

18.1.8 Summary

Capability	Status
`print(fmt, args...)`	True
Format parsing with type checks	True
Output to standard stream	True
`input(fmt, &args...)`	True
Safe type-based value input	True
Simulated reference support	True
User-friendly diagnostics	True

18.1.9 Conclusion

This section lays the foundation for a powerful and safe standard I/O system in your interpreter. The `print()` and `input()` functions offer the flexibility of C's `printf`/`scanf`, while eliminating the memory dangers associated with format string mismatches or unsafe dereferencing. With type-aware formatting and simulated pointer references, you provide an intuitive and professional I/O model suitable for both beginner and advanced users, implemented entirely in Modern C++.

18.2 File I/O Operations

18.2.1 Introduction

File I/O is an essential component of any general-purpose programming language. In a C-style interpreted language, it should provide the familiar semantics of `fopen`, `fclose`, `fread`, `fwrite`, `fprintf`, and `fscanf` but with enhanced safety and modern runtime support. This section describes the implementation of a file I/O subsystem designed to be flexible, safe, and aligned with modern C++20/23 practices.
We will simulate a minimal **standard file API** within the interpreter, supporting:

- File handle abstraction
- Text and binary modes
- Read/write operations
- Safe file closing and error checking

18.2.2 Design Considerations

1. **File Abstraction**: File handles should be wrapped in a managed object (`FileValue`) using `std::shared_ptr<...>` to ensure RAII-style resource control.
2. **File Types**: Support for both text and binary modes.
3. **Error Handling**: Use runtime exceptions with detailed messages instead of error codes.
4. **User Syntax Example**:

```
file f = fopen("data.txt", "w");
fprintf(f, "Name: %s\n", name);
fclose(f);
```

18.2.3 File Handle Representation

Create a new internal type:

```
struct FileValue {
    std::fstream stream;
    std::string mode;
    std::string filename;
};
```

Extend `Value` variant:

```
std::variant<..., std::shared_ptr<FileValue>> data;
```

Create a constructor:

```
static Value File(std::shared_ptr<FileValue> f) {
    return Value{Type::File, f};
}
```

18.2.4 Built-in: `fopen(filename, mode)`

```
Value builtin_fopen(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::String || args[1].type !=
    ↪  Type::String)
        throw std::runtime_error("fopen() requires (string filename, string mode)");

    std::string filename = std::get<std::string>(args[0].data);
    std::string mode = std::get<std::string>(args[1].data);

    std::ios_base::openmode openMode = std::ios::binary; // base mode
```

```
    if (mode == "r")        openMode |= std::ios::in;
    else if (mode == "w")   openMode |= std::ios::out | std::ios::trunc;
    else if (mode == "a")   openMode |= std::ios::out | std::ios::app;
    else if (mode == "r+")  openMode |= std::ios::in | std::ios::out;
    else
        throw std::runtime_error("Invalid file mode: " + mode);

    auto f = std::make_shared<FileValue>();
    f->filename = filename;
    f->mode = mode;
    f->stream.open(filename, openMode);

    if (!f->stream.is_open())
        throw std::runtime_error("Cannot open file: " + filename);

    return Value::File(f);
}
```

18.2.5 Built-in: `fclose(file)`

```
Value builtin_fclose(const std::vector<Value>& args) {
    if (args.size() = 1 || args[0].type = Type::File)
        throw std::runtime_error("fclose() requires (file)");

    auto file = std::get<std::shared_ptr<FileValue>>(args[0].data);
    if (file->stream.is_open())
        file->stream.close();

    return Value::None();
}
```

18.2.6 Built-in: `fprintf(file, format, ...)`

Use the same formatting engine from `print`, but write to file stream instead of `std::cout`.

```
Value builtin_fprintf(const std::vector<Value>& args) {
    if (args.size() < 2 || args[0].type = Type::File || args[1].type = Type::String)
        throw std::runtime_error("fprintf(file, format, ...)");

    auto file = std::get<std::shared_ptr<FileValue>>(args[0].data);
    std::string fmt = std::get<std::string>(args[1].data);
    std::vector<Value> formatArgs(args.begin() + 2, args.end());

    std::string output = format_string(fmt, formatArgs);
    file->stream << output;

    if (!file->stream.good())
        throw std::runtime_error("Failed writing to file: " + file->filename);

    return Value::None();
}
```

18.2.7 Built-in: `freadline(file)` and `fwrite(file, data)`

- **`freadline()`:**

```
Value builtin_freadline(const std::vector<Value>& args) {
    if (args.size() = 1 || args[0].type = Type::File)
        throw std::runtime_error("freadline(file)");
```

```
    auto file = std::get<std::shared_ptr<FileValue>>(args[0].data);
    std::string line;
    if (!std::getline(file->stream, line))
        return Value::String("");  // End of file or error

    return Value::String(line);
}
```

- **`fwrite()`:**

```
Value builtin_fwrite(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::File || args[1].type !=
    ↪  Type::String)
        throw std::runtime_error("fwrite(file, string)");

    auto file = std::get<std::shared_ptr<FileValue>>(args[0].data);
    std::string content = std::get<std::string>(args[1].data);

    file->stream.write(content.c_str(), content.size());
    return Value::None();
}
```

18.2.8 File I/O in the REPL and Scripting

Example program:

```
file f = fopen("output.txt", "w");
fprintf(f, "Name: %s\n", name);
fclose(f);
```

Reading:

```
file f = fopen("output.txt", "r");
string line = freadline(f);
print("Line: %s\n", line);
fclose(f);
```

18.2.9 Modern C++20/23 Use

C++ Feature	Application
`std::shared_ptr`	Safe file ownership
`std::variant`	Multi-type value storage
`std::filesystem` (optional)	File path validation
`ranges` (optional)	Processing file lines in advanced REPL
`std::format` (optional)	Format replacement if enabled internally

18.2.10 Summary

Capability	Implemented
`fopen(filename, mode)`	True
`fclose(file)`	True
`fprintf(file, ...)`	True
`freadline(file)`	True

Capability	Implemented
`fwrite(file, string)`	True
Format-safe I/O	True
Error-checking	True

18.2.11 Conclusion

With file I/O functionality built into your interpreter, you bring it closer to real-world usability. It enables data logging, configuration file reading, script automation, and interaction with external tools. Unlike raw C, your implementation ensures safer, exception-based handling and strict type validation using modern C++ idioms. The file system layer can later be extended to support file buffers, memory maps, or remote sources, preserving the same unified interface for the language user.

18.3 String Manipulation Utilities

18.3.1 Introduction

String manipulation is a cornerstone of both scripting and system programming. In a C-style language, string utilities are expected to be direct, efficient, and predictable. Rather than relying on object-oriented wrappers or implicit behaviors, the functions must provide explicit control similar to `strlen`, `strcat`, `strcmp`, `strstr`, and `substr`, while using safer and more flexible modern practices drawn from C++20/23.
This section details the implementation of a robust, minimal, and extensible **string utility subsystem** in your interpreter, exposing core operations like:

- Length, slicing, concatenation

- Searching and comparison
- Trimming and casing
- Splitting and joining

All designed using value-oriented semantics with clear, well-defined signatures.

18.3.2 Internal Representation

Strings are stored as:

```
Value {
    Type type;
    std::variant<..., std::string> data;
};
```

They are immutable once created — all operations return new string values, preventing aliasing or unsafe mutation.

18.3.3 Core Built-in Functions

- **`strlen(string s)` → int**

```
Value builtin_strlen(const std::vector<Value>& args) {
    if (args.size() = 1 || args[0].type = Type::String)
        throw std::runtime_error("strlen expects a single string argument");

    return
    ↪  Value::Int(static_cast<int>(std::get<std::string>(args[0].data).size()));
}
```

- **substr(string s, int start, int length) → string**

```
Value builtin_substr(const std::vector<Value>& args) {
    if (args.size() = 3 ||
        args[0].type = Type::String ||
        args[1].type = Type::Int ||
        args[2].type = Type::Int)
        throw std::runtime_error("substr(string, int, int)");

    const std::string& str = std::get<std::string>(args[0].data);
    int start = std::get<int>(args[1].data);
    int length = std::get<int>(args[2].data);

    if (start < 0 || length < 0 || start + length >
    ↪  static_cast<int>(str.size()))
        throw std::runtime_error("Invalid substring range");

    return Value::String(str.substr(start, length));
}
```

18.3.4 Concatenation and Comparison

- **strcat(string a, string b) → string**

```
Value builtin_strcat(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::String || args[1].type !=
    ↪  Type::String)
        throw std::runtime_error("strcat expects two string arguments");

    return Value::String(std::get<std::string>(args[0].data) +
    ↪  std::get<std::string>(args[1].data));
}
```

- **strcmp(string a, string b) → int**

  (returns 0 if equal, <0 if a < b, >0 if a > b)

```
Value builtin_strcmp(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::String || args[1].type !=
    ↪  Type::String)
        throw std::runtime_error("strcmp expects two string arguments");

    int result =
    ↪  std::get<std::string>(args[0].data).compare(std::get<std::string>(args[1].data))
    return Value::Int(result);
}
```

18.3.5 Search and Replace Utilities

- **strfind(string haystack, string needle) → int**

```
Value builtin_strfind(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::String || args[1].type !=
    ↪  Type::String)
        throw std::runtime_error("strfind expects two string arguments");

    const auto& haystack = std::get<std::string>(args[0].data);
    const auto& needle = std::get<std::string>(args[1].data);

    size_t pos = haystack.find(needle);
    return Value::Int(pos == std::string::npos ? -1 : static_cast<int>(pos));
}
```

- **strreplace(string input, string from, string to) → string**

```
Value builtin_strreplace(const std::vector<Value>& args) {
    if (args.size() = 3 ||
        args[0].type = Type::String ||
        args[1].type = Type::String ||
        args[2].type = Type::String)
        throw std::runtime_error("strreplace(string, from, to)");

    std::string input = std::get<std::string>(args[0].data);
    const std::string& from = std::get<std::string>(args[1].data);
    const std::string& to = std::get<std::string>(args[2].data);

    size_t pos = 0;
    while ((pos = input.find(from, pos)) != std::string::npos) {
        input.replace(pos, from.length(), to);
        pos += to.length();
    }

    return Value::String(input);
}
```

18.3.6 Advanced String Utilities

- **strsplit(string s, string delim)** → array of strings

```
Value builtin_strsplit(const std::vector<Value>& args) {
    if (args.size() = 2 || args[0].type = Type::String || args[1].type !=
    ↪  Type::String)
        throw std::runtime_error("strsplit(string, delimiter)");
```

```
    const std::string& s = std::get<std::string>(args[0].data);
    const std::string& delim = std::get<std::string>(args[1].data);

    std::vector<Value> result;
    size_t start = 0, end = 0;

    while ((end = s.find(delim, start)) != std::string::npos) {
        result.push_back(Value::String(s.substr(start, end - start)));
        start = end + delim.length();
    }

    result.push_back(Value::String(s.substr(start)));

    return Value::Array(std::make_shared<ArrayValue>(Type::String, result));
}
```

18.3.7 Modern C++20/23 Concepts in Practice

C++ Feature	Usage
`std::string`	Core string representation
`std::string::find`	Search operations
`std::string::substr`	Slicing
`std::string::replace`	Substitution
`std::ranges` (optional)	Advanced functional composition
`consteval`, `constexpr`	Safe compile-time parsing (future use)

C++ Feature	Usage
`std::format` (C++20)	Optional internal implementation layer

18.3.8 REPL Examples and Integration

```
print("Length: %d\n", strlen("hello"));
string part = substr("language", 0, 4); // "lang"
string joined = strcat(part, "code");   // "langcode"
int cmp = strcmp("a", "b");             // -1
print("Compare result: %d\n", cmp);
```

Split:

```
array words = strsplit("red,green,blue", ",");
print("First word: %s\n", words[0]);
```

18.3.9 Summary

Utility	Implemented
`strlen`	True
`substr`	True
`strcat`	True
`strcmp`	True
`strfind`	True

Utility	Implemented
`strreplace`	True
`strsplit`	True

All functions follow C-style call patterns but leverage type-safe, exception-aware implementations.

18.3.10 Conclusion

String manipulation utilities in your interpreter complete the usability layer of your standard library. By exposing foundational and frequently used string functions, you equip language users with tools for scripting, parsing, and formatting tasks with minimal friction. The C-style design philosophy is preserved while offering the robustness of modern C++20/23 features. These functions can later evolve to support Unicode, internationalization, and regex matching, building upon the stable base you've now established.

18.4 Milestone — Usable Standard Library for Our Language

18.4.1 Introduction

This milestone marks the successful formation of a **core standard library** that is functionally complete, extensible, and usable for everyday scripting and procedural programming in your new C-style interpreted language. While the language's grammar, parser, evaluator, and scoping system form the backbone, the standard library provides the muscle — enabling real-world coding without reinventing low-level mechanisms.

This section formalizes and reviews the integrated built-in modules — from I/O to strings — and outlines next steps for maintaining and expanding the standard library using modern C++20/23 idioms.

18.4.2 Components of the Standard Library

By this point, the following components have been implemented and tested interactively through the REPL and scripts:

- **Formatted I/O Layer**
  - `print(...)`, `printf(...)`
  - `input()`, `scanf(...)`
  - Type-safe formatting and variadic argument support
  - Integration with C++20's `std::format` (optional internal optimization)
- **File I/O System**
  - `fopen(filename, mode)`
  - `fclose(file)`
  - `fprintf(file, ...)`, `freadline(file)`, `fwrite(file, string)`
  - RAII-style management using `std::shared_ptr<FileValue>`
  - Text and binary mode support
  - Internal error checking using exception-based design
- **String Utilities**
  - `strlen`, `substr`, `strcat`, `strcmp`, `strfind`, `strreplace`, `strsplit`

  - Immutable string operations using `std::string`
  - Clear and predictable behavior aligned with C semantics
  - Use of standard C++ string manipulation with boundary validation

- **Array Utilities**
  - Dynamic arrays: `int[] a = new int[10]`
  - Static arrays: `int a[5] = {1, 2, 3, 4, 5}`
  - Built-ins: `length(array)`, `push(array, value)`
  - Bounds-checked read/write
  - `Value` representation with `std::variant` and `std::shared_ptr<ArrayValue>`

18.4.3 Architectural Strengths of the Current Design

Feature	Benefit
Type-safe Value System	Built on modern C++ `std::variant` and `std::optional`
RAII Resource Handling	Clean-up of files and arrays managed via `shared_ptr`
Exception-Based Failures	Simplifies error propagation and diagnostics
Interpreter Hooks	Each built-in function can be registered dynamically in the REPL

Feature	Benefit
C-style Syntax Compliance	Familiar function names and semantics preserve ease of migration
Modular Built-in Registration	Decoupled function implementations allow expansion without tight coupling

18.4.4 Usability from a Language User's View

The standard library enables scenarios like:

- **Logging to File:**

```
file f = fopen("log.txt", "a");
fprintf(f, "User logged in at %s\n", timestamp);
fclose(f);
```

- **Parsing CSV Data:**

```
file f = fopen("data.csv", "r");
while (true) {
    string line = freadline(f);
    if (strlen(line) == 0) break;
    array fields = strsplit(line, ",");
    print("Field[0]: %s\n", fields[0]);
}
fclose(f);
```

- **Text Processing:**

```
string name = "  John ";
string trimmed = strreplace(name, " ", "");
print("Welcome, %s\n", trimmed);
```

This level of functionality places the language in a position comparable to interpreted languages like Lua or early Python — lightweight but expressive and usable.

18.4.5 Modern C++ Foundation

All standard library functions are written using **safe, modern C++ constructs**:

C++ Feature	Role in Library Implementation
`std::variant`	Discriminated union for `Value` type
`std::shared_ptr`	Memory-managed resource objects (files, arrays)
`std::string`	Immutable, efficient string operations
`std::optional`	Optional return types and safety wrappers
`constexpr if`	Compile-time logic in helper templates (optional)
`std::format` (optional)	Precise formatted output support

By not relying on legacy C-style `char*` or pointer arithmetic, your interpreter avoids memory corruption, undefined behavior, and type confusion — problems traditionally plaguing C-based runtimes.

18.4.6 Documentation Strategy

Each standard function should be documented in a modular, discoverable format, such as:

```
### Function: strlen(string s)
- **Description**: Returns the number of characters in the string.
- **Parameters**: `s` - the string to measure.
- **Returns**: Integer length.
- **Errors**: Throws if input is not a string.
```

This allows future IDE or REPL autocomplete systems to display contextual help.

18.4.7 Future Expansion Plan

Now that a usable base is achieved, additional areas for future standard library development include:

Area	Description
Math Utilities	`sqrt`, `abs`, `sin`, `cos`, `round`, etc.
Date/Time Functions	`now()`, `format_date()`, `timestamp()`
Data Serialization	JSON string generation and parsing
Regex Support	Pattern matching with string input
Network I/O	Sockets, HTTP client requests
System Commands	File existence, `exec`, environment variables

18.4.8 Summary of This Milestone

Goal	Status
I/O functions (`print`, `scanf`)	True
File stream API (`fopen`, `fprintf`)	True
String functions (`strlen`, `strcat`)	True
Array manipulation	True
C-style syntax & usage	True
REPL integration	True
Error handling with exceptions	True
Modular design and extensibility	True

18.4.9 Conclusion

This milestone concludes the creation of a **usable, consistent, and expressive standard library** that brings your C-style interpreted language into practical usability for general programming. Backed by modern C++20/23 design principles, the library offers strong foundations for extension, safety, and performance.

With this achieved, your language now supports real scripting use cases and educational experiments — while remaining a living system ready to grow into a fully featured runtime. Future efforts can now shift toward concurrency, data structures, and domain-specific modules.

Part VII

Production Quality Features

Chapter 19

Comprehensive Error Handling

19.1 Runtime Error Reporting with C-Style Context

19.1.1 Introduction

Error reporting defines the boundary between a user-friendly language and a frustrating one. C-style languages are typically terse and fast, but they historically suffer from cryptic or unsafe error feedback (e.g., segmentation faults, undefined behavior, and uninitialized memory errors). Your interpreter should **retain the clarity of C's simplicity** while **avoiding its dangers** by providing precise, friendly, and **context-aware** error reports during runtime.
This section introduces an error handling architecture grounded in **C-style semantics**, but elevated by **modern C++20/23 constructs**, delivering:

- Full runtime diagnostics,
- Token-level error tracking,
- Function scope and call-trace visibility,

- Type-safe structured error classes,
- Friendly output for both REPL and script executions.

19.1.2 Runtime Error System Requirements

Before designing, clearly define the goals of your interpreter's error reporting system:

Objective	Explanation
Location-Aware	Errors should be reported with line and column numbers.
Cause-Specific	The error type should describe the actual fault (not just “runtime error”).
Semantic Traceability	Identify in which function or block the error occurred.
Safe Propagation	Errors must never corrupt internal state.
User-Level Readability	Output must be readable even by non-experts.
Interpreter-Controllable	Errors should be catchable or suppressible when required.

19.1.3 Error Kind Classification (Structured Typing)

To offer meaningful diagnostics, we classify runtime errors into a well-defined enumeration:

```
enum class RuntimeErrorKind {
    DivisionByZero,
    InvalidType,
```

```
    IndexOutOfBounds,
    NullDereference,
    UndefinedVariable,
    FunctionNotFound,
    ArgumentMismatch,
    FileIOError,
    ArithmeticOverflow,
    AccessViolation,
    TypeCoercionFailure,
    General
};
```

Each category allows the interpreter to:

- Match errors programmatically,
- Display relevant error messages,
- Act differently based on the error kind (e.g., halt vs. recover).

19.1.4 Error Class Design in Modern C++

Instead of using raw exceptions, we define a **structured error class**:

```
struct RuntimeError {
    RuntimeErrorKind kind;
    std::string message;
    std::string functionName;
    std::string filename;
    int line;
    int column;
    std::vector<std::string> callStack;
```

```
    RuntimeError(RuntimeErrorKind kind,
                 std::string message,
                 std::string funcName,
                 std::string file,
                 int line,
                 int column,
                 std::vector<std::string> trace = {})
        : kind(kind), message(std::move(message)),
          functionName(std::move(funcName)), filename(std::move(file)),
          line(line), column(column), callStack(std::move(trace)) {}
};
```

This format supports:

- File-based execution,
- Script debugging,
- REPL error reporting,
- And future visual debugging tools.

Optional: Use `std::source_location` (C++20) for auto-captured metadata:

```
#include <source_location>

RuntimeError make_error(RuntimeErrorKind kind,
                        std::string message,
                        const std::source_location& loc =
                        ↪  std::source_location::current()) {
    return RuntimeError(kind, std::move(message),
```

```
                        loc.function_name(), loc.file_name(), loc.line(),
                        ↪ loc.column());
}
```

19.1.5 Generating Errors in the Evaluation Engine

Wherever a runtime fault is possible, error generation should follow strict rules.

Example:

Division by Zero:

```
Value eval_div(const Value& lhs, const Value& rhs, EvalContext& ctx) {
    if (rhs.is_zero()) {
        throw RuntimeError(RuntimeErrorKind::DivisionByZero,
            "Division by zero",
            ctx.currentFunction(),
            ctx.currentFilename(),
            ctx.currentLine(),
            ctx.currentColumn(),
            ctx.callStack());
    }
    return lhs / rhs;
}
```

Undefined Variable Access:

```
Value resolve_variable(const std::string& name, EvalContext& ctx) {
    if (!ctx.hasVariable(name)) {
        throw RuntimeError(RuntimeErrorKind::UndefinedVariable,
            "Undefined variable: " + name,
            ctx.currentFunction(),
            ctx.currentFilename(),
```

```
            ctx.currentLine(),
            ctx.currentColumn(),
            ctx.callStack());
    }
    return ctx.getVariable(name);
}
```

19.1.6 Centralized Error Display Logic

At the top-level REPL or script executor:

```
try {
    interpreter.eval(script);
} catch (const RuntimeError& err) {
    print_error(err);
}
```

Error printer function:

```
void print_error(const RuntimeError& err) {
    std::cerr << "[Runtime Error] " << to_string(err.kind) << ": " << err.message <<
    ↪  "\n";
    std::cerr << "  at " << err.functionName << " (" << err.filename
              << ":" << err.line << ":" << err.column << ")\n";

    if (!err.callStack.empty()) {
        std::cerr << "  Call Stack:\n";
        for (const auto& frame : err.callStack) {
            std::cerr << "    → " << frame << "\n";
        }
    }
```

```
}
```

19.1.7 REPL vs File Execution Behavior

Mode	Error Format Example
REPL	`Error in <REPL> (line 1, col 8)`
Script	`Error in main.csl (line 32, col 12)`

Maintain mode-sensitive error formatting using a unified interface, but allow special REPL diagnostics if needed.

19.1.8 Benefits of Structured Error Reporting

Feature	Outcome
Categorized errors	Easier debugging and testable interpreter behavior
Precise line/column info	Accurate source mapping
Call stack awareness	Real-world debugging and language education
File/function tagging	Support for multi-file projects and modularity
Modern C++ safety	No raw pointer crashes or memory faults

19.1.9 Error Propagation and Catching (Optional Feature)

You can provide language-level `try/catch` for advanced users:

```
try {
    int x = 10 / 0;
} catch (RuntimeError e) {
    print("Caught error: %s\n", e.message);
}
```

Implementation involves:

- Wrapping evaluation nodes in `try-catch`
- Converting C++ error to language `Value` form
- Binding the `catch` variable in block scope

19.1.10 Internal Logging and Debugging

For development purposes, include an optional internal debug logger:

```
void log_error_debug(const RuntimeError& err) {
    std::ofstream log("interpreter_errors.log", std::ios::app);
    log << "[" << to_string(err.kind) << "] " << err.message
        << " in " << err.functionName << " at " << err.filename
        << ":" << err.line << ":" << err.column << "\n";
}
```

This silently builds a log that can assist in unit testing or bug triaging in user environments.

19.1.11 Future Work

Enhancement	Description
Stack Trace Compression	Show only relevant user function calls
Source Highlighting	Display offending line with^ marker (like Clang)
Recovery Mode	Continue REPL after recoverable errors
IDE Protocol	Integrate with LSP-style error messages
Multi-language Error Messages	For internationalization of tooling

19.1.12 Summary Checklist

Feature	Implemented
Structured error class with location data	True
Enum-based error classification	True
File + REPL error format distinction	True
Full line/col/function diagnostics	True
Call stack recording	True
Friendly user-readable output	True
Modern C++ safety and flexibility	True

19.1.13 Conclusion

By implementing a C-style runtime error reporting system powered by modern C++20/23, you bridge the simplicity of C with the precision and resilience of modern tooling. Users receive clear, consistent, and contextual feedback during development — making the language not only powerful but approachable and reliable. This design becomes foundational for future debugging tools, static analyzers, REPL enhancements, and educational use cases.

The ability to report and trace runtime errors cleanly is the **hallmark of a mature interpreter** — and your language now has it.

19.2 Stack Traces for Function Calls

19.2.1 Overview

Runtime error handling becomes vastly more useful when accompanied by a precise and informative **stack trace**. In traditional compiled C environments, stack traces are largely left to debuggers or OS-level tools. However, in an interpreter—especially one designed for educational, interactive, or general-purpose use—a well-constructed **function call stack trace** is indispensable. It allows developers to trace the sequence of function calls that led to a runtime fault and to understand the behavior of nested logic or recursion in their scripts.

In this section, we focus on how to design and integrate a full-featured, lightweight, and expressive stack trace mechanism inside your interpreter. The implementation will follow a clear modular structure, make full use of Modern C++20 and C++23 features, and stay faithful to the philosophy of C-style programming.

19.2.2 Motivation for Stack Traces in Interpreters

Stack traces are not just a tool for error reporting. They serve several strategic roles:

- **Debugging and diagnosis**: Help developers trace the source of errors and understand the exact flow of function calls.
- **REPL inspection**: Provide users with real-time feedback during experimentation.
- **Script transparency**: Offer insights during the execution of longer script files with multiple abstraction layers.
- **Educational clarity**: Reveal control flow behavior, particularly helpful for learners and instructors using the language.

