Book of Verse

Book of Verse

The Verse Programming Language by Tim Sweeney

Documentation by the Verse Team — Epic Games

Book of Verse

Documentation for the Verse Programming Language

Source: `https://github.com/verselang/book`

Print-Ready Edition with Crop Marks

Contents

Chapter 1

Preface

Warning

This is an early draft of the Book of Verse. Suggestions for improvements are welcome. Frequent updates are to be expected.

This documentation provides an in-depth look at the Verse programming language, its philosophy, and core concepts.

Verse is a multi-paradigm programming language developed by Epic Games, drawing from functional, logic, and imperative traditions to create a coherent system for building metaverse experiences.

Verse has three core principles:

- **It's just code** - Complex concepts are expressed as primitive Verse constructs
- **Just one language** - Same constructs for compile-time and run-time
- **Metaverse first** - Designed for a global simulation environment

Note

The documentation pertains to the head of the main development branch of Verse, some features may be discussed before they are officially released and are thus subject to change. Some Epic internal features may also be discussed.

1.1 Documentation Sections

- Overview - Introduction to Verse philosophy and features
- Expressions - Everything is an expression paradigm
- Primitives - Integers, floats, rationals, logic, strings, and special types

- Containers - Optionals, tuples, arrays, maps, and weak maps
- Operators - Arithmetic, comparison, logical, and assignment operators with precedence
- Mutability - Mutable variables, references, and state management
- Functions - Open-world vs closed-world functions, parameters, and return values
- Control Flow - If/else, loops, code blocks, and comments
- Failure System - First-class failure, failable expressions, and speculative execution
- Structs & Enums - Value types and fixed sets of named values
- Classes & Interfaces - Object-oriented programming with inheritance and contracts
- Type System - Types as functions and type checking
- Access Specifiers - Public, private, and protected visibility
- Effects - Effect families, specifiers, and capability declarations
- Concurrency - Structured concurrency with sync, race, rush, branch, and spawn
- Live Variables - Reactive values that automatically update
- Modules & Paths - Code organization and the global namespace
- Persistable Types - Types that can be saved and loaded
- Code Evolution - Versioning and backward compatibility

Part I

Part I: Fundamentals

Chapter 2

Chapter 1: Overview

2.1 Overview

Verse is a multi-paradigm programming language developed by Epic Games for creating gameplay in Unreal Editor for Fortnite and building experiences in the metaverse. Drawing from functional, logic, and imperative traditions, Verse represents a departure from traditional programming languages, designed not just for today's needs but with a vision spanning decades or even centuries into the future.

Verse is built on three fundamental principles:

- **It's Just Code**: Complex concepts that might require special syntax or constructs in other languages are expressed as regular Verse code. There's no magic—everything is built from the same primitive constructs, creating a uniform and predictable programming model.
- **Just One Language**: The same language constructs work at both compile-time and run-time. There is no preprocessor. What you write is what executes, whether during compilation or at runtime.
- **Metaverse First**: Verse is designed for a future where code runs in a single global simulation—the metaverse. This influences every aspect of the language, from its strong compatibility guarantees to its effect system that tracks side effects and ensures safe concurrent execution.

Verse aims to be:

- **Simple enough** for first-time programmers to learn, with consistent rules and minimal special cases.
- **Powerful enough** for complex game logic and distributed systems, with advanced features that scale to large codebases.

- **Safe enough** for untrusted code to run in a shared environment, with strong sandboxing and effect tracking.
- **Fast enough** for real-time games and simulations, with an implementation that can optimize pure computations aggressively.
- **Stable enough** to last for decades, with strong backward compatibility guarantees and careful evolution.

Why Verse?

Traditional programming languages carry decades of historical baggage and design compromises. Verse starts fresh, learning from the past but not bound by it. It's designed for a future where:

- Code lives forever in a persistent metaverse
- Millions of developers contribute to a shared codebase
- Programs must be safe, concurrent, and composable by default
- Backward compatibility is not optional but essential
- The boundary between compile-time and runtime is fluid

Ready to dive in? Start with Built-in Types to understand Verse's fundamental data types, or jump to Expressions to see how everything in Verse computes values.

For experienced programmers coming from other languages, the Failure System and Effects sections highlight some of Verse's distinctive features.

2.2 Key Features

Everything is an Expression

In Verse, there are no statements—everything is an expression that produces a value. This creates a highly composable system where any piece of code can be used anywhere a value is expected.

```
# Even control flow produces values
Result := if (Condition[]) then "yes" else "no"

# Loops are expressions
Multiply := for (X : Array) { X * 42 }
```

Failure as Control Flow

Instead of boolean conditions and exceptions, Verse uses failure as a primary control flow mechanism. Expressions can succeed (producing a value) or fail (producing no value), enabling natural control flow patterns:

```
ValidateInput[Data] # Square brackets indicate that this function may fail
ProcessData(Data)   # Data is only processed if valid, parentheses mean must succeed
```

See Failure for complete details on failable expressions and failure contexts, and Control Flow for if expressions.

Strong Static Typing with Inference

Verse features a powerful type system that catches errors at compile time while minimizing the need for type annotations through inference. See Types for complete details on the type system and subtyping.

```
X := 42                    # Type inferred
Name := "Verse"            # Type inferred
```

Effect Tracking

Functions declare their side effects through specifiers like `<computes>`, `<reads>`, `<writes>`, `<transacts>`, `<decides>`, and `<suspends>`. These effect specifiers make it immediately clear what a function can do beyond computing its return value:

```
PureCompute()<computes>:int = 2 + 2            # No side effects
ReadState()<reads>:int = GetCurrentValue()     # Can read mutable state
UpdateGame()<transacts>:void = set Score += 10 # Can read, write, allocate
```

See Effects for complete details on the effect system.

Built-in Concurrency

Concurrency is a first-class feature with structured concurrency primitives that make concurrent programming safe and predictable.

```
# Run tasks concurrently and wait for all
sync:
    TaskA()
    TaskB()
    TaskC()

# Race tasks and take first result
race:
    FastPath()
    SlowButReliablePath()
```

Speculative Execution

Verse can speculatively execute code and roll back changes if the execution fails, enabling powerful patterns for validation and error handling.

```
if (TryComplexOperation[]):
    # Changes performed by TryComplexOperation[] are committed
else:
    # Changes are rolled back automatically
```

Reactive Programming with Live Variables

Verse provides first-class support for reactive programming through live variables that automatically recompute when their dependencies change, decreasing the need for manual event handling.

```
var MaxHealth:int = 100
var Damage:int = 0
var live Health:int = MaxHealth - Damage

# Health automatically updates when dependencies change
set Damage = 20      # Health becomes 80
set MaxHealth = 150  # Health becomes 130

# Reactive constructs for event handling
when(Health < 25):
    Log("Low health warning!")
```

Welcome to Verse—a language built not just for today's games, but for tomorrow's metaverse.

2.3 An Example

Let's explore the language with an example that demonstrates its key features. We'll build an inventory management system for a game, showing how Verse's constructs come together to create robust, maintainable code.

```
# Module declaration - start by importing utility functions
using { /Verse.org/VerseCLR }

# Define item rarity as an enumeration - showing Verse's type system
item_rarity := enum<persistable>:
    common
    uncommon
    rare
    epic
    legendary

# Struct for immutable item data - functional programming style
```

```
item_stats := struct<persistable>:
    Damage:float = 0.0
    Defense:float = 0.0
    Weight:float = 1.0
    Value:int = 0

# Class for game items - object-oriented features with functional constraints
game_item := class<final><persistable>:
    Name:string
    Rarity:item_rarity = item_rarity.common
    Stats:item_stats = item_stats{}
    StackSize:int = 1

    # Method with decides effect - can fail
    GetRarityMultiplier()<decides>:float =
        case(Rarity):
            item_rarity.common => 1.0
            item_rarity.uncommon => 1.5
            item_rarity.rare => 2.0
            item_rarity.epic => 3.0
            _ => false  # Fails if the item is legenday or unexpected

    # Computed property using closed-world function
    GetEffectiveValue()<transacts><decides>:int=
        Floor[Stats.Value * GetRarityMultiplier[]]

# Inventory system with state management and effects
inventory_system := class:
    var Items:[]game_item = array{}
    var MaxWeight:float = 100.0
    var Gold:int = 1000

    # Method demonstrating failure handling and transactional semantics
    AddItem(NewItem:game_item)<transacts><decides>:void =
        # Calculate new weight - speculative execution
        CurrentWeight := GetTotalWeight()
        NewWeight := CurrentWeight + NewItem.Stats.Weight

        # This check might fail, rolling back any changes
        NewWeight <= MaxWeight

        # Only executes if weight check passes
```

```
        set Items += array{NewItem}
        Print("Added {NewItem.Name} to inventory")

    # Method with query operator and failure propagation
    RemoveItem(ItemName:string)<transacts><decides>:game_item =
        var RemovedItem:?game_item = false
        var NewItems:[]game_item = array{}

        for (Item : Items):
            if (Item.Name = ItemName, not RemovedItem?):
                set RemovedItem = option{Item}
            else:
                set NewItems += array{Item}
        set Items = NewItems
        RemovedItem?  # Fails if item not found

    # Purchase with complex failure logic and rollback
    PurchaseItem(ShopItem:game_item)<transacts><decides>:void =
        # Multiple failure points - any failure rolls back all changes
        Price := ShopItem.GetEffectiveValue[]
        Price <= Gold  # Fails if not enough gold

        # Tentatively deduct gold
        set Gold = Gold - Price

        # Try to add item - might fail due to weight
        AddItem[ShopItem]

        # All succeeded - changes are committed
        Print("Purchased {ShopItem.Name} for {Price} gold")

    # Higher-order function with type parameters and where clauses
    FilterItems(Predicate:type{_(:game_item)<decides>:void}):[]game_item =
        for (Item : Items, Predicate[Item]):
            Item

    GetTotalWeight()<transacts>:float =
        var Total:float = 0.0
        for (Item : Items):
            set Total += Item.Stats.Weight
        Total

```

```
# Player class using composition
player_character<public> := class:
    Name<public>:string
    var Level:int = 1
    var Experience:int = 0
    var Inventory:inventory_system = inventory_system{}

    LevelUpThreshold := 100

    GainExperience(Amount:int)<transacts>:void =
        set Experience += Amount

        # Automatic level up check with failure handling
        loop:
            RequiredXP := LevelUpThreshold * Level
            if (Experience >= RequiredXP):
                set Experience -= RequiredXP
                set Level += 1
                Print("{Name} leveled up to {Level}!")
            else:
                break

    # Method showing qualified access
    EquipStarterGear()<transacts><decides>:void =
        StarterSword := game_item{
            Name := "Rusty Sword"
            Rarity := item_rarity.common
            Stats := item_stats{Damage := 10.0, Weight := 5.0, Value := 50}
        }
        # These might fail if inventory is full
        Inventory.AddItem[StarterSword]

# Example usage demonstrating control flow and failure handling
RunExample<public>()<suspends>:void =
    # Create a player (can't fail)
    Hero := player_character{Name := "Verse Hero"}

    # Try to equip starter gear (might fail)
    if (Hero.EquipStarterGear[]):
        Print("Hero equipped with starter gear")

    # Demonstrate transactional behavior
```

```
    ExpensiveItem := game_item{
        Name := "Golden Crown"
        Rarity := item_rarity.epic
        Stats := item_stats{Value := 2000, Weight := 90.0}  # Very heavy!
    }

    # This might fail due to weight or insufficient gold
    if (Hero.Inventory.PurchaseItem[ExpensiveItem]):
        Print("Purchase successful!")
    else:
        Print("Purchase failed - gold remains at {Hero.Inventory.Gold}")

    # Use higher-order functions with nested function predicate
    IsRareOrLegendary(I:game_item)<decides>:void =
        I.Rarity = item_rarity.rare or I.Rarity = item_rarity.legendary

    RareItems := Hero.Inventory.FilterItems(IsRareOrLegendary)

    Print("Found {RareItems.Length} rare items")
```

This example showcases Verse in a practical context. Let's explore what makes this code uniquely Verse:

Type System and Data Modeling

The example begins with Verse's rich type system. Types flow naturally through the code; many type annotations are omitted as they can be infered. When we do specify types, like `Items:[]game_item`, they document intent rather than just satisfy the compiler. The `item_rarity` enum provides type-safe constants without the boilerplate of traditional enumerations. The `item_stats` struct marked as `<persistable>` can be saved and loaded from persistent storage, essential for game saves. The `game_item` class uses `<unique>` to ensure reference equality semantics.

Failure as Control Flow

Throughout the code, failure drives control flow rather than exceptions or error codes. The `<decides>` effect marks functions that can fail, and failure propagates naturally through expressions. When `GetRarityMultiplier()` encounters an unknown rarity, it doesn't throw an exception or return a sentinel value - it simply fails, and the calling code handles this gracefully. The `AddItem` method demonstrates how failure creates elegant validation. The expression `NewWeight <= MaxWeight` either succeeds (allowing execution to continue) or fails (preventing the item from being added). There's no explicit control flow - just a declarative assertion of what must be true.

Transactional Semantics and Speculative Execution

Methods marked with `<transacts>` provide automatic rollback on failure. In `PurchaseItem`, we deduct gold from the player, then try to add the item. If adding fails (perhaps due to weight limits), the gold deduction is automatically rolled back. This eliminates entire categories of bugs related to partial state updates. This transactional behavior extends to complex operations. When multiple changes need to succeed or fail together, Verse ensures consistency without need for manual clean up.

Functions as First-Class Values

The `FilterItems` method accepts a predicate function, demonstrating higher-order programming. The nested function `IsRareOrLegendary` in `RunExample` shows how functions can be defined locally and passed around like any other value. This functional programming style combines naturally with the imperative and object-oriented features.

Optional Types and Query Operators

The inventory removal logic uses optional types (`?game_item`) to represent values that might not exist. The query operator `?` extracts values from options, failing if the option is empty. This eliminates null pointer exceptions while providing convenient syntax for handling absent values.

Pattern Matching and Control Flow

The `case` expression in `GetRarityMultiplier` demonstrates pattern matching. Unlike a switch statement, `case` is an expression that produces a value. The underscore `_` provides a catch-all pattern, though in this example it leads to failure. The `if` expression similarly produces values and can bind variables in its condition. The compound conditions show how multiple operations can be chained with automatic failure propagation.

Module System and Access Control

The code begins with `using` statements that import functionality from other modules. The path-based module system ensures that dependencies are unambiguous and permanently addressable. Access specifiers like `<public>` control visibility at a fine-grained level.

Immutable by Default

Data structures are immutable unless explicitly marked with `var`. This eliminates large classes of bugs and makes concurrent programming safer. When we do need mutation, it's explicit and tracked by the effect system. See Mutability for complete details on `var` and `set`.

2.4 Naming Conventions

Verse has a set of naming conventions that make code readable and predictable. While the language doesn't enforce these conventions, following them ensures your code integrates well with the broader Verse ecosystem and is immediately familiar to other Verse developers.

Identifiers should be in PascalCase (CamelCase starting with uppercase):

```
# Variables and constants use PascalCase
PlayerHealth:int = 100
MaxInventorySize:int = 50
IsGameActive:logic = true

# Functions use PascalCase
CalculateDamage(Base:float, Multiplier:float):float =
    Base * Multiplier

GetPlayerName(Id:int)<decides>:string =
    PlayerDatabase[Id].Name

# Classes and structs use snake_case
player_character := class:
    Name:string
    Level:int

inventory_item := struct:
    ItemId:int
    Quantity:int

# Enums and their values use snake_case
game_state := enum:
    main_menu
    in_game
    paused
    game_over
```

Generic type parameters use single lowercase letters or short descriptive names:

```
# Single letter for simple generics
Find(Array:[]t, Target:t where t:type):?int = false

# Descriptive names for complex relationships
Transform(Input:in_t, Processor:type{_(:in_t):out_t} where in_t:type, out_t:type):?out_t = fals
```

Module names follow the snake_case pattern, while paths use a hierarchical structure with forward slashes and PascalCase for path segments:

```
# Module definition
inventory_system := module:
    # Module contents

# Path structure uses PascalCase for segments
using { /Fortnite.com/Characters/PlayerController }
using { /MyGame.com/Systems/CombatSystem }
using { /Verse.org/Random }
```

Class and struct fields use PascalCase, and methods follow the same PascalCase convention as functions:

```
player := class:
    Name:string          # PascalCase for fields
    var Health:float= 0.0

    # Methods use PascalCase like functions
    TakeDamage(Amount:float):void =
        set Health = Max(0.0, Health - Amount)

    IsAlive():logic =
        logic{Health > 0.0}
```

2.5 Code Formatting

Verse code follows consistent formatting patterns to emphasize readability.

Use four spaces to indent code blocks. The colon introduces a block, with subsequent lines indented:

```
if (Condition[]):
    DoSomething()
    DoSomethingElse()

for (Item : Inventory):
    ProcessItem(Item)
    UpdateDisplay()

class_definition := class:
    Field1:int
    Field2:string

```

```
    Method():void =
        ImplementationHere()
```

Complex expressions benefit from clear formatting that shows structure:

```
# Multi-line conditionals
Result := if (Player.Health > 50):
    "healthy"
else if (Player.Health > 20):
    "injured"
else:
    "critical"

# Chained operations with clear precedence
FinalDamage :=
    BaseDamage *
    LevelMultiplier *
    (1.0 + BonusPercentage / 100.0)

# Pattern matching with aligned cases
DamageMultiplier := case(Rarity):
    rarity_type.common => 1.0
    rarity_type.uncommon => 1.5
    rarity_type.rare => 2.0
    rarity_type.epic => 3.0
    rarity_type.legendary => 5.0
```

Functions follow a consistent pattern with effects and return types clearly specified:

```
# Simple pure function
Add(X:int, Y:int)<computes>:int = X + Y

# Function with effects
ProcessTransaction(Amount:int)<transacts><decides>:void =
    ValidateAmount[Amount]
    DeductBalance(Amount)
    RecordTransaction()

# Multi-line function with clear structure
CalculateReward(
    PlayerLevel:int,
    Difficulty:difficulty_level,
    CompletionTime:float
)<decides>:int =
```

```
    BaseReward := GetBaseReward[Difficulty]?
    LevelBonus := PlayerLevel * 10
    TimeBonus := CalculateTimeBonus(CompletionTime)
    BaseReward + LevelBonus + TimeBonus
```

2.6 Comments

Comments are ignored during execution but are valuable for understanding and maintaining code. Verse offers several styles of comments to suit different documentation needs. The simplest is the single-line comment, which begins with `#` and continues to the end of the line:

```
CalculateDamage := 100 * 1.5  # Apply critical hit multiplier
```

When you need to document something within a line of code without breaking it up, inline block comments provide the perfect solution. These are enclosed between ‘`and#>`‘:

```
Result := BaseValue  original amount #> * Multiplier  scaling factor #> + Bonus
```

The same can be used to write multi-line block comments, making them ideal for explaining complex algorithms or providing detailed context:

```
 This function implements the quadratic damage falloff formula
   used throughout the game. The falloff ensures that damage
   decreases smoothly with distance, creating strategic positioning
   choices for players. #>
CalculateFalloffDamage(Distance:float, MaxDamage:float):float =
    # Implementation here
```

Block comments nest, which allows you to temporarily disable code that already contains comments without having to remove or modify existing documentation:

```
 Temporarily disabled for testing
   OriginalFunction()   This had a bug #>
   NewFunction()       # Testing this approach
#>
```

Indented comments begin with a `>` on its own line; everything indented by four spaces on subsequent lines becomes part of the comment:

```
>
    This entire block is a comment because it's indented.
    It provides a clean way to write longer documentation
    without cluttering each line with comment markers.

DoSomething()  # Not part of the comment.
```

2.7 Syntactic Styles

Verse offers flexible syntax to accommodate different programming styles. The same logic can be expressed using braces, indentation, or inline forms, allowing you to choose the clearest representation for each context.

The braced style uses curly braces to delimit blocks, familiar from C-family languages:

```
Result := if (Score > 90) {
    "excellent"
} else if (Score > 70) {
    "good"
} else {
    "needs improvement"
}
```

The indented style uses colons and indentation to define structure, similar to Python:

```
Result := if (Score > 90):
    "excellent"
else if (Score > 70):
    "good"
else:
    "needs improvement"
```

For simple expressions, the inline style keeps everything on one line:

```
Result := if (Score > 90) then "excellent" else if (Score > 70) then "good" else "needs improve
```

The dotted style uses a period to introduce the expression:

```
Result := if (Score > 90). "excellent" else if (Score > 70). "good" else. "needs improvement"
```

You can even mix styles when it makes sense:

```
Result := if:
    ComplexCondition() and
    AnotherCheck() and
    YetAnotherValidation()
then { "condition met" } else { "condition not met" }
```

All these forms produce the same result. The choice between them is about readability and context. Use braces when working with existing brace-heavy code,

indentation for cleaner vertical layouts, and inline forms for simple expressions. This flexibility lets you write code that reads naturally.

Chapter 3

Chapter 2: Expressions

Everything is an expression. This design principle sets Verse apart from many other languages where statements and expressions are distinct concepts. Every piece of code you write produces a value, even constructs you might expect to be purely side-effecting. This creates a programming model where code can be composed and combined in ways that feel natural and predictable.

3.1 Primary Expressions

Everything starts with primary expressions—the atomic units from which more complex expressions are built. These include literals, identifiers, parenthesized expressions, and the tuple construct that provides lightweight data aggregation.

3.1.1 Basic Values

Literals are source code representations of constant values. Verse provides literals for all its primitive types: integers, floats, characters, strings, booleans, and functions. Each literal type has specific syntax rules that determine what values can be expressed and how they're interpreted.

```
Result := if (Condition?) then 42 else 3.14  # Integer and float literals
array{1, 2, 3}                               # Integer literals in array construction
Point{X:=0.0, Y:=1.0}                        # Float literals in object construction
```

Integer Literals

Integer literals represent whole numbers and can be written in two formats:

Decimal notation uses standard digits:

```
Count := 42
Negative := -17
Zero := 0
Large := 9223372036854775807                # Maximum 64-bit signed integer literals
```

Hexadecimal notation uses the `0x` prefix followed by hex digits (0-9, a-f, A-F):

```
Byte := 0xFF
Address := 0x1F4A
LowercaseHex := 0xabcdef
UppercaseHex := 0xABCDEF
```

Integer literals must fit within a 64-bit signed integer range (`-9223372036854775808` to `9223372036854775807`). At runtime, integer values are arbitrary precision and can grow past the values that can be written as literals. However, integers exceeding 64-bit have limited support (e.g., cannot be used in string interpolation or persisted).

Float Literals

Floating-point literals represent decimal numbers, they must include a decimal point and in some cases the `f64` suffix.

```
Pi := 3.14159
Half := 0.5
Explicit := 12.34f64    # Explicit bit-depth suffix
```

Scientific notation is used for very large or small numbers using exponents:

```
Large := 1.0e10         # 10,000,000,000 (sign optional)
Small := 1.0e-5         # 0.00001
WithSign := 2.5e+3      # 2,500 (explicit + sign)
Compact := 1.5e2        # 150 (no sign defaults to +)
```

Some rules:

- Must have decimal point: `1.0` is valid, `1` is an integer
- Final decimal point without digits is invalid: `1.` is a syntax error
- All floats are 64-bit (IEEE 754 double precision); the `f64` suffix is optional
- Unary operators work as with integers: `-1.0`, `+1.0`

Float literals must fit within IEEE 754 double-precision range or produce compile-time errors:

```
#TooBig := 1.7976931348623159e+308    # ERROR: Overflow
Maximum := 1.7976931348623158e+308    # OK: Maximum float
```

Character Literals

Character literals represent individual text units. Verse has two character types with different literal syntax:

`char` literals represent UTF-8 code units (single bytes, 0-255):

```
LetterA := 'a'          # Printable ASCII character
Space := ' '
Tab := '\t'             # Escape sequence
Hex := 0o61             # Hex notation: 0oXX (97 decimal = 'a')
```

`char32` literals represent Unicode code points:

```
Emoji := ' '           # Non-ASCII automatically char32
Accented := 'é'
ChineseChar := ' '
HexUnicode := 0u1f600   # Hex notation: 0uXXXXX ( )
```

Type inference from literals:

- ASCII characters (`U+0000` to `U+007F`): `'a'` has type `char`
- Non-ASCII characters: `' '` has type `char32`
- No implicit conversion between `char` and `char32`

Escape sequences work in both `char` and strings:

Escape	Meaning	Codepoint
`\t`	Tab	U+0009
`\n`	Newline	U+000A
`\r`	Carriage return	U+000D
`\"`	Double quote	U+0022
`\'`	Single quote	U+0027
`\\`	Backslash	U+005C
`\{`	Left brace (string interpolation)	U+007B
`\}`	Right brace (string interpolation)	U+007D
`\<`	Less than	U+003C
`\>`	Greater than	U+003E
`\&`	Ampersand	U+0026
`\#`	Hash	U+0023
`\~`	Tilde	U+007E

Hex notation work as follows:

- `0oXX` for `char` (two hex digits, `0o00` to `0off`)

- `0uXXXXX` for `char32` (up to six hex digits, `0u00000` to `0u10ffff`)

Character literals can not be empty or have multiple characters.

String Literals

String literals represent text sequences and support interpolation for embedding expressions. Basic strings use double quotes:

```
Greeting := "Hello, World!"
Empty := ""
WithEscapes := "Line 1\nLine 2\tTabbed"
```

String interpolation embeds expressions using curly braces:

```
Name := "Alice"
Age := 30

# Simple interpolation
Message := "Hello, {Name}!"                    # "Hello, Alice!"

# Expression interpolation
Info := "Age next year: {Age + 1}"             # "Age next year: 31"

# Function calls
Score := 100
Text := "Score: {ToString(Score)}"             # "Score: 100"

# Function calls with named arguments
Distance := 5.5
Formatted := "Distance: {Format(Distance, ?Decimals:=2)}"
```

Multi-line strings can span multiple lines using interpolation braces for continuation:

```
LongMessage := "This is a multi-line {
}string that continues across {
}multiple lines."
# Result: "This is a multi-line string that continues across multiple lines."

OtherMessage := "Another message{
}    with some empty{
}    spaces."
# Result := "Another message    with some empty    spaces."
```

Empty interpolants are ignored:

```
Text1 := "ab{}cd"        # Same as "abcd"
Text2 := "ab{
}cd"                     # Same as "abcd" (newline ignored)
```

Special rules:

- Curly braces must be escaped: `"\{ \}"` for literal braces
- `string` is an alias for `[]char` (array of UTF-8 code units)
- Strings are sequences of UTF-8 bytes, not Unicode characters
- `"José".Length = 5` (5 bytes, not 4 characters - é takes 2 bytes)

String-array equivalence:

```
Test1 := logic{"abc" = array{'a', 'b', 'c'}}    # True
Test2 := logic{"" = array{}}                    # True
```

Comments in strings are removed:

```
Text1 := "abccomment#>def"     # Same as "abcdef"
```

Boolean Literals

The `logic` type has two literal values:

```
IsReady := true
IsComplete := false
```

Boolean values are used with the query operator `?` or in comparisons:

```
if (IsReady?):
    StartGame()

if (IsComplete = true):
    ShowResults()
```

The `logic{}` expression creates boolean values from failable expressions (see Failure for details on failable expressions):

```
# Converts <decides> expression to logic value
Success := logic{Operation[]}       # true if succeeds, false if fails
HasValue := logic{Optional?}        # true if optional has value
IsEqual := logic{X = Y}             # true if equal, false otherwise
```

The `logic{}` expression requires at least a superficial possibility of failure. Pure expressions without `<decides>` effect cause errors:

```
# ERROR: logic{0} has no decides effect
# ERROR: logic{} is empty
Valid := logic{false?}               # OK: false? can fail
```

Multiple expressions inside `logic{}` can be separated by semicolons or commas (see Semicolons vs Commas for details):

```
Result1 := logic{true?; true?}       # Semicolon separator
Result2 := logic{true?, true?}       # Comma separator
```

Path Literals

Path literals identify modules and packages using a hierarchical naming scheme:

```
/Verse.org/Verse                     # Standard library path
/YourGame/Player/Inventory           # Custom module path
/user@example.com/MyModule           # Personal namespace
```

Path syntax follows specific rules:

- Starts with `/`
- Contains label (alphanumeric, ., -)
- Optional version after `@`
- Identifiers must start with letter or _

Path literals are covered in detail in the Modules chapter.

3.1.2 Identifiers and References

Identifiers serve as references to values, whether they're constants, variables, functions, or types. The language doesn't syntactically distinguish between these different kinds of identifiers:

```
int                # Reference to the int type
GetValue           # Reference to a function
Counter            # Reference to a variable
my_class           # Reference to a class
```

3.1.3 Parentheses and Grouping

Parentheses serve dual purposes: they group expressions to control evaluation order, and they create tuple expressions. A parenthesized expression simply evaluates to the value of its contents, allowing you to override the default operator precedence or improve readability:

```
(A + B) * C       # Group addition before multiplication
if (X > 0 and Y > 0) then Positive else Negative
```

3.1.4 Tuples

Tuples provide a way to group two or more values with little ceremony. The syntax distinguishes between parentheses used for grouping and those used for tuple construction through the presence of commas:

```
(X, Y)            # Two-element tuple
(1, "hello", true) # Mixed-type tuple
```

Tuples can be accessed using function-call syntax with a single integer argument:

```
point := (10, 20)
x := point(0)     # Access first element
y := point(1)     # Access second element
```

Tuple types are written:

```
tuple(int,int)
tuple(int,string,logic)
```

While the type of an unary element can be accepted by the compiler, `tuple(int)`, there is currently no syntax to write a tuple of one element.

3.2 Postfix Operations

Postfix operations are operations that follow their operand and can be chained together. This creates a left-to-right reading order that feels natural and allows for intuitive composition.

3.2.1 Member Access

The dot operator provides access to members of objects, modules, and other structured values. Member access expressions evaluate to the value of the specified member:

```
Player.Health           # Access field
Config.MaxPlayers       # Access nested value
math.Sqrt(16.0)         # Access module function
Point.X                 # Access struct field
```

Member access can be chained, creating paths through nested structures:

```
Game.Players[0].Inventory.Items[0].Name
```

3.2.2 Computed Access

Square brackets provide computed access to elements, whether for arrays, maps, or other indexable structures. The expression within brackets is evaluated to determine which element to access:

```
Array[0]                # Array indexing
Map["key"]              # Map lookup
Matrix[Row][Col]        # Nested indexing
Data[ComputeIndex()]    # Dynamic index computation
```

The function call syntax with square brackets, `Func[]` is equivalent to `Func()` for functions that may fail. Array indexing can fail, if the index is out of bounds, and thus uses `[]`.

3.2.3 Function Calls

Function calls use parentheses with comma-separated arguments. The language treats function calls as expressions that evaluate to the function's return value:

```
Sqrt(16)                        # Single argument
MaxOf(A, B)                     # Multiple arguments
Initialize()                    # No arguments
Process[GetData(), Transform()] # Nested calls, outer call may fail
```

3.3 Object Construction

Object construction uses a distinctive brace syntax to indicates the creation of a new instance. The syntax requires explicit field initialization using the `:=` operator:

```
point{X:=10, Y:=20}
player{Name:="Hero", Level:=1, Health:=100}
config{
    MaxPlayers := 16,
    EnablePvP := true,
    Difficulty := "normal"
}
```

The use of `:=` for field initialization reinforces that these are binding operations—you're binding values to fields at construction time. Object constructors can be nested, creating complex initialization expressions:

```
Game := game_state{
    Player := player{
        Position := point{X:=0, Y:=0},
        Inventory := inventory{Capacity:=20}
```

```
    },
    Settings := config{Difficulty:="hard"}
}
```

3.4 Control Flow as Expressions

One of Verse's distinctive features is that control flow constructs are expressions, not statements. This means that if-expressions, loops, and case expressions all produce values that can be used in larger expressions.

3.4.1 Conditional

The if-then-else construct is an expression that evaluates to one of two values based on a condition:

```
Result := if (X > 0) then "positive" else "negative"
Value := if (Condition=true) then ComputeA() else ComputeB()
```

The else clause can be omitted, though this affects the type of the expression. Verse supports multiple syntactic forms for if-expressions, including parenthesized conditions and indented bodies:

```
# Standard form
if (Condition?) then Value1 else Value2

# Indented form
if:
    Condition?
then:
    Value1
else:
    Value2
```

3.4.2 For

For expressions iterate over collections and produce values. The basic form iterates over elements:

```
for (Item : Collection) { Process(Item) }
```

An extended form provides access to both index and item–in the case of a `Map`, indices are not limited to integers:

```
for (Index -> Item : Collection) {
    Print("Item at {Index} is {Item}")
}
```

Since for expressions are themseleves expressions, they produce array values and compose with other expressions. The body of a for expression is evaluated for each successful iteration, and the expression as a whole has a value determined by these evaluations.

3.4.3 Loop

Loop expressions provide indefinite iteration, continuing until explicitly terminated through failure or other control flow:

```
loop {
    Value := GetNext()
    if (Done[Value]) then break
    Process(Value)
}
```

The loop construct can use indented syntax for clarity.

3.4.4 Case

Case expressions provide multi-way branching based on value matching:

```
Description := case(Color) {
    color.Red => "Danger",
    color.Yellow => "Warning",
    color.Green => "Safe",
    _ => "Unknown"
}
```

The _ pattern serves as a catch-all, ensuring the case expression is exhaustive. Case expressions evaluate to the value of the matched branch, making them useful for value computation as well as control flow.

3.5 Binary Operations

Binary expressions follow a carefully designed precedence hierarchy that balances mathematical conventions with programming practicality.

3.5.1 Assignment and Binding

At the lowest precedence level, assignment operators bind values to identifiers. The `:=` operator creates immutable bindings, while `set =` performs mutable assignment:

```
X := 42           # Immutable binding
Y := X * 2        # Binding to computed value
Z := W := 10      # Right-associative chaining
```

Assignment operators are right-associative, meaning that `a := b := c` groups as `a := (b := c)`. This allows for natural chaining of assignments while maintaining clarity about evaluation order.

Compound assignments provide shorthand for common update patterns:

```
set Counter += 1      # Equivalent to: set Counter = Counter + 1
set Total *= Factor   # Equivalent to: set Total = Total * Factor
```

3.5.2 Range Expressions

The range operator (`..`) creates integer ranges for iteration in `for` loops. Ranges are **inclusive on both ends** and can only appear directly in for loop iteration clauses:

```
1..10             # Range from 1 to 10 (inclusive)
Start..End        # Variable-defined range
for (I := 0..Count):  # Must use := syntax, not :
    Process(I)
```

Ranges are not first-class values. They cannot be stored in variables or used outside of `for` loop iteration clauses. See the Range Operator Restrictions section for details.

3.5.3 Logical Operations

Logical operators combine boolean values with short-circuit evaluation. Their result is either success or failure. Verse uses keyword operators (`and`, `or`, `not`) rather than symbols, improving readability:

```
if (X > 0 and Y > 0) then ProcessQuadrant()
Result := logic{Validated? or UseDefault[]}
if (not IsReady[]) then Wait()
```

The precedence ensures that `and` binds tighter than `or`, matching mathematical logic conventions, the `logic{}` expression turns succes or failure into a value:

```
# Evaluates as: (ExpA and ExpB) or (ExpC and ExpD)
Condition := logic{ExpA and ExpB or ExpC and ExpD}
```

3.5.4 Comparison Operations

Comparison operators also either succeed or fail and can be chained for range checking:

```
if (0 <= Value <= 100) then InRange()
IsValid := logic{X > Minimum and X < Maximum}
```

```
Same := logic{A = B}
Different := logic{A <> B}
```

All comparison operators have the same precedence and are evaluated left-to-right, allowing natural mathematical notation for range checks.

3.5.5 Arithmetic Operations

Arithmetic operations follow standard mathematical precedence, with multiplication and division binding tighter than addition and subtraction:

```
Result := A + B * C      # Multiplication first
Average := (A + B) / 2   # Parentheses override precedence
```

Unary operators have the highest precedence among arithmetic operations:

```
Negative := -Value
Inverted := logic{not Flag=true}
Result := -X * Y    # Unary minus applies to x only
```

3.6 Set Expressions

While Verse emphasizes immutability, practical programming sometimes requires mutation. Set expressions provide mutation of variables and fields:

```
set X = 10                    # Variable assignment
set Obj.Field = Value         # Field assignment
set Arr[Index] = Element      # Array element assignment
set Map[Key] = MappedValue    # Map entry assignment
```

Set expressions are themselves expressions, though they're typically used for their side effects rather than their value. The left-hand side must be a valid LValue—something that can be assigned to.

Complex LValues are supported, allowing updates deep within data structures:

```
set Game.Players[CurrentPlayer].Inventory.Items[Slot] = NewItem
```

3.7 Semicolons vs Commas

Verse uses semicolons and commas as separators in various contexts, but they have fundamentally different semantics in most situations. Understanding when each is appropriate is essential for writing correct Verse code.

Semicolons create *sequences* - they evaluate expressions in order and return the value of the last expression:

```
Result := (1; 2; 3)     # Evaluates 1, then 2, then 3; returns 3
# Result = 3 (type: int)
```

Commas create *tuples* - they group multiple values into a single composite value:

```
Result := (1, 2, 3)     # Creates a tuple of three elements
# Result = (1, 2, 3) (type: tuple(int, int, int))
```

3.7.1 Context-Specific Behavior

The semicolon-versus-comma distinction is most visible in parenthesized expressions:

```
# Semicolon: sequence (returns last value)
X := (0; 1)              # X = 1, type is int

# Comma: tuple (groups values)
Y := (0, 1)              # Y = (0, 1), type is tuple(int, int)
```

This applies to function return values as well:

```
GetInt():int = (1.0; 2)                    # Returns 2 (int)
GetTuple():tuple(float, int) = (1.0, 2)    # Returns (1.0, 2)
```

Semicolons in argument position create a *sequence that executes before the call*, with only the last value passed as the argument:

```
# Semicolon executes side effects, then passes last value
Process(LogEvent("called"); 42)   # Logs "called", then calls Process(42)

# Equivalent to:
LogEvent("called")
Process(42)
```

This pattern enables side effects in argument position:

```
Result := MultiplyByTen(2; 3)     # Evaluates 2 (discards it), calls Multiply(3)
Result = 30
```

Commas separate distinct arguments in the standard way:

```
Sum := Add(10, 20)               # Two separate arguments
# Sum = 30
```

Semicolons are *not allowed* in parameter lists - you must use commas:

```
# VALID: Comma-separated parameters
ValidFunc(A:int, B:int):void = {}

```

```
# INVALID: Semicolon in parameters
# InvalidFunc(A:int; B:int):void = {}
```

3.7.2 In Specific Scopes

At the top level of a module, semicolons and commas are *interchangeable* - both simply separate definitions:

```
# Both valid and equivalent
X:int = 0; Y:int = 0
X:int = 0, Y:int = 0
```

In `logic{}` constructor - both semicolons and commas work, but with different semantics based on the construct's behavior:

```
# Both evaluate all expressions and return logic value
Result1 := logic{true?; true?}    # Sequence of queries
Result2 := logic{true?, true?}    # Also valid
```

In `option{}` constructor - follows the standard sequence vs tuple rule:

```
# Semicolon: sequence, wraps last value
Option1 := option{1; 2}?          # 2

# Comma: tuple, wraps the tuple
Option2 := option{1, 2}?          # (1, 2)
```

In `for` expressions - semicolon typically separates the iteration clause from filter conditions, while commas separate multiple conditions:

```
# Semicolon separates iteration from filter
for (X := 1..3; X <> 2) { X }

# Comma separates multiple filter conditions
for (X := 1..3, X <> 2) { X }      # Same meaning in this context
```

In `array{}` constructor - use commas to separate elements:

```
Numbers := array{1, 2, 3}          # Array of three elements
```

3.7.3 Newlines as Separators

In addition to semicolons and commas, **newlines** can serve as separators in compound expressions and blocks. Newlines behave like semicolons - they create sequences:

```
# These are equivalent:
Result1 := (1; 2; 3)
```

```
3
4  Result2 := (
5      1
6      2
7      3
8  )
9  # Both return 3
```

3.8 Compound and Block Expressions

Compound expressions, delimited by braces, group multiple expressions into a single expression. The value of a compound expression is the value of its last sub-expression:

```
1  Result := {
2      Temp := ComputeIntermediate()
3      Adjustment := CalculateAdjustment(Temp)
4      Temp + Adjustment
5  }
```

Compound expressions create new scopes for variables, allowing local bindings that don't affect the enclosing scope:

```
1  block:
2      X := 10    # Local to this block
3      Y := 20
4      X + Y
5                 # X and Y no longer accessible
```

Expressions within a compound can be separated by semicolons, commas, or newlines. Semicolons and newlines create sequences (returning the last value), while commas create tuples. See Semicolons vs Commas for the complete rules:

```
1  { A; B; C }           # Semicolon separation (returns C)
2  { A, B, C }           # Comma separation (returns tuple (A, B, C))
3  {                     # Newline separation (returns C)
4      A
5      B
6      C
7  }
```

3.9 Array Expressions

Array expressions create array values using the `array` keyword followed by elements in braces:

```
Numbers := array{1, 2, 3, 4, 5}
Empty := array{}
Mixed := array{1, "two", 3.0}  # Mixed types if allowed
```

Arrays can also be constructed using indented syntax for clarity with longer lists:

```
Colors := array:
    "red"
    "green"
    "blue"
    "yellow"
```

Chapter 4

Chapter 3: Primitive Types

Verse provides a rich set of primitive types that cover fundamental programming needs. The numeric types `int`, `float`, and `rational` handle mathematical operations, counters, and measurements. The `logic` type represents boolean values for conditions and flags. Text is handled through `char`, `char32`, and `string` types for character data, player names, and messages. Two special types, `any` and `void`, serve unique roles in the type hierarchy as the supertype of all types and the empty type respectively.

Let's explore each primitive type in detail, starting with the numeric types that form the backbone of game logic.

4.1 Intrinsics

intrinsic functions are built-in operations provided directly by the runtime that cannot be implemented in pure Verse code. These functions receive special compiler treatment and form the foundation for many language features. Intrinsic functions are special because they:

- **Implemented by the runtime**: Written in C++ or other native code, not Verse
- **Cannot be replicated in Verse**: Require access to runtime internals or low-level operations
- **Receive compiler recognition**: The compiler knows about them and may optimize their use

Examples include mathematical operations like `Abs()`, collection methods like `Find()`, and type conversions like `ToString()`.

Most intrinsic functions *cannot be referenced as first-class values*. This means you can call them directly, but you cannot store them in variables or pass them as function arguments:

```
Result := Abs(-42)  # Returns 42

# Invalid: Cannot reference without calling
# F := Abs  # ERROR
# Invalid: Cannot pass as parameter
# ApplyFunction(Abs, -42)  # ERROR
```

This restriction exists because intrinsics often require special calling conventions or optimizations that don't fit the standard function model. If you need to pass intrinsic functionality around, wrap it in a regular function or nested function.

4.2 Integers

The `int` type represents integer, non-fractional values. An `int` can contain a positive number, a negative number, or zero. At runtime, integers are arbitrary precision and can grow beyond any fixed size. However, integer *literals* must fit within a 64-bit signed range (`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`), and integers exceeding 64-bit have limited support (e.g., cannot be used in string interpolation or persisted).

You can include `int` values within your code as literals.

```
A :int= -42                     # civilian size
#B := 424242424242424242424242424242424242424242424242424242 # scary numbers...
                                # ...can be computed but not written as literals

AnswerToTheQuestion :int= 42   # A variable that never changes
CoinsPerQuiver :int= 100       # A quiver costs this many coins
ArrowsPerQuiver :int= 15       # A quiver contains this many arrows

# Mutable variables (see Mutability chapter for details on var and set)
var Coins :int= 225           # The player currently has 225 coins
var Arrows :int= 3            # The player currently has 3 arrows
var TotalPurchases :int= 0    # Track total purchases
```

You can use the four basic math operations with integers: `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division.

```
var C :int= (-MyInt + MyHugeInt - 2) * 3   # arithmetic
set C += 1                                 # like saying, set C = C + 1
set C *= 2                                 # like saying, set C = C * 2
```

For integers, the operator `/` is failable, and the result is a `rational` type if it succeeds.

4.3 Rationals

The `rational` type represents exact fractions as ratios of integers. Unlike `int` or `float`, you cannot write a `rational` literal directly—rationals are created through integer division using the `/` operator.

```
X := 7 / 3    # X has type rational, representing exactly 7÷3
```

Rationals provide *exact arithmetic* without the precision loss of floating-point numbers, making them ideal for game logic requiring precise fractional calculations (resource distribution, turn-based systems, probability calculations).

Integer division with `/` produces a rational value. Division by zero fails:

```
Half := 5 / 2           # rational: exactly 5/2
Third := 10 / 3         # rational: exactly 10/3
Quarter := 1 / 4        # rational: exactly 1/4

if (not (1 / 0)):
    # Division by zero fails
```

Rationals are automatically reduced to lowest terms for equality comparisons:

```
# All these are equal - reduced to 5/2
(5 / 2) = (10 / 4)      # true
(5 / 2) = (15 / 6)      # true
(10 / 4) = (15 / 6)     # true
```

This normalization ensures that mathematically equivalent rationals compare as equal regardless of how they were constructed.

Negative signs are normalized to the numerator:

```
(1 / -3) = (-1 / 3)     # true: negative moves to numerator
(-1 / -3) = (1 / 3)     # true: double negative becomes positive
```

This canonical form simplifies equality checking and ensures consistent behavior.

An important property: *`int` is a subtype of `rational`*. This means any integer can be used where a rational is expected:

```
ProcessRational(X:rational):rational = X

# Can pass integers directly
```

```
ProcessRational(5) = 5/1     # 5 is implicitly 5/1 (rational)
ProcessRational(0) = 0/1     # 0 is implicitly 0/1 (rational)
```

However, you *cannot* return a rational where an int is expected—that would be a narrowing conversion:

```
BadFunction(X:rational):int = X  # Error
```

Whole number rationals equal their integer equivalents:

```
(2 / 1) = 2             # true
2 = (2 / 1)             # true
(4 / 2) = 2             # true: 4/2 reduces to 2/1, equals 2
(9 / 3) = 3             # true: 9/3 reduces to 3/1, equals 3
```

This enables seamless mixing of integer and rational values in calculations.

Two functions convert rationals to integers:

- `Floor` — rounds toward negative infinity (down on number line)
- `Ceil` — rounds toward positive infinity (up on number line)

```
# Positive rationals
Floor(5 / 2)= 2         # 2.5 → 2 (down)
Ceil(5 / 2) = 3         # 2.5 → 3 (up)

# Negative rationals - note direction!
Floor((-5) / 2) = -3    # -2.5 → -3 (toward negative infinity)
Ceil((-5) / 2) = -2     # -2.5 → -2 (toward positive infinity)

# With negative denominator
Floor(5 / -2) = -3      # Same as (-5)/2
Ceil(5 / -2) = -2       # Same as (-5)/2

# Both negative
Floor((-5) / -2) = 2    # 2.5 → 2
Ceil((-5) / -2) = 3     # 2.5 → 3
```

`Floor` rounds toward negative infinity, *not* toward zero. This matches mathematical convention but differs from truncation. When the argument is a rational, `Floor` does not fail, but if passed a `float` it is a `decides` function.

Rationals can be used as parameter and return types:

```
# Function returning rational
Half(X:int)<computes><decides>:rational = X / 2

```

```
# Use the result
if (Result := Half[7]):
    Floor(Result) = 3   # 7/2 = 3.5, Floor gives 3
    Ceil(Result) = 4    # 7/2 = 3.5, Ceil gives 4
```

Because `int` is a subtype of `rational`, you *cannot* overload based solely on these types:

```
ProcessValue(X:int):void = {}
ProcessValue(X:rational):void = {}  # Error!
```

The compiler sees `int` as more specific than `rational`, so the signatures would be ambiguous.

Rationals excel at resource distribution and fairness calculations:

```
# Fair resource distribution
DistributeResources(TotalGold:int, NumPlayers:int)<decides>:int =
    GoldPerPlayer := TotalGold / NumPlayers
    Floor(GoldPerPlayer)  # Each player gets whole gold pieces or we fail

# Item affordability calculation
Coins:int = 225
CoinsPerQuiver:int = 100
ArrowsPerQuiver:int = 15

if (NumberOfQuivers := Floor(Coins / CoinsPerQuiver)):
    TotalArrows:int = NumberOfQuivers * ArrowsPerQuiver
    # Player can afford 2 quivers = 30 arrows
```

4.4 Floats

The `float` type represents all non-integer numerical values. It can hold large values and precise fractions, such as `1.0`, `-50.5`, and `3.14159`. A float is an IEEE 64-bit float, which means it can contain a positive or negative number that has a decimal point in the range `[-2^1024 + 1, … , 0, … , 2^1024 - 1]`, or has the value `NaN` (Not a Number). The implementation differs from the IEEE standard in the following ways:

- There is only one `NaN` value.
- `NaN` is equal to itself.
- Every number is equal to itself.
- `0` cannot be negative.

You can include float values within your code as literals:

```
A:float = 1.0
B := 2.14
MaxHealth : float = 100.0

var C:float = A + B
C = 3.14              # succeeds
set C -= 3.14
C = 0.0               # succeeds
# C = 0              # compile error; 0 is not a `float` literal
```

You can use the four basic math operations with floats: `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division. There are also combined operators for doing the basic math operations (addition, subtraction, multiplication, and division), and updating the value of a variable:

```
var CurrentHealth : float = 100.0
set CurrentHealth /= 2.0    # Halves the value of CurrentHealth
set CurrentHealth += 10.0   # Adds 10 to CurrentHealth
set CurrentHealth *= 1.5    # Multiplies CurrentHealth by 1.5
```

To convert an `int` to a `float`, multiply it by `1.0`: `MyFloat:=MyInt*1.0`.

4.5 Mathematical Functions

Verse provides intrinsic mathematical functions for common numerical operations. These functions are optimized by the runtime and work with both `int` and `float` types.

The `Abs()` function returns the absolute value of a number—its distance from zero without regard to sign:

```
# Signatures
Abs(X:int):int
Abs(X:float):float
```

```
Abs(5)    # Returns 5
Abs(-5)   # Returns 5
Abs(0)    # Returns 0
Abs(3.14) # Returns 3.14
```

The `Min()` and `Max()` functions return the minimum or maximum of two values:

```
# Signatures
Min(A:int, B:int):int
Min(A:float, B:float):float
```

```
Max(A:int, B:int):int
Max(A:float, B:float):float
```

```
# NaN propagates through comparison
Max(NaN, 5.0)   # Returns NaN
Min(NaN, 5.0)   # Returns NaN
Max(NaN, NaN)   # Returns NaN

# Infinity handling
Max(Inf, 100.0)    # Returns Inf
Min(-Inf, 100.0)   # Returns -Inf
Max(-Inf, -Inf)    # Returns -Inf
Min(Inf, Inf)      # Returns Inf
```

Verse provides multiple rounding functions that convert floats to integers with different rounding strategies:

```
# Signatures
Floor(X:float)<reads><decides>:int   # Round down
Ceil(X:float)<reads><decides>:int    # Round up
Round(X:float)<reads><decides>:int   # Round to nearest even (IEEE-754)
Int(X:float)<reads><decides>:int     # Truncate toward zero
```

Round to nearest even (ties go to even):

```
Round[1.5]    # Returns 2 (tie: 1.5 rounds to even 2)
Round[0.5]    # Returns 0 (tie: 0.5 rounds to even 0)
Round[2.5]    # Returns 2 (tie: 2.5 rounds to even 2)
Round[-1.5]   # Returns -2 (tie: -1.5 rounds to even -2)
Round[-0.5]   # Returns 0 (tie: -0.5 rounds to even 0)

Round[1.4]    # Returns 1 (no tie, rounds down)
Round[1.6]    # Returns 2 (no tie, rounds up)
```

The “round to nearest even” strategy (also called banker’s rounding) avoids bias when rounding many tie values.

Some additional mathematical functions:

```
# Signature
# Sqrt(X:float):float

# Negative inputs return NaN
Sqrt(-1.0)    # Returns NaN

# Special values
```

```
Sqrt(Inf)     # Returns Inf
Sqrt(NaN)     # Returns NaN

# Signature
# Pow(Base:float, Exponent:float):float

Pow(2.0, 3.0)     # Returns 8.0 (2³)
Pow(10.0, 2.0)    # Returns 100.0
Pow(4.0, 0.5)     # Returns 2.0 (square root)
Pow(2.0, -1.0)    # Returns 0.5 (reciprocal)

# Special cases
Pow(0.0, 0.0)     # Returns 1.0 (by convention)
Pow(NaN, 0.0)     # Returns 1.0 (0 exponent always 1)
Pow(1.0, NaN)     # Returns 1.0 (1 to any power is 1)

# Exp(X:float):float

Exp(0.0)      # Returns 1.0
Exp(1.0)      # Returns 2.718... (e)
Exp(-1.0)     # Returns 0.368... (1/e)

# Special values
Exp(-Inf)     # Returns 0.0
Exp(Inf)      # Returns Inf
Exp(NaN)      # Returns NaN

# Signature
# Ln(X:float):float

Ln(1.0)       # Returns 0.0
# Ln(2.718...)     # Returns 1.0 (ln(e) = 1)
Ln(10.0)      # Returns 2.302...

# Invalid inputs
Ln(-1.0)      # Returns NaN (negative)
Ln(0.0)       # Returns -Inf (log of zero)

# Special values
Ln(Inf)       # Returns Inf
Ln(NaN)       # Returns NaN

```

```
# Signature
# Log(Base:float, Value:float):float

Log(10.0, 100.0)   # Returns 2.0 (log (100) = 2)
Log(2.0, 8.0)      # Returns 3.0 (log (8) = 3)
Log(2.0, 2.0)      # Returns 1.0 (log (n) = 1)
```

Verse provides standard trigonometric functions operating on radians:

```
# Signatures
# Sin(Angle:float):float
# Cos(Angle:float):float
# Tan(Angle:float):float

# Common angles (using PiFloat constant)
Sin(0.0)              # Returns 0.0
Sin(PiFloat / 2.0)    # Returns 1.0
Sin(PiFloat)          # Returns 0.0
Sin(-PiFloat / 2.0)   # Returns -1.0

Cos(0.0)              # Returns 1.0
Cos(PiFloat / 2.0)    # Returns 0.0
Cos(PiFloat)          # Returns -1.0

Tan(0.0)              # Returns 0.0
Tan(PiFloat / 4.0)    # Returns 1.0
Tan(-PiFloat / 4.0)   # Returns -1.0

# Special values
Sin(NaN)              # Returns NaN
Sin(Inf)              # Returns NaN

# Signatures
# ArcSin(X:float):float   # Returns angle in [- /2,  /2]
# ArcCos(X:float):float   # Returns angle in [0,  ]
# ArcTan(X:float):float   # Returns angle in [- /2,  /2]
# ArcTan(Y:float, X:float):float  # Two-argument arctangent

# Inverse relationships
ArcSin(0.0)    # Returns 0.0
ArcSin(1.0)    # Returns  /2
ArcSin(-1.0)   # Returns - /2

```

```
ArcCos(1.0)    # Returns 0.0
ArcCos(0.0)    # Returns  /2
ArcCos(-1.0)   # Returns

ArcTan(0.0)    # Returns 0.0
ArcTan(1.0)    # Returns  /4
ArcTan(-1.0)   # Returns - /4

# Verify inverse relationship
Angle := PiFloat / 6.0  # 30 degrees
Sin(ArcSin(Sin(Angle))) = Sin(Angle)  # True

# ArcTan(Y, X) returns angle of point (X, Y) from origin
ArcTan(1.0, 1.0)     # Returns  /4 (45 degrees)
ArcTan(1.0, 0.0)     # Returns  /2 (90 degrees)
ArcTan(0.0, 1.0)     # Returns 0.0 (0 degrees)
ArcTan(1.0, -1.0)    # Returns 3 /4 (135 degrees)
ArcTan(-1.0, -1.0)   # Returns -3 /4 (-135 degrees)
```

Hyperbolic functions are analogs of trigonometric functions for hyperbolas. They are useful in physics simulations, catenary curves, and certain mathematical models.

```
# Signatures
# Sinh(X:float):float    # Hyperbolic sine
# Cosh(X:float):float    # Hyperbolic cosine
# Tanh(X:float):float    # Hyperbolic tangent
# ArSinh(X:float):float  # Inverse hyperbolic sine
# ArCosh(X:float):float  # Inverse hyperbolic cosine
# ArTanh(X:float):float  # Inverse hyperbolic tangent

Sinh(0.0)     # Returns 0.0
Sinh(1.0)     # Returns 1.175...
Cosh(0.0)     # Returns 1.0
Cosh(1.0)     # Returns 1.543...
Tanh(0.0)     # Returns 0.0
Tanh(1.0)     # Returns 0.761...

# Special values
Sinh(-Inf)    # Returns -Inf
Sinh(Inf)     # Returns Inf
Cosh(-Inf)    # Returns Inf
Cosh(Inf)     # Returns Inf
```

```
Tanh(-Inf)    # Returns -1.0
Tanh(Inf)     # Returns 1.0

ArSinh(0.0)   # Returns 0.0
ArCosh(1.0)   # Returns 0.0
ArTanh(0.0)   # Returns 0.0

# Special values
ArSinh(-Inf)  # Returns -Inf
ArSinh(Inf)   # Returns Inf
ArCosh(Inf)   # Returns Inf
ArCosh(-1.0)  # Returns NaN (domain error)
```

For integer division with remainder, Verse provides `Mod` and `Quotient`. Both functions are failable—they fail when the divisor is zero.

```
# Signatures
# Mod(Dividend:int, Divisor:int)<decides>:int
# Quotient(Dividend:int, Divisor:int)<decides>:int

# Positive operands
Mod[15, 4]      # Returns 3
Quotient[15, 4] # Returns 3
# Relationship: 15 = 3*4 + 3

# Negative dividend
Mod[-15, 4]      # Returns 1
Quotient[-15, 4] # Returns -4
# Relationship: -15 = -4*4 + 1

# Negative divisor
Mod[-1, -2]      # Returns 1
Quotient[-1, -2] # Returns 1

# Division by zero fails
if (not Mod[10, 0]):
    Print("Cannot mod by zero")
if (not Quotient[10, 0]):
    Print("Cannot divide by zero")
```

The modulo result always satisfies:

```
Dividend = Quotient[Dividend, Divisor] * Divisor + Mod[Dividend, Divisor]
```

The sign of the result follows specific rules:

- `Mod` result has the same sign as the divisor (Euclidean division)
- `Quotient` adjusts accordingly to maintain the identity

There are also some utility functions:

```
# Signatures
# Sgn(X:int):int
# Sgn(X:float):float

Sgn(10)       # Returns 1
Sgn(0)        # Returns 0
Sgn(-5)       # Returns -1

Sgn(3.14)     # Returns 1.0
Sgn(0.0)      # Returns 0.0
Sgn(-2.71)    # Returns -1.0

# Special float values
Sgn(Inf)      # Returns 1.0
Sgn(-Inf)     # Returns -1.0
Sgn(NaN)      # Returns NaN
```

Lerp interpolates between two values:

```
# Signature
# Lerp(From:float, To:float, Parameter:float):float

Lerp(0.0, 10.0, 0.0)    # Returns 0.0 (0% = From)
Lerp(0.0, 10.0, 0.5)    # Returns 5.0 (50%)
Lerp(0.0, 10.0, 1.0)    # Returns 10.0 (100% = To)
Lerp(0.0, 10.0, 2.0)    # Returns 20.0 (extrapolation)
Lerp(10.0, 20.0, 0.3)   # Returns 13.0

# Works with negative ranges
Lerp(-10.0, 10.0, 0.5)  # Returns 0.0
```

The formula is: `From + Parameter * (To - From)`

`IsFinite` checks if a float is finite and suceeds if the value is not NaN, Inf, or -Inf. And fails otherwise:

```
# Method on float values
# X.IsFinite()<computes><decides>:float

(5.0).IsFinite[]      # succeeds
(0.0).IsFinite[]      # succeeds
```

```
(-100.0).IsFinite[]   # succeeds

(Inf).IsFinite[]  # fails
(-Inf).IsFinite[] # fails
(NaN).IsFinite[]  # fails

# Returns the same number if succeeds
(15.16).IsFinite[] = 15.16 # succeeds, both are equal

# Useful for validation
# SafeCalculation(X:float, Y:float)<decides>:float =
#     X.IsFinite[] and Y.IsFinite[]
#     Result := X / Y
#     Result.IsFinite[]
#     Result
```

Verse provides constants for common mathematical values:

```
PiFloat # 3.14159265358979323846...
Inf     # Positive infinity
-Inf    # Negative infinity (negation of Inf)
NaN     # Not a Number
```

4.6 Booleans

The `logic` type represents the Boolean values `true` and `false`.

```
A:logic = true
B := false

# A = B          # fails
A?                # succeeds
# B?             # fails

true?             # succeeds
# false?         # fails
```

The `logic` type only supports query operations and comparison operations. Query expressions use the query operator `?` to check if a logic value is true and fail if the logic value is `false`. For comparison operations, use the failable operator `=` to test if two logic values are the same, and `<>` to test for inequality.

Many programming languages find it idiomatic to use a type like `logic` to signal the success or failure of an operation. In Verse, we use success and failure instead for

that purpose, whenever possible. The conditional only executes the `then` branch if the guard succeeds:

```
if (TargetLocked?):
    ShowTargetLockedIcon()
```

To convert an expression that has the `<decides>` effect to `true` on success or `false` on failure, use `logic{ exp }`:

```
GotIt := logic{GetRandomInt(0, Frequency) <> 0}   # if success
GotIt?                                            # then this succeeds
GotIt = false                                     # and this fails
not GotIt?                                        # and this fails too
```

4.7 Characters and Strings

Text is represented in terms of characters and strings. A `char` is a single **UTF-8 code unit** (not a full Unicode code point). A string is therefore an array of characters, written as `[]char`. For convenience, the type alias `string` is provided for `[]char`:

```
MyName :string = "Joseph"
MyAlterEgo := "José"
```

UTF-8 is used as the character encoding scheme. Each UTF-8 code unit is one byte. A Unicode code point may require between one and four code units. Code points with lower values use fewer bytes, while higher values require more.

For example:

- `"a"` requires one byte (`{0o61}`),
- `"á"` requires two bytes (`{0oC3}{0oA1}`),
- `" "` (cat emoji) requires four bytes (`{0u1f408}`).

Thus, strings are sequences of code units, not necessarily sequences of Unicode characters in the abstract sense.

Because strings are arrays of `char`, you can index into them with `[]`. Indexing has the `<decides>` effect: it succeeds when the index is valid and fails otherwise.

```
TheLetterJ := MyName[0]     # succeeds
TheLetterJ = 'J'            # succeeds
# MyName[100]               # fails
```

The length of a string is the number of UTF-8 code units it contains, accessed via `.Length`. Note that this is *not the same as the number of Unicode characters*:

```
"José".Length = 5           # succeeds; 5 UTF-8 code units
"Jose".Length = 4           # succeeds; 4 UTF-8 code units
```

Because `string` is just `[]char`, strings declared as `var` can be mutated:

```
var OuterSpaceFriend :string = "Glorblex"
set OuterSpaceFriend[0] = 'F'
```

Strings can be concatenated using the `+` operator:

```
MyAttemptAtFormatting := "My name is " + MyName + " but my alter ego is " + MyAlterEgo + "."
```

Verse also supports string interpolation for more readable formatting:

```
Formatting := "My name is {MyName} but my alter ego is {MyAlterEgo}."
```

Interpolation works for any value that has a `ToString()` function in scope.

Literal characters are written with single quotes. The type depends on whether the character falls within the ASCII range (`U+0000`–`U+007F`) or not:

- `'e'` has type `char`,
- `'é'` has type `char32`.

```
A :char = 'e'                   # ok
B :char32 = 'é'                 # ok
# C :char = 'é'                 # error: type of 'é' is char32
# D :char32 = 'e'               # error: type of 'e' is char
```

Character literals can also be written using numeric escape sequences:

```
E :char = 0o65                  # ok; same as 'e'
F :char32 = 0u00E9              # ok; same as 'é'
```

- `char` represents a single UTF-8 code unit (one byte, `0oXX`).
- `char32` represents a full Unicode code point (`0uXXXXX`).

Hex notation:

- `0oXX` for `char`: two hex digits (0o00 to 0off)
- `0uXXXXX` for `char32`: up to six hex digits (0u00000 to 0u10ffff)

Unlike some languages, Verse does not allow implicit conversion between characters and integers.