In C-style languages, the function call stack is conceptually linear and procedural. Reproducing this model in the interpreter ensures that the language behaves predictably and debuggably.

19.2.3 Architectural Design of the Call Stack

The interpreter's call stack is a dynamic, first-in-last-out (FILO) data structure. It maintains frames of active function calls. Each frame must retain sufficient metadata to support debugging:

```
struct CallFrame {
    std::string functionName;
    std::string fileName;
    int line;
    int column;
};
```

This structure is stored in an ordered container:

```
using CallStack = std::vector<CallFrame>;
```

Why `std::vector`?

- It allows efficient push/pop behavior at the end.
- It preserves call order for tracing.
- It supports backtracing from deepest to shallowest call.

19.2.4 Entering and Exiting Functions

Each time a function is called, a new frame must be pushed onto the stack:

```
void Interpreter::enterFunction(const std::string& funcName,
                                const std::string& file,
                                int line,
                                int col) {
    callStack.emplace_back(CallFrame{funcName, file, line, col});
}
```

On return or error exit, the function frame is removed:

```
void Interpreter::exitFunction() {
    if (!callStack.empty()) {
        callStack.pop_back();
    }
}
```

These operations should be invoked at the start and end of each function call node in the AST evaluation phase.

For **exception safety**, it is best to use RAII (Resource Acquisition Is Initialization):

```
struct StackGuard {
    Interpreter& interp;
    StackGuard(Interpreter& i, const CallFrame& frame) : interp(i) {
        interp.callStack.push_back(frame);
    }
    ~StackGuard() {
        interp.callStack.pop_back();
    }
};
```

Then in evaluation:

```
void Interpreter::evalFunctionCall(...) {
    StackGuard guard(*this, CallFrame{funcName, file, line, col});
    // perform function body evaluation
}
```

This ensures the call stack is correctly unwound even when exceptions are thrown.

19.2.5 Capturing the Stack Trace on Error

Your `RuntimeError` type must carry a **snapshot** of the current call stack at the time the error is thrown. This preserves full call context, regardless of what the program does next.

```
struct RuntimeError {
    RuntimeErrorKind kind;
    std::string message;
    std::string functionName;
    std::string filename;
```

```
    int line;
    int column;
    CallStack trace;

    RuntimeError(RuntimeErrorKind k,
                 std::string msg,
                 std::string func,
                 std::string file,
                 int l,
                 int c,
                 CallStack currentStack)
        : kind(k),
          message(std::move(msg)),
          functionName(std::move(func)),
          filename(std::move(file)),
          line(l),
          column(c),
          trace(std::move(currentStack)) {}
};
```

Usage Example:

```
throw RuntimeError(RuntimeErrorKind::UndefinedVariable,
                   "Variable 'x' is not defined",
                   currentFunction,
                   currentFile,
                   currentLine,
                   currentColumn,
                   interpreter.getCallStack());
```

The function `interpreter.getCallStack()` returns a deep copy of the current call stack.

19.2.6 Stack Trace Presentation

Stack traces are printed **from the innermost function upward** to the root:

```
void printStackTrace(const CallStack& trace) {
    if (trace.empty()) return;

    std::cerr << "Call Stack:\n";
    for (auto it = trace.rbegin(); it != trace.rend(); ++it) {
        std::cerr << "  → " << it->functionName
                  << " (" << it->fileName
                  << ": line " << it->line
                  << ", col " << it->column << ")\n";
    }
}
```

This style mimics modern compilers and runtime systems, such as LLVM or Python, and is easier to read than top-down traces.

19.2.7 Special Considerations for REPL

The REPL lacks an associated file. To address this, adopt a naming convention for virtual sources:

- Use `"<REPL>"` as the default filename.
- Still store line and column numbers (per REPL input buffer).

REPL trace example:

```
Runtime Error: DivisionByZero
  → divide (line 1, col 7) in <REPL>
```

19.2.8 Recursive and Deep Call Chains

For recursive programs or deeply nested functional chains, the call stack may grow indefinitely. To prevent overflow, define a hard limit:

```
constexpr size_t MaxCallDepth = 1024;
```

Guard the call entry function:

```
void Interpreter::enterFunction(...) {
    if (callStack.size() >= MaxCallDepth) {
        throw RuntimeError(RuntimeErrorKind::StackOverflow,
            "Exceeded maximum call depth",
            currentFunction, currentFile, currentLine, currentColumn, callStack);
    }
    ...
}
```

This mirrors stack overflow behavior in compiled C, without risking host process crashes.

19.2.9 Filtering Internal Frames

During development or when exposing a clean trace to users, internal helper functions may clutter the trace. You can introduce filtering:

```
void filterSystemFrames(CallStack& trace) {
    trace.erase(
        std::remove_if(trace.begin(), trace.end(),
                       [](const CallFrame& frame) {
                           return frame.functionName.starts_with("__internal_");
                       }),
```

```
        trace.end());
}
```

This optional step ensures only user-defined functions are shown.

19.2.10 Language-Level Integration (Optional)

Future versions of your language may expose stack trace functionality to users:

```
try {
    call_something();
} catch (e) {
    print_stack_trace(e);
}
```

This requires:

- Capturing the `RuntimeError` as a value in the language
- Exposing its fields and trace through language interfaces

This turns the error system into a **runtime-inspectable object model**.

19.2.11 Benefits of a Structured Stack Trace System

definecolorheadergraygray0.85

Capability	Interpreter Impact
Stack trace per error	Complete visibility into program behavior

Capability	Interpreter Impact
Line/column/function/file metadata	Accurate source diagnostics
REPL-safe handling	Uniform UX in both scripts and console input
Recursive safety via max depth	Protection against interpreter stack blowout
System frame filtering	Cleaner, user-focused debugging
RAII-based entry/exit	Safe stack management without leaks or skips
Debug log export	Support for logging and crash reports

19.2.12 Summary and Next Steps

The call stack system in your interpreter is now capable of capturing, storing, and presenting full execution context for every function call. This enables comprehensive error diagnostics, a crucial aspect of developer trust and system robustness.

Feature	Status
Structured `CallFrame` model	Implemented
Manual stack entry and exit	Implemented
RAII stack safety	Implemented
Stack snapshot with error object	Implemented
Reverse-order trace printing	Implemented
REPL and file support	Implemented
Max depth recursion control	Implemented

Feature	Status
Trace filtering support	Optional

19.2.13 Conclusion

A modern interpreted C-style language must go beyond mere evaluation and parsing—it must offer an **introspective runtime environment**. The function call stack trace system forms a key pillar of that experience. It allows both beginner and advanced users to see the behavior of their code, recognize flaws, and trust the language as a diagnostic partner.

By modeling the stack trace system carefully in C++20/23 and integrating it with structured error handling, your interpreter attains a new level of professionalism and usability. This system also prepares the foundation for stepping debuggers, exception handling mechanisms, and visual development tools in later stages of your language's growth.

19.3 Memory Error Detection and Reporting

19.3.1 Introduction: The Role of Memory Error Handling in C-Style Interpreters

In traditional C or C++ systems, memory errors such as buffer overflows, use-after-free, and null pointer dereferencing are common sources of critical bugs, segmentation faults, and security exploits. Although interpreters operate at a higher level of abstraction, simulating pointer-like and memory-sensitive operations still exposes the runtime to similar issues if not managed correctly.

A robust C-style interpreter must detect and **report memory misuse in real**

time, helping the user understand the *when*, *where*, and *why* behind each fault.
Unlike compiled environments where memory is accessed directly, interpreters offer an opportunity to **instrument every access**, validate its safety, and **deliver meaningful error diagnostics** with precise context.
This section focuses on building a **memory-safe infrastructure** for your interpreter with clear, traceable error messages, leveraging C++20 and C++23 language features to maintain safety, clarity, and efficiency.

19.3.2 Understanding Memory Semantics in C-Style Languages

Before implementing detection, we must model memory according to C-style language expectations:

- Memory may be **manually allocated** and freed.
- Array-like structures simulate **contiguous memory blocks**.
- Pointer semantics may exist, even abstractly.
- Memory addresses can be **dereferenced, indexed, or passed** around.
- Access outside valid regions must be treated as a fatal fault.

Errors to capture include:

- Accessing deallocated memory (use-after-free)
- Out-of-bounds array access
- Dereferencing null or invalid memory
- Double freeing memory

- Reading from uninitialized memory

The interpreter must not crash on such actions, but instead halt the user program with rich diagnostic information.

19.3.3 Virtual Memory Representation

To detect and isolate memory faults, memory should be modeled using managed objects:

```
struct MemoryObject {
    std::vector<std::optional<Value>> data;
    bool isAllocated;
    bool isInitialized;
    std::string sourceLocation;
};
```

Key decisions:

- Use `std::optional<Value>` per element to track initialization.
- Store per-object allocation status and origin metadata.
- Represent memory with an ID or handle rather than pointers.

```
using MemoryID = int;

class VirtualHeap {
    std::unordered_map<MemoryID, MemoryObject> memory;
    MemoryID nextID = 0;

public:
```

```
    MemoryID allocate(size_t size, const std::string& context);
    void deallocate(MemoryID id);
    Value& write(MemoryID id, size_t index);
    Value read(MemoryID id, size_t index) const;
    void markInitialized(MemoryID id, size_t index);
    bool isAllocated(MemoryID id) const;
    ...
};
```

This controlled environment enables full access tracking and safe invalidation detection.

19.3.4 Use-After-Free Detection

When a memory block is deallocated, it must be marked unusable:

```
void VirtualHeap::deallocate(MemoryID id) {
    auto it = memory.find(id);
    if (it == memory.end() || !it->second.isAllocated) {
        throw RuntimeError(
            RuntimeErrorKind::DoubleFree,
            "Attempt to deallocate already-freed memory block (ID = " +
            ↪  std::to_string(id) + ")",
            currentFunction,
            currentFile,
            currentLine,
            currentColumn,
            currentCallStack
        );
    }

    it->second.isAllocated = false;
}
```

All future accesses to this ID raise a use-after-free error:

```
void VirtualHeap::ensureAllocated(MemoryID id) const {
    auto it = memory.find(id);
    if (it == memory.end() || !it->second.isAllocated) {
        throw RuntimeError(
            RuntimeErrorKind::UseAfterFree,
            "Access to memory that has been freed (ID = " + std::to_string(id) + ")",
            ...
        );
    }
}
```

19.3.5 Out-of-Bounds Access Detection

When accessing array-like memory:

```
Value& VirtualHeap::write(MemoryID id, size_t index) {
    ensureAllocated(id);
    auto& block = memory.at(id);
    if (index >= block.data.size()) {
        throw RuntimeError(
            RuntimeErrorKind::OutOfBoundsAccess,
            "Index " + std::to_string(index) +
            " out of bounds (size = " + std::to_string(block.data.size()) + ")",
            ...
        );
    }

    if (!block.data[index].has_value()) {
        block.data[index] = Value(); // Optional: default-initialize
    }
```

```
    return block.data[index].value();
}
```

Every array-like or pointer-like access must go through this boundary-guarded function.

19.3.6 Null Pointer and Uninitialized Access

Null pointers in this model are simulated by a special `MemoryID`, e.g.:

```
constexpr MemoryID NULL_PTR = -1;
```

Any use of `NULL_PTR` must be rejected on access:

```
if (id == NULL_PTR) {
    throw RuntimeError(
        RuntimeErrorKind::NullDereference,
        "Attempted to dereference null pointer",
        ...
    );
}
```

To simulate uninitialized memory, access may check the optional:

```
if (!block.data[index].has_value()) {
    throw RuntimeError(
        RuntimeErrorKind::UninitializedAccess,
        "Reading uninitialized memory at index " + std::to_string(index),
        ...
    );
}
```

This enables behavior similar to tools like Valgrind or Clang Sanitizers, but integrated into your interpreter.

19.3.7 Enhanced Error Reporting

Each memory error must provide:

- Error type (e.g., `OutOfBoundsAccess`)
- Description
- Function and location of the fault
- Stack trace

```
RuntimeError(
    RuntimeErrorKind::OutOfBoundsAccess,
    "Index 8 out of range for array of size 5",
    currentFunction,
    currentFile,
    currentLine,
    currentColumn,
    getCallStack()
);
```

The output should follow a unified format:

```
[Memory Error] UseAfterFree:
  → Memory block at ID 27 was accessed after deallocation
  → in function allocate_buffer() at script.csl:32
  → called from main() at script.csl:6
```

Include:

- ID of the memory object (for debugging)
- Variable name or tag (if assigned)
- Full call stack
- File, function, and line of fault

19.3.8 Debugging Tools and Memory State Dump

Support for memory state introspection during debugging:

```
void VirtualHeap::debugPrint() const {
    for (const auto& [id, obj] : memory) {
        std::cout << "Memory ID: " << id
                  << ", Allocated: " << std::boolalpha << obj.isAllocated
                  << ", Size: " << obj.data.size()
                  << ", Initialized: " << obj.isInitialized
                  << ", Source: " << obj.sourceLocation << "\n";
    }
}
```

Integrate into REPL with a command like:

```
__debug_memory()
```

Which prints all heap objects and states.

19.3.9 Optional: Tagged Memory Blocks

Support variable-based labeling of memory blocks:

```
var buf = malloc(128) as "network_rx_buffer";
```

This label is stored in `MemoryObject` and reflected in error messages:

```
[Memory Error] OutOfBoundsAccess:
  → Index 140 exceeds bounds of memory "network_rx_buffer" (size = 128)
```

Improves debugging in large programs and complex systems.

19.3.10 Preventing Memory Leaks

While interpreters usually free all memory on exit, you can optionally provide leak detection:

```
void VirtualHeap::checkLeaks() {
    for (const auto& [id, obj] : memory) {
        if (obj.isAllocated) {
            std::cerr << "Warning: Memory leak detected for block ID " << id
                      << " allocated at " << obj.sourceLocation << "\n";
        }
    }
}
```

Call `checkLeaks()` at program end or REPL shutdown.

19.3.11 C++20/23 Features Used

definecolorheadergraygray0.85

C++ Feature	Purpose
`std::optional`	Track uninitialized memory
`consteval`	Define compile-time memory constants
`std::format` (C++20)	Advanced error message formatting
Structured bindings	Iterate over memory table cleanly
Concepts / `requires`	Validate heap interfaces and accessors
Ranges and Views	Implement debugging filters and analytics

These features increase code clarity, safety, and maintainability.

19.3.12 Summary of Memory Error Handling Capabilities

Error Type	Handled?	Detection Method
Use-after-free	True	`isAllocated` flag and access guard
Double free	True	Repeated deallocation prevention
Out-of-bounds access	True	Index vs. `vector::size()` check
Null dereference	True	Special sentinel ID
Uninitialized access	True	`std::optional<Value>` check
Leak detection (optional)	True	End-of-life sweep

19.3.13 Conclusion

Simulating C-style memory safely inside an interpreter is both a technical necessity and a powerful learning opportunity. With clear abstraction and rigorous runtime checks, your interpreter can:

- Model the low-level experience of C
- Protect users from silent, undefined behavior
- Educate developers through meaningful feedback
- Provide tooling similar to AddressSanitizer and Valgrind

By integrating memory safety deeply into the interpreter's virtual memory subsystem, your language becomes not only a tool for scripting or prototyping, but a **teaching environment, debugging companion, and experimentation lab**.

19.4 Hands-on — Robust Error Handling System

19.4.1 Overview

As your interpreter approaches production quality, error handling becomes a pillar of reliability, usability, and debuggability. A robust error handling system must:

- Handle errors gracefully without terminating the entire interpreter unexpectedly.
- Provide detailed, structured, and human-readable diagnostics.
- Preserve stack traces and source code context to guide debugging.
- Be extensible to handle future language constructs (e.g., file I/O, networking).

- Offer a unified interface for runtime, parsing, semantic, and system-level errors.

This section guides you through the **design, implementation, and integration** of a complete error handling system suitable for a modern C-style interpreted language built with C++20/23.

19.4.2 Design Objectives

To build a powerful error handling system, the following goals are set:

Objective	Description
Clarity	Report meaningful and concise error messages
Contextual Awareness	Include source file, function, and line/column numbers
Stack Trace Integration	Show precise chain of function calls leading to the error
Granular Categorization	Classify errors by kind (e.g., logic, memory, type, IO)
Extensibility	Enable adding new error kinds without breaking the system
Consistency Across Subsystems	Lexing, parsing, runtime, and REPL should use a shared error framework

19.4.3 Core Error Structure

Every error is represented by a `RuntimeError` object. Here's a minimal but powerful design:

```
enum class RuntimeErrorKind {
    TypeMismatch,
    UndefinedVariable,
    DivisionByZero,
    NullPointerAccess,
    OutOfBounds,
    UseAfterFree,
    DoubleFree,
    InvalidFunctionCall,
    StackOverflow,
    GeneralRuntimeError,
    IOError,
    InternalCompilerError
};

class RuntimeError : public std::exception {
public:
    RuntimeErrorKind kind;
    std::string message;
    std::string file;
    std::string function;
    int line;
    int column;
    std::vector<std::string> callStack;

    RuntimeError(RuntimeErrorKind kind,
                 std::string message,
                 std::string file,
                 std::string function,
                 int line,
                 int column,
                 std::vector<std::string> callStack)
```

```
        : kind(kind), message(std::move(message)), file(std::move(file)),
          function(std::move(function)), line(line), column(column),
          callStack(std::move(callStack)) {}

    const char* what() const noexcept override {
        return message.c_str();
    }

    void print(std::ostream& os = std::cerr) const;
};
```

Each instance encapsulates full context: **what happened**, **where it happened**, and **why it matters**.

19.4.4 Printing a Structured ErrorReport

The `print()` method formats output with complete diagnostic details:

```
void RuntimeError::print(std::ostream& os) const {
    os << "[Runtime Error] " << to_string(kind) << ": " << message << '\n';
    os << "  at " << function << " (" << file << ':' << line << ',' << column <<
    ↪  ")\n";

    if (!callStack.empty()) {
        os << "  Call Stack:\n";
        for (const auto& frame : callStack) {
            os << "    → " << frame << '\n';
        }
    }
}
```

Example output:

```
[Runtime Error] TypeMismatch: Cannot assign 'float' to 'int'
  at assign_value (main.csl:24,14)
  Call Stack:
    → assign_value
    → main
```

19.4.5 Integrating with Runtime Execution

At every sensitive point in your interpreter, throw specific errors:

```
if (!symbolTable.contains(varName)) {
    throw RuntimeError(
        RuntimeErrorKind::UndefinedVariable,
        "Variable '" + varName + "' is not declared in this scope",
        currentFile,
        currentFunction,
        currentLine,
        currentColumn,
        context.callStack()
    );
}
```

For operations like division:

```
if (rhsValue == 0) {
    throw RuntimeError(
        RuntimeErrorKind::DivisionByZero,
        "Attempted to divide by zero",
        ...
    );
}
```

Memory errors use similar logic with access guards and deallocation flags.

19.4.6 Top-Level Catch and Error Propagation

The interpreter's REPL, test harness, or host application should catch and report all `RuntimeError` exceptions:

```
try {
    interpreter.evaluate(inputCode);
} catch (const RuntimeError& err) {
    err.print();
}
```

This ensures errors are not swallowed silently and don't crash the host process.

19.4.7 Lexical and Syntax Error Integration

Beyond runtime, compile-time errors (lexing/parsing) must also conform to the system:

```
struct ParseError {
    std::string expected;
    std::string found;
    std::string file;
    int line;
    int column;
};

struct LexicalError {
    std::string message;
    int line;
    int column;
};
```

These can be caught and translated into structured runtime messages during the REPL cycle.

19.4.8 Extending with Logging

Optionally, support logging all errors to a file or memory for debugging:

```
class ErrorLog {
    std::vector<RuntimeError> history;
    std::ostream* logStream = nullptr;

public:
    void record(const RuntimeError& err) {
        history.push_back(err);
        if (logStream) err.print(*logStream);
    }

    void setOutput(std::ostream& os) { logStream = &os; }
};
```

At shutdown, dump logs:

```
void ErrorLog::dumpAll() const {
    for (const auto& err : history)
        err.print(std::cout);
}
```

19.4.9 C++20/23 Features in Use

Feature	Use Case
`enum class`	Type-safe error categories
`std::format` (C++20)	Message formatting without external libraries
`concepts`	Validate call stack container types
`source_location` (optional)	Capture compiler-injected source info
`std::span` / `ranges`	Traverse stack traces safely and cleanly

Also consider `consteval` to pre-define invariant error messages at compile time.

19.4.10 Debug Commands in REPL

Provide a few internal commands to access and manage errors:

```
.__last_error()     // show the most recent runtime error
.__error_log()      // dump history of recent errors
.__clear_errors()   // clear the error log
```

These commands improve diagnostics in interactive mode and teaching environments.

19.4.11 Example Errors in Action

Example 1: Out-of-bounds

```
[Runtime Error] OutOfBounds: Index 9 exceeds array length 5
  at access_array (array.csl:11,6)
  Call Stack:
    → access_array
    → run_program
```

Example 2: Use-after-free

```
[Runtime Error] UseAfterFree: Attempt to access deallocated memory block #45
  at read_buffer (main.csl:29,9)
  Call Stack:
    → read_buffer
    → process_io
    → main
```

Example 3: Undefined variable

```
[Runtime Error] UndefinedVariable: Variable 'score' is not declared
  at evaluate() (game.csl:14,3)
```

19.4.12 Test Coverage Strategy

You should build unit tests that assert:

- Error of correct type is thrown
- Message matches expected substrings
- Context (file, line, function) is preserved

Example using a testing framework:

```
TEST_CASE("Division by zero triggers error") {
    REQUIRE_THROWS_MATCHES(
        interpreter.evaluate("int x = 1 / 0;"),
        RuntimeError,
```

```
        Catch::Matchers::MessageContains("divide by zero")
    );
}
```

19.4.13 Benefits and Future Expansion

Immediate Benefits:

- Users trust the interpreter.
- Debugging is faster and less frustrating.
- Cleaner logs and better production safety.
- Seamless error propagation in nested expressions.

Future Additions:

- Serialize error objects to JSON for IDE integration
- Annotate variables and stack frames with metadata
- Colored terminal output for enhanced readability
- Support for breakpoint triggering on critical errors

19.4.14 Conclusion

A robust error handling system is not just a tool for internal correctness—it's a communication contract between your interpreter and its users. It enables safe experimentation, teaches best practices, and prevents subtle bugs from becoming production failures.

By centralizing errors into a well-structured, extensible system using modern C++ constructs, you ensure that your language—despite being inspired by low-level C semantics—offers **high-level safety, professionalism, and trust**.

Chapter 20

Debugging and Development Tools

20.1 AST Visualization for C-Style Constructs

20.1.1 Introduction

The Abstract Syntax Tree (AST) is the heart of a programming language's internal structure. It represents the hierarchical nature of code after parsing and is crucial for semantic analysis, evaluation, code generation, and debugging. In modern interpreter and compiler development, **AST visualization** has emerged as a critical debugging and teaching tool.

In this section, you will build a system to **visualize ASTs for C-style constructs**, leveraging modern C++20/23 features such as structured bindings, formatting utilities, and ranges. This tool is useful for:

- Verifying parser correctness
- Understanding how expressions and statements are structured
- Demonstrating language behavior in education and documentation

- Debugging complex nested logic or function flows

20.1.2 Goals of AST Visualization

The objectives of your AST visualization system are:

Goal	Description
Structural Clarity	Clearly reflect the nesting and hierarchy of AST nodes
C-style Awareness	Support C-style constructs like `if`, `while`, `for`, `return`, etc.
Extensibility	Easily extend to new constructs like `switch`, ternary expressions, etc.
Debugging Readability	Output should be clear enough to guide debugging efforts
Interactive Tooling	Optionally integrate into REPL or external tool via export

20.1.3 AST Node Representation

Before visualization, ensure your AST is structured with a base class and variant node types:

```
enum class ASTNodeKind {
    IntegerLiteral,
    FloatLiteral,
    Variable,
    BinaryExpr,
```

```
    UnaryExpr,
    Assignment,
    IfStatement,
    WhileStatement,
    Block,
    FunctionCall,
    ReturnStatement,
    FunctionDefinition,
    Declaration,
    ForLoop
};

struct ASTNode {
    ASTNodeKind kind;
    SourceLocation location;

    virtual ~ASTNode() = default;
    virtual void visualize(std::ostream& out, int indent = 0) const = 0;
};
```

Then derive specific node types. For example:

```
struct BinaryExprNode : public ASTNode {
    std::string op;
    std::unique_ptr<ASTNode> left;
    std::unique_ptr<ASTNode> right;

    BinaryExprNode(...) {
        kind = ASTNodeKind::BinaryExpr;
    }

    void visualize(std::ostream& out, int indent = 0) const override {
```

```
        print_indent(out, indent);
        out << "BinaryExpr (" << op << ")\n";
        left->visualize(out, indent + 2);
        right->visualize(out, indent + 2);
    }
};
```

Helper for indentation:

```
void print_indent(std::ostream& out, int indent) {
    out << std::string(indent, ' ');
}
```

20.1.4 Visualization of C-Style Constructs

- **If Statement**

```
struct IfStatementNode : public ASTNode {
    std::unique_ptr<ASTNode> condition;
    std::unique_ptr<ASTNode> thenBranch;
    std::unique_ptr<ASTNode> elseBranch;

    void visualize(std::ostream& out, int indent = 0) const override {
        print_indent(out, indent);
        out << "IfStatement\n";
        print_indent(out, indent + 2);
        out << "Condition:\n";
        condition->visualize(out, indent + 4);

        print_indent(out, indent + 2);
        out << "Then:\n";
```

```
        thenBranch->visualize(out, indent + 4);

        if (elseBranch) {
            print_indent(out, indent + 2);
            out << "Else:\n";
            elseBranch->visualize(out, indent + 4);
        }
    }
};
```

- **While Loop**

```
struct WhileStatementNode : public ASTNode {
    std::unique_ptr<ASTNode> condition;
    std::unique_ptr<ASTNode> body;

    void visualize(std::ostream& out, int indent = 0) const override {
        print_indent(out, indent);
        out << "WhileLoop\n";
        print_indent(out, indent + 2);
        out << "Condition:\n";
        condition->visualize(out, indent + 4);

        print_indent(out, indent + 2);
        out << "Body:\n";
        body->visualize(out, indent + 4);
    }
};
```

- **Function Definitions**

```
struct FunctionDefinitionNode : public ASTNode {
    std::string name;
    std::vector<std::string> parameters;
    std::unique_ptr<ASTNode> body;

    void visualize(std::ostream& out, int indent = 0) const override {
        print_indent(out, indent);
        out << "FunctionDefinition: " << name << "\n";
        print_indent(out, indent + 2);
        out << "Parameters: ";
        for (const auto& param : parameters)
            out << param << " ";
        out << "\n";
        body->visualize(out, indent + 2);
    }
};
```

20.1.5 Integration into REPL

Make visualization available via a REPL command:

```
> let tree = parse("if (a > b) { x = 1; } else { x = 2; }");
> visualize(tree);

Output:
IfStatement
  Condition:
    BinaryExpr (>)
      Variable: a
      Variable: b
  Then:
```

```
  Block
    Assignment
      Variable: x
      Integer: 1
Else:
  Block
    Assignment
      Variable: x
      Integer: 2
```

20.1.6 Exporting AST for External Tools

Export ASTs to formats for further tooling:

- **Dot/Graphviz**

```
> let tree = parse("if (a > b) { x = 1; } else { x = 2; }");
> visualize(tree);

Output:
IfStatement
  Condition:
    BinaryExpr (>)
      Variable: a
      Variable: b
  Then:
    Block
      Assignment
        Variable: x
        Integer: 1
  Else:
```

```
    Block
      Assignment
        Variable: x
        Integer: 2
```

This allows graph visualization of complex structures.