Character escape sequences work in both character and string literals:

Escape	Meaning	Codepoint
`\t`	Tab	U+0009
`\n`	Newline	U+000A
`\r`	Carriage return	U+000D
`\"`	Double quote	U+0022
`\'`	Single quote	U+0027
`\\`	Backslash	U+005C
`\{`	Left brace	U+007B
`\}`	Right brace	U+007D
`\<`	Less than	U+003C
`\>`	Greater than	U+003E
`\&`	Ampersand	U+0026
`\#`	Hash/pound	U+0023
`\~`	Tilde	U+007E

Examples:

```
Tab := '\t'
Newline := '\n'
Quote := '\"'
Brace := '\{'
```

Strings can be compared using the failable operators `=` (equality) and `<>` (inequality). Comparison is done by code point, and is case sensitive. Equality depends on exact code unit sequences, not visual appearance. Unicode allows multiple encodings for the same abstract character. For example, `"é"` may appear as the single code point `{0u00E9}`, or as the two-code-point sequence `"e"` (`{0u0065}`) plus a combining accent (`{0u0301}`). These two strings look the same, but they are not equal in Verse.

Checking whether a player has selected the correct item:

```
ExpectedItemInternalName :string = "RedPotion"
SelectedItemInternalName :string = "BluePotion"

if (SelectedItemInternalName = ExpectedItemInternalName):
    true
else:
    false
```

Padding a timer with leading zeros:

```
SecondsLeft :int = 30
SecondsString :string = ToString(SecondsLeft)    # convert int to string

```

```
var Combined :string = "Time Remaining: "
if (SecondsString.Length > 2):
    set Combined += "99"               # clamp to maximum
else if (SecondsString.Length < 2):
    set Combined += "0{SecondsString}" # pad with zero
else:
    set Combined += SecondsString
```

String interpolation supports complex expressions, not just simple variables:

```
# Expression interpolation
Age := 30
Message := "Next year: {Age + 1}"

# Function calls with named arguments
Distance := 5.5
Formatted := "Distance: {Format(Distance, ?Decimals:=2)}"
```

Strings can span multiple lines using interpolation braces for continuation:

```
LongMessage := "This is a multi-line {
}string that continues across {
}multiple lines."

# Attention to whitespace:
AnotherMessage := "This is another {
}  multi-line message with     {
    # This comment is ignored
}    many spaces."
```

Empty interpolants `{}` are ignored, which is useful for line continuation without adding content.

Since `string` is `[]char`, strings and character arrays can be compared:

```
"abc" = array{'a', 'b', 'c'}    # Succeeds
"" = array{}                     # Succeeds - empty string equals empty array
```

Block comments within strings are removed during parsing:

```
Text := "abcthis comment is removed#>def"    # Same as "abcdef"
```

4.7.1 ToString()

The `ToString()` function converts values to their string representations. It's polymorphic—multiple overloads exist for different types:

```
# Signatures
ToString(X:int):string
ToString(X:float):string
ToString(X:char):string
ToString(X:string):string  # Identity function
```

String interpolation implicitly calls `ToString()` on embedded values:

```
Age := 25
Score := 98.5

# These are equivalent:
Message1 := "Age: " + ToString(Age) + ", Score: " + ToString(Score)
Message2 := "Age: {Age}, Score: {Score}"
# Both produce: "Age: 25, Score: 98.5"
```

This makes `ToString()` essential for formatting output, even when you don't call it directly.

`ToString()` only works on primitive types. User-defined classes and structs don't have automatic string conversion.

4.7.2 ToDiagnostic()

The `ToDiagnostic()` function converts values to diagnostic string representations, useful for debugging and logging. While similar to `ToString()`, it may provide more detailed or implementation-specific information:

```
# Usage (exact signature depends on type)
DiagnosticText := ToDiagnostic(SomeValue)
```

`ToDiagnostic()` is primarily used for debugging output rather than user-facing strings. The exact format it produces may vary between VM implementations and is not guaranteed to be stable across versions.

4.8 Type type

The `type` type is a *metatype* - a type whose values are themselves types. Every Verse type can be used as a value of type `type`. This enables powerful generic programming through parametric functions, where types are parameters that can be passed around and constrained.

You can create variables and parameters that hold type values:

```
# Variable holding a type value
IntType:type = int
```

```
StringType:type = string
# Function that takes a type as parameter
CreateDefault(t:type):?t = false
# Usage
X:?int = CreateDefault(int)      # T = int, returns false
Y:?string = CreateDefault(string)  # T = string, returns false
```

All Verse types can be type values:

```
# Primitives
PrimitiveType:type = int

# User-defined types
MyClass := class {}
ClassType:type = MyClass

MyStruct := struct {Value:int}
StructType:type = MyStruct

# Collection types
ArrayType:type = []int
MapType:type = [string]int
TupleType:type = tuple(int, string)
OptionType:type = ?int

# Function types
FuncType:type = int->string

# Parametric types
generic_class(t:type) := class {Data:t}
ParametricType:type = generic_class(int)

# Metatypes
SubtypeValue:type = subtype(MyClass)

# Type literals
TypeLiteralValue:type = type{_(:int):string}
```

This universality makes `type` the foundation for Verse's generic programming - any type can be abstracted over.

4.8.1 Type Parameters

The most common use of `type` is in **where clauses** to create parametric (generic) functions:

```
# Identity function - works with any type
Identity(X:t where t:type):t = X

# Usage - type parameter inferred
Identity(42)        # t = int
Identity("hello")   # t = string
Identity(true)      # t = logic
```

The `where t:type` constraint means "`t` can be any Verse type." The type system infers `t` from the argument and ensures type safety throughout the function.

While `where t:type` accepts any type, you can use more specific constraints like `subtype` to limit which types are valid:

```
# Only accepts types that are subtypes of comparable
Sort(Items:[]t where t:subtype(comparable)):[]t =
    # Can use comparison operations because t is comparable
    ...
```

For comprehensive documentation on parametric functions, see the Functions chapter.

4.8.2 Type as First-Class Values

Unlike many languages where types only exist at compile time, Verse treats types as *first-class values* that can be computed, stored, and manipulated:

```
# Function that returns a type value
GetTypeForSize(Size:int):type =
    if (Size <= 8):
        int
    else:
        string

# Store type in data structure
TypeRegistry:[string]type = map{
    "Integer" => int,
    "Text" => string,
    "Flag" => logic
}
```

Passing types between functions:

```
# Helper function that takes a type parameter
CreateArray(ElementType:type, Size:int):[]ElementType =
    # This pattern works in some contexts
    ...

# Function that uses the helper
MakeIntArray():[]int =
    CreateArray(int, 10)
```

4.8.3 Returning Options of Type Parameters

A common pattern is to have functions return `?t` where `t` is a type parameter, allowing the function to work with any type while potentially failing:

```
# return type `t` must be the same type as the `Value` param type
MaybeValue(Value:t, Condition:logic where t:type):?t =
    if (Condition?) then option{Value} else false

# Usage
X:?int = MaybeValue(5, false)  # Returns false as ?int
Y:?float = MaybeValue(3.14, true)  # Returns option{3.14} as ?float
```

```
# `Value` param needs to match the type given on the `T` param. Same for return
MaybeValue(T:param_type, Value:t, Condition:logic where param_type:type):?T =
    if (Condition?):
        return option{Value} # Return the value of param type
    else:
        return false # returns empty optional

# Usage
X:?int = MaybeValue(int, 5, false)  # Returns false as ?int
Y:?float = MaybeValue(float, 3.14, true)  # Returns option{3.14} as ?float
Z:?int = MaybeValue(int, 3.14, true) # Err: params types mismatch
```

This pattern is particularly useful for generic containers and factory functions that may or may not be able to produce a value.

4.8.4 Type Constraints

The `type` constraint in where clauses is the most permissive - it accepts any Verse type. For more specific requirements, Verse provides additional constraints:

```
# Most permissive: any type
Generic(X:t where t:type):t = X

```

```
# More specific: must be subtype of comparable
RequiresComparison(X:t where t:subtype(comparable))<decides>:void =
    X = X  # Can use = because t is comparable

# Even more specific: must be exact subtype
RequiresExactType(X:t, Y:u where t:type, u:subtype(t)):t =
    X  # Y is guaranteed to be compatible with t
```

The type system enforces these constraints at compile time, preventing invalid type usage.

4.8.5 Limitations

While `type` enables powerful abstractions, there are some limitations:

Cannot construct arbitrary types generically:

```
# Cannot do this - no way to construct a value of arbitrary type t
# MakeValue(T:type):T = ???  # What would this return for T=int? T=string?
```

Cannot inspect type structure at runtime:

```
# Cannot do this - no runtime type introspection
# GetFieldNames(T:type):string = ???
```

Type parameters must be inferred or explicit:

```
# Type parameter must be determinable from usage
Identity(X:t where t:type):t = X

# OK: t inferred from argument
Identity(42)

# ERROR: t cannot be inferred from no arguments
# MakeDefault(where t:type):t = ???
```

4.9 Any

The `any` type is the *supertype of all types*. Every type in the language is a subtype of `any`. Because of this, `any` itself supports very few operations: whatever functionality `any` provides must also be implemented by every other type. In practice, there is very little you can do directly with values of type `any`. Still, it is important to understand the type, because it sometimes arises when working with code that mixes different kinds of values, or when the type checker has no more precise type to assign.

One way `any` appears is when combining values that do not share a more specific supertype. For example:

```
Letters := enum:
    A
    B
    C

letter := class:
    Value : char
    Main(Arg : int) : void =
        X := if (Arg > 0) then:
            Letters.A
        else:
            letter{Value := 'D'}
```

In this example, `X` is assigned either a value of type `Letters` or of type `letter`. Since these two types are unrelated, the compiler assigns `X` the type `any`, which is their lowest common supertype.

A more useful role for `any` is as the type of a parameter that is required syntactically but not actually used. This pattern can arise when implementing interfaces that require a certain method signature.

```
FirstInt(X:int, :any) : int = X
```

Here, the second parameter is ignored. Because it can be any value of any type, it is given the type `any`.

In more general code, the same idea can be expressed using *parametric types*, making the function flexible while still precise:

```
First(X:t, :any where t:type) : t = X
```

This version works for any type `t`, returning a value of type `t` while discarding the unused argument of type `any`.

4.10 Void

The `void` type represents the absence of a meaningful result and is used in places where no result is returned. Technically, `void` is a function that accepts any value and evaluates to `false`.

This design allows a function with return type `void` to have a body that evaluates to any type, while ensuring that callers cannot use the result. The value produced by the body is passed to `void`, which discards it and returns `false`.

A function whose purpose is to perform an effect, rather than compute a value, has return type `void`.

```
LogMessage(Msg:string) : void =
    Print(Msg)
```

Here, `LogMessage` performs an action (printing) but does not return a result. The `void` return type makes that explicit.

Chapter 5

Chapter 4: Container Types

Container types in Verse manage collections and structured data. Optionals represent values that may or may not be present. Tuples group multiple values of different types into ordered sequences. Arrays hold zero or more values with efficient indexed access. Maps associate keys with values for fast lookups. Weak maps extend regular maps with weak reference semantics for persistent storage.

Let's explore each container type in detail, starting with optionals that elegantly handle the presence or absence of values.

5.1 Optionals

An optional is an immutable container that either holds a value of type `t` or nothing at all. The type is written `?t`. Optionals are useful whenever a value may or may not be present, such as when looking up a key in a map or calling a function that can fail. By making this possibility explicit in the type, Verse allows programmers to handle "no result" situations directly and consistently, instead of relying on ad hoc error codes or special values.

You can create a non-empty optional with `option{...}`, which wraps a value into an optional. For example:

```
A:?int = option{42}    # an optional containing the integer 42
```

If you want to represent "no value," you use the special constant `false`. This is how Verse spells the empty optional:

```
var B:?int = false     # this optional has no element
B = false              # still empty
```

To extract the element of an optional, you write `?` after the optional expression. This produces a `<decides>` expression that succeeds if the optional has an element and fails otherwise. For example:

```
S := A? + 2            # succeeds with 44 because A contains 42
```

If `A` had been `false`, then the attempt to use `A?` would fail and so would the whole computation. A failing case makes this clearer:

```
X := B? + 1            # Fails because B is false and has no element
```

This shows how Verse integrates optionals tightly with the effect system: the presence or absence of a value can cause an entire computation to succeed or fail.

The `option{...}` form also works in the opposite direction. When you have a computation with the `<decides>` effect, wrapping it in `option{...}` converts it to an optional. On success you get a non-empty optional; on failure you get `false`:

```
MaybeAFloat := option{GetAFloatOrFail[]}
```

This symmetry is important. The `?` operator unwraps an optional into a `<decides>` expression, while `option{...}` wraps a `<decides>` expression into an optional. Together they provide a smooth bridge between computations that may fail and values that may be absent.

Although an optional value itself is immutable, you can keep one in a variable and change which optional the variable points to. The keyword `set` is used for this:

```
var C:?int = false
set C = option{2}      # C now refers to an optional containing 2
C? = 2                 # succeeds, since C is not empty
```

This ability is useful whenever you want to track success or failure over time, such as gradually computing a result and updating the variable only when you succeed.

A common use case is searching for something that may or may not be there. Imagine a function `Find` that looks through an array of integers and returns the index of the element you want. If the element exists, the function returns `option{index}`; if not, it returns `false`. The caller can then safely decide what to do:

```
Find(N:[]int, X:int):?int =
    for (I := 0..N.Length-1):
        if (N[I] = X) then return option{I}
    return false

Idx:?int = Find(NumberArray, 20)    # returns option{1}
Y := Idx?                           # unwraps the optional
Y = 1
```

Here the optional signals the possibility of failure directly in the type. The `?` operator makes it easy to use the result in an expression, while `option{...}` allows you to turn conditional computations back into optionals. The effect is that the idea of "maybe a value, maybe not" becomes a first-class part of the language, rather than an afterthought, and programmers are encouraged to handle the absence of values in a disciplined way.

5.2 Tuple

A tuple is a container that groups two or more values. Unlike arrays, Tuples allow you to combine values of mixed types and treat them as a unit. The elements of a tuple appear in the order in which you list them, and you access them by their position, called the index. Because the number of elements is always known at compile time, a tuple is both simple to create and safe to use.

The term *tuple* is a back formation from *quadruple*, *quintuple*, *sextuple*, and so on. Conceptually, a tuple is like an unnamed data structure with ordered fields, or like a fixed-size array where each element may have a different type.

A tuple literal is written by enclosing a comma-separated list of expressions in parentheses. For example:

```
Tuple1 := (1, 2, 3)
```

The order of elements matters, so `(3, 2, 1)` is a completely different value. Since tuples allow mixed types, you might write:

```
Tuple2 := (1, 2.0, "three")
```

Tuples can also nest inside each other:

```
X:tuple(int,tuple(int,float,string),string) = (1, (10, 20.0, "thirty"), "three")
```

Tuples are useful when you want to return multiple values from a function or when you want a lightweight grouping of values without the overhead of defining a struct or class. The type of a tuple is written with the `tuple` keyword followed by the types of the elements, but in most cases it can be inferred. For instance, you can write `MyTuple : tuple(int, float, string) = (1, 2.0, "three")`, or simply `MyTuple := (1, 2.0, "three")` and let the compiler deduce the type.

The elements of a tuple are accessed using a zero-based index operator written with parentheses. If `MyTuple := (1, 2.0, "three")`, then `MyTuple(0)` is the integer `1`, `MyTuple(1)` is the float `2.0`, and `MyTuple(2)` is the string `"three"`. Because the compiler knows the number of elements in every tuple, tuple indexing cannot fail: any attempt to use an out-of-bounds index results in a compile-time error.

Another feature of tuples is *expansion*. When a tuple is passed to a function as a single argument, its elements are automatically expanded as if the function had been called with each element separately. For example:

```
F(Arg1:int, Arg2:string):void =
    Print("{Arg1}, {Arg2}")

G():void =
    MyTuple := (1, "two")
    F(MyTuple)   # expands to F(1, "two")
```

Tuples also play a role in structured concurrency. The `sync` expression produces a tuple of results, allowing several computations that unfold over time to be evaluated simultaneously. In this way, tuples provide not only a convenient grouping mechanism but also a foundation for composing concurrent computations.

5.3 Arrays

An array is an immutable container that holds zero or more values of the same type `t`. The elements of an array are ordered, and each can be accessed by a zero-based index. Arrays are written with square brackets in their type, for example `[]int` or `[]float`, and are created with the `array{...}` literal form. For instance, `A : []int = array{}` creates an empty array, while `B : []int = array{1, 2, 3}` creates an array of three integers. Accessing elements by index is a failable operation: `B[0]` succeeds with the value `1`, while `B[10]` fails because the index is out of bounds.

Arrays can be concatenated with the `+` operator, and when declared as `var` they can be extended with the shorthand operator `+=`. For example, `var C:[]int= B + array{4}` gives `C` the value `array{1,2,3,4}`, and `set C += array{5}` updates it to `array{1,2,3,4,5}`. The length of an array is available through the `.Length` member, so `C.Length` here would be `5`. Elements are always stored in the order they are inserted, and indexing starts at `0`. Thus `array{10,20,30}[0]` is `10`, and the last valid index of any array is always one less than its length.

Although arrays themselves are immutable, variables declared with `var` can be reassigned to new arrays, or can appear to have their elements changed. For example, `var D:[]int = array{1,2,3}` allows the update `set D[0] = 3`, after which `D` will hold `array{3,2,3}`. What actually happens is that a brand new array is created under the hood, with the specified element updated. In effect, `set D[0] = 3` is compiled into `set D = array{3,D[1],D[2]}`. The old array continues to exist if another variable was referencing it, which means that if `A` and `B` both start as `array{1}` and we update `A[0]`, then `A` and `B` will diverge: `A[0]` is now `2` while `B[0]` is still `1`.

Arrays are useful whenever you want to store multiple values of the same type, such as a list of players in a game: `Players:[]player = array{Player1,Player2}`. Access is by index, for example `Players[0]` is the first player. Since indexing is failable, it is often combined with `if` expressions or iteration. For instance, the following code safely prints out every element of an array:

```
ExampleArray : []int = array{10, 20, 30}
for (Index := 0..ExampleArray.Length - 1):
    if (Element := ExampleArray[Index]):
        Print("{Element} in ExampleArray at index {Index}")
```

produces

```
10 in ExampleArray at index 0
20 in ExampleArray at index 1
30 in ExampleArray at index 2
```

Because arrays are values, "changing" them always means replacing the old array with a new one. With `var` this feels natural, since variables can be reassigned. For example, you can concatenate arrays and then update an element:

```
Array1 : []int = array{10, 11, 12}
var Array2 : []int = array{20, 21, 22}
set Array2 = Array1 + Array2 + array{30, 31}
if (set Array2[1] = 77) {}
```

After this code runs, iterating through `Array2` prints `10, 77, 12, 20, 21, 22, 30, 31`.

Arrays can also be nested to form multi-dimensional structures, similar to rows and columns of a table. For example, the following creates a two-dimensional 4×3 array of integers:

```
var Counter : int = 0
Example : [][]int =
    for (Row := 0..3):
        for (Column := 0..2):
            set Counter += 1
```

This array can be visualized as

```
Row 0:  1  2  3
Row 1:  4  5  6
Row 2:  7  8  9
Row 3: 10 11 12
```

and is accessed with two indices: `Example[0][0]` is `1`, `Example[0][1]` is `2`, and `Example[1][0]` is `4`. You can loop through all rows and columns with nested it-

eration. Arrays in Verse are not restricted to rectangular shapes: each row can have a different length, producing a jagged structure. For example,

```
Example : []int =
    for (Row := 0..3):
        for (Column := 0..Row):
            Row * Column
```

produces a triangular array with rows of increasing length: row 0 has none, row 1 has a single `0`, row 2 has `0, 2, 4`, and row 3 has `0, 3, 6, 9`.

Nested arrays with complex initialization work naturally as class field defaults:

```
# Game board with tile grid
tile_class := class:
    Position:tuple(int, int)
    var IsOccupied:logic = false

game_board := class:
    # Initialize 10×10 grid of tiles
    Tiles:[][]tile_class =
        for (Y := 0..9):
            for (X := 0..9):
                tile_class{Position := (X, Y)}

    # Get tile at specific position
    GetTile(X:int, Y:int)<computes><decides>:tile_class =
        Row := Tiles[Y]
        Row[X]

# Create board instance
Board := game_board{}

# Access specific tile
if (CenterTile := Board.GetTile[5, 5]):
    set CenterTile.IsOccupied = true
```

When you create an empty array with `array{}`, Verse infers the element type from the variable's type annotation:

```
IntArray : []int = array{}       # Empty array of integers
FloatArray : []float = array{}   # Empty array of floats
```

Without a type annotation, the compiler cannot determine what type of array you want, so you must either provide the type explicitly or include at least one element that establishes the type.

Arrays determine their element type from the common supertype of all elements. When you create an array with values of different but related types, Verse finds the most specific type that encompasses all elements:

```
# Array element type is class1 (common supertype)
MixedArray : []class1 = array{class2{}, class3{}}
```

This applies to any type hierarchy, including interfaces. If you mix completely unrelated types, the element type becomes `any`:

```
# Array of comparable - different types sharing comparable in common
DisjointArray : []comparable = array{42, 13.37, true}

# Array of any - different types with no common supertype
AnyArray : []any = array{15.61, "Message", void}
```

5.3.1 From Tuples to Arrays

Verse provides automatic conversion between tuples and arrays in specific contexts, enabling flexible function calls while maintaining type safety. This conversion is *one-way*: tuples can become arrays, but arrays cannot become tuples.

Tuples can be directly assigned to array variables when all tuple elements are compatible with the array's element type:

```
# Homogeneous tuple to array
X:tuple(int, int) = (1, 2)
Y:[]int = X            # Valid - both elements are int
Y[1] = 2               # Can use as normal array

# Longer tuples work too
NumTuple:tuple(int, int, int, int) = (1, 2, 3, 4)
NumberArray:[]int = NumTuple
NumberArray.Length = 4
```

This conversion creates an array containing all the tuple's elements in order.

When a function has a single array parameter, you can call it with multiple arguments, which automatically form an array:

```
ProcessNumbers(Numbers:[]int):int = Numbers.Length

# All these are equivalent:
ProcessNumbers(1, 2, 3)           # Multiple args → array
ProcessNumbers((1, 2, 3))         # Tuple literal → array
Values := (1, 2, 3)
ProcessNumbers(Values)             # Tuple variable → array
```

This "variadic-like" syntax provides convenience while keeping the function signature simple:

```
Sum(Nums:[]int):int =
    var Total:int = 0
    for (N : Nums):
        set Total += N
    Total

Sum(1, 2, 3, 4)                  # Returns 10
Sum((5, 6))                      # Returns 11
Values := (10, 20, 30)
Sum(Values)                      # Returns 60
```

Array conversion only succeeds when **all tuple elements are compatible** with the array's element type:

```
# Homogeneous tuple - all int
F(X:[]int):int = X.Length
F(1, 2, 3)                       # Valid

# Subtype compatibility
entity := class:
    ID:int

player := class(entity):
    Name:string

ProcessEntities(E:[]entity):int = E.Length

P := player{ID := 1, Name := "Alice"}
E := entity{ID := 2}
ProcessEntities(P, E)            # Valid - player is subtype of entity
```

Functions taking `[]any` accept **any tuple**, regardless of element types:

```
GetLength(Items:[]any):int = Items.Length

# All valid - any tuple works
GetLength(1, 2.0)                # Mixed types OK
GetLength("a", 42, true)         # Different types OK
GetLength((1, 2.0, "hello"))     # Explicit tuple OK
```

This enables generic functions that work with heterogeneous data.

When tuple elements share a common supertype (via inheritance or interface), they convert to an array of that supertype:

```
interface1 := interface:
    GetID():int

class1 := class(interface1):
    GetID<override>():int = 1

class2 := class(interface1):
    GetID<override>():int = 2

ProcessInterfaces(Items:[]interface1):int = Items.Length

X:class1 = class1{}
Y:class2 = class2{}

# Valid - both classes implement interface1
ProcessInterfaces(X, Y)           # Returns 2
```

The compiler finds the most specific common supertype and uses it for the array element type.

Tuple-to-array conversion works with nested structures:

Nested arrays:

```
# Nested tuples → nested arrays
MatrixData := ((1, 2), (3, 4))
ProcessMatrix(MatrixData)             # Valid

# Or with explicit nesting
ProcessMatrix((1, 2), (3, 4))   # Valid
```

Optional arrays:

```
ProcessOptional(Items:?[]int)<decides>:int = Items?[0]

# Optional tuple → optional array
Values := option{(1, 2)}
ProcessOptional[Values]           # Valid
```

Tuples containing arrays:

```
ProcessComplex(Data:tuple([]int, int)):int = Data(0).Length

```

```
# First element of tuple becomes array
ProcessComplex(((1, 2), 3))       # Valid - (1,2) becomes []int
```

5.3.2 Array Slicing

Arrays support slicing operations through the `.Slice` method, which extracts a contiguous portion of an array. Slicing is a failable operation—it succeeds only when the indices are valid.

The two-parameter form `Array.Slice[Start, End]` returns elements from index `Start` up to but not including index `End`:

```
Numbers : []int = array{10, 20, 30, 40, 50}
if (Slice := Numbers.Slice[1, 4]):
    Slice = array{20, 30, 40}
```

The one-parameter form `Array.Slice[Start]` returns all elements from `Start` to the end:

```
if (Slice := Numbers.Slice[2]):
    Slice = array{30, 40, 50}
```

Slicing fails if indices are negative, out of bounds, or if `Start` is greater than `End`. Creating an empty slice is valid when `Start` equals `End`:

```
NumArray.Slice[2, 2]  # Succeeds with array{}
# NumArray.Slice[2, 1]  # Would fail - Start > End
# NumArray.Slice[-1, 2] # Would fail - negative index
# NumArray.Slice[0, 10] # Would fail - End beyond array length
```

Slicing also works on strings and character tuples, returning a string:

```
"hello".Slice[1, 4] = "ell"
```

5.3.3 Array Methods

Arrays provide intrinsic methods for searching, removing, and replacing elements. These operations create new arrays rather than modifying existing ones, maintaining Verse's immutability guarantees.

The `Find()` method searches for the first occurrence of an element and returns its index, or fails if not found:

```
Array.Find(Element:t)<decides>:int
```

```
NumArray := array{1, 2, 3, 1, 2, 3}

if (Index := NumArray.Find[2]):
```

```
    # Index is 1 (first occurrence)
    Print("Found at index {Index}")

if (not NumArray.Find[0]):
    # Element not in array
    Print("Not found")

# With strings
Strings := array{"Apple", "Orange", "Strawberry"}

if (Index := Strings.Find["Strawberry"]):
    Print("Found at {Index}") # Prints "Found at 2"
```

`Find()` returns the first found index on success (`int`), or fails if the element was not found, enabling safe handling of missing elements without exceptions or special sentinel values.

`RemoveFirstElement()` removes the first occurrence:

```
Array.RemoveFirstElement(Element:t)<decides>:[]t
```

```
NumArray := array{1, 2, 3, 1, 2, 3}

if (Updated := NumArray.RemoveFirstElement[2]):
    # Updated is array{1, 3, 1, 2, 3}
    Print("Removed first 2")

if (not NumArray.RemoveFirstElement[0]):
    # Element not found
    Print("Element not in array")
```

`RemoveAllElements()` removes all occurrences:

```
NumArray := array{1, 2, 3, 1, 2, 3}
Updated := NumArray.RemoveAllElements(2)
Updated = array{1, 3, 1, 3}

# Returns unchanged array if element not found
Same := NumArray.RemoveAllElements(0)
Same = array{1, 2, 3, 1, 2, 3}
```

`Remove()` removes element at specific position:

```
Array.Remove(From:int, To:int)<decides>:[]t
```

```
NumArray := array{10, 20, 30, 40}

if (Updated := NumArray.Remove[1,1]):
    # Updated is array{10, 30, 40}

# Negative index would fail
# if (not NumArray.Remove[-1,0]):

# Out of bounds would fail
# if (not NumArray.Remove[6,10]):
```

`ReplaceFirstElement()` replace first occurrence:

```
Array.ReplaceFirstElement(OldValue:t, NewValue:t)<decides>:[]t
```

```
NumArray := array{1, 2, 3, 1, 2, 3}

if (Updated := NumArray.ReplaceFirstElement[2, 99]):
    # Updated is array{1, 99, 3, 1, 2, 3}

if (not NumArray.ReplaceFirstElement[0, 99]):
    # Element not found - fail
```

`ReplaceAllElements()` replace all occurrences:

```
Array.ReplaceAllElements(OldValue:t, NewValue:t):[]t
```

```
NumArray := array{1, 2, 3, 1, 2, 3}
Updated := NumArray.ReplaceAllElements(2, 99)
# Updated is array{1, 99, 3, 1, 99, 3}

# Returns unchanged array if element not found
Same := NumArray.ReplaceAllElements(0, 99)
# Same is array{1, 2, 3, 1, 2, 3}
```

`ReplaceElement()` replaces at specific index:

```
Array.ReplaceElement(Index:int, NewValue:t)<decides>:[]t
```

```
NumArray := array{10, 20, 30, 40}

if (Updated := NumArray.ReplaceElement[1, 99]):
    # Updated is array{10, 99, 30, 40}

if (not NumArray.ReplaceElement[-1, 99]):
    # Negative index fails
```

```
8
9  if (not NumArray.ReplaceElement[10, 99]):
10     # Out of bounds fails
```

`ReplaceAll()` is a pattern-based replacement:

```
1  NumArray := array{1, 2, 3, 4, 2, 3, 5}
2  Pattern := array{2, 3}
3  Replacement := array{99}
4  Updated := NumArray.ReplaceAll(Pattern, Replacement)
5  Updated = array{1, 99, 4, 99, 5}
6
7  # Works with different length patterns
8  NumArray2 := array{1, 2, 2, 1, 2, 2, 1}
9  Updated2 := NumArray2.ReplaceAll(array{2, 2}, array{9, 9, 9})
10 Updated2 = array{1, 9, 9, 9, 1, 9, 9, 9, 1}
11
12 # Strings are []char
13 SomeMessage := "Hey, this is a string, Hello!"
14 NewMessage := SomeMessage.ReplaceAll("He", "Apples") # Note: Case sensitive!
15 NewMessage = "Applesy, this is a string, Applesllo!"
```

`ReplaceAll()` finds contiguous subsequences matching `Pattern` and replaces each with `Replacement`. The replacement can be any length, including empty.

`Insert()` inserts an element at a specific position:

```
1  Array.Insert(Index:int, Element:[]t)<decides>:[]t
```

```
1  NumArray := array{10, 20, 40}
2
3  if (Updated := NumArray.Insert[2, array{30}]):
4      # Updated is array{10, 20, 30, 40}
5      # Inserted at index 2, existing elements shift right
6
7  # Can insert at start
8  if (Updated2 := NumArray.Insert[0, array{5}]):
9      # Updated2 is array{5, 10, 20, 40}
10
11 # Can insert at end (index = Length is valid)
12 if (Updated3 := NumArray.Insert[NumArray.Length, array{50}]):
13     # Updated3 is array{10, 20, 40, 50}
14
15 # Out of bounds fails
16 if (not NumArray.Insert[-1, array{5}]):
```

```
    # Negative index fails

if (not NumArray.Insert[NumArray.Length + 1, array{5}]):
    # Beyond Length fails
```

The `Concatenate()` function is a variadic intrinsic that combines any number of arrays into one:

```
Concatenate(Arrays:[]t...):[]t
```

Unlike the `+` operator which joins two arrays, `Concatenate()` accepts zero or more arrays:

```
# Empty call returns empty array
Empty := Concatenate()
Empty = array{}

# Single array returns that array
# Single := Concatenate(array{1, 2, 3})
# Single = array{1, 2, 3}

# Two arrays
TwoArrays := Concatenate(array{1, 2}, array{3, 4})
TwoArrays = array{1, 2, 3, 4}

# Multiple arrays
Many := Concatenate(array{1}, array{2, 3}, array{4}, array{5, 6})
Many = array{1, 2, 3, 4, 5, 6}
```

Empty arrays are handled seamlessly:

```
# Empty arrays contribute nothing
Result1 := Concatenate(array{1, 2}, array{}, array{3})
Result1 = array{1, 2, 3}
Result2 := Concatenate(array{}, array{}, array{})
Result2 = array{}

# Can concatenate many empty arrays
# EmptyResult := Concatenate(for (I := 0..100): array{})
# EmptyResult = array{}
```

Comparison with `+` operator:

```
# Using + operator (binary)
A1 := array{1, 2}
A2 := array{3, 4}
```

```
A3 := array{5, 6}
Result1 := A1 + A2 + A3  # Works but requires multiple operations

# Using Concatenate (variadic)
Result2 := Concatenate(A1, A2, A3)  # Single operation

Result1 = Result2
```

Arrays in Verse are thus immutable values with predictable behavior, but through `var` they offer the convenience of mutable variables. They can be concatenated, iterated, sliced, searched, and manipulated, making them one of the most flexible and fundamental data structures in the language.

5.4 Maps

Maps are one of the core container types, alongside arrays and optionals. If arrays are ordered sequences indexed by integers, and optionals are the smallest container of all, holding either zero or one value, then Maps generalize both ideas: like arrays, they provide efficient lookup, but instead of being limited to integer indices, they allow any *comparable* type as a key. You can think of a map as an array indexed by arbitrary keys, or as a larger optional that can hold many key–value associations at once.

A map is an immutable associative container that stores zero or more key–value pairs of type `[k]v`, written as `(Key:k, Value:v)`. Maps are the standard way to associate values with other values: you supply a key, and the map returns the value associated with it.

Maps are useful whenever you want to store data that is naturally indexed by something other than an integer position. For example, you might want to store the weights of different objects keyed by their names:

```
Empty := map{}

var Weights:[string]float = map{
    "ant" => 0.0001,
    "elephant" => 500.0,
    "galaxy" => 500000000000.0
}
```

Looking up a value in a map uses square brackets. The expression succeeds if the key is present and fails if it is not. Lookups are designed to be fast, with amortized *O(1)* time complexity:

```
Weights["ant"]  # succeeds, since "ant" key exists in map
# Weights["car"] would fail
```

If you want to update a map stored in a variable, you use `set`. This works both for adding a new key–value pair and for changing the value of an existing key. If you try to modify a key that is not present, the operation fails:

```
var Friendliness:[string]int = map{"peach" => 1000}

set Friendliness["pelican"] = 17     # succeed: add a new value with the given key
set Friendliness["peach"] += 2000    # succeed: update an existing value with the given key
# set Friendliness["tomato"] += 1000   # would fail: can't update a value which key does not ex
```

Every map also carries its size, accessible as the `Length` field:

```
Friendliness.Length = 2         # succeed: the map has 2 entries
```

When constructing a map with duplicate keys, only the last value is kept. This is because a map enforces uniqueness of keys, so earlier entries are silently overwritten:

```
WordCount:[string]int = map{
    "apple" => 0,
    "apple" => 1,
    "apple" => 2
}
# WordCount contains only {"apple" => 2}
```

Maps can also be iterated over, letting you traverse all key–value pairs exactly in the order they were inserted:

```
ExampleMap:[string]string = map{
    "a" => "apple",
    "b" => "bear",
    "c" => "candy"
}

for (Key -> Value : ExampleMap):
    Print("{Value} in ExampleMap at key {Key}")
```

This produces:

- “apple in ExampleMap at key a”
- “bear in ExampleMap at key b”
- “candy in ExampleMap at key c”

Sometimes you want to remove an entry from a map. Since maps are immutable, "removing" means creating a new map that excludes the given key. For example, here is a function that removes an element from a `[string]int` map:

```
RemoveKeyFromMap(TheMap:[string]int, ToRemove:string):[string]int =
    var NewMap:[string]int = map{}
    for (Key -> Value : TheMap, Key <> ToRemove):
        set NewMap = ConcatenateMaps(NewMap, map{Key => Value})
    return NewMap
```

The key type of a map must belong to the class `comparable`, which guarantees that two keys can be checked for equality. All basic scalar types such as `int`, `float`, `rational`, `logic`, `char`, and `char32` are comparable, and so are compound types like arrays, maps, tuples, and `structs` whose components are comparable. Classes and interfaces cannot be used as keys, since their instances do not provide a built-in notion of equality.

Not all types can be used as map keys. A type must be comparable—meaning values of that type can be checked for equality. Here's a comprehensive guide to what can and cannot be used as map keys:

Types that can be used as map keys:

- `logic` - boolean values
- `int`, `float`, `rational` - numeric types
- `char`, `char32` - character types
- `string` - text
- Enumerations - custom enum types
- Classes and Interfaces marked with `<unique>`
- `?t` where `t` is comparable - optionals of comparable types
- `[]t` where `t` is comparable - arrays of comparable elements
- `tuple(t0, t1, ...)` where all elements are comparable - tuples of comparable types
- `struct` types where all fields are comparable

Types that cannot be used as map keys:

- `false` - the empty type
- `type` - type values themselves
- Function types like `t -> u`
- `subtype(t)` - subtype expressions
- Regular classes (without `<unique>`)
- Interfaces (without `<unique>`)

Attempting to use a non-comparable type as a key results in a compile-time error.

Like arrays, maps infer their key and value types from the common supertype of all keys and values. When you create a map with mixed but related types, Verse finds the most specific types that encompass all keys and all values:

```
Instance2 := class2{}
Instance3 := class3{}

# Key type is class1 (common supertype of class2 and class3)
# Value type remains int
MixedKeyMap : [class1]int = map{Instance2 => 1, Instance3 => 2}
```

5.4.1 Ordering and Equality

Maps preserve insertion order, which is significant for both iteration and equality checks. When you insert entries into a map, they maintain the order of insertion. Two maps are equal only if they contain the same key–value pairs **in the same order**:

```
var Scores:[string]int = map{}
set Scores["Alice"] = 100
set Scores["Bob"] = 90
set Scores["Carol"] = 95

# This map equals Scores
Map1 := map{"Alice" => 100, "Bob" => 90, "Carol" => 95}
Scores = Map1

# This map does NOT equal Scores - different order
Map2 := map{"Bob" => 90, "Alice" => 100, "Carol" => 95}
not Scores = Map2
```

When a map literal contains duplicate keys, the last value overwrites earlier values, but the key's position remains from its **first** occurrence:

```
Map := map{0 => "zero", 1 => "one", 0 => "ZERO", 2 => "two"}
# Equivalent to map{0 => "ZERO", 1 => "one", 2 => "two"}
# The key 0 stays in its original position
```

Iteration over the map will visit entries in their preserved insertion order.

5.4.2 Empty Map Types

Empty maps can infer their key and value types from context, similar to arrays:

```
StringToInt : [string]int = map{}  # Empty map with inferred types

```

```
var Scores : [string]int = map{}
set Scores = ConcatenateMaps(Scores, map{"Alice" => 100})
```

Without type context, you may need to provide explicit type annotations.

5.4.3 Variance

Maps exhibit different variance behavior for keys and values. A map type `[K1]V1` is a subtype of `[K2]V2` when:

- **Keys are contravariant**: `K2` is a subtype of `K1` (more general keys → more specific keys)
- **Values are covariant**: `V1` is a subtype of `V2` (more specific values → more general values)

You can create maps with class hierarchy types as keys and values:

```
# Map with general keys, specific values: [class1]class2
GeneralKeyMap : [class1]class2 = map{class1{} => class2{}}
```

When modifying a mutable map through `set`, you can only insert keys and values that match the map's declared types:

```
var Map : [class2]int = map{}
Key2 : class2 = class2{}
Key1 : class1 = Key2

set Map[Key2] = 1      # Succeeds - exact type match
# set Map[Key1] = 2    # ERROR - cannot use supertype as key
```

5.4.4 Nested Maps

Maps can contain other maps as values, enabling multi-level associations:

```
# Map from strings to maps of ints to strings
NestedMap : [string][int]string = map{
    "numbers" => map{1 => "one", 2 => "two"},
    "letters" => map{0 => "a", 1 => "b"}
}

if (InnerMap := NestedMap["numbers"]):
    if (Value := InnerMap[1]):
        Value = "one"
```

Maps can be used as keys of other maps if all values and keys from it are comparable.

5.4.5 Concatenating Maps

The `ConcatenateMaps()` function merges multiple maps into a single map, similar to how `Concatenate()` combines arrays:

```
ConcatenateMaps(Maps:[]map(k,v)...):map(k,v)
```

`ConcatenateMaps()` is variadic—it accepts any number of maps and combines them into one. When maps contain duplicate keys, values from **later** maps override values from earlier ones:

```
Map1 := map{1 => "one", 2 => "two"}
Map2 := map{3 => "three", 4 => "four"}
Map3 := map{5 => "five"}

Combined := ConcatenateMaps(Map1, Map2, Map3)
# Combined is map{1 => "one", 2 => "two", 3 => "three", 4 => "four", 5 => "five"}
#>
# Test with two maps since three causes type inference issues
Map1 := map{1 => "one", 2 => "two"}
Map2 := map{3 => "three", 4 => "four"}

Combined := ConcatenateMaps(Map1, Map2)
Combined = map{1 => "one", 2 => "two", 3 => "three", 4 => "four"}
```

Handling duplicate keys:

```
Base := map{1 => "original", 2 => "base"}
Override := map{2 => "updated", 3 => "new"}

Result := ConcatenateMaps(Base, Override)
Result = map{1 => "original", 2 => "updated", 3 => "new"}
# Key 2 was overridden by the later map
```

The right-to-left precedence ensures that later maps take priority, enabling a natural override pattern.

Empty maps:

```
# Empty maps contribute nothing
M1 := map{1 => "a"}
M2 := map{}
M3 := map{2 => "b"}

Result := ConcatenateMaps(M1, M2, M3)  # map{1 => "a", 2 => "b"}

# Concatenating only empty maps produces an empty map
```

```
Empty := ConcatenateMaps(map{}, map{}, map{})  # map{}

# Single map returns that map
Single := ConcatenateMaps(map{1 => "one"})  # map{1 => "one"}
#>
# Test with two maps (three causes type inference issues)
M1 := map{1 => "a"}
M2 : [int]string = map{}

Result := ConcatenateMaps(M1, M2)
Result = map{1 => "a"}
```

Type constraints:

The resulting map type will coerce to the most specific shared type from the input maps:

```
# All maps have same types
M1 := map{1 => "a"}
M2 := map{2 => "b"}
Combined := ConcatenateMaps(M1, M2)  # [int]string

# Maps with different types
M3 := map{1 => "a"}
M4 := map{"string" => "b"}
Combined2 := ConcatenateMaps(M3, M4)  # [comparable]string

# Mismatched key and value types
M5 := map{1 => "a"}        # [int]string
M6 := map{5 / 3 => "b"} # [rational]string
Combined3 := ConcatenateMaps(M5, M6) # [rational]string
#>
# Test that maps with same types can be concatenated
M1 := map{1 => "a"}
M2 := map{2 => "b"}
Combined := ConcatenateMaps(M1, M2)
Combined = map{1 => "a", 2 => "b"}
```

5.5 Weak Maps

The `weak_map` type is a specialized supertype of `map` designed for persistent data storage with weak key references. It behaves similarly to ordinary maps for individual entry access, but deliberately restricts bulk operations. You cannot ask for

its length, you cannot iterate over its entries, and you cannot use `ConcatenateMaps`. These restrictions enable efficient weak reference semantics and integration with Verse's persistence system.

A `weak_map` is declared with `weak_map(k,v)` and can be initialized from an ordinary `map{}`. Updating and accessing individual entries works the same way as regular maps:

```
var MyWeakMap:weak_map(int,int) = map{}

set MyWeakMap[0] = 1
Value := MyWeakMap[0]         # succeeds with 1

set MyWeakMap = map{0 => 2}   # reassignment still works (for local variables)
```

Because `weak_map` is a supertype of `map`, you can assign regular maps to weak_map variables when needed, but you lose the ability to count or iterate once you are working with a weak map.

5.5.1 Restrictions

No Length Property:

```
var MyWeakMap:weak_map(int,int) = map{1 => 2}
# ERROR: weak_map has no Length property
# Size := MyWeakMap.Length
```

No Iteration:

```
var MyWeakMap:weak_map(int,int) = map{1 => 2, 3 => 4}
# ERROR: Cannot iterate over weak_map
# for (Entry : MyWeakMap) {}
```

Cannot Coerce to Comparable:

```
var MyWeakMap:weak_map(int,int) = map{}
# ERROR: weak_map cannot be converted to comparable
# C:comparable = MyWeakMap
```

Cannot Join with Regular Maps:

```
var MyWeakMap:weak_map(int,int) = map{1 => 2}

# ERROR: Cannot join weak_map with regular map to produce regular map
# Result:[int]int = if (true?) then MyWeakMap else map{3 => 4}
```

5.5.2 Module-Scoped weak_map Variables

When using `weak_map` as a module-scoped variable (for persistent data), there are additional restrictions:

Cannot Read Complete Map:

```
# Module-scoped persistent weak_map
var PlayerData:weak_map(player, int) = map{}

GetAllData():weak_map(player, int) =
    # ERROR: Cannot read complete module-scoped weak_map
    # PlayerData
    map{}  # Must construct new map instead
```

Cannot Write Complete Map:

```
var PlayerData:weak_map(player, int) = map{}

ResetAllData():void =
    # ERROR: Cannot replace module-scoped weak_map
    # set PlayerData = map{}
    {}
```

Individual Entry Access Works:

```
var PlayerData:weak_map(player, int) = map{}

# OK: Can read individual entries
GetPlayerScore(Player:player):int =
    if (Score := PlayerData[Player]):
        Score
    else:
        0

# OK: Can write individual entries
SetPlayerScore(Player:player, Score:int):void =
    set PlayerData[Player] = Score
```

This restriction exists because module-scoped weak_maps integrate with the persistence system, which only tracks individual entry updates, not complete map replacements.

For module-scoped `var weak_map` variables, both key and value types have strict requirements:

Key Type Must Have `<module_scoped_var_weak_map_key>` Specifier:

```
# Valid key type
persistent_class := class<unique><allocates><computes><persistent><module_scoped_var_weak_map_k

var ValidData:weak_map(persistent_class, int) = map{}

# Invalid key type - missing specifier
regular_class := class<unique><allocates><computes> {}

# ERROR: Key type lacks <module_scoped_var_weak_map_key>
# var InvalidData:weak_map(regular_class, int) = map{}
```

Value Type Must Be Persistable:

```
persistent_class := class<unique><allocates><computes><persistent><module_scoped_var_weak_map_k

# Valid: persistable value type
persistable_struct := struct<persistable>:
    Value:int

var ValidData:weak_map(persistent_class, persistable_struct) = map{}

# Invalid: non-persistable value type
regular_struct := struct:
    Value:int

# ERROR: Value type must be persistable
# var InvalidData:weak_map(persistent_class, regular_struct) = map{}
```

Common key types that satisfy the requirements:

- `player` - The standard key type for player-specific data
- `persistent_key` - Custom persistent keys with validity tracking
- `session_key` - Transient keys that don't persist across sessions

5.5.3 Covariance

The `weak_map` type is **covariant** in its key type, meaning you can use a weak_map with a subclass key type where a parent class key type is expected:

```
base_class := class<unique> {}
derived_class := class(base_class) {}

value_struct := struct {}

CreateDerivedMap():weak_map(derived_class, value_struct) =
```

```
    map{}

# OK: weak_map is covariant in key type
BaseMap:weak_map(base_class, value_struct) = CreateDerivedMap()

# ERROR 3509: Cannot go the other way (contravariance)
# DerivedMap:weak_map(derived_class, value_struct) = BaseMap
```

This covariance also allows regular maps to be assigned to weak_maps with compatible key types:

```
DerivedKey := derived_class{}
RegularMap:[derived_class]value_struct = map{DerivedKey => value_struct{}}

# OK: Regular map converts to weak_map with covariant key
WeakMap:weak_map(base_class, value_struct) = RegularMap
```

5.5.4 Partial Field Updates

When the value type is a struct or class, you can update individual fields of stored values:

```
player_data := struct<persistable>:
    Level:int
    Score:int

var PlayerData:weak_map(player, player_data) = map{}

UpdatePlayerLevel(Player:player, NewLevel:int):void =
    # Set entire struct first
    set PlayerData[Player] = player_data{Level := NewLevel, Score := 0}

    # Then update just one field
    set PlayerData[Player].Level = NewLevel + 1
```

5.5.5 Transaction and Rollback Semantics

Like all mutable state in Verse, `weak_map` updates participate in transaction semantics. If a `<decides>` expression fails, all changes are rolled back:

```
var GameData:weak_map(int, int) = map{}

AttemptUpdate():void =
    if:
        set GameData[1] = 100
```

```
        set GameData[2] = 200
        false?  # Transaction fails

    # Both updates rolled back
    # GameData[1] still does not exist
    # GameData[2] still does not exist
```

This applies to complete map replacements (for local variables), individual entries, and partial field updates.

5.5.6 Island Limits

Important

Current island limits and rules may vary and not match exactly the values shown bellow

There is a **limit on the number of persistent** `weak_map` **variables** per island. In the standard environment, this limit is 4 persistent weak_maps. Exceeding this limit produces an error:

```
key_class := class<unique><allocates><computes><persistent><module_scoped_var_weak_map_key> {}

var Map1:weak_map(key_class, int) = map{}  # OK
var Map2:weak_map(key_class, int) = map{}  # OK
var Map3:weak_map(key_class, int) = map{}  # OK
var Map4:weak_map(key_class, int) = map{}  # OK

# ERROR 3502: Exceeds island limit
# var Map5:weak_map(key_class, int) = map{}
```

Exception: If the value type is a class (not a primitive or struct), the weak_map doesn't count toward this limit:

```
value_class := class<final><persistable> {}

var Map1:weak_map(key_class, int) = map{}       # Counts (1/4)
var Map2:weak_map(key_class, int) = map{}       # Counts (2/4)
var Map3:weak_map(key_class, int) = map{}       # Counts (3/4)
var Map4:weak_map(key_class, value_class) = map{}  # Doesn't count (class value)
```

Chapter 6

Chapter 5: Operators

Operators are functions that perform actions on their operands. They provide concise syntax for common operations like arithmetic, comparison, logical operations, and assignment.

6.1 Operator Formats

Verse operators come in three formats based on their position relative to their operands:

Prefix Operators

Prefix operators appear before their single operand:

- `not Expression` - Logical negation
- `-Value` - Numeric negation
- `+Value` - Numeric positive (for alignment)

Infix Operators

Infix operators appear between their two operands:

- `A + B` - Addition
- `A * B` - Multiplication
- `A = B` - Equality comparison
- `A and B` - Logical AND

Postfix Operators

Postfix operators appear after their single operand:

- `Value?` - Query operator for logic values

6.2 Precedence

When multiple operators appear in the same expression, they are evaluated according to their precedence level. Higher precedence operators are evaluated first. Operators with the same precedence are evaluated left to right (except for assignment and unary operators which are right-associative).

The precedence levels from highest to lowest are:

Precedence	Operators	Category	Format	Associativity	Example
11	`.`, `[]`, `()`, `{}`, `?` (postfix)	Member access, Indexing, Call, Construction, Query	Postfix	Left	`BossDefeated?`, `Player.Respawn()`
10	`+`, `-` (unary), `not`	Unary operations	Prefix	Right	`+Score`, `-Distance`, `not HasCooldown?`
9	`*`, `/`	Multiplication, Division	Infix	Left	`Score * Multiplier`
8	`+`, `-` (binary)	Addition, Subtraction	Infix	Left	`X + Y`, `Health - Damage`
7	`=` (relational), `<>`, `<`, `<=`, `>`, `>=`	Relational comparison	Infix	Right	`Player <> Target`, `Score > 100`
5	`and`	Logical AND	Infix	Left	`HasPotion? and TryUsePotion[]`
4	`or`	Logical OR	Infix	Left	`IsAlive? or Respawn()`

Precedence	Operators	Category	Format	Associativity	Example
3	`..`	Range	Infix	Left	`0..100`, `-15..50`
2	~~Lambda expressions~~	~~Function literals~~ (not yet supported)	Special	N/A	N/A
1	`:=`, `set =`	Assignment	Infix	Right	`X := 15`, `set Y = 25`

6.3 Arithmetic Operators

Arithmetic operators perform mathematical operations on numeric values. They work with both `int` and `float` types, with some special behaviors for type conversion and integer division.

6.3.1 Basic Arithmetic

Operator	Operation	Types	Notes
`+`	Addition	`int`, `float`	Also concatenates strings and arrays
`-`	Subtraction	`int`, `float`	Can be used as unary negation
`*`	Multiplication	`int`, `float`	Converts `int` to `float` when mixed
`/`	Division	`int` (failable), `float`	Integer division returns `rational`

```
# Basic arithmetic
Sum := 10 + 20      # 30
Diff := 50 - 15     # 35
Prod := 6 * 7       # 42
Quot := 20.0 / 4.0  # 5.0

# Unary operators
Negative := -42     # -42
Positive := +42     # 42 (for alignment)

# Integer division (failable, returns rational)
if (Result := 10 / 3):
    IntResult := Floor(Result)  # 3

# Type conversion through multiplication
IntValue:int = 42
FloatValue:float = IntValue * 1.0  # Converts to 42.0
```

6.3.2 Compound Assignments

Compound assignment operators combine an arithmetic operation with assignment:

Operator	Equivalent To	Types
`set +=`	`set X = X + Y`	`int`, `float`, `string`, `array`
`set -=`	`set X = X - Y`	`int`, `float`
`set *=`	`set X = X * Y`	`int`, `float`
`set /=`	`set X = X / Y`	`float` only

```
var Score:int = 100
set Score += 50    # Score is now 150
set Score -= 25    # Score is now 125
set Score *= 2     # Score is now 250

var Health:float = 100.0
set Health /= 2.0  # Health is now 50.0

# Note: set /= doesn't work with integers due to failable division
# var IntValue:int = 10
# set IntValue /= 2  # Compile error!
```

6.4 Comparison Operators

Comparison operators test relationships between values and are failable expressions that succeed or fail based on the comparison result.

6.4.1 Relational Operators

Operator	Meaning	Supported Types	Example
`<`	Less than	`int`, `float`	`Score < 100`
`<=`	Less than or equal	`int`, `float`	`Health <= 0.0`
`>`	Greater than	`int`, `float`	`Level > 5`
`>=`	Greater than or equal	`int`, `float`	`Time >= MaxTime`

6.4.2 Equality Operators

Operator	Meaning	Supported Types	Example
`=`	Equal to	All comparable types	`Name = "Player1"`
`<>`	Not equal	All comparable types	`State <> idle`

```
# Numeric comparisons
if (Score > HighScore):
    Print("New high score!")

if (Health <= 0.0):
    HandlePlayerDeath()

# Example with other comparable types
if (PlayerName = "Admin"):
    EnableAdminMode()

if (CurrentState <> GameState.Playing):
    ShowMenu()

# Comparison in complex expressions
if (Level >= 10 and Score > 1000):
    UnlockAchievement()
```

The following types support equality comparison operations (`=` and `<>`):

- Numeric types: `int`, `float`, `rational`
- Boolean: `logic`

- Text: `string`, `char`, `char32`
- Enumerations: All `enum` types
- Collections: `array`, `map`, `tuple`, `option` (if elements are comparable)
- Structs: If all fields are comparable
- Unique classes: Classes marked with `<unique>` (identity equality only)

Comparisons between different types generally fail:

```
0 = 0.0  # Fails: int vs float
"5" = 5  # Fails: string vs int
```

6.5 Logical Operators

Logical operators work with failable expressions and control the flow of success and failure.

6.5.1 Query Operator (?)

The query operator checks if a `logic` value is `true` (see Failure for how `?` works with other types):

```
var IsReady:logic = true

if (IsReady?):
    StartGame()

# Equivalent to:
if (IsReady = true):
    StartGame()
```

6.5.2 Not Operator

The `not` operator negates the success or failure of an expression:

```
if (not IsGameOver?):
    ContinuePlaying()

# Effects are not committed with not
var X:int = 0
if (not (set X = 5, IsGameOver?)):
    # X is still 0 here, even though the assignment "tried" to happen
    Print("X is {X}")  # Prints "X is 0"
```

6.5.3 And Operator

The `and` operator succeeds only if both operands succeed:

```
if (HasKey? and DoorUnlocked?):
    EnterRoom()

# Both expressions must succeed
if (Player.Level > 5 and Player.HasItem?):
    AllowQuestAccess()
```

6.5.4 Or Operator

The `or` operator succeeds if at least one operand succeeds:

```
if (HasKeyCard? or HasMasterKey?):
    OpenDoor()

# Short-circuit evaluation - second operand not evaluated if first succeeds
if (QuickCheck[] or ExpensiveCheck[]):
    ProcessResult()
```

6.5.5 Truth Table

Consider two expressions `P` and `Q` which may either succeed or fail, the following table shows the result of logical operators applied to them:

Expression P	Expression Q	P and Q	P or Q	not P
Succeeds	Succeeds	Succeeds (Q's value)	Succeeds (P's value)	Fails
Succeeds	Fails	Fails	Succeeds (P's value)	Fails
Fails	Succeeds	Fails	Succeeds (Q's value)	Succeeds
Fails	Fails	Fails	Fails	Succeeds

6.6 Assignment and Initialization

The `:=` operator initializes constants and variables:

```
# Constant initialization (immutable)
MaxHealth:int = 100
PlayerName:string = "Hero"

```

```
# Variable initialization (mutable)
var CurrentHealth:int = 100
var Score:int = 0

# Type inference
AutoTyped := 42  # Inferred as int
```

The `set =` operator updates variable values:

```
var Points:int = 0
set Points = 100

var Position:vector3 = vector3{X := 0.0, Y := 0.0, Z := 0.0}
set Position = vector3{X := 10.0, Y := 20.0, Z := 0.0}
```

6.7 Special Operators

6.7.1 Indexing

The square bracket operator is used for multiple purposes in Verse:

1. **Array/Map indexing** - Access elements in collections
2. **Function calls** - Call functions which may fail
3. **Computed member access** - Access object members dynamically

```
# Array indexing (failable)
MyArray := array{10, 20, 30}
if (Element := MyArray[1]):
    Print("Element at index 1: {Element}")  # Prints 20

# Map lookup (failable)
Scores:[string]int = map{"Alice" => 100, "Bob" => 85}
if (AliceScore := Scores["Alice"]):
    Print("Alice's score: {AliceScore}")

# String indexing (failable)
Name:string = "Verse"
if (FirstChar := Name[0]):
    Print("First character: {FirstChar}")  # Prints 'V'

# Function call that can fail
Result1 := MyFunction1[Arg1, Arg2]          # Can fail
```

```
Result2 := MyFunction2[?X:=Arg1, ?Y:=Arg2]  # Named arguments
EmptyCall := MyFunction2[]                  # and optional values
```

6.7.2 Member Access

The dot operator accesses fields and methods of objects:

```
Player.Health
Player.GetName()
MyVector.X
Config.Settings.MaxPlayers

# Line continuation supported after dot
LongExpression := MyObject.FirstMethod().
                           SecondMethod()
```

6.7.3 Range

The range operator creates ranges for iteration:

```
# Inclusive range
for (I := 0..4):
    Print("{I}")  # Prints 0, 1, 2, 3, 4
```

6.7.4 Object Construction

Curly braces are used to construct objects when placed after a type:

```
# Object construction with type name
Point := point{X:= 10, Y:= 20}

# Fields can be separated by commas or newlines
Player := player_data {
    Name := "Hero"
    Level := 5
    Health := 100.0
}

# Trailing commas are not allowed
Config := game_config{
    MaxPlayers := 100,
    EnablePvP := true # ,  -- comma not allowed here
}
```

6.7.5 Tuple Access

Round braces when used with a single argument after a tuple expression, accesses tuple elements:

```
MyTuple := (10, 20, 30)
FirstElement := MyTuple(0)  # Access first element
SecondElement := MyTuple(1)  # Access second element
```

6.8 Type Conversions

Verse has limited implicit type conversion. Most conversions must be explicit:

```
# No implicit int to float conversion
MyInt:int = 42
# MyFloat:float = MyInt  # Error!
MyFloat:float = MyInt * 1.0  # OK: explicit conversion

# No implicit numeric to string conversion
Score:int = 100
# Message:string = "Score: " + Score  # Error!
Message:string = "Score: {Score}"  # OK: string interpolation
```

When operators work with mixed types, specific rules apply:

```
# int * float -> float
Result := 5 * 2.0  # Result is 10.0 (float)

# Comparisons must be same type
if (5 = 5):     # OK
if (5.0 = 5.0): # OK
# if (5 = 5.0):   # Error: different types
```

Part II

Part II: Core Features

Chapter 7

Chapter 6: Mutability

Immutability is the default in Verse. When you create a value, it stays that value forever — unchanging, predictable, and safe to share. This foundational principle makes programs easier to reason about, eliminates entire categories of bugs, and enables powerful optimizations. But games are dynamic worlds where state constantly evolves: health decreases, scores increase, inventories change. Verse embraces both paradigms, providing immutability by default while offering controlled, explicit mutation when you need it.

The distinction between immutable and mutable data in Verse goes deeper than just whether values can change. It fundamentally affects how data flows through your program, how values are shared between functions, and how the compiler reasons about your code. Understanding this distinction is crucial for writing efficient, correct Verse programs.

7.1 The Pure Foundation

In Verse's pure fragment, computation happens without side effects. Values are created but never modified. Functions transform inputs into outputs without changing anything along the way. This isn't a limitation — it's a powerful foundation that makes code predictable and composable.

```
# Immutable values and structures
point := struct<computes>:
    X:float = 0.0
    Y:float = 0.0

Origin := point{}
UnitX := point{X := 1.0}
UnitY := point{Y := 1.0}
```

```
9
10  # These values are eternal - Origin will always be (0, 0)
11  Distance(P1:point, P2:point)<reads>:float =
12      DX := P2.X - P1.X
13      DY := P2.Y - P1.Y
14      Sqrt(DX * DX + DY * DY)
```

In this pure world, equality means structural equality — two values are equal if they have the same shape and content. For primitive types and structs, this happens automatically. For classes, which have identity beyond their content, equality requires more careful consideration.

```
1   # Recursive data structures using classes
2   linked_list := class:
3       Value:int = 0
4       Next:?linked_list = false
5
6       # Custom equality check for structural comparison
7       Equals(Other:linked_list)<computes><decides>:void =
8           Self.Value = Other.Value
9           # Both have no next, or both have next and those are equal
10          if (Self.Next?):
11              Tmp := Self.Next?
12              OtherNext := Other.Next?
13              Tmp.Equals[OtherNext]
14          else:
15              not Other.Next?
16
17  List1 := linked_list{Value := 1, Next := option{linked_list{Value := 2}}}
18  List2 := linked_list{Value := 1, Next := option{linked_list{Value := 2}}}
19
20  if (List1.Equals[List2]):
21      Print("Structurally equal")  # This succeeds
```

Pure computation forms the backbone of functional programming in Verse. It's predictable, testable, and parallelizable. When a function is marked `<computes>`, you know it will always produce the same output for the same input, with no hidden dependencies or surprising behaviors.

7.2 Introducing Mutation

Mutation enters through two keywords: `var` and `set`. The `var` annotation declares that a variable can be reassigned. The `set` keyword performs that reassignment. Together, they provide controlled mutation with clear visibility.

```
Score:int = 100   # Immutable variable - cannot be reassigned
                  # Mutable variable - can be reassigned
var Health:float = 100.0       # type annotation is required
set Health = 75.0              # Allowed
```

Every use of `var` and `set` has implications for effects. Reading from a `var` variable requires the `<reads>` effect. Using `set` requires both `<reads>` and `<writes>` effects. This isn't bureaucracy — it's transparency. The effects make mutation visible in function signatures, so callers know when functions might observe or modify state.

7.2.1 Requirements for var Declarations

Mutable variable declarations have strict requirements that prevent common errors:

Must provide explicit type:

```
# Valid - explicit type
var X:int = 0

# Invalid - cannot use := syntax with var
# var X := 0  # Error
```

The type inference syntax `:=` cannot be used with `var`. You must explicitly declare the type.

Must provide initial value:

```
# Valid - initialized
var Health:float = 100.0

# Invalid - no initial value
# var Score:int  # Error
```

Every `var` declaration requires an initial value. Uninitialized mutable variables are not allowed.

Cannot be completely untyped:

```
# Invalid - neither type nor value
# var X
```

7.2.2 var Declarations as Expressions

Variable declarations with `var` can be used as expressions, evaluating to their initial value:

```
X := (var Y:int = 42)  # X = 42, Y declared and mutable
X = 42
```

However, `var` declarations **cannot be the target of** `set`:

```
# Invalid
# set (var Z:int = 0) = 1  # Error
```

The `var` keyword declares a new mutable variable; you cannot assign to the declaration itself.

7.2.3 set with Block Expressions

The `set` statement can use block expressions, which allows complex computations and side effects:

```
var X:int = 0
var Y:int = 1

set X = block:
    set Y = X      # Side effect: Y becomes 0
    2              # Block result: X becomes 2

X = 2 and Y = 0
```

This pattern is useful when the new value requires intermediate computations or when you need multiple side effects during assignment.

7.2.4 Scope and Redeclaration Restrictions

Cannot redeclare in same scope:

Variables cannot be redeclared with `:=` once they exist in scope:

```
var X:int = 0

# Invalid - X already exists
# X := 1  # Error
```

This applies even in conditional branches:

```
var A:int = 1

if (SomeCondition?):
    # Invalid - A already declared in outer scope
    # A := 2  # Error
```

Cannot redeclare with assignment syntax:

```
var A:int = 1
var B:int = 2

# Invalid - looks like assignment but A already exists
# A := B  # ERROR
```

Use `set A = B` instead to assign to existing mutable variables.

Cannot nest var declarations:

```
# Invalid
# var (var X):int = 0  # ERROR 3549
```

The `var` keyword cannot be nested within itself.

7.3 Deep vs Shallow Mutability

Verse's approach to mutability differs significantly between structs and classes, reflecting their different roles in the language.

7.3.1 Struct Mutability: Deep and Structural

When you declare a struct variable with `var`, you're declaring the entire structure as mutable — the variable itself and all its nested fields, recursively. This deep mutability means you can modify any part of the structure tree.

```
player_stats := struct<computes>:
    Level:int = 1
    Position:point = point{}
    Inventory:[]string = array{}

# Immutable struct variable - nothing can change
Stats1:player_stats = player_stats{}
# set Stats1.Level = 2  # ERROR: Cannot modify immutable struct

# Mutable struct variable - everything can change
var Stats2:player_stats = player_stats{}
set Stats2.Level = 2  # OK
set Stats2.Position.X = 100.0  # OK - nested fields are mutable
set Stats2.Inventory = Stats2.Inventory + array{"Sword"}  # OK
```

When you assign one struct variable to another, Verse performs a deep copy. The two variables become independent, each with their own copy of the data. Changes to one don't affect the other.

```
var Original:player_stats = player_stats{Level := 5}
var Copy:player_stats = Original

set Copy.Level = 10
Original.Level = 5   # unchanged, they're independent copies
```

This deep-copy semantics extends to all value types: structs, arrays, maps, and tuples. When you pass a struct to a function, the function receives its own copy. When you store a struct in a container, the container holds a copy. This prevents aliasing and makes reasoning about struct mutations local and predictable.

7.3.2 Class Mutability: Reference Semantics

Classes behave differently. They have reference semantics — when you assign a class instance, you're sharing a reference to the same object, not creating a copy. The `var` annotation on a class variable only affects whether that variable can be reassigned to reference a different object. It doesn't affect the mutability of the object's fields.

```
game_character := class:
    Name:string = "Hero"
    var Health:float = 100.0  # This field is always mutable
    MaxHealth:float = 100.0   # This field is always immutable

# Immutable variable, but mutable fields can still change
Player1:game_character = game_character{}
# set Player1 = game_character{}  # ERROR: Cannot reassign non-var variable
set Player1.Health = 50.0  # OK: Health field is mutable

# Mutable variable allows reassignment
var Player2:game_character = Player1  # Same object
set Player2 = game_character{Name := "Villain"}  # OK: Can reassign
set Player2.Health = 75.0  # OK: Modifies the new object

# Player1 and the original Player2 reference were the same object
# After reassignment, Player2 references a different object
```

The key insight: for classes, field mutability is determined at class definition time, not at variable declaration time. A `var` field is always mutable, regardless of how you access it. A non-`var` field is always immutable, even if accessed through a `var` variable.

```
container := class:
    ImmutableData:point= point{}  # Always immutable
    var MutableData:int = 0       # Always mutable
```

```
4
5  # Even through an immutable variable, mutable fields can change
6  Box:container = container{}
7  set Box.MutableData = 42         # Allowed
8  # set Box.ImmutableData = Point{X := 1.0}  # ERROR: Field is immutable
```

7.3.3 Collection Mutability: Arrays and Maps

Arrays and maps follow struct semantics—they are values, not references. When you copy a collection, you get an independent copy. Mutations to one copy don't affect the other.

Basic Array Mutation

Mutable arrays allow element replacement:

```
1  var Numbers:[]int = array{0, 1}
2  Numbers[0] = 0
3  Numbers[1] = 1
4
5  set Numbers[0] = 42
6  Numbers[0] = 42
7  Numbers[1] = 1  # Unchanged
8
9  set Numbers[1] = 666
10 Numbers[0] = 42
11 Numbers[1] = 666
```

You cannot add elements beyond the array's current length:

```
1  var A:[]int = array{0}
2  not (set A[1] = 1)  # Fails - index out of bounds
3  # Must use concatenation: set A = A + array{1}
```

Basic Map Mutation

Mutable maps allow both updating existing keys and adding new keys:

```
1  var Scores:[int]int = map{0 => 1, 1 => 2}
2  set Scores[1] = 42
3  Scores[1] = 42
4
5  # Adding new keys
6  set Scores[2] = 100
7  Scores[2] = 100
8
```

```
# Map with string keys
var Config:[string]int = map{"volume" => 50}
set Config["brightness"] = 75
```

Looking up a non-existent key doesn't add it:

```
M:[int]int := map{}
not (M[0] = 0)  # Key doesn't exist, comparison fails
# M is still empty - lookup didn't add the key
```

Nested Collection Mutation

Collections can be nested, and `set` works through multiple levels:

Map of arrays:

```
var Data:[int][]int = map{}
set Data[666] = array{42}
Data[666] = array{42}

# Mutate nested array element
set Data[666][0] = 1234
Data = map{666 => array{1234}}
Data[666] = array{1234}
```

Array of maps:

```
var Grid:[][int]int = array{map{}}

# Replace entire map at index
set Grid[0] = map{42 => 666}
Grid[0] = map{42 => 666}
Grid[0][42] = 666

# Add new key to nested map
set Grid[0][1234] = 4321
Grid[0] = map{42 => 666, 1234 => 4321}
Grid[0][42] = 666
Grid[0][1234] = 4321

# Update existing key in nested map
set Grid[0][42] = 1122
Grid[0][42] = 1122
```

Array of arrays:

```
var Matrix:[][]int = array{array{1234}}
set Matrix[0][0] = 42
Matrix = array{array{42}}
Matrix[0] = array{42}
Matrix[0][0] = 42

# Replace inner array
set Matrix[0] = array{666}
Matrix[0] = array{666}
Matrix[0][0] = 666
```

Important: All nested levels should exist to use `set`, if any of the higher levels don't exist, the entire set will fail

```
var Grid:[string][]int = map{"apples"=>array{1,2,3,4}}

set Grid["bananas"] = array{} # ok - no nesting
set Grid["apples"][2] = 7 # ok - changes nested array "3" to "7"

set Grid["oranges"][0] = 10 # fail: "oranges" key not found in map

# Alternative (make sure that higher levels exist first):
if (not Grid["oranges"]):
    set Grid["oranges"] = array{}
set Grid["oranges"][0] = 10 # succeeds
```

Value Semantics for Collections

Extracting a value from a mutable collection creates an independent copy:

```
var X:[][int]int = array{map{42 => 1122, 1234 => 4321}}

# Y gets a copy of the map, not a reference
Y := X[0]
Y = map{42 => 1122, 1234 => 4321}

# Mutating X doesn't affect Y
set X[0][0] = 111
X[0] = map{42 => 1122, 1234 => 4321, 0 => 111}
Y = map{42 => 1122, 1234 => 4321}  # Unchanged

# Replacing entire element doesn't affect Y
set X[0] = map{42 => 4242}
```

```
X[0] = map{42 => 4242}
Y = map{42 => 1122, 1234 => 4321}  # Still unchanged
```

This is different from class reference semantics—collections copy, classes share.

Collections with Mutable Values

When collections contain classes or structs with mutable fields, you can mutate through the collection:

```
C := the_class{}
set C.X[0] = 4266642
C.X[0] = 4266642
```

Map values with mutable members:

```
var M:[int]class0 = map{0 => class0{}}
M[0].AM = 20

# Mutate class field through map
set M[0].AM = 30
M[0].AM = 30
```

The map constructed from a `var` doesn't track changes to the source variable:

```
var I0:int = 42
M:[int]int = map{0 => I0}
M[0] = 42

set I0 = 0
M[0] = 42  # Still 42! Map has a copy of the value
```

7.3.4 Arrays of Structs: Independent Copies

When you store structs in an array, each element is an independent copy:

```
S0 := struct0{A := 88}
var A0:[]struct0 = array{S0, S0}

# All three have the value 88, but are independent
S0.A = 88
A0[0].A = 88
A0[1].A = 88

# Mutating one doesn't affect the others
set A0[0].A = 99
```

```
S0.A = 88     # Unchanged
A0[0].A = 99  # Changed
A0[1].A = 88  # Unchanged
```

7.3.5 Arrays of Classes: Shared References

Arrays of classes behave very differently—all references to the same object share mutations:

```
C0 := class0{}
var A1:[]class0 = array{C0, C0, C0}

# All three array elements reference the same object
A1[0].AM = 20
A1[1].AM = 20
A1[2].AM = 20

# Mutating through one affects all references
set A1[0].AM = 30
A1[0].AM = 30
A1[1].AM = 30  # Changed!
A1[2].AM = 30  # Changed!

set A1[1].AM = 40
A1[0].AM = 40  # All three see the change
A1[1].AM = 40
A1[2].AM = 40

# Replacing an element breaks the sharing for that element
set A1[1] = class0{}
A1[0].AM = 40  # Still references original
A1[1].AM = 20  # New object with default value
A1[2].AM = 40  # Still references original
```

This is a critical distinction: **structs in collections are copies, classes in collections are shared references**.

7.3.6 Compound Assignment Operators

Verse supports compound assignment operators that combine arithmetic with mutation:

```
var S0:struct0 = struct0{}

```

```
set S0.A += 10
S0.A = 20

set S0.A -= 3
S0.A = 17

set S0.A *= 4
S0.A = 68
```

Available compound operators:

- `set +=` - Addition assignment (int, float, string, array)
- `set -=` - Subtraction assignment (int, float)
- `set *=` - Multiplication assignment (int, float)
- `set /=` - Division assignment (float only)

Important: `set /=` doesn't work with integers because integer division is failable.

Compound assignments work anywhere regular assignment does:

```
var Score:int = 100
set Score += 50
set Score *= 2

var Data:[]int = array{1, 2, 3}
set Data += array{4, 5}  # Array concatenation
Data = array{1, 2, 3, 4, 5}

var Numbers:[][]int = array{array{1}}
set Numbers[0][0] *= 42
Numbers[0][0] = 42
```

7.3.7 Tuple Mutability: Replacement Only

Tuples can be replaced entirely but individual elements cannot be mutated:

```
var T0:tuple(int, int) = (10, 20)
T0(0) = 10
T0(1) = 20

# Can replace entire tuple
set T0 = (30, 40)
T0(0) = 30
T0(1) = 40
```

Cannot mutate elements:

```
var T0:tuple(int, int) = (50, 60)
set T0(0) = 70  # ERROR: Cannot mutate tuple elements
```

This restriction applies even when the tuple is mutable. You must replace the entire tuple to change its contents.

7.3.8 Map Ordering and Mutation

Maps preserve **insertion order**, and this order is maintained through mutations:

New Keys Append to End

```
var M:[int]int = map{2 => 2}

set M[1] = 1  # Appends to end
set M[0] = 0  # Appends to end

# Iteration order is insertion order: 2, 1, 0
Keys := array{2, 1, 0}
var Index:int = 0
for (Key->Value : M):
    Keys[Index] = Key
    set Index += 1

M = map{2 => 2, 1 => 1, 0 => 0}
```

Updating Existing Keys Preserves Position

```
var M:[string]int = map{"a" => 3, "b" => 1, "c" => 2}

# Mutating value keeps key position
set M["a"] = 0
M = map{"a" => 0, "b" => 1, "c" => 2}  # Same order

# Another update
set M["c"] = 0
set M["a"] = 2
M = map{"a" => 2, "b" => 1, "c" => 0}  # Still same order
```

Order Matters for Equality

Map equality considers both keys/values **and order**:

```
var M:[string]int = map{"a" => 3, "b" => 1, "c" => 2}
set M["a"] = 0

# Same keys and values, same order = equal
M = map{"a" => 0, "b" => 1, "c" => 2}

# Same keys and values, different order = not equal
M <> map{"b" => 1, "c" => 2, "a" => 0}
```

7.4 Critical Mutability Restrictions

Verse imposes several important restrictions on where and how mutation can occur. These aren't arbitrary—they prevent unsound behaviors and maintain type safety.

7.4.1 Cannot Mutate Immutable Class Fields

Classes might contain unique pointers or other resources that cannot be safely cloned. Therefore, you cannot mutate immutable fields of a class instance:

```
classX := class:
    AI:int = 20  # Immutable field

CX:classX = classX{}
CX.AI = 20
set CX.AI = 30  # Error: Cannot mutate immutable class field
```

This restriction applies even when the class instance itself is immutable. Only `var` fields of classes can be mutated.

7.4.2 Only Structs Allow Field Mutation

Only structs marked `<computes>` (pure structs) allow field mutation through a variable:

```
# OK: <computes> struct allows field mutation
s1 := struct<computes>{M:int = 0, J:float = 3}

var S1:s1 = s1{}

Old := S1 # makes a copy of the struct

set S1.M = 1 # makes a copy of the struct, but updates `M` in the process

```

```
S1.M = 1 # Succeeds
Old = S1 # Fails (structs does not pass as references)
```

When a new struct is constructed, it is assigned the updated value and copied other fields. If there is other places referencing the old struct, they will not have the updated values (unlike classes)

This restriction ensures that only predictable, effect-free structs can be mutated.

7.4.3 Cannot Have Immutable Class in Mutation Path

When mutating nested structures, you cannot have an immutable class in the "middle" of the path:

```
struct0 := struct<computes>{A:int = 10}
struct1 := struct<computes>{S0:struct0 = struct0{}}
class0 := class{CI:struct1 = struct1{}}  # Immutable class
struct2 := struct<computes>{C0:class0 = class0{}}
struct3 := struct<computes>{S2:struct2 = struct2{}}

var S3:[]struct3 = array{struct3{}, struct3{}}
set S3[1].S2.C0.CI.S0.A = 7  # ERROR: class0 is immutable
```

But you CAN mutate through `var` members of that class.

Even with a mutable index, you cannot mutate an immutable array:

```
I:int = 2
A:[]int = array{5, 6, 7}
set A[I] = 2  # ERROR: A is not var
```

The array itself must be declared `var` to allow element mutation:

```
I:int = 2
var A:[]int = array{5, 6, 7}
set A[I] = 2  # OK: A is var
```

7.5 Identity and Uniqueness

The `<unique>` specifier gives classes identity-based equality. Without it, classes can't be compared for equality at all (you'd need to write custom comparison methods). With it, equality means identity — two references are equal only if they refer to the exact same object.

```
unique_item := class<unique>:
    var Count:int = 0

```

```
Item1:unique_item = unique_item{}
Item2:unique_item = Item1  # Same object
Item3:unique_item = unique_item{}  # Different object

if (Item1 = Item2):
    Print("Same object")  # This prints

if (Item1 = Item3):
    Print("Same object")  # This doesn't print - different objects
```

This identity-based equality is crucial for game objects that need distinct identities even when their data is identical. Two monsters might have the same stats, but they're still different monsters.

Chapter 8

Chapter 7: Functions

Functions are reusable code blocks that perform actions and produce outputs based on inputs. Think of them as abstractions for behaviors, much like ordering food from a menu at a restaurant. When you order, you tell the waiter what you want from the menu, such as `OrderFood("Ramen")`. You don't need to know how the kitchen prepares your dish, but you expect to receive food after ordering. This abstraction is what makes functions powerful - you define the instructions once and reuse them in different contexts throughout your code.

8.1 Parameters

Functions can accept any number of parameters, from none at all to as many as needed. The syntax follows a straightforward pattern where each parameter has an identifier and a type, separated by commas:

```
ProcessData(Name:string, Age:int, Score:float):string =
    "{Name} is {Age} years old with a score of {Score}"
```

For functions with many parameters or optional configuration, Verse supports named and default parameters.

8.1.1 Named Parameters

Named parameters with defaults make functions more flexible and ergonomic. They allow you to:

- Specify arguments by name rather than position
- Provide default values for optional parameters
- Call functions with only the arguments you need
- Add new optional parameters without breaking existing code

Named parameters are declared with a `?` prefix and called with the name and a `:=` followed by a value:

```
# A function with named parameters
Greet(?Name:string, ?Greeting:string):string = "{Greeting} {Name}!"

# A call with named arguments
Greet(?Name := "Alice", ?Greeting := "Hello")
```

Named parameters with default values are truly optional:

```
# Named parameters with defaults
Log(Message:string, ?Level:int=1, ?Color:string="white"):string =
    "[Level {Level}] {Message} ({Color})"

# Call with all defaults
Log("Starting")                          # Returns "[Level 1] Starting (white)"

# Call with some arguments
Log("Warning", ?Level:=2)                # Returns "[Level 2] Warning (white)"

# Call with arguments in any order
Log("Error", ?Color:="red", ?Level:= 3)  # Returns "[Level 3] Error (red)"
```

After the first named parameter, all subsequent parameters must also be named:

```
Invalid: named followed by positional
Invalid(? Named:int, Positional:string):void = {}  # ERROR
```

When calling functions with named parameters, you must use the `?Name:=Value` syntax. All parameters without default must be specified. Positional arguments come first:

```
Configure(Required:int, ?Option1:string, ?Option2:logic):void = { }

# Valid
Configure(42, ?Option1:="test", ?Option2:=true)

# Invalid: named arg before positional
# Configure(?Option1:="test", 42, ?Option2:=true)  # ERROR
```

Default values are evaluated in the function's defining scope; they can reference:

- Module-level definitions
- Class or interface members
- Earlier parameters

```
# Module-level definition
ModuleTimeout:int = 30

# Access module-level definition
Connect(?Host:string, ?Timeout:int = ModuleTimeout):void =...

# Access member definition
game_config := class:
    DefaultLives:int = 3

    StartGame(?Lives:int = DefaultLives):void =...

# Access earlier parameter
CreateRange(?Start:int, ?End:int = Start + 10):[]int =...
```

Default values work with overridden members in class hierarchies:

```
base_game := class:
    DefaultSpeed:float = 1.0

    Move(?Speed:float = DefaultSpeed):void =...
    # Uses DefaultSpeed from current instance

fast_game := class(base_game):
    DefaultSpeed<override>:float = 2.0

base_game{}.Move()         # Uses 1.0
fast_game{}.Move()         # Uses 2.0 (overridden value)
```

Named and default parameters interact with the type system. A function with default parameters is a subtype of the same function without those parameters:

```
Process(?Required:int, ?Optional:int = 0):int = Required + Optional

# Can assign to type without optional parameter
F1:type{_(?Required:int):int} = Process
F1(?Required := 5)                          # Returns 5 (uses default)

# Can assign to type with optional parameter
F2:type{_(?Required:int, ?Optional:int):int} = Process
F2(?Required := 5, ?Optional := 3)          # Returns 8

# Can even assign to type with no parameters (all have defaults)
DefaultAll(?A:int = 1, ?B:int = 2):int = A + B
```

```
F3:type{_():int} = DefaultAll
F3()                                        # Returns 3
```

Function types preserve named parameter names:

```
Calculate(?Amount:float, ?Rate:float):float = Amount * Rate

# Valid: names match
F1:type{_(?Amount:float, ?Rate:float):float} = Calculate

# Invalid: different names
# F2:type{_(?Value:float, ?Factor:float):float} = Calculate  # ERROR
```

Function types do not include default values:

```
F1(?X:int=1):int = X

F2:type{_(?X:int=99):int} = F1    # F1 and F2 are of the same type
```

Named parameters participate in function overload resolution:

```
Process(Value:int):string = "One parameter"
Process(Value:int, ?Option:string):string = "Two parameters"
Process(Value:int, ?Option1:string, ?Option2:logic):string = "Three parameters"

Process(42)                                          # Calls first overload
Process(42, ?Option := "test")                       # Calls second overload
Process(42, ?Option1 := "test", ?Option2 := true)  # Calls third overload
```

The compiler selects the overload that matches the provided arguments. Named parameters make overload resolution more precise since names must match exactly.

Named parameters have specific rules for *overload distinctness* that differ from positional parameters. Two function signatures are considered **indistinct** (cannot overload) if they could be called with the same arguments.

Order doesn't matter for named parameters: Named parameters are matched by name, not position, so reordering doesn't create distinctness:

```
# Not distinct - same parameters, different order
F(?Y:int, ?X:int):int = X + Y
F(?X:int, ?Y:int):int = X - Y  # ERROR
```

Defaults don't create distinctness: The presence or absence of default values doesn't make signatures distinct if the parameter names are the same:

```
# Same parameter name with/without default
F(?X:int=42):int = X
F(?X:int):int = X  # ERROR
```

The all-defaults rule: If all parameters in both overloads have default values, the signatures are indistinct because both can be called with no arguments:

```
# ERROR Both can be called as F()
# F(?X:int=42):int = X
# F(?Y:int=42):int = Y         # ERROR

# ERROR Both callable with no args
# F(?X:int=42):int = X
# F(?X:float=3.14):float = X  # ERROR
```

Different parameter names are distinct: Functions with different named parameter names can overload:

```
# Valid: Different names
F(?X:int):int = X
F(?Y:int):int = Y  # OK - distinct parameter names
```

Named vs positional parameters are distinct: A named parameter is distinct from a positional parameter, even with the same name and type:

```
# Valid: Named vs positional
F(?X:int):int = X
F(X:int):int = X  # OK
```

At least one required parameter must differ: If the set of required (no default) named parameters differs, the overloads are distinct:

```
# Valid: First requires ?Y, second doesn't
F(?Y:int, ?X:int=42):int = X
F(?X:int):int = X  # OK - different required parameter set
```

Positional parameters create distinctness: Different positional parameter types make signatures distinct, even if named parameters are the same:

```
# Valid: Different positional parameter types
F(Arg:float, ?X:int):int = X
F(Arg:int, ?X:int):int = X  # OK
```

Superset of calls: If one signature can handle all the calls that another can, they're indistinct:

```
# ERROR 3532: First can handle all calls to second
# F(?Y:int=42, ?X:int=42):int = X
# F(?X:int):int = X  # ERROR - can call first as F(?X := 10)
```

8.1.2 Tuple as Arguments

Tuples can be used to provide positional arguments. However, you cannot mix a pre-constructed tuple variable with additional named arguments:

```
Calculate(A:int, B:int, ?C:int = 0):int = A + B + C

# Valid: tuple provides positional arguments
Args:tuple(int, int) = (1, 2)
Calculate(Args)  # Returns 3

# Valid: all arguments provided directly
Calculate(1, 2, ?C := 5)  # Returns 8

# Invalid: cannot mix tuple variable with named arguments
# Calculate(Args, ?C := 5)  # ERROR
```

Functions can destructure tuple parameters directly in the parameter list, allowing you to extract tuple elements inline without manual indexing:

```
# Destructure tuple parameter in place
Func(A:int, (B:int, C:int), D:int):int =
    A + B + C + D

Func(1, (2, 3), 4)        # Direct tuple literal - returns 10
X := (2, 3)
Func(1, X, 4)             # Tuple variable - returns 10
Y := (1, (2, 3), 4)
Func(Y)                   # Entire argument list as tuple - returns 10
```

The parameter `(B:int, C:int)` destructures the tuple, giving direct access to `B` and `C` instead of requiring `Tuple(0)` and `Tuple(1)` indexing.

Tuples can be destructured to arbitrary depth:

```
# Simple nesting
H(A:int, (B:int, (C:int, D:int)), E:int):int =
    A + B + C + D + E

H(1, (2, (3, 4)), 5)              # Returns 15
T := (2, (3, 4))
H(1, T, 5)                        # Returns 15
```

```
T2 := (1, (2, (3, 4)), 5)
H(T2)                           # Returns 15
```

You can mix destructured tuple parameters with regular tuple parameters that aren't destructured:

```
# Destructured form - access elements directly
F(A:int, (B:int, C:int), D:int):int =
    A + B + C + D

# Non-destructured form - use tuple indexing
G(A:int, T:tuple(int, int), D:int):int =
    A + T(0) + T(1) + D

# Both work identically
F(1, (2, 3), 4)  # Returns 10
G(1, (2, 3), 4)  # Returns 10
```

Choose destructured form when you need direct access to individual elements, and non-destructured when you need to pass the tuple as a whole to other functions.

Tuple parameters can contain named/optional parameters, allowing for flexible APIs that combine structural decomposition with optional values:

```
# Named parameter inside nested tuple
SumValues(A:int, (X:int, (Y:int, (?Z:int = 0)))):int =
    A + X + Y + Z

# Can provide Z explicitly
SumValues(1, (2, (3, (?Z := 4))))  # Returns 10

# Can omit Z to use default
SumValues((1, (2, (3))))           # Returns 6
```

A tuple can contain multiple named parameters, and they can be specified in any order:

```
ProcessData(Base:int, (Items:[]int, ?Scale:int = 1, ?Offset:int = 0)):int =
    if (First := Items[0]):
        First * Scale + Offset + Base
    else:
        Base

Data := array{100, 200}

ProcessData(10, Data)                           # Uses defaults: 110
```

```
ProcessData(10, (Data, ?Scale := 2))               # 210
ProcessData(10, (Data, ?Offset := 5))              # 115
ProcessData(10, (Data, ?Scale := 2, ?Offset := 5)) # 215
ProcessData(10, (Data, ?Offset := 5, ?Scale := 2)) # 215 (order doesn't matter)
```

When a tuple parameter contains **only** named parameters (no positional parameters), you must provide an empty tuple `()` even when using all defaults:

```
# Tuple with only named parameters
Configure(Base:int, (?Width:int = 10, ?Height:int = 20)):int =
    Base + Width + Height

# Must provide empty tuple when using all defaults
Configure(5, ())  # Returns 35

# Cannot omit the tuple entirely
# Configure(5)  # ERROR - tuple parameter required
```

This is a known limitation in the current implementation. When the tuple contains at least one positional parameter, this restriction doesn't apply.

Refined types with `where` clauses are not allowed in destructured tuple parameters:

```
# ERROR 3624: Refined types not supported in tuple destructuring
# H(A:int, ((B:int where B > 0), C:int), D:int):int =
#     A + B + C + D
```

This restriction applies to the types within the tuple destructuring. Regular parameter refinements outside tuples work normally.

8.1.3 Flattening and Unflattening

Verse provides automatic conversion between tuples and multiple arguments at function call sites, enabling flexible calling conventions without explicit packing or unpacking.

Flattening: A function expecting multiple parameters can be called with a single tuple. In the following, the tuple `Args` is automatically unpacked into the `Add` function's parameters:

```
Add(X:int, Y:int):int= X + Y
Args:= (3, 5)
Add(Args)       # Returns 8 - tuple automatically flattened
```

Unflattening: A function expecting a single tuple parameter can be called with flattened arguments. The individual arguments of the call to `F` are automatically packed into the tuple parameter:

```
F(P:tuple(int, int)):int = P(0) + P(1)

F(3, 5)  # Returns 8 - args automatically packed into tuple
```

The empty tuple has the same flattening behavior:

```
F(X:tuple()):int = 42

F(())   # Explicit empty tuple
F()     # No arguments - automatically creates empty tuple
```

8.1.4 Evaluation Order

Arguments are evaluated in a specific order to maintain predictable behavior:

1. *Positional arguments*: Left to right in the call
2. *Named arguments*: Left to right as encountered in the call
3. *Default values*: Filled in for omitted parameters, left to right in parameter order

If named arguments appear in a different order than parameters, the compiler uses temporary variables to preserve the evaluation order you specified:

```
Process(A:int, ?B:int, ?C:int, ?D:int):string =
    "{A}, {B}, {C}, {D}"

# Call with reordered named args
Process(1, ?D := 4, ?B := 2, ?C := 3)

# Evaluation order: 1, 4, 2, 3 (as written)
# But passed to function in parameter order: 1, 2, 3, 4
```

This ensures that side effects in argument expressions happen in the order you write them, not in parameter order.

8.2 Extension Methods

Extension methods allow you to add new methods to existing types without modifying their original definitions. This powerful feature enables you to extend any type in Verse—including built-in types like `int`, `string`, arrays, and maps—with custom functionality while maintaining clean separation between different concerns.

Extension methods are particularly valuable when:

- You want to add domain-specific operations to built-in types
- You need to extend types from libraries you don't control
- You're building fluent or builder-style APIs
- You want to organize related functionality separately from type definitions

Extension methods use a special syntax where the extended type appears in parentheses before the method name:

```
# Extend int with a custom method
(Value:int).Double()<computes>:int = Value * 2

# Call the extension method using dot notation
X := 5
Y := X.Double()  # Returns 10

# Can also call on literals
Z := 7.Double()  # Returns 14
```

The type in parentheses can be any Verse type: primitives, tuples, classes, interfaces, arrays, maps, or structs.

Extending primitives:

```
(N:int).IsEven()<decides>:void = N = 0 or Mod[N,2] = 0
(S:string).FirstChar()<decides>:char = S[0]

42.IsEven[]           # Returns true
"Hello".FirstChar[]   # Returns 'H'
```

Extending tuples:

```
# Extend a specific tuple type (Note: Sqrt is <reads>)
(Point:tuple(int, int)).Distance()<reads>:float =
    Sqrt( (Point(0) * Point(0) + Point(1) * Point(1)) * 1.0)

(3, 4).Distance()  # Returns 5.0
```

Extending arrays:

```
(Numbers:[]int).Sum()<transacts>:int =
    var Total:int = 0
    for (N:Numbers):
        set Total += N
    Total

array{1, 2, 3, 4, 5}.Sum()  # Returns 15
```

Extending maps:

```
(M:[int]string).Keys()<computes>:[]int =
    for (Key->X:M):
        Key

map{1=>"a", 2=>"b", 3=>"c"}.Keys()  # Returns array{1, 2, 3}
```

Extending classes:

```
player := class:
    Name:string
    var Score:int
```

```
# Add method to existing class
(P:player).AddScore(Points:int):void =
    set P.Score += Points

Player1 := player{Name := "Alice", Score := 100}
Player1.AddScore(50)  # Score becomes 150
```

Extension methods support all parameter features including named and default parameters:

```
#(Text:string).Pad(?Left:int = 0, ?Right:int = 0):string = ...

"Hello".Pad(?Left:=5)               # "     Hello"
"Hello".Pad(?Right:=5)              # "Hello     "
"Hello".Pad(?Left:= 2, ?Right:=3)   # "  Hello   "
```

8.2.1 Overloading

You can define multiple extension methods with the same name for different types:

```
# Overloaded Extension method for different types
(N:int).Format():string = "int:{N}"
(B:logic).Format():string = if (B?) {"logic:true"} else {"logic:false"}

42.Format()      # Returns "int:42"
true.Format()    # Returns "logic:true"
```

The compiler selects the appropriate overload based on the receiver type.

8.2.2 On the Empty Tuple

The empty tuple `tuple()` represents the unit type and can have extension methods:

```
(Unit:tuple()).GetMagicNumber():int = 42

().GetMagicNumber()  # Returns 42
```

This can be useful for creating namespace-like groupings of functions.

8.2.3 Rules

Must be called: Extension methods cannot be referenced as first-class values without calling them:

```
(N:int).Double():int = N * 2

# Valid: calling the method
X := 5.Double()

# Invalid: referencing without calling
# F := 5.Double  # ERROR
```

Conflicts with Class Methods: Extension methods cannot have the same signature as methods defined directly in classes or interfaces:

```
player := class:
    Health():int = 100

# Invalid: Conflicts with class method
# (P:player).Health():int = 50  # ERROR
```

This prevents ambiguity and ensures that class methods always take precedence.

Scope and Visibility: Extension methods are scoped like regular functions. They're only visible where they're defined or imported:

```
# In module A
utils := module:
    (S:string).Reverse<public>():string =
        # Implementation

# In module B
using { utils }

"Hello".Reverse()  # Available after importing
```

Extension Methods in Class Scope: Extension methods can be defined inside classes and access class members:

```
game_manager := class:
    Multiplier:int = 10

    (Score:int).ScaledScore()<computes>:int =
        Score * Multiplier  # Accesses class field

    ProcessScore(Value:int)<computes>:int =
        Value.ScaledScore()  # Uses extension method

GM := game_manager{}
GM.ProcessScore(5)  # Returns 50
```

This creates a lexical closure where the extension method can reference the enclosing class's members.

Tuple Argument Conversion: When an extension method has multiple parameters, you can pass a tuple to provide all arguments at once:

```
point := class:
    X:int
    Y:int

(P:point).Translate(DX:int, DY:int)<allocates>:point =
    point{X := P.X + DX, Y := P.Y + DY}

Origin := point{X := 0, Y := 0}
Delta := (5, 10)
NewPoint := Origin.Translate(Delta)  # Tuple expands to two arguments
```

This works when the tuple type matches the parameter list.

8.3 Lambdas

Note: Lambda expressions with the `=>` operator are not yet supported in the current version of Verse. For creating function values and closures, use nested functions instead (see the Nested Functions section below).

Functions are first-class values; they can be stored in variables, passed as parameters, and returned from other functions. This enables powerful functional programming patterns including higher-order functions, callbacks, and composable operations. Currently, these capabilities are provided through nested functions rather than lambda expressions.

8.3.1 Types, Variance and Effects

Function types follow specific subtyping rules based on *variance*:

- *Parameters are contravariant*: A function accepting more general types can substitute for one accepting specific types.
- *Returns are covariant*: A function returning more specific types can substitute for one returning general types.

```
animal := class:
    Name:string

dog := class(animal):
    Breed:string
```

```
# Functions with different parameter/return types
F1(X:animal):dog = dog{Name := X.Name, Breed := "Unknown"}
F2(X:dog):animal = X     # Returns supertype
F3(X:dog):dog = X

# Function type accepting dog, returning animal
var ProcessDog:type{_(:dog):animal} = F1  ##############TODO

# Valid: F1 accepts animal (more general), returns dog (more specific)
set ProcessDog = F1  # OK: tuple(animal)->dog <: tuple(dog)->animal

# Valid: F3 accepts dog, returns dog (more specific than animal)
set ProcessDog = F3  # OK: tuple(dog)->dog <: tuple(dog)->animal

# Invalid: F2 returns animal but parameter is not contravariant enough
# ProcessDog = F2  # ERROR: tuple(dog)->animal </: tuple(dog)->animal
#                  # (same parameters, same return - no variance issue here)
```

Effects are part of the function type. A function with fewer effects can be used where a function with more effects is expected - effects are **covariant** (fewer effects = subtype):

```
Pure():int = 42
Transactional()<transacts>:int = 42
Suspendable()<suspends>:int = 42

# Functions expecting specific effects
UsePure(F():int):int = F()
UseTransactional(F()<transacts>:int):int = F()
UseSuspendable(F()<suspends>:int):task(int) = spawn{ F() }
```

```
9
10 UsePure(Pure)                    # OK
11 UseTransactional(Transactional)  # OK
12 UseSuspendable(Suspendable)      # OK
13
14 # Covariance: fewer effects can substitute for more effects
15 UseTransactional(Pure)           # OK: ():int <: ()<transacts>:int
16
17 # Invalid: more effects cannot substitute for fewer
18 # UsePure(Transactional)         # ERROR: ()<transacts>:int </: ():int
```

A `<computes>` function can be passed where `<transacts>` is expected because fewer effects means the function is more constrained.

When you assign different functions conditionally, Verse finds the least upper bound (join) of their types:

```
1  # Assume the following:
2  # base := class{Value:int}
3  # derived := class(base){Extra:string}
4
5  F1():base = base{Value:=1}
6  F2():derived = derived{Value:=2, Extra:="test"}
7
8  # Join: ()->base (common supertype)
9  G := if(true?) {F1} else {F2}
10 G().Value  # Can access base members
```

8.3.2 Using `type{}`

The `type{_(...):...}` syntax declares function types with full detail. This is the mechanism for creating function type signatures that include parameter types, return types, and effects. Underscore `_` is a placeholder for the function name, emphasizing that it describes a signature, not a specific function:

```
1  # Function type variable
2  var Handler:?type{_(:string, :int)<decides>:void} = false
3
4  # Nested function matching the signature
5  MakeHandler(Name:string, Count:int)<decides>:void =
6      Print("{Name}: {Count}")
7      Count > 0  # Decides effect
8
9  set Handler = option{MakeHandler}
10
```

```
# Function accepting function parameter
Process(F:type{_(:int):int}, Value:int):int =
    F(Value)

# Nested function to pass
Double(X:int):int = X * 2
Process(Double, 5)  # Returns 10
```

The `type{}` construct *exclusively declares function type signatures.* It cannot be used for general type expressions or to extract types from values:

```
ValidType1 := type{_():int}
ValidType2 := type{_(:string, :int):float}
ValidType3 := type{_()<transacts><decides>:void}
```

Within `type{}`, function declarations must have return types but *cannot have bodies.*

Function types work as field types in classes:

```
# Assume:
# calculator := class:
#    Operation:type{_(:int,:int):int}

Add(X:int, Y:int):int = X + Y
Multiply(X:int, Y:int):int = X * Y

# Create instances with different operations
Adder := calculator{Operation := Add}
Multiplier := calculator{Operation := Multiply}

Adder.Operation(5, 3)      # Returns 8
Multiplier.Operation(5, 3) # Returns 15
```

Function types can be used for local variables, enabling conditional function selection:

```
ProcessA():int = 10
ProcessB():int = 20

SelectFunction(UseA:logic):int =
    # Choose function based on condition
    Fn:type{_():int} =
        if (UseA?):
            ProcessA
        else:
```

```
            ProcessB
    Fn()

SelectFunction(true)   # Returns 10
SelectFunction(false)  # Returns 20
```

Combine `type{}` with `?` to create optional function types:

```
DefaultHandler()<computes>:int = -1
CustomHandler()<computes>:int = 42

Process(Handler:?type{_()<computes>:int})<computes><decides>:int =
    # Use handler if provided, otherwise use default
    Handler?() or DefaultHandler()

Process[false]                  # Returns -1 (no handler)
Process[option{CustomHandler}]  # Returns 42 (custom handler)
```

Create arrays of functions sharing the same signature:

```
GetZero():int = 0
GetOne():int = 1
GetTwo():int = 2

SumFunctions(Functions:[]type{_():int}):int =
    var Result:int = 0
    for (Fn : Functions):
        set Result += Fn()
    Result

SumFunctions(array{GetZero, GetOne, GetTwo})  # Returns 3
```

8.3.3 Examples

Map-Filter-Reduce:

```
# Generic map
Map(Items:[]t, F(:t)<transacts>:u where t:type, u:type)<transacts>:[]u =
    for (Item:Items):
        F(Item)

# Generic filter
Filter(Items:[]t, Pred(:t)<computes><decides>:void where t:type)<computes>:[]t =
    for (Item:Items, Pred[Item]):
        Item
```

```

# Generic fold/reduce
Fold(Items:[]t, Initial:u, F(:u, :t)<transacts>:u where t:type, u:type)<transacts>:u =
    var Acc:u = Initial
    for (Item:Items):
        set Acc = F(Acc, Item)
    Acc

# Usage with nested functions
Numbers := array{1, 2, 3, 4, 5}

# Define nested functions for operations
Square(X:int)<computes>:int = X * X
IsEven(X:int)<computes><decides>:void = X = 0 or Mod[X,2] = 0
AddTo(Acc:int, X:int)<computes>:int = Acc + X

Squared := Map(Numbers, Square)
Evens := Filter(Numbers, IsEven)
Sum := Fold(Numbers, 0, AddTo)
```

Function composition:

```
Compose(F(:b):c, G(:a):b where a:type, b:type, c:type):type{_(:a):c} =
    # Return a nested function that composes F and G
    Composed(X:a):c = F(G(X))
    Composed

Add1(X:int):int = X + 1
Double(X:int):int = X * 2

# Compose: first doubles, then adds 1
DoubleThenIncrement := Compose(Add1, Double)
DoubleThenIncrement(5)  # Returns 11 (5*2 + 1)
```

Partial application:

```
Partial(F(:a, :b):c, X:a where a:type, b:type, c:type):type{_(:b):c} =
    # Return a nested function with X captured
    PartialFunc(Y:b):c = F(X, Y)
    PartialFunc

Add(X:int, Y:int):int = X + Y
Add5 := Partial(Add, 5)
Add5(3)  # Returns 8
```

8.4 Nested Functions

Unreleased Feature

Nested functions have not yet been released. This section documents planned functionality that is not currently available.

Nested functions (also called local functions) are functions defined inside other functions. They provide encapsulation, enable closures over local variables, and help organize complex logic within a function's scope. Nested functions have names, can be recursive, and are the primary way to create function values and closures in Verse.

A nested function is declared just like a top-level function, but inside another function's body:

```
Outer(X:int):int =
    # Nested function definition
    Inner(Y:int):int = Y * 2

    # Call nested function
    Inner(X)

Outer(5)  # Returns 10
```

Nested functions are only visible within their enclosing function's scope. They cannot be accessed from outside.

Nested functions capture (close over) variables from any enclosing scope, creating powerful closures:

```
MakeGreeter(Name:string):type{_():string} =
    # Greeting captures Name from outer scope
    Greeting():string = "Hello, {Name}!"

    # Return the nested function
    Greeting

SayHello := MakeGreeter("Alice")
SayHello()  # Returns "Hello, Alice!"

SayHi := MakeGreeter("Bob")
SayHi()  # Returns "Hello, Bob!"
```

Each call to `MakeGreeter` creates a new closure with its own captured `Name` value.

Nested functions support overloading by parameter types:

```
Process(X:int):string =
    # Overloaded nested functions
    Format(Value:int):string = "int: {Value}"
    Format(Value:float):string = "float: {Value}"

    # Calls appropriate overload
    IntResult := Format(42)       # Calls int version
    FloatResult := Format(3.14)   # Calls float version

    "{IntResult}, {FloatResult}"

Process(1)  # Returns "Int: 42, Float: 3.14"
```

Overload resolution works the same as for top-level functions.

8.4.1 Closures with State

Nested functions can capture `var` variables and mutate them, creating stateful closures:

```
MakeCounter(Initial:int):tuple(type{_():int}, type{_():void}) =
    var Count:int = Initial

    # Getter captures Count
    GetCount():int = Count

    # Incrementer mutates captured Count
    Increment():void = set Count = Count + 1

    (GetCount, Increment)

Counter := MakeCounter(0)
GetValue := Counter(0)
IncrementValue := Counter(1)

GetValue()        # Returns 0
IncrementValue()  # Increments count
GetValue()        # Returns 1
IncrementValue()  # Increments count
GetValue()        # Returns 2
```

This pattern creates a closure that maintains private mutable state.

8.4.2 Restrictions

Nested functions have several important restrictions that distinguish them from top-level functions:

- Nested functions **cannot** have access specifiers like `<public>`, `<internal>`, or `<private>`:
- Nested functions are always private to their enclosing function.
- You cannot define classes inside functions (nested or otherwise):

```
# ERROR: Cannot define classes in local scope
F():void =
    my_class := class {}  # ERROR

# Correct: Define classes at module level
my_class := class {}

F():void =
    Instance := my_class{}  # OK - can use class
```

- Nested functions cannot reference variables or other nested functions defined later in the same scope (this also means mutually recursive nested functions are not allowed):

```
# ERROR 3506: G used before defined
F():void =
    X := G()     # ERROR: G not yet defined
    G():int = 42

# Correct: Define before use
F():void =
    G():int = 42
    X := G()     # OK: G is defined
```

- The `(super:)` syntax for calling parent class methods **cannot** be used in nested functions:

```
# ERROR 3612: super not allowed in nested function
base_class := class:
    F(X:int):int = X

derived_class := class(base_class):
    F<override>(X:int):int =
        G():int =
```

```
            (super:)F(X)  # ERROR: super not allowed here
        G()

# Correct: Use super directly in the overriding method
derived_class := class(base_class):
    F<override>(X:int):int =
        BaseResult := (super:)F(X)  # OK
        G():int = BaseResult * 2
        G()
```

8.5 Parametric Functions

Parametric functions (also called generic functions) allow you to write code that works with multiple types while maintaining complete type safety. Rather than writing separate functions for each type, you define a single function with type parameters that adapt to whatever types you use them with.

A parametric function declares type parameters using a `where` clause that specifies constraints on those types:

```
# Simple identity function - works with any type
Identity(X:t where t:type):t = X
# Usage - type parameter inferred automatically
Identity(42)        # t inferred as int, returns 42
Identity("hello")   # t inferred as string, returns "hello"
```

The `where t:type` clause declares `t` as a type parameter with the constraint `type`, meaning it can be any Verse type. The function signature `(X:t):t` means "takes a value of type `t` and returns a value of that same type `t`."

```
FunctionName(Parameters where TypeParameter:Constraint, ...):ReturnType = Body
```

- *Type parameters* appear in the `where` clause
- *Constraints* specify requirements (e.g., `type`, `subtype(comparable)`)
- *Multiple type parameters* are comma-separated in the `where` clause

Verse automatically infers type parameters from the arguments you pass, eliminating the need for explicit type annotations in most cases:

```
# Function with two type parameters
Pair(X:t, Y:u where t:type, u:type):tuple(t, u) = (X, Y)

# All type parameters inferred
Pair(1, "one")        # t = int, u = string, returns (1, "one")
Pair(true, 3.14)      # t = logic, u = float, returns (true, 3.14)
```

Inference with collections:

```
# Generic first element function
First(Items:[]t where t:type)<decides>:t = Items[0]

Numbers := array{1, 2, 3}
Result :int= First[Numbers]  # t inferred as int from []int
```

When you pass multiple values to a parametric function expecting a single type parameter, Verse can infer either a tuple or an array:

```
# Returns the argument unchanged
Identity(X:t where t:type):t = X

# Passing multiple values creates a tuple
Result1:tuple(int, int) = Identity(1, 2)  # t = tuple(int, int)

# Can also be treated as an array
Result2:[]int = Identity(1, 2)  # t = []int via conversion
```

8.5.1 Type Constraints

Type constraints restrict which types can be used with type parameters, enabling operations that require specific capabilities.

The most permissive constraint accepts any type:

```
# Works with absolutely any type
Store(Value:t where t:type):t = Value
```

Restricts to types that are subtypes of a specified type:

```
vehicle := class:
    Speed:float = 0.0

car := class(vehicle):
    NumDoors:int = 4

# Only accepts vehicle or its subtypes
ProcessVehicle(V:t where t:subtype(vehicle)):t =
    # Can access Speed because we know V is a vehicle
    Print("Speed: {V.Speed}")
    V
```

```
# Valid calls
ProcessVehicle(vehicle{})      # t = vehicle
ProcessVehicle(car{})          # t = car (subtype of vehicle)
```

The function returns type `t`, not the base type. This preserves the specific type:

```
MyCar := car{NumDoors:=4, Speed:=60.0}
Result:car= ProcessVehicle(MyCar)  # Result has type car, not vehicle
Result.NumDoors                  # Can access car-specific fields
```

The `subtype(comparable)` constraint enables equality comparisons:

```
# Can use = and <> operators on t
FindInArray(Items:[]t, Target:t where t:subtype(comparable))<decides>:[]int =
    for (Index -> Item : Items, Item = Target):
        Index
```

Type parameters can reference each other in constraints:

```
# u must be a subtype of t
Convert(Base:t, Derived:u where t:type, u:subtype(t)):t = Base
# This ensures type safety across related types
```

8.5.2 Member Access

When using subtype constraints, you can access members that exist on the base type:

```
entity := class:
    Name:string = "Entity"
    Health:int = 100

player := class(entity):
    Score:int = 0
```

```
# Can access entity members through type parameter
GetInfo(E:t where t:subtype(entity)):tuple(t, string, int) =
    (E, E.Name, E.Health)            # Can access Name and Health

P := player{Name := "Alice", Health := 100, Score := 1500}
Info := GetInfo(P)                   # Returns (player instance, "Alice", 100)
                                     # Info(0) has type player, not entity
```

Method calls work too:

```
entity := class:
    GetStatus():string = "Active"

```

```
# Call methods on parametrically-typed values
CheckStatus(E:t where t:subtype(entity)):string =
    E.GetStatus()  # Method call through type parameter
```

8.5.3 Polarity and Variance

Type parameters must be used consistently according to variance rules. This ensures type safety when functions are used as values or passed as arguments.

Covariant positions (safe for return types):

- Function return types
- Tuple/array element types (as return)
- Map value types (as return)

Contravariant positions (safe for parameter types):

- Function parameter types
- Map key types

The polarity check: Verse validates that type parameters appear only in positions compatible with their intended use:

```
# Valid: t appears covariantly (return type)
GetValue(X:t where t:type):t = X

# Valid: t appears contravariantly (parameter)
Consume(X:t where t:type):void = {}

# Valid: t appears in both positions (through function parameter and return)
Apply(F:type{_(:t):t}, X:t where t:type):t = F(X)
```

Invariant types cause errors:

```
# ERROR 3552: Cannot return type that's invariant in t
# c(t:type) := class{var X:t}  # Mutable field makes c invariant in t
# MakeContainer(X:t where t:type):c(t) = c(t){X := X}
```

The error occurs because `c(t)` contains a mutable field of type `t`, making it invariant - neither covariant nor contravariant. Returning such a type from a parametric function is unsafe.

Map polarity: Maps are contravariant in keys and covariant in values:

```
# Valid: contravariant key, covariant value
ProcessMap(M:[t]u where t:subtype(comparable), u:type):[t]u = M
```

8.6 Overloading

Function overloading allows you to define multiple functions with the same name but different parameter types. The compiler selects the correct version based on the types of the arguments provided at the call site.

Define multiple functions with the same name but different parameter types:

```
# Overload by parameter type
Process(Value:int):string = "Integer: {Value}"
Process(Value:float):string = "Float: {Value}"
Process(Value:string):string = "String: {Value}"

# Calls select the appropriate overload
Process(42)        # Returns "Integer: 42"
Process(3.14)      # Returns "Float: 3.14"
Process("hello")   # Returns "String: hello"
```

The compiler determines which overload to call based on the argument types. Each overload must have a distinct parameter type signature.

8.6.1 Capture

You cannot take a reference to an overloaded function name:

```
# ERROR 3502: Cannot capture overloaded function
f(x:int):void = {}
f(x:float):void = {}

# Error: which f?
# g:void = f
```

This restriction exists because the compiler cannot determine which overload you mean without seeing the call site with arguments.

8.6.2 Effects

You can overload functions with different effects, but only if the parameter types are also different:

Valid: Different types, different effects:

```
Process(x:float):float = x
Process(x:int)<transacts><decides>:int = x = 1

Process(3.0)   # Returns 3.0 (non-failable)
Process[1]     # Returns option{1} (failable)
```

Invalid: Same types, different effects:

```
# ERROR 3532: Same parameter type
f(x:int):void = {}
f(x:int)<transacts><decides>:void = {}  # ERROR
```

Effects alone don't create distinctness - you need different parameter types.

8.6.3 Overloads in Subclasses

Subclasses can add new overloads to methods:

```
C0 := class:
    f(X:int):int = X

C1 := class(C0):
    # Add new overload for float
    f(X:float):float = X
```

```
C0{}.f(5)     # OK - int overload
C1{}.f(5)     # OK - inherited int overload
C1{}.f(5.0)   # OK - new float overload
```

When a subclass defines a method that shares a name with a parent method, it must either:

1. Provide a **distinct parameter type** (different from all parent overloads)
2. **Override exactly one** parent overload using `<override>`

```
C := class{}
D := class(C){}

# Parent class with overloads
E := class:
    f(c:C):C = c
    f(e:E):E = e

# Valid: Overrides one parent overload
F := class(E):
    f<override>(c:C):D = D{}

# ERROR: D is subtype of C, overlaps but doesn't override
# G := class(E):
#     f(d:D):D = d  # ERROR - ambiguous with f(c:C)
```

8.6.4 Interfaces with Overloaded Methods

Interfaces can declare overloaded methods:

```
formatter := interface:
    Format(X:int):string = "{X}"
    Format(X:float):string = "{X}"

entity := class(formatter):
    Format<override>(X:int):string = "Entity-{X}"
    Format<override>(X:float):string = "Entity-{X}"
```

8.6.5 Restrictions

Cannot use `var` with overloaded functions:

Function-valued variables cannot be overloaded:

```
# ERROR 3502: Cannot have var overloaded functions
# var f():void = {}
# var f(x:int):void = {}

# ERROR: Cannot mix var and regular
# var f():void = {}
# f(x:int):void = {}
```

Cannot overload functions with non-functions:

A name cannot be both a function and a non-function value:

```
# ERROR: Cannot overload with variable
# f:int = 0
# f():void = {}
```

Cannot overload classes:

Class names cannot be overloaded:

```
# ERROR 3588, 3532: Cannot overload class name
# C := class{}
# C(x:int):C = C{}
```

Bottom type cannot resolve overloads:

The bottom type (from `return` without a value) cannot be used for overload resolution:

```
# ERROR 3518: Cannot determine which overload
F(X:int):int = X
F(X:float):float = X
```

```
4
5  # G():void =
6  #     F(@ignore_unreachable return)  # ERROR - which F?
7  #     0
```

8.6.6 Overloading with <suspends>

You can mix suspending and non-suspending overloads if the parameter types differ:

```
1  f(x:int)<suspends>:void =
2      Sleep(1.0)
3
4  f(x:float):void =
5      Print("Non-suspending")
6
7  # Call non-suspending directly
8  f(1.0)
9
10 # Call suspending with spawn
11 spawn{f(1)}
```

Cannot call suspending overload without spawn:

```
1  # ERROR 3512: suspends version needs spawn context
2  f(x:int):void = {}
3  f(x:float)<suspends>:void = {}
4
5  # g():void = f(1.0)  # ERROR - float version is suspends
```

Cannot spawn non-suspending overload:

```
1  # ERROR 3538: Cannot spawn non-suspends function
2  f(x:int):void = {}
3  f(x:float)<suspends>:void = {}
4
5  # g():void = spawn{f(1)}  # ERROR - int version not suspends
```

8.6.7 Types

Every function has a type that captures its parameters, effects, and return value. The type syntax uses an underscore as a placeholder for the function name:

```
1  type{_(:int,:string)<decides>:float}
```

This represents any function that takes an integer and a string, might fail, and returns a float when successful.

Multiple functions may share a name through overloading, as long as their signatures do not create ambiguity. The compiler can distinguish between overloads based on the argument types:

```
Transform(X:int):string = "I:{X}"
Transform(X:float):string = "F:{X}"
Transform(X:string):string = "S:{X}"

Result1 := Transform(42)        # Calls int version
Result2 := Transform(3.14)      # Calls float version
Result3 := Transform("Hello")   # Calls string version
```

However, overloading has strict limitations based on **type distinctness**. Two types are considered "distinct" for overload purposes only if there is no possible value that could match both types. This restriction prevents ambiguity and ensures that function calls can always be resolved unambiguously at compile time.

Verse uses precise rules to determine whether two parameter types are distinct enough to allow overloading. Understanding these rules is critical for designing clear APIs.

The following type pairs are **not distinct** and cannot be used to overload functions:

1. Optional and Logic. `?t` and `logic` are not distinct because `logic` (true or false) is internally equivalent to `?t` (value or false):

```
# ERROR: Not distinct
F(:?any):void = {}
F(:logic):void = {}
```

2. Arrays and Maps. Arrays `[]t` and maps `[k]t` are not distinct:

```
# ERROR: Not distinct
F(:[]int):void = {}
F(:[string]int):void = {}
```

3. Functions and Maps. Function types and maps are not distinct:

```
# ERROR: Not distinct
F(:[string]int):void = {}
F(G(:string)<transacts><decides>:int):void = {}
```

4. Functions and Arrays. Function types and arrays are not distinct because an overloaded function could match both:

```
# ERROR: Not distinct
F(:[]int):void = {}
F(G(:string)<transacts><decides>:int):void = {}
```

5. Interfaces and Classes. An interface and any class are never distinct, even if the class doesn't implement the interface, because a subtype of the class might:

```
i := interface{}
t := class{}

# ERROR: Not distinct (subtype of t might implement i)
f(:i):void = {}
f(:t):void = {}
```

6. Functions with Different Effects. Functions are not distinct based on effects alone. Changing or removing effects doesn't create a distinct overload:

```
a := class{}
b := class{}

# ERROR: Not distinct
F(G(:a)<transacts><decides>:b):void = {}
F(G(:a):b):void = {}
```

7. Functions with Different Signatures. Functions with different parameter or return types are not distinct because of function overloading:

```
# ERROR: Not distinct
F(G(:b):b):void = {}
F(G(:a):b):void = {}
```

8. void as Top Type. `void` is treated as equivalent to the top type (accepts `any`), so it's not distinct from any other type:

```
# ERROR: Not distinct
F(:int):void = {}
F(:void):void = {}
```

9. Subtype Relationships. Classes with subtype relationships are not distinct:

```
a := class{}
b := class(a){}

# ERROR: Not distinct
F(:a):void = {}
F(:b):void = {}
```

10. Tuple Distinctness Rules. Tuples have complex distinctness rules:

Empty tuples and arrays are not distinct:

```
a := class{}

# ERROR: Not distinct
F(:tuple(), :a):void = {}
F(:[]a, :a):void = {}
```

Tuples and arrays are distinct only if tuple element types are completely distinct:

```
a := class{}
b := class(a){}

# ERROR: Not distinct (b is subtype of a)
F(:tuple(a, b), :a):void = {}
F(:[]a, :a):void = {}
```

Tuples and maps with `int` key are not distinct:

```
a := class{}

# ERROR: Not distinct
F(:tuple(a), :a):void = {}
F(:[int]a, :a):void = {}
```

Tuples and maps with non-`int` key ARE distinct:

```
a := class{}

# Valid: Distinct types
F(:tuple(a), :a):void = {}
F(:[logic]a, :a):void = {}  # OK
```

Singleton tuples and optional for `int` are not distinct:

```
a := class{}

# ERROR: Not distinct
F(:tuple(int), :a):void = {}
F(:?int, :a):void = {}
```

Singleton tuples and optional for non-`int` ARE distinct:

```
# Valid: Distinct types
F(:tuple(a), :a):void = {}
F(:?a, :a):void = {}  # OK
```

8.7 Publishing Functions

Publishing a function is a promise of backwards compatibility between the function and its clients. Consider this function:

```
F1<public>(X:int):int = X + 1
```

The type annotation (`X:int):int`) tells us that this function promises that given any integer it will always return an integer. That contract cannot be broken in future versions of the code. Because it has the default effect, which includes the `<reads>` effect, the implementation could change in the future, perhaps to perform additional operations or optimizations, as long as it maintains its signature.

Functions that do not have the `<reads>` effect are less flexible. Consider this function:

```
F2<public>(X:int)<computes>:int = X + 1
```

Because it has the `<computes>` effect specifier, it does not have the `<reads>` effect. Without the `<reads>` effect, this function promises to always return the same result for some given parameters. Changing it to return, for example, `X + 2` would break that promise, and so must be rejected by the compiler as backward incompatible.

Functions such as `F1` and `F2` are sometimes called *opaque* as the return type abstracts the function's body. Future version of Verse will support *transparent* functions:

```
F2<public>(X:int) := X + 1
```

A transparent function does not declare its return type, instead the function's type is inferred from its body. This implies a very different promise: a forever guarantee that the function's body will remain exactly the same throughout the lifetime of the module code.

Chapter 9

Chapter 8: Control Flow

Every program has a natural rhythm to its execution, a sequence in which instructions are processed and decisions are made. In Verse, this flow is more than just a mechanical progression through lines of code - it's a carefully orchestrated dance between different types of expressions, each contributing to the overall behavior of your program.

9.1 Blocks

A code block is a fundamental organizational unit, it groups related expressions together and creates a new scope for variables and constants. Unlike many languages where blocks are merely syntactic conveniences, blocks are expressions themselves, meaning they produce values just like any other expression.

The concept of scope is crucial to understanding code blocks. When you create a variable or constant within a block, it exists only within that block's context. This containment ensures that your code remains organized and that names don't accidentally conflict across different parts of your program. Consider this function, it's body is a code block that contains one if-then-else expression, itself composed of three different code blocks.

```
CalculateReward(PlayerLevel:int)<reads>:int =
    if:
        PlayerLevel > 10
        Multiplier := 2.0  # Only exists within this if block
        Base := 100
        Result := Floor[(Base+PlayerLevel) * Multiplier] # Fails on infinity
    then:
        Result  # This block extends the scope of the if
    else:
```

```
        50      # Different branch, different scope
                # Multiplier and Result don't exist here
```

Verse has a flexible syntax with three equivalent formats for writing blocks. The spaced format is the most common, using a colon to introduce the block and indentation to show structure:

```
if (IsPlayerReady[]):
    StartMatch()
    BeginCountdown()
```

The multi-line braced format offers familiarity for programmers coming from C-style languages:

```
if (IsPlayerReady[]) {
    StartMatch()
    BeginCountdown()
}
```

For simple operations, the single-line dot format keeps code concise:

```
HasPowerup()<computes><decides>:void={}
ApplyBoost():void={}
IncrementCounter():void={}
F():void=
    if (HasPowerup[]). ApplyBoost(); IncrementCounter()
```

Since everything is an expression, blocks themselves have values. The value of a block is given by the last expression executed within it. This enables elegant patterns where complex computations can be encapsulated in blocks that seamlessly integrate with surrounding code:

```
FinalScore := block:              # The variable has the block's value
    Base := CalculateScore()
    Bonus := CalculateBonus(CompletionTime)
    Accuracy := Floor[AccuracyValue * 100.0]
    Base + Bonus + Accuracy       # This becomes the block's value
```

9.2 If Expressions

The `if` expression uses success and failure to drive decisions (see Failure for details). When an expression in the condition succeeds, the corresponding branch executes:

```
HandlePlayerAction(Player:player, Action:string):void =
    if (Action = "jump", Player.CanJump[]):
        Player.Jump()
```

```
        PlayJumpSound()
    else if (Action = "attack", Weapon := Player.GetEquippedWeapon[]):
        Weapon.Fire()
        ConsumeAmmo()
    else:
        # Default action
        Player.Idle()
```

This approach allows you to chain conditions that might fail without explicit error handling at each step.

An alternative syntax uses `then:` and `else:` keywords to explicitly label branches:

```
ProcessValue(Value:int):string =
    if:
        Value > 0
        Value < 100
    then:
        "Valid"
    else:
        "Out of range"

ProcessValue(50) = "Valid"
```

This syntax can improve readability when you have multiple conditions or want to emphasize the condition-action separation.

The condition in an `if` must contain at least one expression that can fail. This requirement ensures `if` is used for its intended purpose—handling uncertain outcomes:

```
# Error: condition cannot fail
if (1 + 1):  # Compile error - no fallible expression
    DoSomething()

# Valid: array access can fail
if (FirstItem := Items[0]):
    Process(FirstItem)
```

Empty conditions are also not allowed—every `if` must test something.

If any expression in the condition fails, control flow proceeds to the `else` branch if present. Any effects performed while evaluating the condition are automatically rolled back (see Failure for details):

```
var Counter:int = 0

```

```
if:
    set Counter = Counter + 1  # Provisional change
    Score := GetPlayerScore[]  # Might fail
    Score > 100
then:
    # Counter was incremented
else:
    # Counter rolled back to original value - increment undone!
```

This speculative execution makes conditional logic safer—you can perform operations optimistically, knowing they'll be reversed if subsequent conditions fail.

Variables defined in the condition are available in the `then` branch but not in the `else` branch:

```
if:
    Player := FindPlayer[Name]  # Define Player
then:
    AwardBonus(Player)  # OK - Player available
else:
    Penalize(Player)  # Compile error
```

This scoping reflects the logical flow: in the `else` branch, the condition failed, so any variables bound during the condition might not have meaningful values.

Since `if` is an expression, it produces a value. When all branches return compatible types, the `if` can be used anywhere a value is expected:

```
Damage := if (IsCritical?):
    BaseDamage * 2
else:
    BaseDamage

# Ternary-style
Status := if (Health > 50). "Healthy" else. "Wounded"
```

When branches have incompatible types, the result is widened to `any`:

```
# Different types in branches yields any
Result:any = if (UseNumber?) then 42 else "text"
```

All branches must produce a value for the `if` to be used as an expression.

9.3 Case Expressions

When you need to make decisions based on multiple possible values, the `case` expression provides clear, readable branching:

```
GetWeaponDamage(WeaponType:string):float =
    case(WeaponType):
        "sword"  => 50.0
        "bow"    => 35.0
        "staff"  => 40.0
        "dagger" => 25.0
        _        => 10.0  # Default damage for unknown weapons

GetWeaponDamage("sword") = 50.0
```

The `case` expression is used when you have discrete values to match against, making your intent clearer than a series of `if-else` conditions.