- **JSON Format**

```
json to_json(const ASTNode& node);
```

Used for integration with web tools, visual editors, or remote debuggers.

20.1.7 Modern C++ Enhancements

C++20/23 provides elegant tools for writing AST visualizers:

Feature	Use Case
`std::format` (C++20)	Format node outputs cleanly
`ranges::views`	Iterating child nodes of blocks, loops, functions
`concepts`	Enforce interface for ASTNode visualizers
`std::variant`	Represent heterogenous node payloads in simpler nodes
`constexpr` / `consteval`	Generate node name strings at compile-time

20.1.8 Use Cases

Use Case	Benefit
Debugging Parser Output	Spot mismatches between expected and actual parsed structures
Error Localization	Trace errors in malformed or deeply nested expressions
Teaching Tool	Show new learners how code translates to tree structures
Code Generation Validation	Visual inspection before emitting bytecode or machine code
Serialization/Inspecti	Export AST for web-based visualizers or IDE plugins

20.1.9 Testing Visualization Output

Build a suite of tests that parse known code snippets and compare the visualization output against golden files:

```
TEST_CASE("Visualize simple if statement") {
    auto ast = parser.parse("if (a) b = 1;");
    std::stringstream ss;
    ast->visualize(ss);

    REQUIRE(ss.str().find("IfStatement") != std::string::npos);
    REQUIRE(ss.str().find("BinaryExpr") == std::string::npos); // only variable
}
```

Use test frameworks like Catch2 or GoogleTest.

20.1.10 Conclusion

AST visualization transforms the internal workings of your language from a black box into an accessible and powerful learning and debugging interface. By applying clear formatting, support for all major C-style constructs, and modern C++ idioms, you can empower both the language developer and user to understand exactly how code is interpreted.
This feature not only enhances development but also becomes a central part of **education**, **error diagnostics**, and **future IDE integration**.

20.2 Step-by-Step Execution Tracing

20.2.1 Introduction

Step-by-step execution tracing is a cornerstone of modern interpreter debugging infrastructure. It enables developers to pause, inspect, and resume execution while tracking the evaluation of expressions, the resolution of variables, and control flow changes such as jumps and function calls.
In this section, we explore how to design and implement an interactive and programmable **execution tracer** in your C-style language interpreter. It will support stepping through each instruction, optionally observing the call stack, variable states, control branches, and more.
This capability not only assists language developers in verifying interpreter correctness but also empowers language users to debug complex logic in their scripts.

20.2.2 Objectives of Execution Tracing

Objective	Description
Instruction-level stepping	Evaluate one statement or expression at a time
Variable state tracking	View current values of symbols at each step
Call stack visualization	Display function call transitions, entry/exit points
Control flow tracing	Mark transitions in `if`, `while`, `for`, `return`, etc.
Interactive or programmable	Let users step manually or set automatic trace modes
Integration with AST & Error Logs	Link traced steps to AST nodes and any errors encountered

20.2.3 Design Strategy

To support step-by-step tracing, you must:

1. **Instrument the interpreter engine** with trace hooks at key stages (evaluation, function calls, statements, etc.).

2. **Maintain a structured execution context** that can be inspected and serialized.

3. **Control execution flow** using a central trace controller or tracer interface.

4. **Print human-readable or machine-readable output** after each step.

20.2.4 Tracer Interface Design

Define a lightweight, extendable interface:

```
enum class TraceEventKind {
    EnterStatement,
    ExitStatement,
    EvaluateExpression,
    AssignVariable,
    EnterFunction,
    ExitFunction,
    ConditionCheck,
    ReturnValue
};

struct TraceEvent {
    TraceEventKind kind;
    std::string message;
    SourceLocation location;
    std::string functionName;
    std::map<std::string, Value> localVariables;
};

class ExecutionTracer {
public:
    virtual void on_event(const TraceEvent& event) = 0;
    virtual ~ExecutionTracer() = default;
};
```

This allows attaching custom tracers that observe or log the runtime.

20.2.5 Instrumenting Execution Engine

Update the interpreter core to emit trace events. Example in statement execution:

```
void Interpreter::execute(const Statement& stmt) {
    if (tracer) {
        tracer->on_event({
            TraceEventKind::EnterStatement,
            "Executing statement",
            stmt.location,
            currentFunction,
            currentScope.snapshot()
        });
    }

    // Actual statement evaluation
    stmt.evaluate(*this);

    if (tracer) {
        tracer->on_event({
            TraceEventKind::ExitStatement,
            "Finished statement",
            stmt.location,
            currentFunction,
            currentScope.snapshot()
        });
    }
}
```

Similarly, add tracing to function calls, loops, conditionals, and assignments.

20.2.6 Sample Console Tracer Implementation

```
class ConsoleTracer : public ExecutionTracer {
public:
    void on_event(const TraceEvent& event) override {
        std::cout << "[" << to_string(event.kind) << "] "
                  << event.message << " at "
                  << event.location.file << ":" << event.location.line << "\n";

        if (!event.localVariables.empty()) {
            std::cout << "  Locals:\n";
            for (const auto& [name, val] : event.localVariables)
                std::cout << "    " << name << " = " << val.to_string() << "\n";
        }
    }
};
```

Enable tracing via REPL command:

```
.__trace_on()
```

And turn it off with:

```
.__trace_off()
```

20.2.7 Step Mode vs Auto Mode

Support two primary tracing modes:

Mode	Behavior
Step	Waits for user input before continuing after each statement
Auto	Prints execution steps automatically but continues without pause

In step mode:

```
std::string input;
std::cout << ">> Press ENTER to continue, or 'q' to quit tracing\n";
std::getline(std::cin, input);
if (input == "q") { stop_tracing(); }
```

20.2.8 Trace Output Example

Given this code:

```
int main() {
    int a = 10;
    int b = 20;
    if (a < b) {
        a = b;
    }
    return a;
}
```

The trace output:

```
[EnterStatement] Executing declaration of 'a' at main.csl:2
  Locals:
    a = 10
```

```
[EnterStatement] Executing declaration of 'b' at main.csl:3
  Locals:
    a = 10
    b = 20

[ConditionCheck] Checking condition 'a < b' at main.csl:4
  Result: true

[EnterStatement] Executing assignment 'a = b' at main.csl:5
  Locals:
    a = 10
    b = 20

[AssignVariable] Set 'a' = 20

[ReturnValue] Returning 20 from function 'main'
```

This allows debugging both variable changes and control flow logic.

20.2.9 Trace Control API (Advanced Use)

To support IDEs or GUI-based environments, expose a structured tracing API:

```
struct ExecutionSnapshot {
    std::string function;
    SourceLocation location;
    std::map<std::string, Value> variables;
    std::string currentStatement;
    std::vector<std::string> callStack;
};

class TraceController {
```

```
public:
    ExecutionSnapshot current() const;
    void step();
    void continue_execution();
    void pause();
};
```

This enables:

- External debugger UIs
- Web-based visualization
- Recording execution history

20.2.10 Modern C++20/23 Enhancements

Feature	Role in Tracing System
`std::format`	Clear formatting for trace logs
`std::source_location` (C++20)	Automatically track source info (line, file, function)
`coroutines` (optional)	Model step-resumable execution engine
`concepts`	Ensure tracer interface conformance
`std::ranges`	Manage local variable iteration

These features simplify tracing logic while keeping it type-safe and expressive.

20.2.11 Test Coverage and Validation

Write test cases that ensure the trace log contains expected steps:

```
TEST_CASE("Trace assignment and return") {
    TraceLogger logger;
    interpreter.set_tracer(&logger);
    interpreter.evaluate("int x = 5; return x;");

    auto log = logger.get_events();
    REQUIRE(log.size() == 3);
    REQUIRE(log[0].kind == TraceEventKind::EnterStatement);
    REQUIRE(log[2].kind == TraceEventKind::ReturnValue);
}
```

20.2.12 Conclusion

Step-by-step execution tracing brings visibility into every operation your interpreter performs. By leveraging event-based tracing, structured snapshots, and modern C++ facilities, you provide a highly professional and educational debugging experience. Such tooling not only enhances correctness and reliability of the interpreter but also lays the foundation for **IDE integration**, **unit test generation**, and **AI-assisted debugging**.

20.3 Performance Profiling Integration

20.3.1 Introduction

Performance profiling is a crucial tool for both developers of the interpreter itself and users of the language who write performance-sensitive code. Profiling enables you to

collect metrics about runtime behavior—such as function execution time, memory usage, instruction count, and call frequency—which are essential for optimization and debugging performance bottlenecks.
This section presents a practical and modern approach to **integrating profiling into a C-style interpreted language**, using **C++20/23 features** such as `std::chrono`, structured bindings, and modern data structures for low-overhead instrumentation.

20.3.2 Objectives of Performance Profiling

Objective	Description
Function-level timing	Measure how long each function takes to execute
Call frequency counting	Track how often specific functions, statements, or expressions are invoked
Memory access tracking	Optionally log dynamic allocations, array accesses, etc.
Profiling granularity	Support configurable levels (coarse: per-function, fine: per-statement)
Result visualization	Output profiling results in readable and structured formats
Low overhead mode	Ensure profiling does not degrade interpreter performance significantly

20.3.3 Instrumentation Architecture

Integrate a **profiler interface** into the interpreter core:

```
enum class ProfileEventKind {
    FunctionEntry,
    FunctionExit,
    StatementExecuted
};

struct ProfileEvent {
    ProfileEventKind kind;
    std::string name;
    std::chrono::steady_clock::time_point timestamp;
    size_t memory_used = 0;
};

class Profiler {
public:
    virtual void on_profile_event(const ProfileEvent& event) = 0;
    virtual void report(std::ostream& out) const = 0;
    virtual void reset() = 0;
    virtual ~Profiler() = default;
};
```

Attach it to the interpreter:

```
interpreter.set_profiler(std::make_unique<FunctionProfiler>());
```

20.3.4 Measuring Execution Time

Use modern C++ time utilities from `<chrono>` to collect timing data:

```
auto start = std::chrono::steady_clock::now();
// ... execute code block ...
```

```
auto end = std::chrono::steady_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
```

Record this into a function performance table:

```
struct FunctionProfile {
    size_t call_count = 0;
    uint64_t total_microseconds = 0;
};

std::unordered_map<std::string, FunctionProfile> profile_table;
```

Update profile information:

```
void FunctionProfiler::on_profile_event(const ProfileEvent& event) {
    if (event.kind == ProfileEventKind::FunctionExit) {
        auto& data = profile_table[event.name];
        data.call_count++;
        data.total_microseconds +=
        ↪  std::chrono::duration_cast<std::chrono::microseconds>(
            event.timestamp - last_entry_time[event.name]
        ).count();
    } else if (event.kind == ProfileEventKind::FunctionEntry) {
        last_entry_time[event.name] = event.timestamp;
    }
}
```

20.3.5 Profiling Control Flow and Statements

Support optional fine-grained profiling:

- Add a flag `--profile-statements` or REPL command `.profile on`

- Wrap all `Statement::execute()` calls with timer logic
- Track statement types (assignments, conditionals, etc.)

Optional output:

```
[Profile] AssignmentStatement: avg = 4us, count = 50
[Profile] IfStatement: avg = 12us, count = 10
```

This granularity helps locate performance bottlenecks in interpreted scripts.

20.3.6 Memory Access Profiling (Optional)

If your interpreter supports dynamic memory management (e.g. dynamic arrays, structs), add hooks for tracking memory usage:

```
struct MemoryProfiler {
    size_t total_allocated = 0;
    size_t total_deallocated = 0;

    void on_alloc(size_t size) { total_allocated += size; }
    void on_dealloc(size_t size) { total_deallocated += size; }

    size_t current_usage() const {
        return total_allocated - total_deallocated;
    }
};
```

Hook into array allocation, function local frame creation, and heap-like data structures.

20.3.7 Report Generation and Visualization

At the end of script execution or on REPL command `.profile_report`, output a summary:

```
--- Profiling Report ---
Function: factorial
  Calls: 100
  Total time: 1,304 µs
  Avg time: 13.04 µs

Function: computeSum
  Calls: 10
  Total time: 822 µs
  Avg time: 82.2 µs

Memory:
  Peak allocated: 32 KB
  Current usage: 8 KB
```

For graphical UIs or web-based tools, support exporting to JSON:

```
{
  "functions": {
    "factorial": { "calls": 100, "time_us": 1304 },
    "computeSum": { "calls": 10, "time_us": 822 }
  },
  "memory": { "peak": 32768, "current": 8192 }
}
```

20.3.8 Leveraging Modern C++ Features

C++20/23 Feature	Role in Profiler
`std::chrono::steady_c`	Accurate cross-platform time measurement
`std::format`	Elegant, readable profiler reports without `std::stringstream`
`source_location`	Tag source locations automatically in profiling logs
`std::unordered_map`	Fast aggregation of profiling counters
`constexpr` / `consteval`	Used to define statically known function/statement names
`std::ranges`	Sort or filter profiling data (e.g. top N slowest functions)

These modern features reduce boilerplate and enhance maintainability.

20.3.9 Integration with Debugging and Tracing

A robust language tooling ecosystem benefits from unified introspection. The profiler should be able to:

- **Cooperate with the tracer**: i.e., trace logs can indicate how much time passed per function.
- **Highlight expensive operations**: Use warning messages or annotations during trace or AST inspection.
- **Hook into error diagnostics**: Show performance impact of functions involved in an error.

Example: "Function `computePath` that caused exception took 150ms and was called 15 times."

20.3.10 Test Cases and Validation

Unit test the profiler:

```
TEST_CASE("Profiler measures function time correctly") {
    Profiler profiler;
    interpreter.set_profiler(&profiler);

    interpreter.evaluate("int f() { int a = 0; return a; } f(); f();");
    auto report = profiler.get_report();

    REQUIRE(report["f"].call_count == 2);
    REQUIRE(report["f"].total_microseconds > 0);
}
```

Benchmark real-world scripts to validate profiler output under realistic load.

20.3.11 Conclusion

Integrating a performance profiler into your C-style interpreter transforms it from a functional prototype into a **production-quality system**. It provides visibility into runtime behavior, guides optimization efforts, and builds trust with developers using the language for real-world tasks.

By combining execution timing, memory tracking, and structured reporting—backed by modern C++ constructs—you empower users to write efficient, optimized programs and allow your interpreter to evolve with performance awareness as a core value.

20.4 Milestone — Complete Debugging Toolkit

20.4.1 Introduction

Achieving a complete debugging toolkit is a crucial milestone in building a professional-grade interpreter. A language is not truly usable until developers can **inspect, trace, profile, and correct their programs** efficiently. This section defines the architectural and implementation milestones for a complete debugging toolkit that supports C-style semantics, aligned with modern programming expectations.
By utilizing C++20/23 enhancements, we can build a toolkit that is interactive, structured, scriptable, and extensible.

20.4.2 Definition of "Complete Debugging Toolkit"

To qualify as complete, your debugging toolkit should support the following core components:

Component	Description
Trace Engine	Logs runtime steps like function entry, statements, and expression evaluation
Step-by-Step Mode	Allows controlled execution for detailed program inspection
Breakpoints	User-defined stop-points in code, triggered by line number or condition
Watchpoints	Triggered when a specific variable changes or reaches a target value
Call Stack Viewer	Displays current and nested function calls with local variable scopes

Component	Description
Value Inspector	Queries current values across different environments and scopes
Profiler	Gathers performance metrics such as timing, memory usage, and call frequency
Error Tracker	Captures and categorizes runtime or semantic-level errors
REPL Debug Commands	Provides live debugging interaction through a command interface

20.4.3 Implementation Strategy

Break the toolkit into modular services or components, injected into the interpreter context:

```
struct DebuggingToolkit {
    std::unique_ptr<ExecutionTracer> tracer;
    std::unique_ptr<BreakpointManager> breakpoints;
    std::unique_ptr<Profiler> profiler;
    std::unique_ptr<ErrorReporter> errors;
    std::unique_ptr<CallStackInspector> stackViewer;
    std::unique_ptr<MemoryInspector> memoryView;
};
```

These are accessible from both REPL and internal diagnostics:

```
interpreter.debugger().tracer->step_over();
interpreter.debugger().breakpoints->add("main.csl", 12);
```

20.4.4 Step Execution and Breakpoint Engine

Use an execution loop controlled by trace flags and breakpoints:

```
bool Interpreter::run_statement(const Statement& stmt) {
    if (debugger.breakpoints->is_triggered(stmt.location)) {
        debugger.tracer->pause_execution("Breakpoint hit at line",
        ↪  stmt.location.line);
    }

    debugger.tracer->before_statement(stmt);
    stmt.evaluate(*this);
    debugger.tracer->after_statement(stmt);
    return true;
}
```

Breakpoints can be line-based or condition-based:

```
debugger.breakpoints->add("main.csl", 15);
debugger.breakpoints->add_conditional("main.csl", 20, "counter > 100");
```

20.4.5 Watchpoints and Variable Monitoring

Use scoped variable lookups to monitor watched variables:

```
debugger.watchpoints->watch("x", [](const Value& v) {
    return v.as_int() == 42;
});
```

Trigger a pause or log when condition is met:

```
if (debugger.watchpoints->check(variableName, currentValue)) {
    debugger.tracer->pause_execution("Watchpoint triggered on 'x'");
}
```

20.4.6 Value and Scope Inspector

Allow inspecting scopes using REPL commands:

```
.inspect global
.inspect stack
.inspect locals
```

Display variable names, types, and current values:

```
--- Local Variables ---
x (int) = 10
y (float) = 2.5
name (string) = "Alpha"
```

Provide programmatic access:

```
auto vars = interpreter.current_scope().all_variables();
for (auto& [name, value] : vars) {
    std::cout << name << " = " << value.to_string() << "\n";
}
```

20.4.7 Call Stack View and Navigation

Store function call frames on a stack:

```
struct CallFrame {
    std::string functionName;
    SourceLocation entryPoint;
    std::map<std::string, Value> locals;
};

std::vector<CallFrame> call_stack;
```

Display as:

```
--- Call Stack ---
#3: compute() at main.csl:22
#2: process() at utils.csl:15
#1: main() at main.csl:5
```

Support REPL commands to query:

```
.stack
.stack inspect 2
```

20.4.8 Profiling Summary Integration

At any breakpoint or pause, show current profiling snapshot:

```
--- Profiling ---
Function: compute() - 15 calls, total 1012 µs
Function: loadData() - 3 calls, total 455 µs
```

Integrate with `.profile on`, `.profile_report`, or export to JSON:

```
.debug export profile.json
```

20.4.9 Enhanced Error Reporting and Live Fixing

Enable live diagnostics in the REPL:

- Show error stack trace
- Jump to failing statement
- Re-run fixed expression

Example output:

```
Runtime Error: Division by zero at main.csl:10
Function: divide(a, b)
Call stack:
  main() → divide(10, 0)
```

Command to inspect and retry:

```
.fix divide(10, 1)
```

20.4.10 Modern C++ Tools Used

Feature	Usage
`std::source_location`	Track precise source line for logs, breakpoints, and stack views

Feature	Usage
`std::format`	Human-readable, structured logging output
`std::chrono::steady_c`	Accurate profiling for timing information
`std::variant`, `std::any`	Flexible value containers for variable inspection
`std::ranges`, `views`	Iterate filtered or sorted profiler reports
`concepts` and `requires`	Interface contracts for debuggers and plugins

These tools result in more readable, concise, and correct implementations.

20.4.11 Final Milestone Checklist

Feature	Status
Execution step-by-step engine	Completed
Breakpoints and watchpoints	Completed
REPL-based introspection	Completed
Stack and variable viewer	Completed
Performance profiler integration	Completed
Error tracking with trace	Completed
Exportable debug reports	Completed

20.4.12 Conclusion

Reaching this milestone transforms your C-style interpreted language into a **developer-friendly platform** with full introspection and diagnostic capability. This empowers both learners and professionals to **trust, explore, debug, and optimize** their code in a clear and structured environment.

The complete debugging toolkit you've built ensures that your language is no longer a conceptual prototype but a usable system suitable for production scripting, education, testing, and complex systems development.

In future expansions, you may consider:

- Visual debuggers over WebAssembly
- Remote debugging interfaces via sockets
- Machine learning-based runtime optimization hints

But for now, this milestone marks a major engineering accomplishment: a fully equipped C-style interpreted language, built using modern C++ practices and production-level tooling.

Chapter 21

File Execution and Script Support

21.1 Executing `.lang` Files from Command Line

21.1.1 Introduction

One of the final steps in transforming your interpreter from a development tool into a usable language is enabling execution of full scripts via the command line. This feature allows users to run `.lang` source files directly, much like C files compiled and executed, but through interpretation. The interpreter should support a minimal yet flexible command-line interface (CLI), similar in behavior to scripting languages such as Python, Lua, or Ruby.

This section details how to implement command-line execution of `.lang` files, how to parse and process file input, how to attach arguments and environment contexts, and how to make use of modern C++20/23 features to enhance performance and extensibility.

21.1.2 CLI Design Principles

To make your interpreter practical and user-friendly, your CLI design should support:

- **Direct script execution:**
  `myLang script.lang`
- **Pass arguments to scripts:**
  `myLang script.lang arg1 arg2`
- **Enable optional flags:**
  - `--trace`, `--profile`, `--version`, `--help`, etc.
- **Run in interactive (REPL) mode if no file is provided.**

The interpreter's entry point should interpret command-line input and switch modes accordingly.

21.1.3 Parsing Command Line Arguments

Modern C++ provides standard
facilities for parsing arguments via `std::vector<std::string>` and optionally using `std::span` for slicing.

```
int main(int argc, char* argv[]) {
    std::vector<std::string> args(argv, argv + argc);

    if (args.size() < 2) {
        launch_repl();
        return 0;
    }
```

```
    if (args[1] == "--help") {
        print_help();
        return 0;
    }

    if (ends_with(args[1], ".lang")) {
        execute_lang_file(args[1], {args.begin() + 2, args.end()});
        return 0;
    }

    std::cerr << "Unrecognized command or file.\n";
    return 1;
}
```

Helper to detect file extensions:

```
bool ends_with(const std::string& str, const std::string& suffix) {
    return str.size() >= suffix.size() &&
           str.compare(str.size() - suffix.size(), suffix.size(), suffix) == 0;
}
```

21.1.4 File Loading and Parsing

To execute a `.lang` file, your interpreter must:

1. Load the file contents into memory.
2. Lex and parse the file into an abstract syntax tree (AST).
3. Evaluate the AST in the global environment.

```
void execute_lang_file(const std::string& path, const std::vector<std::string>& args)
↪ {
    std::ifstream file(path);
    if (!file) {
        std::cerr << "Cannot open file: " << path << "\n";
        return;
    }

    std::string source((std::istreambuf_iterator<char>(file)),
                       std::istreambuf_iterator<char>());

    Interpreter interpreter;
    interpreter.set_script_arguments(args); // optional

    try {
        auto ast = Parser(Lexer(source)).parse_program();
        interpreter.evaluate(ast);
    } catch (const InterpreterError& err) {
        std::cerr << "Runtime error: " << err.what() << "\n";
    }
}
```

You can optionally expose `args` within the language as a global array `argv` and `argc` for script-side processing.

21.1.5 Error Handling for Script Execution

Ensure graceful handling of:

- **Missing files**
- **Malformed syntax**

- **Uncaught runtime exceptions**
- **Permission issues or invalid characters**

Use modern exception handling and rich error messages:

```
try {
    execute_lang_file(path, args);
} catch (const std::exception& e) {
    std::cerr << "Fatal error: " << e.what() << "\n";
    return 1;
}
```

In C++23, use `std::expected` (or your own Result-like struct) to manage optional errors without throwing.

21.1.6 Flags and Options for Script Control

Support standard debugging and inspection flags:

Flag	Functionality
`--trace`	Enables statement-level execution logging
`--profile`	Activates the profiler module
`--verbose`	Prints token stream and AST
`--no-color`	Disables colored console output

Example usage:

```
myLang main.lang --trace --profile
```

Parsing can be done using a minimal internal parser or external argument library if needed.

21.1.7 Integration with REPL and Standard I/O

If no file is passed, fall back to the REPL automatically:

```
if (argc == 1) {
    launch_repl();
}
```

Additionally, allow redirecting standard input or reading from a pipeline:

```
cat code.lang | myLang
```

This requires reading from `std::cin` when no file is detected and input stream is not terminal.

21.1.8 Modern C++ Enhancements

C++ Feature	Application in CLI Execution
`std::span`	For slicing arguments vector
`std::format` (C++20)	Format rich command-line and error messages
`std::filesystem`	Validate paths, file existence, and extensions

C++ Feature	Application in CLI Execution
`std::expected` (C++23)	Graceful parsing and execution results without throws
`std::ranges`	Filter or transform flags and arguments
`std::source_locatic`	Enhanced error logs with source info

21.1.9 Testing and Validation

Write CLI unit and integration tests using frameworks like **Catch2** or **doctest**:

```
TEST_CASE("Script file execution") {
    std::string file = "examples/hello.lang";
    REQUIRE(interpreter.execute_file(file) == 0);
}
```

Validate scenarios:

- Empty script
- Syntax errors
- File with command-line arguments
- Profiling enabled

21.1.10 Sample Execution Session

Assuming the file `hello.lang`:

```
print("Hello, " + argv[1]);
```

Run:

```
myLang hello.lang World
```

Output:

```
Hello, World
```

21.1.11 Conclusion

Enabling execution of `.lang` scripts from the command line is a critical step toward practical language deployment. It bridges development tools and real-world scripting use. Using modern C++20/23 ensures your CLI is **robust, extensible, and maintainable**, supporting structured argument parsing, profiling, tracing, and REPL fallback.
With this foundation in place, the next steps can include **file-based module imports, packaging**, and **interpreter embedding inside larger applications**.

21.2 Basic Module/Include System

21.2.1 Introduction

A core feature of any serious programming language is the ability to **organize and reuse code across multiple files**. This is achieved through an **include system** or a **module system**. In traditional C-style languages, this typically means using

preprocessor-style includes (e.g., `#include "file.h"`). In modern languages, modules aim to replace raw includes with structured imports.
In this section, we design a **basic include/module system** tailored for a C-style interpreted language. The system should support:

- Importing source files at parse or runtime
- Preventing duplicate inclusion
- Isolated symbol management
- Relative and absolute path resolution
- Forward extensibility toward a full module system

The implementation will use **modern C++20/23 features** for path handling, context tracking, and caching.

21.2.2 Design Philosophy

This include system should feel familiar to C programmers but behave more safely and cleanly:

- Includes will work at parse time, not preprocess time
- Inclusion happens **only once** per module per run (like `#pragma once`)
- Imported files execute in a **separate environment scope**, optionally exported to global or caller scope
- Future upgrades may support named exports, visibility control, and native binary modules

21.2.3 Syntax Proposal

Basic syntax mirrors traditional includes or simple modules:

```
include "math.lang";
include "./utils/io.lang";
```

Or a modular variant:

```
import "math.lang";
```

The keyword `include` or `import` is a reserved statement in the parser, triggering loading and evaluation of another file.

21.2.4 Parser and AST Extension

Add `IncludeStatement` to your AST structure:

```
struct IncludeStatement : public Statement {
    std::string path;
    SourceLocation location;
};
```

Update your parser:

```
std::unique_ptr<Statement> Parser::parse_include() {
    consume(TokenType::Include); // or TokenType::Import
    std::string path = parse_string_literal();
    return std::make_unique<IncludeStatement>(IncludeStatement{path,
    ↪  current_location()});
}
```

21.2.5 Interpreter Execution Logic

Upon encountering an `IncludeStatement`, the interpreter must:

1. Resolve the file path (absolute or relative)
2. Check if the file has already been included
3. If not, load and parse the file
4. Execute the file in a separate scope (optional)

```
void Interpreter::execute_include(const IncludeStatement& stmt) {
    std::filesystem::path resolved = resolve_path(stmt.path, stmt.location);

    if (included_files_.contains(resolved)) return; // prevent re-execution

    std::string source = read_file_contents(resolved);
    auto ast = Parser(Lexer(source)).parse_program();

    included_files_.insert(resolved);

    ScopedEnvironment module_scope(global_environment());
    Interpreter sub_interpreter(module_scope);
    sub_interpreter.execute(ast);
}
```

Where `included_files_` is a `std::unordered_set<std::filesystem::path>` tracking loaded modules.

21.2.6 Scope Management: Global vs Module

To prevent polluting the global namespace, included files can run in a scoped environment:

```
class ScopedEnvironment {
    Environment* parent_;
    std::unordered_map<std::string, Value> locals_;
};
```

Use this to allow namespacing or encapsulation. In future versions, support for `export` syntax could selectively expose symbols:

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

For now, a simple model could just copy all variables from the module scope into the parent:

```
void merge_scope(ScopedEnvironment& from, Environment& to) {
    for (auto& [key, value] : from.locals()) {
        to.define(key, value);
    }
}
```

21.2.7 Path Resolution

Use `std::filesystem` (C++17+, enhanced in C++20) to locate and normalize file paths:

```
std::filesystem::path Interpreter::resolve_path(const std::string& path, const
↪  SourceLocation& loc) {
    auto current_file_path = loc.file_path;
    auto base_dir = std::filesystem::path(current_file_path).parent_path();
    auto full_path = std::filesystem::canonical(base_dir / path);
    return full_path;
}
```

This enables support for:

- Relative paths
- Platform portability
- Preventing circular includes

21.2.8 Preventing Redundant Inclusion

Track includes using canonical paths to avoid duplicates:

```
std::unordered_set<std::filesystem::path> included_files_;
```

Insert only canonical paths:

```
auto full_path = std::filesystem::canonical(path);
if (included_files_.contains(full_path)) return;
included_files_.insert(full_path);
```

This handles both:

- Repeated includes of the same file
- Includes from different relative paths resolving to same target

21.2.9 Command-Line and Module Search Paths

Support custom module search paths:

```
myLang main.lang --mod-path=libs/
```

Scan `libs/` for modules, and allow fallback to default paths:

```
std::vector<std::filesystem::path> module_paths = { ".", "./libs", user_path };
```

Attempt to resolve includes in each path.

21.2.10 Modern C++ Enhancements

C++ Feature	Role in Include System
`std::filesystem`	Canonical path resolution, cross-platform file operations
`std::unordered_set`	Deduplication of includes using fast hashing
`std::format` (C++20)	Clean error and include logs
`std::source_location`	Better tracking for error reporting in imported files
`std::string_view`	Lightweight string handling during parsing
`std::optional` / `expected` (C++23)	Graceful file loading with fallback handling

21.2.11 Example Usage

math.lang

```
int square(int x) { return x * x; }
```

main.lang

```
include "math.lang";
print(square(5));
```

Run:

```
myLang main.lang
```

Output:

```
25
```

21.2.12 Conclusion

A basic include/module system elevates your interpreter from a REPL-only tool into a structured language capable of **real-world scripting**, **library development**, and **code modularization**. By integrating path resolution, scope isolation, and file caching using modern C++ techniques, your language supports safe and reliable multi-file development.
In future iterations, this foundation can evolve into:

- Named module systems
- Dependency graphs
- Import guards and package managers
- Native (compiled) module support

But even in this basic form, the system enables scalable development and long-term project architecture, a key milestone in the transition from prototype to full language.

21.3 Command-Line Interface Design

21.3.1 Introduction

A powerful and user-friendly **Command-Line Interface (CLI)** is critical for scripting languages, developer workflows, automation pipelines, and integration with system tools. In this section, we design and implement a modern, extensible CLI for your C-style interpreted language, enabling developers to run scripts, pass arguments, activate debugging tools, enable profiling, or inspect environment configurations.
We will use features introduced in C++20 and C++23 to make this CLI **type-safe, ergonomic, and extensible**, following the best practices in toolchain design.

21.3.2 CLI Roles and Requirements

Your CLI should cover the following:

- Run `.lang` script files
- Fall back to interactive REPL if no file is provided
- Pass arguments to scripts via `argv[]`
- Support flags like:
  - `--help`, `--version`, `--trace`, `--profile`, `--debug`
- Optional environment settings (e.g., `--mod-path`, `--no-color`)
- Allow piping input and redirecting output
- Return proper exit codes

21.3.3 C++20/23-Oriented CLI Architecture

A clean CLI design separates these components:

Component	Responsibility
Argument parser	Parses raw `argc`, `argv` into structured data
Command dispatcher	Maps parsed commands to interpreter actions
Execution controller	Orchestrates REPL vs script vs error fallback
Environment injector	Sets environment flags for profiling/debugging

We'll use `std::vector<std::string>` (or `std::span` from C++20) for argument access, and optional struct-based flags with C++23 `std::expected` for parsing.

21.3.4 CLI Syntax Design

Define consistent command usage:

```
Usage: myLang [options] [script.lang] [args...]

Options:
  --help            Show usage information
  --version         Show interpreter version
  --trace           Enable tracing of execution
  --profile         Enable basic profiling
  --debug           Enable internal AST dumping
  --mod-path PATH   Add module search path
  --no-color        Disable color output
```

Support positional and optional arguments:

```
myLang hello.lang John
```

Or:

```
myLang --trace --mod-path ./lib math.lang
```

21.3.5 CLI Argument Parsing in Modern C++

Here's a lightweight, idiomatic approach using structured configuration.

```
struct CLIOptions {
    bool trace = false;
    bool profile = false;
    bool debug = false;
    bool show_help = false;
    bool show_version = false;
    std::string script_path;
    std::vector<std::string> script_args;
    std::vector<std::filesystem::path> module_paths;
};
```

Argument Parsing Logic

```
CLIOptions parse_arguments(int argc, char* argv[]) {
    CLIOptions opts;
    std::vector<std::string> args(argv + 1, argv + argc);

    auto it = args.begin();
    while (it != args.end()) {
        if (*it == "--trace") opts.trace = true;
```

```
        else if (*it == "--profile") opts.profile = true;
        else if (*it == "--debug") opts.debug = true;
        else if (*it == "--help") opts.show_help = true;
        else if (*it == "--version") opts.show_version = true;
        else if (*it == "--mod-path") {
            ++it;
            if (it != args.end()) {
                opts.module_paths.push_back(*it);
            }
        } else if (opts.script_path.empty() && it->ends_with(".lang")) {
            opts.script_path = *it;
        } else {
            opts.script_args.push_back(*it);
        }
        ++it;
    }

    return opts;
}
```

C++20 features like `std::string::ends_with()` improve clarity here.

21.3.6 Dispatch Logic

```
int main(int argc, char* argv[]) {
    CLIOptions options = parse_arguments(argc, argv);

    if (options.show_help) {
        print_help();
        return 0;
    }
```

```
    if (options.show_version) {
        std::cout << "ForgeLang Interpreter v0.1\n";
        return 0;
    }

    if (options.script_path.empty()) {
        launch_repl(options);
    } else {
        execute_script(options);
    }

    return 0;
}
```

This flow ensures graceful fallback, intuitive behavior, and flags that can be passed in any order.

21.3.7 Environment Injection

Interpreter flags (like profiling or tracing) must propagate to the evaluator engine:

```
InterpreterConfig config;
config.enable_tracing = options.trace;
config.enable_profiling = options.profile;
config.module_paths = options.module_paths;
```

These options feed into the interpreter's evaluation engine, which enables or disables runtime features like logging, AST printing, and profiling hooks.

21.3.8 Error and Exit Code Handling

Exit with standard exit codes:

Code	Meaning
0	Success
1	General error
2	Script file not found
3	Syntax error in script
4	Runtime error during eval

Use `std::expected` or `std::variant` to carry status in future refactoring:

```
std::expected<void, InterpreterError> execute_script(const CLIOptions& opts);
```

21.3.9 Logging and Output

C++20 introduced `std::format` which is ideal for printing structured messages:

```
std::cout << std::format("Loading script: {}\n", opts.script_path);
```

To support `--no-color`, wrap color output using a boolean toggle:

```
if (use_color) std::cout << "\033[32mSuccess\033[0m\n";
```

21.3.10 Testing and Validation

Automated test cases should check CLI behavior:

```
TEST_CASE("Help option prints usage") {
    auto opts = parse_arguments({"myLang", "--help"});
    REQUIRE(opts.show_help == true);
}

TEST_CASE("Positional args parsed correctly") {
    auto opts = parse_arguments({"myLang", "test.lang", "arg1", "arg2"});
    REQUIRE(opts.script_path == "test.lang");
    REQUIRE(opts.script_args == std::vector{"arg1", "arg2"});
}
```

Use `doctest`, `Catch2`, or `GoogleTest` for unit tests.

21.3.11 Extensibility for Future Features

Future CLI extensions could include:

- `--benchmark` for timed runs
- `--import` for one-shot import testing
- `--dump-ast` to serialize the AST to file
- `--watch` to enable hot-reload during development

With a structured parser, all such extensions are manageable via flags.

21.3.12 Conclusion

A modern, structured CLI empowers your interpreter to be a **first-class tool in developer workflows**. By combining the clarity of flag-based parsing with the power of modern C++ (ranges, `std::filesystem`, `std::expected`, `std::format`), the CLI

becomes maintainable, testable, and adaptable. It connects your language to the OS environment and empowers users to integrate it into scripts, editors, and tools seamlessly.
This CLI will evolve into a full tooling front-end, supporting package management, language server protocols, and embedding. But its foundation must begin here: clear, robust, and idiomatic to modern C++.

21.4 Milestone — Standalone Interpreter for Our C-Style Language

21.4.1 Overview

Reaching this milestone—building a **standalone interpreter**—represents a critical achievement in the development of your C-style programming language. It marks the transition from an experimental REPL or internal library to a production-level **command-line executable** capable of parsing, evaluating, and executing `.lang` scripts independently of any IDE or development environment.
In this section, we consolidate all previous interpreter components and integrate them into a single deployable binary that fulfills several essential roles:

- Script execution from files
- Interactive REPL fallback
- Command-line option handling
- Error reporting and debugging
- Compatibility with external tools and OS scripting environments

This interpreter is written using **Modern C++20/23 idioms** to ensure strong typing, modularity, and maintainability.

21.4.2 Interpreter Entry Point: `main`

At the center of your standalone interpreter is the `main()` function. It acts as the **controller**, delegating responsibilities to the CLI parser, the execution engine, and the REPL system:

```
int main(int argc, char* argv[]) {
    auto options = parse_arguments(argc, argv);

    if (options.show_help) {
        print_help();
        return 0;
    }

    if (options.show_version) {
        std::cout << "ForgeLang v0.1\n";
        return 0;
    }

    try {
        if (!options.script_path.empty()) {
            return execute_script_file(options);
        } else {
            return launch_repl(options);
        }
    } catch (const InterpreterException& ex) {
        std::cerr << "Error: " << ex.what() << "\n";
        return 1;
    }
```

```
}
```

This pattern separates concerns cleanly and enables robust error trapping.

21.4.3 File Execution Integration

To support script execution via command line:

```
forgec script.lang arg1 arg2
```

You implement a function like:

```
int execute_script_file(const CLIOptions& options) {
    auto source = read_file(options.script_path);
    auto lexer = Lexer(source);
    auto parser = Parser(lexer);
    auto ast = parser.parse();

    Interpreter interpreter;
    interpreter.set_args(options.script_args);
    interpreter.set_module_paths(options.module_paths);
    interpreter.enable_tracing(options.trace);
    interpreter.enable_profiling(options.profile);

    interpreter.execute(ast);

    return 0;
}
```

This design reflects modern software layering: parsing, evaluation, and runtime environment are modular and testable.

21.4.4 REPL Fallback

If no `.lang` file is specified, the interpreter launches an interactive shell:

```
int launch_repl(const CLIOptions& options) {
    std::cout << "ForgeLang REPL - Version 0.1\n";

    Interpreter interpreter;
    while (true) {
        std::cout << ">> ";
        std::string line;
        if (!std::getline(std::cin, line)) break;

        try {
            auto ast = Parser(Lexer(line)).parse_expression();
            auto result = interpreter.evaluate(ast);
            std::cout << "= " << result.to_string() << "\n";
        } catch (const InterpreterException& ex) {
            std::cerr << "Error: " << ex.what() << "\n";
        }
    }

    return 0;
}
```

REPL support is essential for beginners, debugging, and test-driven development workflows.

21.4.5 Integration of All Language Subsystems

By this point in the book, your interpreter supports:

Feature	Implemented In
Type system	Chapter 10: `int`, `float`, `bool`, etc.
Expressions and operators	Chapter 12: precedence, evaluation
Scoping and environment	Chapter 11: block/lexical scoping
Statement execution	Chapter 14: `if`, `while`, `for`, etc.
Functions and recursion	Chapter 15: call stack, parameters
Module system	Chapter 21: includes/imports
CLI and script arguments	Chapter 21: this chapter

All of these must be brought together in a coherent, resilient execution model driven by the CLI and the file-based runner.

21.4.6 Deployment Readiness

To make the interpreter ready for real-world usage:

1. **Compilation as Static Binary**
   Use modern CMake and LTO settings for optimized builds:

```
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_INTERPROCEDURAL_OPTIMIZATION ON)
add_executable(forgec main.cpp ...)
```

2. **Executable Output**
   Build the interpreter as `forgec` (Forge Compiler or Forge Console):

```
g++ -std=c++23 -O3 -o forgec *.cpp
```

3. **Cross-Platform Path Handling**
   Rely on `std::filesystem` to support Linux, macOS, and Windows:

```
std::filesystem::path path = std::filesystem::canonical(script_path);
```

4. **Portable Execution**
   Ensure no hardcoded dependencies; scripts should execute from any directory:

```
./forgec scripts/hello.lang
```

21.4.7 Exit Codes and External Tooling

To support system integration, return appropriate codes:

- `0`: success
- `1`: parse error
- `2`: runtime error
- `3`: missing file
- `4`: invalid CLI usage

Also, allow standard output and error streams to be redirected for scripting:

```
./forgec test.lang > result.txt 2> error.txt
```

21.4.8 Optional: Static Embedding or Packaging

To go further, you may add:

- A `--bundle` flag that creates a single file combining the interpreter and source
- A packaging tool to produce `.forge` bundles with metadata, main file, and resources
- Embedding scripts directly in the binary via `constexpr` strings for demo apps

21.4.9 Summary: What This Milestone Unlocks

By completing this standalone interpreter milestone, your language becomes a:

- Deployable tool for scripting, automation, and education
- Minimal shell interpreter for use in CI pipelines
- Testbed for rapid language evolution and experimentation
- Platform for building libraries, modules, and long-term codebases

This point in the book marks the transition from a learning tool into an actual **language platform**. The design, thanks to Modern C++ features and a modular interpreter core, is extensible, testable, and ready for the next chapters—optimization, packaging, and possibly compiling to bytecode or native code.

Part VIII

Optimization and Advanced Topics

Chapter 22

Performance Optimization

22.1 Optimizing C-style expression evaluation

22.1.0.1 1.0 Introduction

Expression evaluation forms the computational backbone of any programming language. In C-style languages, expressions are dense, nested, and performance-critical. Once your interpreter reaches functional maturity, expression performance becomes a major concern—especially when targeting real-time usage, scripting within tight loops, or compute-heavy applications like simulation, graphics, or math libraries.
This section provides a focused strategy to **optimize C-style expression evaluation** using **modern C++20/23 capabilities**, emphasizing in-place evaluation, AST flattening, and dispatch compression, without sacrificing correctness or language semantics.

22.1.1 Identifying Performance Bottlenecks

In a basic interpreter, C-style expression evaluation tends to be slow due to:

- Deep recursion in AST node evaluation
- Virtual dispatch for expression types
- Repeated allocations for intermediate values
- Lack of constant folding or short-circuit optimizations
- No memoization of constant subtrees

To address these, we aim to minimize allocations, reduce indirection, and apply modern C++ techniques such as `std::variant`, `std::optional`, lambdas, and expression flattening.

22.1.2 Evaluation Model: Recursive vs Iterative

- Recursive Evaluation (baseline)

```
Value evaluate(const Expr& expr) {
    switch (expr.kind()) {
        case ExprKind::Binary:
            return evaluate_binary(expr.as<BinaryExpr>());
        case ExprKind::Unary:
            return evaluate_unary(expr.as<UnaryExpr>());
        case ExprKind::Literal:
            return expr.value;
        ...
    }
}
```

  Although easy to implement, this can lead to:

  – Deep call stacks

  - Performance hits on branch misprediction
  - Memory overhead due to recursive allocations

- Iterative Evaluation via Flattening

  By flattening expressions into **postfix** or **evaluation trees**, we can simulate a virtual stack machine:

```
std::vector<Instruction> compile_to_postfix(const Expr& expr);
Value evaluate_postfix(const std::vector<Instruction>& code);
```

  This removes recursion and enables better CPU caching and branch prediction. It mimics JIT-like speed while remaining within an interpreter.

22.1.3 Optimizing Constant Expressions: Folding and Hoisting

Implement a **pre-evaluation pass** to compute static values:

```
ExprPtr optimize_constants(const ExprPtr& expr) {
    if (auto bin = dynamic_cast<BinaryExpr*>(expr.get())) {
        auto left = optimize_constants(bin->left);
        auto right = optimize_constants(bin->right);

        if (left->is_constant() && right->is_constant()) {
            return make_literal(evaluate_binary(*left, *right, bin->op));
        }
    }
    return expr;
}
```

C++20 features like `consteval`, `constinit`, and `constexpr if` aid in distinguishing compile-time evaluable expressions even in your interpreter's internal logic.

22.1.4 Short-Circuit Boolean Evaluation

C-style logic expressions (using `&&` and `||`) require **short-circuit semantics** to avoid evaluating unnecessary operands.

```
Value evaluate_logical_and(const Expr& lhs, const Expr& rhs) {
    Value lval = evaluate(lhs);
    if (!lval.as_bool()) return lval;
    return evaluate(rhs);
}
```

Avoid calling both sides unless required. This boosts performance and avoids unintended side effects.

22.1.5 Memory Efficiency and Value Reuse

To reduce heap allocations and copy costs:

- Use `std::variant` or `ValueRef`-style pointers with small object optimization
- Store temporary values in an **evaluation context** or **scratch space**
- Prefer `std::optional<T&>` or `std::span` over raw copies

Example:

```
class EvaluationContext {
public:
    std::array<Value, 128> value_stack;
    size_t top = 0;

    Value& push(Value v) { return value_stack[top++] = std::move(v); }
```

```
    Value pop() { return std::move(value_stack[--top]); }
};
```

This approach removes most allocations per expression, improving speed significantly.

22.1.6 AST Layout for Cache Locality

Use **inline storage** and **struct-based AST** for better memory layout and cache behavior.
Bad (pointer-heavy):

```
struct BinaryExpr {
    Expr* left;
    Expr* right;
    Token op;
};
```

Better (flat object):

```
struct BinaryExpr {
    std::unique_ptr<Expr> left;
    std::unique_ptr<Expr> right;
    Token op;
};
```

Best (tight layout using variant):

```
using Expr = std::variant<LiteralExpr, BinaryExpr, UnaryExpr, VariableExpr>;
```

This allows fast dispatch via `std::visit`, reducing vtable lookup.

22.1.7 Dispatch Optimization: `std::visit` vs `if constexpr`

With C++20 `std::variant`, use static dispatch:

```
std::visit([&](auto&& expr) {
    using T = std::decay_t<decltype(expr)>;
    if constexpr (std::is_same_v<T, BinaryExpr>) {
        return evaluate_binary(expr);
    } else if constexpr (std::is_same_v<T, LiteralExpr>) {
        return expr.value;
    }
}, expr_variant);
```

This removes virtual function overhead, improves branch prediction, and unlocks compiler inlining.

22.1.8 Operator Table Optimization

For interpreters with many operators, maintain a **fixed operator dispatch table** using `unordered_map<TokenType, OpHandler>` or flat dispatch arrays.
Even better, create a compact enum-to-function map:

```
constexpr std::array<std::function<Value(const Value&, const Value&)>, OperatorCount>
↪  binary_op_table = {
    &add_op, &sub_op, &mul_op, ...
};
```

Access via index:

```
return binary_op_table[static_cast<size_t>(op_token)](lhs, rhs);
```

22.1.9 Inlining and Precomputed Expression Paths

When interpreting frequently repeated expressions (e.g., in loops), cache compiled instructions (or mini bytecode):

```
if (auto cached = cache.find(expr_id)) {
    return evaluate_postfix(cached->second);
}
```

This avoids reparsing and re-evaluating the same structure each time.

22.1.10 Testing and Profiling Optimizations

Use C++23 `std::chrono::utc_clock` and `std::chrono::high_resolution_clock` for precise benchmarking:

```
auto start = std::chrono::high_resolution_clock::now();
// evaluate expression
auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
std::cout << "Eval took " << duration.count() << " ns\n";
```

Add benchmarks for:

- Expression depth
- Number of binary operations
- Short-circuit evaluation
- Function call overhead

22.1.11 Summary

By optimizing C-style expression evaluation with Modern C++20/23 features, your interpreter achieves:

- Elimination of excessive recursion and allocations
- Constant folding for faster execution
- Efficient short-circuiting logic
- Inlined dispatch and cache-friendly AST layout
- Foundation for future JIT compilation or bytecode engines

This step transforms the interpreter from an educational prototype into a performant, production-grade tool ready to support non-trivial scripting and embedded usage.

22.2 Memory Management for Language Runtime

22.2.1 Introduction

Memory management is a foundational pillar in any language runtime, and for a C-style interpreter, getting it right means balancing **performance**, **correctness**, and **safety**. Unlike higher-level managed languages, C-style languages follow explicit and deterministic lifetime rules. As the designer of such a language, you must implement a memory system that honors these expectations—mimicking stack-based lifetimes, allowing manual deallocation in the future if needed, and ensuring efficient memory reuse.