Case expressions work with specific types that support direct value comparison:

- **Primitives**: `int`, `logic`, `char`
- **Strings**: `string`
- **Enums**: Both open and closed enums
- **Refinement types**: Custom types with constraints

They do not work on `float`, objects and tuples because these types either don't have well-defined equality (`float` with `NaN`), lack value semantics (classes are references), or have structural complexity (tuples).

Exhaustiveness Checking with Enums. `case` with `enum` are checked for exhaustiveness. For closed enums where all values are known, the compiler verifies you've handled all cases:

```
# Exhaustive - no wildcard needed
GetVector(Dir:direction):tuple(int, int) =
    case (Dir):
        direction.North => (0, 1)
        direction.South => (0, -1)
        direction.East => (1, 0)
        direction.West => (-1, 0)

GetVector(direction.North) = (0, 1)
```

If you add a wildcard when all cases are covered, you'll get a warning that the wildcard is unreachable:

```
    case (Dir):
        direction.North => (0, 1)
        direction.South => (0, -1)
        direction.East => (1, 0)
        direction.West => (-1, 0)
        _ => (0, 0)  # Warning: all cases already covered
```

Incomplete case coverage is allowed in a `<decides>` context:

```
# Without wildcard in <decides> context - OK
GetPrimaryDirection2(Dir:direction)<decides>:string =
    case (Dir):
        direction.North => "Primary"
        # Other directions cause function to fail
```

Open enums can have values added after publication, so they can never be exhaustive. They always require either a wildcard or a `<decides>` context.

9.4 Loop Expressions

The `loop` expression creates an infinite loop that continues until explicitly broken:

```
GameLoop():void =
    loop:
        UpdatePlayerPositions()
        CheckCollisions()
        RenderFrame()
        if (GameOver[]). break
```

The `break` expression exits the loop entirely, terminating iteration. `break` has "bottom" type—a type that represents a computation that never returns normally. Since the bottom type is a subtype of all other types, `break` can be used in any type context:

```
var X:int = 0
loop:
    set X = if(ShouldExit[]) then break else ComputeValue()
    # break is compatible with int type because bottom  int
```

This allows `break` to be used flexibly in expressions where a value is expected, since the compiler knows that path never produces a value.

When `break` appears in nested loops, it exits only the innermost enclosing loop:

```
var Outer:int = 0
var Inner:int = 0
```

```
loop:
    set Outer += 1
    loop:
        set Inner += 1
        if (Inner = 5):
            break        # Exits inner loop
    if (Outer = 10):
        break            # Exits outer loop
```

The following restrictions apply. The `break` statement must appear in a code block, not as part of a complex expression. A loop must contain at least one non-break statement. Finally, using `break` outside a `loop` produces an error:

```
ProcessData():void =
   if (ShouldStop[]):
           break      # Error
```

9.5 For Expressions

The `for` expression iterates over collections, ranges, and other iterable types, providing a more structured approach to repetition:

```
CalculateTotalScore(Players:[]player)<transacts>:int =
    var Total:int = 0
    for (Player : Players):
        PlayerScore := GetScore(Player)
        set Total += PlayerScore
    Total
```

While it may look familiar from earlier imperative languages, `for` is best thought of as a functional construct that combines iteration, filtering with speculative execution, and construction of a collection of results.

```
Values:[]float= array{1.0, 10.1, 100.2}
Result :=
   for:
      V : Values
      V >= 10.0
      R := Floor[V]
   do:
      R*2.0

Result = array{20.0, 200.0}
```

The above is written with an alternative multi-clause syntax using the `do:` keyword to separate the iteration specification from the body. The `for` iterates over the `Values` array, discarding values smaller than 10 and rounding down numbers. It returns an array of floats. The `Floor` function is defined as `decides` –if it were to fail that iterate would be discarded.

There is another alternative syntax: the single-line dot syntax for simple operations:

```
# Single-line dot style
for (V : Values). DoSomething(V)
```

Index and Value Pairs:

When iterating arrays, you can access both the index and the value using the pair syntax `Index -> Value`:

```
PrintRoster(Players:[]player):void =
    for (Index -> Player : Players):
        Print("Player {Index}: {Player.Name}")
```

The index is zero-based, matching Verse's array indexing convention.

Defining Variables in For Clauses:

The for loop allows you to define intermediate variables that can be used in subsequent filters or the loop body:

```
# Define Y based on X
Doubled := for (X := 1..5, Y := X * 2):
    Y  # Returns array{2, 4, 6, 8, 10}

# Combine with filtering
SafeDivision := for (X := -3..3, X <> 0, Y := Floor[10.0 / (X*1.0)]):
    Y  # Skips X=0, returns array{-3, -5, -10, 10, 5, 3}
```

These intermediate variables are scoped to the iteration and can reference earlier variables in the same clause.

Multiple Filters:

You can chain multiple filter conditions using comma-separated expressions. Each filter must be failable, and if any fails, that iteration is skipped:

```
# Multiple independent filters
Filtered := for (X := 1..10, X <> 3, X <> 7):
    X  # Returns array{1, 2, 4, 5, 6, 8, 9, 10}

# Filters with intermediate variables
```

```
Complex := for (X := 1..5, X <> 2, Y := X * 2, Y < 10):
    Y  # Only includes values where X 2 and Y<10
```

Each filter condition is evaluated in order, and iteration continues only if all conditions succeed.

Iterating Over Maps:

Maps can be iterated over in two ways: values only, or key-value pairs using the pair syntax:

```
# Iterate over values only
Scores:[int]int = map{1 => 100, 2 => 200, 3 => 150}
TopScores := for (Score : Scores):
    Score  # Returns array{100, 200, 150}

# Iterate over key-value pairs
PlayerScores:[string]int = map{"Alice" => 100, "Bob" => 200}
for (PlayerName -> Score : PlayerScores):
    Print("{PlayerName} scored {Score}")
```

Maps preserve insertion order, so iteration order matches the order in which keys were added to the map.

String Iteration:

Strings can be iterated character by character:

```
CountVowels(Text:string):int =
    var Count:int = 0
    for (Char : Text):
        if (Char = 'a' or Char = 'e' or Char = 'i' or Char = 'o' or Char = 'u'):
            set Count += 1
    Count
```

Nested Iteration (Cartesian Products):

Multiple iteration sources create nested loops, producing the cartesian product:

```
PrintGrid():void =
    for (X := 1..3, Y := 1..3):
        Print("({X}, {Y})")
    # Produces: (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)
```

Filtering with Failure:

Verse's `for` expressions are particularly powerful when they leverage failure contexts, as they can naturally filter:

```
GetHighScorers(Players:[]player):[]player =
    for (Player : Players, Score := GetScore(Player), Score > 1000):
        Player  # Only players with score > 1000 are included
```

When any expression in the iteration header fails, that iteration is skipped. This allows elegant filtering without explicit `if` statements:

```
# Filter items under budget and apply transformation
AffordableItems(Items:[]item, Budget:float):[]float =
    for (Item : Items, Item.Price <= Budget):
        Item.Price * 1.1  # Apply 10% markup
```

For as an Expression:

Like other control flow constructs, `for` is an expression. When the body produces values, `for` collects them into an array:

```
# Collect player names
GetNames(Players:[]player):[]string =
    for (Player : Players):
        Player.Name  # Each iteration produces a string
```

This makes `for` a powerful tool for transforming collections without explicit accumulator variables.

Breaking from For Loops:

The `break` statement cannot exit `for` loops early.

Note on Continue:

Unlike many languages, Verse does not currently support a `continue` statement to skip to the next iteration. Instead, use conditional logic or failure-based filtering to achieve similar results:

```
# Instead of continue, use conditional blocks
ProcessItems(Items:[]item):void =
    for (Item : Items):
        if (Item.IsValid?):
            ProcessItem(Item)
        # No continue needed - just structure with conditions

# Or use failure-based filtering in the header
ProcessValidItems(Items:[]item):void =
    for (Item : Items, Item.IsValid?):
        ProcessItem(Item)  # Only valid items reach here
```

Range Iteration. The range operator `..` provides numeric iteration over integer sequences. Ranges are inclusive on both ends:

```
# Iterates: 1, 2, 3, 4, 5 (both bounds included)
for (I := 1..5):
    Print("Count: {I}")

# Single element range
for (I := 42..42):
    Print("Answer: {I}")  # Prints once: "Answer: 42"

# Empty range (start > end produces no iterations)
for (I := 5..1):
    Print("Never executes")  # Loop body never runs
```

The `..` operator is always inclusive. There is no exclusive range syntax.

Range bounds are evaluated in a specific order, and side effects occur predictably:

1. **Left bound evaluated first**, then right bound
2. **Both bounds always evaluated**, even if the range is empty
3. **Side effects happen in order**, regardless of whether iterations occur

While you cannot store ranges as values, you can create arrays using for expressions:

```
# This works because for produces an array, not because ranges are storable
Numbers:[]int = for (I := 1..5){ I * 2 }

# Can then iterate over the array normally
for (N : Numbers):
    Print("{N}")
```

The range exists only during the for expression evaluation; the resulting array is what gets stored.

Restrictions. The for loop has several important restrictions:

1. **Iteration source must be iterable:** Only ranges (`1..10`), arrays, maps, and strings can be iterated.
2. **Filters must be failable:** Filter conditions must contain at least one expression that can fail.
3. **Cannot redefine iteration variables:** You cannot redefine the iteration variable in the same clause.
4. **Cannot define mutable variables:** Using `var` to declare variables in the for clause is not allowed.

The range operator `..` has strict limitations that distinguish it from other iterable types. Ranges are *not first-class values*—they exist solely as syntactic sugar within for loop iteration clauses. Ranges cannot be used in some contexts where you might expect them to work:

```
# ERROR: Cannot store range in variable
MyRange := 1..10
for (I := MyRange):

# ERROR: Cannot pass range to function
ProcessRange(1..10)

# ERROR: Cannot use range as standalone expression
Result := 1..10

# ERROR: Cannot put range in array
Ranges := array{1..10}

# ERROR: Cannot index range
Value := (1..10)(5)

# ERROR: Cannot access members on range
Length := (1..10).Length
```

Ranges work exclusively with the `int` type. Other numeric types, booleans, types, or objects are not supported.

9.6 Return Statements

The `return` statement provides explicit early exits from functions, allowing you to terminate execution and return a value before reaching the end of the function body:

```
ValidateInput(Value:int):string =
    if (Value < 0):
        return "Error: Negative value"

    if (Value > 1000):
        return "Error: Value too large"

    "Valid"     # Implicit return
```

Return statements can only appear in specific positions within your code—they must be in "tail position," meaning they must be the last operation performed before control exits a scope. This restriction ensures predictable control flow:

```
# Valid: return is last operation
ProcessOrder(OrderId:int)<transacts>:string =
    if (Order := GetOrder[OrderId]):
        if (Order.IsValid[]):
            return "Processed"
    "Invalid order"

# Valid: return in both branches
GetStatus(Value:int):string =
    if (Value > 0):
        return "Positive"
    else:
        return "Non-positive"
```

Verse functions implicitly return the value of their last expression, so `return` is only needed for early exits:

```
# Implicit return
GetValue():int = 42  # Returns 42

# Explicit early return
GetDiscount(Price:float):float =
    if (Price < 10.0):
        return 0.0  # Early exit with no discount

    Price * 0.1  # Implicit return with 10% discount
```

In functions with the `<decides>` effect, `return` allows you to provide successful values from early exits, while still allowing other paths to fail:

```
RetryableOperation()<transacts>:string =
    if (Config := GetConfig[]):
        for (Retry := 1..Config.MaxRetries):
            if (Result := AttemptOperation[Retry]):
                return Result  # Success - exit immediately
    "Failed" # All retries exhausted
```

This pattern is common for search operations where you want to return immediately upon finding a match, but fail if no match is found.

9.7 Defer Statements

The `defer` statement schedules code to run just before successfully exiting the current scope. This makes it invaluable for cleanup operations like closing files, releasing resources, or logging:

```
ProcessFile(FileName:string)<transacts><decides>:void =
    File := OpenFile(FileName)?
    defer:
        CloseFile(File)  # Runs on success or early exit

    Contents := ReadFile(File)?
    ProcessContents[Contents]
    SaveResults[]
```

Deferred code executes when the scope exits successfully or through explicit control flow like `return`:

```
ProcessQuery()<transacts>:void =
    ConnId := OpenConnection()
    defer:
        CloseConnection(ConnId)  # Cleanup always needed

    for (Attempt := 1..5):
        if (Result := Query[ConnId]):
            ProcessResult(Result)
            return  # defer executes after return being called

    # defer executes before leaving the function scope on success
```

This is a subtle but crucial point: if a function fails due to speculative execution, deferred code does **not** execute. This is because failure triggers a rollback that undoes all effects, including the scheduling of defer blocks:

```
ExampleWithFailure()<transacts><decides>:void =
    ResourceId := AcquireResource[]
    defer:
        ReleaseResource(ResourceId) # Scheduled...

    RiskyOperation[ResourceId] # This fails!
    # defer does NOT run - entire scope was speculative and rolled back
```

When the `RiskyOperation` fails, the entire function also fails, and speculative execution undoes everything—including the defer registration. The resource cleanup never happens because the resource acquisition itself is rolled back.

This behavior ensures consistency: if a function fails, it's as if it never ran, including any cleanup code that was scheduled.

Execution Order:

When multiple `defer`s exist in the same scope, they execute in reverse order of definition (last-in, first-out), mimicking the stack-based cleanup of nested resources:

```
DatabaseTransaction()<transacts><decides>:void =
    DbId := OpenDatabase()
    defer:
        CloseDatabase(DbId)  # Executes second (outer resource)

    TxnId := BeginTransaction[DbId]
    defer:
        CommitTransaction(TxnId)  # Executes first (inner resource)

    DoWork[]  # Work happens with both resources active
    # Defers execute: CommitTransaction, then CloseDatabase
```

Defers and Async Cancellation:

Deferred code also executes when async operations are cancelled, such as when a `race` completes or a `spawn` is interrupted:

```
ProcessWithTimeout()<suspends><transacts>:void =
    race:
        block:
            Resource := AcquireResource()
            defer:
                ReleaseResource(Resource)  # Runs if cancelled

            LongRunningTask(Resource)

        block:
            Sleep(10.0)  # Timeout
    # If timeout wins, first block is cancelled and defer runs
```

This ensures cleanup happens even when concurrency control interrupts your code.

Nested Defers:

Defer statements can be nested within other defer blocks, creating a cascade of cleanup operations:

```
ProcessWithCleanup():void =
    Log("A")
    defer:
```

```
        Log("B")
        defer:
            Log("inner")  # Runs after B
        Log("C")
    Log("D")
    # Output: A D B C inner
```

The execution order follows the LIFO principle at each nesting level—inner defers execute after the outer defer's code, maintaining the stack-like cleanup order.

Defers in Control Flow:

Defers work correctly within all control flow constructs:

```
ProcessLoop():void =
    for (I := 0..2):
        Log("Start")
        defer:
            Log("Cleanup")  # Runs after each iteration
        Log("End")
    # Output: Start End Cleanup Start End Cleanup Start End Cleanup

ProcessWithIf(Condition:logic):void =
    if (Condition?):
        defer:
            Log("Then cleanup")
        Log("Then body")
    else:
        defer:
            Log("Else cleanup")
        Log("Else body")
```

Each control flow path executes its own defers independently.

Defer Restrictions. The defer statement has important restrictions to ensure predictable behavior:

1. **Cannot be empty:** Defer blocks must contain at least one expression:
2. **Cannot be used as expression:** Defer cannot be used in positions where a value is expected.
3. **Cannot cross boundaries:** Defer blocks cannot contain `return`, `break`, or other control flow that would exit the defer's scope.
4. **Cannot fail:** Expressions in defer blocks cannot fail.
5. **Cannot suspend directly:** Defer blocks cannot contain suspend expressions, but they can use `branch` or `spawn` for fire-and-forget async operations.

9.8 Profiling

Understanding how your code performs is crucial for optimization, and the `profile` expression measures execution time:

```
OptimizedCalculation():float =
    profile("Complex Math"):
        var Result:float = 0.0
        for (I := 1..1000000):
            set Result += Sin(I*1.0) * Cos(I*1.0)
        Result
```

The profile expression wraps around the code you want to measure, logging the execution time to the output. You can add descriptive tags to organize your profiling output, making it easier to identify bottlenecks in complex systems.

Profile expressions pass through their results transparently, meaning you can wrap them around any expression without changing the program's behavior:

```
PlayerDamage := profile("Damage Calculation"):
    BaseDamage * GetMultiplier() * GetCriticalBonus()
```

Chapter 10

Chapter 9: Failure

Most programming languages treat control flow as a matter of true or false, yes or no, one or zero. They evaluate boolean conditions and branch accordingly, creating a world of binary decisions that often requires checking conditions twice - once to see if something is possible, and again to actually do it. Verse takes a different approach. Instead of asking "is this true?", Verse asks "does this succeed?"

This distinction might seem subtle, but it changes how programs are written and reasoned about. Failure isn't an error or an exception-it's a first-class concept that drives control flow. When an expression fails, it doesn't crash your program or throw an exception that needs to be caught. Instead, failure is a normal, expected outcome that your code handles gracefully through the structure of the language itself.

Consider the simple act of accessing an array element. In traditional languages, you might write:

```
if (Index < Array.length) {  # Traditional, non-Verse
    Value = Array[index]
    Process(Value)
}
```

This checks validity separately from access, creating opportunities for bugs if the check and access become separated or if the array changes between them. In Verse, validation and access are unified:

```
if (Value := Array[Index]):
    Process(Value)
```

The array access either succeeds and binds the value, or it fails and execution moves on. There is no separate validation step, so the check and access cannot become inconsistent, and no undefined behavior from accessing invalid indices.

10.1 Failable Expressions

A failable expression is one that can either succeed and produce a value, or fail and produce nothing. This isn't the same as returning null or an error code - when an expression fails, it literally produces no value at all. The computation stops at that point in that particular path of execution.

Many operations are naturally failable. Array indexing fails when the index is out of bounds. Map lookups fail when the key doesn't exist. Comparisons fail when the values aren't equal. Division fails when dividing by zero. Even simple literals can be made to fail:

```
42      # Always succeeds with value 42
false?  # Always fails - the query of false
true?   # Always succeeds - the query of true
```

The query operator `?` turns any value into a failable expression. When applied to `false`, it always fails. When applied to any other value, it succeeds with that value. This simple mechanism provides immense power for controlling program flow.

You can create your own failable expressions through functions marked with the `<decides>` effect:

```
ValidateAge(Age:int)<decides>:int =
    Age >= 0    # Fails if age is negative
    Age <= 150  # Fails if age is unrealistic
    Age         # Returns the age if both checks pass
```

This function doesn't just check conditions - it embodies them. If the age is invalid, the function fails. If it's valid, it succeeds with the age value. The validation and the value are inseparable.

10.2 Failure Contexts

Not every part of a program can execute failable expressions. They can only appear in failure contexts–places where the language knows how to handle both success and failure. Each failure context defines what happens when expressions within it fail.

The most common failure context is the condition of an `if` expression:

```
if (Player := GetPlayerByName[Name], Score := GetPlayerScore[Player], Score > 100):
    Print("High scorer: {Name} with {Score} points!")
```

This `if` condition contains three potentially failable expressions. All must succeed for the body to execute. If any fails, the entire condition fails, and control moves

to the `else` branch (if present) or past the `if` entirely. The beauty is that each expression can use the results of previous ones - `Score` is only computed if we successfully found the `Player`.

The `for` expression creates a failure context for each iteration:

```
for (Item : Inventory, IsWeapon[Item], Damage := GetDamage[Item], Damage > 50):
    Print("Powerful weapon: {Item} with {Damage} damage")
```

Each iteration attempts the failable expressions. If they all succeed, the body executes for that item. If any fails, that iteration is skipped, and the loop continues with the next item. This creates a natural filtering mechanism without explicit conditional logic.

Functions marked with `<decides>` create a failure context for their entire body:

```
FindBestWeapon(Inventory:[]item)<decides>:item =
    var BestWeapon:?item = false
    var MaxDamage:int = 0

    for (Item : Inventory, IsWeapon[Item], Damage := GetDamage[Item]):
        if (Damage > MaxDamage):
            set BestWeapon = option{Item}
            set MaxDamage = Damage

    BestWeapon?  # Fails if no weapon was found
```

The function body is a failure context, allowing failable expressions throughout. The final line extracts the value from the option, failing if no weapon was found.

10.3 Speculative Execution

When you execute code in a failure context, changes to mutable variables are provisional—they only become permanent if the entire context succeeds. Functions that modify state in failure contexts must use the `<transacts>` or the `<writes>` effect specifier (see Effects):

```
m:=module:
    buyer := class:
        var PlayerGold:int
        AttemptPurchase(Cost:int)<transacts><decides>:void =
           set PlayerGold = PlayerGold - Cost  # Provisional change
           PlayerGold >= 0                    # Check if still valid
           # If this fails, PlayerGold reverts to original value
```

If the check fails, the subtraction is automatically rolled back. You don't need to manually restore the original value or check conditions before modifying state.

This transactional behavior makes complex state updates safe and predictable. Either everything succeeds and all changes are committed, or something fails and nothing changes.

```
game := class:
    var State:game_state
    ComplexOperation()<transacts><decides>:void =
       ModifyHealth()       # All these operations
       UpdateInventory()    # are provisional
       ChargeResources()    # until all succeed
       ValidateFinalState[] # If this fails, everything rolls back
```

The `game` class has multiple methods that update the `game_state`, before returning `ComplexOperation` validates that the object is in a valid state, if it is not, all changes performed in the method are rolled back.

10.4 The Logic of Failure

Verse provides logical operators that work with failure, creating an algebra for combining failable expressions.

The `and` operator ensures that both expression succeed. The `not` operator inverts success and failure:

```
if (not (Enemy := GetNearestEnemy[]) and Score > 0):
    Print("Coast is clear!")  # Executes when GetNearestEnemy fails
```

It is noteworthy that `Enemy` is not in scope within the `then` branch because it is under a `not`.

The `or` operator provides alternatives:

```
Weapon := PrimaryWeapon[] or SecondaryWeapon[] or DefaultWeapon?
```

This tries each option in order, stopping at the first success. It's not evaluating boolean conditions - it's attempting computations and taking the first one that succeeds.

You can combine these operators to create sophisticated control flow:

```
ValidatePlayer(Player:player)<decides>:void =
    IsAlive[Player]
    not IsStunned[Player]
    HasAmmunition[Player] or HasMeleeWeapon[Player]
```

This function succeeds only if the player is alive, not stunned, and has either ammunition or a melee weapon. Each line is a separate failable expression that must succeed.

Another interesting use case is `not not Exp` – it succeeds if `Exp` succeeds but all effects of `Exp` are thrown away. This is a way to try to see if a complex operation would succeed.

10.5 Expressions in Decides

A subtle feature is how relational expressions behave in decides contexts. When a comparison appears in a context that can handle failure, it doesn't just test a condition—it produces a value, specifically it returns its left-hand side. So `X>0` returns `X` and `0<=X` returns `0`. This behavior applies to all comparison operators in decides contexts:

```
GetIfNotEqual(X:int, Y:int)<decides>:int =
    X <> Y  # Returns X when X   Y, fails when X = Y

GetIfLessOrEqual(X:int, Limit:int)<decides>:int =
    X <= Limit  # Returns X when X   Limit, fails otherwise

GetIfGreaterThan(X:int, Threshold:int)<decides>:int =
    X > Threshold  # Returns X when X > Threshold, fails otherwise
```

Comparison expressions of the form `A op B` return `A` when the comparison succeeds, and fail when the comparison is false.

This creates concise validation functions that either return `Value` or fail:

```
ValidateInRange(Value:int, LwrBnd:int, UprBnd:int)<decides>:int =
    Value >= LwrBnd and Value <= UprBnd
```

10.6 Option Types

The option type and failure are intimately connected. An option either contains a value or is empty (represented by `false`). The query operator `?` converts between options and failure:

```
M()<decides>:void=
    MaybeValue:?int = option{42}  # An optional int
    Value := MaybeValue?          # Succeeds with 42

    Empty:?int = false            # An empty value
    Other := Empty?               # Failure
```

The `option{}` constructor works in reverse, converting failure to an empty option:

```
Result := option{RiskyComputation[]} # option{value} if computation succeeds
                                     # otherwise false
```

This bidirectional conversion makes options and failure interchangeable, allowing you to choose the most appropriate representation for your specific use case.

The option type `?T` represents values that may or may not be present. The question mark appears *before* the type, not after:

```
ValidSyntax:?int = option{42}      # Correct
```

The `?` prefix applies to any type:

```
MaybeNumber:?int = option{42}
MaybeText:?string = option{"hello"}
MaybePlayer:?player = option{player{}}
```

Use the `option{}` constructor to wrap a value:

```
Filled:?int = option{42}
Empty:?int  = false
Result:?int = option{RiskyComputation[]}  # false if computation fails
```

Empty options and `false` are equivalent—an empty option *is* `false`:

```
EmptyOption:?int = false
EmptyOption = false  # This comparison succeeds
```

Verse has a rich and flexible syntax which can also sometimes cause subtle bugs. A comma gives rise to a tuple in an `option` whereas a semicolon evaluates all values but retain only the last one:

```
# Comma creates tuple
option{1, 2}? = (1, 2)

# Semicolon creates sequence - last value is used
option{1; 2}? = 2
```

10.6.1 Unwrapping

The query operator `?` extracts values from options, failing if the option is empty:

```
M()<decides>:void=
    MaybeValue:?int = option{42}
    Value := MaybeValue?  # Succeeds with 42

```

```
    Empty:?int = false
    Other := Empty?  # Fails - cannot unwrap empty option
```

Unwrapping is only allowed in failure contexts:

```
# Valid: In if condition (failure context)
if (Value := MaybeInt?):
    Print("Got {Value}")

# Valid: In for loop (failure context)
for (Item : Items, ValidItem := ProcessItem(Item)?):
    UseItem(Item)

# Valid: In <decides> function body (failure context)
GetRequired(Maybe:?int)<decides>:int =
    Maybe?  # Fails if Maybe is empty
```

10.6.2 Nesting

Options can be nested to represent multiple layers of absence:

```
# Double-nested option
Double:??int = option{option{42}}

# Single unwrap gets outer option
if (Inner := Double?):
    if (TheValue := Inner?):
        # TheValue has type int, equals 42

# Double unwrap gets the value directly
Value := Double??  # Fails if either layer is empty
```

Helper functions also work with nested options:

```
UnpackNested(MaybeValue:??int):?int =
    if (Inner := MaybeValue?):
        Inner
    else:
        option{-1}  # Default for outer empty

DirectUnpack(MaybeValue:??int):int =
    if (Value := MaybeValue??):
        Value
    else:
        -1  # Default for any level empty
```

10.6.3 Chained Access

The `?.` operator provides safe member access on optional values:

```
entity := class:
    Name:string = "Unknown"
    Health:int = 100

MaybeEntity:?entity = option{entity{}}

# Safe field access
if (Name := MaybeEntity?.Name):
    Print("Entity: {Name}")  # Succeeds

# Safe method call
MaybeEntity?.TakeDamage(10)  # Only calls if entity present

# Chaining through multiple optionals
linked_list := class:
    Value:int = 0
    Next:?linked_list = false

Head:?linked_list = option{linked_list{Value := 1}}
SecondValue := Head?.Next?.Value  # Fails if any link is empty
```

The `?.` operator short-circuits—if the option is empty, the entire expression fails without evaluating the member access.

10.6.4 Defaulting

Use the `or` operator to provide fallback values for empty options:

```
MaybeValue:?int = false
Value := MaybeValue? or 42  # Yields 42

# Chaining multiple options
Primary:?string = false
Secondary:?string = option{"backup"}
Default:string = "default"

Result := Primary? or Secondary? or Default
```

10.6.5 Comparison

Empty options equal `false`, and filled options equal their unwrapped values when compared properly:

```
EmptyOption:?int = false
EmptyOption = false  # Succeeds

FilledOption:?int = option{1}
FilledOption? = 1  # Succeeds - unwrap then compare
```

However, you cannot directly compare optional and non-optional values without unwrapping:

```
Opt:?int = option{42}
Regular:int = 42

# Must unwrap to compare
if (Opt? = Regular):
    Print("Equal")
```

10.7 Failure with Optionals

Combining decides functions with optional return types, creates a system with multiple layers of failure. This pattern enables expressing complex conditions concisely while maintaining clarity.

A function can fail at two levels:

- *Function-level failure*: The entire function fails using `<decides>`
- *Value-level failure*: The function succeeds but returns an empty option

```
FindEligiblePlayer(Name:string)<decides>:?player =
    Name <> ""           # Layer 1: Fail if name is empty
    Player := LookupPlayer[Name]  # Layer 1: Fail if player not found
    option{IsActive[Player]}      # Layer 2: Empty option if player inactive
```

This function has three possible outcomes:

- *Function fails*: Empty name or player not found
- *Function succeeds with empty option*: Player found but inactive
- *Function succeeds with filled option*: Player found and active

Calling this function demonstrates the layered failure:

```
# Function-level failure
Result1 := FindEligiblePlayer[""]  # Fails, Result1 never assigned

# Function succeeds, returns empty option
if (Player := FindEligiblePlayer["InactiveUser"]?):
    # Won't execute - function succeeds but ? query fails
else:
    # Executes here

# Function succeeds, returns filled option
if (Player := FindEligiblePlayer["ActiveUser"]?):
    # Executes with Player bound to the active player
```

This pattern is particularly powerful for validation with different failure modes:

```
ValidateScore(Score:int)<decides>:?int =
    Score >= 0           # Layer 1: Reject negative scores (invalid input)
    option{Score <= 100} # Layer 2: Reject high scores (out of range)
```

The distinction between function-level and value-level failure lets you express different kinds of errors. Function-level failure typically means "this operation couldn't complete" while value-level failure means "the operation completed but the result doesn't meet the expected criteria."

10.8 Casts as Decides

Type casting in Verse is integrated into the failure system. A dynamic cast behaves just like a `<decides>` function call and similarly uses bracket syntax. For example `Type[value]` attempts to cast `value`'s type to `Type` and fails if unsuccessful.

This is also works with user defined types which must specify `<castable>`:

```
component := class<castable>:
    Name:string = "Component"

physics_component := class<castable>(component):
    Velocity:float = 0.0

# Casting as a decides operation
TryGetPhysics(Comp:component)<decides>:physics_component =
    physics_component[Comp]  # Succeeds if Comp is actually a physics_component
```

This makes type-based dispatch easily expressible:

```
ProcessComponent(Comp:component):void =
    if (Physics := physics_component[Comp]):
        UpdatePhysics(Physics)
    else if (Render := render_component[Comp]):
        UpdateRendering(Render)
    else:
        # Unknown component type
        UpdateGeneric(Comp)
```

The cast itself is the condition—no separate type checking needed. When the cast succeeds, you have both confirmed the type and obtained a properly-typed reference.

You can chain casts with other decides operations:

```
GetActivePhysicsComponent(Entity:entity)<decides>:physics_component =
    Comp := Entity.GetComponent[]  # Fails if no component
    Physics := physics_component[Comp]  # Fails if not physics
    IsActive[Physics]  # Fails if inactive
    Physics
```

Each step must succeed for the function to return a value. This creates self-documenting validation chains where type requirements are explicit.

Casts work with the `or` combinator for fallback types:

```
GetInteractable(Entity:entity)<decides>:component =
    physics_component[Entity] or
    trigger_component[Entity] or
    scripted_component[Entity]
```

This tries each cast in order, returning the first successful one. It's type-safe because all options share the common `component` base type.

10.9 Idioms and Patterns

As you work with failure, certain patterns emerge that solve common problems elegantly.

The validation chain pattern uses sequential failures to ensure all conditions are met:

```
ProcessAction(Action:action)<decides>:void =
    Player := GetActingPlayer[Action]
    IsValidTurn[Player]
    HasRequiredResources[Player, Action]
```

```
    Location := GetTargetLocation[Action]
    IsValidLocation[Location]
    ExecuteAction[Action]
```

Each line must succeed for execution to continue. This creates self-documenting code where preconditions are explicit and checked in order.

The first-success pattern tries alternatives until one works:

```
FindPath(Start:location, End:location)<decides>:path =
    DirectPath[Start, End] or
    PathAroundObstacles[Start, End] or
    ComplexPathfinding[Start, End]
```

This naturally expresses trying simple solutions before complex ones.

The filtering pattern uses failure to select items:

```
GetEliteEnemies(Enemies:[]enemy):[]enemy =
    for (Enemy : Enemies, Level := GetLevel[Enemy], Level >= 10):
        Enemy
```

Only enemies that have a level and whose level is at least 10 are included in the result.

The transaction pattern groups related changes:

```
TradeItems(var PlayerA:player, var PlayerB:player, ItemA:item, ItemB:item)<transacts><decides>
    RemoveItem(PlayerA, ItemA)
    RemoveItem(PlayerB, ItemB)
    AddItem(PlayerA, ItemB)
    AddItem(PlayerB, ItemA)
    ValidateTrade[PlayerA, PlayerB]
```

Either the entire trade succeeds, or nothing changes.

Optional Indexing

When working with optional containers, you can access their contents using specialized query syntax that combines optional checking with element access. Optional tuples support direct element access through the query operator:

```
MaybePair:?tuple(int, string) = option{(42, "answer")}

# Access first element
if (FirstValue := MaybePair?(0)):
    # FirstValue is 42 (type: int)
    Print("First: {FirstValue}")
```

```
7
8  # Access second element
9  if (SecondValue := MaybePair?(1)):
10     # SecondValue is "answer" (type: string)
11     Print("Second: {SecondValue}")
```

The syntax `Option?(index)` simultaneously:

- Queries whether the option is non-empty
- Accesses the tuple element at the given index
- Binds the element value if both succeed

Composition and Call Chains

Decides functions compose naturally, allowing complex operations to be built from simple, reusable pieces. When a decides function calls another decides function, failures propagate automatically.

```
1  ValidatePositive(X:int)<decides>:int =
2      X > 0
3
4  Double(X:int)<decides>:int =
5      Validated := ValidatePositive[X]  # Fails if X   0
6      Validated * 2
```

If `ValidatePositive` fails, `Double` fails immediately. The validated value flows through the chain.

Preserving failure context:

When calling decides functions in non-decides contexts, you must handle failure explicitly:

```
1  # This won't compile - ProcessPlayer doesn't have <decides>
2  # BadProcessPlayer(Name:string):void =
3  #    Player := FindPlayer[Name]  # ERROR: Unhandled failure
4
5  # Handle with if
6  ProcessPlayerWithIf(Name:string):void =
7      if (Player := FindPlayer[Name]):
8          UsePlayer(Player)
9
10 # Handle with or
11 ProcessPlayerWithOr(Name:string):void =
12     Player := FindPlayer[Name] or GetDefaultPlayer()
13     UsePlayer(Player)
```

Understanding composition helps you build complex validation logic from simple, testable pieces.

10.10 Runtime Errors

While failure (`<decides>`) represents normal control flow with transactional rollback, *runtime errors* represent unrecoverable conditions that terminate execution. Runtime errors propagate up the call stack, bypassing normal failure handling, and cannot be caught or recovered within Verse code.

The `Err()` function explicitly triggers a runtime error with an optional message:

```
ValidateInput(Value:int):int =
    if (Value < 0):
        Err("Negative values not allowed")
    Value
```

When a runtime error occurs, execution unwinds through the call stack, terminating the current operation:

```
DeepFunction()<transacts>:int =
    Log("C")
    Err("Fatal error")  # Runtime error here
    Log("D")            # Never executes
    return 1

MiddleFunction():int =
    Log("B")
    Result := DeepFunction()  # Error propagates through here
    Log("E")                  # Never executes
    return Result

TopFunction():void =
    Log("A")
    Value := MiddleFunction()  # Error propagates to here
    Log("F")                   # Never executes

# Execution order: A, B, C, then terminates
# Output: "ABC"
```

The runtime error propagates immediately, bypassing all subsequent code in the call chain.

Runtime errors propagate through asynchronous operations, terminating spawned tasks:

```
AsyncOperation()<suspends>:int =
    Log("Start")
    WaitTicks(1)
    Err("Async error")  # Runtime error during async execution
    WaitTicks(1)        # Never executes
    return 1

KickOff()<suspends>:void=
    # Error propagates out of spawned task
    spawn{ AsyncOperation() }
```

When a spawned task encounters a runtime error, that specific task terminates. The runtime error does not automatically propagate to the spawning context.

10.11 Living with Failure

Verse's approach to failure has roots in logic programming, where computations search for solutions rather than executing steps. When a path fails, the computation backtracks and tries alternatives. This non-deterministic model, while powerful, can be hard to reason about in its full generality. Verse tames this power by making failure contexts explicit and limiting backtracking to specific constructs. You get the benefits of logic programming - declarative code, automatic search, elegant handling of alternatives - without the complexity of full unification and unbounded backtracking.

Consider a simple logic puzzle solver:

```
SolvePuzzle(Constraints:[]constraint)<decides>:solution =
    var State:solution = InitialState()
    for (Constraint : Constraints):
        ApplyConstraint(State, Constraint)
    ValidateSolution[State]
    State
```

If any constraint can't be satisfied, the entire attempt fails. In a full logic programming language, this might trigger complex backtracking. In Verse, the failure model is simpler and more predictable while still being expressive enough for most problems.

Working effectively with failure in Verse requires a shift in mindset. Instead of thinking about error conditions that need to be avoided, think about success conditions that need to be met. Instead of defensive programming that checks everything before acting, write optimistic code that attempts operations and handles failure gracefully.

This perspective makes code more readable and intent more clear. When you see a function marked with `<decides>`, you know it represents a computation that might not have a result. When you see expressions in sequence within a failure context, you know they represent conditions that must all be met. When you see the `or` operator, you know it represents alternatives to try.

Failure in Verse isn't something to be feared or avoided - it's a tool to be embraced. It makes programs safer by eliminating certain categories of bugs. It makes code clearer by unifying validation and action. It makes complex operations simpler by providing automatic rollback. Most importantly, it aligns the way we write programs with the way we think about actions and decisions in the real world.

As you write more Verse code, you'll find that failure becomes second nature. You'll reach for failable expressions naturally when expressing conditions. You'll structure your functions to fail early when preconditions aren't met. You'll compose failures to create sophisticated control flow without nested conditionals. And you'll appreciate how this different way of thinking about control flow leads to code that is both more robust and more expressive than traditional approaches.

Chapter 11

Chapter 10: Structs and Enums

Structs and enums represent Verse's value-oriented type system, providing lightweight alternatives to classes for simple data aggregation and fixed sets of named values. Unlike classes with their object-oriented features, structs and enums focus on simplicity, immutability, and value semantics.

Structs bundle related data without methods or inheritance, perfect for mathematical types, configuration data, and simple records. Enums define fixed sets of named constants, replacing magic numbers with meaningful names and providing compile-time safety through exhaustive pattern matching.

Together, structs and enums complement classes and interfaces by offering simpler, more constrained type constructors optimized for specific use cases.

11.1 Structs

Structs provide lightweight data containers without the object-oriented features of classes. They're value types optimized for simple data aggregation, making them perfect for mathematical types, data transfer objects, and any scenario where you need a simple bundle of related values without behavior.

Structs group related data with minimal overhead:

```
damage_type:= enum:
    Physical
character := struct{}
vector2 := struct:
    X : float = 0.0
    Y : float = 0.0

color := struct:
```

```
    R : int = 0
    G : int = 0
    B : int = 0
    A : int = 255  # Alpha channel

damage_info := struct:
    Amount : int = 0
    Type : damage_type = damage_type.Physical
    Source : ?character = false
    IsCritical : logic = false
```

All struct fields are public and immutable by default. Structs cannot have methods, constructors, or participate in inheritance hierarchies. This simplicity makes them efficient and predictable.

11.1.1 Construction

Creating struct instances uses the same archetype syntax as classes:

```
Origin := vector2{}  # Uses defaults: (0.0, 0.0)
PlayerPos := vector2{X := 100.0, Y := 250.0}
RedColor := color{R := 255}  # Other channels default to 0/255

# Structs are values - assignment creates a copy
NewPos := PlayerPos
# NewPos is a separate instance with the same values
```

Since structs are value types, assigning a struct to a variable creates a copy of all its data. This differs from classes, which use reference semantics.

11.1.2 Comparison

Structs with all comparable fields support equality comparison:

```
vector3i := struct:
    X : int = 0
    Y : int = 0
    Z : int = 0

Origin := vector3i{}
UnitX := vector3i{X := 1}

if (Origin = vector3i{}):  # Succeeds - all fields match
    Print("At origin")

```

```
if (Origin = UnitX):  # Fails - X fields differ
    Print("Same position")
```

Comparison happens field by field, succeeding only if all corresponding fields are equal.

11.1.3 Persistable Structs

Structs can be marked as persistable for use with Verse's persistence system:

```
player_stats := struct<persistable>:
    HighScore : int = 0
    GamesPlayed : int = 0
    WinRate : float = 0.0

# Can be used in persistent storage
PlayerData : weak_map(player, player_stats) = map{}
```

Once published, persistable structs cannot be modified, ensuring data compatibility across game updates.

11.2 Enums

Enums define types with a fixed set of named values, perfect for representing states, types, or any concept with a known, finite set of alternatives. They make code more readable by replacing magic numbers with meaningful names and provide compile-time safety by restricting values to the defined set.

An enum lists all possible values for a type:

```
game_state := enum:
    MainMenu
    Playing
    Paused
    GameOver

damage_type := enum:
    Physical
    Fire
    Ice
    Lightning
    Poison

direction := enum:
    North
```

```
    East
    South
    West
```

Each value in the enum becomes a named constant of that enum type. The compiler ensures that variables of an enum type can only hold one of these defined values. Enums can even be empty:

```
placeholder := enum{}  # Valid but rarely useful
```

Enums introduce both a type and a set of values, and it's crucial to distinguish between them:

```
status := enum:
    Active
    Inactive

# status is the TYPE
# status.Active and status.Inactive are VALUES

CurrentStatus:status = status.Active  # OK - value of type status
```

You cannot use the enum type where a value is expected:

```
# ERROR: Cannot use type as value
BadAssignment:status = status  # Compile error
set CurrentStatus = status     # Compile error

# CORRECT: Use enum values
GoodAssignment:status = status.Active  # OK
set CurrentStatus = status.Inactive    # OK
```

This distinction prevents confusion and ensures type safety. The enum type defines what values are possible, while enum values are the actual constants you use in your code.

11.2.1 Restrictions

Enums have specific syntactic requirements that keep their usage clear and unambiguous:

Enums must be direct right-hand side of definitions:

```
# Valid
Priority := enum:
    Low
    Medium
```

```
    High

# Invalid - cannot use enum in expressions
Result := -enum{A, B}      # Compile error
Value := enum{X, Y} + 1    # Compile error
```

Enums must be module or class-level definitions:

```
# Valid
MyEnum := enum:
    Value1
    Value2

# Invalid - cannot define local enums
ProcessData():void =
    LocalEnum := enum{A, B}  # Compile error - no local enums
```

These restrictions ensure enums remain stable, referenceable definitions throughout your codebase rather than ephemeral local values.

11.2.2 Using Enums

Enums provide type-safe alternatives to error-prone string or integer constants:

```
var CurrentState:game_state = game_state.MainMenu

ProcessInput(Input:string):void =
    case (CurrentState):
        game_state.MainMenu =>
            if (Input = "Start"):
                set CurrentState = game_state.Playing
        game_state.Playing =>
            if (Input = "Pause"):
                set CurrentState = game_state.Paused
        game_state.Paused =>
            if (Input = "Resume"):
                set CurrentState = game_state.Playing
            else if (Input = "Quit"):
                set CurrentState = game_state.MainMenu
        game_state.GameOver =>
            if (Input = "Restart"):
                set CurrentState = game_state.MainMenu
```

The `case` expression with enums provides powerful pattern matching with exhaustiveness checking that ensures you handle all possible values correctly.

11.2.3 Open vs Closed Enums

Enums can be marked as open or closed, fundamentally affecting how they can evolve and how they interact with pattern matching:

```
# Closed enum - cannot add values after publication
day_of_week := enum<closed>:  # <closed> is the default
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday

# Open enum - can add new values after publication
weapon_type := enum<open>:
    Sword
    Bow
    Staff
    # Can add Wand, Dagger, etc. in updates
```

Closed enums (the default) commit to a fixed set of values forever. This allows the compiler to verify that case expressions handle all possibilities exhaustively. Use closed enums for truly fixed sets: days of the week, cardinal directions, fundamental game states.

Open enums allow new values to be added in future versions. This flexibility comes at a cost: case expressions cannot be exhaustive since future values might exist. Use open enums for extensible sets: item types, enemy types, damage types, or any content that may grow.

11.2.4 Exhaustiveness

The interaction between enum types and case expressions follows sophisticated rules that prevent bugs while enabling both safety and flexibility. Understanding these rules is essential for working with enums effectively.

Closed Enums with Full Coverage:

When your case expression handles every value in a closed enum, no wildcard is needed:

```
day := enum:
    Monday
    Tuesday
    Wednesday
```

```
5
6  # Exhaustive - all values covered
7  GetDayType(D:day):string =
8      case (D):
9          day.Monday => "Weekday"
10         day.Tuesday => "Weekday"
11         day.Wednesday => "Weekday"
12     # No wildcard needed - all values handled
```

Adding a wildcard when all cases are covered triggers an unreachable code warning:

```
1  # Warning: unreachable wildcard
2  GetDayType(D:day):string =
3      case (D):
4          day.Monday => "Weekday"
5          day.Tuesday => "Weekday"
6          day.Wednesday => "Weekday"
7          _ => "Unknown"  # WARNING: unreachable - all values already matched
```

Closed Enums with Partial Coverage:

If you don't match all values, you must either provide a wildcard or be in a `<decides>` context:

```
1  day := enum:
2      Monday
3      Tuesday
4      Wednesday
5      Thursday
6
7  # With wildcard - OK
8  GetWeekStartWildCard(D:day):string =
9      case (D):
10         day.Monday => "Week start"
11         _ => "Mid-week"
12
13 # Without wildcard but in <decides> context - OK
14 GetWeekStartDecides(D:day)<decides>:string =
15     case (D):
16         day.Monday => "Week start"
17         # Missing other days causes failure
18
19 # Without either - COMPILE ERROR
20 # GetWeekStartBad(D:day):string =
21 #    case (D):
```

```
#        day.Monday => "Week start"
#        # ERROR: Missing cases and no wildcard
```

Open Enums Always Require Wildcard or `<decides>`:

Open enums can have new values added after publication, so they can never be exhaustive.
This is to ensure backwards compatibility of functions using them (see also Publishing Functions):

```
weapon := enum<open>:
    Sword
    Bow
    Staff

# Must have wildcard - OK
GetWeaponClassWildCard(W:weapon):string =
    case (W):
        weapon.Sword => "Melee"
        weapon.Bow => "Ranged"
        weapon.Staff => "Magic"
        _ => "Unknown"  # REQUIRED - future values may exist

# In <decides> context without wildcard - OK
GetWeaponClassDecides(W:weapon)<decides>:string =
    case (W):
        weapon.Sword => "Melee"
        weapon.Bow => "Ranged"
        weapon.Staff => "Magic"
        # Can fail for unknown (future) values

# Without either - COMPILE ERROR
# GetWeaponClassBad(W:weapon):string =
#    case (W):
#        weapon.Sword => "Melee"
#        weapon.Bow => "Ranged"
#        weapon.Staff => "Magic"
#        # ERROR: Open enum requires wildcard or <decides>
```

Even if you match all currently defined values in an open enum, you still need a wildcard or `<decides>` context because new values might be added in future versions.

Summary of Exhaustiveness Rules:

Enum Type	Case Coverage	Wildcard	Context	Result
Closed	Full	No	Any	Valid - exhaustive
Closed	Full	Yes	Any	Warning - unreachable wildcard
Closed	Partial	Yes	Any	Valid
Closed	Partial	No	`<decides>`	Valid - unmatched values fail
Closed	Partial	No	Non-`<decides>`	Error - missing cases
Open	Any	Yes	Any	Valid
Open	Any	No	`<decides>`	Valid - unmatched values fail
Open	Any	No	Non-`<decides>`	Error - open enum needs wildcard

These rules ensure that closed enums provide safety through exhaustiveness while open enums require explicit handling of unknown values.

11.2.5 Unreachable Case Detection

The compiler actively detects unreachable cases in case expressions, helping you identify dead code and logic errors:

Duplicate cases are flagged as unreachable:

```
status := enum:
    Active
    Inactive
    Pending

# ERROR: Duplicate case is unreachable
GetStatusCode(S:status):int =
```

```
    case (S):
        status.Active => 1
        status.Inactive => 2
        status.Pending => 3
        status.Pending => 4  # ERROR: unreachable - already matched above
```

Cases after wildcards are always unreachable:

```
# ERROR: Case after wildcard
GetStatusCode(S:status):int =
    case (S):
        status.Active => 1
        _ => 0  # Wildcard matches everything
        status.Inactive => 2  # ERROR: unreachable - wildcard already matched
```

These errors prevent logic bugs where you think you're handling specific cases but the code will never execute.

11.2.6 The `@ignore_unreachable` Attribute

Sometimes you intentionally want unreachable cases—for testing, migration, or defensive programming. The `@ignore_unreachable` attribute suppresses unreachable warnings and errors for specific cases:

```
status := enum:
    Active
    Inactive

ProcessStatus(S:status):int =
    case (S):
        status.Active => 1
        status.Inactive => 2
        @ignore_unreachable status.Inactive => 3  # No error
        @ignore_unreachable _ => 0  # No unreachable warning
```

This attribute only affects cases it's applied to. Other unreachable cases without the attribute still produce errors:

```
ProcessStatus(S:status):int =
    case (S):
        status.Active => 1
        status.Inactive => 2
        @ignore_unreachable status.Inactive => 3  # Suppressed
        status.Active => 4  # ERROR: still unreachable without attribute
```

Use `@ignore_unreachable` sparingly, primarily during refactoring or when maintaining multiple code paths for testing purposes.

11.2.7 Explicit Qualification

Enumerators can collide with identifiers in parent scopes. When this happens, you can use explicit qualification to disambiguate:

```
# Top level 'Start'
Start:int = 0

# Enum wants to use 'Start' as enumerator
game_state := enum:
    (game_state:)Start  # Explicit qualification avoids collision
    Playing
    Paused

# Now both are accessible
OuterStart:int = Start             # References the int
StateStart:game_state = game_state.Start  # References the enum value
```

The syntax `(enum_name:)enumerator` explicitly qualifies the enumerator, preventing conflicts with outer-scope symbols.

Using Reserved Words as Enum Values:

Qualification also allows you to use reserved words and keywords as enum values, which would otherwise cause errors:

```
# Using reserved words as enum values
keyword_enum := enum:
    (keyword_enum:)public    # OK: reserved word qualified
    (keyword_enum:)for       # OK: keyword qualified
    (keyword_enum:)class     # OK: reserved word qualified
    Regular                  # Normal enum value

# Without qualification - errors
# bad_enum := enum:
#    public    # Error: reserved word
#    for       # Error: reserved keyword
```

This is particularly useful when modeling language constructs, access levels, or any domain where reserved words make natural value names.

Self-Referential Enum Values:

You can even use the enum's own name as a value when qualified:

```
recursive_enum := enum:
    (recursive_enum:)recursive_enum  # OK: qualified with enum name
    OtherValue

# Without qualification - error
# bad_recursive := enum:
  #  bad_recursive  # Error: shadows the type name
```

11.2.8 Attributes on Enums

Enums support custom attributes, both on the enum type itself and on individual enumerators:

```
# Define my_attribute by inheriting from the attribute class
@attribscope_enum
my_attribute := class(attribute):
    MyMetaData:string = "I'm default metadata"
    # category<constructor>(Name:string)<computes> := my_attribute{}

    # Apply to enum and enumerators
@my_attribute{}
game_state := enum:
    @my_attribute{MyMetaData = "Initial"}
    MainMenu

    @my_attribute{MyMetaData = "Active"}
    Playing

    @my_attribute{MyMetaData = "Paused"}
    Paused
```

Attributes must be marked with the appropriate scopes (`@attribscope_enum` for enum types, `@attribscope_enumerator` for individual values) or the compiler will reject them. This provides metadata capabilities for reflection, serialization, or custom tooling.

11.2.9 Comparison

Enum values are fully comparable, meaning they support both equality (`=`) and inequality (`<>`) operators. This makes them ideal for state tracking and conditional logic:

```
CurrentWeapon := weapon_type.Sword
if (CurrentWeapon = weapon_type.Sword):
    PlaySwordAnimation()
```

```
4
5  PreviousState := game_state.Playing
6  if (CurrentState <> PreviousState):
7      OnStateChanged(PreviousState, CurrentState)
```

Enum values from the same enum type can be compared, while values from different enum types are always unequal:

```
1  letters := enum:
2      A, B, C
3
4  numbers := enum:
5      One, Two, Three
6
7  Test()<decides>:letters =
8      letters.A = letters.A    # Succeeds - same value
9      letters.A <> letters.B   # Succeeds - different values
10     letters.A <> numbers.One # Succeeds - different enum types
```

Because enums are comparable, they can be used as map keys, stored in sets, and used with generic functions that require comparable types:

```
1  # Enums as map keys
2  StateIDs:[game_state]int = map{
3      game_state.Menu => 0,
4      game_state.Playing => 1,
5      game_state.Paused => 2
6  }
7
8  # In generic functions
9  FindStateID(States:[]game_state, Target:game_state)<decides>:int =
10     for (
11         State : States, State = Target,
12         ID := StateIDs[State]
13     ):
14         return ID
15     false? # fails if state is not found
```

Part III

Part III: Object-Oriented Programming

Chapter 12

Chapter 11: Classes and Interfaces

Classes and interfaces are Verse's object-oriented building blocks that enable rich type hierarchies with inheritance, polymorphism, and interface-based contracts. Classes provide object-oriented programming with fields, methods, and single inheritance, enabling you to model complex hierarchies of game entities with shared behavior and specialized implementations. Interfaces define contracts that classes must fulfill, promoting loose coupling and enabling multiple inheritance of behavior specifications.

Together, classes and interfaces form a powerful system for modeling game entities, components, and systems with both is-a relationships (through class inheritance) and can-do contracts (through interface implementation).

Let's explore classes first, then delve into interfaces and how they complement each other.

12.1 Classes

Classes form the backbone of object-oriented programming in Verse. A class serves as a blueprint for creating objects that share common properties and behaviors. When you define a class, you're creating a new type that bundles data (fields) with operations on that data (methods), encapsulating related functionality into a cohesive unit.

Class definitions occur at module scope. You cannot define a class inside another class, struct, interface, or function. Classes are top-level type definitions that establish the type system's structure:

```
# Valid: class at module scope
my_module := module:
    entity := class:
```

```
        ID:int

# Invalid: class inside another class
# outer := class:
#     inner := class:  # ERROR: classes must be at module scope
#         Value:int
```

The simplest form of a class groups related data together. Consider modeling a character in your game:

```
character := class:
    Name : string
    var Health : int = 100
    var Level : int = 1
    MaxHealth : int = 100
```

This class definition establishes several important concepts. Fields without the `var` modifier are immutable after construction—once you create a character with a specific name, that name cannot change. Fields marked with `var` are mutable and can be modified after the object is created (see Mutability for details on `var` and `set`). Default values provide sensible starting points, making object construction more convenient while ensuring objects start in valid states.

12.1.1 Object Construction

Creating instances of a class involves specifying values for its fields through an archetype expression:

```
Hero := character{Name := "Aldric", Health := 100, Level := 5}
Villager := character{Name := "Martha"}  # default values for unspecified fields
```

The archetype syntax uses named parameters, making the construction explicit and self-documenting. Any field with a default value can be omitted from the archetype, and the default will be used. Fields without defaults must be specified, ensuring objects are always fully initialized. Fields can be passed to an archetype in any order.

12.1.2 Methods

Classes become truly powerful when you add methods that operate on the class's data:

```
character := class:
    Name : string
    var Health : int = 100
    var Level : int = 1
```

```
    var MaxHealth : int = 100

    TakeDamage(Amount : int) : void =
        set Health = Max(0, Health - Amount)

    Heal(Amount : int) : void =
        set Health = Min(MaxHealth, Health + Amount)

    IsAlive()<decides>:void= Health > 0

    LevelUp() : void =
        set Level += 1
        set MaxHealth = 100 + (Level * 10)
        set Health = MaxHealth  # Full heal on level up
```

Methods have access to all fields of the class and can modify mutable fields. They encapsulate the logic for how objects of the class should behave, ensuring that state changes happen in controlled, predictable ways.

All methods in non-abstract classes must have implementations. Unlike interfaces (which can declare abstract methods), a concrete class method declaration without an implementation is an error:

```
# Valid: method with implementation
valid_class := class:
    Compute():int = 42

# Invalid: method without implementation in concrete class
# invalid_class := class:
#     Compute():int  # ERROR: needs implementation
```

12.1.3 Blocks for Initialization

Classes can include `block` clauses in their body, which execute when an instance is created. These blocks run initialization code that goes beyond simple field assignment, allowing you to perform setup logic, validation, or side effects during construction:

```
logged_entity := class:
    ID:int
    var CreationTime:float = 0.0

    block:
        # This executes when an instance is created
        Print("Creating entity with ID: {ID}")
```

```
        set CreationTime = GetCurrentTime()

# Entity := logged_entity{ID := 42}
# Prints: "Creating entity with ID: 42"
```

Block clauses have access to all fields of the class, including `Self`, and can modify mutable fields. They execute in the order they appear in the class definition:

```
multi_step_init := class:
    var Step1:int = 0
    var Step2:int = 0

    block:
        set Step1 = 10

    var Step3:int = 0

    block:
        set Step2 = Step1 + 5  # Can access earlier fields
        set Step3 = Step2 * 2

# Instance := multi_step_init{}
# Instance.Step1 = 10, Step2 = 15, Step3 = 30
```

Execution order with inheritance: When a class inherits from another class, the Verse VM executes blocks in subclass-before-superclass order, while the BP VM uses superclass-before-subclass order. For portable code, avoid depending on the execution order of blocks across inheritance hierarchies.

Constraints on block clauses:

- Blocks cannot contain failure (`<decides>`) operations
- Blocks cannot call suspending (`<suspends>`) functions
- Blocks can use `defer` statements, which execute when the block exits
- Block clauses are only allowed in classes, not in interfaces, structs, or modules

Block clauses are particularly useful for:

- Logging object creation
- Computing derived values during initialization
- Registering objects with global systems
- Performing validation that goes beyond simple field checks

12.1.4 Self

Within class methods, `Self` is a special keyword that refers to the current instance of the class. Each method invocation has its own `Self` that refers to the specific object the method was called on.

You can use `Self` in multiple ways within method bodies:

- access fields of the instance
- calling methods of the instance
- pass the instance to other functions
- return the instance

```
character := class:
    var Name : string
    var Config:[string]string = map{}

    Announce() : void =
        # Using Self to pass the whole object
        LogCharacterAction(Self, "announced")


    SetOption(Key:string, Value:string):builder =
        set Config[Key] = Value
        Self  # Return this instance for method chaining


    SetName(NewName:stirng):void =
       set Self.Name = NewName    # Set the name of this instance
       Self.Announce()            # Call a method of this instance
```

You can capture `Self` when creating nested objects:

```
container := class:
    ID:int

    CreateChild():child_with_parent =
        child_with_parent{Parent := Self}  # Capture this instance

child_with_parent := class:
    Parent:container

# C := container{ID := 42}
# Child := C.CreateChild()
# Child.Parent.ID = 42  # Child stores reference to C
```

12.1.5 Inheritance

Classes support single inheritance, allowing you to create specialized versions of existing classes. This creates an "is-a" relationship where the subclass is a more specific type of the superclass:

```
entity := class:
    var Position : vector3 = vector3{}
    var IsActive : logic = true

    Activate() : void = set IsActive = true
    Deactivate() : void = set IsActive = false

character := class(entity):  # character inherits from entity
    Name : string
    var Health : int = 100

    TakeDamage(Amount : int) : void =
        set Health = Max(0, Health - Amount)
        if (Health = 0):
            Deactivate()  # Can call inherited methods

player := class(character):  # player inherits from character
    var Score : int = 0
    var Lives : int = 3

    AddScore(Points : int) : void =
        set Score += Points
```

Inheritance creates a type hierarchy where a `player` is also a `character`, and a `character` is also an `entity`. This means you can use a `player` object anywhere a `character` or `entity` is expected, enabling polymorphic behavior.

Important constraints on inheritance:

1. **Single class inheritance only:** A class can inherit from at most one other class, though it can implement multiple interfaces. Multiple class inheritance is not supported:

```
base1 := class:
    Value1:int

base2 := class:
    Value2:int

```

```
# Valid: inherit from one class and multiple interfaces
interface1 := interface:
    Method1():void

interface2 := interface:
    Method2():void

derived := class<abstract>(base1, interface1, interface2):
    # Valid: one class, multiple interfaces
    Method1<override>():void = {}
    Method2<override>():void = {}

# Invalid: cannot inherit from multiple classes
# invalid := class(base1, base2):  # ERROR
```

2. **No shadowing of data members:** Subclasses cannot declare fields with the same name as fields in their superclass. This prevents ambiguity and ensures clear data ownership:

```
base := class:
    Value:int

# Invalid: cannot shadow parent's field
# derived := class(base):
#     Value:int  # ERROR: shadowing base.Value
```

3. **No method signature changes:** When overriding a method, you must use the exact same signature. Changing parameter types or return types creates a shadowing error:

```
base := class:
    Compute():int = 42

# Invalid: different return type
# derived := class(base):
#     Compute():float = 3.14  # ERROR: signature doesn't match
```

To override a method, use the `<override>` specifier with the matching signature.

12.1.6 Super

Within a subclass, you can use the `super` keyword to refer to the superclass type. This is primarily used to access the superclass's implementation or to construct a superclass instance:

```
entity := class:
    ID:int
    Name:string

    Display():void =
        Print("Entity {ID}: {Name}")

character := class(entity):
    Health:int

    Display<override>():void =
        # Create a superclass instance to call its method
        super{ID := ID, Name := Name}.Display()
        Print("Health: {Health}")
```

The `super` keyword represents the superclass type itself. When you write `super{...}`, you're creating an instance of the superclass with the specified field values. This allows you to delegate to superclass behavior while adding subclass-specific functionality.

Within an overriding method, you can call the parent class's implementation using the `(super:)` syntax. This is the primary way to invoke parent method implementations while adding or modifying behavior:

```
base := class:
    Method():void =
        Print("Base implementation")

derived := class(base):
    Method<override>():void =
        # Call parent implementation first
        (super:)Method()
        Print("Derived implementation")

# Creates instance and calls Method()
# derived{}.Method()
# Output:
# Base implementation
# Derived implementation
```

The `(super:)` syntax explicitly calls the parent class's version of the current method. This is cleaner and more efficient than constructing a parent instance with `super{...}` when you only need to call parent methods.

Basic Usage:

```
entity := class:
    Position:vector3

    Move(Delta:vector3):void =
        Print("Entity moving by {Delta}")
        # Update position logic here

character := class(entity):
    var Stamina:float = 100.0

    Move<override>(Delta:vector3):void =
        # Call parent movement logic
        (super:)Move(Delta)
        # Add character-specific behavior
        set Stamina -= 1.0
```

With Effect Specifiers:

The `(super:)` syntax works seamlessly with all effect specifiers:

```
async_base := class:
    Process()<suspends>:void =
        Sleep(1.0)
        Print("Base processing")

async_derived := class(async_base):
    Process<override>()<suspends>:void =
        # Parent method suspends, so this suspends too
        (super:)Process()
        Print("Derived processing")

transactional_base := class:
    var Value:int = 0

    Update()<transacts>:void =
        set Value += 1

transactional_derived := class(transactional_base):
    var Counter:int = 0

```

```
    Update<override>()<transacts>:void =
        (super:)Update()
        set Counter += 1
```

Virtual Dispatch Through Parent Methods:

When parent methods call other methods, virtual dispatch still applies based on the actual object type. This means `Self` binds to the derived instance even when calling through `(super:)`:

```
base := class:
    # Virtual method that can be overridden
    GetValue()<computes>:int = 10

    # Parent method that uses GetValue
    ComputeDouble()<computes>:int =
        2 * GetValue()  # Calls derived GetValue if overridden

derived := class(base):
    # Override GetValue to return different value
    GetValue<override>()<computes>:int = 20

    # Override ComputeDouble to call parent, but GetValue dispatch is virtual
    ComputeDouble<override>()<computes>:int =
        # Calls base.ComputeDouble, which calls derived.GetValue!
        (super:)ComputeDouble()

# derived{}.ComputeDouble()  # Returns 40, not 20
```

In this example, even though `ComputeDouble` calls the parent implementation, the `GetValue()` call inside the parent uses virtual dispatch and calls the derived version.