With Modern C++20/23, memory management in your interpreter can be designed with **strong ownership semantics**, **region-based allocators**, **arena pools**, and

compile-time memory contract enforcement via `constexpr`, `constinit`, and `span` types.

22.2.2 Memory Categories in C-Style Languages

Your interpreter will manage different categories of memory:

1. **Temporary Expression Values**
    - Created during evaluation and discarded after the expression completes.
2. **Function Local Variables (Stack-like)**
    - Exist within function scopes or `{}` blocks.
3. **Global Variables and Constants**
    - Persistent throughout the program lifecycle.
4. **Heap-like Allocations (Arrays, Objects)**
    - May be used explicitly by the language or internally.
5. **Runtime Structures**
    - AST nodes, environments, symbols, and error traces.

22.2.3 Modern C++ Memory Tools for Interpreter Design

C++20/23 offers memory tools that align well with interpreter design:

C++ Feature	Use Case
`std::unique_ptr`	Ownership of runtime objects
`std::shared_ptr`	Shared references (avoid overuse)
`std::pmr::memory_resource`	Arena-based memory pooling
`std::span`	Safe views into arrays, buffers
`std::optional`	Nullable return values without heap
`consteval` / `constexpr`	Compile-time computed constants
`std::vector<T, Alloc>`	Pooled memory containers

22.2.4 Stack Frame and Local Variable Lifetime

Implementing a stack-based lifetime model is crucial. Mimic the call stack using scoped frames:

```
struct StackFrame {
    std::unordered_map<std::string, Value> locals;
    size_t base_pointer;
};
```

The interpreter keeps a stack of frames:

```
std::vector<StackFrame> call_stack;
```

Push a new frame on function entry, and pop it on return. Local variable memory can be managed in-frame, avoiding heap allocations for primitive types.
Use **inline buffers** for small objects:

```
struct Value {
    std::variant<int64_t, double, bool, std::string> data;
};
```

For C-style arrays declared inside functions, use `std::vector<Value>` with reserve to avoid dynamic growth at runtime.

22.2.5 Arena Allocation for AST and Environments

To optimize parsing and runtime object creation, use **arena (region-based) allocation** for short-lived objects like AST nodes or temporary evaluation contexts.

```
class AstArena {
    std::pmr::monotonic_buffer_resource buffer;
    std::pmr::vector<std::unique_ptr<AstNode>> nodes;
public:
    template <typename T, typename... Args>
    T* create(Args&&... args) {
        T* ptr = static_cast<T*>(buffer.allocate(sizeof(T)));
        new (ptr) T(std::forward<Args>(args)...);
        return ptr;
    }
};
```

This makes allocation extremely fast and allows bulk deallocation.

22.2.6 Managing Heap-like Memory Safely

Even if your interpreted language supports `new` or dynamic arrays, do not rely on raw heap allocations.
Implement a custom `HeapManager`:

```
class HeapManager {
    std::unordered_map<Handle, Value> heap_store;
    Handle next_id = 0;

public:
    Handle allocate(Value v) {
        heap_store[next_id] = std::move(v);
        return next_id++;
    }

    Value& get(Handle h) { return heap_store[h]; }
    void deallocate(Handle h) { heap_store.erase(h); }
};
```

This model is essential for garbage collection later and ensures all memory is under your control.

22.2.7 Avoiding Memory Fragmentation

Interpreters may execute thousands or millions of instructions. Fragmentation from repetitive `new`/`delete` calls leads to performance degradation.
Solutions:

- Use `std::pmr::unsynchronized_pool_resource` for pooled allocations.
- Prefer `std::vector::reserve()` to minimize reallocations.
- Reuse memory for value objects (e.g., evaluation registers).

```
class ValuePool {
    std::vector<Value> buffer;
    size_t cursor = 0;

public:
    Value& next() {
        if (cursor >= buffer.size()) buffer.emplace_back();
        return buffer[cursor++];
    }

    void reset() { cursor = 0; }
};
```

22.2.8 Memory Profiling and Leak Detection

Modern interpreters benefit from built-in memory profiling tools:

1. Count allocations:

```
size_t alloc_count = 0;
void* operator new(size_t size) {
    ++alloc_count;
    return std::malloc(size);
}
```

2. Use tools:

   - Valgrind (Linux)
   - Dr. Memory (Windows)
   - AddressSanitizer (GCC/Clang)

- LeakSanitizer

3. Integrate counters:

   - Count memory per file, per scope, per function
   - Track maximum memory usage for large scripts

22.2.9 Object Lifetime Contracts

Use C++20 `constinit`, `[[nodiscard]]`, and `[[no_unique_address]]` to ensure clean semantics:

```
struct [[nodiscard]] HeapObject {
    Value data;
    bool marked;
};

struct Environment {
    [[no_unique_address]] std::pmr::polymorphic_allocator<> alloc;
};
```

`constinit` ensures global singletons are initialized correctly:

```
constinit static RuntimeState runtime;
```

22.2.10 Memory Error Prevention

Prevent use-after-free and null dereferencing:

- Enforce strict ownership models with `std::unique_ptr`

- Avoid raw pointers
- Use `std::optional<Value>` to indicate absence
- Define clear object life boundaries (function scope, global scope)

For arrays and strings, always expose through `std::span<T>` or `std::string_view`:

```
void print_array(std::span<const Value> arr) {
    for (const auto& val : arr) std::cout << val << ", ";
}
```

22.2.11 Summary

Efficient and safe memory management is a cornerstone of a reliable interpreter. With C++20/23, you can build a runtime that avoids the pitfalls of manual memory while preserving the performance and predictability of C-style languages.
By:

- Mimicking stack-frame lifetimes
- Using arena and pool allocators
- Managing heap via handles
- Preventing fragmentation
- Enforcing lifetime contracts

…you enable both high performance and maintainability. These techniques future-proof your runtime for extensions like garbage collection, foreign function interfaces, and long-lived script sessions.

22.3 Profiling and Bottleneck Identification

22.3.1 Introduction

To design a performant interpreter in Modern C++20/23, writing fast code alone is not enough. You must **profile** your interpreter to identify and eliminate real-world bottlenecks. Optimization should always be **data-driven**, not based on assumptions. This section focuses on how to analyze performance, collect accurate measurements, and uncover the slowest parts of your interpreter.
We'll cover:

- Granular instrumentation techniques
- Using built-in C++20/23 tools
- Custom timing utilities
- Integration with third-party profilers
- Evaluation strategies for language-specific hot paths

22.3.2 Why Profiling is Critical

Interpreters are inherently performance-sensitive:

- Each instruction may involve dynamic dispatch
- Function calls and scoping increase overhead
- Evaluating expressions or parsing strings introduces latency

Without profiling:

- You optimize code that isn't slow
- You miss structural inefficiencies
- You don't quantify the gains of optimization passes

Profiling bridges this gap between intuition and truth.

22.3.3 Instrumentation Using Modern C++

C++20 and C++23 provide tools to build **custom profiling utilities** without external dependencies.

- **High-resolution Timing**

  Use `std::chrono::high_resolution_clock` or `std::chrono::steady_clock`:

  ```
  auto start = std::chrono::high_resolution_clock::now();
  // interpret code
  auto end = std::chrono::high_resolution_clock::now();
  auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end -
  ↪  start).count();
  std::cout << "Execution took " << elapsed << " µs\n";
  ```

  This technique can be embedded into:

  – Expression evaluation
  – Function execution
  – Loop bodies
  – Built-in function calls

  Use macros or RAII wrappers for repeated usage.

- **Scoped Profiler**

```
struct ScopedTimer {
    std::string label;
    std::chrono::high_resolution_clock::time_point start;

    ScopedTimer(std::string name)
        : label(std::move(name)),
        ↪  start(std::chrono::high_resolution_clock::now()) {}

    ~ScopedTimer() {
        auto end = std::chrono::high_resolution_clock::now();
        auto duration =
        ↪  std::chrono::duration_cast<std::chrono::microseconds>(end -
        ↪  start).count();
        std::cout << label << " took " << duration << " µs\n";
    }
};
```

Use like this:

```
void eval_expr(const Expr& expr) {
    ScopedTimer profiler("Expression Evaluation");
    ...
}
```

22.3.4 Building a Centralized Profiler

To aggregate results across the interpreter:

```
class Profiler {
    struct Entry {
        uint64_t count = 0;
        uint64_t total_time = 0;
    };

    std::unordered_map<std::string, Entry> metrics;

public:
    void record(const std::string& name, uint64_t duration) {
        auto& entry = metrics[name];
        entry.count++;
        entry.total_time += duration;
    }

    void report() const {
        for (const auto& [name, entry] : metrics) {
            std::cout << name << ": " << entry.total_time << " µs over "
                      << entry.count << " calls\n";
        }
    }
};
```

This lets you measure cost across interpreter stages: parsing, evaluation, I/O, symbol resolution, etc.

22.3.5 Targeting Hot Paths

Focus profiling on parts that are:

- Called frequently (expression trees in loops)
- Recursively invoked (function calls)

- Memory-sensitive (AST creation)
- String-heavy (variable lookup, I/O)

Use sampling or flag-based tracing to avoid slowdown:

```
if (profiler_enabled)
    profiler.record("FunctionCall", elapsed);
```

22.3.6 External Tools for Full-Scale Profiling

In large projects or compiled interpreters, use professional tools to complement your own data.

Cross-platform:

- **Valgrind (Linux)**: `callgrind` and `massif` for CPU and memory profiling.
- **gprof** or **perf**: Linux system profilers.
- **Instruments (macOS)**: Time profiler and memory inspector.
- **Visual Studio Profiler**: Deep integration for Windows-based interpreters.

Sampling vs Instrumentation:

- Sampling gives low-overhead stack snapshots
- Instrumentation measures every event, more precise but heavier

Compile with `-pg` or `-fprofile-arcs -ftest-coverage` if needed.

22.3.7 Profiling the Interpreter Lifecycle

- **Stage 1: Startup**
  - Lexing and parsing performance
  - AST construction
  - Memory allocation spikes
- **Stage 2: Execution**
  - Hot loops (e.g., repeated conditions or math)
  - Function call overhead
  - Symbol table lookups
  - Expression evaluation trees
- **Stage 3: Cleanup**
  - Destruction of value trees
  - Memory deallocation delays
  - Releasing pooled resources

  All phases should be profiled independently for clarity.

22.3.8 Interpreted Language Profiling Considerations

Unlike compiled code, interpretation adds abstraction layers:

- Evaluating `a = b + c * d;` involves multiple function calls
- Dispatch logic can be costly if not flattened

- Temporary objects often incur frequent allocations

Optimize by:

- Replacing polymorphism with `std::variant` + `std::visit`
- Using in-place storage for `Value` objects
- Avoiding dynamic allocations for small types
- Reusing evaluation contexts

22.3.9 Summary Reports

In production interpreters, expose a built-in `profile` command:

```
.profile on
.run script.lang
.profile report
```

This helps users understand which parts of their script are slow and why.

22.3.10 Summary

Profiling is not an afterthought—it is a continuous process throughout interpreter development. By using Modern C++ features and custom profilers, you can:

- Eliminate hotspots in evaluation
- Track excessive memory or function calls
- Replace naive structures with optimized versions
- Deliver a language runtime that performs well under real-world pressure

Accurate profiling combined with targeted optimizations is the key to turning a working interpreter into a fast, reliable, and production-ready language engine.

22.4 Hands-on — Performance Measurement and Tuning

22.4.1 Overview

This section is a **practical guide** to applying performance analysis techniques and **actively tuning** your interpreter. Building on profiling insights, we now focus on how to:

- Measure performance at key runtime points
- Eliminate bottlenecks using real data
- Apply C++20/23 features to boost speed and resource usage efficiency
- Design reproducible test scripts to benchmark improvements

This hands-on section is crucial to making your interpreter not just correct but competitive and scalable for larger programs.

22.4.2 Setting Up a Benchmark Harness

Before tuning, build a reliable way to measure:

```
struct Benchmark {
    std::string name;
    std::chrono::high_resolution_clock::time_point start;
```

```
    Benchmark(std::string n) : name(std::move(n)),
    ↪  start(std::chrono::high_resolution_clock::now()) {}

    ~Benchmark() {
        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end -
        ↪  start).count();
        std::cout << "[BENCH] " << name << " = " << elapsed << " µs\n";
    }
};
```

Wrap sections like:

```
{
    Benchmark bench("Function call loop");
    run_script("bench/function_loop.lang");
}
```

You can enhance it with aggregate stats, CSV outputs, or integration with external visualization tools.

22.4.3 Tuning Evaluation Performance

Expression evaluation is one of the most frequent operations. Focus on:

- **a. Inline Simple Arithmetic**

  Instead of polymorphic dispatch:

  ```
  Value eval_add(const Value& lhs, const Value& rhs) {
      if (lhs.is_int() && rhs.is_int())
  ```

```
        return Value(lhs.as_int() + rhs.as_int());
    // fallback path
}
```

Use **`std::visit` on `std::variant`** for type-safe and branchless operations:

```
Value apply_op(Value a, Value b, OpKind op) {
    return std::visit([&](auto x, auto y) -> Value {
        using T = std::decay_t<decltype(x)>;
        using U = std::decay_t<decltype(y)>;
        if constexpr (std::is_same_v<T, int> && std::is_same_v<U, int>) {
            return Value(op == Add ? x + y : x - y);
        }
        // more combinations
    }, a.data, b.data);
}
```

Avoid unnecessary heap allocations—store small types **in-place** using `std::variant`.

22.4.4 Memory Allocation Reduction

Many runtime bottlenecks stem from frequent small allocations (AST nodes, intermediate expressions).

- **Apply Arena Allocators:**

  Use `std::pmr::monotonic_buffer_resource` for fast, pooled allocations:

```
std::pmr::monotonic_buffer_resource pool;
std::pmr::vector<Expr*> nodes{&pool};
```

This reduces fragmentation and speeds up allocation/deallocation significantly.

- **Reuse Temporary Buffers:**

```
struct EvalContext {
    std::vector<Value> temp_values;
    void reset() { temp_values.clear(); }
};
```

Use per-evaluation context buffers to avoid rebuilding containers every time.

22.4.5 Optimize Variable Lookup

Symbol resolution can be slow in interpreted languages with lexical scopes.

- **Cache local variable offsets:**

```
struct Variable {
    size_t stack_offset;
};

struct StackFrame {
    std::vector<Value> locals;
    Value& get(Variable var) { return locals[var.stack_offset]; }
};
```

By precomputing the offset during parsing, the runtime cost of lookup is reduced to a single index operation.

22.4.6 Loop Execution Optimizations

Loops (e.g., `for`, `while`) often dominate execution time in scripts.

- Avoid repeated re-evaluation of constant loop bounds
- Unroll very short loops (manually or via JIT hints if extended later)
- For numeric loops, precompute start/end/step outside the loop execution body

22.4.7 Real-World Benchmarking Scripts

To measure improvement across builds, create fixed `.lang` scripts that stress various components:

- **Script: `math_perf.lang`**

```
int x = 0;
int sum = 0;
while (x < 100000) {
    sum = sum + x * x;
    x = x + 1;
}
```

- **Script: `func_call.lang`**

```
int f(int x) {
    return x * x + 5;
}

int result = 0;
int i = 0;
```

```
while (i < 100000) {
    result = f(i);
    i = i + 1;
}
```

- **Script: `symbol_resolve.lang`**

```
{
    int a = 1;
    {
        int b = 2;
        {
            int c = 3;
            result = a + b + c;
        }
    }
}
```

Run these under different interpreter versions and compare output times.

22.4.8 Multi-phase Optimization Loop

1. **Profile**: Using tools or built-in benchmark tags
2. **Isolate**: Find dominant functions or operations
3. **Refactor**: Apply scoped tuning (e.g., remove virtual dispatch, add fast paths)
4. **Benchmark again**: Confirm impact with script timing
5. **Repeat**: Optimization is incremental

22.4.9 Other C++20/23 Features for Performance

Feature	Benefit
`[[likely]]`, `[[unlikely]]`	Branch prediction hints for tight loops
`consteval`	Avoid runtime if logic can be evaluated at compile-time
`std::span`	Avoid heap copying for buffer operations
`no_unique_address`	Compact structs used in value systems
`constexpr` containers	Build immutable lookup tables at compile time

22.4.10 Summary

Performance tuning transforms a basic interpreter into a production-ready engine. Using Modern C++ capabilities, you can measure precisely, optimize effectively, and verify improvements consistently.

A well-tuned interpreter doesn't just pass tests—it executes user scripts with responsiveness, low memory usage, and scalability. This hands-on practice is where your design becomes a reliable tool in real developer workflows.

Chapter 23

Bytecode Virtual Machine (Optional)

23.1 Compiling C-Style Constructs to Bytecode

23.1.1 Introduction

C-style programming constructs—expressions, assignments, control flow, and function calls—are typically evaluated directly in AST interpreters. However, for faster runtime performance and better portability, compiling these constructs into **bytecode** for a virtual machine offers significant advantages. This approach separates **semantic analysis** from **execution**, allowing ahead-of-time validation, optimization, and compact representation.

This section explores how to design a **bytecode compiler** for a C-style language using modern C++20/23, translating high-level constructs into low-level bytecode instructions consumable by a virtual machine.

23.1.2 Bytecode Overview

Bytecode is a compact, instruction-oriented representation of program logic. It is:

- Platform-independent
- Easier to interpret than AST
- Often uses a stack-based or register-based VM model

Your language compiler walks the AST and emits opcodes such as:

```
PUSH_INT 42
LOAD_VAR x
ADD
STORE_VAR y
JUMP_IF_ZERO addr
```

Each high-level C-style statement or expression maps to a sequence of bytecode instructions.

23.1.3 Compiler Architecture

The core of the bytecode compiler is a **tree walker** that converts each AST node into a list of opcodes. The structure involves:

- `BytecodeEmitter`: The primary class that holds the instruction buffer and output logic
- `Opcode`: Enum representing the instruction set
- `Instruction`: A structured form of emitted operations

- `LabelManager`: For managing jumps, forward references, and patches

```
enum class Opcode : uint8_t {
    PUSH_INT, PUSH_FLOAT, PUSH_BOOL,
    LOAD_VAR, STORE_VAR,
    ADD, SUB, MUL, DIV,
    JUMP, JUMP_IF_ZERO,
    CALL, RETURN,
    HALT
};

struct Instruction {
    Opcode opcode;
    std::variant<int32_t, float, bool, size_t> operand;
};
```

23.1.4 Expression Compilation

Each expression is compiled recursively in post-order to respect C-style evaluation:

- **Example: `a = b + 3;`**

    1. Load `b` → `LOAD_VAR b`
    2. Push literal `3` → `PUSH_INT 3`
    3. Apply operator → `ADD`
    4. Store in `a` → `STORE_VAR a`

```
void compile_expression(const Expr& expr) {
    if (expr.kind == Binary) {
        compile_expression(expr.left);
        compile_expression(expr.right);
        switch (expr.op) {
            case '+': emit(Opcode::ADD); break;
            case '*': emit(Opcode::MUL); break;
        }
    } else if (expr.kind == Variable) {
        emit(Opcode::LOAD_VAR, expr.name);
    } else if (expr.kind == LiteralInt) {
        emit(Opcode::PUSH_INT, expr.value);
    }
}
```

23.1.5 Control Flow Compilation

C-style constructs like `if`, `while`, and `for` need label tracking and conditional jumps.

- **`if (cond) { ... } else { ... }`**

```
<cond>          --> compile condition
JUMP_IF_ZERO L1
<then block>    --> emit instructions for then-branch
JUMP L2
L1:
<else block>    --> emit instructions for else-branch
L2:
```

Track jump targets using a `LabelManager` or `PlaceholderIndex`:

```
size_t jump_if_false = emit_with_placeholder(Opcode::JUMP_IF_ZERO);
compile_block(then_branch);
size_t jump_end = emit_with_placeholder(Opcode::JUMP);
patch_jump(jump_if_false, current_address());
compile_block(else_branch);
patch_jump(jump_end, current_address());
```

23.1.6 Loop Constructs

- **`while (cond) { body }`**
  - Start: label `L_start`
  - Evaluate `cond`
  - If false, jump to `L_end`
  - Compile `body`
  - Jump back to `L_start`

```
L_start:
<cond>
JUMP_IF_ZERO L_end
<body>
JUMP L_start
L_end:
```

23.1.7 Function Compilation

Each function gets a separate bytecode stream (or segment). A function definition includes:

- Function name (symbol)
- Entry point index into bytecode
- Parameter count
- Local variable size
- Emitted instructions

Functions are compiled once and stored in a map:

```
struct CompiledFunction {
    std::vector<Instruction> body;
    size_t param_count;
    size_t local_count;
};
std::unordered_map<std::string, CompiledFunction> function_table;
```

Function calls use:

```
PUSH_ARG n
CALL <function_index>
```

The virtual machine handles frame setup, local variable space, and return address.

23.1.8 Optimization Before Emission (Optional)

Before emitting bytecode, the compiler may:

- **Fold constants**: `3 + 5 → 8`
- **Remove dead code** after unconditional jumps

- **Inline trivial functions** where allowed
- **Convert variable names to integer indexes** (symbol resolution)

23.1.9 Bytecode Format Design Considerations

To be efficient:

- Use numeric indexes instead of string names
- Pack opcodes and operands compactly (e.g., single struct)
- Prefer stack-based instructions for simplicity
- Consider operand types (int, float, bool) to reduce runtime checking
- Emit line numbers for error mapping

Example compact bytecode instruction:

```
struct CompactInstruction {
    uint8_t opcode;
    union {
        int32_t int_val;
        float float_val;
        uint32_t index;
    };
};
```

23.1.10 Integration with VM

Once emitted, the compiled bytecode is fed into the virtual machine's dispatcher loop:

```
switch (current.opcode) {
    case Opcode::PUSH_INT: stack.push(current.operand); break;
    case Opcode::ADD: {
        auto b = stack.pop(); auto a = stack.pop();
        stack.push(a + b); break;
    }
    ...
}
```

The interpreter now executes bytecode rather than walking the AST. This:

- Reduces interpretation cost
- Improves cache locality
- Simplifies future optimizations (e.g., JIT or AOT)

23.1.11 Summary

Compiling C-style constructs into bytecode transforms your language into a more efficient, VM-driven model. It enables compact execution, faster control flow handling, and portable distribution.
With C++20/23, modern design features like `std::variant`, `constexpr`, and memory-efficient containers help build a bytecode compiler that is modular, extensible, and high-performance. This foundational phase prepares your language for advanced features like multi-function modules, runtime linking, and possible JIT in future chapters.

23.2 Stack-Based VM for Better Performance

23.2.1 Introduction

A **stack-based virtual machine (VM)** is one of the most efficient and portable execution engines for interpreted and bytecode-compiled languages. In the context of a C-style interpreted language, it aligns perfectly with expression-based execution, supports straightforward function call frames, and reduces memory overhead compared to register-based VMs.

This section details how to design and implement a high-performance, stack-based VM using modern C++20/23 features, based on the bytecode instruction set introduced in Section 1.

23.2.2 Why Stack-Based?

Stack-based VMs push and pop operands to/from an evaluation stack rather than using named or numbered registers. For example, evaluating `a = b + 3` becomes:

```
LOAD_VAR b       ; push value of b
PUSH_INT 3       ; push literal 3
ADD              ; pop two, add, push result
STORE_VAR a      ; pop and store in a
```

Advantages:

- Simple to implement and understand
- Compact bytecode format
- No need to allocate or map registers
- Naturally recursive and function-call-friendly

23.2.3 Core Components of the Stack-Based VM

- **a) Value Stack**

  The central data structure is the **value stack**, where operands and intermediate results are stored.

```
class VMStack {
public:
    std::vector<Value> data;

    void push(const Value& v) { data.push_back(v); }
    Value pop() {
        assert(!data.empty());
        Value v = data.back(); data.pop_back();
        return v;
    }
    Value& top() { return data.back(); }
};
```

  Use `std::vector` for performance, and optionally replace with `std::pmr::vector` for arena-backed allocations.

- **b) Bytecode Instruction Format**

  Each instruction is a lightweight structure containing an opcode and optional operand:

```
enum class Opcode : uint8_t {
    PUSH_INT, LOAD_VAR, STORE_VAR,
    ADD, SUB, MUL, DIV,
    JUMP, JUMP_IF_ZERO, CALL, RETURN, HALT
};
```

```
struct Instruction {
    Opcode opcode;
    int32_t operand; // use as int, address index, var index, etc.
};
```

Modern C++ allows constexpr and strong typing for Opcode, improving safety.

- **c) Execution Engine**

  At the heart is the VM loop (instruction dispatch loop):

```
void run(const std::vector<Instruction>& code) {
    size_t ip = 0; // instruction pointer
    VMStack stack;

    while (ip < code.size()) {
        const Instruction& instr = code[ip++];
        switch (instr.opcode) {
            case Opcode::PUSH_INT:
                stack.push(Value(instr.operand));
                break;
            case Opcode::ADD: {
                auto b = stack.pop().as_int();
                auto a = stack.pop().as_int();
                stack.push(Value(a + b));
                break;
            }
            case Opcode::JUMP:
                ip = instr.operand;
                break;
            case Opcode::HALT:
```

```
                return;
            }
        }
    }
```

23.2.4 Call Stack for Function Support

Each function call uses an **activation record (stack frame)**:

```
struct CallFrame {
    size_t return_ip;
    size_t local_base;
};
```

On a `CALL` instruction:

- Push arguments
- Store current `ip`
- Jump to function's entry
- After `RETURN`, restore `ip` and pop frame

Use a separate `call_stack` for managing `CallFrame` entries.

23.2.5 Stack Discipline and Scope

Stack-based discipline requires:

- **Pushing operands in order**

- **Ensuring every instruction pops exactly what it expects**
- **Managing local variables separately from the operand stack**

Typical layout:

```
| argN |
| ...  |
| arg1 | ← local_base
| var1 |
| var2 |
|-----| ← stack top
```

Variable access like `LOAD_VAR index` is resolved as `stack[local_base + index]`.

23.2.6 Performance Considerations

To ensure high performance:

- Minimize virtual dispatch: use `switch`-based opcode handling
- Avoid heap allocations in the hot path
- Use inlined arithmetic for primitive operations
- Use `std::span` for slices of stack frames (safe and efficient)

C++20 improvements like **`[[likely]]`/`[[unlikely]]`** can guide the compiler:

```
switch (instr.opcode) {
    case Opcode::ADD: [[likely]]
        // common path
        break;
```

```
    case Opcode::HALT: [[unlikely]]
        return;
}
```

23.2.7 Exception-Free Execution

Your VM should avoid exceptions in hot code. Use error codes or `assert` for developer builds, and `Result<T>` patterns (or optional logging) for user error handling.
Example:

```
bool safe_add(VMStack& stack) {
    if (stack.size() < 2) return false;
    auto b = stack.pop(); auto a = stack.pop();
    stack.push(a + b);
    return true;
}
```

23.2.8 Extending the VM with New Instructions

You can easily add more instructions to support:

- Logical operators: `AND`, `OR`, `NOT`
- Comparison: `LT`, `GT`, `EQ`, `NEQ`
- Native functions or system calls via opcode `CALL_NATIVE`

23.2.9 Instruction Tracing for Debugging

For debug builds, provide tracing per instruction:

```
void trace(const Instruction& instr, size_t ip) {
    std::cout << "[TRACE] IP=" << ip << " OPCODE=" << to_string(instr.opcode)
              << " OPERAND=" << instr.operand << "\n";
}
```

You can enable this via flags or `constexpr bool` at compile-time.

23.2.10 Summary

A stack-based VM is ideal for interpreting bytecode for a C-style language. It is simple, efficient, and aligns closely with the postfix expression evaluation model that C uses internally during compilation.
Using C++20/23's modern design tools, such as strong typing (`enum class`), `std::variant` (if needed for polymorphic values), structured bindings, and memory arenas, your VM can remain both elegant and high-performance.

23.3 Instruction Set Design for C-Style Operations

23.3.1 Introduction

The instruction set is the language of the virtual machine. It defines the atomic operations that the interpreter or VM can execute. Designing an instruction set for a **C-style language** requires careful attention to **semantic fidelity**, **performance**, and **expandability**. This section focuses on constructing an instruction set that mirrors the operational behavior of C-like expressions, statements, and control flow using efficient modern C++20/23 constructs.
Your bytecode instruction set must represent fundamental concepts such as arithmetic, variable access, type handling, branching, and function calls—while balancing simplicity and extensibility.

23.3.2 Instruction Set Requirements for C Semantics

To support a faithful C-style execution model, the instruction set should:

- Encode **expression evaluation** with C precedence and associativity.
- Support **lvalue/rvalue** concepts via load/store separation.
- Include **type-specific operations**: integer, float, bool.
- Implement **short-circuit logic** for `&&` and `||`.
- Allow **control flow**: `if`, `while`, `for`, `break`, `continue`.
- Enable **function calls**, parameter passing, and return handling.
- Optionally handle **pointer-like behavior**, if desired.

23.3.3 Instruction Structure

A compact instruction structure can be encoded as:

```
enum class Opcode : uint8_t {
    // Literals and values
    PUSH_INT, PUSH_FLOAT, PUSH_BOOL, PUSH_STRING,

    // Variable access
    LOAD_VAR, STORE_VAR,

    // Arithmetic
    ADD_INT, SUB_INT, MUL_INT, DIV_INT,
    ADD_FLOAT, SUB_FLOAT, MUL_FLOAT, DIV_FLOAT,

    // Comparison
```

```
    EQ_INT, NE_INT, LT_INT, LE_INT, GT_INT, GE_INT,
    EQ_FLOAT, NE_FLOAT, LT_FLOAT, LE_FLOAT, GT_FLOAT, GE_FLOAT,

    // Logical
    LOGICAL_AND, LOGICAL_OR, LOGICAL_NOT,

    // Control Flow
    JUMP, JUMP_IF_TRUE, JUMP_IF_FALSE,

    // Function Calls
    CALL, RETURN, RETVAL,

    // Special
    HALT, NOP
};

struct Instruction {
    Opcode opcode;
    int32_t operand; // can be index, literal ID, jump target, etc.
};
```

23.3.4 Expression Evaluation Instructions

C expressions require left-to-right evaluation with operator precedence. In a stack-based VM, operands are pushed first, then operators are applied.

Example: `x = a + b * c` becomes:

```
LOAD_VAR a
LOAD_VAR b
LOAD_VAR c
MUL_INT
```

```
ADD_INT
STORE_VAR x
```

You can also use specialized bytecodes like `LOAD_VAR_BY_INDEX` for optimized symbol resolution.

23.3.5 Control Flow Instructions

Control flow in C uses conditionals and loops with jump semantics. You need:

- **JUMP**: unconditional jump to an address
- **JUMP_IF_TRUE / JUMP_IF_FALSE**: based on top of stack (boolean)
- **LABEL** (logical, for jump targets at compile-time)
- **BREAK / CONTINUE** can be modeled using conditional jumps

Example: compiling `if (cond) { body } else { alt }`

```
<cond>
JUMP_IF_FALSE L1
<body>
JUMP L2
L1:
<alt>
L2:
```

Use a label patching system during bytecode emission.

23.3.6 Function Instructions

To implement functions:

- **CALL <func_index>**: pushes a new frame and jumps to function
- **RETURN**: returns control to the caller
- **RETVAL**: optionally return a value

These rely on a separate call stack to hold `return_ip`, `local_base`, and arguments. Example:

```
PUSH_ARG 1
CALL f
POP_RETVAL
```

Parameter passing uses `PUSH_ARG`, and results can be managed via the data stack or dedicated `RETVAL`.

23.3.7 Type-Specific Operations

You can encode type-aware variants of arithmetic to prevent runtime type ambiguity:

- `ADD_INT`, `ADD_FLOAT`
- `EQ_INT`, `EQ_FLOAT`
- `PUSH_BOOL`, `LOGICAL_NOT`

This design ensures that each opcode knows the expected operand type and eliminates the need for runtime type dispatching, improving performance and simplifying debugging.

Modern C++ concepts allow templated dispatch for these instructions in the interpreter:

```
template <typename T>
Value eval_add(const Value& a, const Value& b) {
    return Value(a.get<T>() + b.get<T>());
}
```

23.3.8 Variable Instructions

Support for:

- `LOAD_VAR <index>`: push variable onto stack
- `STORE_VAR <index>`: pop and assign to variable
- Optionally: `LOAD_GLOBAL`, `STORE_GLOBAL`, `LOAD_LOCAL`, etc.

Symbols should be compiled into fixed indexes per frame to avoid name lookup at runtime.

23.3.9 Literal Handling

String, int, float, and bool constants can be stored in a literal pool and referenced via index:

```
PUSH_STRING 3   ; Load string at literal table index 3
PUSH_INT 42
```

The VM must have access to the literal table at runtime. C++20 `std::variant` can model the literal pool type.

```
using Literal = std::variant<int, float, bool, std::string>;
std::vector<Literal> literal_table;
```

23.3.10 Special and System Instructions

- `NOP`: No operation, useful for alignment or patching
- `HALT`: Terminate execution
- `DEBUG_BREAK`: optional, for debugging hooks

23.3.11 Instruction Encoding Strategy

To improve memory efficiency:

- Use a fixed-size 4 or 8-byte format
- Encode opcodes as one byte
- Use 3 bytes or 7 bytes for operand as needed
- Consider compression of high-frequency instructions using shorter formats (not required initially)

For example:

```
struct CompactInstr {
    uint8_t opcode;
    int32_t operand;
};
```

Or even packed instruction streams using `std::byte[]` for low-level control.

23.3.12 Expanding the Instruction Set in Future

Design the instruction set to be extensible:

- Reserve unused opcode values
- Use dispatch tables or variant-based decoding
- Define feature groups (e.g., arithmetic, control flow, memory ops)

You may even support versioned instruction sets in the future if the language evolves toward JIT compilation.

23.3.13 Summary

Designing a robust instruction set is critical for an efficient VM and a faithful execution model of your C-style language. The instruction set must:

- Accurately encode semantics
- Optimize for simplicity and speed
- Avoid runtime ambiguity through type-specific instructions
- Be extensible for future features like pointers or native modules

With modern C++20/23 features like strong enums, `std::variant`, concepts, and safer memory models, you can structure a clear and maintainable instruction set that balances performance with flexibility. This lays the groundwork for the next section, where the **execution engine** interprets and runs this instruction stream in a virtual environment.

23.4 Advanced Milestone — High-Performance VM Option

23.4.1 Introduction

A high-performance virtual machine (VM) is a critical component for scaling your interpreter to support real-world applications, performance testing, and production scripting. While a simple stack-based interpreter is an excellent starting point for clarity and correctness, building an optimized execution engine requires deeper consideration of memory layout, instruction dispatch, caching strategies, and type specialization.

In this advanced milestone, we define how to architect a **high-performance VM** tailored to the C-style language you are designing, with full leverage of Modern C++20/23. The goal is to bridge the gap between educational interpreter and performance-grade runtime engine.

23.4.2 Key Goals for High-Performance VM

The advanced VM should meet the following goals:

- **Fast instruction dispatch** with low branch misprediction
- **Typed instruction execution** avoiding variant overhead
- **Efficient memory layout** for call frames, locals, and literals
- **Caching mechanisms** for global symbols and literals
- **Optional register-based execution model**
- **Scalable function calling and recursion**

- **Optional JIT backend (optional milestone)**

23.4.3 Core Optimization Strategies

1. **Direct Threaded Dispatch**

   The most performant dispatch technique for interpreters is **direct threading** via computed gotos (or function pointer tables in C++).

   Instead of using a switch-case (which has branch prediction cost), use a lookup table of instruction handlers:

```
using InstructionHandler = void(*)(VM&);
std::unordered_map<Opcode, InstructionHandler> dispatch_table;
```

   In C++23, `consteval` and `constexpr` lambdas can help build this table at compile-time for better safety and clarity.

2. **Instruction Specialization**

   Split instructions by type and operation to avoid runtime type inspection:

   - `ADD_INT`, `ADD_FLOAT`
   - `MUL_INT`, `MUL_FLOAT`
   - `EQ_BOOL`, `EQ_INT`

   This reduces branching and enables vectorization or inlining by the compiler.

3. **Stack Frame Optimization**

   Use **preallocated frame objects** instead of dynamic memory per function call:

```
struct Frame {
    std::array<Value, MaxLocals> locals;
    uint32_t return_ip;
    uint32_t base_pointer;
};
```

Allocate frames in a vector and reuse them to reduce heap churn. C++20 `std::span` is useful to pass safe references to frame slices.

4. **Literal Pool Caching**

   Store all literals in a fixed-size table and refer to them by offset. Cache the most-used literals in a small per-frame cache to avoid repeated lookup.

   Use:

```
std::vector<std::variant<int, float, bool, std::string>> literal_table;
```

   Modern C++ `std::variant`, `std::visit`, and pattern matching (C++23) allow fast and safe access.

23.4.4 Register-Based VM (Alternative to Stack-Based)

A register VM reduces memory traffic and increases performance by avoiding excessive push/pop operations.
Instruction format changes to:

```
ADD r1, r2, r3  // r1 = r2 + r3
LOAD r1, [var_index]
STORE [var_index], r1
```

Implementing a register allocator at bytecode compile-time simplifies runtime complexity.
Modern C++ helps here with:

- `std::array<Value, N>` to model a register file
- `enum class Register : uint8_t` for typed register access

23.4.5 Inlined Built-ins and Fast-Path Execution

Common functions like `print`, `input`, `len`, etc., can be inlined into the VM with zero overhead.
Mark these as built-in and embed them directly into the opcode dispatch:

```
case Opcode::PRINT_STRING:
    std::cout << get_string(stack.pop()) << '\n';
    break;
```

Use C++20 `constexpr` lookups to register built-in function pointers with known signatures. This avoids late binding.

23.4.6 Function Call Optimization

Tail Call Optimization (TCO)

A high-performance VM can benefit from **tail call optimization**, especially in recursive-style user code.
Detect patterns like:

```
return func(...);
```

And reuse the current stack frame instead of pushing a new one.

Inline Caching
If the same function is called repeatedly from the same call site, inline cache the function pointer and skip resolution.
Use:

```
std::unordered_map<CallSiteID, Function*> call_cache;
```

23.4.7 Parallel Execution and Fibers (Optional)

Once your interpreter is stable, introduce fiber-based concurrency using C++20 `std::jthread` or `coroutine` support.
You can simulate lightweight green threads (used in scripting engines like Lua) to run independent execution contexts.

23.4.8 Performance Metrics and Tuning

Profile hotspots using `std::chrono` or performance profilers. Areas to measure:

- Opcode frequency
- Dispatch time
- Memory usage per frame
- Function call depth
- Time per loop iteration

Tune your VM to prioritize the most common instructions and patterns (e.g., integer arithmetic, loop control, comparisons).
Use `[[likely]]`, `[[unlikely]]` annotations (C++20) to hint branch prediction.

23.4.9 Modular Design for Future JIT

A well-designed high-performance VM can serve as the foundation for later introduction of:

- Bytecode-to-native JIT translation
- WebAssembly backend
- LLVM IR generation

Ensure your VM internals (call stack, registers, instructions) are abstracted cleanly.

23.4.10 Summary

This advanced milestone transforms your interpreter into a near production-grade execution engine, tailored for both scripting flexibility and performance-sensitive tasks. Key achievements include:

- Fully specialized and typed instruction set
- Direct-threaded dispatch with minimal branching
- Efficient memory layout with reused stack frames
- Optimized function calling, inlining, and tail call support
- Readiness for advanced concurrency or JIT enhancements

With the power of C++20/23, you gain modern compile-time safety, span-based memory management, and strong typing for your VM internals. This design provides a robust foundation for supporting real-world scripting, plugin systems, and embedded use cases.

Chapter 24

Language Distribution and Embedding

24.1 Embedding Our Language in C++ Applications

24.1.1 Introduction

One of the most practical and powerful design objectives in a custom C-style programming language is making it embeddable within larger C++ applications. Embedding a scripting or interpreted language allows developers to offer **runtime extension**, **domain-specific configuration**, **custom automation**, and **interactive control** to end-users, without recompilation or low-level dependency.

Modern C++20 and C++23 introduce enhanced features that significantly simplify and empower the embedding process — from safe module APIs to constexpr-enabled dispatch and thread-local interpreters. This section focuses on designing your interpreter's public interface for clean and efficient embedding.

24.1.2 Embedding Architecture Overview

To embed your language in a host C++ application, the following components must be clearly defined and implemented:

- **Interpreter API Interface**: Functions and classes that expose the interpreter to the host program
- **Execution Environment Isolation**: Ability to run multiple scripts without interference
- **Binding Mechanism**: Register host functions, classes, or variables into the interpreter
- **Error Capture and Propagation**: Allow host to receive and respond to interpreter errors
- **Memory Ownership Clarity**: Clear semantics on who owns parsed trees, values, etc.

24.1.3 Creating a C++ Embedding Interface

Define a dedicated interface header, such as `forge_vm_api.hpp`, containing:

```
class ForgeVM {
public:
    ForgeVM();
    ~ForgeVM();

    void loadScript(const std::string& code);
    void run();
    void runFile(const std::string& filename);
```

```
    void setGlobal(const std::string& name, Value val);
    Value getGlobal(const std::string& name);

    void bindFunction(const std::string& name, HostFunction fn);
};
```

The `Value` type represents your interpreter's dynamic value system. A `HostFunction` might be:

```
using HostFunction = std::function<Value(const std::vector<Value>&)>;
```

This design allows seamless integration with any C++ backend module or business logic.

24.1.4 Host Binding: From C++ to the Interpreter

Modern C++'s lambda support allows direct binding of logic from the host:

```
vm.bindFunction("log", [](const std::vector<Value>& args) -> Value {
    for (const auto& arg : args)
        std::cout << to_string(arg) << " ";
    std::cout << "\n";
    return {};
});
```

This enables writing scripts like:

```
log("Hello from embedded script");
```

You may support type conversions using `std::variant`, `std::visit`, or C++23's pattern matching (`if consteval` or `std::match` when available).

24.1.5 Execution Context Control

Support for **multiple virtual machines** in the same process is crucial when embedding into complex software. Each `ForgeVM` instance should:

- Maintain its own symbol table
- Hold a separate call stack
- Optionally allow configuration of standard input/output redirection

You can make the interpreter **thread-local** using:

```
thread_local ForgeVM vm;
```

Or offer global singleton access for simple use cases.

24.1.6 Embedding via Scripting Files

For real-world applications, you may want to load and execute `.lang` files dynamically:

```
vm.runFile("startup.lang");
```

Your language should support ”entry point” functions or script-defined command handlers.
You can even expose functions to host UI events or system services:

```
vm.setGlobal("onLoad", loadHandler);
```

Where `loadHandler` is defined in a `.lang` script and invoked from C++.

24.1.7 Bi-directional Communication

To allow the script to call host functions **and vice versa**, expose function handles:

```
Value handler = vm.getGlobal("onClick");
if (handler.isFunction()) {
    handler.asFunction()(args);
}
```

This enables rich GUI integration, plugin systems, or simulation scripting.
If you need deeper C++ integration, you can allow script-defined classes to extend C++ base types (polymorphic exposure).

24.1.8 Error Handling Strategy

When embedding, the interpreter must not crash the host application on runtime errors. Instead:

- All interpreter execution should be wrapped in `try-catch`
- `RuntimeError` (custom error class) must include full diagnostics
- Errors should be retrievable through the API

Example:

```
try {
    vm.run();
} catch (const RuntimeError& err) {
    std::cerr << "Script error: " << err.message << "\n";
}
```

Expose optional callbacks to propagate errors into GUI environments, logs, or debug consoles.

24.1.9 Embedding Use Cases

Embedding your interpreter unlocks many use cases:

- **Game Engines**: User scripts for NPC behavior, triggers
- **Media Applications**: Custom playback logic or filters
- **Dev Tools**: Macro systems, batch scripts, or configuration languages
- **Scientific Software**: Inline DSLs for computation or simulation
- **Financial Systems**: Rule-based scripting or user-defined strategies

24.1.10 Modern C++ Features Used

Feature	Use Case
`std::function`	Function binding from host to VM
`std::variant`	Dynamic type system for value representation
`std::shared_ptr`	AST ownership and memory safety
`std::optional`	Graceful error reporting and state queries
`constexpr` (C++20)	Compile-time constants for bindings
`source_location`	Debug info on errors
`modules` (C++20)	Modular distribution of the interpreter API

24.1.11 Summary

A well-designed embedding system allows your C-style interpreter to evolve into a fully programmable component inside large C++ systems. With a clear API, type-safe value

handling, safe error management, and modern C++ constructs, your language becomes a powerful extension point for real-world software.
This section finalizes the foundational link between your language runtime and host applications, ensuring adoption potential in diverse development scenarios — from tooling and simulation to automation and end-user scripting.

24.2 Creating Language Runtime Library

24.2.1 Introduction

Creating a **runtime library** for your custom C-style language is a pivotal step that transforms a standalone interpreter into a **reusable engine**. A well-designed runtime enables the interpreter to be distributed as a compiled library (.dll/.so/.a), which can be embedded in applications, linked into tools, and accessed via public APIs. This section explains how to architect, implement, and distribute such a runtime using C++20 and C++23.

24.2.2 Definition and Purpose of the Runtime Library

The runtime library is a collection of **compiled components** that expose the core functionality of the language through a clean and stable **API boundary**. It encapsulates:

- Tokenizer and parser definitions
- AST representation and visitor logic
- Evaluation engine and type system
- Error handling and diagnostics

- Standard functions and runtime utilities
- Execution context and environment (scopes, memory)

24.2.3 Runtime Library Structure

A minimal but expandable structure for the runtime may look like:

```
/src
  /vm
    lexer.hpp / lexer.cpp
    parser.hpp / parser.cpp
    ast.hpp / ast.cpp
    evaluator.hpp / evaluator.cpp
    environment.hpp / environment.cpp
    value.hpp / value.cpp
    runtime_api.hpp
/lib
  forge_runtime_static.lib / forge_runtime.so
/include
  forge_runtime/
    runtime_api.hpp
    value.hpp
    error.hpp
```

Use modern **module partitioning** or **namespace scoping** for internal separation (e.g., `forge::ast`, `forge::eval`, `forge::errors`).

24.2.4 API Interface: Clean and Stable

Public-facing headers should be lean, containing:

- Only forward declarations or type-safe handles

- No internal pointers or raw memory logic
- Clear documentation and naming conventions

Example entry point:

```
namespace forge {

class Runtime {
public:
    Runtime();
    ~Runtime();

    void load(const std::string& code);
    void run();
    Value evaluateExpression(const std::string& expr);

    void setGlobal(const std::string& name, const Value&);
    Value getGlobal(const std::string& name) const;
};

}
```

All implementation details should remain hidden via the **Pimpl idiom** (C++20's `std::unique_ptr` for internal engine).

24.2.5 Compilation and Export Mechanics

For shared library distribution across platforms:

- Use `__declspec(dllexport)` or `__attribute__((visibility("default")))` as needed

- For header files, define a macro `FORGE_API` to wrap export visibility

Example:

```
#ifdef _WIN32
  #define FORGE_API __declspec(dllexport)
#else
  #define FORGE_API __attribute__((visibility("default")))
#endif
```

Then annotate your public classes:

```
class FORGE_API Runtime { ... };
```

24.2.6 C++20/23 Features for Cleaner Runtime

Modern C++ helps reduce runtime library complexity:

- `std::span<T>`: Safe array and memory views
- `std::variant`: Dynamic types for interpreter values
- `std::source_location`: Built-in debug info in errors
- `constexpr` and `consteval`: Compile-time verified constants and functions
- `std::format` (C++20): Safer replacements for `printf`-style output

Also, consider using **C++20 Modules** to enforce logical boundaries and speed up compilation if your toolchain supports it.

24.2.7 Standard Library Initialization

Runtime initialization must register all core functions:

```
void Runtime::initStandardLibrary() {
    setGlobal("print", Builtins::makePrint());
    setGlobal("clock", Builtins::makeClock());
}
```

Functions are injected as C++ lambdas or `std::function<Value(const std::vector<Value>&)>` and wrapped into the `Value` system.
The runtime should also support optional **lazy loading** of modules or external `.lang` files.

24.2.8 Language Runtime as a Static vs. Dynamic Library

Feature	Static Library	Shared Library
Linking	At compile time	At runtime
Portability	Fully contained	Requires dynamic linking support
Size	Larger binaries	Smaller host app
Use case	Embedded devices, tight control	Desktop apps, plugin systems

C++20's module interface units also simplify shared binary builds.

24.2.9 Versioning and Compatibility

It is crucial to maintain a **clear versioning** system for the runtime:

- Version macros (e.g., `FORGE_VERSION_MAJOR`, etc.)
- Version negotiation function
- Serialized AST/bytecode version compatibility
- Optional `buildInfo()` function returning interpreter details

You may expose:

```
std::string getVersion(); // "ForgeLang Runtime 1.0.0"
```

This helps applications handle updates and avoid ABI issues.

24.2.10 Example Usage in a Host App

A basic application embedding the runtime:

```
#include "forge_runtime/runtime_api.hpp"

int main() {
    forge::Runtime vm;

    vm.load("int main() { print(42); }");
    vm.run();

    return 0;
}
```

For scripting support in larger applications, provide a CLI or REPL using the same API.

24.2.11 Testing and Continuous Validation

Ensure that:

- All public APIs are unit-tested
- You provide a **minimal stub executable** that links against the library
- Compiler settings include C++20 or C++23 explicitly (`-std=c++20` or `/std:c++20`)
- You provide debug and release versions

Use `ctest` or GoogleTest to verify ABI stability across builds.

24.2.12 Distribution Guidelines

To distribute the runtime:

- Provide `.lib` or `.so` / `.dll` with matching headers
- Include a minimal README and usage example
- Package using `CMake`, `vcpkg`, or `conan`
- Use semantic versioning (e.g., `v1.2.0`)

Optionally, support a **header-only fallback**, useful for small-scale usage or educational purposes.

24.2.13 Conclusion

The runtime library is the backbone of your interpreter's integration story. By using modern C++ features, you ensure performance, maintainability, and safety. Whether embedded in a game engine, CAD tool, or scripting system, your runtime must present a powerful but clean interface to the outside world. This section prepares your interpreter for real-world reuse, setting the stage for professional adoption and distribution.

24.3 Cross-Platform Distribution

24.3.1 Introduction

Cross-platform distribution is a critical milestone in making your C-style interpreted language practical and accessible beyond development machines. Whether your users run on Linux, Windows, or macOS, your runtime, REPL, and any associated tools must work seamlessly across these environments. Using modern C++20/23 and portable tooling (CMake, standard libraries), this section details how to structure your language for universal availability.

24.3.2 Target Platforms and Considerations

To be considered cross-platform, your language system should reliably support at minimum:

- **Windows 10/11 (x64)**
- **Linux (Ubuntu/Debian/RHEL/Fedora)**
- **macOS (x64 and Apple Silicon)**

Optionally:

- WebAssembly (WASI-based interpreter)
- Mobile (via native wrappers or scripting interfaces)

Each platform introduces specific runtime behavior and filesystem handling, but C++20/23's standard library has reduced the friction compared to earlier standards.