With Overloaded Methods:

The `(super:)` syntax works with overloaded methods, calling the parent's version of the same overload:

```
base := class:
    Process(X:int):void =
        Print("Base int: {X}")

    Process(S:string):void =
        Print("Base string: {S}")

derived := class(base):
    Process<override>(X:int):void =
```

```
        (super:)Process(X)  # Calls parent's int overload
        Print("Derived int: {X}")

    Process<override>(S:string):void =
        (super:)Process(S)  # Calls parent's string overload
        Print("Derived string: {S}")
```

Return Type Covariance:

When overriding methods with `(super:)`, the return type can be a subtype of the parent's return type (covariant return types):

```
base_type := class:
    Name:string

derived_type := class(base_type):
    Value:int

base := class:
    Create():base_type =
        base_type{Name := "base"}

derived := class(base):
    # Override with more specific return type
    Create<override>():derived_type =
        # Can still call parent even with different return type
        Parent := (super:)Create()
        derived_type{Name := Parent.Name, Value := 42}
```

12.1.7 Method Overriding

Subclasses can override methods defined in their superclasses to provide specialized behavior:

```
entity := class:
    OnUpdate<public>() : void = {}  # Default no-op implementation

enemy := class(entity):
    var Target : ?character = false

    OnUpdate<override>()<transacts> : void =
        if (Target?.IsAlive[]):
            MoveToward(Target)
        else:
            Patrol()
```

```
12
13 turret := class(entity):
14     var Rotation:int= 0
15
16     OnUpdate<override>()<transacts>: void =
17         if (V:= Mod[Rotation, 360]):
18             set Rotation = V
19         ScanForTargets()
```

The override mechanism ensures that the correct method implementation is called based on the actual type of the object, not the type of the variable holding it. This is the foundation of polymorphic behavior in object-oriented programming.

12.1.8 Constructor Functions

Classes don't have traditional constructor methods like you might find in other object-oriented languages. Instead, Verse provides two approaches to object construction: direct field initialization through archetype expressions, and constructor functions for complex initialization scenarios.

For simple cases where you just need to set field values, use archetype expressions directly:

```
1 player := class:
2     Name:string
3     var Health:int = 100
4     Level:int = 1
5
6 # Direct construction with archetype
7 # Hero := player{Name := "Aldric", Health := 150, Level := 5}
```

When you need validation, computation, or complex initialization logic, use constructor functions annotated with `<constructor>`:

```
1 MakePlayer<constructor>(InName:string, InLevel:int)<transacts> := player:
2     Name := InName
3     Level := InLevel
4     Health := InLevel * 100
```

Here's an example of calling this constructor:

```
1 Hero := MakePlayer("Aldric", 5) # Call constructor function
```

Constructor functions are regular functions that return class instances, but the `<constructor>` annotation enables special capabilities like delegating to other constructors. When calling a constructor function from normal code, use just the function name—the `<constructor>` annotation only appears in the definition.

Constructor functions can have effects that control their behavior. Common effects include `<computes>`, `<allocates>`, and `<transacts>`. A particularly useful effect is `<decides>`, which allows constructors to fail if preconditions aren't met:

```
MakeValidPlayer<constructor>(InName:string, InLevel:int)<transacts><decides> :=
    player:
         Name := InName
         Level := block:
                 InLevel > 0
                 InLevel <= MaxLevel
                 InLevel
         Health := InLevel * 100
```

Here's an example using the validated constructor with failure handling:

```
# Constructor can fail - use with failure syntax
if (Player := MakeValidPlayer["Hero", 5]):
    # Construction succeeded
    AddPlayer(Player)
else:
    # Construction failed - level out of range
```

Constructor functions cannot use the `<suspends>` effect. Construction must complete synchronously to maintain object consistency.

12.1.9 Overloading Constructors

You can provide multiple constructor functions with different parameter signatures, allowing flexible object creation:

```
entity := class:
    Name:string
    var Health:int = 100
    Position:vector3

# Constructor with all parameters
MakeEntity<constructor>(Name:string, Health:int, Position:vector3) := entity:
    Name := Name
    Health := Health
    Position := Position

# Constructor with defaults
MakeEntity<constructor>(Name:string, Position:vector3) := entity:
    Name := Name
    Health := 100
    Position := Position
```

```
17
18  # Constructor for origin placement
19  MakeEntity<constructor>(Name:string) := entity:
20      Name := Name
21      Health := 100
22      Position := vector3{X := 0.0, Y := 0.0, Z := 0.0}
23
24  # Each overload can be called based on arguments
25  # Enemy1 := MakeEntity("Goblin", 50, SpawnPoint)
26  # Enemy2 := MakeEntity("Guard", PatrolPoint)
27  # NPC := MakeEntity("Shopkeeper")
```

12.1.10 Delegating Constructors

Constructor functions can delegate to other constructors, enabling code reuse and constructor chaining. This is particularly important for inheritance hierarchies where subclass constructors need to initialize superclass fields.

When delegating to a parent class constructor from a subclass, you must initialize the subclass fields first, then call the parent constructor using the qualified `<constructor>` syntax within the archetype:

```
1   entity := class:
2       Name:string
3       var Health:int
4
5   MakeEntity<constructor>(Name:string, Health:int) := entity:
6       Name := Name
7       Health := Health
8
9   character := class(entity):
10      Class:string
11      Level:int
12
13  # Subclass constructor delegates to parent constructor
14  MakeCharacter<constructor>(Name:string, Class:string, Level:int) := character:
15      # Initialize subclass fields first
16      Class := Class
17      Level := Level
18      # Then delegate to parent constructor
19      MakeEntity<constructor>(Name, Level * 100)
20
21  # Hero := MakeCharacter("Aldric", "Warrior", 5)
```

Constructor functions can also forward to other constructors of the same class:

```
player := class:
    Name:string
    var Score:int

# Primary constructor
MakePlayer<constructor>(Name:string, Score:int) := player:
    Name := Name
    Score := Score

# Convenience constructor forwards to primary
MakeNewPlayer<constructor>(Name:string) := player:
    # Delegate to another constructor of the same class
    MakePlayer<constructor>(Name, 0)
```

Here's an example of calling the constructor:

```
NewPlayer := MakeNewPlayer("Alice")
```

When delegating to a constructor of the same class, the delegation replaces all field initialization—any fields you initialize before the delegation are ignored. When delegating to a parent class constructor, your subclass field initializations are preserved, and the parent constructor initializes the parent fields.

12.1.11 Order of Execution

Understanding execution order is crucial for correct initialization:

1. **Archetype expression:** Field initializers execute in the order they're written in the archetype
2. **Delegating constructor:** Subclass fields are initialized first, then the parent constructor runs
3. **Class body blocks:** When using direct archetype construction, blocks in the class definition execute before field initialization

For delegating constructors to parent classes:

```
base := class:
    BaseValue:int

MakeBase<constructor>(Value:int) := base:
    block:
        Print("Base constructor")
    BaseValue := Value

```

```
derived := class(base):
    DerivedValue:int

MakeDerived<constructor>(Base:int, Derived:int) := derived:
    # This executes first
    DerivedValue := Derived
    # Then parent constructor executes
    MakeBase<constructor>(Base)
```

Here's an example showing execution order:

```
# Prints: "Base constructor"
# Results in: derived{BaseValue := 10, DerivedValue := 20}
Instance := MakeDerived(10, 20)
```

For classes with mutable fields, initialization sets starting values that can change during the object's lifetime. Immutable fields must be initialized during construction and cannot be modified afterward. This distinction makes the construction phase critical for establishing invariants that will hold throughout the object's existence.

12.2 Shadowing and Qualification

Verse has strict rules about name shadowing to prevent ambiguity and maintain code clarity. Understanding these rules and the qualification syntax is essential for working with inheritance hierarchies, multiple interfaces, and nested modules.

In most contexts, you **cannot redefine names** that already exist in an enclosing scope. This applies to functions, variables, classes, interfaces, and modules:

```
# ERROR 3532: Function at module level shadows class method
# F(X:int):int = X + 1
# c := class:
#     F(X:int):int = X + 2  # ERROR - shadows outer F
```

This prohibition extends across various contexts:

```
# ERROR: Cannot shadow classes
something := class {}

m := module:
    something := class {}  # ERROR

# ERROR: Cannot shadow variables
Value:int = 1

```

```
m := module:
     Value:int = 2        # ERROR

# ERROR: Cannot shadow data members
c := class { A:int }

A():void = {}             # ERROR - order doesn't matter

# ERROR: Module and function cannot share name

id():void = {}
id := module {}           # ERROR
```

The shadowing prohibition exists **regardless of definition order** - it doesn't matter whether the outer name is defined before or after the inner scope.

To define methods with the same name in different contexts, use **qualified names** with the syntax `(ClassName:)MethodName`:

```
# Class with qualified method of same name
# c := class:
#    (c:)F(X:int):int = X + 2

# Module-level function
F(X:int):int = X + 1

# Call the module-level function
F(10)  # Returns 11

# Call the class method
c{}.F(10)  # Returns 12

# Explicit qualification (optional here)
c{}.(c:)F(10)  # Returns 12
```

The `(c:)` qualifier indicates this `F` is defined specifically in the `c` class context, distinguishing it from the module-level `F`. This allows the same name to coexist without shadowing errors.

12.2.1 Methods with Same Name

Using qualifiers, you can define *new methods* with the same name as inherited methods, creating multiple distinct methods in the same class:

```
c := class<abstract> { F(X:int):int }

d := class(c):
    F<override>(X:int):int = X + 1

e := class(d):
    (e:)F(X:int):int = X + 2 # NEW method with same name, not an override

# e now contains BOTH methods:
#    - (d:)F inherited from d
#    - (e:)F newly defined in e
```

Using the above:

```
E := e{}
E.(c:)F(10)  # Returns 11 (inherited from d's override)
E.(e:)F(10)  # Returns 12 (new method in e)
```

Key distinction:

- `F<override>` without qualifier: Overrides the inherited `F`
- `(e:)F` without `<override>`: Defines a **new** `F` specific to `e`

This allows a class to have multiple methods with the same name, differentiated by their qualifiers, each serving different purposes in the class hierarchy.

12.2.2 `(super:)` Qualified

The `(super:)` qualifier works with qualified method names to call the parent class's implementation:

```
i := interface { F(X:int):int }

ci := class(i):
    (i:)F<override>(X:int):int = X + 1
    (ci:)F(X:int):int = X + 2

dci := class(ci):
    # Override both inherited methods, calling super implementations
    (i:)F<override>(X:int):int = 100 + (super:)F(X)
    (ci:)F<override>(X:int):int = 200 + (super:)F(X)
```

And a use case:

```
DCI := dci{}
DCI.(i:)F(10)  # Returns 111 (100 + ci's 11)
DCI.(ci:)F(10)  # Returns 212 (200 + ci's 12)
```

`(super:)F(X)` within the qualified method calls the parent class's implementation of that same qualified method. This enables you to extend behavior for multiple method variants independently.

12.2.3 Interface Collisions

When implementing multiple interfaces with methods of the same name, qualifiers disambiguate which interface's method you're implementing:

```
i := interface:
    B(X:int):int

j := interface:
    B(X:int):int

collision := class(i, j):
    # Implement both B methods separately
    (i:)B<override>(X:int):int = 20 + X
    (j:)B<override>(X:int):int = 30 + X
```

And a use case:

```
Obj := collision{}
Obj.(i:)B(1)  # Returns 21
Obj.(j:)B(1)  # Returns 31
```

Without qualifiers, the compiler cannot determine which interface's method you're implementing, resulting in an error. The qualification makes your intent explicit.

Complex interface hierarchies:

```
i := interface:
    C(X:int):int

j := interface(i):
    A(X:int):int

k := interface(i):
    B(X:int):int
    (k:)C(X:int):int  # k redefines C

multi := class(j, k):
```

```
    A<override>(X:int):int = 10 + X
    B<override>(X:int):int = 20 + X
    # Must implement C from both inheritance paths
    (i:)C<override>(X:int):int = 30 + X
    (k:)C<override>(X:int):int = 40 + X
```

A use case:

```
Obj := multi{}
Obj.(i:)C(1)  # Returns 31
Obj.(k:)C(1)  # Returns 41
```

When an interface redefines a method from a parent interface using qualification `(k:)C`, implementing classes must provide separate implementations for both variants.

12.2.4 Nested Module Qualification

Modules can be nested, and deeply qualified names reference members through the entire hierarchy:

```
top := module:
    (top:)m<public> := module:
        (top.m:)Value<public>:int = 1
        (top.m:)F<public>(X:int):int = X + 10

        (top.m:)m<public> := module:
            (top.m.m:)Value<public>:int = 3
            (top.m.m:)F<public>(X:int):int = X + 100
```

And a use case:

```
# using { top.m }
# using { top.m.m }

# Access with full qualification
(top.m:)F(0)          # Returns 10
(top.m.m:)F(0)        # Returns 100

# Access via path
top.m.F(1)            # Returns 11
top.m.m.F(1)          # Returns 101
```

Nested modules can have the same simple name (e.g., both `m`) when qualified with their full path, allowing hierarchical organization without naming conflicts.

12.2.5 Restrictions

Qualifiers can only be used in appropriate contexts. You cannot use class qualifiers for local variables:

```
C := class:
    f():void =
        (C:)X:int = 0  # ERROR - wrong context
```

Certain qualifiers are not supported. Function qualifiers for local variables are not allowed:

```
C := class:
    f():void =
        (C.f:)X:int = 0  # ERROR - unsupported pattern
```

Similarly, using module function paths as qualifiers is not supported:

```
M := module:
    f():void =
        (M.f:)X:int = 0  # ERROR
```

Local variables cannot shadow class members:

```
A := class:
    I:int
    F(X:int):void =
        I:int = 5  # ERROR - shadows member I
```

Currently, there is no `(local:)` qualifier to disambiguate, so this pattern is not supported. You must use different names for local variables and members.

12.3 Parametric Classes

Parametric classes, also known as generic classes, allow you to define classes that work with any type. Rather than writing separate container classes for integers, strings, players, and every other type, you write one parametric class that accepts a type parameter.

A parametric class takes one or more type parameters in its definition:

```
# Simple container that holds a single value
container(t:type) := class:
    Value:t
```

Here are examples of instantiating this parametric class with different types:

```
# Can be instantiated with any type
IntContainer := container(int){Value := 42}
```

```
StringContainer := container(string){Value := "hello"}
PlayerContainer := container(player){Value := player{Name := "Hero", Health := 100}}
```

The syntax `container(t:type)` defines a class that is parameterized by type `t`. Within the class definition, `t` can be used anywhere a concrete type would appear—in field declarations, method signatures, or return types.

Multiple type parameters:

Classes can accept multiple type parameters:

```
pair(t:type, u:type) := class:
    First:t
    Second:u
```

Here are examples of using the parametric pair class:

```
# Different types for each parameter
Coordinate := pair(int, int){First := 10, Second := 20}
NamedValue := pair(string, float){First := "score", Second := 99.5}
```

Type parameters in methods:

Type parameters are available throughout the class, including in methods:

TODO

```
optional_container(t:type) := class:
    var MaybeValue:?t = false

    Set(Value:t):void =
        set MaybeValue = option{Value}

    Get()<decides>:t =
        MaybeValue?

    Clear():void =
        set MaybeValue = false
```

Methods automatically know about the type parameter from the class definition—you don't redeclare it in method signatures.

12.3.1 Instantiation and Identity

When you instantiate a parametric class with specific type arguments, Verse creates a concrete type. Critically, **multiple instantiations with the same type arguments produce the same type**:

```
container(t:type) := class:
    Value:t

# These are the same type
Type1 := container(int)
Type2 := container(int)
Type3 := container(int)

# All three are equal - they're the same type
```

This type identity is guaranteed across the program:

```
# Create instances
C1 := container(int){Value := 1}
C2 := container(int){Value := 2}

# Both have the same type: container(int)
# Type checking treats them identically
```

The instantiation process is **deterministic and memoized**. The first time you write `container(int)`, Verse generates a concrete type. Every subsequent use of `container(int)` refers to that same type, not a new copy.

This matters for:

- **Type compatibility**: Two values of `container(int)` can be used interchangeably
- **Memory efficiency**: Not creating duplicate type definitions
- **Semantic correctness**: Same type arguments always mean the same type

While the same type arguments always produce the same type, different type arguments produce distinct, incompatible types:

```
container(t:type) := class:
    Value:t
```

Here's an example showing that different instantiations create distinct types:

```
IntContainer := container(int){Value := 42}
StringContainer := container(string){Value := "text"}

# These are different types and cannot be mixed
# IntContainer = StringContainer  # Type error!
```

`container(int)` and `container(string)` are completely different types, with no subtype relationship. They happen to share the same structure (both defined from `container`), but that doesn't make them compatible.

While different instantiations of a parametric class are distinct types, Verse allows certain instantiations to be used in place of others based on **variance**. Variance determines when `parametric_class(subtype)` can be used where `parametric_class(supertype)` is expected (or vice versa).

The variance of a parametric type depends on how the type parameter is used within the class definition:

Covariant

When a type parameter appears only in **return positions** (method return types, field types being read), the parametric class is **covariant** in that parameter (see Types for details on variance). This means instantiations follow the same subtyping direction as their type arguments:

```
# Base class hierarchy
entity := class:
    ID:int

player := class(entity):
    Name:string

# Covariant class - type parameter only in return position
producer(t:type) := class:
    Value:t

    Get():t = Value  # Returns t - covariant position

# Can use producer(player) where producer(entity) expected
ProcessProducer(P:producer(entity)):int = P.Get().ID
```

Here's an example demonstrating covariance:

```
# Covariance allows subtype → supertype
PlayerProducer:producer(player) = producer(player){Value := player{ID := 1, Name := "Alice"}}
EntityProducer:producer(entity) = PlayerProducer  # Valid!

Result := ProcessProducer(PlayerProducer)  # Works!
```

Why this is safe: If you expect to get an `entity` from a producer, receiving a `player` (which is a subtype of `entity`) is always valid—a `player` has all the properties of an `entity`.

Direction: `producer(player)` → `producer(entity)` (follows subtype direction)

Contravariant

When a type parameter appears only in **parameter positions** (method parameters being consumed), the parametric class is **contravariant** in that parameter (see Types for details on variance). This means instantiations follow the **opposite** subtyping direction:

```
entity := class:
    ID:int

player := class(entity):
    Name:string

# Contravariant class - type parameter only in parameter position
consumer(t:type) := class:
    Process(Item:t):void = {}  # Accepts t - contravariant position
```

And a use case:

```
# Contravariance allows supertype → subtype
EntityConsumer:consumer(entity) = consumer(entity){}
PlayerConsumer:consumer(player) = EntityConsumer  # Valid!

# Can use consumer(entity) where consumer(player) expected
ProcessPlayers(C:consumer(player)):void =
    C.Process(player{ID := 1, Name := "Bob"})

ProcessPlayers(EntityConsumer)                    # Works!
```

Why this is safe: If you have a function that accepts any `entity`, it can certainly handle the more specific `player` type. A `consumer(entity)` can consume anything a `consumer(player)` can consume, plus more.

Direction: `consumer(entity)` → `consumer(player)` (opposite of subtype direction)

Invariant

When a type parameter appears in **both parameter and return positions**, the parametric class is **invariant** in that parameter. No subtyping relationship exists between different instantiations:

```
entity := class:
    ID:int

player := class(entity):
```

```
    Name:string

# Invariant class - type parameter in both positions
transformer(t:type) := class:
    Transform(Input:t):t = Input  # Both parameter and return
```

Here's an example showing that no variance exists between different instantiations:

```
# No variance - cannot convert in either direction
EntityTransformer:transformer(entity) = transformer(entity){}
PlayerTransformer:transformer(player) = transformer(player){}

# Invalid: Cannot use one where the other is expected
# X:transformer(entity) = PlayerTransformer  # ERROR 3509
# Y:transformer(player) = EntityTransformer  # ERROR 3509
```

Why this is necessary: If a `transformer(player)` could be used as a `transformer(entity)`, you could pass any `entity` to its `Transform` method, which expects specifically a `player`. This would be unsafe.

Direction: No conversion allowed in either direction

Bivariant

When a type parameter is not used in any method signatures (only in private implementation details or not at all), the parametric class is **bivariant**. Any instantiation can be converted to any other:

```
entity := class:
    ID:int

player := class(entity):
    Name:string

# Bivariant class - type parameter not used in public interface
container(t:type) := class:
    DoSomething():void = {}  # Doesn't use t at all
```

Here's an example showing that bivariant classes allow conversion in both directions:

```
# Bivariant allows conversion in both directions
EntityContainer:container(entity) = container(entity){}
PlayerContainer:container(player) = container(player){}

# Both directions work
```

```
X:container(entity) = PlayerContainer  # Valid
Y:container(player) = EntityContainer  # Also valid
```

Why this works: Since the type parameter doesn't affect the observable behavior, the instantiations are interchangeable.

Common Pitfalls

Attempting invalid conversions:

TODO broken...

```
# Invariant parameter - neither direction works
refer(t:type) := class:
    var Value:t
    Get():t = Value
    Set(V:t):void = set Value = V

PlayerRef:refer(player) = refer(player){Value := player{ID := 1, Name := "Test"}}

# Invalid: refer is invariant
# EntityRef:refer(entity) = PlayerRef  # ERROR
```

Confusing variance direction:

```
# Common mistake: thinking contravariance works like covariance
consumer(t:type) := class:
    Accept(Item:t):void = {}

EntityConsumer := consumer(entity){}

# Invalid: Wrong direction for contravariance
# PlayerConsumer:consumer(player) = consumer(entity){}  # ERROR

# Valid: Contravariance goes opposite direction
PlayerConsumer:consumer(player) = EntityConsumer  # Correct!
```

12.3.2 Parameter Constraints

You can constrain type parameters to require certain properties:

```
# Only comparable types allowed
sorted_list(t:type where t:subtype(comparable)) := class:
    var Items:[]t = array{}

    Add(Item:t):void =
```

```
        # Can compare because t is comparable
        set Items = InsertSorted(Items, Item)

    Contains(Item:t):logic =
        for (Element : Items):
            if (Element = Item):
                return true
        false

# Valid: int is comparable
IntList := sorted_list(int){}

# Invalid: regular classes aren't comparable by default
# PlayerList := sorted_list(player){}  # Error if player isn't comparable
```

The `where` clause specifies requirements on the type parameter. Common constraints include:

- `t:subtype(comparable)` - requires equality comparison
- `t:subtype(SomeClass)` - requires inheriting from a specific class
- `t:type` - any type (the default if no constraint specified)

12.3.3 Restrictions

Parametric classes have certain limitations:

Cannot be `<castable>`:

Parametric classes cannot use the `<castable>` specifier because runtime type checks require knowing the concrete type:

```
# Invalid: parametric classes cannot be castable
container(t:type) := class<castable>:  # Error!
    Value:t
```

However, specific instantiations can be used where `<castable>` types are needed:

```
component := class<castable>{}

container(t:type) := class:
    Value:t

# Error not supported
ProcessComponent(Comp:component):void =
    if (Wrapped := container(component)[Comp]):
        # Wrapped is container(component)
```

Cannot cast between different parametric instantiations:

Even when instantiations are fixed (non-parametric), you cannot use cast syntax to convert between different instantiations of the same parametric class or interface. This restriction is enforced at compile time:

```
container(t:type) := class:
    Value:t
```

Here's an example showing that you cannot cast between different instantiations:

```
X := container(int){Value := 42}

# Invalid: Cannot cast between different instantiations
# if (Y := container(float)[X]):     # ERROR 3502
#     # This will not compile
# if (Z := container([]int)[X]):     # ERROR 3502
#     # This also will not compile
```

Different instantiations like `container(int)` and `container(float)` are completely distinct types with no subtype relationship, so cast expressions between them are disallowed. The compiler rejects these casts even though both are concrete types.

This restriction extends to parametric class hierarchies:

```
base := class:
    Property:int

parametric_child(t:type) := class(base):
    GetProperty():int = Property
```

Here's an example showing that you cannot cast between different instantiations in parametric class hierarchies:

```
# Cannot cast between different instantiations of parametric_child
Foo:base = parametric_child(float){Property := 42}

# Invalid: Different type parameters prevent casting
# if (FooChild := parametric_child(int)[Foo]):  # ERROR 3502
#     # Cannot cast parametric_child(float) to parametric_child(int)
```

Even though both `parametric_child(int)` and `parametric_child(float)` inherit from `base`, you cannot cast between them because they are different instantiations of a parametric type.

Parametric interfaces also cannot be used in casts:

Cast expressions involving parametric interfaces with type parameters are disallowed:

```
parametric_interface(t:type) := interface:
    Foo():t

child := class{}

impl := class(child, parametric_interface(float)):
    Foo<override>():float = 42.42
```

Here's an example showing that you cannot cast to parametric interfaces:

```
# Invalid: Cannot cast to parametric interface with type parameter
X:child := impl{}
# if (X_Casted := parametric_interface(float)[X]):  # ERROR 3502
#     # Parametric interface casts not allowed
```

However, specialized (non-parametric) interfaces derived from parametric interfaces can be used in casts:

```
parametric_interface(t:type) := interface:
    Foo():t

# Specialized interface fixes the type parameter
specialized_interface := interface(parametric_interface(float)){}

impl := class(specialized_interface):
    Foo<override>():float = 42.42
```

Here's an example showing that specialized interfaces work with casts:

```
# Valid: specialized_interface is no longer parametric
X := impl{}
if (X_Casted := specialized_interface[X]):
    X_Casted.Foo()  # Works!
```

Valid casting scenarios:

While casts between different parametric instantiations fail, the following patterns work:

1. **Non-parametric class hierarchies** support normal casting:

```
base := class<castable>:
    ID:int

child := class(base):
    Name:string
```

Here's an example of normal class hierarchy casting:

```
B:base = child{ID := 1, Name := "Test"}
if (C := child[B]):
    # Valid: Normal class hierarchy cast
    Print(C.Name)
```

2. **Fixed parametric instantiations** where the type parameter is locked in the subclass:

```
parametric_base(t:type) := class:
    Property:t

# Child fixes the type parameter to int
int_child := class(parametric_base(int)):
    GetProperty():int = Property
```

Here's an example of casting with fixed parametric instantiation:

```
Foo:parametric_base(int) = int_child{Property := 42}
if (FooChild := int_child[Foo]):
    # Valid: Type parameter is fixed to int in both
    FooChild.Property = 42
```

Cannot be `<persistable>` directly:

While you can define parametric classes, making them persistable requires special consideration for how the type parameter is serialized. Specific instantiations with persistable types may work depending on the implementation.

12.3.4 Recursive Parametric Types

Parametric classes can reference themselves in their field types, enabling recursive generic data structures like linked lists, trees, and graphs. However, Verse imposes specific restrictions on how recursion can occur.

The most common form of recursive parametric type is when a class references itself with **the same type parameter**:

```
# Linked list node
list_node(t:type) := class:
    Value:t
    Next:?list_node(t)  # Same type parameter 't'

# Helper to create lists
Cons(Head:t, Tail:?list_node(t) where t:type):list_node(t) =
    list_node(t){Value := Head, Next := Tail}
```

```
9
10  # Sum a linked list
11  SumList(List:?list_node(int)):int =
12      if (Head := List?):
13          Head.Value + SumList(Head.Next)
14      else:
15          0
```

Here's an example of using the linked list:

```
1   # Usage
2   IntList := list_node(int){
3       Value := 1
4       Next := option{list_node(int){
5           Value := 2
6           Next := false
7       }}
8   }
```

Binary trees work similarly:

TODO Broken

```
1   tree_node(t:type) := class:
2       Value:t
3       var Left:?tree_node(t) = false   # Same parameter
4       var Right:?tree_node(t) = false  # Same parameter
5
6   # Create a tree
7   Root := tree_node(int){
8       Value := 5
9       Left := option{tree_node(int){Value := 3}}
10      Right := option{tree_node(int){Value := 7}}
11  }
```

Why this works: Each instantiation creates a complete, consistent type. `list_node(int)` always contains `int` values and references other `list_node(int)` nodes. The type system can verify this recursion is well-formed.

Disallowed: Direct Type Alias Recursion

You cannot define a parametric type that directly aliases to a structural type containing itself:

```
1   # Invalid: Direct array recursion
2   # t(u:type) := []t(u)  # ERROR 3502
3
```

```
# Invalid: Direct map recursion
# t(u:type) := [int]t(u)  # ERROR 3502

# Invalid: Direct optional recursion
# t(u:type) := ?t(u)  # ERROR 3502

# Invalid: Direct function recursion
# t(u:type) := u->t(u)  # ERROR 3502
# t(u:type) := t(u)->u  # ERROR 3502
```

These fail because they create infinite type expansion—the compiler cannot determine the actual structure of the type.

Valid alternative: Wrap in a class:

```
# Valid: Indirect recursion through class
nested_list(t:type) := class:
    Items:[]nested_list(t)  # OK - wrapped in class
```

Here's an example of using nested_list:

```
Tree := nested_list(int){
    Items := array{
        nested_list(int){Items := array{}},
        nested_list(int){Items := array{}}
    }
}
```

Disallowed: Polymorphic Recursion

Polymorphic recursion occurs when a parametric type references itself with a **different type argument**:

```
# Invalid: Type parameter changes
# my_type(t:type) := class:
#     Next:my_type(?t)  # ERROR 3509 - ?t is different from t

# Invalid: Alternating type parameters
# bi_list(t:type, u:type) := class:
#     Value:t
#     Next:?bi_list(u, t)  # ERROR 3509 - parameters swapped
```

Why this is disallowed: Polymorphic recursion makes type inference undecidable and can create infinitely complex types. When you instantiate `my_type(int)`, it would need `my_type(?int)`, which needs `my_type(??int)`, and so on forever.

Current limitation: While polymorphic recursion is theoretically sound in some type systems, Verse currently does not support it to keep type checking tractable.

Disallowed: Mutual Recursion

Mutual recursion between multiple parametric types is not supported:

```
# Invalid: Mutual recursion
# t1(t:type) := class:
#     Next:?t2(t)  # References t2
#
# t2(t:type) := class:
#     Next:?t1(t)  # References t1
#
# # ERROR 3509, 3509
```

Why this is disallowed: Similar to polymorphic recursion, mutual recursion complicates type inference and can create circular dependencies that are difficult for the compiler to resolve.

Workaround: Combine into a single type:

```
# Valid: Single type with multiple cases
node_type := enum:
    TypeA
    TypeB

combined_node(t:type) := class:
    Type:node_type
    Value:t
    Next:?combined_node(t)
```

Disallowed: Inheritance Recursion

You cannot inherit from a type variable or create recursive inheritance through parametric types:

```
# Invalid: Inheriting from parametric self
# t(u:type) := class(t(u)){}  # ERROR 3590

# Invalid: Inheriting from type variable
# inherits_from_variable(t:type) := class(t){}  # ERROR 3590
```

Why this is disallowed: Inheritance requires knowing the parent's structure,but with parametric recursion, this structure would be self-referential before being defined.

12.3.5 Parametric Interfaces

While parametric classes get most of the attention, interfaces can also be parametric, enabling abstract contracts that work with any type:

```
# Generic equality interface
equivalence(t:type, u:type) := interface:
    Equal(Left:t, Right:u)<transacts><decides>:t

# Generic collection interface
collection_ifc(t:type) := interface:
    Add(Item:t)<transacts>:void
    Remove(Item:t)<transacts><decides>:void
    Has(Item:t)<reads>:logic
```

Classes implement parametric interfaces by providing concrete types for the parameters:

```
equivalence(t:type, u:type) := interface:
    Equal(Left:t, Right:u)<transacts><decides>:t

# Implement with specific types
int_equivalence := class(equivalence(int, comparable)):
    Equal<override>(Left:int, Right:comparable)<transacts><decides>:int =
        Left = Right

# Or with type parameters matching the class
comparable_equivalence(t:subtype(comparable)) := class(equivalence(t, comparable)):
    Equal<override>(Left:t, Right:comparable)<transacts><decides>:t =
        Left = Right
```

Here's an example of using the parametric interface:

```
# Usage
Eq := comparable_equivalence(int){}
Eq.Equal[5, 5]  # Succeeds
```

Parametric interfaces follow the same variance rules as parametric classes:

```
entity := class:
    ID:int

player := class(entity):
    Name:string

# Covariant interface - returns t
producer_interface(t:type) := interface:
```

```
    Produce():t

player_producer := class(producer_interface(player)):
    Produce<override>():player = player{ID := 1, Name := "Test"}
```

Here's an example of covariant subtyping:

```
# Covariant subtyping works
EntityProducer:producer_interface(entity) = player_producer{}
```

You can create specialized (non-parametric) interfaces from parametric ones:

```
generic_handler(t:type) := interface:
    Handle(Item:t):void

# Specialize to a concrete type
int_handler := interface(generic_handler(int)):
    # Inherits Handle(Item:int):void
    # Can add more methods here

int_processor := class(int_handler):
    Handle<override>(Item:int):void =
        Print("Handling: {Item}")
```

Here's an example of using specialized interfaces in casts:

```
# Can use in casts now (specialized interfaces are non-parametric)
Base := int_processor{}
if (Handler := int_handler[Base]):
    Handler.Handle(42)
```

Multiple Type Parameters

Interfaces can have multiple type parameters with independent variance:

```
converter_interface(input:type, output:type) := interface:
    Convert(In:input):output
    # input is contravariant, output is covariant

entity := class:
    ID:int

player := class(entity):
    Name:string

# Implement with specific types
```

```
player_to_entity := class(converter_interface(player, entity)):
    Convert<override>(In:player):entity = entity{ID := In.ID}
```

Is used here:

```
# Variance allows flexible usage
C:converter_interface(player, entity) = player_to_entity{}
```

12.3.6 Advanced Parametric Types

First-Class Parametrics

Parametric type definitions can be used as first-class values, allowing dynamic type application:

```
# Parametric class
container(t:type) := class:
    Value:t

# Store parametric type as value
TypeConstructor := container
```

And a use case:

```
# Apply type argument dynamically
IntContainer := TypeConstructor(int)

# Construct instance
Instance := IntContainer{Value := 42}
Instance.Value = 42  # Success
```

This enables powerful patterns for generic factories and type-driven programming:

And a use:

```
X := CreateContainer(container1, 42)  # container1(int)
Y := CreateContainer(container2, "hello")  # container2(string)
```

Effects

Parametric types can have effect specifiers that apply to all instantiations:

```
# Parametric class with effects
async_container(t:type) := class<computes>:
    Property:t

# All instantiations inherit the effect
X:async_container(int) = async_container(int){Property := 1}  # <computes> effect
```

```
7
8  # Multiple effects
9  transactional_container(t:type) := class<transacts>:
10     Property:t
11
12 # Constructor inherits effects
13 # Y:transactional_container(int) = transactional_container(int){Property := 2}
```

Allowed effects:

- `<computes>` - Allows non-terminating computation
- `<transacts>` - Participates in transactions
- `<reads>` - Reads mutable state
- `<writes>` - Writes mutable state
- `<allocates>` - Allocates resources

Not allowed:

- `<decides>` - Can fail
- `<suspends>` - Can suspend execution
- `<converges>` - Would conflict with parametric instantiation

Effect propagation:

```
1  # Effect on parametric type propagates to constructor
2  my_type(t:type) := class<computes>:
3      Property:t
4
5  # This requires <computes> in the context
6  CreateInstance()<computes>:my_type(int) =
7      my_type(int){Property := 1}
```

The effect becomes part of the type's contract—all code constructing or working with instances must account for these effects.

Aliases

You can create type aliases that simplify complex parametric type expressions:

```
1  # Alias for map type
2  string_map(t:type) := [string]t
3
4  # Use the alias
5  PlayerScores:string_map(int) = map{
6      "Alice" => 100,
```

```
    "Bob" => 95
}

# Alias for optional array
optional_array(t:type) := []?t

# Simplifies type signatures
FilterValid(Items:optional_array(int)):[]int =
    for (Item : Items; Value := Item?):
        Value
```

Composing parametric aliases:

```
# Nested parametric aliases
map_alias(k:type where k:subtype(comparable), v:type) := [k]v   # k must be comparable (int,
array_alias(t:type) := []t

# Compose them
nested(t:type) := array_alias(map_alias(string, t))

# Usage: [](string]t)
Data:nested(int) = array{
    map{"a" => 1, "b" => 2},
    map{"c" => 3}
}
```

Structural type aliases:

```
# Function type aliases
transformer(input:type, output:type) := input -> output
predicate(t:type) := t -> logic

# Tuple type aliases
pair(t:type, u:type) := tuple(t, u)
triple(t:type) := tuple(t, t, t)

# Use in signatures
ApplyTransform(T:transformer(int, string), Value:int):string =
    T(Value)

CheckCondition(P:predicate(int), Value:int):logic =
    P(Value)
```

Type aliases improve readability and maintainability for complex generic types.

Advanced Type Constraints

Beyond basic `subtype` constraints, parametric types support specialized constraints:

Subtype constraints:

```
# Constrain to subtype of a class
bounded_container(t:subtype(entity)) := class:
    Value:t

    GetID():int = Value.ID  # Can access entity members

# Valid: player is subtype of entity
# PlayerContainer := bounded_container(player){}

# Invalid: int is not subtype of entity
# IntContainer := bounded_container(int){}  # Type error
```

Castable subtype constraints:

```
# Requires castable subtype
dynamic_handler(t:castable_subtype(component)) := class:
    Handle(Item:component):void =
        if (Typed := t[Item]):
            # Typed has the specific subtype
            ProcessTyped(Typed)
```

Multiple constraints:

TODO BROKEN

```
# Combine multiple requirements
sorted_unique(t:type where t:subtype(comparable)) := class<unique>:
    var Items:[]t = array{}

    Add(Item:t):void =
        # Can use comparison because t:subtype(comparable)
        if (not Contains(Item)):
            set Items = Sort(Items + array{Item})

    Contains(Item:t):logic =
        for (Element : Items):
            if (Element = Item):
                return true
        false
```

Constraint propagation:

```
# Constraints propagate through function calls
wrapper(t:subtype(comparable)) := class:
    Data:t

Process(W:wrapper(t) where t:subtype(comparable))<computes><decides>:void =
    # Compiler knows t is comparable here
    W.Data = W.Data
```

When defining parametric functions that work with parametric types, the constraints must be compatible:

```
base_class := class:
    ID:int

constrained(t:subtype(base_class)) := class:
    Data:t

# Valid: Constraint matches
UseConstrained(C:constrained(t) where t:subtype(base_class)):int =
    C.Data.ID

# Invalid: Missing or incompatible constraint
# UseConstrained(C:constrained(t) where t:type):int =  # ERROR 3509
#     C.Data.ID
```

12.3.7 Access Specifiers

Classes support fine-grained control over member visibility through access specifiers:

```
game_state := class:
    Score<public> : int = 0                    # Anyone can read
    var Lives<private> : int = 3               # Only this class can access
    var Shield<protected> : float = 100.0      # This class and subclasses
    DebugInfo<internal> : string = ""          # Same module only

    # Public method - anyone can call
    GetLives<public>() : int = Lives

    # Protected method - subclasses can override
    OnLifeLost<protected>() : void = {}

    # Private helper - only this class
    ValidateState<private>() : void = {}
```

Access specifiers apply to both fields and methods, controlling who can read fields and call methods. The default visibility is `internal`, restricting access to the same module. This encapsulation is crucial for maintaining class invariants and hiding implementation details.

12.3.8 Concrete

The `<concrete>` specifier enforces that all fields have default values, allowing construction with an empty archetype:

```
config := class<concrete>:
    MaxPlayers : int = 8
    TimeLimit : float = 300.0
    FriendlyFire : logic = false

# Can construct with empty archetype
DefaultConfig := config{}
```

This is particularly useful for configuration classes where reasonable defaults exist for all values.

A concrete class `C` can be constructed by writing `C{}`, that is to say with the empty archetype.

A concrete class may have non-concrete subclasses.

12.3.9 Unique

The `<unique>` specifier creates classes and interfaces with reference semantics where each instance has a distinct identity. When a class or interface is marked as `<unique>`, instances become comparable using the equality operators (= and <>), with equality based on object identity rather than field values.

Classes marked with `<unique>` compare by identity, not by value:

```
entity := class<unique>:
   Name : string
   Position : vector3

E1 := entity{Name := "Guard", Position := vector3{X := 0.0, Y := 0.0, Z := 0.0}}
E2 := entity{Name := "Guard", Position := vector3{X := 0.0, Y := 0.0, Z := 0.0}}
E3 := E1

E1 = E2  # Fails - different instances despite identical field values
E1 = E3  # Succeeds - same instance
```

Without `<unique>`, class instances cannot be compared for equality at all—the language prevents meaningless comparisons. With `<unique>`, you gain the ability to use instances as map keys, store them in sets, and perform identity checks, essential for tracking specific objects throughout their lifetime.

Interfaces

Interfaces can also be marked with `<unique>`, which makes all instances of classes implementing that interface comparable by identity:

```
component := interface<unique>:
    Update():void
    Render():void

physics_component := class(component):
    Update<override>():void = {}
    Render<override>():void = {}
```

And a use case:

```
# Instances are comparable because component is unique
P1 := physics_component{}
P2 := physics_component{}

P1 <> P2  # true - different instances
P1 = P1   # true - same instance
```

The `<unique>` property propagates through interface inheritance. If a parent interface is marked `<unique>`, all child interfaces and classes implementing those interfaces automatically become comparable:

```
base_component := interface<unique>:
    Update():void

# Child interface inherits <unique> from parent
advanced_component := interface(base_component):
    AdvancedUpdate():void

# Classes implementing any interface in the hierarchy become comparable
player_component := class(advanced_component):
    Update<override>():void = {}
    AdvancedUpdate<override>():void = {}
```

And a use case:

```
C1 := player_component{}
C2 := player_component{}
C1 <> C2  # true - comparable due to base_component being unique
```

When a class implements multiple interfaces, comparability is determined by whether ANY of the inherited interfaces is `<unique>`:

```
updateable := interface:  # Not unique
    Update():void

renderable := interface<unique>:  # Unique
    Render():void

game_object := class(updateable, renderable):
    Update<override>():void = {}
    Render<override>():void = {}
```

And a use case:

```
# game_object is comparable because renderable is unique
G1 := game_object{}
G2 := game_object{}
G1 <> G2  # true - comparable due to renderable interface
```

Even if most interfaces are non-unique, a single `<unique>` interface in the hierarchy makes the entire class comparable.

Unique in Default Values

When a `<unique>` class appears in a field's default value, each containing object receives its own distinct instance. This guarantee applies even when the unique class is nested within complex parametric types:

```
token := class<unique>:
    ID:int = 0

container := class:
    MyToken:token = token{}
```

And a use case:

```
C1 := container{}
C2 := container{}
C1.MyToken <> C2.MyToken  # true - each container has its own unique token
```

This behavior extends to `<unique>` instances within arrays, optionals, tuples, and maps:

```
item := class<unique>{}

# Each class instantiation creates fresh unique instances in default values
with_array := class:
    Items:[]item = array{item{}}

with_optional := class:
    MaybeItem:?item = option{item{}}

with_map := class:
    ItemMap:[int]item = map{0 => item{}}
```

And a use case:

```
A := with_array{}
B := with_array{}
A.Items[0] <> B.Items[0]  # true - different unique instances

C := with_optional{}
D := with_optional{}
if (ItemC := C.MaybeItem?, ItemD := D.MaybeItem?):
    ItemC <> ItemD  # true - different unique instances
```

The same principle applies when parametric classes contain unique instances in their fields:

```
entity := class<unique>{}

registry(t:type) := class:
    DefaultEntity:entity = entity{}
    Data:t
```

And a use case:

```
R1 := registry(int){Data:=1}
R2 := registry(int){Data:=2}
R1.DefaultEntity <> R2.DefaultEntity  # true

R3 := registry(string){Data:="hi"}
R3.DefaultEntity <> R1.DefaultEntity  # true - even across different type parameters
```

This guarantee ensures that identity-based operations remain reliable. If you store objects in maps keyed by unique instances, or maintain sets of unique objects, each container genuinely owns distinct instances rather than sharing references. The

language prevents subtle bugs where multiple objects might unexpectedly share the same identity.

Overload Resolution

Types marked with `<unique>` are subtypes of the built-in `comparable` type. This can create overload ambiguity:

```
# Valid: non-unique interface doesn't conflict with comparable
regular_interface := interface:
    Method():void

Process(A:comparable, B:comparable):void = {}
Process(A:regular_interface, B:regular_interface):void = {}  # OK - no conflict

# Invalid: unique interface conflicts with comparable
unique_interface := interface<unique>:
    Method():void

Handle(A:comparable, B:comparable):void = {}
# Handle(A:unique_interface, B:unique_interface):void = {}  # ERROR - ambiguous!
```

Since `unique_interface` is a subtype of `comparable`, both overloads could match when called with `unique_interface` arguments, causing a compilation error. When designing overloaded functions, be aware that `<unique>` types participate in the `comparable` type hierarchy.

Use Cases

The `<unique>` specifier is ideal for:

Game Entities: Where each entity in the world must be distinguishable regardless of current state

```
#entity := class<unique>:
#    var Health:int = 100
#    var Position:vector3

# Can track specific entities in collections
var ActiveEntities:[entity]logic = map{}
```

Component Interfaces: Where you need identity-based equality for interface types

```
#component := interface<unique>:
#    Owner:entity

```

```
# Can use interface references as map keys
var ComponentRegistry:[component]string = map{}
```

Session Objects: Where identity matters more than current property values

```
#player_session := class<unique>:
#    PlayerID:string
#    var ConnectionTime:float

# Track specific sessions
var ActiveSessions:[player_session]connection_info = map{}
```

Resource Handles: Where you need to track specific instances rather than equivalent values

```
#texture_handle := class<unique>:
#    ResourceID:int
#    FilePath:string

# Manage resource lifecycle
var LoadedTextures:[texture_handle]gpu_resource = map{}
```

The `<unique>` specifier enables these patterns by providing identity-based equality semantics, making it possible to use instances as map keys, maintain sets of unique objects, and distinguish between different instances even when their data is identical.

12.3.10 Abstract

The `<abstract>` specifier marks classes that cannot be instantiated directly — they exist solely as base classes for inheritance. When you declare a class with `<abstract>`, you're creating a template that defines structure and behavior for subclasses to inherit and implement.

Abstract classes serve as architectural foundations in a type hierarchy. They define contracts through abstract methods that subclasses must implement, while potentially providing concrete methods and fields that subclasses inherit. This creates a powerful pattern for code reuse and polymorphic behavior.

```
vehicle := class<abstract>:
    Speed():float             # Abstract method
    MaxPassengers:int = 1

    # Concrete method all vehicles share
    CanTransport(Count:int)<decides>:void =
        Count <= MaxPassengers
```

```
8
9   car := class(vehicle):
10      Speed<override>():float = 60.0
11      MaxPassengers<override>:int = 4
12
13  bicycle := class(vehicle):
14      Speed<override>():float = 15.0
```

Abstract methods within abstract classes have no implementation — they're pure declarations that establish what subclasses must provide. An abstract method creates a contract: any non-abstract subclass must override all abstract methods or the code won't compile.

12.3.11 Castable

The `<castable>` specifier enables runtime type checking and safe downcasting for classes. When a class is marked with `<castable>`, you can use dynamic type tests and casts to determine if an object is an instance of that class or its subclasses at runtime.

Without `<castable>`, Verse's type system operates purely at compile time. The `<castable>` specifier adds runtime type information, allowing code to inspect and react to actual object types during execution. This bridges the gap between static type safety and dynamic polymorphism.

Verse provides two forms of type casting: **fallible casts** (which can fail at runtime) and **infallible casts** (which are verified at compile time).

Fallible casts use bracket syntax `Type[Value]` and return an optional result. These are runtime checks that succeed only if the value is actually an instance of the target type:

```
1  component := class<abstract><castable><allocates>:
2      Name:string
3
4  physics_component := class<allocates>(component):
5      Name<override>:string = "Physics"
6      Velocity:vector3
7
8  render_component := class<allocates>(component):
9      Name<override>:string = "Render"
10     Material:string
11
12 ProcessComponent(Comp:component):void =
13     # Attempt to cast to physics_component
```

```
    if (PhysicsComp := physics_component[Comp]):
        # Cast succeeded - PhysicsComp has type physics_component
        Print("Physics component with velocity: {PhysicsComp.Velocity}")
    else if (RenderComp := render_component[Comp]):
        # Cast succeeded - RenderComp has type render_component
        Print("Render component with material: {RenderComp.Material}")
    else:
        # Neither cast succeeded
        Print("Unknown component type")
```

The cast expression has the `<decides>` effect—it fails if the object is not an instance of the target type. This integrates naturally with Verse's failure handling:

```
GetPhysicsComponent(Comp:component)<computes><decides>:physics_component =
    # Returns physics_component or fails
    physics_component[Comp]

# Use with failure handling
if (Physics := GetPhysicsComponent[SomeComponent]):
    UpdatePhysics(Physics)
```

Infallible casts use parenthesis syntax `Type(Value)` and only work when the compiler can verify the cast is safe—that is, when the value type is a subtype of the target type:

```
base := class:
    ID:int

derived := class(base):
    Name:string

GetDerived():derived = derived{ID := 1, Name := "Test"}
```

Use case:

```
# Infallible upcast - derived is a subtype of base
BaseRef:base = base(GetDerived())  # Always safe
```

Attempting an infallible downcast (from supertype to subtype) is a compile error, as the compiler cannot guarantee safety:

```
DerivedRef := derived(BaseRef)  # ERROR: not a subtype relationship
```

Castable and Inheritance

The `<castable>` property is inherited by all subclasses. When you mark a class as `<castable>`, every class that inherits from it automatically becomes castable as well:

```
base := class<castable>:
    Value:int

child := class(base):
    # Automatically castable - inherits from castable base
    Name:string

grandchild := class(child):
    # Also automatically castable
    Extra:string

# Can cast through the hierarchy
ProcessBase(Instance:base):void =
    if (AsChild := child[Instance]):
        Print("It's a child: {AsChild.Name}")
    if (AsGrandchild := grandchild[Instance]):
        Print("It's a grandchild: {AsGrandchild.Extra}")
```

Important constraint: Parametric types cannot be `<castable>`. This prevents type erasure issues at runtime:

```
# Valid: non-parametric castable class
valid_castable := class<castable>:
    Data:int

# Invalid: parametric classes cannot be castable
# invalid_castable(t:type) := class<castable>:  # ERROR
#     Data:t
```

However, a non-parametric class can be `<castable>` even if it inherits from or contains parametric types:

```
container(t:type) := class:
    Value:t

# Valid: concrete instantiation of parametric type
int_container := class<castable>(container(int)):
    Extra:string
```

Using castable_subtype

The `castable_subtype` type constructor works with `<castable>` classes to enable type-safe filtered queries and dynamic type dispatch:

```
component<public> := class<abstract><unique><castable>:
    Parent<public>:entity

entity<public> := class<concrete><unique><transacts><castable>:
    FindDescendantEntities(entity_type:castable_subtype(entity)):generator(entity_type)
```

When you call `FindDescendantEntities(player)`, the function returns only entities that are actually player instances or subclasses thereof, verified at runtime through the castable mechanism. The type parameter ensures type safety—the returned values have the specific subtype you requested.

Permanence of Castable

Once a class is published with `<castable>`, this decision becomes permanent. You cannot add or remove the `<castable>` specifier after publication because doing so would break existing code that relies on runtime type checking. Code that performs casts would suddenly fail or behave incorrectly if the castable property changed.

This permanence is enforced through the versioning system—attempting to change the `<castable>` status of a published class will result in a compatibility error.

12.3.12 Final

The `<final>` specifier prevents inheritance, creating a terminal point in a class hierarchy. When you mark a class with `<final>`, no other class can inherit from it. For methods, `<final>` prevents overriding in subclasses, locking the implementation at that level of the hierarchy.

Classes marked with `<final>` serve as concrete implementations that cannot be extended. This is particularly important for persistable classes, which require `<final>` to ensure their structure remains stable for serialization:

```
player_profile := class<final><persistable>:
    Username:string = "Player"
    Level:int = 1
    Gold:int = 0

player_data := class<final><persistable>:
    Version:int = 1
    LastLogin:string = ""
    Statistics:player_stats = player_stats{}
```

The <final> requirement for persistable classes prevents schema evolution problems. If subclasses could extend persistable classes, the serialization system would face ambiguity about which fields to persist and how to handle polymorphic deserialization.

For methods, <final> locks behavior at a specific point in the inheritance chain:

```
  base_entity := class:
      GetName():string = "Entity"

  game_object := class(base_entity):
      GetName<override><final>():string = "GameObject"
      # Any subclass of game_object cannot override GetName
```

The related <final_super> specifier marks classes as terminal base classes — they can be inherited from but their subclasses cannot be further extended. <final_super_base> marks a class as the ultimate root of a restricted inheritance tree. Classes with this specifier can be inherited from, but their subclasses automatically become final — they cannot be further extended. This creates a two-level inheritance limit starting from the base:

```
component := class<abstract><unique><castable><final_super_base>:
      Parent:entity

  # Can inherit from component (first level)

physics_component := class<final_super>(component):
      Mass:float = 1.0

 # Cannot inherit from physics_component - it's implicitly final

 gravity_component := class(physics_component): # COMPILE ERROR
```

So, <final_super> marks a class that inherits from a <final_super_base> class, explicitly declaring it as the final inheritance point. While classes inheriting from <final_super_base> are implicitly final, using <final_super> makes this finality explicit and self-documenting.

```
# Explicitly marking as final_super (though implicitly final anyway)
name_component := class<final_super>(component):
      Name:string = ""

copter_camera_component := class<final_super>(copter_camera_component_director_version):
      # Terminal implementation
```

This pattern is particularly valuable in component architectures where you want a base component interface that various concrete components implement, but don't want those implementations to spawn their own inheritance subtrees. The base class defines the contract, immediate subclasses provide implementations, and inheritance stops there — clean, controlled, and predictable.

This design enforces architectural discipline, preventing the "inheritance explosion" that can occur when every class becomes a potential base for further specialization. By limiting inheritance depth, these specifiers promote composition over deep inheritance, leading to more maintainable and understandable code structures.

12.3.13 Persistable

The `<persistable>` specifier marks types that can be saved and restored across game sessions, enabling permanent storage of player progress, achievements, and game state. This specifier transforms ephemeral gameplay into lasting progression, creating the foundation for meaningful player investment.

Persistence works through module-scoped `weak_map(player, t)` variables, where `t` is any persistable type. These special maps automatically synchronize with backend storage — when players join, their data loads; when they leave or data changes, it saves. The system handles all serialization, network transfer, and storage management transparently.

```
player_inventory := class<final><persistable>:
    Gold:int = 0
    Items:[]string = array{}
    UnlockedAreas:[]string = array{}

# This variable automatically persists across sessions

SavedInventories : weak_map(player, player_inventory) = map{}
```

The `<persistable>` specifier enforces strict structural requirements to guarantee data integrity across versions. Classes must be `<final>` because inheritance would complicate serialization schemas. They cannot contain `var` fields, preserving immutability guarantees even in persistent storage. They cannot be `<unique>` since identity-based equality doesn't survive serialization. These constraints ensure that what you save today can be reliably loaded tomorrow, next month, or next year.

12.4 Interfaces

Interfaces define contracts that classes can implement, specifying both the data and behavior that implementing classes must provide. Unlike many traditional languages where interfaces only declare method signatures, Verse interfaces are

rich contracts that can include fields, default method implementations, and even custom accessor logic.

An interface can declare method signatures, provide default implementations, and define data members:

```
damageable := interface:
    # Abstract method - implementing classes must provide
    TakeDamage(Amount:int)<transacts>:void

    # Method with default implementation
    GetHealth()<computes>:int = 100

    # Data member - implementing classes inherit or must provide
    MaxHealth:int = 100

    IsAlive()<computes>:logic = logic{GetHealth() > 0}

healable := interface:
    Heal(Amount:int):void
    GetMaxHealth():int
```

Interfaces establish contracts that can be purely abstract (method signatures only), partially concrete (some default implementations), or fully implemented (complete behavior that classes inherit). Any class implementing an interface must provide implementations for abstract methods, but inherits concrete implementations and default field values.

12.4.1 Implementing Interfaces

Classes implement interfaces by inheriting from them and providing concrete implementations where required:

```
character := class(damageable, healable):
    var Health : int = 100
    MaxHealth : int = 100

    TakeDamage<override>(Amount:int)<transacts>:void =
        set Health = Max(0, Health - Amount)

    GetHealth<override>()<reads>:int = Health

    Heal<override>(Amount:int)<transacts>:void =
        set Health = Min(MaxHealth, Health + Amount)
```

A class can implement multiple interfaces, effectively achieving multiple inheritance of both behavior contracts and data specifications. This provides more flexibility than single class inheritance while maintaining type safety.

12.4.2 Interface Fields

Interfaces can declare data members that implementing classes must provide or inherit. These fields can be either immutable or mutable, and may include default values:

```
# Interface with various field types
entity_properties := interface:
    # Immutable field with default - classes inherit this value
    EntityID:int = 0

    # Mutable field with default
    var Health:float = 100.0

    # Field without default - classes must provide a value
    Name:string

    # Field that can be overridden
    MaxHealth:float = 100.0

player_entity := class(entity_properties):
    # Must provide Name (no default in interface)
    Name<override>:string = "Player"

    # Can override to change default
    MaxHealth<override>:float = 150.0

    # Inherits EntityID and Health with their defaults
```

When an interface field has a default value, implementing classes automatically inherit that default unless they override it. Fields without defaults must be provided either by the implementing class or through construction parameters.

12.4.3 Default Implementations

Interfaces can provide complete method implementations that implementing classes inherit automatically:

```
animated := interface:
    var CurrentFrame:int = 0
    TotalFrames:int = 10
```

```
4
5       # Concrete implementation provided by interface
6       NextFrame()<transacts><decides>:void =
7           set CurrentFrame = Mod[(CurrentFrame + 1),TotalFrames] or 0
8
9       # Can access interface fields
10      ProgressPercent()<reads><decides>:rational =
11          CurrentFrame / TotalFrames
12
13  sprite := class(animated):
14      TotalFrames<override>:int = 20
15      # Automatically inherits NextFrame and ProgressPercent implementations
```

Classes inherit these implementations without modification, allowing interfaces to provide reusable behavior. Implementing classes can override these methods if they need specialized behavior, but the interface provides a working default.

12.4.4 Overriding Members

Classes can override both fields and methods from interfaces to provide specialized implementations:

```
1   base_stats := interface:
2       BaseHealth:int = 100
3
4       CalculateFinalHealth():int = BaseHealth
5
6   warrior := class(base_stats):
7       # Override field with different default
8       BaseHealth<override>:int = 150
9
10      # Override method for specialized calculation
11      CalculateFinalHealth<override>():int =
12          BaseHealth * 2  # Warriors get double health
13
14  mage := class(base_stats):
15      BaseHealth<override>:int = 75
16
17      CalculateFinalHealth<override>():int =
18          BaseHealth + MagicBonus
19
20      MagicBonus:int = 25
```

Field overrides can provide different default values or specialize to subtypes. Method overrides replace the interface's implementation entirely. All overrides must maintain type compatibility—fields can only be overridden with subtypes, and method signatures must match exactly.

12.4.5 Multiple Interfaces with Sharing

When a class implements multiple interfaces that declare fields or methods with the same name, you must use qualified names to disambiguate:

```
magical := interface:
    Power:int = 50
    GetPowerLevel()<computes>:int = Power

physical := interface:
    Power:int = 75
    GetPowerLevel()<computes>:int = Power * 2

hybrid := class(magical, physical):
    UseHybridPowers():void =
       MagicPower := (magical:)Power         # Access magical's Power
       PhysicalPower := (physical:)Power     # Access physical's Power
       MagicLevel := (magical:)GetPowerLevel()
       PhysicalLevel := (physical:)GetPowerLevel()
```

The qualified name syntax `(InterfaceName:)MemberName` specifies which interface's member you're accessing. Each interface maintains its own instance of the field, allowing the class to support both contracts simultaneously without conflict.

12.4.6 Interface Hierarchies

Interfaces can extend other interfaces, creating hierarchies of contracts that combine data and behavior requirements:

```
combatant := interface(damageable, healable):
    var AttackPower:int = 10

    Attack(Target:damageable):void =
        Target.TakeDamage(AttackPower)

    GetAttackPower():int = AttackPower

boss := interface(combatant):
    Phase:int = 1

```

```
    UseSpecialAbility():void
    GetPhase():int = Phase
```

A class implementing `boss` inherits all fields and methods from the entire hierarchy—`boss`, `combatant`, `damageable`, and `healable`. Diamond inheritance (where an interface is inherited through multiple paths) is fully supported, with fields properly merged so each field exists only once in the implementing class.

Important: A class cannot directly inherit the same interface multiple times (e.g., `class(interface1, interface1)` is an error), but can inherit it indirectly through diamond inheritance. This means `class(interface2, interface3)` is valid even if both `interface2` and `interface3` inherit from the same base interface.

12.4.7 Fields with Accessors

Interfaces can define fields with custom getter and setter logic, encapsulating complex behavior behind simple field access syntax:

```
subscribable_property := interface:
    # External field with accessor methods
    var Value<getter(GetValue)><setter(SetValue)>:int = external{}

    # Internal storage
    var Storage:int = 100

    # Getter adds computation
    GetValue(:accessor):int = Storage + 10

    # Setter adds validation
    SetValue(:accessor, NewValue:int):void =
        if (NewValue >= 0):
            set Storage = NewValue

tracked_value := class(subscribable_property):

UseTrackedValue():void =
    Object := tracked_value{}

    # Uses getter - returns 110 (Storage + 10)
    Current := Object.Value

    # Uses setter - validates and updates Storage
    set Object.Value = 150
```

The `external{}` keyword indicates the field has no direct storage—all access goes through the accessor methods. This pattern is powerful for implementing property change notifications, validation, computed properties, and other scenarios requiring logic around field access.

Important: Fields with accessors defined in interfaces cannot be overridden in implementing classes. The accessor implementation is fixed by the interface.

Chapter 13

Chapter 12: Type System

Every value has a type, and understanding the type system is fundamental to mastering any language. Types aren't merely labels - they form a rich hierarchy that governs how values flow through your program, what operations are permitted, and how the compiler reasons about your code. The type system combines static verification with practical flexibility, catching errors at compile time while still allowing sophisticated patterns of code reuse and abstraction.

At the top of this hierarchy sits `any`, the universal supertype from which all other types descend. At the bottom lies `false`, the empty type that contains no values at all (the uninhabited type). Between these extremes exists a carefully designed lattice of types, each with its own capabilities and constraints.

13.1 Understanding Subtyping

Subtyping is the foundation of the type hierarchy. When we say that type A is a subtype of type B, we mean that every value of type A can be used wherever a value of type B is expected. This relationship creates a natural ordering among types, from the most specific to the most general.

Consider the relationship between `rational` and `int`. Every integer is a rational number, but not every rational is an integer. Therefore, `int` is a subtype of `rational`. This means you can pass an `int` to any function expecting a `rational`, but not vice versa:

```
GetInt(X:int):void = Print("Integer: {X}")
GetRat(X:rational):void = Print("Rational: {X}")

MyRat:rational = 1/3
MyInt:int = -10
```

```
6
7 GetRat(MyInt)  # OK -- int is a subtype of rational
8 GetInt(MyRat)  # Compile error -  rational is not a subtype of int
```

The subtyping relationship extends to composite types in sophisticated ways. Arrays and tuples follow covariant subtyping rules for their elements. This means that `[]int` is a subtype of `[]rational`. Similarly, `tuple(int, int)` is a subtype of `tuple(rational, rational)`. This covariance allows collections of more specific types to be used where collections of more general types are expected.

Maps exhibit more complex subtyping behavior. A map type `[K1]V1` is a subtype of `[K2]V2` when `K2` is a subtype of `K1` (contravariant in keys) and `V1` is a subtype of `V2` (covariant in values). The contravariance in keys might seem counterintuitive at first, but it ensures type safety: if you can look up values using a more general key type, you must be able to handle more specific key types as well.

Classes and interfaces introduce nominal subtyping through inheritance. When a class inherits from another class or implements an interface, it explicitly declares a subtyping relationship:

```
1 vehicle := class:
2     Speed:float = 0.0
3
4 car := class(vehicle):  # car is a subtype of vehicle
5     NumDoors:int = 4
6
7 sports_car := class(car):  # sports_car is a subtype of car (and vehicle)
8     Turbo:logic = true
```

This inheritance hierarchy means that a `sports_car` can be used anywhere a `car` or `vehicle` is expected, but not the reverse. The subtype inherits all fields and methods from its supertypes while potentially adding new ones or overriding existing ones.