24.3.3 Using CMake for Portable Builds

CMake is the de facto tool for cross-platform C++ build generation. Your interpreter and runtime should be organized in a way that allows CMake to generate build systems for:

- `Visual Studio` (Windows)
- `Unix Makefiles` or `Ninja` (Linux/macOS)
- `Xcode` (macOS)
- `MSYS2/MinGW` (lightweight Windows alternative)

Sample `CMakeLists.txt` structure:

```
cmake_minimum_required(VERSION 3.20)
project(ForgeLang VERSION 1.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_library(forge_runtime STATIC
    src/lexer.cpp
```

```
    src/parser.cpp
    src/evaluator.cpp
    src/runtime.cpp
)

target_include_directories(forge_runtime PUBLIC include)

add_executable(forge_repl src/repl.cpp)
target_link_libraries(forge_repl PRIVATE forge_runtime)
```

Important flags for compatibility:

- `-fPIC` on Unix for shared libraries
- `/EHsc` on MSVC for standard exception semantics

24.3.4 Handling Platform-Specific Differences

4.1 Filesystem and Path Handling

Use `std::filesystem` (C++17 onward) to manage:

- File paths (`std::filesystem::path`)
- Iteration over directories
- Cross-platform file loading

4.2 Newline Differences

Detect newline style when reading text files:

- `\r\n` (Windows)
- `\n` (Unix/macOS)

Normalize internally for consistency.

4.3 Dynamic Library Loading (Optional Plugins)

Abstract dynamic loading across platforms:

- Use `LoadLibrary` / `GetProcAddress` on Windows
- Use `dlopen` / `dlsym` on Linux/macOS

Wrap in a platform abstraction layer.

24.3.5 Compiler Compatibility

Your language should compile successfully with:

- **MSVC 2019/2022** (Visual Studio)
- **Clang 12+**
- **GCC 10+**

Ensure:

- Use of standard features only (no compiler extensions)
- Avoid platform-specific intrinsics
- Enable warnings: `/W4` on MSVC, `-Wall -Wextra` on GCC/Clang

Test `std::variant`, `std::span`, and `std::format` thoroughly on all compilers, as their behavior varies slightly in earlier C++20 implementations.

24.3.6 Packaging for Distribution

6.1 Static Binary + Assets

Bundle interpreter as a single binary:

```
/forge/
    forge.exe or forge (binary)
    stdlib/
         io.lang
    examples/
          hello.lang
```

6.2 Shared Library with Headers

For embedding:

```
/forge-runtime/
    include/
         forge_runtime.hpp
    lib/
          libforge_runtime.a / .so / .dll
```

6.3 Archive Formats

- `.zip` for Windows
- `.tar.gz` for Linux/macOS

Use `CMake install` directives to automate packaging.

24.3.7 Testing Across Platforms

Use CI systems that support multiple OS targets:

- **GitHub Actions**
  - Windows: `windows-latest`
  - Linux: `ubuntu-latest`
  - macOS: `macos-latest`

Sample GitHub Action for testing:

```yaml
jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
    steps:
      - uses: actions/checkout@v3
      - name: Configure
        run: cmake -B build -DCMAKE_BUILD_TYPE=Release
      - name: Build
        run: cmake --build build
      - name: Test
        run: build/forge_repl --version
```

24.3.8 Optional: WebAssembly (WASI) Target

To run the interpreter in the browser or WASI-based containers:

- Use **Emscripten** (`emcc`) or **Clang's WASI SDK**
- Avoid threads and system-specific calls
- Strip down file I/O, replacing with in-memory streams

CMake Emscripten example:

```
emcmake cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

This enables deploying your interpreter as a `.wasm` file with JavaScript wrappers.

24.3.9 API Compatibility Across Platforms

Your interpreter must ensure:

- The same type system behavior on all platforms (e.g., 32 vs. 64-bit)
- `int`, `float`, and memory layout should be deterministic
- Endianness is consistent, or data is abstracted

Use `static_assert(sizeof(T) == N)` checks and `std::int32_t`, `std::float32_t` explicitly.

24.3.10 Summary and Best Practices

Principle	Practice
Portability	Use standard C++20/23, avoid OS-specific calls
Tooling	CMake + GitHub Actions + clang/gcc/msvc
Abstraction	Wrap system-specific behavior cleanly
Testing	CI across OSes with minimal dependencies
Packaging	Separate binary/runtime vs. headers/lib

24.3.11 Conclusion

Cross-platform distribution transforms your language from a research project into a professional-grade tool. By combining modern C++20/23 features with a portable build system, careful API planning, and automated validation, you ensure that developers on any platform can adopt and embed your language smoothly. This section is foundational for community growth, language adoption, and long-term success.

24.4 Final Milestone – Production-ready C-style Language

24.4.1 Introduction

This section represents the culmination of the entire design and implementation journey. By this stage, the interpreter has evolved from a minimal REPL into a full-featured, production-ready C-style language system that is robust, modular, embeddable, cross-platform, and performant. The goal is to finalize all critical components and ensure the entire stack is clean, maintainable, testable, and capable of serving real-world programming scenarios.

A **production-ready interpreter** means it:

- Supports all core language features (functions, variables, types, control flow)
- Offers comprehensive error handling and debugging capabilities
- Performs reliably on major platforms
- Can be embedded or extended
- Has a clean build and packaging system

24.4.2 Checklist for Production Readiness

To consider the language implementation production-ready, all the following must be fulfilled:

1. **Language Core**
    - Full support for:
        - `int`, `float`, `bool`, `string`, and array types
        - Variable declarations with proper scoping rules
        - C-style control flow: `if`, `else`, `while`, `for`, `do-while`, `return`
        - Function definitions and calls with recursion
        - Expression evaluation respecting operator precedence and type promotion
    - Consistent and predictable behavior that mirrors the mental model of C programmers
2. **Evaluation Engine**
    - Clean separation of:
        - AST construction
        - Interpretation (value stack or environment-based)
        - Execution (statement interpreter)
    - Safe and optimized value operations
    - Accurate error propagation and diagnostics
3. **Scoping and Symbol Resolution**

- Lexical scoping with full block-level scope handling
- Shadowing and resolution as per C-style rules
- Proper symbol table hierarchy with memory isolation per block

4. **Runtime Library**

- Implementation of key built-in functions (`print`, `input`, `read`, `length`)
- File I/O abstraction
- String and array manipulation utilities
- Math helpers (optionally with C++ `<cmath>` or similar mappings)

5. **REPL and Script Execution**

- Support for `.lang` file execution via command line
- Interactive REPL with:
  - Multi-line input
  - Error trace
  - State persistence
- Module or include system for reusable code

6. **Debugging and Error Handling**

- AST visualization support (for internal testing or developer tooling)
- Error messages with file name, line number, column
- Stack traces during runtime exceptions
- Detection of:

  - Undefined variables
  - Division by zero
  - Type mismatches
  - Bounds violations (arrays)

7. **Performance**

   - Basic profiling tools integrated or supported
   - Efficient runtime memory model (value store + environment hierarchy)
   - Optional bytecode virtual machine support
   - Tail recursion support (optional optimization)

8. **Cross-Platform Distribution**

   - Works on Windows, Linux, macOS
   - Easily buildable using `CMake`
   - Can be distributed as:
     - Single binary (with static runtime)
     - Interpreter + dynamic runtime
     - Embedded library with C++ headers
   - Integration tested via CI/CD pipelines

9. **Embedding Support**

   - The interpreter must expose a clean API for embedding
   - Ability to pass values between host application and interpreter (input/output bindings)

- Host can evaluate code snippets or full scripts via function calls

10. **Documentation and Examples**

    - Full documentation for:
        - Language syntax
        - Built-in functions
        - Embedding interface
        - Examples and tutorials
    - Sample programs and use-cases:
        - Calculators
        - Scripted I/O handlers
        - Recursive problem solvers (e.g., Fibonacci, factorial)

24.4.3 Packaging and Deployment

A production-ready interpreter should come with:

- Well-structured build system:
    - `build/`, `src/`, `include/`, `tests/`, `scripts/`
- Installation instructions and automation (`make install`, `ninja install`, or CPack)
- Optionally:
    - Versioning system (using `git tags` or `project(x VERSION ...)`)
    - Precompiled binaries
    - Language server or editor integration stub (future extension)

24.4.4 Maintainability

Refactor the codebase before release to ensure:

- Each module is single-responsibility and testable
- Headers are guarded and properly modularized
- Avoidance of globals or tightly-coupled internals
- Extensive unit tests for:
  - Lexer
  - Parser
  - Evaluator
  - Runtime errors
- Optional: Use C++20 `concepts` to enforce type constraints and interface usage

24.4.5 Extensibility

A production-ready system should be open to future enhancements:

- Support for structs or user-defined types
- More built-in functions (math, string, network)
- Optional garbage collector
- Integration with GUI or game engines
- WebAssembly target (via Emscripten or WASI)

24.4.6 Final Words

This **Final Milestone** wraps up the long journey from concept to compiler. Your interpreter, implemented using Modern C++20/23, is now mature, stable, and adaptable. By following well-defined design practices, leveraging the best of modern C++, and honoring the C-style heritage, you've created a language that not only teaches and demonstrates deep compiler/interpreter construction but is also viable for real programming and scripting tasks.

This phase is where open-source collaboration can begin, showcasing your language to the developer world, receiving contributions, and building a user community.

Your interpreter is now no longer a prototype. It is a **production-grade programming language system**, capable of taking its place in the world of developer tools.

Appendices

Appendix A: Complete Language Grammar (EBNF)

Target Interpreter: C++20/23 Implementation
Focus: Syntax, Semantics, Modularity, and Tooling Suitability

Purpose and Context

This appendix presents the full grammar specification of the new C-style programming language, formalized in **Extended Backus–Naur Form (EBNF)**. It is designed for direct integration into a recursive-descent parser or a parser combinator engine implemented in Modern C++ (C++20/23). The grammar serves multiple goals:

- Provides the canonical source of truth for syntax parsing.
- Enables development of syntax-highlighting and static analysis tools.
- Supports grammar-driven test case generation.
- Facilitates language bootstrapping and transpiler compatibility.
- Maps directly to an AST node system implemented via `std::variant`, `std::shared_ptr`, and `std::visit`.

Grammar Notation (Clarification)

This grammar uses the following conventions (subset of ISO EBNF):

- `::=` defines a rule
- `[...]` denotes an optional item
- `{...}` denotes zero or more repetitions
- `"..."` denotes literal characters
- `<...>` denotes non-terminal identifiers
- Terminals are defined via the lexer or token stream.

Top-Level Program Rule

```
<program> ::= { <declaration> | <function-declaration> | <import-statement> }
```

A program is a sequence of either top-level declarations, function definitions, or import statements. Top-level expressions are not allowed outside of function bodies.

Import and Module Support (Optional)

```
<import-statement> ::= "import" <string-literal> ";"
```

> Semantic Note: Each string represents a `.lang` module file path. Your language runtime is responsible for source-to-AST resolution and caching to avoid cyclic imports.

Type System

```
<type> ::= "int" | "float" | "bool" | "string"
         | <type> "[" <expression> "]"     (* array type *)
         | <type> "*"                      (* pointer-like *)
         | "void"
```

Extendable to include `struct`, `enum`, and `function` types. Pointers and arrays are first-class citizens in the value system.

Identifiers

```
<identifier> ::= <letter> { <letter> | <digit> | "_" }
```

Lexical rules enforce that identifiers may not begin with digits or reserved keywords. Optional Unicode support can be added for identifiers.

Declarations

```
<declaration> ::= <type> <identifier> [ "=" <expression> ] ";"
```

Variable declarations are strongly typed and may include initialization. Hoisting is not supported—variables are block-scoped.

Function Declarations

```
<function-declaration> ::= <type> <identifier> "(" [ <parameter-list> ] ")" <block>

<parameter-list> ::= <parameter> { "," <parameter> }
<parameter> ::= <type> <identifier>
```

Functions are single-return and have fixed parameters. Variadic functions and default arguments can be added in the extended syntax layer.

Statements

```
<block> ::= "{" { <statement> } "}"

<statement> ::= <block>
              | <declaration>
              | <expression-statement>
              | <if-statement>
              | <while-statement>
              | <do-while-statement>
              | <for-statement>
              | <return-statement>
              | <break-statement>
              | <continue-statement>

<expression-statement> ::= [ <expression> ] ";"
<return-statement> ::= "return" [ <expression> ] ";"
<break-statement> ::= "break" ";"
<continue-statement> ::= "continue" ";"
```

Control Flow Statements

```
<if-statement> ::= "if" "(" <expression> ")" <statement>
                  [ "else" <statement> ]

<while-statement> ::= "while" "(" <expression> ")" <statement>
<do-while-statement> ::= "do" <statement> "while" "(" <expression> ")" ";"

<for-statement> ::= "for" "(" [ <expression> ] ";" [ <expression> ] ";" [
↪  <expression> ] ")" <statement>
```

> Each control statement introduces a new block scope in the environment. All branches are required to be exhaustively typed in the interpreter's evaluator phase.

Expressions (Precedence Correct)

```
<expression> ::= <assignment>

<assignment> ::= <logical-or> [ "=" <assignment> ]

<logical-or> ::= <logical-and> { "||" <logical-and> }
<logical-and> ::= <equality> { "&&" <equality> }
<equality> ::= <comparison> { ("==" | "!=") <comparison> }
<comparison> ::= <term> { ("<" | ">" | "<=" | ">=") <term> }
<term> ::= <factor> { ("+" | "-") <factor> }
<factor> ::= <unary> { ("*" | "/") <unary> }
<unary> ::= ("!" | "-") <unary> | <postfix>
<postfix> ::= <primary> { <postfix-suffix> }
<postfix-suffix> ::= "(" [ <argument-list> ] ")" | "[" <expression> "]"
```

```
<argument-list> ::= <expression> { "," <expression> }
```

AST is generated left-associatively for most binary operations. Precedence is preserved through rule layering. You may extend this to include ternary `? :` and compound assignments.

Literals

```
<literal> ::= <int-literal> | <float-literal> | <string-literal> | <bool-literal> |
↪  <array-literal>

<int-literal> ::= <digit> { <digit> }
<float-literal> ::= <digit> { <digit> } "." <digit> { <digit> }
<string-literal> ::= '"' { <character> } '"'
<bool-literal> ::= "true" | "false"
<array-literal> ::= "[" [ <expression> { "," <expression> } ] "]"
```

Lexical Support

Handled at the lexer level:

- Unicode strings: UTF-8 encoded
- Comments:

  ```
  <line-comment> ::= "//" { any character except newline }
  <block-comment> ::= "/*" { any character } "*/"
  ```

- Escape sequences inside strings: `\n`, `\t`, `\\`, `\"`, etc.

Future Extensions (Reserved for V2)

- `struct` and `enum` definitions
- `match` expressions for pattern matching
- Function overloading
- Attributes (`@inline`, `@deprecated`)
- Modules and packages
- Inline assembly

Mapping to C++20/23 AST Types

Use `std::variant`, `std::monostate`, `std::shared_ptr<BaseNode>`, and `std::visit` to implement all AST types in a safe, type-rich, and reflection-friendly manner.
For example:

```
struct BinaryExpr {
    std::shared_ptr<Expr> left;
    Token op;
    std::shared_ptr<Expr> right;
};

using Expr = std::variant<LiteralExpr, BinaryExpr, UnaryExpr, CallExpr, VariableExpr,
↪  AssignmentExpr>;
```

Tooling Possibilities

- Syntax highlighters generated via EBNF definitions

- Parser generators for bootstrapping
- Grammar validation via ANTLR/PEG parsers
- Visual parser tree renderers
- LSP support for editor autocompletion

Conclusion

This formal grammar captures the full syntactic structure of the language. Implementers should use this as the primary contract between the frontend parser and the runtime evaluation engine. Maintaining this grammar ensures extensibility and compatibility with future enhancements.
Let me know if you'd like this grammar rendered as a downloadable `.ebnf`, `.txt`, or LaTeX appendix.

Appendix B: C++20/23 Implementation Reference

Introduction

Modern C++ (C++20 and C++23) introduces advanced features that make the development of interpreters and virtual machines more expressive, safer, and more maintainable than ever before. This appendix provides an in-depth reference on how the book's concepts are realized using cutting-edge language constructs and standard library enhancements from the last five years. It helps both readers building the interpreter from scratch and experienced C++ developers seeking modernization of legacy language tooling.
The core design of your C-style interpreted language relies on:

- Value-oriented types (`std::variant`, `std::optional`, `std::expected`)
- Declarative algorithms (`std::ranges`, `std::views`)
- Safer compile-time constructs (`consteval`, `constexpr`)
- Simplified formatting and debugging (`std::format`, `std::source_location`)
- Enhanced modularization (`modules`, `concepts`)

Type Variants with `std::variant` and `std::visit`

In a statically typed but dynamically dispatched language like the one you're building, you'll need to encode many runtime types in a single C++ type structure. This is where `std::variant` shines.

- **Why `std::variant`?**
  - Replaces `void*`-style union hacks with full type safety
  - Allows discriminated union of possible expression/value types
  - Enables pattern matching via `std::visit` instead of type checking with `dynamic_cast`
- **Use Case in Interpreter**

```
using Value = std::variant<int, float, bool, std::string, std::monostate>;

struct LiteralExpr {
    Value value;
};

struct BinaryExpr {
```

```
    ExprPtr left;
    Token op;
    ExprPtr right;
};

using Expr = std::variant<LiteralExpr, BinaryExpr, UnaryExpr>;
```

- **Evaluation via Visitor**

```
struct EvalVisitor {
    Value operator()(const LiteralExpr& expr) { return expr.value; }
    Value operator()(const BinaryExpr& expr) {
        auto lhs = std::visit(*this, *expr.left);
        auto rhs = std::visit(*this, *expr.right);
        return evalBinary(lhs, rhs, expr.op);
    }
    // ...
};
```

Optional and Expected: `std::optional`, `std::expected`

- **Use Cases**
  - **`std::optional<T>`**: For missing or non-binding values (e.g., variable not declared)
  - **`std::expected<T, E>`** (C++23): For error propagation with payloads
- **Example: Variable Lookup**

```
std::optional<Value> lookup(const std::string& name) const;
```

- **Example: Error Propagation**

```
std::expected<Value, RuntimeError> eval(const Expr& expr);
```

This pattern avoids exceptions and improves readability and testability of the interpreter's evaluation logic.

Ranges and Views: Declarative AST and Token Processing

C++20 introduces `std::ranges` and `std::views` for functional-style, lazy evaluation pipelines.

- **Example: Filtering Tokens**

```
auto identifiers = tokens | std::views::filter([](const Token& t) {
    return t.type == TokenType::Identifier;
});
```

- **Benefits**
  - No manual loops
  - Composable, testable, lazy pipelines
  - More readable parsing and token stream transformation

Pattern Matching Using `std::visit` and `if constexpr`

Although C++ does not have native `match` yet (possibly in future standards), you can simulate matching using `std::visit` and `if constexpr`.

```
std::visit([](auto&& node) {
    using T = std::decay_t<decltype(node)>;
    if constexpr (std::is_same_v<T, BinaryExpr>) {
        ...
    } else if constexpr (std::is_same_v<T, UnaryExpr>) {
        ...
    }
}, expr);
```

`std::format`: Replacing `printf`, `ostringstream`

Error messages and debug output should be formatted consistently and safely.

```
std::string err = std::format("Unexpected token '{}' at line {}", token.lexeme,
↪  token.line);
```

Advantages:

- Type-safe formatting
- Supports user-defined types
- More readable than old `printf` or stream chaining

`consteval`, `constinit`, `constexpr`: Compile-Time Language Metadata

Use `consteval` or `constexpr` to define lookup tables and keyword maps at compile time.

```
consteval auto makeKeywordMap() {
    return std::unordered_map<std::string, TokenType>{
        {"if", TokenType::If}, {"else", TokenType::Else}
    };
}
```

Use `constinit` to ensure these are initialized at compile-time, not runtime.

`std::source_location`: Diagnostic Tooling

Great for debugging and REPL error reporting.

```
void reportError(std::string_view msg, std::source_location loc =
↪  std::source_location::current()) {
    std::cerr << "Error at " << loc.file_name() << ":" << loc.line() << ": " << msg
    ↪  << "\n";
}
```

No macros needed for line/file tracking anymore.

Modules: True Modular Compilation (C++20)

C++20 introduces true modules for splitting large codebases.

```
export module lexer;
export module parser;
export module evaluator;
```

Modules reduce compile times, enforce interface boundaries, and remove header duplication.

Coroutines (Optional Use in Token Stream or Debug Tracing)

Use C++20 coroutines for lazy evaluation:

```
generator<Token> tokenize(std::string_view source);
```

You can use this for on-demand token stream or REPL input.

Concepts: Type Constraints for Extensible Evaluator Code

Modern interpreters often need to be generic, yet constrained.

```
template <typename T>
concept Numeric = std::is_integral_v<T> || std::is_floating_point_v<T>;

template <Numeric T>
Value evalAdd(T lhs, T rhs) { return lhs + rhs; }
```

`std::filesystem`: For `.lang` File Management

Used in the command-line interpreter and import systems.

```
std::filesystem::path source = "scripts/main.lang";
if (std::filesystem::exists(source)) {
    ...
}
```

Struct Bindings and Destructuring

Used heavily in parser and evaluator:

```
auto [lhs, op, rhs] = binExpr;
```

Also for `std::pair`, `std::tuple`, and `map` iterations.

VM Registers, Stack, and Memory Using STL Containers

Use STL containers over raw arrays or pointers for safety and performance:

- `std::vector<Value>` for stack
- `std::array<Value, 256>` for fixed register sets
- `std::deque<Frame>` for call stack

Compile-Time Tests Using `constexpr` and `static_assert`

Create `constexpr` interpreters or evaluators for critical testing:

```
constexpr bool testAddition() {
    return evalExpr("1 + 2") == 3;
}
static_assert(testAddition());
```

Lightweight Testing with `doctest` or `Catch2`

Use modern test frameworks to test:

- AST construction
- Expression evaluation
- Error handling

Optional Advanced Topics

- `std::atomic_ref`, `std::latch` for multithreaded REPL
- `std::stacktrace` for error reports (pending standardization)
- `std::bit_cast` for performance-sensitive VM work

Conclusion

C++20 and C++23 have revolutionized how interpreters can be implemented:

- Type-safe unions (`variant`)
- Pattern-like evaluation (`visit`)
- Compile-time maps (`consteval`)
- Modular compilation (`modules`)
- Declarative ranges and views
- Safer runtime error tracking (`source_location`, `expected`)

This appendix serves as your ultimate reference to implementing a high-quality interpreter fully in Modern C++. Use it to guide the design of future-proof, clean, and scalable tools in your language ecosystem.

Appendix C: CMake Templates for Language Projects

Overview

In building a modern interpreted or compiled programming language project, especially one using C-style syntax, maintainability, modularity, and cross-platform support are crucial. CMake is the industry-standard meta build system that generates native build scripts for platforms such as Ninja, Unix Makefiles, or Visual Studio. When paired with modern C++20/23 features, CMake helps you manage complex multi-target projects with reusable components (parser, runtime, interpreter, REPL, etc.).
This appendix presents a full-featured CMake scaffolding tailored to your language project. It covers:

- Clean modular layout for different language subsystems
- Use of latest CMake and C++20/23 features
- Separation of interpreter, REPL, testing, and tooling
- VM, bytecode, and backend extensibility
- Optional C++20 module support
- Developer experience enhancements (IDE, LSP, Clangd)
- Testing integration and CI-readiness

- Packaging, installation, and export support

1. **Directory Layout**

   Your interpreter project might look like this:

```
forgelang/

  CMakeLists.txt
  config/
      config.hpp.in
  src/
      core/
      lexer/
      parser/
      ast/
      evaluator/
      runtime/
      vm/                   # optional backend (bytecode)

  tools/
      interpreter/          # main CLI interpreter
      repl/                 # optional interactive shell

  tests/
      unit/
      integration/

  include/
      forgelang/

  examples/
       hello.lang
```

2. **Root `CMakeLists.txt` – Global Configuration**

```
cmake_minimum_required(VERSION 3.22)
project(forgelang LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

include(CTest)
enable_testing()

# Optional config header
configure_file(config/config.hpp.in config.hpp)

# Add subprojects
add_subdirectory(src)
add_subdirectory(tools)
add_subdirectory(tests)

# Install target support
include(GNUInstallDirs)
```

Modular Library Structure

Each language component is organized as an internal static library, simplifying dependency management and testing.

- **Example: `src/lexer/CMakeLists.txt`**

```
add_library(lexer STATIC
    lexer.cpp
    lexer.hpp
)

target_include_directories(lexer PUBLIC
    ${CMAKE_CURRENT_SOURCE_DIR}
    ${PROJECT_SOURCE_DIR}/include
)
```

- **src/parser/CMakeLists.txt**

```
add_library(parser STATIC
    parser.cpp
    parser.hpp
)

target_link_libraries(parser PUBLIC lexer)
target_include_directories(parser PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

You can apply the same pattern to `ast`, `evaluator`, `runtime`, and even `vm`.

Executable Targets

Interpreter, REPL, and debugging tools can be added under `tools/`.

- **tools/interpreter/CMakeLists.txt**

```
add_executable(interpreter main.cpp)
```

```
target_link_libraries(interpreter
    PRIVATE
        core
        parser
        evaluator
        runtime
)

target_include_directories(interpreter PRIVATE ${PROJECT_BINARY_DIR})
```

Optional: C++20 Module Support

If your parser or evaluator uses modules (`.ixx` files):

```
target_sources(parser
    PUBLIC
        FILE_SET CXX_MODULES TYPE CXX_MODULES FILES parser.ixx
)
```

CMake 3.25 is required for this feature. You must also use a module-aware compiler (Clang 16+, MSVC 17.5+, GCC 13+).

Testing Framework Integration

You can use `Catch2`, `Doctest`, or even GTest. Here's how to integrate Catch2:

```
include(FetchContent)
FetchContent_Declare(
  Catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
```

```
  GIT_TAG v3.4.0
)
FetchContent_MakeAvailable(Catch2)

add_executable(unit_tests
    test_main.cpp
    test_parser.cpp
    test_evaluator.cpp
)

target_link_libraries(unit_tests PRIVATE Catch2::Catch2WithMain parser evaluator)
add_test(NAME ParserTests COMMAND unit_tests)
```

Configuration Header for Versioning

- **config/config.hpp.in**

```
#pragma once

#define FORGELANG_VERSION "@PROJECT_VERSION@"
#define FORGELANG_PLATFORM "@CMAKE_SYSTEM_NAME@"
```

- **In `CMakeLists.txt`**

```
configure_file(
    config/config.hpp.in
    ${PROJECT_BINARY_DIR}/config.hpp
)
```

Then include `config.hpp` wherever version info is needed.

Cross-Platform Support

CMake enables easy platform detection:

```
if(WIN32)
    target_compile_definitions(interpreter PRIVATE PLATFORM_WINDOWS)
elseif(APPLE)
    target_compile_definitions(interpreter PRIVATE PLATFORM_MAC)
elseif(UNIX)
    target_compile_definitions(interpreter PRIVATE PLATFORM_LINUX)
endif()
```

You can also set up platform-specific includes or linker options.

Install and Package Targets

Install targets make your language distributable:

```
install(TARGETS interpreter DESTINATION bin)

install(DIRECTORY include/ DESTINATION include)

install(FILES LICENSE README.md DESTINATION share/forgelang)
```

Optional: Export Targets for Reuse

If you expect other developers to build on your interpreter libraries:

```
install(TARGETS core parser lexer ast EXPORT forgelangTargets)

install(EXPORT forgelangTargets
```

```
        FILE forgelangTargets.cmake
        NAMESPACE forgelang::
        DESTINATION lib/cmake/forgelang)
```

Developer Productivity Enhancements

- `compile_commands.json` generation for Clangd and VSCode
- Target grouping in CLion with `set_target_properties`
- Color output for Ninja builds (`-DCMAKE_COLOR_DIAGNOSTICS=ON`)
- Linting integration via `clang-tidy`, `cpplint`, `include-what-you-use`

Conclusion

The CMake templates provided in this appendix are carefully designed to:

- Scale from toy interpreters to full-featured systems
- Be modular, testable, and developer-friendly
- Work with cutting-edge C++20/23 features (modules, concepts, ranges)
- Ensure portability and professional project maintenance

With these templates, you can bootstrap your C-style language interpreter with professional-grade tooling from day one. You are encouraged to further automate build, format, and test processes via `make test`, GitHub Actions, or CMake presets.

Appendix D: Testing Strategies for Language Interpreters

Purpose of This Appendix

The goal of this appendix is to provide a professional, production-grade testing framework blueprint tailored specifically for C-style interpreted languages. From lexers to runtime evaluators, and from symbol resolution to REPL behavior, every phase of the interpreter must be thoroughly tested to ensure **correctness**, **stability**, and **long-term maintainability**.

Unlike traditional application testing, interpreter testing must simulate real user inputs, malformed constructs, dynamic execution, variable state tracking, and memory behavior—making it a highly dynamic and challenging task. Leveraging modern C++20/23 makes this more expressive and manageable, thanks to new tools, libraries, and features that improve test composition, runtime introspection, and compiler-time checks.

Categories of Tests in a Language Interpreter

To organize your testing approach effectively, break down the interpreter into testable layers:

1. **Unit Testing**

   What it covers:

   - Lexical analyzer (tokenizer)
   - Parser components (expression parsing, statement parsing)
   - AST node construction

- Value system types (`int`, `float`, `bool`, `string`, `array`)
- Type conversions and operations
- Symbol table and scope resolution
- Evaluator functions (value application, control flow, returns)

Modern C++20/23 techniques:

- `constexpr`-enabled functions and compile-time validation
- Use of `std::variant` and `std::visit` to test dynamic values
- Concept-based assertions (e.g., validating that a value meets `Arithmetic` or `StringLike` traits)

Typical Unit Test Case (Catch2):

```
TEST_CASE("Lexer tokenizes identifiers and operators") {
    Lexer lexer("a + b");
    std::vector<Token> tokens = lexer.tokenize();
    REQUIRE(tokens[0].type == TokenType::Identifier);
    REQUIRE(tokens[1].type == TokenType::Plus);
}
```

2. **Integration Testing**

Tests that involve **multiple subsystems** together:

- Lexer + Parser + AST
- Parser + Evaluator
- Full program execution (e.g., `int main() { return 5; }`)

Design Strategy:

- Run `.lang` files through the complete interpreter stack
- Compare the final value or output stream
- Simulate expected user-written programs

```
TEST_CASE("Simple C-style program executes correctly") {
    Interpreter interp;
    auto result = interp.run("int x = 3; int y = x + 2; y;");
    CHECK(result.as_int() == 5);
}
```

3. **Regression Testing**

   Used to **preserve past fixes** and prevent breaking behaviors during refactors or feature additions.

   Method:

   - Maintain a directory `tests/regression/` with previously failing programs
   - Add test expectations (result, output, error code)
   - Automatically verify the outputs match previous working states

   Example Structure:

```
tests/
  regression/
     bug123_missing_semicolon.lang
     bug123.expected_output
```

4. **Golden File Testing**

   For scenarios where output is **too verbose** for inline testing (e.g., AST structure, error messages, debug dumps).

   Implementation:

   - Store expected output in `.golden` files
   - Compare using line-by-line or hash-based verification
   - Normalize whitespace before comparing

```
std::string actual = interpreter.run("source.lang");
std::string expected = read_file("source.golden");
CHECK(normalize(actual) == normalize(expected));
```

5. **REPL Testing and Simulation**

   Testing the **interactive shell** (REPL) is different:

   - Simulate terminal I/O (input prompts, command history)
   - Redirect `stdin` and `stdout` streams
   - Batch process inputs as if typed live

   Tooling:

   - Use `std::stringstream` for fake user inputs
   - Capture REPL outputs using redirect macros

Using Modern C++20/23 in Tests

C++20/23 brings new capabilities to your test suite:

- **Modules** for isolating test interfaces
- **Concepts** to constrain test inputs
- **`constexpr` functions** for compile-time tests
- **`std::ranges`** to filter test data
- **`std::format`** for precise debug output
- **Structured bindings** for improved assertions

```
template<typename T>
concept ValidValue = requires(T v) {
    { v.to_string() } -> std::convertible_to<std::string>;
};

static_assert(ValidValue<IntValue>);
```

Project Directory Layout for Testing

```
project_root/
  src/
  tests/
     unit/
     integration/
     regression/
```

```
    golden/
    repl/
  expected/
  CMakeLists.txt
```

Keep test inputs, expected outputs, and logs organized for long-term scalability.

Continuous Integration and Automation

Use modern CI platforms to automate test runs:

- GitHub Actions
- GitLab CI/CD
- Azure Pipelines
- Local automation using `ctest`

Example CMake + Catch2 Setup:

```
enable_testing()

add_executable(all_tests
    test_lexer.cpp
    test_parser.cpp
    test_eval.cpp
)

target_link_libraries(all_tests PRIVATE Catch2::Catch2WithMain)

add_test(NAME InterpreterTests COMMAND all_tests)
```

Add nightly jobs to run:

```
- name: Run All Tests
  run: |
    mkdir build && cd build
    cmake ..
    make
    ctest --output-on-failure
```

Memory Checking and Leak Detection

You must validate memory safety:

Tools:

- **AddressSanitizer (ASan)**: Detects overflows, invalid accesses
- **Valgrind**: Full memory profiler
- **LeakSanitizer (LSan)**: Memory leak detector
- **MSVC Debug CRT**: For Windows

Enable with:

```
add_compile_options(-fsanitize=address)
add_link_options(-fsanitize=address)
```

Coverage Metrics

Coverage helps understand what is not tested:

- Use `gcov`, `lcov`, `llvm-cov` or `OpenCppCoverage`
- Integrate into CI with HTML reports
- Target high coverage on parser, evaluator, and runtime logic

```
gcovr -r . --html-details -o coverage.html
```

Functional vs Structural Testing

- **Functional**: Check that final values/output match user expectation
- **Structural**: Validate the internal structure of generated AST, symbol table, or execution frames

```
CHECK(ast.to_string() == "(+ 1 2)");
CHECK(frame.lookup("x").type == Type::Int);
```

Handling Invalid Programs

Always include **failing test cases**:

- Missing semicolon
- Undeclared variable
- Type mismatch
- Scope violation

```
CHECK_THROWS_AS(interpreter.run("int x; x = true;"), TypeError);
```

Snapshot Testing

Snapshot testing tools (like ApprovalTests) can be used to:

- Dump intermediate stages (AST, symbol table)
- Validate that internal compiler/interpreter state hasn't changed unexpectedly

Fuzzing and Property-Based Testing

- Use generators to create random valid/invalid programs
- Test properties like idempotency, equivalence, and determinism
- Catch edge cases that developers rarely write

Conclusion

Testing a C-style interpreter is a **systemic effort**. Every phase—from lexing to evaluation, scope resolution to function calls—must be validated through automated, reproducible, and scalable tests. By utilizing the strengths of modern C++20/23, and structuring tests around unit, integration, and regression layers, your language infrastructure becomes durable and production-ready.
This appendix equips you with a blueprint for building a **robust test ecosystem**—one that grows alongside your language implementation and protects it from regressions, instability, and undetected edge cases.

Appendix E: Performance Benchmarking for Interpreters

Overview

Performance benchmarking is not a luxury—it's a necessity in any interpreter design intended for real-world use. Although interpreters inherently execute slower than compiled code due to their runtime nature, modern techniques, careful engineering, and the new capabilities introduced in **C++20 and C++23** allow us to achieve highly responsive and optimized behavior.
This appendix provides a **methodical blueprint** for integrating benchmarking directly into the lifecycle of the interpreter. It covers:

- Measuring latency and throughput for specific constructs
- Managing memory usage awareness
- Detecting regressions early through automated benchmarking
- Validating architectural improvements such as caching, precompilation, or just-in-time optimizations
- Organizing interpretable performance data for review and visualization

Benchmarking Goals for Interpreters

The benchmarking system aims to answer questions like:

- How fast can an arithmetic expression be parsed and executed?
- How does nested scope resolution impact performance?

- Does function call overhead scale linearly with depth?
- How much memory is consumed during iterative vs. recursive execution?
- What is the overhead of symbol resolution within blocks and functions?

With benchmarks in place, you gain:

- **Performance baselines** for future optimizations
- **Comparison metrics** between interpreter versions
- **Confidence** that new features do not introduce performance regressions
- **Insight** into bottlenecks across the interpreter lifecycle

Benchmark Categories

Below are essential categories of benchmarks relevant to C-style interpreted languages:

Table 4-4: Benchmark Areas for Language Evaluation

Benchmark Area	Description
Expression Evaluation	Integer/float expression parsing and evaluation
Variable Resolution	Symbol lookup in nested scopes and environments
Function Invocation	Direct and recursive function calls, argument binding
Loop Control Flow	Evaluation of `for`, `while`, and `do-while` constructs
Conditionals	Evaluation cost of `if-else` branches and block execution

Benchmark Area	Description
Scoping Cost	Overhead of entering/exiting nested `{}` blocks
File Execution	Running `.lang` scripts (short and long) to simulate real usage
Memory Allocation	Performance impact of allocating arrays, objects, or strings

Benchmark Harness Design

C++20 makes it significantly easier to build a concise and flexible benchmarking toolset.

1. **Structuring a Benchmark Unit**

```
struct Benchmark {
    std::string name;
    std::function<void()> code;

    void run() const {
        using Clock = std::chrono::high_resolution_clock;
        auto start = Clock::now();
        code();
        auto end = Clock::now();

        auto time = std::chrono::duration_cast<std::chrono::microseconds>(end -
        ↪  start).count();
        std::cout << std::format("{:<40} {:>10}  s\n", name, time);
    }
};
```

2. **Example Usage**

```
Benchmark{"Basic Expression Evaluation", [] {
    Interpreter interp;
    interp.run("int a = 5 * (3 + 2);");
}}.run();
```

For repeated runs:

```
void runRepeated(const Benchmark& bench, int iterations = 1000) {
    using Clock = std::chrono::high_resolution_clock;
    std::chrono::microseconds total{0};

    for (int i = 0; i < iterations; ++i) {
        auto start = Clock::now();
        bench.code();
        auto end = Clock::now();
        total += std::chrono::duration_cast<std::chrono::microseconds>(end -
        ↪  start);
    }

    auto avg = total.count() / iterations;
    std::cout << std::format("{:<40} {:>10}  s (avg over {} runs)\n",
    ↪  bench.name, avg, iterations);
}
```

Benchmark Script Corpus

A good benchmarking system depends on a **diverse and growing set** of test programs:

1. **Microbenchmarks**

```
int x = 5 + 3 * 2;   // Arithmetic expression
langCopyEditint sum = 0;
for (int i = 0; i < 10000; ++i) {
    sum = sum + i;
}
langCopyEditint fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
fib(10);
```

2. **Macrobenchmarks**

   These include real `.lang` files that simulate:

   - Configuration file parsing
   - Text processing
   - Simple mathematical modeling
   - CLI tool scripting

Memory Benchmarking

Memory usage should be monitored alongside execution time.

1. **Manual Estimation (Tracking Allocations)**

   If your interpreter uses `std::shared_ptr`, `std::vector`, or dynamic memory:

   - Use **custom allocators** that count memory blocks.

- Use wrappers around allocation points (e.g., AST node creation).

2. **Platform-Based Measurement**

   Use:

   - `getrusage` (Linux/macOS)
   - `GlobalMemoryStatusEx` (Windows)
   - Tools like `valgrind`, `massif`, or `heaptrack`

   Or wrap memory introspection:

```
struct MemoryTracker {
    size_t before = get_memory_usage();

    ~MemoryTracker() {
        size_t after = get_memory_usage();
        std::cout << "Memory used: " << (after - before) << " bytes\n";
    }
};
```

Long-Running Benchmark Patterns

For sustained usage, simulate long scripts and ensure:

- No memory leaks
- Stack growth remains controlled
- Repeated runs do not degrade performance

```
std::string program = read_file("benchmark.lang");
for (int i = 0; i < 100; ++i) {
    Interpreter interp;
    interp.run(program);
}
```

Benchmark Output Logging

Generate structured logs:

```
std::ofstream log("benchmark_results.csv");
log << "test_name,time_us\n";
log << "Arithmetic,10\n";
log << "Loop_10000,1500\n";
```

This enables further analysis and graphing using:

- Python (matplotlib, pandas)
- Excel
- R
- Grafana + Prometheus (CI integration)

Regression Detection

Store prior benchmark data:

```
struct RegressionCheck {
    std::map<std::string, long> baseline;
```

```
    void compare(const std::string& name, long new_time) {
        long base = baseline[name];
        if (new_time > base * 1.2) {
            std::cerr << " ! Performance regression in " << name << ": " << new_time
            ↪  << " > " << base << "\n";
        }
    }
};
```

Integration in Build System and CI

Extend your **CMake build**:

```
add_executable(lang_benchmarks benchmark.cpp)
target_link_libraries(lang_benchmarks PRIVATE interpreter_lib)
```

Use GitHub Actions or GitLab CI:

```
- name: Run Performance Benchmarks
  run: |
    cmake --build .
    ./lang_benchmarks > output.csv
```

Interpretation vs Compilation: Bridging the Gap

Over time, your interpreter may evolve into:

- A bytecode virtual machine
- A JIT-accelerated runtime

- A compiled backend

By benchmarking today, you ensure that future stages remain **measurably faster**, not just **theoretically cleaner**.

Conclusion

Performance benchmarking is the language designer's telescope: it reveals invisible limitations, detects regressions, and enables confident scaling.
By applying modern C++ tools, CMake integration, structured testing, and metrics gathering, you turn your interpreter into a professional-grade execution engine that can evolve toward compilation, JIT, or hybrid models without sacrificing quality or predictability.
This appendix provides the **performance DNA** for any serious interpreted language project.

Appendix F: Language Extension Ideas and Future Work

Introduction

A successful programming language is not just defined by its current capabilities but by how well it prepares for growth. The language described throughout this book begins with a foundational C-style syntax, emphasizing clarity, performance, and simplicity. However, real-world use cases inevitably push a language beyond its initial boundaries. This appendix presents a forward-looking blueprint for extending the language into a more powerful and expressive system, while staying true to its C-style heritage and

relying on robust and modern C++20/23 constructs. These ideas are grouped into six core areas:

1. **Syntax-Level Enhancements**
2. **Semantic Capabilities and Type System Extensions**
3. **Runtime Enhancements and Libraries**
4. **Tooling, Debugging, and Ecosystem Growth**
5. **Deployment and Embedding**
6. **Long-Term Vision for Compilation**

Each section includes actionable recommendations, architectural impact, and how modern C++ enables these future paths efficiently.

Syntax-Level Enhancements

1. **Enums and Type Constants**

   Enums simplify logic readability and debugging. Implementing enums involves extending the type system to associate symbolic names with integer values, allowing constructs like:

   ```
   enum Status { OK = 0, ERROR = 1, TIMEOUT = 2 };
   Status s = OK;
   ```

   C++ Implementation Notes:

   - Use `std::unordered_map<std::string, int>` for enum definition storage.

- Extend the parser to recognize enum blocks and store constants in the symbol table.
- Support implicit conversion from enum to integer for evaluation consistency.

2. **Type Aliasing (`typedef`, `using`)**

   For readability and reusability:

```
typedef int Age;
```

   Interpreter Enhancement:

   - Maintain a type alias map in the parser or type checker stage.
   - Replace aliases during parsing or store them as metadata for error reporting.

3. **Inline Annotations and Attributes**

   Add optional metadata for variables, functions, or types:

```
@deprecated
func oldApi() {
    // ...
}
```

   Applications:

   - Debug tooling
   - Warning generation
   - Optimizer hints

   Modern C++ Use:

- Attributes such as `[[nodiscard]]`, `[[likely]]`, `[[nodiscard]]` inspire a clear structure.

Semantic Capabilities and Type System Extensions

1. **Closures and Lexical Environments**

   Enable functions to capture outer variables:

```
func makeCounter() {
    int count = 0;
    return func() { count = count + 1; return count; };
}
```

   Requirements:

   - Functions must hold a reference to their enclosing environment.
   - Environments must be copyable or shareable across call boundaries.

   C++ Implementation:

   - `std::shared_ptr<Environment>` ensures safe shared lexical environments.
   - C++20's `move-only` closures (with `std::move_only_function`) simplify stateful closure handling.

2. **Exception Handling**

   Support structured error management:

```
try {
    riskyOperation();
} catch (e) {
    print("Caught error:", e);
}
```

Interpreter Structure:

- Error context stack with catchable values.
- Long jump or exception-like unwind logic.

C++ Tools:

- Use `std::exception_ptr` or custom `InterpreterException` types.
- Manage control flow using `std::optional` or variants to propagate success/error states.

3. **User-Defined Structs and Methods**

Enable structured types:

```
struct Point {
    int x;
    int y;
    func move(dx, dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Data Model:

- Composite `Value::Struct` type with field map.
- Method table for dynamic dispatch or static inline binding.

Runtime Enhancements and Library Features

1.

2. **Standard Modules**

   Define standard libraries such as:

   - `math`: `abs`, `sqrt`, `sin`, `pow`
   - `time`: `now()`, `sleep(ms)`
   - `string`: `split`, `join`, `find`

   Implementation:

   - Add internal native functions via function registration.
   - Use C++20 `<chrono>`, `<cmath>`, and `<string_view>` for fast, efficient mappings.

3. **Reflection and Introspection**

   Enable:

   ```
   print(typeOf(x));
   print(getMembers(someStruct));
   ```

   Usage:

   - Debugging

- Meta-programming
- Serializing/deserializing

C++ Backing:

- Maintain runtime `ValueMeta` attached to each object.
- Use C++20's `std::source_location` for internal introspection support.

Tooling, Debugging, and Ecosystem Growth

1. **Debugger Integration**

   Provide:

   - Step execution
   - Breakpoints
   - Stack trace display

   C++ Side:

   - Wrap each instruction in traceable execution blocks.
   - Track `source_line` and `source_file` inside AST nodes or IR.

2. **Language Server Protocol (LSP)**

   LSP allows integration into modern IDEs like VSCode or CLion. To support:

   - Symbol resolution
   - Syntax diagnostics
   - Auto-completion

Architecture:

- Export the parser and type checker as callable services.
- Provide a JSON-over-IPC interface.

3. **Testing Tools and Coverage Reports**

   Expose runtime test harness:

```
@test
func testAdd() {
    assert(add(1, 2) == 3);
}
```

   Advanced:

   - Store test coverage in bitmaps or AST annotations.
   - Emit performance metrics during testing.

Deployment and Embedding

1. **WebAssembly Target**

   By ensuring POSIX-free, file-independent behavior, the interpreter can be compiled via Clang to `.wasm`.

   Steps:

   - Avoid C-style I/O.
   - Use C++20 standard I/O abstraction.
   - Add virtual file and memory systems.

2. **Native Embedding**

   Make the interpreter usable as a scripting backend:

   ```
   LangVM vm;
   vm.load("config.lang");
   vm.call("initialize");
   ```

   Embed in:

   - Games
   - GUIs
   - Embedded devices

   Use a modular C++ API to expose internal functions, error management, and memory context.

Long-Term Vision: Compilation

The language could evolve into a fully compiled one:

- Target **LLVM IR**
- Generate **intermediate bytecode**
- Use register allocation and JIT techniques

Approach:

- Compile AST to SSA or bytecode.
- Use type inference or a type checker for optimization.
- Generate `.obj`/`.o` or `.wasm` using LLVM backend.

Final Thoughts

The roadmap here is not just a set of ”nice-to-have” features. Each item reflects a necessary response to modern language design requirements. With clean architecture and a strong C++20/23 foundation, this language is well-positioned to become a complete development platform, supporting:

- Professional development teams
- Educational systems
- Embedded domains
- High-performance scripting environments

These ideas aim to sustain the language for the next decade and beyond, encouraging contributors, learners, and power users to adopt and extend its capabilities.

References

Books on Programming Language Design and Implementation

- **"Crafting Interpreters" by Robert Nystrom**
  A highly practical guide to writing interpreters, with clear implementation paths for both tree-walk and bytecode VMs. Fundamental to understanding interpreter structures.

- **"Programming Language Pragmatics" by Michael L. Scott**
  A comprehensive academic textbook that explores both theory and implementation of programming languages.

- **"Engineering a Compiler" by Keith D. Cooper and Linda Torczon**
  Ideal for understanding compiler design from IRs to optimization. Especially useful when extending the interpreter into a compiler or bytecode VM.

- **"Modern Compiler Implementation in C" by Andrew W. Appel**
  A classic that helps grasp practical compiler design in C, which aligns well with your C-style syntax design goals.

- **"Types and Programming Languages" by Benjamin C. Pierce**

Excellent for readers who want to understand type systems and type checking at a theoretical and practical level.

- **“Language Implementation Patterns” by Terence Parr**
  A pragmatic book filled with techniques for building parsers, lexers, ASTs, and error handling systems.

Books on Modern C++ and Advanced Features (C++20/C++23)

- **“Effective Modern C++” by Scott Meyers**
  Key insights into modern C++ idioms and safety. Especially relevant for applying `std::move`, `auto`, `unique_ptr`, etc.

- **“C++20: The Complete Guide” by Nicolai Josuttis**
  Explains in detail the modern features like ranges, concepts, and coroutines, which you can use for implementing interpreters and REPL systems.

- **“C++ Templates: The Complete Guide (2nd Edition)” by David Vandevoorde, Nicolai Josuttis, and Doug Gregor**
  For mastering templates, generic programming, and building reusable parsing or AST libraries.

- **“Clean Code in C++” by Stephan Roth**
  Reinforces modern software engineering principles while developing complex C++ systems.

Research Papers and Academic Sources

- **"The Next 700 Programming Languages" by Peter J. Landin (1966)**
  A foundational theoretical paper inspiring many modern interpreters and DSLs.
- **"Structure and Interpretation of Computer Programs (SICP)" by Abelson and Sussman**
  Not C-style, but this book offers valuable insights into evaluation strategies, environments, and closures.
- **"An Efficient Interpreter for a Procedural Language" – Mitchell Wand**
  Lays out an early stack-based interpreter design for procedural languages similar to C.
- **"A Correspondence Between Continuation Passing Style and Static Single Assignment Form" – Appel & Flanagan (1992)**
  Useful if your interpreter aims to generate IRs or transitions into a compiled form in future iterations.

Tools and Libraries Consulted or Recommended

- **LLVM Project Documentation**
  For understanding modular compiler and VM design, including frontend parser and IR-level design ideas.
- **Clang AST Internals**
  Used as inspiration for building your own AST node hierarchy and traversal utilities.
- **GoogleTest & Catch2**

Frameworks used in testing interpreter correctness (see Appendix D).

- **ANTLR (ANother Tool for Language Recognition)**
  While not used in this book, its theory of recursive descent and parser generators influenced parser design strategy.

- **Valgrind and AddressSanitizer (ASan)**
  Tools used to test runtime memory behavior in interpreters (especially useful in Appendices D and E).

- **CMake (version 3.20+) and Modern CMake Practices**
  Key for the build system and project modularity shown in Appendix C.

Online Courses and Video Lectures

- **CS143: Compilers (Stanford University, Prof. Alex Aiken)**
  For foundational theory in compiler and interpreter implementation.

- **MIT 6.001 and 6.035 – Structure and Design of Programming Languages**
  Provide strong grounding in recursive evaluation and design of language interpreters.

Other Influential Open-Source Interpreters and Languages

- **Wren Language (by Bob Nystrom)** – A small, fast class-based scripting language. Its VM structure inspired parts of Chapter 23.

- **Zig Language** – Not interpreted, but its simplicity and build system influenced language clarity goals.
- **Lua Interpreter (PicoLua)** – Extremely lightweight interpreter. Its error handling and memory design informed Appendices D and E.
- **CPython Internals** – Source of many insights into memory handling, symbol tables, and debugging outputs.

Final Note

The book aims to be **self-contained**, but it was influenced deeply by the above resources. Readers who seek to dive deeper into parsing algorithms, interpreter theory, or low-level language design will find the above references highly useful.