13.2 Numeric and String Conversions

All type conversions must be explicit, a design choice that eliminates entire categories of bugs while making the programmer's intent clear. Converting between numeric types illustrates this principle clearly. To convert an integer to a float, you multiply by 1.0:

```
1 MyI:int   = 42
2 MyF:float = MyI * 1.0  # Explicit conversion to float
```

Note

The strongest reason for dissallowing implicit conversions is that

```
they can cause code to break when new overloadings to a function
are added. Imagine a call to function `f` that takes a float such
as `f(1)`, if the integer argument was implicitly converted to a
float and, in some future library release, an overload `f(:int)`
was added, the call would silently invoke that new function
and potentially change the result of the computation.
```

The reverse conversion, from float to integer, requires choosing a rounding strategy:

```
MyF:float = 3.7
Opt1:int = Floor[MyF]  # Results in 3
Opt2:int = Ceil[MyF]   # Results in 4
Opt3:int = Round[MyF]  # Results in 4 (rounds to nearest)
```

These conversion functions are failable - they have the `<decides>` effect and will fail if passed non-finite values like `NaN` or `Inf`. The explicit failure forces you to handle edge cases:

```
SafeConvert(Value:float):int =
    if:
       Value <> NaN
       Value <> Inf
       Result:= Floor[Value]
    then:
       Result
    else:
       0  # Assuming that this is safe value
```

String conversions follow similar principles. The `ToString()` function converts various types to their string representations, while string interpolation provides a convenient syntax for embedding values in strings:

```
Score:int  = 1500
Msg:string = "Your score: {Score}"  # Implicit ToString() call
```

13.3 Type any

Type `any` is at the top of the type hierarchy it is the universal supertype that can hold a value of any type. Every type in Verse is a subtype of `any`, making it the most permissive type. It serves as an escape hatch when you genuinely need to work with values of unknown or varying types.

Once a value is typed as `any`, you've effectively told the compiler "I don't know what this is," and the compiler responds by preventing most operations. This is by design—without knowing the actual type, the compiler cannot verify that operations are safe.

You can explicitly coerce any value to `any` using function call syntax, `any(42)`.

Verse automatically coerces values to `any` when their types would otherwise be incompatible. Understanding these rules help when working with heterogeneous data.

Mixed-type arrays and maps automatically coerces to the most specific shared type, if no common type is found, the array coerces to `any`:

```
MixedArray := array{42, "hello", true, 3.14} # []comparable
MixedMap := map{0=>"zero", 1=>1, 2=>2.0} # [int]comparable
ConfigMap := map{"count"=>42, "process"=>SomeFunction, "name"=>"Player"} # [string]any
```

Conditional expressions with disjoint branch types produce `any`:

```
# If branches return different types
GetValue(UseString:logic):any =
    if (UseString?):
        "text result"
    else:
        42
```

Logical OR with disjoint types coerces to `any`:

```
# Returns either int or string
OneOf(Flag:logic, I:int, S:string):any =
    (if (Flag?) then {option{I}} else {1=2}) or S
```

The `any` type has restrictions that reflect its role as a generic container:

- You cannot use equality operators with `any`
- Because `any` is not comparable, it cannot be used as a map key type
- Because `any` is not castable, it is a sticky type.

13.4 Class and Interface Casting

Verse provides two distinct casting mechanisms for classes and interfaces: fallible casts for runtime type checking, and infallible casts for compile-time verified conversions. Understanding when and how to use each is essential for working with inheritance hierarchies and polymorphic code.

Fallible casts use square bracket syntax `TargetType[value]` to perform runtime type checks. These casts succeeds and return the casted value (`TargetType`), and failing if the value is not of a valid target type or a subtype:

```
# Define a class hierarchy
component := class<castable>:
    Name:string = "Component"

physics_component := class<castable>(component):
    Velocity:float = 0.0

render_component := class<castable>(component):
    Material:string = "default"

# Runtime type checking with fallible casts
ProcessComponent(Comp:component):void =
    if (PhysicsComp := physics_component[Comp]):
        # Successfully cast - PhysicsComp is physics_component
        Print("Physics velocity: {PhysicsComp.Velocity}")
    else if (RenderComp := render_component[Comp]):
        # Different type - RenderComp is render_component
        Print("Render material: {RenderComp.Material}")
    else:
        # Neither type matched
        Print("Unknown component type")
```

The cast expression evaluates to `false` if the runtime type doesn't match, allowing you to use it directly in conditionals. The optional binding pattern `(Variable := Expression)` both performs the cast and binds the result to a variable when successful.

For classes marked `<unique>`, fallible casts preserve identity—a successful cast returns the same instance, not a copy:

```
entity := class<unique><castable>:
    ID:int

player := class<unique>(entity):
    Name:string

# Create an instance
P := player{ID := 1, Name := "Alice"}

# Cast to base type
```

```
if (E := entity[P]):
    E = P  # True - same instance
```

Fallible casts work **only with class and interface types**. You cannot dynamically cast from or to primitive types, structs, arrays, or other value types:

```
component := class<castable>{}

# Error: cannot cast from primitives
Comp := component[42]          # int to class - not allowed
Comp := component[3.14]        # float to class - not allowed
Comp := component["text"]      # string to class - not allowed
Comp := component[array{1,2}]  # array to class - not allowed

# Error: cannot cast to non-class types
Value := int[component{}]      # class to int - not allowed
Value := logic[component{}]    # class to logic - not allowed
Value := (?int)[component{}]   # class to option - not allowed
```

The restriction exists because fallible casts rely on runtime type information that only classes and interfaces maintain. Value types like integers and structs don't have runtime type tags.

Infallible casts use parenthesis syntax `TargetType(value)` for conversions that the compiler can verify will always succeed. These casts require the source type to be a compile-time subtype of the target type:

```
component := class<castable>:
    Name:string = "Component"

physics_component := class<castable>(component):
    Velocity:float = 0.0

# Upcasting: always safe, always succeeds
Base:physics_component = physics_component{Velocity := 10.0}

BaseComp:component = component(Base) # upcast during expression
# or
AlsoBaseComp:component = Base # upcast during assignment
```

Any type can be infallibly cast to `void`, which discards the value:

```
void(42)           # Discard an integer
void("result")     # Discard a string
void(component{})  # Discard an object
```

This implictly happens when you call a function for its side effects and want to ignore its return value.

13.4.1 Dynamic Type-Based Casting

Types in Verse are first-class values, which means you can store types in variables and use them dynamically for casting. This enables powerful patterns for runtime polymorphism:

```
# Type hierarchy
component := class<castable>{}
physics_component := class<castable>(component){}
render_component := class<castable>(component){}

# Store types as values
ComponentType:castable_subtype(component) = physics_component

# Cast using the stored type
Test(Comp:component, ExpectedType:castable_subtype(component)):logic =
    if (Specific := ExpectedType[Comp]):
        true  # Component matches expected type
    else:
        false

# Use with different types
P := physics_component{}
Test(P, physics_component)  # true
Test(P, render_component)   # false
```

This pattern is particularly powerful when the type to check isn't known until runtime:

```
# Select type based on configuration
GetComponentType(Config:string):castable_subtype(component) =
    if (Config = "physics"):
        physics_component
    else if (Config = "render"):
        render_component
    else:
        component

# Use the dynamically selected type
RequiredType := GetComponentType(LoadedConfig)
for (Comp : Components):
    if (Specific := RequiredType[Comp]):
```

```
        # Process components of the required type
        ProcessSpecific(Specific)
```

This bridges compile-time type safety with runtime flexibility, allowing type decisions to be made based on program state while maintaining type correctness.

13.5 Where Clauses

Where clauses are the mechanism for constraining type parameters in generic code. They appear after type parameters and specify requirements that types must satisfy to be valid arguments. This creates a powerful system for writing generic code that is both flexible and type-safe.

```
# Simple subtype constraint
Process(Value:t where t:subtype(comparable)):void =
    if (Value = Value):  # We know it supports equality
        Print("Value equals itself")
```

Using the same type in multiple constraints is not yet supported, when implemented, it will allow to write code such as:

```
# Multiple constraints on the same type
F(In:t where t:subtype(comparable), t:subtype(printable)):t = # Not supported
    Print("Processing: {In}")
    In
```

Where clauses become more powerful when working with multiple type parameters:

```
# Independent constraints on different parameters
Combine(A:t1, B:t2 where t1:type, t2:type):tuple(t1, t2) =
    (A, B)

# Related constraints
Convert(From:t1, Converter:type{_(:t1):t2} where t1:type, t2:type):t2 =
    Converter(From)
```

Where clauses can express sophisticated relationships between types:

```
# Constraint that ensures compatible types for an operation
Merge(Container1:[]t, Container2:[]t where t:subtype(comparable)):[]t =
    var Result:[]t = Container1
    for (Element : Container2, not Contains[Result, Element]):
        set Result += array{Element}
    Result

```

```
# Function type constraints
ApplyTwice(F:type{_(:t):t}, Value:t where t:type):t =
    F(F(Value))
```

Where clauses enable sophisticated generic programming patterns:

```
MapFunction(F:type{_(:a):b}, Container:[]a where a:type, b:type):[]b =
    for (Element : Container):
        F(Element)
```

13.6 Refinement Types

While `where` clauses constrain type parameters in generic code, **refinement types** use `where` to constrain the *values* a type can hold. This creates subtypes that only accept values satisfying specific conditions, enabling domain-specific constraints enforced by the type system.

A refinement type defines a constrained subtype using value predicates:

```
# Percentages: floats between 0.0 and 1.0
# percent := type{_X:float where 0.0 <= _X, _X <= 1.0}

# Valid assignments
Opacity:percent = 0.5
Alpha:percent = 1.0

# Invalid: out of range (runtime check fails)
# BadPercent:percent = 1.5  # Fails at assignment
```

Syntax structure:

```
TypeName := type{_Variable:BaseType where Constraint1, Constraint2, ...}
```

- `_Variable` is a placeholder for the value being constrained
- `BaseType` is `int` or `float`
- Constraints are comparison expressions using `<=`, `<`, `>=`, or `>`

Integer refinements restrict int values to specific ranges:

```
# Age between 0 and 120
age := type{_X:int where 0 <= _X, _X <= 120}

ValidAge:age = 25
# InvalidAge:age = 150  # Fails constraint

# Positive integers
```

```
positive_int := type{_X:int where _X > 0}

Count:positive_int = 42
# Zero:positive_int = 0  # Fails: not positive

# Range with single bound
small_int := type{_X:int where _X < 100}
```

Float refinements handle continuous ranges with IEEE 754 semantics:

```
# Unit interval [0.0, 1.0]
normalized := type{_X:float where 0.0 <= _X, _X <= 1.0}

# Positive floats
positive := type{_X:float where _X > 0.0}

# Temperature in Celsius above absolute zero
celsius := type{_X:float where _X >= -273.15}
```

Finite Floats (Excluding Infinity):

```
# Finite values only (no ±Inf)
finite := type{_X:float where -Inf < _X, _X < Inf}

# Maximum and minimum finite IEEE 754 doubles
MaxFinite:finite = 1.7976931348623157e+308
MinFinite:finite = -1.7976931348623157e+308

# Invalid: infinities excluded
# Infinite:finite = Inf  # Fails constraint
```

13.6.1 IEEE 754 Edge Cases

Negative and Positive Zero:

IEEE 754 distinguishes between `+0.0` and `-0.0`. In verse Zero is just Zero, with no distinction between positve or negative.

When any expression evaluates to Zero, the sign is discarded:

```
# Integer Zero (type{0})
Value1 := -0
Value2 := +0

Value1 = Value2 # Succeeds
-0 = +0         # Succeeds
```

```
7
8  # Float Zero (type{0.0})
9  Value3 := -0.0
10 Value4 := +0.0
11
12 Value3 = Value4 # Succeeds
13 -0.0 = +0.0     # Succeeds
```

Floating-Point Precision:

Constraints respect exact IEEE 754 representations:

```
1  # Values strictly less than 0.1
2  small_float := type{_X:float where _X < 0.1}
3
4  # Valid: largest float before 0.1
5  Tiny:small_float = 0.09999999999999999167332731531132594682276248931884765625
6
7  # Invalid: 0.1's actual representation is slightly above 0.1
8  # NotSmall:small_float = 0.10000000000000000555111512312578270211815834045410156250
```

The decimal `0.1` cannot be represented exactly in binary floating-point, so the actual stored value is slightly above the mathematical 0.1.

13.6.2 Constraint Expression Restrictions

Refinement type constraints have strict limitations on what expressions are allowed:

Only Literal Values: Constraints must use literal numbers, not variables or expressions:

```
1  # Valid: literal float
2  bounded := type{_X:float where _X < 100.0}
3
4  # Invalid: cannot use variables
5  Limit:float = 100.0
6  bad_type := type{_X:float where _X < Limit}  # ERROR
7
8  # Invalid: cannot use function calls
9  GetMax():float = 100.0
10 bad_type := type{_X:float where _X < GetMax()}  # ERROR
11
12 # Invalid: cannot use qualified names
13 config := module{Max:float = 100.0}
14 bad_type := type{_X:float where _X < (config:)Max}  # ERROR
```

This ensures constraints are statically known at compile time.

Float Literals Required for Float Types: When constraining floats, bounds must be float literals (with decimal point):

```
# Invalid: integer literal in float constraint
# bad_float := type{_X:float where _X <= 142}  # ERROR 3502

# Valid: float literal
good_float := type{_X:float where _X <= 142.0}
```

NaN Not Allowed: Not a Number cannot appear in constraints:

```
# Invalid: NaN in constraint
# nan_type := type{_X:float where _X <= NaN}      # ERROR 3502
# nan_type := type{_X:float where NaN <= _X}      # ERROR 3502
# nan_type := type{_X:float where 0.0/0.0 <= _X}  # ERROR 3502
```

Since `NaN` comparisons are always false, such constraints would be meaningless.

Allowed Literal Forms:

- Float literals: `1.0`, `3.14`, `-2.5`, `1.7976931348623157e+308`
- Integer literals: `0`, `42`, `-100` (for int refinements)
- Special float values: `Inf`, `-Inf`

13.6.3 Fallible Casts

Refinement types are checked at assignment and through fallible casts:

```
percent := type{_X:float where 0.0 <= _X, _X <= 1.0}

# Direct assignment (compile-time known)
Valid:percent = 0.5  # OK

# Runtime check with fallible cast
UserInput:float = GetInputFromUser()
if (Value := percent[UserInput]):
    # UserInput was in [0.0, 1.0]
    ProcessPercent(Value)
else:
    # Out of range
    ShowError()
```

The cast `percent[UserInput]` returns `percent` succeeding if the value satisfies the constraint, or failing otherwise.

13.6.4 Examples

Refinement types work as parameter and return types:

```
finite := type{_X:float where -Inf < _X, _X < Inf}

# Parameter with constraint
Half(X:finite):float = X / 2.0

Half(100.0)  # Returns 50.0
Half(1.0)    # Returns 0.5

# Cannot pass infinity
# Half(Inf)  # ERROR 3509: Inf not in finite
```

Coercion and Negation:

```
percent := type{_X:float where 0.0 <= _X, _X <= 1.0}
negative_percent := type{_X:float where _X <= 0.0, _X >= -1.0}

MakePercent():percent = 0.5

# Negation preserves constraint compatibility
NegValue:negative_percent = -MakePercent()  # -0.5 valid

# Multiple negations
NegValue2:negative_percent = ---0.7  # Triple negation = -0.7
```

13.6.5 Overloading Restrictions

Overlapping refinement types cannot be used for function overloading—they're ambiguous:

```
percent := type{_X:float where 0.0 <= _X, _X <= 1.0}
not_infinity := type{_X:float where Inf > _X}

# ERROR 3532: Cannot distinguish - percent  not_infinity
# F(X:percent):float = 0.0
# F(X:not_infinity):float = X

# Calling F(0.5) would be ambiguous - which overload?
```

However, **disjoint** refinement types can overload:

```
positive := type{_X:float where _X > 0.0}
negative := type{_X:float where _X < 0.0}
```

```
3
4  # Valid: ranges don't overlap (zero excluded from both)
5  F(X:positive):float = X
6  F(X:negative):float = X + 1.0
7
8  F(1.0)   # Returns 1.0 (positive overload)
9  F(-1.0)  # Returns 0.0 (negative overload)
10 # F(0.0)  # Would fail - neither overload matches
```

13.7 Comparable and Equality

The `comparable` type represents a special subset of types that support equality comparison. Not all types can be compared for equality - this is a deliberate design choice that prevents meaningless comparisons and ensures that equality has well-defined semantics.

A type is comparable if its values can be meaningfully tested for equality. The basic scalar types are all comparable: `int`, `float`, `rational`, `logic`, `char`, and `char32`. Compound types are comparable if all their components are comparable. This means arrays of integers are comparable, tuples of floats and strings are comparable, and maps with comparable keys and values are comparable.

The equality operators `=` and `<>` are defined in terms of the comparable type:

```
operator'='(X:t, Y:t where t:subtype(comparable))<decides>:t
operator'<>'(X:t, Y:t where t:subtype(comparable))<decides>:t
```

The signatures requires that both operands be subtypes of comparable and the return type is the least upper bound of their types.

```
0 = 0        # Succeeds - both are int
0.0 = 0.0    # Succeeds - both are float
0 = 0.0      # Fails - there is no implicit conversion from int to float
```

Here is an example that highlights how the return type of `=` is computed:

```
I:int=1
R:rational=1/3
X:rational= (I=R)  # Compiles and fails at runtime

I:int=1
S:string="hi"
Y:comparable= (I=S)  # Compiles and fails at runtime
```

In the case of variable `X`, its type can be either `rational` or `comparable`. For variable `Y`, the only common type between `int` and `string` is `comparable`.

Classes require special handling for comparability. By default, class instances are not comparable because there's no universal way to define equality for user-defined types. However, you can make a class comparable using the `unique` specifier:

```
entity := class<unique>:
    ID:int
    Name:string

Player1 := entity{ID := 1, Name := "Alice"}
Player2 := entity{ID := 1, Name := "Alice"}
Player3 := Player1

Player1 = Player2  # Fails - different instances
Player1 = Player3  # Succeeds - same instance
```

With the `unique` specifier, instances are only equal to themselves (identity equality), not to other instances with the same field values (structural equality). This provides a clear, predictable semantics for class equality.

13.7.1 Comparable as a Generic Constraint

The `comparable` type is commonly used as a constraint in generic functions to ensure operations like equality testing are available:

```
Find(Items:[]t, Target:t where t:subtype(comparable))<decides>:int =
    for (Index->Item:Items):
        if (Item = Target):
            return Index
    false? # Force fail if not found

# Works with any comparable type
Position := Find[array{"apple", "banana", "cherry"}, "banana"]  # Succeeds and returns 1
```

13.7.2 Array-Tuple Comparison

A notable feature of Verse's equality system is that arrays and tuples of comparable elements can be compared with each other:

```
# Arrays can equal tuples
array{1, 2, 3} = (1, 2, 3)       # Succeeds
(4, 5, 6) = array{4, 5, 6}       # Succeeds - bidirectional

# Inequality also works
array{1, 2, 3} <> (1, 2, 4)      # Succeeds - different values
```

This comparison works structurally - the sequences must have the same length and corresponding elements must be equal. This feature allows functions expecting arrays to accept tuples, increasing flexibility.

13.7.3 Overload Distinctness with Comparable

You cannot create overloads where one parameter is a specific comparable type and another is the general `comparable` type, as this creates ambiguity:

```
# Not allowed - ambiguous overloads
F(X:int):void = {}
F(X:comparable):void = {}  # ERROR: int is already comparable

# Not allowed with unique classes either
unique_class := class<unique>{}
G(X:unique_class):void = {}
G(X:comparable):void = {}  # ERROR: unique_class is comparable
```

However, you can overload with non-comparable types:

```
# This is allowed
regular_class := class{}  # Not comparable
H(X:regular_class):void = {}
H(X:comparable):void = {}  # OK: no ambiguity
```

13.7.4 Dynamic Comparable Values

When working with heterogeneous collections, you may need to box comparable values into the `comparable` type explicitly. These boxed values maintain their equality semantics:

```
AsComparable(X:comparable):comparable = X

# Boxed values compare correctly with both boxed and unboxed
array{AsComparable(1)} = array{1}              # Succeeds
array{AsComparable(1)} = array{AsComparable(1)} # Succeeds
array{AsComparable(1)} <> array{2}             # Succeeds

# With direct upcasting:
comparable(15) = comparable(15)     # Succeeds
comparable("Hello") = "Hello"       # Succeeds
```

This allows you to create collections that mix different comparable types by boxing them all to `comparable`.

13.7.5 Map Keys and Comparable

Map keys must be comparable types. Most comparable types can be used as map keys, including:

- All numeric types: `int`, `float`, `rational`
- Character types: `char`, `char32`
- Text: `string`
- Enumerations
- `<unique>` classes
- Optionals of comparable types: `?t` where `t` is comparable
- Arrays of comparable types: `[]t` where `t` is comparable
- Tuples of comparable types
- Maps with comparable keys and values: `[k]v`
- Structs with comparable fields

Note that while `float` can be used as a map key, floating-point special values have specific equality semantics (see Map documentation for details on `NaN` and zero handling).

There is currently no way to make a regular class comparable by writing a custom comparison method. Only the `<unique>` specifier enables class comparability through identity equality.

13.8 Generators

Generators represent lazy sequences that produce values on demand rather than storing all elements in memory. Unlike arrays which materialize all elements upfront, generators compute each value only when requested during iteration. This makes them memory-efficient for large or infinite sequences, and essential for scenarios where you're processing streaming data or expensive computations.

Generators use the parametric type `generator(t)` where `t` is the element type:

```
# Generator of integers
IntSequence:generator(int) = MakeIntegerSequence()

# Generator of entities
EntityStream:generator(entity) = GetAllEntities()
```

Syntax restrictions:

```
# Correct: Use parentheses
ValidGenerator:generator(int) = GetSequence()

# Wrong: Square brackets are invalid
```

```
# BadGenerator:generator[int] = GetSequence()  # ERROR

# Wrong: Curly braces are invalid
# BadGenerator:generator{int} = GetSequence()  # ERROR
```

Element types must be valid Verse types, not literals or expressions:

```
# Valid
generator(int)
generator(string)
generator(my_class)

# Invalid: Cannot use literals
# generator(1)        # ERROR 3547
# generator("text")   # ERROR 3547
```

Constrained types work as element types:

```
# Valid: Constrained element type
PositiveInts:generator(type{X:int where X > 0, X < 10}) = GetConstrainedSequence()
```

13.8.1 For Loops

The primary way to consume generators is through `for` expressions:

```
# Direct iteration
ProcessStream()<transacts>:void =
    for (Item : GetIntegerSequence()):
        Print("{Item}")

# Store in variable first
ProcessWithVariable()<transacts>:void =
    Sequence := GetIntegerSequence()
    for (Item : Sequence):
        Print("{Item}")
```

Generators work with arrow syntax in loops, showing that domain and range are identical:

```
DoubleCheck():logic =
    for (Index->Value : GetFloatSequence()):
        # Index and Value are the same
        Index = Value
```

Multiple generators in one loop:

```
ProcessPairs()<transacts>:void =
    var Total:float = 0.0
    for (A : GetFloatSequence(), B : GetFloatSequence()):
        set Total += A + B
```

Combining generators with conditions:

```
FilteredSum()<transacts>:float =
    var Total:float = 0.0
    for (
        A : GetFloatSequence(),
        B : array{1.0, 2.0, 4.0, 8.0},
        A <> 4.0,
        B <> 4.0
    ):
        set Total += A + B
    Total
```

13.8.2 Restrictions

Generators have strict type conversion rules to maintain safety:

Cannot convert arrays to generators:

```
Numbers := array{1, 2, 3}
# Seq:generator(int) = Numbers  # ERROR 3509
```

Cannot convert between incompatible element types:

```
IntSeq := GetIntegerSequence()
# FloatSeq:generator(float) = IntSeq  # ERROR 3509
```

Cannot index generators like arrays:

```
Seq := GetIntegerSequence()
# Value := Seq[0]  # ERROR 3509
# Generators don't support random access
```

Converting generators to arrays:

Use a `for` expression to materialize the sequence:

```
GeneratorToArray(Gen:generator(t) where t:type):[]t =
    for (Item : Gen):
        Item

Numbers := GeneratorToArray(GetIntegerSequence())
# Numbers is now array{1, 2, 3, 4}
```

13.8.3 Covariance

Generators are **covariant** in their element type when the element type has subtyping relationships:

```
animal := class:
    Name:string

dog := class(animal):
    Breed:string

# Covariant: generator(dog) is a subtype of generator(animal)
DogStream:generator(dog) = GetDogSequence()
AnimalStream:generator(animal) = DogStream  # OK - covariance

# Cannot upcast: generator(animal) is NOT a subtype of generator(dog)
GeneralStream:generator(animal) = GetAnimalSequence()
# SpecificStream:generator(dog) = GeneralStream  # ERROR
```

This covariance enables flexible APIs:

```
# Function accepting generator of base type
ProcessAnimals(Animals:generator(animal)):void =
    for (A : Animals):
        Print(A.Name)

# Can pass generator of derived type
ProcessAnimals(GetDogSequence())  # OK due to covariance
```

13.8.4 Type Joining

When conditionally selecting between generators, Verse finds the least common supertype:

```
base := class:
    ID:int

child1 := class(base):
    Extra1:string

child2 := class(base):
    Extra2:int

# Conditional selection finds common supertype
GetStream(UseFirst:logic):generator(base) =
    if (UseFirst?):
```

```
        GetChild1Sequence()  # Returns generator(child1)
    else:
        GetChild2Sequence()  # Returns generator(child2)
    # Result type: generator(base)
```

Similar to effect joining, the compiler computes the least upper bound (join) of the generator element types.

13.8.5 Constraints and Limitations

- **No random access:** Generators don't support indexing or random access operations. They're strictly sequential.
- **No reusability:** Most generators can only be iterated once. After consuming a generator, it's exhausted.

13.9 Type Hierarchies

The type system forms a lattice rather than a simple tree. This means types can have multiple supertypes, though multiple inheritance is currently limited to interfaces. Understanding these relationships helps you design flexible, reusable code.

13.9.1 Understanding void

Unlike `any`, which erases type information, `void` serves as a "discard" type indicating that a value's specific type doesn't matter.

Functions with `void` return type can return any value, which is then discarded by the type system:

```
LogEvent(Message:string)<transacts>:void =
    WriteToFile(Message)
    42                  # Returns int, but typed as void

F():void = 1            # Valid - returns int, typed as void
F()                     # Result is void
```

Despite being typed as `void`, these functions still produce their computed values—the values are simply not accessible through the type system. This ensures side effects and computations occur even when the return value is discarded:

```
MakePair(X:string, Y:string):void = (X, Y)

# Function computes the pair even though return type is void
MakePair("hello", "world")  # Still creates ("hello", "world")
```

Functions with `void` parameters accept any argument type:

```
Discard(X:void):int = 42

Discard(0)               # int → void
Discard(1.5)             # float → void
Discard("test")          # string → void
```

Class fields can be typed as `void`, accepting any initialization value:

```
config := class:
    Setting:void = array{1, 2}  # Default with array
```

In function types, `void` participates in variance:

```
IntIdentity(X:int):int = X

# Contravariant return: supertype in return position
F:int->void = IntIdentity  # int->int → int->void
# void is supertype of int, so this works

AcceptVoid(X:void):int = 19

# Contravariant parameter: supertype in parameter position
G:int->int = AcceptVoid    # void->int → int->int
# Can use function accepting void where function accepting int expected
```

However, `void` in parameter position does NOT allow conversion the other way:

```
IntFunction(X:int):int = X
# F:void->int = IntFunction  # ERROR 3509
# Cannot convert int parameter to void parameter in function type
```

void vs false: The `false` type is the empty/bottom type (uninhabited type) with no values. It's the opposite of `void`:

- `void`: Universal supertype - all types are subtypes of void, contains all values
- `false`: Bottom type - subtype of all types, contains zero values

Between the universal supertypes (`any`, `void`) and the bottom type (`false`), types form natural groupings. The numeric types (`int`, `float`, `rational`) share common arithmetic operations but don't form a single hierarchy - they're siblings rather than ancestors and descendants. The container types (arrays, maps, tuples, options) each have their own subtyping rules based on their element types.

Understanding variance is crucial for working with generic containers. Arrays and options are covariant in their element type - if A is a subtype of B, then `[]A` is a subtype of `[]B` and `?A` is a subtype of `?B`. This allows natural code like:

```
ProcessNumbers(Numbers:[]rational):void =
    for (N : Numbers):
        Print("{N}")

Numbers:[]int = array{1, 2, 3}
ProcessNumbers(Numbers)  # Works due to covariance
```

Functions exhibit more complex variance. They're contravariant in their parameter types and covariant in their return types. A function type `(T1)->R1` is a subtype of `(T2)->R2` if T2 is a subtype of T1 (contravariance) and R1 is a subtype of R2 (covariance). This ensures that function subtyping preserves type safety:

```
function_type1 := type{_(:any):int}
function_type2 := type{_(:int):any}

# function_type1 is a subtype of function_type2
# It accepts more general input (any vs int)
# And returns more specific output (int vs any)
```

13.10 Type Aliases

Type aliases allow you to create alternative names for types, making complex type signatures more readable and maintainable. They're particularly valuable for function types, parametric types, and frequently-used type combinations.

A type alias is created using simple assignment syntax at module scope:

```
# Simple type aliases
coordinate := tuple(float, float, float)
entity_map := [string]entity
player_id := int

# Function type aliases
update_handler := type{_(:float):void}
validator := int -> logic
transformer := type{_(:string):int}
```

Type aliases are compile-time only - they create no runtime overhead and are purely for programmer convenience and code clarity.

Type aliases are alternative names, not new types. They don't create distinct types like `newtype` in some languages. Values of the alias and the original type are completely interchangeable:

```
# Assume
# player_id := int
# game_id := int

ProcessPlayer(ID:player_id):void = {}
ProcessGame(ID:game_id):void = {}

PID:player_id = 42
GID:game_id = 42

# These all work - aliases are just names
ProcessPlayer(PID)      # OK
ProcessPlayer(GID)      # OK - game_id is also int
ProcessPlayer(42)       # OK - int literal works too
ProcessGame(PID)        # OK - player_id is also int
```

Type aliases can have access specifiers that control their visibility across modules:

```
# Public alias - accessible from other modules
PublicAlias<public> := int

# Internal alias - only accessible within defining module
InternalAlias<internal> := string

# Protected/private also work
ProtectedAlias<protected> := float  # only in classes and interfaces
```

Type aliases cannot be more public than the types they alias:

```
PrivateClass := class{}      # No specifier = internal scope

# INVALID: Public alias to internal type (ERROR 3593)
# PublicToPrivate<public> := PrivateClass

# VALID: Same or less visibility
InternalToInternal<internal> := PrivateClass
InternalAlias := PrivateClass  # Defaults to internal
```

This restriction applies to all type constructs:

```
PrivateType := class{}

```

```
# All INVALID - trying to make internal type public (ERROR 3593)
# Pub1<public> := ?PrivateType           # Optional
# Pub2<public> := []PrivateType          # Array
# Pub3<public> := [int]PrivateType       # Map value
# Pub4<public> := [PrivateType]int       # Map key
# Pub5<public> := tuple(int, PrivateType)  # Tuple
# Pub6<public> := PrivateType -> int     # Function parameter
# Pub7<public> := int -> PrivateType     # Function return
# Pub8<public> := type{_():PrivateType}  # Function type
```

13.10.1 Requirement

- **Type aliases can only be defined at module scope.** They cannot be defined inside classes, functions, or any nested scope. This restriction ensures type aliases have consistent visibility and prevents scope-dependent type interpretations.
- Type aliases must be defined **before** they are used. Forward references are not allowed.
- Type aliases are not first-class values and cannot be used as such.

13.11 Metatypes

Verse provides advanced type constructors that allow you to work with types as values, enabling powerful patterns for runtime polymorphism and generic instantiation. These metatypes—`subtype`, `concrete_subtype`, and `castable_subtype`—bridge the gap between compile-time type safety and runtime flexibility.

13.11.1 subtype

The `subtype(T)` type constructor represents runtime type values that are subtypes of `T`. Unlike `concrete_subtype` and `castable_subtype`, which are specialized for classes and interfaces, `subtype(T)` works with **any type** in Verse, including primitives, enums, collections, and function types.

```
C0 := class {}
C1 := class(C0) {}

C2 := class:
    var m0:subtype(C0)  # Can hold C0, C1, or any subtype of C0
    var m1:subtype(C2)  # Can hold C2 or any subtype of C2

    # Assign class types
    f0():void = set m0 = C0
```

```
    f1():void = set m0 = C1  # C1 is subtype of C0

    # Accept as parameter
    f3(classArg:subtype(C0)):void = set m0 = classArg
```

The key capability of `subtype(T)` is holding type values at runtime while maintaining type safety through the subtype relationship.

Unlike the other metatypes, `subtype(T)` accepts any type as its parameter:

```
# Primitives
IntType:subtype(int) = int
LogicType:subtype(logic) = logic
FloatType:subtype(float) = float

# Enums
#my_enum := enum { A, B, C }
EnumType:subtype(my_enum) = my_enum

# Collections
ArrayType:subtype([]int) = []int
OptionType:subtype(?string) = ?string

# Function types
FuncType:subtype(type{_():void}) = type{_():void}

# Classes and interfaces
ClassType:subtype(my_class) = my_class
InterfaceType:subtype(my_interface) = my_interface
```

This universality makes `subtype(T)` the most flexible of the metatypes, suitable for any scenario where you need to store or pass type values.

Subtyping Relationship:

The `subtype` constructor preserves the subtyping relationship: `subtype(T) <: subtype(U)` if and only if `T <: U`. This means you can assign a more specific subtype to a less specific one:

```
super_class := class{}
sub_class := class(super_class) {}

# Covariance: sub_class <: super_class
SubtypeVar:subtype(sub_class) = sub_class
SupertypeVar:subtype(super_class) = SubtypeVar  # Valid

```

```
# Reverse fails - super_class is not <: sub_class
# SubtypeVar2:subtype(sub_class) = super_class
```

This also applies to interfaces:

```
super_interface := interface{}
sub_interface := interface(super_interface) {}

class_impl := class(sub_interface) {}

# Covariance through interface hierarchy
SpecificType:subtype(sub_interface) = class_impl
GeneralType:subtype(super_interface) = SpecificType  # Valid
```

Using with Interfaces:

When working with interfaces, `subtype(T)` can hold any class that implements the interface:

```
printable := interface:
    PrintIt():void

document := class(printable):
    PrintIt<override>():void = {}

# Can hold any type implementing printable
DocumentType:subtype(printable) = document
```

Relationship to `type`:

Both `subtype(T)` and `castable_subtype(T)` are subtypes of `type`, meaning they can be used where `type` is expected:

```
C := class:
    f(c:subtype(C)):type = return(c)  # Valid: subtype(C) <: type

T := interface {}
g(x:subtype(T)):type = x  # Valid: subtype(T) <: type
```

Restrictions:

While `subtype(T)` is flexible, it has important restrictions:

1. **Cannot use as value:** `subtype(T)` is a type constructor, not a value. You cannot use `subtype(T)` itself as a value.
2. **Exactly one argument:** `subtype` requires exactly one type argument.

3. **Cannot use with attributes:** `subtype` cannot be used with classes that inherit from `attribute`.

13.11.2 concrete_subtype

The `concrete_subtype(t)` type constructor creates a type that represents concrete (instantiable) subclasses of `t`. A concrete class is one that can be instantiated directly—it has the `<concrete>` specifier and provides default values for all fields:

```
# Abstract base class
entity := class<abstract>:
    Name:string
    GetDescription():string

# Concrete implementations
player := class<concrete>(entity):
    Name<override>:string = "Player"
    GetDescription<override>():string = "A player character"

enemy := class<concrete>(entity):
    Name<override>:string = "Enemy"
    GetDescription<override>():string = "An enemy creature"

# Class that stores a type and can instantiate it
spawner := class:
    EntityType:concrete_subtype(entity)

    Spawn():entity =
        # Instantiate using the stored type
        EntityType{}

# Use it
# NewEntity := spawner{EntityType := player}.Spawn()
```

The key feature of `concrete_subtype` is that it ensures the stored type can be instantiated. Without this constraint, you couldn't safely call `EntityType{}` because abstract classes cannot be instantiated.

Requirements

A type can be used with `concrete_subtype` only if it's a class or interface type. Additionally, the actual type value assigned must be a concrete class—one marked with `<concrete>` and having all fields with defaults:

```
# Valid: concrete class with all defaults
config := class<concrete>:
    MaxPlayers:int = 8
    TimeLimit:float = 300.0

ConfigType:concrete_subtype(config) = config  # Valid

# Invalid: abstract class cannot be concrete_subtype
abstract_base := class<abstract>:
    Value:int

# This would be an error:
# BaseType:concrete_subtype(abstract_base) = abstract_base
```

When you have a `concrete_subtype`, you can instantiate it with the empty archetype `{}`, but you cannot provide field initializers—the concrete class must provide all necessary defaults:

```
entity_base := class<abstract>:
    Health:int

warrior := class<concrete>(entity_base):
    Health<override>:int = 100

EntityType:concrete_subtype(entity_base) = warrior

# Valid: empty archetype uses defaults
# Instance := EntityType{}

# Invalid: cannot initialize fields through metatype
# Instance := EntityType{Health := 150}
```

13.11.3 castable_subtype

The `castable_subtype(t)` type constructor represents types that are subtypes of `t` and marked with the `<castable>` specifier. This enables runtime type queries and dynamic casting, which is essential for component systems and polymorphic hierarchies:

```
# Castable base class
component := class<abstract><castable>:
    Owner:entity

# Castable subtypes
```

```
physics_component := class<castable>(component):
    Velocity:vector3

render_component := class<castable>(component):
    Material:string

# Function accepting castable subtype
ProcessComponent(CompType:castable_subtype(component), Comp:component):void =
    # Can use CompType to perform type-safe casts
    if (Specific := CompType[Comp]):
        # Comp is now known to be of type CompType
```

13.11.4 final_super and Type Queries

The `castable_subtype` works with the `<final_super>` specifier and `GetCastableFinalSuperClass` function to enable sophisticated runtime type queries. This combination provides a powerful mechanism for component systems and polymorphic architectures.

The `<final_super>` specifier marks classes as stable anchor points in inheritance hierarchies. These "final super classes" act as canonical representatives for families of related types:

```
component := class<castable>:
    Owner:entity

# Stable anchor for the physics component family
physics_component := class<final_super>(component):
    Velocity:vector3

# Specific implementations inherit from the anchor
rigid_body := class(physics_component):
    Mass:float

soft_body := class(physics_component):
    SpringConstant:float
```

By marking `physics_component` as `<final_super>`, you declare it as the canonical representative for all physics-related components. Even though `rigid_body` and `soft_body` are distinct types, they both belong to the "physics_component family" anchored at `physics_component`.

GetCastableFinalSuperClass

The `GetCastableFinalSuperClass` function queries the type hierarchy to find the `<final_super>` class between a base type and a derived type. Two variants exist:

```
# Takes an instance
GetCastableFinalSuperClass(BaseType, instance)<decides>:castable_subtype(BaseType)

# Takes a type
GetCastableFinalSuperClassFromType(BaseType, Type)<decides>:castable_subtype(BaseType)
```

Both return a `castable_subtype` representing the most specific `<final_super>` class that:

1. Directly inherits from the specified base type
2. Is in the inheritance chain of the instance/type

The function fails if no appropriate `<final_super>` class exists.

Consider this hierarchy:

```
component := class<castable>:
    ID:int

# Direct final_super subclass of component
physics_component := class<final_super>(component):
    Velocity:vector3

# Descendants of physics_component
rigid_body := class(physics_component):
    Mass:float

character_body := class(rigid_body):
    Health:int
```

Query results:

```
# All instances in the physics_component family return physics_component
Body := character_body{ID:=1, Velocity:=vector3{}, Mass:=10.0, Health:=100}

if (Family := GetCastableFinalSuperClass[component, Body]):
    # Family = physics_component (the final_super anchor)
    # Even though Body is character_body, the family anchor is physics_component
```

The function "walks up" the inheritance chain from `character_body` → `rigid_body` → `physics_component` and stops at `physics_component` because:

1. It has `<final_super>`
2. It directly inherits from the queried base (`component`)

When Queries Succeed and Fail?

Succeeds when:

- A `<final_super>` class directly inherits from the base type
- The instance/type inherits from that `<final_super>` class

```
base := class<castable>:
    Value:int

anchor := class<final_super>(base):
    Extra:string

derived := class(anchor):
    More:string

# Valid: anchor is final_super of base, derived inherits from anchor
GetCastableFinalSuperClass[base, derived{}]  # Returns anchor
GetCastableFinalSuperClass[base, anchor{}]   # Returns anchor
```

Fails when:

- No `<final_super>` class exists between base and instance
- The queried type itself is the instance type (cannot query from same level)
- Instance is not a subtype of the base

Multiple Final Supers

You can have multiple `<final_super>` classes at different levels. The function returns the one directly inheriting from the queried base:

```
base := class<castable>:
    ID:int

first_anchor := class<final_super>(base):
    Category:string

second_anchor := class<final_super>(first_anchor):
    Subcategory:string

leaf := class(second_anchor):
    Specific:string
```

```
12
13 # Query from base returns first_anchor
14 GetCastableFinalSuperClass[base, leaf{}]  # Returns first_anchor
15
16 # Query from first_anchor returns second_anchor
17 GetCastableFinalSuperClass[first_anchor, leaf{}]  # Returns second_anchor
```

This layered approach allows hierarchical categorization where different levels represent different granularities of type families.

GetCastableFinalSuperClassFromType

The type-based variant works identically but takes a type instead of instance:

```
1 # Same behavior, different syntax
2 TypeFamily := GetCastableFinalSuperClassFromType[component, rigid_body]
3 InstanceFamily := GetCastableFinalSuperClass[component, rigid_body{}]
4
5 # Both return the same castable_subtype
```

This is useful when working with type values directly rather than instances.

13.11.5 classifiable_subset

Building on the concept of runtime type queries introduced by `castable_subtype`, Verse provides `classifiable_subset`—a sophisticated mechanism for maintaining sets of runtime types. Where `castable_subtype` represents a single type value, `classifiable_subset` represents a collection of types, tracking which classes are present in a system and supporting queries based on type hierarchies.

This feature is particularly valuable for component-based architectures, where you need to track which component types an entity possesses, query for specific capabilities, or filter operations based on type compatibility. Rather than maintaining separate boolean flags or type tags, `classifiable_subset` provides a type-safe, hierarchy-aware registry of runtime types.

Three related types work together to provide both immutable and mutable type sets:

`classifiable_subset(t)` represents an immutable set of runtime types, where `t` must be a `<castable>` base type. Once created, the set cannot be modified, making it suitable for configuration, capability descriptions, or any scenario where the type set should remain stable.

`classifiable_subset_var(t)` provides a mutable variant with `Read()` and `Write()` operations, enabling dynamic type sets that change during program execution.

This is essential for runtime systems where component types are added or removed as entities evolve.

`classifiable_subset_key(t)` represents keys used to identify specific instances when adding them to a mutable set. These keys enable removal of specific instances later, supporting lifecycle management of registered types.

Unlike ordinary classes, `classifiable_subset` types cannot be directly instantiated. You must use the constructor functions `MakeClassifiableSubset()` and `MakeClassifiableSubsetVar()`:

```
# Immutable set, initially empty
EmptySet:classifiable_subset(component) = MakeClassifiableSubset()

# Immutable set with initial instances
InitialSet:classifiable_subset(component) =
    MakeClassifiableSubset(array{physics_component{}, render_component{}})

# Mutable set
DynamicSet:classifiable_subset_var(component) = MakeClassifiableSubsetVar()
```

The base type `t` must be `<castable>`, ensuring runtime type queries are possible. This restriction is enforced at compile time:

```
ComponentSet:classifiable_subset(component) = MakeClassifiableSubset()

# Invalid: non-castable types cannot be used
regular_class := class:
    Value:int

# This would be an error:
# BadSet:classifiable_subset(regular_class) = MakeClassifiableSubset()
```

You cannot subclass these types or create instances through ordinary construction syntax. This ensures that all sets use the proper internal representation for efficient type queries.

Type Hierarchy Semantics

The crucial insight of `classifiable_subset` is that it tracks runtime types, not individual instances. When you add an instance to the set, the system records that instance's actual runtime type. More importantly, type queries respect the inheritance hierarchy:

```
# Add a rigid body instance
Set:classifiable_subset(component) =
    MakeClassifiableSubset(array{rigid_body_component{}})

```

```
# Query results respect hierarchy
Set.Contains[component]             # true - rigid_body is a component
Set.Contains[physics_component]     # true - rigid_body is a physics_component
Set.Contains[rigid_body_component]  # true - directly present
```

This hierarchy awareness makes `classifiable_subset` fundamentally different from a simple set of type tags. The `Contains` operation asks "does this set contain any type that is-a T?" rather than "does this set contain exactly T?".

When you add instances of different types, each distinct runtime type is tracked separately:

```
# Add multiple different types
TheSet:classifiable_subset_var(component) = MakeClassifiableSubsetVar()
Key1 := TheSet.Add(physics_component{})
Key2 := TheSet.Add(render_component{})
Key3 := TheSet.Add(audio_component{})

TheSet.Contains[component]          # succeeds - all three are components
TheSet.Contains[physics_component]  # succeeds - physics_component present
TheSet.Contains[render_component]   # succeeds - render_component present
```

The set remembers each distinct type that was added. When you remove an instance by its key, that specific type is removed only if it was the last instance of that type:

```
# Add multiple instances of same type
TheSet:classifiable_subset_var(component) = MakeClassifiableSubsetVar()
Key1 := TheSet.Add(physics_component{})
Key2 := TheSet.Add(physics_component{})

TheSet.Contains[physics_component]  # succeeds

TheSet.Remove[Key1]
TheSet.Contains[physics_component]  # still succeeds - Key2 remains

TheSet.Remove[Key2]
# TheSet.Contains[physics_component]  # fail - last instance removed
```

Core Operations

The `classifiable_subset` types provide several operations for querying and manipulating type sets:

Contains checks whether any type in the set matches or is a subtype of the queried type:

```
TheSet:classifiable_subset(component) =
    MakeClassifiableSubset(array{physics_component{}})

if (TheSet.Contains[component]):
    # Physics component is present (and is a component)

if (TheSet.Contains[render_component]):
    # No render component present
```

ContainsAll verifies that all types in an array are present in the set:

```
TheSet:classifiable_subset(component) =
    MakeClassifiableSubset(array{physics_component{}})

if (TheSet.ContainsAll[array{physics_component, render_component}]):
    # Both physics and render components are present
```

ContainsAny checks whether at least one type from an array is present:

```
if (TheSet.ContainsAny[array{physics_component, audio_component}]):
    # Either physics or audio component (or both) is present
```

Add (mutable sets only) adds an instance and returns a key for later removal:

```
TheSet:classifiable_subset_var(component) = MakeClassifiableSubsetVar()
Key := TheSet.Add(physics_component{})
# Can later remove using Key
```

Remove (mutable sets only) removes a previously added instance by its key:

```
TheSet:classifiable_subset(component) = MakeClassifiableSubsetVar()

Key := TheSet.Add(physics_component{})

if (TheSet.Remove[Key]):
    # Successfully removed
else:
    # Key was not present (already removed or never added)
```

FilterByType creates a new set containing only types that are compatible (assignable to or from) the specified type:

```
TheSet:classifiable_subset(component) = MakeClassifiableSubset(array{
    physics_component{}, render_component{}, audio_component{}})

# Filter to physics-related types
PhysicsSet := TheSet.FilterByType(physics_component)
```

```
PhysicsSet.Contains[physics_component]  # true
PhysicsSet.Contains[render_component]   # false - unrelated sibling
PhysicsSet.Contains[component]          # true - base type compatible
```

The filtering respects both upward and downward compatibility in the type hierarchy, keeping types that could be assigned to or from the filter type.

Union combines two sets using the `+` operator:

```
Set1:classifiable_subset(component) =
    MakeClassifiableSubset(array{physics_component{}})
Set2:classifiable_subset(component) =
    MakeClassifiableSubset(array{render_component{}})

Combined := Set1 + Set2
Combined.Contains[physics_component]  # true
Combined.Contains[render_component]   # true
```

For mutable sets, the Read/Write operations enable copying and updating:

```
Set1:classifiable_subset_var(component) = MakeClassifiableSubsetVar()
Set1.Add(physics_component{})

Set2:classifiable_subset_var(component) = MakeClassifiableSubsetVar()
Set2.Write(Set1.Read())  # Copy Set1's contents to Set2
```

Design Considerations

Several important constraints govern `classifiable_subset` usage:

The base type must be `<castable>` to enable runtime type queries. This requirement ensures that type checks can be performed efficiently.

You cannot subclass `classifiable_subset` types or create instances except through the designated constructor functions. This restriction maintains internal invariants required for correct type tracking.

Keys from one set cannot be used with a different set—they're bound to the specific set instance where the element was added.

The type parameter must be consistent across operations. You cannot add a `physics_component` to a `classifiable_subset(render_component)` even if both inherit from `component`:

```
render_set:classifiable_subset(render_component) = MakeClassifiableSubset()
physics_comp:physics_component = physics_component{}

```

```
# This would be a type error - physics_component is not a render_component
# render_set.Add(physics_comp)
```

Mutable sets require careful lifetime management. Keys become invalid when their corresponding instances are removed, and attempting to remove an already-removed key triggers a failure.

Performance characteristics matter for large type sets. While `Contains` queries are efficient due to the internal representation, operations like `FilterByType` may need to examine each type in the set.

When designing systems with `classifiable_subset`, consider whether immutable or mutable sets better fit your needs. Immutable sets provide stronger guarantees and work well for configuration, while mutable sets support dynamic systems where component types change frequently.

The hierarchy-aware semantics mean that adding a derived type makes queries for base types succeed. This is usually desirable but requires awareness—if you only want exact type matches, `classifiable_subset` may not be the right tool.

Chapter 14

Chapter 13: Access Specifiers

Access specifiers control visibility and accessibility of code elements. They provide a nuanced spectrum of access levels that reflect the complex reality of modern software development, particularly in the context of a persistent, global metaverse where code from many authors must coexist safely.

Five primary visibility levels are defined that form a carefully designed hierarchy, each serving specific architectural needs. Understanding when and why to use each level is crucial for creating well-structured, maintainable code.

Specifier	Visibility	Usage
`<public>`	Universally accessible	Members intended for external use
`<internal>`	Only within the module (default)	Module-private implementation
`<private>`	Only in immediate enclosing scope	Local to class/struct
`<protected>`	Current class and subtypes	Inheritance hierarchies
`<scoped>`	Current scope and enclosing scopes	Special use cases
`<epic_internal>`	Scopes with the /Verse.org, /UnrealEngine.com, and /Fortnite.com domains	`<epic_internal>` is only usable by Epic-authored code

14.1 Public

The `<public>` specifier represents the broadest level of access, making an identifier universally accessible from any code that can reference the containing module or type. When you mark something as public, you're making a strong commitment about its availability and stability:

```
player_manager<public> := module:
    MaxPlayers<public>:int = 100

    player<public> := class:
        Name<public>:string
        Level<public>:int = 1
```

Public members form the contract between your code and the outside world. In the metaverse context, public declarations are particularly significant because they represent guarantees that extend potentially forever—once published, removing or incompatibly changing a public member breaks the promise you've made to other developers who depend on your code.

The public specifier can be applied to modules, classes, interfaces, structs, enums, methods, and data members. When applied to a type definition itself, it makes the type available for use outside its defining module. When applied to members within a type, it makes those members accessible to any code that has access to an instance of that type.

14.2 Protected

The `<protected>` specifier creates a middle ground between public and private, allowing access within the defining class and any classes that inherit from it. This level exists specifically to support inheritance hierarchies while maintaining encapsulation:

```
game_entity := class:
    var Position<protected>:vector3 = vector3{x:=0.0, y:=0.0, z:=0.0}
    var Health<protected>:int = 100

    UpdatePosition<protected>(NewPos:vector3):void =
        set Position = NewPos
        OnPositionChanged()

    OnPositionChanged<protected>():void = {}  # Overridable by subclasses

player := class(game_entity):
    MoveToSpawn():void =
```

```
        UpdatePosition(GetSpawnLocation())  # Can access protected member
        set Health = MaxHealth              # Can modify protected variable
```

Protected access enables the template method pattern and other inheritance-based designs while preventing external code from accessing implementation details that should remain within the class hierarchy. This is particularly valuable for game entities and other hierarchical structures where parent classes need to share behavior with children without exposing that behavior to the world.

14.3 Private

The `<private>` specifier provides the strictest access control, limiting visibility to the immediately enclosing scope. Private members are truly internal implementation details that can be changed freely without affecting any external code:

```
inventory := class:
    var Items<private>:[]item = array{}
    var Capacity<private>:int = 20
    var CurrentWeight<private>:float = 0.0
    MaxWeight:float=20.0

    AddItem<public>(NewItem:item, At:int)<transacts><decides>:void =
        ValidateCapacity[NewItem]
        set Items[At] = NewItem
        set CurrentWeight = CurrentWeight + NewItem.Weight

    ValidateCapacity<private>(NewItem:item)<reads><decides>:void =
        Items.Length < Capacity
        CurrentWeight + NewItem.Weight <= MaxWeight
```

Private members are the building blocks of encapsulation. They allow you to maintain invariants, hide complexity, and create clean abstractions. Changes to private members never break external code, giving you the freedom to refactor and optimize implementation details as needed.

14.4 Internal

The `<internal>` specifier, which is the default access level when no specifier is provided, makes members accessible within the defining module but not outside it. This creates a natural boundary for collaborative code that needs to share implementation details without exposing them publicly:

```
physics := module:
    # Internal types and constants
    gravity_constant:float = 9.81

    collision_detector := class<abstract>:
        DetectCollision<internal>(A:game_entity, B:game_entity):?collision_info

    physics_world := class:
        var Entities<internal>:[]game_entity = array{}

        SimulateStep<internal>(DeltaTime:float):void =
            for (Entity : Entities):
                ApplyGravity(Entity, DeltaTime)
                CheckCollisions(Entity)
```

Internal access is ideal for module-wide utilities, shared implementation details, and helper functions that multiple classes within a module need but shouldn't be exposed to external code. It provides a clean separation between the module's public interface and its implementation machinery.

14.5 Scoped

The `<scoped>` specifier creates custom access boundaries between modules or code locations. Unlike the fixed visibility levels of `public`, `internal`, and `private`, `scoped` access allows you to explicitly grant access to particular modules while excluding all others—creating a kind of "friend" relationship between program entities.

14.5.1 Scoped Definitions

A scoped access level is created using the `scoped{...}` expression, which takes one or more module references:

```
collaboration := module:
    # Create a scope that includes both ModuleA and ModuleB
    Shared<public> := scoped{ModuleA, ModuleB}

    # This class is only accessible within ModuleA and ModuleB
    SharedResource<Shared> := class:
        Data<public>:int = 42
```

The scoped definition creates an access level that can then be used as a specifier on classes, functions, variables, and other definitions. Code within any of the listed entities can access the scoped member, while code outside those modules cannot—even if it can see the containing scope.

14.5.2 Cross-Module Collaboration

The most powerful use of scoped access is enabling controlled collaboration between modules. A definition can be created in one module but scoped to another, making it accessible where it's needed while keeping it hidden elsewhere:

```
graphics := module:
    # Define an interface scoped to the physics module
    CollidableShape<scoped{physics}> := interface:
        GetBounds():bounding_box

physics := module:
    using{graphics}

    # Physics can implement the interface even though it's defined in graphics
    sphere_collider := class<abstract>(CollidableShape):
        GetBounds<override>():bounding_box
```

This pattern allows graphics to define contracts that physics implements without exposing those implementation details publicly. The interface exists at the boundary between the two modules but isn't part of either module's public API.

You can scope a definition to multiple modules, creating a shared private space for collaboration:

```
gameplay := module:
    # This scope includes both the inventory and crafting modules
    SharedGameplayScope := scoped{inventory, crafting}

    # Items can be accessed by both inventory and crafting
    Item<SharedGameplayScope> := class:
        ID<public>:int
        Properties<public>:[string]string

    # Factory function available to both systems
    CreateItem<SharedGameplayScope>(TheID:int):Item = Item{ID:=TheID, Properties:=map{}}

inventory := module:
    using{gameplay}

    AddToInventory(ItemID:int):void =
        NewItem := CreateItem(ItemID)  # Can access scoped function
        # Implementation...

crafting := module:
```

```
    using{gameplay}

    CraftItem(Recipe:[]int)<decides>:Item =
        # Can create items and access their properties
        CreateItem(Recipe[0])
```

14.5.3 Scoped Read or Write Access

Like other access specifiers, scoped can be applied separately to read and write operations on variables:

```
SharedScope := scoped{ModuleA, ModuleB}

state_manager := class:
    # Public read access, but only ModuleA and ModuleB can write
    var<SharedScope> GameState<public>:game_state = game_state{}

    # Only ModuleA and ModuleB can read or write this internal state
    var<SharedScope> SyncCounter<SharedScope>:int = 0
```

This pattern is particularly useful for shared state that multiple modules need to coordinate on without exposing write access publicly.

14.5.4 Visibility and Access Paths

An important subtlety of scoped access is that it grants access to a specific member, but doesn't make intermediate types or modules visible. To access a scoped member, you must be able to see the entire path to it:

```
outer := module:
    # Internal to outer
    inner := module:
        # Scoped to target_module
        SharedClass<scoped{target_module}> := class:
            Value:int = 42

target_module := module:
    using{outer}

    # ERROR: Can't see outer.inner because inner is internal to outer
    # even though SharedClass is scoped to us
    UseShared():void = outer.inner.SharedClass{}
```

For scoped access to work, either the containing scope must be accessible (public or also scoped appropriately), or the scoped member must be accessed through a public interface that exposes it.

A definition can only have one scoped access level—you cannot apply multiple scoped specifiers:

```
# ERROR: Cannot have multiple access level specifiers
InvalidScope<scoped{ModuleA}><scoped{ModuleB}> := class{}
```

14.5.5 Scoped Access and Inheritance

When a class member has scoped access, overriding members in subclasses can maintain or narrow that access, following normal inheritance rules:

```
SharedScope := scoped{ModuleA, ModuleB}

base := class:
    # Accessible only in ModuleA and ModuleB
    ComputeValue<SharedScope>():int = 42

derived := class(base):
    # Can override with same or more restrictive access
    ComputeValue<override>():int = 100  # Now internal to this module
```

14.5.6 Using Scoped for API Boundaries

Scoped access excels at creating controlled API boundaries where certain functionality should be shared between specific modules but not exposed as part of the public interface:

```
networking := module:
    # Public scope for modules that need network access
    NetworkScope<public> := scoped{player_system, matchmaking, telemetry}

    # Core networking available to specific systems
    SendPacket<NetworkScope>(Data:[]uint8):void =
        # Implementation...

    # Internal statistics
    var<NetworkScope> BytesSent<NetworkScope>:int = 0
```

This creates an explicit architectural boundary—only the modules listed in the scope can access the networking primitives, while other code must use higher-level public APIs.

14.5.7 Design Considerations

Scoped access represents an architectural commitment between modules. When using it effectively:

- Use scoped for legitimate cross-module collaboration that doesn't belong in the public API
- Keep scope definitions at the module level where they can be documented and maintained
- Prefer scoping to explicit modules rather than deeply nested scopes
- Consider whether protected or internal access might be simpler for your use case
- Document why particular modules are included in a scope

The scoped specifier fills a unique niche between internal and public access, enabling sophisticated module architectures where multiple components need to collaborate intimately without exposing those implementation details to the wider codebase.

14.6 Separating Read and Write Access

An innovative feature is the ability to apply different access specifiers to reading and writing operations on the same variable. This fine-grained control allows you to create variables that are widely readable but narrowly writable, implementing common patterns like read-only properties elegantly:

```
game_state := class:
    # Public read, protected write
    var<protected> Score<public>:int = 0

    # Public read, private write
    var<private> PlayerCount<public>:int = 0

    # Internal read, private write
    var<private> SessionID<internal>:string
```

This dual-specifier system solves a common problem in object-oriented programming where you want to expose state for reading without allowing external modification. Rather than requiring getter methods or property syntax, Verse makes this pattern a first-class language feature.

The syntax places the write-access specifier on the `var` keyword and the read-access specifier on the identifier itself. This visual separation makes the access levels immediately clear when reading code. The write specifier must be at least

as restrictive as the read specifier — you cannot write to a variable that's privately readable but publicly writable, as this would violate basic encapsulation principles.

14.7 Best Practices

Understanding when to use each access level requires thinking about your code's architecture and evolution. The principle of least privilege suggests starting with the most restrictive access that works and only broadening it when necessary.

For public APIs, every public member is a commitment. Before making something public, consider whether it truly needs to be part of your module's contract or if it's an implementation detail that happens to be needed elsewhere temporarily. Public members should be stable, well-documented, and designed for longevity.

Protected access should be used thoughtfully in inheritance hierarchies. Not everything in a base class needs to be protected—only those members that form the inheritance contract between parent and child classes. Overuse of protected access can create tight coupling between classes in a hierarchy.

Private access is your default for implementation details. Most helper functions, intermediate calculations, and state management should be private. This gives you maximum flexibility to refactor and optimize without breaking dependent code.

The dual-specifier pattern for variables is particularly powerful for maintaining invariants. By making variables publicly readable but privately or protectively writable, you can expose state for observation while maintaining complete control over modifications:

```
resource_manager := class:
    var<private> TotalResources<public>:int = 1000
    var<private> AllocatedResources<public>:int = 0
    var<private> AvailableResources<public>:int = 1000

    AllocateResources<public>(Amount:int)<decides><transacts>:void =
        Amount <= AvailableResources
        set AllocatedResources = AllocatedResources + Amount
        set AvailableResources = AvailableResources - Amount
```

14.8 Annotations and Metadata

Verse provides an annotation system for attaching metadata to definitions using the `@` prefix syntax. Annotations provide compiler directives and metadata that affect how code is treated during compilation and evolution.

14.8.1 Built-in Annotations

@deprecated

Internal Feature

@deprecated attribute is currently an internal feature and cannot be used by end-users.

The `@deprecated` annotation marks definitions that should no longer be used. When code references a deprecated definition, the compiler produces a warning, alerting developers to update their code:

```
# Mark a definition as deprecated
@deprecated
OldFunction():void =
    Print("This function is deprecated")

# Mark a class as deprecated
@deprecated
legacy_player := class:
    Name:string

# Attempting to use deprecated code produces a warning
UseDeprecated():void =
    OldFunction()  # Warning: OldFunction is deprecated
```

Deprecated definitions can use other deprecated definitions without warnings, but non-deprecated code cannot use deprecated definitions without triggering warnings. This allows gradual migration of deprecated APIs:

```
@deprecated
OldAPI():int = 42

# Valid: deprecated can call deprecated
@deprecated
MigrateOldAPI():int = OldAPI()

# Warning: non-deprecated calling deprecated
# NewCode():int = OldAPI()
```

The `@deprecated` annotation can be applied to: - Functions and methods - Classes, interfaces, structs, and enums - Individual enum values - Data members - Modules

@experimental

Internal Feature

@deprecated attribute is currently an internal feature and cannot be used by end-users.

The `@experimental` annotation marks features that are not yet stable and may change or be removed in future versions. Experimental features can only be used when the `AllowExperimental` package flag is enabled:

```
# Mark a feature as experimental
@experimental
experimental_class := class:
    NewFeature:int

# Using experimental features requires AllowExperimental flag
# Without flag: error
# With AllowExperimental:=true: allowed
UseExperimental(Obj:experimental_class):void =
    Print("Using experimental feature")
```

Experimental definitions behave similarly to deprecated ones—experimental definitions can freely use other experimental definitions, but stable code cannot use experimental definitions unless the `AllowExperimental` flag is set.

The `@experimental` annotation cannot be applied to: - Local variables - Override methods (base method's experimental status is inherited)

@available

The `@available` annotation controls when a definition becomes available based on version numbers. This enables gradual API rollout and version-specific functionality:

```
using { /Verse.org/Native }  # Required for @available

# Available only in version 3000 and later
@available{MinUploadedAtFNVersion := 3000}
NewFeature():void =
    Print("New feature")

# Multiple definitions can coexist for different versions
@available{MinUploadedAtFNVersion := 2900}
OldImplementation():int = 42

@available{MinUploadedAtFNVersion := 3000}
NewImplementation():int = 100
```

The `@available` annotation can be applied to the same kinds of definitions as `@deprecated`.

14.8.2 Custom Attributes

Internal Feature

Custom attributes are currently an internal feature and cannot be created by end-users.

You can create custom attributes by inheriting from the special `attribute` class. Custom attributes allow you to attach domain-specific metadata to your code:

```
# Define a custom attribute
@attribscope_class
gameplay_element := class<computes>(attribute):
    Category:string
    Priority:int

# Use the custom attribute
@gameplay_element{Category := "Combat", Priority := 1}
weapon_system := class:
    Damage:int
```

Attribute Scopes

When defining custom attributes, you must specify where they can be applied using scope annotations:

- **@attribscope_class** - Can be applied to regular classes
- **@attribscope_attribclass** - Can be applied to attribute classes (classes that inherit from `attribute`)
- **@attribscope_enum** - Can be applied to enums
- **@attribscope_interface** - Can be applied to interfaces
- **@attribscope_function** - Can be applied to functions and methods
- **@attribscope_data** - Can be applied to data members

Example of scoped custom attributes:

```
# Attribute that can only be applied to functions
@attribscope_function
performance_critical := class(attribute):
    MaxExecutionTimeMs:int

# Attribute that can only be applied to data members
@attribscope_data
```

```
serializable_field := class(attribute):
    SerializationKey:string

# Use them appropriately
entity := class<abstract>:
    @serializable_field{SerializationKey := "entity_id"}
    ID:int

    @performance_critical{MaxExecutionTimeMs := 16}
    Update():void
```

Attempting to use an attribute in the wrong location produces a compiler error. For example, a function-scoped attribute cannot be applied to a class.

14.8.3 Getter and Setter Accessors

Internal Feature

Getter and setter accessors are currently an internal feature and cannot be used by end-users.

While not strictly annotations, the `<getter(...)>` and `<setter(...)>` specifiers provide a related form of metadata for controlling field access. These can be applied to both class and interface fields to define custom access logic:

```
entity := class:
    # External field with custom accessors
    var Health<getter(GetHealth)><setter(SetHealth)>:int = external{}

    var InternalHealth:int = 100

    GetHealth(:accessor):int = InternalHealth

    SetHealth(:accessor, NewValue:int):void =
        if (NewValue >= 0, NewValue <= 100):
            set InternalHealth = NewValue
```

Constraints on accessors:

- Must include both `<getter(...)>` and `<setter(...)>` - cannot have only one
- The field must have `= external{}` or no default value (with archetype initialization required)
- Fields with accessors cannot be overridden in subclasses
- The field must be mutable (marked with `var`)
- Not all types are supported for accessor fields

- Accessor fields are currently only allowed in epic_internal scopes

For more details on accessor patterns, see Fields with Accessors.

14.8.4 Localization

The `<localizes>` specifier marks definitions as localizable messages for internationalization. Localized messages use the `message` type and can be extracted for translation into different languages:

```
# Simple localized message
WelcomeMessage<localizes> : message = "Welcome to the game!"

# Call Localize to get the string
ShowWelcome():void =
    Print(Localize(WelcomeMessage))
```

Message Parameters

Localized messages can accept parameters for dynamic content interpolation:

```
# Message with parameter interpolation
GreetPlayer<localizes>(PlayerName:string) : message = "Hello, {PlayerName}!"

# Use with arguments
ShowGreeting(Name:string):void =
    Print(Localize(GreetPlayer(Name)))
    # Outputs: "Hello, Aldric!" (if Name = "Aldric")
```

Supported parameter types: - `string` - Text values - `int` - Integer values (formatted with comma separators) - `float` - Floating-point values

Parameter interpolation syntax: - Use `{ParameterName}` to insert parameter values - Parameters can be used multiple times or not at all - Only parameter names and Unicode code points allowed in braces

```
# Multiple parameters, some repeated
ScoreMessage<localizes>(Player:string, Score:int) : message =
    "Congratulations {Player}! Your score is {Score}. Great job, {Player}!"

# Outputs: "Congratulations Alice! Your score is 1,500. Great job, Alice!"

# Not all parameters required in message text
OptionalParam<localizes>(Name:string, Score:int) : message =
    "Thanks for playing!"  # Score parameter ignored
```

Integer Formatting

Integer parameters are automatically formatted with comma separators for readability:

```
HighScore<localizes>(Points:int) : message = "New record: {Points} points!"

# Localize(HighScore(190091)) produces: "New record: 190,091 points!"
```

Named and Default Parameters

Localized messages support named parameters and default values:

```
ConfigMessage<localizes>(?MaxPlayers:int = 8, ?TimeLimit:int = 300):message =
    "Game settings: {MaxPlayers} players, {TimeLimit} seconds"

# Can be called with any combination
Localize(ConfigMessage())                          # Uses defaults
Localize(ConfigMessage(?MaxPlayers := 16))         # Override one
Localize(ConfigMessage(?TimeLimit := 600, ?MaxPlayers := 32))  # Override both
```

Tuple Parameters

Messages can accept tuple parameters, which are destructured in the parameter list:

```
LocationMessage<localizes>(Player:string, (X:int, Y:int)) : message =
    "{Player} is at position ({X}, {Y})"

# Call with tuple
Localize(LocationMessage("Hero", (10, 20)))
# Outputs: "Hero is at position (10, 20)"
```

String Escaping and Unicode

Unicode code points:

```
UnicodeMessage<localizes> : message = "The letter is {0u004d}"
# Outputs: "The letter is M"
```

Escaped braces (to show literal braces):

```
EscapedMessage<localizes>(Name:string) : message =
    "Use \{Name\} to insert {Name}"
# Localize(EscapedMessage("value")) produces: "Use {Name} to insert value"
```

Special characters:

```
SpecialChars<localizes> : message =
    "Supports: \\r\\n\\t\\\"\\'\\#\\<\\>\\&\\~"
```

Whitespace and comments are allowed in interpolation:

```
SpacedParam<localizes>(Name:string) : message = "Hello { Name }"
CommentedParam<localizes>(Name:string) : message = "Hello { comment #>Name}"
```

Scope Requirements

Localized messages **must be defined at module or snippet scope**. They cannot be defined inside functions:

```
# Valid: module scope
MyModule := module:
    ModuleMessage<localizes> : message = "Valid"

# Valid: snippet scope
TopLevelMessage<localizes> : message = "Valid"

BadFunction():void =
    LocalMessage<localizes> : message = "Invalid"  # ERROR
```

Inheritance and Override

Localized messages can be overridden in class hierarchies:

```
base_ui := class:
    Title<localizes>:message = "Base Title"
    Description<localizes>:message = "Base description"

derived_ui := class(base_ui):
    # Override the title message
    Title<localizes><override>:message = "Derived Title"
    # Inherits Description from base
```

Localized messages can also be abstract:

```
quest_base := class<abstract>:
    # Abstract message - must be implemented by subclasses
    TaskDescription<localizes><public> : message
    # Concrete message with default
    CompletionMessage<localizes><protected> : message = "Quest complete!"

fetch_quest := class<final>(quest_base):
    TaskDescription<localizes><override> : message = "Collect 10 items"
```

Restrictions and Errors

Must use explicit type annotation:

The type annotation `: message` is required. Implicit typing is not supported:

```
# ERROR: Missing type annotation
# BadMessage<localizes> := "Text"  # ERROR 3639

# Valid: Explicit type
GoodMessage<localizes> : message = "Text"
```

RHS must be string literal:

```
# ERROR: Expression not allowed
# InvalidMessage<localizes> : message = "A" + "B"  # ERROR 3638

# Valid: Literal only
ValidMessage<localizes> : message = "AB"
```

Restricted parameter types:

Not all types are supported as parameters:

```
# ERROR: Optional types not supported
# OptionalMsg<localizes>(Player:?string) : message = "{Player}"  # ERROR 3509

# ERROR: Custom classes not supported
my_class := class{Value:int}
# ClassMsg<localizes>(Obj:my_class) : message = "{Object}"  # ERROR 3509
```

Interpolation syntax restrictions:

Only parameter names and Unicode code points are allowed inside `{}`:

```
# ERROR: Expressions not allowed
# ExprMessage<localizes>(Name:string) : message = "{"Hello"}"  # ERROR 3652

# Valid: Parameter names only
ParamMessage<localizes>(Name:string) : message = "{Name}"
```

Non-parameter identifiers are escaped:

If you reference an identifier that isn't a parameter, it gets escaped in the output:

```
GlobalName:string = "World"

RefMessage<localizes>(Greeting:string) : message =
    "{Greeting} to {GlobalName}"

```

```
# Localize(RefMessage("Hello")) produces: "Hello to \{GlobalName\}"
# Note: GlobalName is escaped because it's not a parameter
```

Access Specifiers

Localized messages support standard access specifiers:

```
my_module := module:
    PublicMessage<localizes><public> : message = "Public message"
    InternalMessage<localizes> : message = "Internal message"

    some_class := class:
        PrivateMessage<localizes><private> : message = "Private message"  # private not allowed
```

Best Practices

Keep messages translatable: - Use complete sentences, not fragments that might be concatenated - Avoid gender or number assumptions that don't translate well - Provide context through parameter names

Design for different languages: - Don't assume word order - let translators rearrange parameter positions - Allow repeated parameter use for languages that need it - Keep formatting codes (like comma separators) automated

Organization: - Group related messages in the same module - Use descriptive names that indicate message purpose - Consider using abstract base classes for message families

```
# Good: Clear, complete, flexible
PlayerJoined<localizes>(PlayerName:string, TeamName:string) : message =
    "{PlayerName} joined team {TeamName}"

# Avoid: Fragments that might be concatenated
# PlayerPrefix<localizes>(Name:string) : message = "Player {Name}"
# JoinedSuffix<localizes>(Team:string) : message = "joined {Team}"
```

14.9 Evolution

Access specifiers play a crucial role in code evolution. Changing access levels after publication can break compatibility:

- Narrowing access (public to private) breaks external code that depends on the member
- Widening access (private to public) is generally safe but creates new commitments

- Changing protected members affects the inheritance contract

The `<castable>` specifier on classes has special compatibility requirements—once published, it cannot be added or removed, as this would affect the safety of dynamic casts throughout the codebase.

When designing for long-term evolution, consider using internal access for members that might eventually become public. This allows you to test and refine APIs within your module before committing to public exposure.

Part IV

Part IV: Advanced Topics

Chapter 15

Chapter 14: Effects

Every function tells two stories. The first story, told through types, describes what data flows in and what data flows out. The second story, told through effects, describes what the function does along the way — whether it reads from memory, writes to storage, might fail, or could suspend execution. While most languages leave this second story implicit, Verse makes it explicit, turning side effects from hidden surprises into documented contracts.

Think about a simple game function that updates a player's score. In most languages, you'd see a signature like `UpdateScore(player, points)` and have to guess what happens inside. Does it modify the player object? Write to a database? Print to a log? Trigger animations? Without reading the implementation, you can't know. In Verse, effects are part of the signature itself, declaring upfront exactly what kinds of operations the function might perform.

This explicitness might seem like extra work at first, but it fundamentally changes how you reason about code. When you see `<reads>` on a function, you know it observes mutable state. When you see `<writes>`, you know it modifies that state. When you see `<decides>`, you know it might fail. These aren't comments or documentation that might be wrong — they're compiler-enforced contracts that must be accurate.

15.1 Understanding Effects

Effects represent observable interactions between your code and the world around it. Reading a player's health, updating a score, spawning a particle effect, waiting for an animation to complete — all these operations have effects that ripple beyond simple computation. Verse's effect system captures these interactions, making them visible and verifiable.

Consider this simple function that greets a player:

```
GreetPlayer()<transacts>:void =
    set CurrentGreeting = "Hello, adventurer!"
    Print(CurrentGreeting)
```

The `<transacts>` effect tells you immediately that this function modifies mutable state. You don't need to read the implementation to know that calling `GreetPlayer()` will change something in your program's memory. The effect is a promise about behavior, checked and enforced by the compiler.

Effects compose naturally through function calls. If function A calls function B, and B has certain effects, then A must declare at least those same effects (with some exceptions we'll explore). This propagation ensures that effects can't be hidden or laundered through intermediate functions — the true nature of an operation is always visible at every level of the call stack.

Why Effects Matter

Making effects explicit serves both human understanding and compiler optimization. For developers, effects act as documentation that can't lie. When you're debugging why a value changed unexpectedly, you can trace through the call chain looking only at functions with `<writes>`. When you're trying to understand why a function might fail, you look for `<decides>`. This isn't guesswork — it's guaranteed by the type system.

For the compiler, explicit effects enable powerful optimizations and safety guarantees. Pure functions marked `<computes>` can be memoized, their results cached because they'll always return the same output for the same input. Functions without `<writes>` can be safely executed in parallel without locks. Functions without `<decides>` can be called without failure handling.

The effect system also enforces architectural decisions. Want to ensure your math library remains pure? Mark its functions `<computes>`. Building a predictive client system that must run on players' machines? Use `<predicts>` to ensure no server-only operations sneak in. These aren't just conventions — they're compiler-enforced guarantees.

15.2 Effect Families and Specifiers

Verse organizes effects into families, each tracking a specific aspect of computation. Each family contains fundamental effects, and effect specifiers declare which effects a function may perform.

The six effect families are:

- **Cardinality**: Whether and how a function returns

- **Heap**: Access to mutable memory
- **Suspension**: Whether a function may suspend execution
- **Divergence**: Whether a function may run forever
- **Prediction**: Where a function runs
- **Internal**: Reserved for internal use

Some effects have no specifier, while some specifiers imply multiple effects. For instance, `<transacts>` implies `reads`, `writes` and `allocates`, and belongs to the Heap family.

Effect specifiers can be further divided into *exclusive* specifiers (`<converges>`, `<computes>`, `<transacts>`) and *additive* specifiers (`<suspends>`, `<decides>`, `<reads>`, `<writes>`, `<allocates>`). A function may have at most one exclusive specifier but can combine multiple additive ones. For example, `<computes><decides>` is valid (pure computation that may fail), but `<computes><transacts>` is an error (cannot have two exclusive effects).

Fundamental Effect	Effect Specifier	Effect Family	Effects implied by Specifier	Notes
succeeds		Cardinality		*No specifier; Must Succeed*
fails		Cardinality		*No specifier; Can Fail*
	`<decides>`	Cardinality	`{succeeds, fails}`	*Cannot combine with* `<suspends>`
reads	`<reads>`	Heap	`{reads}`	*Allows reading mutable states*
writes	`<writes>`	Heap	`{writes}`	*Allows writing mutable states*
allocates	`<allocates>`	Heap	`{allocates}`	*Allows allocation of mutable memory*

Fundamental Effect	Effect Specifier	Effect Family	Effects implied by Specifier	Notes
	`<transacts>`	Heap	`{reads, writes, allocates}`	*Exclusive; default*
	`<computes>`	Heap	`{}`	*Exclusive; Pure computation*
suspends	`<suspends>`	Suspension	`{suspends}`	*Cannot combine with* `<decides>`
diverges		Divergence	`{diverges}`	*No specifier; May run forever*
	`<converges>`	Divergence	`{}`	*Exclusive; Native functions only*
dictates		Prediction	`{dictates}`	*No specifier; Server Authority*
	`<predicts>`	Prediction	`{}`	*Allows Client Prediction*
no_rollback		Internal	`{no_rollback}`	*To be deprecated; Transactions disallowed*

The following restrictions are in effect:

- `<suspends>` and `<decides>` cannot be combined on the same function,
- `<converges>` is only allowed on `<native>` functions,
- duplicate specifiers (e.g., `<computes><computes>`) are errors.

15.3 How Effects Compose

Think of effect specifiers as setting bits in a bit vector: one bit per fundamental effect. Without any annotation, a function such as `GameUpdate` has the following effects:

```
GameUpdate():void = ...  # No explicit effects specified
```

dictates	suspends	reads	writes	allocates	succeeds	fails

This means it has effects `dictates`, `reads`, `writes`, `allocates` and `succeeds`. It's almost like writing `<dictates><transacts>` except we lack a way to say the function cannot fail.

As an aside: the absence of specifiers for `fails` and `succeeds` can be explained by the fact that a specifier like `<fails>` means the function always fails, never returns a value, and cannot have observable side effects (they would be undone by failure). The `succeeds` effect is implicit.

Annotating a function only affects the bits in that specifier's family. For example, function `CheckPlayerStatus` with the `<reads>` and `<predicts>` specifier:

```
CheckPlayerStatus()<reads><predicts>:string = ...
```

has the following effects:

dictates	suspends	reads	writes	allocates	succeeds	fails

Specifying `<reads>` clears the `writes` and `allocates` bits, and `<predicts>` clears the `dictates` bit, everything else is unchanged.

15.4 Effect Families in Detail

15.4.1 Cardinality effects

The cardinality family deals with whether functions return values successfully. Every function either succeeds (returning its declared type) or fails (producing no

value). Most functions always succeed — they're deterministic transformations that always produce output. But functions marked with `<decides>` can fail, turning failure into a control flow mechanism.

```
ValidateHealth(Health:float)<transacts><decides>:void =
    Health > 0.0      # Fails if health is zero or negative
    Health <= 100.0   # Fails if health exceeds maximum

# Usage
if (ValidateHealth[Player.Health]):
    # Health is valid, continue processing
    StartCombat()
```

The beauty of the decides effect is that it unifies validation with control flow. You don't check conditions and then act on them — the check itself drives the program's path.

15.4.2 Heap effects

The heap family governs access to mutable memory. This is perhaps the most important family for understanding program behavior, as it determines whether functions can observe or modify state.

The `<computes>` specifier marks pure functions — those that neither read nor write mutable state. These functions are deterministic: given the same inputs, they always produce the same outputs. They're the mathematical ideal of computation, transforming data without side effects.

```
CalculateDamage(BaseDamage:float, Multiplier:float)<computes>:float =
    BaseDamage * Multiplier
```

The `<reads>` effect allows functions to observe mutable state. They can see the current values of variables and mutable fields, but cannot modify them. This is useful for queries and calculations based on current game state.

```
player := class:
    Name:string
    var Health:float = 100.0
    var Score:int = 0

GetPlayerStatus(P:player)<reads>:string =
    if (P.Health > 50.0):
        "Healthy"
    else if (P.Health > 0.0):
        "Injured"
```

```
    else:
        "Defeated"
```

The `<writes>` effect permits modification of mutable state. Functions with this effect can use `set` to update variables and mutable fields. `<writes>` often requires `<reads>` as well, for instance when modification involves reading the current value.

In fact, the `set` instruction is by default `<transacts>` due to the addition of *live variables* to the language. A live variable is variable whose value depends on other variables; when one of those variables is updated by a `set` the live variable will be evaluated with potentially some `reads` and `allocates`.

```
HealPlayer(P:player, Amount:float)<transacts>:void =
    NewHealth := P.Health + Amount
    set P.Health = Min(NewHealth, 100.0)
```

The `<allocates>` effect indicates functions that create observably unique values — either objects marked `<unique>` or values containing mutable fields. Each call to such a function returns a distinct value, even if the inputs are identical.

```
game_entity := class<allocates>:
    ID:id
    var Position:vector3

CreateEntity(Pos:vector3)<allocates>:game_entity =
    game_entity{ID := GenerateID(), Position := Pos}
```

The `<transacts>` is the default for functions.

15.4.3 Suspension effects

The suspension family contains a single effect: `<suspends>`. Functions with this effect can pause their execution and resume later, potentially across multiple game frames. This is essential for operations that take time: animations, cooldowns, waiting for player input, or any multi-frame behavior.

```
PlayVictorySequence()<suspends>:void =
    PlayAnimation(VictoryDance)
    Sleep(2.0)  # Wait 2 seconds
    PlaySound(VictoryFanfare)
    Sleep(1.0)
    ShowRewardsScreen()
```

The `suspends` effect is viral — any function that calls a suspending function must itself be marked `<suspends>`. This ensures you always know which functions might take time to complete.

While `<suspends>` and `<decides>` cannot be combined on the same function, they have specific rules for how they interact across function calls. A `<suspends>` function can call a `<decides>` function, but *only within a failure context* using the square bracket `[]` syntax – this ensures that the failure is handled locally and doesn't propagate as a failure effect:

```
ValidateInput(Value:int)<decides><computes>:void =
    Value > 0
    Value < 100

ProcessAsync(Value:int)<suspends>:void =
    # Valid: calling decides function in failure context
    if (ValidateInput[Value]):
        # Process valid input
        DoAsyncWork()

# Invalid: calling decides function outside failure context
# ProcessAsync(Value:int)<suspends>:void =
#     ValidateInput(Value)  # ERROR: must use [] syntax
```

A `<suspends>` function can call another `<suspends>` function, but *must not use failure-handling syntax* like `?`:

```
AsyncOp()<suspends>:?int = false

CallAsync()<suspends>:void =
    # Valid: calling suspends function normally
    X := AsyncOp()

    # Invalid: cannot use ? with suspends in suspends context
    # if (Value := AsyncOp()?):
```

The asymmetry exists because `<suspends>` and `<decides>` represent fundamentally different control flow mechanisms—suspension is about time, while failure is about success/failure. Mixing their syntactic forms creates ambiguity about what's being handled.

15.4.4 Internal effects

[**Pre-release**]: The `<no_rollback>` effect is deprecated.

Prediction effects

Unreleased Feature

The `<predicts>` effect is not yet released.

The prediction family determines where code runs in a client-server architecture. By default, functions have the `dictates` effect, meaning they run authoritatively on the server. The `<predicts>` specifier allows functions to run predictively on clients for responsiveness, with the server later validating and potentially correcting the results.

```
HandleJumpInput()<predicts>:void =
    # Runs immediately on the client for responsiveness
    StartJumpAnimation()
    PlayJumpSound()

    # Server will validate and correct if needed
    PerformJump()
```

This enables responsive gameplay even with network latency, as players see immediate feedback for their actions while the server maintains authoritative state.

Divergence effects

Currently in planning, the divergence family will track whether functions are guaranteed to terminate. The `<converges>` specifier will mark functions that provably complete in finite time, while functions without it might run forever. This is particularly important for constructors and initialization code.

15.5 Effect Composition

Effects generally propagate up the call chain — a function must declare all the effects of the functions it calls. However, certain language constructs can hide specific effects, preventing them from propagating further.

An `if` expression hides `fails` effects in its failure context, thus failure failure in a condition does not propagate to the enclosing function:

```
SafeMod(A:int, B:int)<computes>:int =
    if (V:= Mod[A,B])  then V else 0
```

The `spawn` expression hides the `suspends` effect, allowing immediate functions to start asynchronous operations that continue independently:

```
Play()<suspends>:void =
        loop:
            PlayTrack(GetNextTrack())
            Sleep(180.0)

StartBackgroundMusic():void =  # No <suspends>
```

```
    spawn:
        Play() # Suspends effect hidden by spawn
```

As mentioned above failure is not allowed within `<suspends>` code including `spawn`. One way around this restriction is to use the `option` expression to convert failure into an optional value, transforming the `fails` effect into a regular value that can be handled without `<decides>`:

```
TryGetItem(Items:[]item, Index:int):?item =
    option{Items[Index]}  # Array access might fail, option catches it
```

The `defer` expression provides cleanup code that runs when exiting a scope, but has strict effect limitations:

- Cannot contain `<suspends>` operations—deferred code must execute synchronously
- Cannot contain `<decides>` operations—deferred code must always succeed

```
AcquireResource()<transacts>:resource = GetResource()
ReleaseResource(R:resource)<transacts>:void = {}

ProcessResource()<suspends>:void =
    R := AcquireResource()
    defer:
        ReleaseResource(R)  # Valid: transacts allowed in defer

    # Process resource with async operations
    DoAsyncWork()
```

These constraints ensure that cleanup code executes predictably and completely, without the possibility of suspension or failure that could leave resources in an inconsistent state.

15.6 Subtyping and Type Compatibility

Effect annotations create a subtyping relationship between function types. Understanding how effects interact with type compatibility is essential when storing functions in variables, passing them as parameters, or choosing between different implementations.

A function with **fewer effects** can be used where a function with **more effects** is expected. This is effect subtyping—a function that does less is compatible with a context that allows more:

```
# Pure function with only computes
PureAdd(X:int)<computes>:int = X + 1
```

```
3
4  # Variable that expects computes and decides
5  F:type{_(:int)<computes><decides>:int} = PureAdd
6
7  # Calling through the variable
8  Result := F[5]  # Must use [] syntax since type has <decides>
9  # Returns option{6} since PureAdd never fails
```

In this example, `PureAdd` has only `<computes>`, but it can be assigned to a variable expecting `<computes><decides>`. The pure function is a valid implementation of the failable interface—it simply never exercises the failure capability.

This principle applies to all effects:

```
1  # Function with <computes>
2  Compute(X:int)<computes>:int = X * 2
3
4  # Can assign to types expecting more effects
5  F1:type{_(:int)<computes><decides>:int} = Compute
6  F2:type{_(:int)<transacts>:int} = Compute
7  F3:type{_(:int)<reads>:int} = Compute
8
9  # All valid - Compute does less than what's allowed
```

When deciding subtyping, effects have the following impact:

- `<computes>` is a subtype of `<reads>`, `<transacts>`, and any combination with `<decides>`
- `<reads>` is a subtype of `<transacts>`
- Functions without `<decides>` are subtypes of functions with `<decides>`
- Functions without `<suspends>` are subtypes of functions with `<suspends>` (when compatible)

While you can add effects through subtyping, you **cannot remove** effects that a function actually has:

```
1  Validate(X:int)<computes><decides>:int =
2      X > 0
3      X
4
5  # ERROR: Cannot assign to type without <decides>
6  # F:type{_(:int)<computes>:int} = Validate
7  # The function CAN fail, but the type doesn't allow it
```

Similarly, functions with heap effects cannot be assigned to pure types:

```
counter := class:
    var Count:int = 0

Increment(C:counter)<transacts>:int =
    set C.Count = C.Count + 1
    C.Count

# ERROR: Cannot assign transacts function to computes type
# F:type{_(:counter)<computes>:int} = Increment
# The function writes state, type doesn't permit it
```

This restriction ensures type safety—the type signature is a promise about what effects the function might perform, and the actual function must honor that promise.

When you conditionally select between functions with different effects, the resulting expression has the union of all possible effects. This is *effect joining*—the compiler conservatively assumes the result might perform any effect that any branch could perform:

```
# Functions with different effects
PureFunction(X:int)<computes>:int = X + 1
FailableFunction(X:int)<computes><decides>:int =
    X > 0
    X + 1

# Conditional selection joins effects
SelectFunction(UseFailable:logic):type{_(:int)<computes><decides>:int} =
    if (UseFailable?):
        FailableFunction  # Has <computes><decides>
    else:
        PureFunction      # Has <computes>
    # Result type must account for both: <computes><decides>

# The returned function might fail (from FailableFunction)
# or might not (from PureFunction), so type must include <decides>
F := SelectFunction(true)
Result := F[5]  # Must use [] because result type has <decides>
```

Effect joining applies to all control flow that selects between functions:

```
Identity(X:int)<computes>:int = X

DecidesIdentity(X:int)<computes><decides>:int =
    X > 0
    X
```

```
6
7  TransactsIdentity(X:int)<transacts>:int = X
8
9  # Joining <computes> and <computes><decides>
10 F1:type{_(:int)<computes><decides>:int} =
11     if (true?):
12         Identity
13     else:
14         DecidesIdentity
15 # Result: <computes><decides> (union of effects)
16
17 # Joining <computes><decides> and <transacts>
18 F2:type{_(:int)<decides><transacts>:int} =
19     if (true?):
20         DecidesIdentity  # <computes><decides>
21     else:
22         TransactsIdentity  # <transacts>
23 # Result: <decides><transacts> (union of effects)
```

Effect subtyping enables flexible function parameters:

```
# Accepts any function that doesn't exceed <transacts><decides>
ProcessValues(
    Data:[]int,
    Transform(:int)<transacts><decides>:int
):[]int =
    for (Value:Data, Result := Transform[Value]):
        Result

# Can pass pure functions
ProcessValues(array{1, 2, 3}, PureAdd)

# Can pass failable functions
ProcessValues(array{1, 2, 3}, Validate)

# Can pass transactional functions
ProcessValues(array{1, 2, 3}, Increment)
```

Effect subtyping makes function composition work naturally:

```
Compose(
    F(:int)<computes>:int,
    G(:int)<computes>:int
):type{_(:int)<computes>:int} =
```

```
    Local(X:int)<computes>:int = G(F(X))
    Local

# If we want to allow more effects:
ComposeFlexible(
    F(:int)<transacts><decides>:int,
    G(:int)<transacts><decides>:int
):type{_(:int)<transacts><decides>:int} =
    Local(X:int)<transacts><decides>:int =
        if (IntermediateResult := F[X]):
            G[IntermediateResult]
        else:
            1=2; 0
    Local

# Can pass functions with fewer effects
ComposeFlexible(PureFunction, PureFunction)
ComposeFlexible(PureFunction, FailableFunction)
```

The following table summarize the interaction of effects and types:

Scenario	Valid?	Explanation
Assign `<computes>` to `<computes><decides>` type		Adding effects via subtyping
Assign `<computes>` to `<transacts>` type		Pure is subtype of transactional
Assign `<reads>` to `<transacts>` type		Reads is subtype of transactional
Assign `<computes><decides>` to `<computes>` type		Cannot remove `<decides>`
Assign `<transacts>` to `<computes>` type		Cannot remove heap effects
Select between `<computes>` and `<decides>`	Result: `<computes><decides>`	Effect joining
Select between `<reads>` and `<transacts>`	Result: `<transacts>`	Effect joining

These rules ensure that effect annotations remain trustworthy contracts—functions can do less than declared (subtyping), but never more, and conditional selection conservatively accounts for all possibilities (joining).

15.7 Effects on Data Types

Classes, structs, and interfaces can be annotated with effect specifiers, which apply to their constructors. This is particularly useful for ensuring that creating certain objects remains pure or has limited effects:

```
# Pure data structure - constructor has no effects
vector3 := struct<computes>:
    X:float = 0.0
    Y:float = 0.0
    Z:float = 0.0

# Entity that requires allocation due to unique identity
monster := class<unique><allocates>:
    Name:string
    var Health:float = 100.0
```

Classes and structs **cannot** be marked with `<suspends>` or `<decides>`:

```
# Valid effect specifiers for classes/structs:
valid_class := class<computes>{}
valid_struct := struct<transacts>{}

# Invalid: async and failable effects not allowed
# invalid_class := class<suspends>{}   # ERROR
# invalid_struct := struct<decides>{}  # ERROR
```

This restriction exists because constructors must complete synchronously and successfully. An object's construction cannot suspend across time boundaries or fail partway through—the object either exists fully formed or doesn't exist at all.

Field default values and block clauses in classes have strict effect requirements:

```
# Field initializers must use pure functions
HelperFunction()<transacts>:int = 42

# Invalid: field initializers cannot call transacts functions
# bad_class := class:
#     Value:int = HelperFunction()  # ERROR

# Block clauses must respect class effects
valid_class := class<transacts>:
    var Counter:int = 0
    block:
        set Counter = 1  # Valid: class has transacts

```

```
# Invalid: block effect exceeds class effect
# bad_class := class<computes>:
#     var Counter:int = 0
#     block:
#         set Counter = 1  # ERROR: computes class cannot write
```

Class member initializers and block clauses are implicitly restricted to have no more effects than what the class declares. This ensures that constructing an instance of the class respects the class's effect contract.

Limiting constructor effects helps maintain architectural boundaries. Data transfer objects can be kept pure with `<computes>`, ensuring they're just data carriers. Game entities might require `<allocates>` for unique identity, while service objects might need full `<transacts>` to initialize their state.

15.8 Working with Effects

When designing functions, start with the minimal effects needed and expand only when necessary. Pure functions with `<computes>` are the easiest to test, reason about, and compose. Add `<reads>` when you need to observe state, `<writes>` when you need to modify it, and `<decides>` when you need failure-based control flow.

Effects are part of your API contract. Once published, removing effects is a backwards-compatible change (your function does less than before), but adding effects is breaking (your function now does more than callers might expect). Design your effect signatures thoughtfully, as they become promises to your users.

Remember that over-specifying effects is allowed and sometimes beneficial. A function marked `<reads>` can be implemented as pure `<computes>` internally. This provides flexibility for future changes without breaking existing callers.

```
# API promises it might read state
GetDefaultWeapon<public>()<reads>:weapon =
    # But current implementation is pure
    weapon{Type := weapon_type.Sword, Dammage := 10}
```

Effect over-specification can future-proof APIs and avoid breaking changes later. For example, marking a currently pure function as `<reads>` allows you to add state observation in the future without breaking compatibility.

15.9 Backwards Compatibility

The effects of a function are part of what is checked for backwards compatibility. When updating a function that is part of a published API, the new version can have "fewer bits" but not more. So, a function that was marked as `<reads>` in

a previous version cannot be changed to `<transacts>`, but it can be refined to `<computes>`.

Effects transform side effects from hidden gotchas into visible, verifiable contracts. By making the implicit explicit, Verse helps you write more predictable, maintainable, and correct code. The effect system isn't a burden — it's a tool that helps you express your intent clearly and have the compiler verify that your implementation matches that intent.

Chapter 16

Chapter 15: Concurrency

Concurrency is a fundamental aspect of Verse, allowing you to control time flow as naturally as you control program flow. Unlike traditional programming languages that bolt on concurrency as an afterthought, Verse integrates time flow control directly into the language through dedicated expressions and effects.

Game development inherently requires managing multiple simultaneous activities. Think about a typical game scene: NPCs patrol their routes while particle effects play, UI elements animate as cooldown timers count down, and background music fades between tracks. All these activities happen concurrently, overlapping in time. Verse recognizes this reality and provides first-class language constructs to express these parallel behaviors naturally.

The language achieves this through a combination of structured and unstructured concurrency primitives, all built on the concept of async expressions that can suspend and resume across multiple simulation updates. This approach makes concurrent programming feel as natural as writing sequential code, while avoiding the traditional pitfalls of thread-based concurrency like data races and deadlocks.

16.1 Core Concepts

16.1.1 Immediate vs Async Expressions

Every expression falls into one of two categories: immediate or async. Understanding this distinction is crucial for working with Verse's concurrency model.

Immediate expressions evaluate with no delay, completing entirely within the current simulation update or frame. These include most basic operations you'd expect to happen instantly: arithmetic calculations, variable access, simple function calls, and data structure manipulation. When you write `X := 5 + 3`, the addition hap-

pens immediately, the assignment completes instantly, and execution moves to the next statement without any possibility of interruption.

Async expressions, on the other hand, have the possibility of taking time to evaluate, potentially spanning multiple simulation updates. They represent operations that inherently take time in the game world: animations playing out, timers counting down, network requests completing, or simply waiting for the next frame. An async expression might complete immediately if its conditions are already met, or it might suspend execution, allowing other code to run while it waits for the right moment to resume.

16.1.2 Simulation Updates

A simulation update represents one tick of the game's simulation, typically corresponding to a frame being rendered. Most games target 30 or 60 updates per second, creating the smooth motion players expect. Each update processes input, updates game logic, runs physics simulations, and prepares the next frame for rendering.

In networked games, the relationship between simulation updates and rendering becomes more complex. Multiple simulation updates might occur before rendering to maintain synchronization with the server, or updates might be interpolated to smooth out network latency. Verse's concurrency model abstracts these complexities, allowing you to think in terms of logical time flow rather than platform-specific timing details.

Async expressions naturally align with this update cycle. When an async expression suspends, it yields control back to the game engine, which continues processing other tasks and rendering frames. The suspended expression resumes in a future update when its conditions are met, seamlessly continuing from where it left off. This cooperative model ensures that long-running operations don't block the game's responsiveness.

16.1.3 The `suspends` Effect

Concurrent operations require the `<suspends>` effect specifier (see Effects). Functions marked with `<suspends>` can use concurrency expressions, call other suspending functions, and cooperatively yield execution:

```
# Function marked with suspends can use async expressions
MyAsyncFunction()<suspends>:void =
    Sleep(1.0)  # Pause execution
    Print("One second later!")

# Regular functions cannot use async expressions
MyImmediateFunction():void =
```

```
    # Sleep(1.0)  # ERROR: Cannot use Sleep without suspends
    Print("This happens immediately")
```

The `<suspends>` effect propagates through the call chain—any function calling a suspending function must itself be marked `<suspends>`.

16.2 Structured Concurrency

Structured concurrency represents one of Verse's most elegant design decisions. Rather than spawning threads or tasks that live independently and require manual lifecycle management, structured concurrency expressions have lifespans naturally bound to their enclosing scope. When you enter a structured concurrency block, you know that all concurrent operations within it will be properly managed and cleaned up when the block exits, preventing resource leaks and making code easier to reason about.

This approach mirrors how we think about sequential code. Just as a block of sequential statements has a clear beginning and end, structured concurrent operations have a defined lifetime. You can nest them, compose them, and reason about them using the same mental model you use for regular code blocks.

16.2.1 Effect Requirements

All structured concurrency expressions (`sync`, `race`, `rush`, and `branch`) require the `<suspends>` effect. You cannot use these constructs in immediate (non-suspending) functions:

```
# Valid: structured concurrency in suspends function
ProcessConcurrently()<suspends>:void =
    sync:
        Operation1()
        Operation2()

# Invalid: cannot use sync without suspends
# ProcessImmediate():void =
#     sync:  # ERROR: sync requires suspends
#         Operation1()
```

16.2.2 The sync Expression

The `sync` expression embodies the simplest concurrent pattern: doing multiple things at once and waiting for all of them to finish. When you have independent operations that can benefit from parallel execution, `sync` provides a clean way to express this parallelism while maintaining deterministic behavior.

```
# All expressions start simultaneously and must all complete
Results := sync:
    AsyncOperation1()  # Returns value1
    AsyncOperation2()  # Returns value2
    AsyncOperation3()  # Returns value3

Print("All operations complete with results: {Results(0)} {Results(1)} {Results(2)}")
```

Inside a `sync` block, all subexpressions begin execution at essentially the same moment. The sync expression then waits patiently for every single subexpression to complete, regardless of how long each takes individually. If one operation finishes in milliseconds while another takes several seconds, sync continues waiting until that last operation completes. Only then does execution continue past the sync block.

The beauty of sync lies in its predictability. You always get results from all subexpressions, always in the same order you wrote them, packaged neatly in a tuple. This makes sync perfect for scenarios where you need multiple pieces of data or need to ensure multiple systems are ready before proceeding. Loading game assets in parallel, initializing multiple subsystems simultaneously, or gathering data from multiple sources all benefit from sync's all-or-nothing approach.

Consider a more sophisticated example that demonstrates sync's composability:

```
# Nested blocks for complex operations
sync:
    block:  # Task 1 - sequential operations
        LoadTexture()
        ApplyTexture()
    block:  # Task 2 - parallel to task 1
        LoadSound()
        PlaySound()
    LoadModel()  # Task 3 - parallel to tasks 1 and 2

# Using sync results directly as function arguments
ProcessData(sync:
    FetchDataA()
    FetchDataB()
    FetchDataC()
)
```

16.2.3 The race Expression

Where `sync` embodies cooperation, `race` represents competition. The race expression starts multiple async operations simultaneously, but only cares about the

first one to cross the finish line. As soon as one subexpression completes, race immediately cancels all the others and continues with the winner's result. This winner-takes-all semantics makes race perfect for timeout patterns, fallback mechanisms, and any situation where you want the fastest possible response.

```
# First to complete wins, others are canceled
Winner := race:
    SlowOperation()     # Takes 5 seconds
    FastOperation()     # Takes 1 second - wins!
    MediumOperation()   # Takes 3 seconds

Print("Winner result: {Winner}")  # Prints FastOperation's result
```

The power of race becomes apparent when you consider real game scenarios. Imagine querying multiple servers for data, where you want to use whichever responds first. Or implementing a player action with a timeout, where either the player completes the action or time runs out. Race elegantly expresses these patterns without complex state management or manual cancellation logic.

Cancellation in race is immediate and thorough. The moment a winner emerges, all losing subexpressions receive a cancellation signal and begin cleanup. This isn't just an optimization; it's crucial for resource management and preventing unwanted side effects from operations that are no longer needed.

Type handling in race:

The type system handles race elegantly. Since only one subexpression's result will be returned, the result type of a race is the most specific common supertype of all the subexpressions. This ensures type safety while maintaining flexibility in what kinds of operations you can race against each other:

```
base_class := class:
    Value:int

derived_a := class(base_class):
    Name:string = "A"

derived_b := class(base_class):
    Name:string = "B"

GetA()<suspends>:derived_a = derived_a{Value := 1}
GetB()<suspends>:derived_b = derived_b{Value := 2}

# Result type is base_class (common supertype)
Result:base_class = race:
    GetA()  # Returns derived_a
```

```
    GetB()  # Returns derived_b
# Result is base_class, can hold either derived type

# If all expressions return the same type, that's the result type
SameTypeResult:int = race:
    block:
        Sleep(1.0)
        42
    block:
        Sleep(2.0)
        100
# Result type is int
```

A pattern involves adding identifiers to determine which subexpression won:

```
# Adding identifiers to determine which expression won
WinnerID := race:
    block:
        SlowOperation()
        1  # Return 1 if this wins
    block:
        FastOperation()
        2  # Return 2 if this wins
    block:
        loop:
            InfiniteOperation()
        3  # Never returns

case(WinnerID):
    1 => Print("Slow operation won somehow!")
    2 => Print("Fast operation won as expected")
    _ => Print("Impossible!")
```

16.2.4 The rush Expression

The `rush` expression occupies a unique middle ground between `sync` and `race`. Like race, it completes as soon as the first subexpression finishes. Unlike race, it doesn't cancel the losers. This creates an interesting pattern where you can start multiple operations, proceed as soon as one provides a result, while allowing the others to continue their work in the background.

```
# First to complete allows continuation, others keep running
FirstResult := rush:
    LongBackgroundTask()   # Continues after rush completes
```

```
    QuickCheck()          # Finishes first
    MediumTask()          # Also continues after rush

Print("First result: {FirstResult}")
# LongBackgroundTask and MediumTask are still running!
```

Rush shines in scenarios where you want to be responsive while still completing all operations eventually. Consider preloading game assets: you might start loading multiple levels simultaneously, begin gameplay as soon as the current level loads, while continuing to cache the other levels in the background. Or think about achievement checking, where you want to notify the player as soon as one achievement unlocks while continuing to check for others.

The non-canceling nature of rush requires careful consideration. Those background tasks continue consuming resources and performing their operations even after rush completes. They'll keep running until they naturally complete or until their enclosing async context ends. This makes rush powerful but also potentially dangerous if misused with operations that might never complete or that consume significant resources.

There's an important technical restriction to be aware of: rush cannot be used directly in the body of iteration expressions like `loop` or `for`. The interaction between rush's background tasks and iteration could lead to resource accumulation. If you need rush-like behavior in a loop, wrap it in an async function and call that function from your iteration.

16.2.5 The branch Expression

The `branch` expression represents fire-and-forget concurrency within a structured context. When you encounter a branch, it immediately starts executing its body as a background task and then, without any pause or hesitation, continues with the next expression. There's no waiting, no result collection, just a task spinning off to do its work while the main flow proceeds unimpeded.

```
branch:
    # This block runs independently
    AsyncOperation1()
    ImmediateOperation()
    AsyncOperation2()

# Execution continues immediately here
Print("Branch started, continuing main flow")
# Branch block is still running in background
```

Branch excels at handling side effects that shouldn't interrupt the main game flow. Think about logging player actions to analytics, triggering particle effects that play out over time, or starting background music that fades in gradually. These operations need to happen, but there's no reason to make the player wait for them to complete. Branch lets you express this "start it and move on" pattern directly.

The relationship between a branch and its enclosing scope maintains the structured concurrency guarantee. While the branch task runs independently, it's still tied to the lifetime of its parent async context. If that parent context completes, either naturally or through cancellation, the branch task is automatically canceled too. This prevents orphaned tasks from accumulating and consuming resources indefinitely.

Like rush, branch faces restrictions with iteration expressions. You cannot use branch directly inside a loop or for body, as this could lead to an unbounded number of background tasks. The workaround remains the same: encapsulate the branch in an async function and call that function from your iteration.

16.3 Unstructured Concurrency

16.3.1 The spawn Expression

While structured concurrency handles most concurrent programming needs elegantly, sometimes you need to break free from the hierarchical task structure. The `spawn` expression is Verse's single concession to unstructured concurrency, allowing you to start an async operation that lives independently of its creating scope. Think of spawn as an emergency escape hatch—powerful when needed, but not your first choice for typical concurrent patterns.

```
# spawn returns a task(t) object you can control
BackgroundTask:task(int) = spawn{LongRunningTask()}

# Or fire-and-forget without capturing the task
spawn{LongRunningTask()}
Print("Spawned task continues even after this scope exits")
```

What makes spawn unique is its ability to work anywhere. Unlike all the structured concurrency expressions that require an async context, spawn works in immediate functions, class constructors, module initialization—anywhere you can write code. This universality comes with responsibility. The task you spawn becomes a free agent, continuing its work regardless of what happens to the code that created it. There's no automatic cleanup, no parent-child relationship, just an independent task pursuing its goal.

The spawned function must have the `<suspends>` effect. You **cannot** spawn functions with the `<decides>` effect:

```
AsyncWork()<suspends>:void =
    Sleep(1.0)
    Print("Background work complete")

FailableWork()<decides>:void =
    false?  # Might fail

# Valid: spawning suspends function
spawn{AsyncWork()}

# Invalid: cannot spawn decides function
# spawn{FailableWork()}  # ERROR: spawn requires suspends, not decides
```

This restriction exists because spawned tasks run independently without a parent to handle their failure. Since `<suspends>` and `<decides>` cannot be combined on the same function, and spawn needs `<suspends>`, functions with `<decides>` cannot be spawned. If you need to spawn failable work, wrap it in a suspends function that handles the failure internally:

```
SafeFailableWork()<suspends>:void =
    if (FailableWork[]):
        Print("Work succeeded")
    else:
        Print("Work failed, but handled gracefully")

spawn{SafeFailableWork()}  # Valid - failure handled inside
```

The syntax deliberately constrains spawn to launching a single function call. You can't spawn a block of code with multiple operations; you're limited to spawning one async function. This constraint encourages you to think carefully about what you're spawning and encapsulate complex operations properly in functions rather than creating ad-hoc background tasks.

Spawn finds its place in specific architectural patterns. Global background services that monitor game state throughout the entire session, cleanup tasks that must complete even if the triggering context ends, or integration points where immediate code needs to trigger async operations—these scenarios justify reaching for spawn over the structured alternatives.

The contrast with branch illuminates the design philosophy. Branch gives you structured concurrency's safety within an async context, allowing multiple expressions in its body while maintaining parent-child relationships. Spawn trades these safeguards for the flexibility to work anywhere, but restricts you to a single func-

tion call. Each has its place, and choosing between them depends on whether you need structure or freedom.

Working with spawned tasks:

The `spawn` expression returns a `task(t)` object where `t` is the return type of the spawned function. This task object provides methods to control and query the spawned operation—you can cancel it, wait for it to complete, or check its current state. While spawn creates independent tasks that don't require management, having access to the task object gives you the power to intervene when needed. See the "The task(t) Type" section below for complete details on task objects and their capabilities.

16.4 The task(t) Type

The `task(t)` type represents a handle to an executing async operation, where `t` is the return type of the operation. While Verse creates tasks automatically behind the scenes for all async expressions, only `spawn` gives you direct access to a task object that you can control and query.

```
# spawn returns task(t) where t is the return type
BackgroundWork()<suspends>:int =
    Sleep(2.0)
    42

MyTask:task(int) = spawn{BackgroundWork()}
# MyTask is a handle to the spawned operation
```

Task objects provide a rich interface for managing async operations: you can cancel them, wait for their completion, and query their current state. This control is essential for implementing robust concurrent systems where you need to coordinate multiple independent operations.

A task moves through several distinct states during its lifetime:

Active: The task is currently running or suspended, but has not yet finished. It's still doing work or waiting to resume.

Completed: The task finished successfully and returned a result. Once completed, a task never changes state again. (Terminal state)

Canceled: The task was canceled before it could complete. This is a terminal state — canceled tasks cannot resume.

Settled: A task is settled if it has reached either the Completed or Canceled state. Settled tasks are no longer executing. (Terminal state)

Uninterrupted: A task is uninterrupted if it completed successfully without being canceled. This is equivalent to the Completed state. (alias)

Interrupted: A task is interrupted if it was canceled. This is equivalent to the Canceled state. (alias)

16.4.1 Task.Cancel()

Unreleased Feature

The Cancel() method has not be released at this time.

The `Cancel()` method requests cancellation of a task. This is a safe operation that can be called on any task in any state:

```
LongTask:task(void) = spawn{BackgroundWork()}

# Request cancellation
LongTask.Cancel()

# Safe to call multiple times
LongTask.Cancel()  # No error

# Safe to call on completed tasks (has no effect)
```

Cancellation is cooperative—the task doesn't stop immediately. Instead, it receives a cancellation signal that is checked at the next suspension point. The task then unwinds gracefully, allowing cleanup code to run. See "Suspension Points and Cancellation" below for details on when cancellation takes effect.

Calling `Cancel()` on an already completed task is safe and has no effect. This means you can cancel tasks without worrying about race conditions between completion and cancellation.

16.4.2 Task.Await()

The `Await()` method suspends the calling context until the task completes, then returns the task's result:

```
ComputeTask:task(int) = spawn{BackgroundWork()}

# Wait for task to complete and get result
Result:int = ComputeTask.Await()
Print("Task returned: {Result}")
```

Key behaviors of Await():

- **Blocks until completion**: If the task is still running, `Await()` suspends until it finishes
- **Returns immediately if complete**: If the task already finished, `Await()` returns the cached result instantly (Sticky)
- **Can be called multiple times**: You can await the same task repeatedly, always getting the same result
- **Propagates cancellation**: If the awaited task was canceled, `Await()` propagates the cancellation to the caller

```
MyTask:task(int) = spawn{ComputeValue()}

# First await - waits for completion
FirstResult := MyTask.Await()

# Second await - returns cached result immediately
SecondResult := MyTask.Await()

# FirstResult = SecondResult
```

16.4.3 Common Task Patterns

Canceling a task after timeout:

```
StartTask()<suspends>:void =
    DataTask:task(void) = spawn{ProcessData()}

    race:
        block:
            DataTask.Await()
            Print("Task completed")
        block:
            Sleep(5.0)
            DataTask.Cancel()
            Print("Task timed out and was canceled")
```

Waiting for multiple spawned tasks:

```
RunMultipleTasks()<suspends>:void =
    T1 := spawn{Task1()}
    T2 := spawn{Task2()}
    T3 := spawn{Task3()}

    # Wait for all to complete
    Results := sync:
        T1.Await()
```

```
        T2.Await()
        T3.Await()

    Print("All tasks complete: {Results(0)}, {Results(1)}, {Results(2)}")
```

16.4.4 Suspension Points and Cancellation

Task cancellation in Verse follows a cooperative model. Rather than forcefully terminating tasks, which could leave resources in inconsistent states, Verse sends cancellation signals that tasks check at **suspension points**. When a task receives a cancellation signal, it has the opportunity to clean up resources before terminating. This cooperative approach prevents data corruption while ensuring responsive cancellation.

Suspension points are the specific locations where async tasks can pause and resume. These are the only places where:

- A task can be suspended to allow other tasks to run
- Cancellation signals are checked and processed
- The runtime can switch between concurrent tasks

Common suspension points include:

Timing operations:

```
Sleep(1.0)  # Suspends for duration, checks cancellation when resuming
NextTick()  # Waits one simulation update, checks cancellation
```

Calling suspends functions:

```
Result := SomeAsyncFunction()  # Suspension point at the call
```

Structured concurrency expressions:

```
sync:  # Suspension point when entering sync
    Op1()
    Op2()
# Suspension point when sync completes
```

Task operations:

```
Result := MyTask.Await()  # Suspension point while waiting
```

Important: Immediate code between suspension points runs without interruption. If you write a long computation loop without any suspension points, that task cannot be canceled until it reaches the next suspension point:

```
# Cannot be canceled during the loop
LongComputation()<suspends>:void =
    for (I := 0..1000000):
        # No suspension points - runs to completion
        ComputeExpensiveOperation(I)
    Sleep(0.0)  # First cancellation check happens here!

# Can be canceled every iteration
ResponsiveComputation()<suspends>:void =
    for (I := 0..1000000):
        ComputeExpensiveOperation(I)
        Sleep(0.0)  # Cancellation checked every iteration
```

If you need to make long-running computations cancellable, insert periodic suspension points using `Sleep(0.0)` or `NextTick()`, which yield control without actual delay but allow cancellation checking.

Cancellation cascades through the task hierarchy. When a parent task is canceled, all its child tasks receive cancellation signals too. This cascading behavior maintains the invariant that child tasks don't outlive their parents in structured concurrency, preventing resource leaks and ensuring predictable cleanup. In a race expression, for example, when the winner completes, the race task sends cancellation signals to all losing subtasks, which then cascade to any tasks those losers might have created.

16.5 Cleanup and Resource Management

16.5.1 The defer: Block

The `defer:` block provides guaranteed cleanup code execution when a function scope exits, whether through normal completion, failure, or cancellation. Deferred blocks always execute during stack unwinding, making them the perfect tool for resource cleanup, logging, and finalization.

```
UseResourceSafely()<suspends>:void =
    Resource := AcquireResource()

    defer:
        # This ALWAYS runs when function exits
        ReleaseResource(Resource)
        Print("Resource released")

    ProcessWithResource(Resource)
    # defer block executes here on normal completion
```

Key defer: behaviors:

1. **Always executes on scope exit**: Whether the function returns normally, fails, or is canceled, the defer block runs
2. **Runs in reverse order**: Multiple defer blocks execute in LIFO order (last-in-first-out - most recent defer runs first)
3. **Captures current scope**: defer blocks close over variables from the enclosing scope
4. **Cannot be suspended**: defer blocks must execute immediately and cannot contain suspending operations

Multiple defer blocks execute in reverse order:

```
Print("Start")

defer:
    Print("First defer (runs last)")

defer:
    Print("Second defer (runs second)")

defer:
    Print("Third defer (runs first)")

Print("End")

# Output:
# Start
# End
# Third defer (runs first)
# Second defer (runs second)
# First defer (runs last)
```

This reverse ordering mirrors the natural stacking of resource acquisition—resources are released in the opposite order they were acquired, preventing dependency issues.

Common defer: patterns:

Resource cleanup:

```
ProcessFileWithCleanup(Path:string)<suspends>:void =
    FileHandle := OpenFile(Path)

    defer:
        CloseFile(FileHandle)
```

```
6
7       ProcessFile(FileHandle)
8       # File always closed, even on cancellation
```

State restoration:

```
1   TemporarilyModifyState()<suspends>:void =
2       OriginalState := SaveState()
3
4       defer:
5           RestoreState(OriginalState)
6
7       ModifyState()
8       # State restored to original value
```

Logging and debugging:

```
1   TrackedOperation()<suspends>:void =
2       Print("Operation starting")
3
4       defer:
5           Print("Operation finished (or canceled)")
6
7       Operation()
```

defer: with cancellation:

When a task is canceled, defer blocks execute as the stack unwinds from the cancellation point:

```
1   CancellableWork()<suspends>:void =
2       Setup()
3
4       defer:
5           Teardown()
6           Print("Cleanup after cancellation")
7
8       # If this task is canceled, defer runs during unwinding
9       LongOperation()
```

defer blocks **cannot** contain suspending operations. This ensures cleanup happens immediately without delay:

```
1   # ERROR: Cannot use suspending operations in defer
2   BadDefer()<suspends>:void =
3       defer:
```

```
        Sleep(1.0)  # ERROR: defer blocks cannot suspend
        NextTick()  # ERROR: defer blocks cannot suspend
```

This restriction is essential—if defer blocks could suspend, cleanup could be delayed indefinitely, defeating their purpose as guaranteed finalization.

defer: scope:

defer blocks belong to their enclosing function scope and execute when that function exits:

```
HelperFunction():void =
    defer:
        Print("Helper defer")
    Print("In helper")

OuterFunction()<suspends>:void =
    defer:
        Print("Outer defer")

    HelperFunction()  # Helper's defer runs immediately after HelperFunction returns
    Print("After helper")

    # Output:
    # Helper defer
    # After helper
    # Outer defer
```

Each function has its own defer stack. When a function returns, only its defer blocks execute, not those of calling functions.

16.6 Timing Functions

The fundamental timing function that suspends execution for a specified duration:

```
# Suspend for 1 second
Sleep(1.0)

# Suspend for one frame (smallest possible delay)
Sleep(0.0)
```

The `Sleep(0.0)` pattern deserves special attention. While it doesn't add actual delay, it serves two critical purposes:

1. **Creates a suspension point** for cancellation checking

2. **Yields control** to other concurrent tasks, preventing one task from monopolizing execution

This makes `Sleep(0.0)` essential for responsive concurrent code:

```
# Without Sleep(0.0) - cannot be cancelled during loop
UnresponsiveLoop()<suspends>:void =
    for (I := 0..10000):
        ExpensiveOperation(I)
    # Cancellation only checked after all iterations

# With Sleep(0.0) - responsive to cancellation
ResponsiveLoop()<suspends>:void =
    for (I := 0..10000):
        ExpensiveOperation(I)
        Sleep(0.0)  # Yields and checks cancellation each iteration
```

Best practice: Insert `Sleep(0.0)` in long-running loops to ensure tasks remain responsive to cancellation and share execution time fairly with other concurrent operations.

16.6.1 NextTick()

Unreleased Feature

NextTick() have not yet been released.

The `NextTick()` function suspends execution until the next simulation update (tick). Unlike `Sleep(0.0)` which yields control and may resume in the same tick if no other work is pending, `NextTick()` guarantees that at least one simulation update will occur before resuming:

```
# Wait for exactly one simulation tick
NextTick()

# Multiple ticks
NextTick()  # Wait 1 tick
NextTick()  # Wait another tick
NextTick()  # Wait a third tick
```

`NextTick()` is essential for game logic that needs to be synchronized with simulation updates:

```
# Process game logic every tick
GameLoop()<suspends>:void =
    loop:
        ProcessGameLogic()
```

```
        UpdatePhysics()
        CheckCollisions()
        NextTick()  # Wait for next simulation update

# Delay action by specific number of ticks
DelayByTicks(TickCount:int)<suspends>:void =
    for (I := 1..TickCount):
        NextTick()

# Wait 5 ticks before executing action
DelayByTicks(5)
PerformAction()
```

Sleep(0.0) vs NextTick():

Feature	Sleep(0.0)	NextTick()
Timing	May resume in same tick	Always waits for next tick
Use case	Yield for cancellation checks	Synchronize with simulation updates
Guarantee	Creates suspension point	Guarantees tick boundary

Both create suspension points for cancellation, but `NextTick()` provides stronger timing guarantees when you need to align with the simulation clock.

```
# Common patterns
LoopWithDelay()<suspends>:void =
    loop:
        ProcessFrame()
        Sleep(0.033)  # ~30 FPS

TickBasedLoop()<suspends>:void =
    loop:
        if (ProcessFrame()=false):
             break
        NextTick()  # Once per simulation tick
```

Timing Patterns are:

```
# Delayed action
PerformDelayedAction()<suspends>:void =
    Sleep(2.0)  # Wait 2 seconds
    DoAction()

```

```
# Periodic execution
PeriodicUpdate()<suspends>:void =
    loop:
        UpdateLogic()
        Sleep(1.0)  # Update every second

# Animation timing
AnimateMovement(Start:float,End:float)<suspends>:void =
    for (T := 0..10):
        SetPosition(Lerp(Start, End, Float(T)/10.0))
        Sleep(0.0)  # One frame
```

16.6.2 Getting Current Time: GetSecondsSinceEpoch

The `GetSecondsSinceEpoch()` function returns the current Unix timestamp—the number of seconds elapsed since January 1, 1970, 00:00:00 UTC. This function is essential for timestamping events, measuring durations, and synchronizing with external systems that use Unix time.

```
# Get current timestamp
CurrentTime := GetSecondsSinceEpoch()
# Returns something like 1716411409.0 (May 22, 2024)

# Log an event with timestamp
LogEvent(Message:string):void =
    Timestamp := GetSecondsSinceEpoch()
    Print("[{Timestamp}] {Message}")
```

Critical transactional behavior:

Within a single transaction, `GetSecondsSinceEpoch()` returns the **same value** every time it's called. This ensures deterministic behavior and prevents time-related race conditions:

```
MeasureTransactionTime()<transacts>:void =
    StartTime := GetSecondsSinceEpoch()

    # Perform complex operations
    DoExpensiveWork()
    PerformDatabaseUpdates()

    EndTime := GetSecondsSinceEpoch()

    # StartTime = EndTime!
```

```
    # Time is "frozen" within the transaction
    Duration := EndTime - StartTime  # Always 0.0
```

This transactional consistency is intentional—it prevents non-deterministic behavior where transaction retry could produce different results due to time progression. If the transaction fails and is retried, all calls to `GetSecondsSinceEpoch()` in the retried attempt will return a new consistent timestamp.

Measuring elapsed time across transactions:

To measure actual elapsed time, take timestamps in separate transactions:

```
game_timer := class:
    var StartTime:float = 0.0

    Start()<transacts>:void =
        set StartTime = GetSecondsSinceEpoch()

    GetElapsed()<transacts>:float =
        CurrentTime := GetSecondsSinceEpoch()
        CurrentTime - StartTime

Timer := game_timer{}
Timer.Start()

# Later, in a different transaction
ElapsedSeconds := Timer.GetElapsed()
# ElapsedSeconds reflects actual time passed
```

Use cases:

Event logging and debugging:

```
logger := class:
    var EventLog:[]tuple(float, string) = array{}

    Log(Message:string)<transacts>:void =
        Timestamp := GetSecondsSinceEpoch()
        set EventLog = EventLog + array{(Timestamp, Message)}

    GetRecentEvents(LastSeconds:float)<transacts>:[]string =
        Now := GetSecondsSinceEpoch()
        Cutoff := Now - LastSeconds
        for ((Time, Message) : EventLog, Time >= Cutoff):
            Message
```

Session tracking:

```
player_session := class:
    LoginTime:float

MakeSession()<transacts>:player_session =
    player_session{LoginTime := GetSecondsSinceEpoch()}

GetSessionDuration(Session:player_session)<transacts>:float =
    GetSecondsSinceEpoch() - Session.LoginTime
```

Rate limiting:

```
rate_limiter := class:
    var LastAction:float = 0.0
    Cooldown:float = 5.0  # 5 second cooldown

    CanAct()<transacts><decides>:void =
        Now := GetSecondsSinceEpoch()
        TimeSinceLastAction := Now - LastAction
        TimeSinceLastAction >= Cooldown
        set LastAction = Now

Limiter := rate_limiter{}

if (Limiter.CanAct[]):
    PerformAction()
else:
    ShowCooldownMessage()
```

Absolute timestamps for external systems:

When interfacing with external systems, databases, or APIs that use Unix timestamps:

```
# Timestamp for external analytics
AnalyticsEvent := map{
    "event_type" => "player_action",
    "timestamp" => GetSecondsSinceEpoch(),
    "player_id" => MyPlayerID
}
SendToAnalytics(AnalyticsEvent)

# Comparing with server timestamps
ServerTime := FetchServerTime()
```

```
LocalTime := GetSecondsSinceEpoch()
ClockSkew := LocalTime - ServerTime
```

Important notes:

- Returns `float` representing seconds (may have fractional parts for millisecond precision)
- Located in `/Verse.org/Verse` module—use `using { /Verse.org/Verse }` to access
- Not affected by `Sleep()` or other suspension—measures real-world time
- Consistent within transactions for determinism
- Each new transaction gets a fresh timestamp

Combining with Sleep for time-based logic:

```
# Wait until a specific time
WaitUntil(TargetTime:float)<suspends>:void =
    loop:
        if (GetSecondsSinceEpoch() >= TargetTime) then:
            break
        Sleep(0.1)  # Check every 100ms

# Schedule an action for the future
ScheduleDelayedAction(DelaySeconds:float)<suspends>:void =
    TargetTime := GetSecondsSinceEpoch() + DelaySeconds
    WaitUntil(TargetTime)
    PerformAction()
```

Note that the transactional consistency means you cannot use `GetSecondsSinceEpoch()` to measure time within a single transaction. For measuring execution time of operations that don't span transactions, use profiling tools or external timing mechanisms.

16.7 Events and Synchronization

Events provide synchronization primitives for coordinating between concurrent tasks. They implement producer-consumer and observer patterns, allowing tasks to signal each other and wait for specific conditions. Events bridge the gap between independent concurrent operations, enabling communication without shared mutable state.

16.7.1 Basic Events

The `event(t)` type creates a communication channel where producers signal values and consumers await them. Each signal delivers one value to each awaiting task:

```
# Create an event channel for integers
GameEvent := event(int){}

# Producer: signals values to the event
ProducerTask()<suspends>:void =
    Sleep(1.0)
    GameEvent.Signal(42)

# Consumer: awaits values from the event
ConsumerTask()<suspends>:void =
    Value := GameEvent.Await()
    ProcessValue(Value)

sync:
    ProducerTask()
    ConsumerTask()
```

When `Await()` is called on an event, the calling task suspends until another task calls `Signal()` with a value. The signaled value is delivered to one waiting task, and execution resumes. If multiple tasks await the same event, each `Signal()` wakes exactly one awaiter—signals and awaits pair up one-to-one.

This one-to-one matching makes events perfect for task coordination. Think of a player action system: the input handler signals button presses while the gameplay system awaits them. Or consider an AI pathfinding request: the game logic signals destination requests while the pathfinding system awaits and processes them.

Events work naturally with structured concurrency. You can use them within `sync` blocks to coordinate parallel operations, or combine them with `race` to implement timeouts on event waiting:

```
# Wait for event with timeout
Result := race:
    block:
        Value := GameEvent.Await()
        option{Value}
    block:
        Sleep(5.0)
        false  # Timeout - no value received
```

16.7.2 Sticky Events

Unreleased Feature

Sticky Events have not yet been released and is not currently available.

While basic events deliver each signal to exactly one awaiter, `sticky_event(t)` remembers the last signaled value and delivers it to all subsequent awaits until a new value is signaled:

```
StateEvent := sticky_event(int){}

# Signal once
StateEvent.Signal(100)

# Multiple awaits all receive the same value
Value1 := StateEvent.Await()  # Gets 100
Value2 := StateEvent.Await()  # Gets 100 again
Value3 := StateEvent.Await()  # Still gets 100

# New signal updates the sticky value
StateEvent.Signal(200)
Value4 := StateEvent.Await()  # Gets 200
Value5 := StateEvent.Await()  # Also gets 200
```

Sticky events excel at representing state changes that multiple consumers need to observe. Unlike basic events where each signal disappears after one await, sticky events maintain the current state. Consider a game phase system: when the phase changes from "Lobby" to "Playing", every system that checks the phase should see "Playing", not have one system consume the signal while others miss it.

The sticky behavior creates a form of eventually consistent state. If a task awaits a sticky event, it's guaranteed to see the most recent signal, even if that signal occurred before the await. This makes sticky events ideal for configuration updates, mode switches, or any scenario where "what's the current state?" matters more than "what just changed?".

16.7.3 Subscribable Events

Unreleased Feature

Subscribable Events have not yet been released and is not currently available.

The `subscribable_event` type implements the observer pattern, allowing multiple handlers to react to each signal. Unlike events where awaiting tasks explicitly wait, subscribable events let you register callback functions that execute automatically when values are signaled:

```
ScoreEvent := subscribable_event(int){}

# Subscribe multiple handlers
Logger := ScoreEvent.Subscribe(LogScore)
UIUpdater := ScoreEvent.Subscribe(UpdateUI)
AchievementChecker := ScoreEvent.Subscribe(CheckAchievements)

# Signal invokes all subscribed handlers
ScoreEvent.Signal(1000)  # Calls LogScore(1000), UpdateUI(1000), CheckAchievements(1000)

# Unsubscribe to stop receiving signals
Logger.Cancel()
ScoreEvent.Signal(2000)  # Only calls UpdateUI and CheckAchievements
```

Each subscription returns a `cancelable` object that lets you unsubscribe by calling `Cancel()`. Once canceled, that handler stops receiving signals. This provides fine-grained control over handler lifetimes, essential for systems that come and go during gameplay.

Subscribable events shine in broadcast scenarios where multiple independent systems need to react to the same occurrence. When a player scores points, the UI needs to update, the audio system needs to play a sound, the achievement system needs to check for unlocks, and the analytics system needs to log the event. With subscribable events, each system registers its handler independently, and every signal reaches all interested parties.

16.7.4 The awaitable and signalable Interfaces

Events are built on two fundamental interfaces that you can use to create custom synchronization types:

```
awaitable(t:type) := interface:
    Await()<suspends>:t

signalable(t:type) := interface:
    Signal(Value:t):void
```

The `awaitable` interface represents anything that can be waited on, while `signalable` represents anything that can send signals. By separating these capabilities, Verse enables precise control over who can produce values versus who can consume them.

You can pass `awaitable` parameters to functions that should only read from an event, preventing accidental signals:

```
# This function can only await, not signal
ConsumerFunction(Source:awaitable(int))<suspends>:void =
    Value := Source.Await()
    ProcessValue(Value)
    # Source.Signal(123)  # ERROR: awaitable doesn't have Signal

# This function can only signal, not await
ProducerFunction(Target:signalable(int)):void =
    Target.Signal(42)
    # Value := Target.Await()  # ERROR: signalable doesn't have Await
```

This separation creates clear interfaces for producer-consumer relationships. A queue implementation might expose an `awaitable` interface to consumers for reading and a `signalable` interface to producers for writing, ensuring neither side can accidentally use the wrong operation.

16.7.5 Transactional Behavior

Event subscriptions participate in Verse's transactional system. If a transaction containing a `Subscribe()` call fails and rolls back, the subscription never takes effect:

```
MyEvent := subscribable_event(int){}

# Subscription in a failing transaction
if:
    Sub := MyEvent.Subscribe(Handler)
    false?  # Transaction fails and rolls back

# Subscription was rolled back - handler not called
MyEvent.Signal(100)
```

Similarly, `Cancel()` operations are transactional. If you cancel a subscription within a transaction that later fails, the subscription remains active:

```
MyEvent := subscribable_event(int){}
Sub := MyEvent.Subscribe(Handler)

# Cancel in a failing transaction
if:
    Sub.Cancel()
    false?  # Transaction fails

# Cancel was rolled back - subscription still active
MyEvent.Signal(100)  # Handler still gets called
```

This transactional integration ensures that event subscriptions maintain consistency with other transactional operations. If you're setting up a complex system where subscribing to events is part of a larger initialization that might fail, the transaction system guarantees that either all initialization succeeds or none of it does, preventing partial setups that could cause subtle bugs.

16.7.6 Event Patterns and Use Cases

Request-Response: Use basic events to implement request-response patterns between systems:

```
PathRequest := event(tuple(int, int)){}  # (start, goal)
PathResponse := event(int){}             # path result

PathfindingService()<suspends>:void =
    loop:
        (Start, Goal) := PathRequest.Await()
        FindPath(Start, Goal)
        PathResponse.Signal(42)

RequestPath(Start:int, Goal:int)<suspends>:int =
    PathRequest.Signal((Start, Goal))
    PathResponse.Await()
```

State Broadcasting: Use sticky events for state that multiple systems need to observe:

```
PhaseChange := sticky_event(game_phase){}

# Systems await current phase without missing updates
UISystem()<suspends>:void =
    loop:
        Phase := PhaseChange.Await()
        UIUpdate(Phase)

AISystem()<suspends>:void =
    loop:
        Phase := PhaseChange.Await()
        AIUpdate(Phase)

AudioSystem()<suspends>:void =
    loop:
```

```
        Phase := PhaseChange.Await()
        AudioUpdate(Phase)
```

Multi-System Notifications: Use subscribable events when many systems need to react to the same events:

```
ItemPickedUp := subscribable_event(int){}

# Each system subscribes independently
InitializeSystems():void =
    ItemPickedUp.Subscribe(UpdateInventoryUI)
    ItemPickedUp.Subscribe(PlayPickupSound)
    ItemPickedUp.Subscribe(CheckCollectionAchievement)
    ItemPickedUp.Subscribe(LogItemPickup)

# Single signal reaches all systems
OnPlayerPickupItem(ItemID:int):void =
    ItemPickedUp.Signal(ItemID)
```

Events complement structured concurrency by providing communication channels that outlive individual concurrent operations. While `sync`, `race`, `rush`, and `branch` organize how tasks execute relative to each other, events organize how tasks share information and coordinate their actions.

16.8 Common Patterns and Best Practices

Implement operations with timeouts using `race`:

```
PerformWithTimeout()<suspends>:logic =
    race:
        block:
            ActualOperation()
            true  # Success
        block:
            Sleep(5.0)  # 5 second timeout
            false  # Timeout
```

Initialize multiple systems concurrently:

```
InitializeGame()<suspends>:void =
    sync:
        LoadAssets()
        ConnectToServer()
        InitializeUI()
```

```
        PrepareAudio()
    Print("Game ready!")
```

Start background tasks that don't block gameplay:

```
StartBackgroundSystems()<suspends>:void =
    branch:
        MonitorPlayerStats()
    branch:
        UpdateLeaderboards()
    branch:
        ProcessAchievements()
    # Main game continues while background tasks run
```

Spawn entities with delays:

```
SpawnWave(Enemies:[]enemy_class)<suspends>:void =
    for (Enemy : Enemies):
        spawn{Enemy.Spawn()}
        Sleep(0.5)  # Half second between spawns
```

16.9 Limitations and Considerations

16.9.1 Iteration Restrictions

The interaction between iteration and certain concurrency expressions requires careful consideration. Rush and branch cannot be used directly inside loop or for bodies, a restriction that prevents unbounded task accumulation. When you write a loop that might execute hundreds or thousands of times, allowing rush or branch directly would create that many background tasks, potentially overwhelming the system.

```
# Not allowed
for (I := 0..10):
    rush:  # ERROR: Cannot use rush in loop
        Operation1()
        Operation2()

# Workaround - wrap in function
ProcessWithRush(I:int)<suspends>:void =
    rush:
        Operation1()
        Operation2()

```

```
for (I := 0..10):
    ProcessWithRush(I)  # OK
```

This restriction forces you to be intentional about creating background tasks in iterations. By wrapping the concurrent operation in a function, you acknowledge the task creation and make it explicit in your code structure. This small friction prevents accidental resource exhaustion while maintaining the flexibility to use these patterns when genuinely needed.

16.9.2 Abstraction Over Implementation

Verse deliberately abstracts away the underlying threading and scheduling mechanisms. You won't find thread creation APIs, thread-local storage, or explicit synchronization primitives like mutexes or semaphores. This isn't a limitation but a design philosophy. By working with higher-level task abstractions, Verse eliminates entire categories of bugs—no data races, no deadlocks from incorrect lock ordering, no forgotten unlock calls.

The concurrency model is cooperative rather than preemptive. Tasks voluntarily yield control at suspension points rather than being forcibly interrupted by a scheduler. This cooperative nature makes reasoning about concurrent code easier since you know exactly where task switches can occur. It also integrates naturally with game engines' frame-based execution models, where predictable timing is crucial.

16.9.3 Effect Interactions

The effect system that makes Verse's concurrency safe also introduces some restrictions. The `decides` effect, which marks functions that can fail, cannot be combined with the `suspends` effect. This separation keeps the failure model and the concurrency model orthogonal, preventing complex interactions that would be difficult to reason about. Transactional operations and certain device-specific operations may also have restrictions when used in concurrent contexts, ensuring that operations that must be atomic remain so.

Chapter 17

Chapter 16: Live Variables

Unreleased Feature

Live variables have not yet been released. This chapter documents planned functionality that is not currently available.

Live variables represent a reactive programming paradigm in Verse, enabling variables to automatically recompute their values when dependencies change. Rather than requiring explicit callbacks or event handlers, live variables establish dynamic relationships between data, creating a declarative system where changes propagate naturally through your code.

Traditional programming requires manual tracking of dependencies and explicit updates when values change. If variable `A` depends on variable `B`, you must remember to update `A` whenever `B` changes, often through callback functions or observer patterns. Live variables eliminate this bookkeeping by automatically tracking which variables are read during evaluation and re-evaluating when those dependencies change. This creates more maintainable code where the intent—that `A` should always reflect some function of `B` — is expressed directly in the code itself.

Live variables build a foundation for reactive programming constructs, including `await`, `upon`, and `when`. Understanding live variables is essential for working with Verse's event-driven programming model, particularly for game development scenarios where many values must stay synchronized.

17.1 Live Expressions

A *live expression* establishes a dynamic relationship between a variable and a guard. Once established, the target is automatically re-evaluated whenever any of the guard's dependencies change, keeping the variable in sync.

```
var X:int = 0
var Y:int = 0
set live X = Y+1  # X now tracks Y
set Y = 5         # X automatically becomes 6
```

In the above, `set live X = Y+1` is a live expression, the target is the previously declared variable `X` and the guard is the expression `Y+1` with a dependency on variable `Y`.

Live variables extend mutable variables (see Mutability) with automated dependency tracking: any variable read during the evaluation of the guard expression is tracked. When any of those variables change, the guard is re-evaluated, and the target variable updates automatically.

17.1.1 Declaration Forms

Live variables can be declared in several ways, each suited to different use cases:

```
# Live variable declaration
var live X:int = Exp

# Live assignment to existing variable
var X:int = 0
# ... later ...
set live X = Exp

# Immutable live variable
live Y:int = Exp

# Variable with a function type (with <reads> effect)
var X: F = Exp            # Initial value computed normally
var live X: F = Exp       # Initial value tracked for dependencies

# Immutable variable with a function type (with <reads> effect)
X: F = Exp                # Initial value computed normally
live X: F = Exp           # Initial value tracked for dependencies

# Input-output variable pairs
var In->Out: F = Exp      # Initial value computed normally
var live In->Out: F = Exp # Initial value tracked for dependencies

In->Out: F = Exp          # Initial value computed normally
live In->Out: F = Exp     # Initial value tracked for dependencies
```

The most common form, `var live X = Exp`, creates a mutable variable whose initial value comes from evaluating the guard and subsequently updates whenever dependencies change. The guard expression can read other variables, and those reads are tracked to establish the dependency relationship.

The assignment form, `set live X = Exp`, converts an existing variable into a live variable by attaching a guard. This is useful when you need to make a variable reactive after initialization or conditionally based on program state.

Immutable live variables, declared with just `live Y = Exp`, cannot be directly written but still update automatically when their guard's dependencies change. This provides a read-only reactive value, useful for derived computations that should never be manually overridden.

When a variable's type is a function with the `<reads>` effect, the variable becomes live through its type (assignments are filtered through the function, and changes to the function's dependencies trigger recalculation). The `live` keyword in the declaration determines whether the initial expression `Exp` is also tracked for dependencies. Without `live`, `Exp` is evaluated once; with `live`, dependencies in `Exp` are tracked and can trigger updates before the first assignment.

Input-output pairs create two variables where one captures raw values and the other holds transformed values. Again, the `live` keyword controls whether the initial expression `Exp` is tracked for dependencies.

The following sections detail these more complex forms.

17.1.2 Functions as Types

Verse allows functions to be used as types for variables. When a function with the `<reads>` effect is used as a type, the variable automatically becomes live, updating whenever the function's dependencies change.

```
var Mult:int = 2

Multiply(Arg: int)<reads>:int = Arg * Mult

var X : Multiply

set X = 10        # X gets 20
set Mult = 1      # X gets 10
```

In this example, `Multiply` serves dual roles: it's both a function and a type for variable `X`.

As a type: When you declare `var X : Multiply`, several things happen:

- The storage type of `X` becomes `int` (the function's return type)
- Values assigned to `X` must be `int` (the function's parameter type)
- Each assignment passes through the function: `set X = 10` calls `Multiply(10)` and stores the result

As a live expression: Because `Multiply` has a `<reads>` effect (it reads mutable variable `Mult`), the variable declaration becomes a live expression with `Multiply` as its guard. This creates two ways the value changes:

1. **Direct assignment:** `set X = 10` filters the value through `Multiply`, storing 20
2. **Dependency updates:** `set Mult = 1` triggers recalculation, updating `X` to 10

This pattern elegantly combines transformation (every write is filtered) with reactivity (changes to dependencies trigger updates).

17.1.3 Input-Output Variables

Input-output variable pairs capture both raw input values and their transformed outputs. The syntax `var In->Out:F=Exp` creates two related variables where `Out` is the writable variable and `In` automatically stores the untransformed value before it passes through function `F`.

This pattern elegantly handles common game scenarios where values must stay within dynamic constraints. Consider health that must remain within bounds:

```
clamp := class:
    var Lower:int = 0
    var Upper:int = 100
    Evaluate(Value:int)<reads>:int =
        if (Value < Upper) then:
           if (Value > Lower) then Value else Lower
        else:
           Upper

Clamp := clamp{}
var BaseHealth->Health: Clamp.Evaluate = 50

# Health = 50 (clamped to [0, 100])
set Health = 75      # BaseHealth = 75, Health = 75
set Health = 120     # BaseHealth = 120, Health = 100 (clamped)
set Clamp.Upper = 60 # BaseHealth = 120, Health = 60 (reclamped)
```

When you write to `Health`, two things happen:

1. The raw value is stored in `BaseHealth`
2. The value is passed through `Clamp.Evaluate`, and the result is stored in `Health`

Because `Clamp.Evaluate` has a `<reads>` effect (it reads the mutable variables `Lower` and `Upper`), this becomes a live expression. When the constraints change, `Health` is automatically recalculated from `BaseHealth`.

How It Works

The declaration `var BaseHealth->Health: Clamp.Evaluate = 50` creates a live expression where:

- `BaseHealth` stores the raw input value (read-only from external perspective)
- `Health` stores the clamped value (read-write)
- `Clamp.Evaluate` is the transformation function with a `<reads>` effect

The object `Clamp` is an instance of class `clamp` with mutable bounds `Lower` and `Upper`. Because `Evaluate` reads these mutable variables, changes to them trigger recalculation:

- `set Health=75` — The value passes through unchanged, so both `BaseHealth` and `Health` become 75
- `set Health=120` — Exceeds `Upper`, so `BaseHealth` becomes 120 but `Health` becomes 100
- `set Clamp.Upper=60` — The constraint changes, triggering recalculation: `Health` updates to 60 while `BaseHealth` remains 120

Using an instance method like `Clamp.Evaluate` allows multiple independent clamps in the same context, each with its own dynamic bounds.

Access Control

The scope of input and output variables can be controlled independently by adding access specifiers: for example `var In<private>->Out<public>:t` makes the base value private while exposing the constrained value publicly.

17.1.4 Restricted Effects and Stability

Live variable guards cannot have `<writes>` or `<allocates>` effects. This fundamental restriction prevents side effects during guard evaluation, which Verse must be able to perform freely whenever dependencies change.

```
# ERROR: guard cannot write
var X:int = 0
var GlobalCounter:int = 0
set live X = block:
```

```
    set GlobalCounter += 1  # Not allowed!
    GlobalCounter
```

Live variables with interdependencies can form cycles. When target expressions use idempotent operations and values are comparable, these cycles can naturally converge to fixed points.

```
var X:int = 2
var Y:int = 2

set live X = if (Y < 0) then 0 else Y - 1
set live Y = if (X < 0) then 0 else X - 1

# Evaluates as: X=1, Y=0, X=-1, Y=0 (stable)
```

If the type of the variable is comparable, the guards are re-evaluated until values stabilize. In this example, `X` decrements to -1, `Y` clamps to 0, and `X` would recompute but produces -1 again, so the system stabilizes.

However, cycles without proper termination conditions can diverge. Verse cannot prevent all divergence—care must be taken when designing interdependent live variables.

This has a subtle implication: since any variable might become live after creation, reading any variable must be assumed to potentially trigger guard evaluation and, in the worst case, trigger a cycle. The effect system accounts for this: the `<writes>` effect implies `<diverges>` because any write might trigger cyclic live variable evaluation. The following illustrates a cyclic definition when `X` is larger than 0:

```
var X:int = 0
var live Y:int = if (X>0) then X+1 else 0

set live X = Y
set X = 1  # Error! Cyclic evaluation
```

17.1.5 Tracking Dependencies

Live variables track dependencies dynamically at runtime, not statically from source code. A variable becomes a dependency only when it's actually read during evaluation, not merely when it appears in the guard expression:

1. *Runtime tracking:* Dependencies are determined by which variables are actually accessed during each evaluation
2. *Transitive tracking:* Dependencies include variables read in called functions
3. *Dynamic changes:* The dependency set can change from one evaluation to the next

Consider this example:

```
var X:int = 1
var Y:int = 2
var Z:int = 3

SomeFun(Value:int):int =
   if(Value > 0) then X else Y

var live W:int = SomeFun(Z)   # W is 1, Dependencies: {Z, X}
set Z = 0                     # W is 2, Dependencies: {Z, Y}
```

Initially, `SomeFun(Z)` reads `Z` (which is 3) and evaluates the `then` branch, reading `X`, yielding `W=1` with dependencies `{Z, X}`.

After `set Z=0`, the change to `Z` triggers re-evaluation. Now `SomeFun(Z)` reads `Z` (which is 0) and evaluates the `else` branch, reading `Y`. This results in `W=2` with new dependencies `{Z, Y}`.

Notice how `Y` became a dependency only when the execution path changed. If `X` is subsequently modified, `W` will *not* update because `X` is no longer in the dependency set. This dynamic tracking ensures that live variables only react to changes that actually affect their current value.

17.1.6 Turning Off Liveness

A live variable established through its guard (not its type) can be turned off by a subsequent regular assignment.

```
var X:int = 0
var Y:int = 5
set live X = Y  # X is now live, tracking Y

set Y = 10      # X becomes 10
set X = 20      # X is now a regular variable again
set Y = 15      # X remains 20 (no longer tracking Y)
```

This allows temporary reactive behavior that can be disabled when no longer needed. However, variables that are live through their type expression remain live permanently—their reactive behavior is intrinsic to their type.

17.2 Reactive Constructs

Live variables form the foundation for three reactive constructs that handle asynchronous events without explicit callbacks: `await`, `upon`, and `when`.

17.2.1 The await Expression

The `await` expression suspends execution until a target expression succeeds, providing a synchronization primitive for asynchronous programming.

```
F()<suspends>:void =
    var X:int = 0

    OldX := X # copy the old value

    # Suspend until X changes from OldX (0)
    await{X <> OldX}
    Print("X changed to: {X}")
```

The target expression is evaluated immediately. If it fails, the task suspends. Verse tracks which variables were read during evaluation. Whenever those variables change, the guard is re-evaluated. If it succeeds, execution resumes immediately.

The practical implications are that you can write code that naturally expresses "wait for this condition" without manually managing event handlers or callback registration. The code suspends at the await point and resumes exactly when the condition becomes true.

```
# Wait for a specific condition
await{X.Contents > 10}
set Y.Contents = X.Contents * 2
```

The guard expression must have effects `<reads><computes><decides>` (see Effects)—it can read and compute but cannot write or allocate. This ensures re-evaluation is side-effect free.

17.2.2 The upon Expression

The `upon` expression provides one-shot reactive behavior: when a condition becomes true, execute some code once. Unlike `await`, which resumes the current task, `upon` creates a new concurrent task that runs when triggered.

```
var Health:int = 100
var IsDead:logic = false

upon(Health <= 0):
    set IsDead = true
    Print("Player died!")

set Health = 50  # Nothing happens
set Health = 0   # Triggers: prints "Player died!"
set Health = -10 # Nothing happens (already triggered once)
```

The `upon` expression evaluates its guard immediately and records the variables read. It then yields a `task(t)` where `t` is the result type of the body, representing the pending reactive behavior. When dependencies change, the guard is re-evaluated. If it succeeds, the body executes once in a new concurrent task, and the upon completes.

This one-shot behavior makes `upon` perfect for state transitions and event notifications. When a threshold is crossed, when a resource becomes available, when a timer expires—these scenarios naturally map to `upon`'s "fire once when condition becomes true" semantics.

The body must have the `<transacts>` effect (see Effects), allowing it to read and write variables (including other live variables), with execution guaranteed to be atomic with respect to notifications.

17.2.3 The when Expression

The `when` expression provides continuous reactive behavior: every time a condition is true, execute some code. This creates a persistent observer that runs whenever its guard succeeds.

```
var Score:int = 0
var DisplayedScore:int = 0

when(Score):
    set DisplayedScore = Score
    Print("Score updated to: {Score}")

set Score = 100  # Triggers: prints "Score updated to: 100"
set Score = 100  # No trigger (value unchanged)
set Score = 200  # Triggers: prints "Score updated to: 200"
```

The `when` expression evaluates its guard immediately. If the guard succeeds, the body executes. Then it records the variables read by the guard and yields a `task(void)`. Whenever dependencies change and the guard succeeds, the body executes again, creating a continuous observation loop.

This makes `when` ideal for maintaining derived state and responding to ongoing changes. Synchronizing UI with game state, updating AI behavior based on player actions, or maintaining consistency between related variables all benefit from `when`'s persistent reactivity.

```
var X:int = 2
var Y:int = 2

when(Y):
```

```
    Z := if (Y < 0) then 0 else Y - 1
    if (Z <> X):
        set X = Z

when(X):
    Z := if (X < 0) then 0 else X - 1
    if (Z <> Y):
        set Y = Z

# These when expressions will stabilize at X = -1, Y = 0
```

The body executes with the `<transacts>` effect, and the when immediately re-registers after each execution, creating the continuous observation pattern.

17.2.4 Cancellation

All three reactive constructs—`await`, `upon`, and `when`—return a `task` that can be canceled, allowing dynamic control over reactive behavior.

```
var X:int = 0
var Y:int = 0

Task := upon(X > 5):
    set Y = X

Task.Cancel()  # Cancels the reactive behavior
set X = 10     # Y remains 0
```

Canceling a task immediately removes all dependency tracking and prevents the associated code from running. This provides fine-grained control over the lifecycle of reactive behaviors, allowing you to enable and disable observations based on game state or user actions.

17.3 The batch Expression

The `batch` expression groups multiple variable updates together, delaying notifications until the entire group completes. This prevents intermediate states from triggering reactive behaviors and ensures observers see consistent snapshots of related changes.

```
var X:int = 0
var Y:int = 0

when(X > 1 and Y < 10):
```

```
    Print("Fired!") # Never prints

when(X):
    Print("X Changed to {X}!") # Prints once

batch:
    set X = 2
    set Y = 10
    set X += 5
    Print("Inside batch")

Print("After batch")

# Output order:
# -"Inside batch"
# -"X Changed to 7!"
# -"After batch"
```

Inside a `batch` block, variable updates occur immediately but notifications to awaiting tasks and reactive constructs are deferred. When the batch completes, all pending notifications fire in the order their triggers occurred, but observers see the final consistent state rather than intermediate values.

If the same notification occurs twice, only the first of them will be delivered.

Batch expressions nest: notifications are delayed until all enclosing batches complete. This composability ensures that no matter how deeply nested your code, you can guarantee atomic updates of related variables.

The body of a batch must not have the `<suspends>` effect—all operations must complete immediately. This ensures batch blocks have well-defined boundaries and can't leave the system in an inconsistent state by suspending mid-update.

17.4 Issues and Patterns

17.4.1 API Design

Any variable appearing in the public interface of a class or module can be made live by external code, potentially violating class invariants. To avoid this, one could limit the exposure of mutable variables or at least use access modifiers to control this:

```
var<private> live X<public>:int = Exp
```

Here `X` is publicly visible for reading but can only be updated by the class itself. This prevents external code from attaching arbitrary guards that might break the class's invariants.

17.4.2 Failures and Liveness

Live variable updates and reactive construct triggers are integrated in the failure semantics of Verse. When there is a failure, live variable updates are rolled back and their notifications are suppressed.

```
var X:int = 0
var Y:int = 0

if:
    set live X = Y + 5  # Establishes live relationship
    false?          # Transaction fails

upon(X):
    Print("{X}") # Does not print when Y changes

# Live relationship was not established
set Y = 10  # X remains 0
```

This ensures that reactive behaviors only observe committed changes, maintaining consistency even in the presence of speculative execution and failure.

17.4.3 Derived Synchronization

A common pattern is for multiple UI elements to reflect the same game state, `when` provides automatic synchronization:

```
var PlayerScore:int = 0
var DisplayedScore:int = 0
var ScoreText:string = ""

when(PlayerScore):
    set DisplayedScore = PlayerScore
    set ScoreText = "Score: {PlayerScore}"
```

Every change to `PlayerScore` automatically updates both the numeric display value and the formatted text, keeping the UI consistent without manual coordination.

17.4.4 Conditional Reactivity

Live variables can track different sources based on conditions:

```
var UseAlternate:logic = false
var PrimaryValue:int = 10
var AlternateValue:int = 20
var CurrentValue:int = 0

set live CurrentValue =
    if (UseAlternate) then AlternateValue else PrimaryValue

# CurrentValue = 10
set UseAlternate = true
# CurrentValue = 20
set AlternateValue = 30
# CurrentValue = 30
set PrimaryValue = 15
# CurrentValue = 30 (still tracking AlternateValue)
```

The dependency tracking is dynamic: when the condition changes, the set of tracked variables changes accordingly, allowing flexible reactive routing.

17.4.5 Resource Loading

Use `upon` for one-time initialization when resources become available:

```
ResourceManager := class:
    var TextureLoaded:logic = false
    var ModelLoaded:logic = false

    Initialize()<suspends>:void =
        upon(TextureLoaded and ModelLoaded):
            Print("All resources loaded, starting game")
            StartGame()
```

This pattern eliminates manual tracking of loading state. When both resources finish loading, the game starts automatically.

17.4.6 Modifier Stack (Under Consideration)

The design of modifier_stack has not been finalized; material presented here is likely to change.

Game development often requires applying multiple modifiers to a single value. For instance, a player's health might need to be clamped to a valid range, temporarily boosted by a health potion and automatically recomputed when dependencies change.

The `modifier_stack` pattern provides a composable solution using live variables and function-as-type, allowing ordered transformations that automatically update when any modifier's dependencies change.

The modifier stack consists of three components:

1. `modifier_ifc(t)` - An interface for modifiers that transform values of type `t`
2. `modifier_stack(t)` - A container that orders and composes modifiers
3. **Live variable** - Uses `modifier_stack.Evaluate` as its type for automatic reactivity

When you assign to a live variable with a modifier stack type, the value flows through each modifier in position order, and the final result is stored. Because `modifier_stack.Evaluate` has the `<reads>` effect, changes to any modifier's dependencies (or adding/removing modifiers) trigger automatic recalculation.

The public API is as follows:

```
modifier_ifc(t : type) := interface:
   Evaluate(Value:t)<reads> : t

modifier_stack(t:type) := class:
   # Insert a Modifier at Position; return a cancelable used to remove the Modifier.
   AddModifier<final>(Modifier:modifier_ifc(t), Position:rational)<transacts>: cancelable

   # Returns the input Value evaluated against each modifier in the stack in position order.
   Evaluate<final>(Value:t)<reads> : t
```

The `AddModifier` method returns a `cancelable` which can be used to remove the inserted modifier. Removing a modifier triggers recalculation of any live variable associated with this stack.

For example, consider the following which creates a live variable `Health` filtered through a modifier stack containing a magic potion modifier that doubles the input value:

```
HealthStack := modifier_stack(float){}
HealthStack.AddModifier(magic_potion{Value:=2.0})
var RawHealth -> Health : HealthStack.Evaluate = 10.0
# RawHealth = 10.0, Health = 20.0
```

The variable automatically recomputes when the multiplier changes or when modifiers are added to the stack.

In more detail, this example demonstrates two modifiers working together: a `magic_potion` that multiplies health, and a `clamp` that bounds values within a range.

```
# Define modifier implementations
magic_potion := class(modifier_ifc(float)):
   var Value:float
   Evaluate<override>(Arg:float)<reads>:float = Arg * Value

clamp := class(modifier_ifc(float)):
   var Low:float
   var High:float
   Evaluate<override>(Arg:float)<reads>:float =
       if (Arg<Low) then Low else { if (Arg>High) then High else Arg }

# Create instances
Potion := magic_potion{ Value:= 2.0 }
Clamp := clamp{Low:=1.0, High:= 12.0 }

# Build the modifier stack
HealthStack := modifier_stack(float){}
RevokePotion := HealthStack.AddModifier(Potion, 0.0)  # Apply first (position 0.0)
HealthStack.AddModifier(Clamp, 1.0)                   # Apply second (position 1.0)

# Create live variable with modifier stack
var Health : HealthStack.Evaluate = 5.0  # 5.0 * 2.0 = 10.0 (then clamped to [1.0, 12.0])
set Potion.Value = 3.0                   # 5.0 * 3.0 = 15.0 (clamped to 12.0)
RevokePotion.Cancel()                    # 5.0 (no potion, just clamp to [1.0, 12.0])
```

The value flows through modifiers in position order:

1. **Initial:** 5.0 → Potion (×2.0) → 10.0 → Clamp → 10.0
2. **After changing** `Potion.Value`**:** 5.0 → Potion (×3.0) → 15.0 → Clamp → 12.0
3. **After removing potion:** 5.0 → Clamp → 5.0

There are plans to enforce via the compiler that: each modifier instance can only be added to one stack, and each stack instance can be associated with one variable. This will enable future features where modifier stacks maintain state specific to their associated live variable.

17.4.7 Common Errors

Unnecessary Live Declarations

Defining a live variable with no dependencies that can change is unnecessary and misleading:

```
var live X:int = 10    # X is 10 and will never change
set live X = 20        # X is 20 and will never change
```

In both cases, `X` does not update automatically, so the program behaves identically without the `live` keyword. The `live` annotation falsely suggests reactive behavior where none exists.

Missing Mutable Dependencies

Similarly, a live variable that only depends on immutable values will never update:

```
X:int = 10
var live Y = X+1    # Y is 11 and will never change
```

Since `X` is immutable, `Y` has no mutable dependencies and will remain at 11 forever. The `live` declaration is pointless.

Function-as-Type Confusion

A subtle error occurs when trying to make a variable live through a function type:

```
var Mult:int = 10

Multiply(Value:int):type{_(:int):int} =
    Fun(Arg:int):int = Value * Arg
    Fun

var X:Multiply(Mult) = 10    # X = 100

set Mult = 20                 # X is still 100 (not live!)
```

This code is mistaken. The programmer likely thought that `Multiply(Mult)` would make `X` live because the expression has a `<reads>` effect (it reads `Mult`) and returns a function type `int->int`.

The error: For a variable to be live through its type, the *returned function itself* must have the `<reads>` effect, not the expression that produces the function.

To see why, consider this equivalent transformation:

```
MFun = Multiply(Mult)
var X:MFun = 10
```

Now it's clear that `X` is not live—`MFun` is just a function value with type `int->int`, and that function does not have a `<reads>` effect.

The correct approach: Use the pattern where the function used as a type directly has the `<reads>` effect:

```
var Mult:int = 10

Multiply(Arg:int)<reads>:int = Arg * Mult

var X:Multiply = 10    # X = 100
set Mult = 20          # X = 200 (now live!)
```

Here `Multiply` itself has `<reads>`, so using it as a type makes `X` live.

If the same function has to be reused with different variables as dependent, one can package it in an object as shown earlier.

17.5 Evolution

When publishing a new version of a system, it is allowed to remove `live` from a variable definition. This forward compatibility guarantee means that reactive behavior is an implementation detail that can be optimized away without breaking client code.

Converting a regular variable to a live variable in a new version is generally safe if the computed value matches what the previous version maintained manually. However, if external code depends on being able to set arbitrary values, this could break expectations.

The ability to cancel reactive constructs provides an important upgrade path: code that creates `when` or `upon` observers can later be modified to cancel them under different conditions without breaking existing behavior.

Chapter 18

Chapter 17: Modules and Paths

Modules and paths are fundamental concepts for code organization, namespace management, and the ability to share and reuse code across projects. Think of modules as containers that group related functionality together, similar to packages in other programming languages, but with stronger guarantees about versioning and compatibility.

In the context of game development, modules allow you to separate different aspects of your game logic into manageable, reusable pieces. For example, you might have one module for player inventory management, another for combat mechanics, and yet another for UI interactions. Each module encapsulates its own functionality while exposing only the necessary interfaces to other parts of your code.

The module system is designed to support the vision of a persistent, shared Metaverse where code can be published once and used by anyone, anywhere, with confidence that it will continue to work even as the original author updates and improves it. This is achieved through strict backward compatibility rules and a global namespace system that ensures every piece of published code has a unique, permanent address.

Each module is intrinsically linked to the file system structure of your project. When you create a folder in your Verse project, that folder automatically becomes a module. The module's name is simply the folder's name, making the relationship between your file organization and your code organization completely transparent.

All `.verse` files within the same folder are considered part of that module and share the same namespace. This means that if you have three files - `player.verse`, `inventory.verse`, and `equipment.verse` - all in a folder called `player_systems`, they all contribute to the `player_systems` module and can reference each other's definitions without any import statements. This automatic grouping makes it easy to split

large modules across multiple files for better organization while maintaining the logical unity of the module.

18.1 Paths

Paths are the addressing system that makes Verse's vision of a shared, persistent Metaverse possible. Just as every website on the internet has a unique URL, every module has a unique path that identifies it globally. This path system is more than just a naming convention - it's a fundamental part of how Verse manages code distribution, versioning, and dependencies.

Paths borrow conceptually from web domains with adaptations for the needs of a programming language. A path starts with a forward slash `/` and typically includes a domain-like identifier followed by one or more path segments. This creates a hierarchical namespace that is both human-readable and globally unique.

The format `/domain/path/to/module` serves several important purposes:

- **Persistent and unique identification**: Once a module is published at a particular path, that path belongs to it forever. No other module can ever claim the same path, ensuring that dependencies always resolve to the correct code.
- **Ownership and authority**: The domain portion of the path (like `Fortnite.com` or `Verse.org`) indicates who owns and maintains the module. This helps developers understand the source and trustworthiness of the code they're using.
- **Discoverability**: Because paths follow a predictable pattern, developers can often guess or easily find the modules they need. Documentation and tooling can also leverage this structure to provide better discovery experiences.
- **Hierarchical organization**: The path structure naturally supports organizing related modules together. For example, all UI-related modules might live under `/YourGame.com/UI/`, making them easy to find and understand as a group.

Epic Games provides several standard modules that are commonly used:

- `/Verse.org/Verse` - Core language features and standard library functions
- `/Verse.org/Random` - Random number generation utilities
- `/Verse.org/Simulation` - Simulation and timing utilities
- `/Fortnite.com/Devices` - Integration with Fortnite Creative devices
- `/UnrealEngine.com/Temporary/Diagnostics` - Debugging and diagnostic tools
- `/UnrealEngine.com/Temporary/SpatialMath` - 3D math and spatial operations

The use of "Temporary" in some paths indicates that these modules are provisional and may be reorganized in future versions of Verse. This naming convention helps set expectations about the stability of the API.

When you create your own modules, they can exist at various levels of the path hierarchy:

- `/YourGame/` - Top-level module for your game
- `/YourGame/Player/` - Player-related functionality
- `/YourGame/Player/Inventory/` - Specific inventory management
- `/pizlonator@fn.com/NightDeath/` - Personal or experimental modules

The ability to include email-like identifiers (such as `pizlonator@fn.com`) allows individual developers to claim their own namespace without needing to own a domain. This democratizes the module system while still maintaining uniqueness guarantees.

18.2 Creating Modules

A module can contain:

- Constants and variables
- Functions
- Classes, interfaces, and structs
- Enums
- Other module definitions
- Type definitions

When you create a subfolder in a Verse project, a module is automatically created for that folder. The file structure directly maps to the module hierarchy.

You can create modules within a `.verse` file using the following syntax:

```
# Colon syntax
module1 := module:
    # Module contents here
    MyConstant<public>:int = 42

    MyClass<public> := class:
        Value:int = 0

# Bracket syntax (also supported)
module2 := module
{
    # Module contents here
```

```
    AnotherConstant<public>:string = "Hello"
}
```

Modules can contain other modules, creating a hierarchy:

```
base_module<public> := module:
    submodule<public> := module:
        submodule_class<public> := class:
            Value:int = 100

    module_class<public> := class:
        Name:string = ""
```

The file structure `module_folder/base_module` is equivalent to:

```
module_folder := module:
    base_module := module:
        submodule := module:
            submodule_class := class:
                # Class definition
```

18.2.1 Restrictions

Module bodies have strict requirements about what they can contain. Understanding these restrictions helps avoid common errors when defining modules.

Modules Can Only Contain Definitions:

A module body can only contain definition statements—declarations that bind names to values. You cannot include arbitrary expressions or executable statements:

```
# Valid: All definitions
config := module:
    MaxValue:int = 100
    DefaultName:string = "Player"

    CalculateScore(Base:int):int = Base * 10

    player_class := class:
        Name:string

# Invalid: Contains non-definition expressions
bad_module := module:
    MaxValue:int = 100
    1 + 2  # ERROR 3560: Not a definition
```

```
15
16 # Invalid: Contains function call
17 bad_module2 := module:
18     InitFunction():void = {}
19     InitFunction()  # ERROR 3585: Cannot call function in module body
```

The restriction ensures that module initialization is deterministic and doesn't execute arbitrary code when the module is loaded.

Type Annotations Required:

All data definitions at module scope must explicitly specify their type. Type inference with `:=` alone is not allowed:

```
1 # Invalid: Missing type annotation
2 bad_module := module:
3     Value := 42  # ERROR 3547: Must specify type domain
4
5 # Valid: Explicit type annotation
6 good_module := module:
7     Value:int = 42  # OK: Type explicitly specified
```

This requirement makes module interfaces explicit and helps with separate compilation and module evolution.

Valid Module Contents:

Modules can contain these categories of definitions:

```
1 utilities := module:
2     # Constants with explicit types
3     Version:int = 1
4     AppName:string = "MyApp"
5
6     # Functions
7     Calculate(X:int):int = X * 2
8
9     # Classes, interfaces, structs
10    data_class := class:
11        Value:int
12
13    data_interface := interface:
14        GetValue():int
15
16    data_struct := struct:
17        X:float
```

```
        Y:float

    # Enums
    status := enum:
        Active
        Inactive

    # Nested modules
    nested := module:
        NestedFunction():void = {}

    # Type aliases
    coordinate := tuple(float, float)

    # Refinement types
    positive_int := type{X:int where X > 0}
```

Unlike functions, classes, or data values, modules are not first-class citizens in Verse. You cannot treat modules as values that can be stored, passed, or manipulated at runtime.

Cannot Assign Modules to Variables:

```
my_module := module:
    Value<public>:int = 42

# Invalid: Cannot treat module as value
M:my_module = my_module  # ERROR
```

Modules exist purely as namespaces and organizational constructs at compile time. The module identifier `my_module` can only be used in specific contexts.

Cannot Pass Modules as Arguments:

```
my_module := module:
    X<public>:int = 1

# Invalid: Cannot pass module as parameter
ProcessModule(M:module):void = {}  # ERROR
ProcessModule(my_module)  # ERROR
```

There is no `module` type that can be used in function signatures.

Cannot Create Collections of Modules:

```
module_a := module:
    Value:int = 1
```

```
3
4  module_b := module:
5      Value:int = 2
6
7  # Invalid: Cannot create tuple or array of modules
8  Modules := (module_a, module_b)  # ERROR
```

18.3 Importing Modules

The import system is designed to be explicit and predictable. Unlike some languages that automatically import commonly used modules or search multiple locations for dependencies, Verse requires you to explicitly declare every external module you want to use. This explicitness helps prevent naming conflicts and makes dependencies clear.

The `using` statement is the primary mechanism for importing modules into your Verse code. It usually is placed at the top of your file, before any other code definitions, and makes the contents of the specified module available in your current scope.

The basic syntax is straightforward - the keyword `using` followed by the module path in curly braces:

```
1  using { /Verse.org/Random }
2  using { /Fortnite.com/Devices }
3  using { /Verse.org/Simulation }
4  using { /UnrealEngine.com/Temporary/Diagnostics }
```

When you import a module, all its public members become available in your code. However, you still need to qualify them with the module name unless the names are unambiguous. This qualification requirement helps maintain code clarity and prevents accidental use of the wrong definition when multiple modules define similar names.

Using is a Statement, Not an Expression:

The `using` directive is a statement-level declaration that must appear at the top level of your code. You cannot use it as an expression or embed it in other expressions:

```
1  # Invalid: using in expression context
2  # f():void = using{MyModule}  # ERROR 3669
3
4  # Invalid: using in conditional
5  # if (using{MyModule}, Condition?):
6  #     DoSomething()  # ERROR 3669
```

```
7
8  # Invalid: using in class/struct/interface body
9  # my_class := class:
10 #     using{MyModule}  # ERROR 3537
11 #     Field:int
12
13 # Invalid: using module path in function body
14 # ProcessData():void =
15 #     using{/MyProject/UtilityModule}  # ERROR 3669
16 #     Calculate()
```

Module `using` statements must appear at the file or module level, not nested within other constructs. This ensures that imports are visible and consistent throughout the scope where they're declared.

While module imports with paths are not allowed in function bodies, Verse does support **local scope** `using` with local variables and parameters. See Local Scope Using below for details.

Valid using placement:

```
1  # At file level (most common)
2  using { /Verse.org/Random }
3  using { /Verse.org/Simulation }
4
5  ProcessData():void =
6      # Use imported functions
7      Value := GetRandomFloat(0.0, 1.0)
8
9  # Within module definition
10 utilities := module:
11     using { /Verse.org/Random }
12
13     GenerateId<public>():int =
14         GetRandomInt(1, 1000000)
```

18.3.1 Import Resolution

When Verse encounters a `using` statement, it follows a specific resolution process:

1. **Absolute paths** (starting with `/`) are resolved from the global module registry
2. **Relative paths** (without leading `/`) are resolved relative to the current module's location
3. **Nested modules** can be accessed through their parent modules

This resolution process happens at compile time, meaning that all imports must be resolvable when your code is compiled. There's no runtime module loading or dynamic imports in Verse.

18.3.2 Local and Relative Imports

For modules within your own project, you have flexibility in how you reference them:

```
# Absolute import from your project root
using { /MyGameProject/Systems/Combat }

# Import from a sibling folder
using { ../UI/MainMenu }

# Import from the same directory
using { player_controller }

# Import from a subdirectory
using { Subsystems/WeaponSystem }
```

The choice between absolute and relative imports often depends on your project structure and whether you plan to reorganize your modules. Absolute imports are more stable when refactoring, while relative imports can make module groups more portable.

18.3.3 Nested Imports

Nested modules present special considerations for importing. The order in which you import modules matters, and there are multiple valid approaches:

```
# Method 1: Import parent first, then child
using { game_systems }
using { inventory }  # Assumes inventory is nested in game_systems

# Method 2: Direct path to nested module
using { game_systems.inventory }

# Method 3: Import parent and access child through qualification
using { game_systems }
# Later in code: game_systems.inventory.Item

# IMPORTANT: This order causes an error
# using { inventory }      # Error: inventory not found
# using { game_systems }   # Too late, inventory import already failed
```

The restriction on import order exists because Verse resolves imports sequentially. When you import a nested module directly, Verse needs to know about its parent module first. This is why importing the parent before the child always works, while the reverse order fails.

18.3.4 Scope and Visibility

Imports have file scope - they only affect the file in which they appear. If you have multiple `.verse` files in the same module, each file needs its own import statements for external modules. However, files within the same module can see each other's definitions without imports:

```
# File: player_module/health.verse
health_component := class:
    CurrentHealth:float = 100.0

# File: player_module/armor.verse
# No import needed for health_component since it's in the same module
armor_component := class:
    HealthComp:health_component = health_component{}
```

18.3.5 Import Conflicts

When two imported modules define members with the same name, you need to disambiguate:

```
using { /GameA/Combat }
using { /GameB/Combat }

# Both modules might define CalculateDamage
# You must use qualified names:
DamageA := Combat.CalculateDamage(10.0)  # Error: ambiguous
DamageA := (/GameA/Combat:)CalculateDamage(10.0)  # OK: fully qualified
DamageB := (/GameB/Combat:)CalculateDamage(10.0)  # OK: fully qualified
```

18.3.6 Qualified Names

After importing, you can refer to module contents using qualified names. Verse provides two forms of qualification: standard dot notation for most cases, and special qualified access syntax for disambiguation.

When you need to disambiguate between identifiers with the same name from different modules, or when you want to explicitly specify the scope of an identifier, use a qualified access expression using parentheses and a colon:

```
# Qualified access syntax: (qualifier:)identifier

using { combat_module }
using { magic_module }

ProcessDamage():void =
    # Both modules define CalculateDamage
    PhysicalDamage := (combat_module:)CalculateDamage(100.0)
    MagicalDamage := (magic_module:)CalculateDamage(100.0)

    # Explicitly qualify local vs module identifiers
    LocalItem := item{Name := "Sword"}  # Local definition
    ModuleItem := (inventory_module:)item{Name := "Shield"}  # From module
```

The qualified access expression `(module:)identifier` is particularly useful in several scenarios:

1. **Resolving naming conflicts**: When multiple imported modules export the same identifier
2. **Explicit scoping**: When you want to make it clear which module an identifier comes from for readability
3. **Accessing shadowed names**: When a local definition shadows a module member
4. **Generic programming**: When working with parametric types where the qualifier might be computed

18.4 Module-Scoped Variables

Variables defined at module scope are global to any game instance where the variable is in scope.

Use `weak_map(session, t)` for variables that persist for the duration of a game session:

```
var GlobalCounter:weak_map(session, int) = map{}

IncrementCounter()<transacts>:void =
    CurrentValue := if (Value := GlobalCounter[GetSession()]) then Value + 1 else 0
    if (set GlobalCounter[GetSession()] = CurrentValue) {}
```

Use `weak_map(player, t)` for data that persists across game sessions:

```
var PlayerSaveData:weak_map(player, player_data) = map{}

player_data := class<final><persistable>:
```

```
    Level:int = 1
    Experience:int = 0
    UnlockedItems:[]string = array{}

SavePlayerProgress(Player:player, NewData:player_data)<decides>:void =
    set PlayerSaveData[Player] = NewData
```

18.5 Metaverse and Publishing

When you publish a module to the Metaverse, the module path becomes globally accessible, its public members become part of the module's API, and from that point the module must maintain backward compatibility.

The following example of shows how evolution works:

```
# Initial publication
Thing<public>:rational = 1/3

# Valid updates:
# - Change the value (not the type)
Thing<public>:rational = 10/3

# - Make the type more specific (subtype)
Thing<public>:int = 20  # nat is a subtype of int

# Invalid updates (would be rejected):
# - Remove the member
# - Change to incompatible type
# Thing<public>:string = "hello"  # Would fail
```

18.6 Local Qualifiers

The `(local:)` qualifier can disambiguate identifiers within functions. This is critical for evolution compatibility—when external modules add new public definitions after your code is published, `(local:)` ensures your local definitions take precedence.

```
# External module adds ShadowX after your code published
ExternalModule<public> := module:
    ShadowX<public>:int = 10  # Added later!

MyModule := module:
    using{ExternalModule}
```

```

    # Without (local:) - shadowing conflict
    # Foo():float =
    #     ShadowX:float = 0.0  # Error: conflicts with ExternalModule.ShadowX
    #     ShadowX

    # With (local:) - clear disambiguation
    Foo():float =
        (local:)ShadowX:float = 0.0  # Local variable
        (local:)ShadowX              # Returns 0.0, not 10
```

The `(local:)` qualifier can be used in these contexts:

Function parameters:

```
ProcessValue((local:)Value:int):int =
    (local:)Value + 1
```

Function body data definitions:

```
Compute():int =
    (local:)Result:int = 42
    (local:)Result
```

For loop variables:

```
SumValues():int =
    var Total:int = 0
    for ((local:)I := 0..10):
        set Total += (local:)I
    Total
```

If conditions:

```
CheckValue():float =
    if (X := GetValue[], (local:)X > 5.0):
        (local:)X
    else:
        0.0
```

Block scopes:

```
ComputeInBlock():int =
    block:
        (local:)Temp:int = 10
        (local:)Temp * 2
```

Class blocks:

```
my_class := class:
    var Value<public>:int = 0
    block:
        (local:)Value:int = 42
        set (my_class:)Value = (local:)Value
```

The `(local:)` qualifier **cannot** be used in these contexts:

Nested Scope Limitation:

Currently, you **cannot** redefine a `(local:)` qualified identifier in nested blocks:

```
# Error: cannot redefine local identifier
F((local:)X:int):int =
    block:
        (local:)X:float = 5.5  # Error: X already defined in function
    (local:)X
```

This limitation may be lifted in future versions to support more complex scoping patterns.

18.7 Automatic Qualification

When you write Verse code, you use simple, unqualified identifiers for clarity and readability. However, the Verse compiler internally transforms all identifiers into fully-qualified forms that explicitly specify their scope and origin. This process, called **automatic qualification**, ensures that every identifier is unambiguous and can be resolved to exactly one definition.

Understanding automatic qualification helps you understand how Verse resolves names, why certain errors occur, and how the module system maintains correctness even in complex codebases with many modules and overlapping names.

The compiler qualifies several categories of identifiers:

1. **Top-level definitions** - Functions, variables, classes, modules at package scope
2. **Type references** - All types, including built-in types like `int` and `string`
3. **Function parameters** - Local parameters get the `(local:)` qualifier
4. **Class and interface members** - Methods, fields, nested within composite types
5. **Module members** - Public and internal definitions within modules
6. **Nested scopes** - References within nested modules, classes, and functions

Verse uses several patterns to qualify identifiers based on their scope:

Package-level qualification: Definitions at the root of a package are qualified with the package path:

```
# What you write:
Function(X:int):int = X

# How the compiler sees it:
(/YourPackage:)Function((local:)X:(/Verse.org/Verse:)int):(/Verse.org/Verse:)int = (local:)X
```

The package path `/YourPackage` becomes the qualifier for `Function`, while the parameter `X` gets the special `(local:)` qualifier, and the built-in type `int` is qualified with its standard library path `/Verse.org/Verse`.

Local scope qualification: Function parameters and local variables are marked with `(local:)`:

```
# What you write:
ProcessValue(Input:int, Multiplier:int):int =
    Input * Multiplier

# How the compiler sees it:
(/YourPackage:)ProcessValue((local:)Input:(/Verse.org/Verse:)int, (local:)Multiplier:(/Verse
    (local:)Input * (local:)Multiplier
```

Nested scope qualification: Members within classes, interfaces, or modules get qualified with their container's path:

```
# What you write:
player_class := class:
    Health:float = 100.0

    TakeDamage(Amount:float):void =
        set Health = Health - Amount

# How the compiler sees it:
(/YourPackage:)player_class := class:
    (/YourPackage/player_class:)Health:(/Verse.org/Verse:)float = 100.0

    (/YourPackage/player_class:)TakeDamage((local:)Amount:(/Verse.org/Verse:)float):(/Verse.
        set (/YourPackage/player_class:)Health = (/YourPackage/player_class:)Health - (local
```

Notice how `Health` and `TakeDamage` are qualified with `/YourPackage/player_class` to indicate they're members of the class.

Module member qualification: Definitions within modules are qualified with the module path:

```
# What you write:
config := module:
    MaxPlayers<public>:int = 100

    GetPlayerLimit<public>():int = MaxPlayers

# How the compiler sees it:
(/YourPackage:)config := module:
    (/YourPackage/config:)MaxPlayers<public>:(/Verse.org/Verse:)int = 100

    (/YourPackage/config:)GetPlayerLimit<public>():(/Verse.org/Verse:)int =
        (/YourPackage/config:)MaxPlayers
```

All built-in types are qualified with their standard library paths. This makes it explicit where these types come from and maintains consistency with user-defined types:

```
# Common built-in types and their full qualifications:
int       → (/Verse.org/Verse:)int
float     → (/Verse.org/Verse:)float
string    → (/Verse.org/Verse:)string
logic     → (/Verse.org/Verse:)logic
message   → (/Verse.org/Verse:)message
```

When you write `X:int`, the compiler expands it to `X:(/Verse.org/Verse:)int`, making the type's origin explicit.

18.7.1 Example

Here's a more realistic example showing how qualification works across multiple scopes:

```
# What you write:
game_system := module:
    BaseValue:int = 42

    calculator := module:
        Multiplier:int = 2

        Calculate(Input:int):int =
            Input * Multiplier + BaseValue

# How the compiler sees it:
(/YourGame:)game_system := module:
    (/YourGame/game_system:)BaseValue:(/Verse.org/Verse:)int = 42
```

```
14
15      (/YourGame/game_system:)calculator := module:
16          (/YourGame/game_system/calculator:)Multiplier:(/Verse.org/Verse:)int = 2
17
18          (/YourGame/game_system/calculator:)Calculate((local:)Input:(/Verse.org/Verse:)int):(/
19              (local:)Input * (/YourGame/game_system/calculator:)Multiplier + (/YourGame/game_s
```

Notice how:

- The parameter `Input` is `(local:)`
- `Multiplier` is qualified with its containing module path
- `BaseValue` is qualified with the outer module path
- All type references are qualified with the Verse standard library path

18.7.2 Qualification with Using

When you import modules with `using`, the compiler still qualifies all identifiers, but it can resolve unqualified names to the imported modules:

```
1  # What you write:
2  using { /Verse.org/Random }
3
4  GenerateRandomValue():float =
5      GetRandomFloat(0.0, 1.0)
6
7  # How the compiler sees it:
8  using { /Verse.org/Random }
9
10 (/YourGame:)GenerateRandomValue():(/Verse.org/Verse:)float =
11     (/Verse.org/Random:)GetRandomFloat(0.0, 1.0)
```

The compiler resolves `GetRandomFloat` to `(/Verse.org/Random:)GetRandomFloat` based on the `using` statement.

18.7.3 When It Matters

You rarely need to think about automatic or manual qualification during normal development, as the compiler handles it transparently. However, understanding it helps in several situations:

Debugging name resolution errors: When the compiler reports ambiguous or unresolved identifiers, understanding qualification helps you see why:

```
1  using { /ModuleA }
2  using { /ModuleB }
3
```

```
# Both modules define Calculate
Result := Calculate(10)  # ERROR: Ambiguous - could be either module
```

The error occurs because the compiler cannot automatically qualify `Calculate` - it could be either `(/ModuleA:)Calculate` or `(/ModuleB:)Calculate`.

Shadowing conflicts: When a local variable has the same name as a module member:

```
MyModule := module:
    Value:int = 100

    Process(Value:int):int =
        # Without explicit qualification, this is ambiguous
        Value + Value  # Which Value? Module or parameter?
```

The compiler needs qualification to distinguish `(/MyModule:)Value` from `(local:)Value`.

Understanding error messages: Compiler error messages sometimes show qualified names to precisely identify which definition is involved:

```
Error: Cannot assign (/Verse.org/Verse:)string to (/Verse.org/Verse:)int at line 42
```

This makes it clear that the error involves the built-in `string` and `int` types, not user-defined types with the same names.

Working with generated or reflected code: Tools that generate Verse code or analyze code structure work with the qualified form, so understanding it helps when working with such tools.

18.7.4 Explicit Qualification

While the compiler automatically qualifies identifiers, you can also explicitly qualify them using the qualified access syntax `(qualifier:)identifier`. This is useful when you want to override automatic resolution or make your intent explicit:

```
game_system := module:
    Value:int = 100

    # Explicitly qualify to avoid any ambiguity
    GetValue():int = (game_system:)Value

    # Use local qualifier for parameters
    SetValue((local:)Value:int):void =
        set (game_system:)Value = (local:)Value
```

Explicit qualification is particularly valuable when:

- Resolving naming conflicts between imported modules
- Making code more self-documenting
- Overriding shadowing behavior
- Working with dynamic or computed qualifiers

18.8 Local Scope Using

While module-level `using` imports modules by their paths, Verse also supports **local scope** `using` within function bodies to enable member access inference from local variables and parameters. This feature makes code cleaner when working with objects that have many member accesses.

Local scope `using` takes a local variable or parameter identifier (not a module path) and makes its members accessible without explicit qualification:

```
entity := class:
    Name:string = "Entity"
    var Health:int = 100

    UpdateHealth(Amount:int):void =
        set Health = Health + Amount

ProcessEntity(E:entity):void =
    # Explicit member access
    Print(E.Name)
    E.UpdateHealth(-10)
    Print("{E.Health}")

    # With local using - inferred member access
    using{E}
    Print(Name)         # Inferred as: E.Name
    UpdateHealth(-10)   # Inferred as: E.UpdateHealth(-10)
    Print("{Health}")       # Inferred as: E.Health
```

The `using{E}` expression makes all members of `E` accessible without the `E.` prefix within the current scope.

18.8.1 With Local Variables

Local `using` works with variables defined in the same function:

```
CreateAndProcess():void =
    Player := player{Name := "Alice", Score := 100}

    # Without using
```

```
    Print(Player.Name)
    set Player.Score = Player.Score + 50

    # With using
    using{Player}
    Print(Name)         # Inferred as: Player.Name
    set Score = Score + 50  # Inferred as: Player.Score
```

18.8.2 Block Scoping

The `using` scope is limited to the block where it appears and any nested blocks:

Using in same block:

```
ProcessData():void =
    block:
        Data := data_record{}
        using{Data}
        UpdateField(Value)  # Inferred as: Data.UpdateField(Data.Value)
    # Data members no longer accessible here
```

Using from outer block:

```
ProcessData():void =
    Data := data_record{}
    block:
        using{Data}  # Can use variable from outer scope
        UpdateField(Value)  # Works - Data in scope
```

Nested block inheritance:

```
ProcessData():void =
    Data := data_record{}
    using{Data}  # Applies to this block and nested blocks

    block:
        # Inner block inherits outer using
        UpdateField(Value)  # Still infers Data.UpdateField(Data.Value)
```

18.8.3 Order

Member inference only works **after** the `using` expression is encountered:

```
# ERROR: Cannot infer before using
ProcessData(Data:data_record):void =
    UpdateField()  # ERROR - before using
```

```
    using{Data}
    UpdateField()  # OK - after using

# ERROR: Using scope doesn't extend backward
ProcessData(Data:data_record):void =
    block:
        using{Data}
        UpdateField()  # OK - within using scope
    UpdateField()  # ERROR - after using scope ended
```

The `using` statement acts as a declaration point - inference is not retroactive.

18.8.4 Conflict Resolution

You can have multiple `using` expressions in the same scope, but conflicting member names must be explicitly qualified:

```
player_stats := class:
    Health:int = 100
    Mana:int = 50
    GetInfo():string = "Player"

enemy_stats := class:
    Health:int = 80
    Armor:int = 20
    GetInfo():string = "Enemy"

ProcessCombat(Player:player_stats, Enemy:enemy_stats):void =
    using{Player}
    Print(GetInfo())  # Player.GetInfo()
    Print("{Mana}")       # Player.Mana (no conflict)

    using{Enemy}
    # Now both are in scope
    Print("{Armor}")      # Enemy.Armor (no conflict with Player)

    # ERROR: Conflicts must be qualified
    # Print(Health)   # Ambiguous - both have Health
    # Print(GetInfo())  # Ambiguous - both have GetInfo

    # Must qualify conflicting members
    Print("{Player.Health}")
    Print("{Enemy.Health}")
```

```
    Print(Player.GetInfo())
    Print(Enemy.GetInfo())
```

When members exist in multiple `using` contexts, you must explicitly qualify to disambiguate.

18.8.5 Mutable Member

Local `using` works with mutable fields through the `set` keyword:

```
config := class:
    var Volume:float = 1.0
    var Quality:int = 2

UpdateSettings(Settings:config):void =
    using{Settings}

    # Mutable field access
    set Volume = 0.8     # Inferred as: set Settings.Volume = 0.8
    set Quality = 3      # Inferred as: set Settings.Quality = 3
```

18.9 Troubleshooting

When working with modules, you may encounter various issues. Understanding these common problems and their solutions will help you debug module-related errors more efficiently.

18.9.1 Module Not Found Errors

Problem: The compiler reports that a module cannot be found when you try to import it.

Common Causes and Solutions:

1. **Incorrect path**: Double-check the module path in your `using` statement. Remember that paths are case-sensitive.

```
# Wrong: different case
using { /verse.org/random }  # Error: module not found

# Correct: proper case
using { /Verse.org/Random }  # Works
```

2. **Missing parent module import**: When importing nested modules, ensure the parent is imported first.

```
# Wrong: child before parent
using { inventory }  # Error if inventory is nested

# Correct: parent first
using { game_systems }
using { inventory }
```

3. **File location mismatch**: Ensure your file structure matches your module structure. If you have a folder named `player_systems`, all files in that folder are part of the `player_systems` module.

18.9.2 Access Denied Errors

Problem: You can't access a member of an imported module.

Common Causes and Solutions:

1. **Missing access specifier**: Members without the `<public>` specifier are internal by default.

```
# In module_a
SecretValue:int = 42  # Internal by default
PublicValue<public>:int = 100  # Explicitly public

# In another module
using { module_a }
X := module_a.SecretValue  # Error: not accessible
Y := module_a.PublicValue  # Works
```

2. **Protected or private members**: These are not accessible outside their defining scope.

```
# In a class
class_a := class:
    PrivateField<private>:int = 10
    ProtectedField<protected>:int = 20
    PublicField<public>:int = 30

# Outside the class
Obj := class_a{}
```

```
X := Obj.PrivateField  # Error: private
Y := Obj.PublicField   # Works
```

18.9.3 Circular Dependency Errors

Problem: Two modules try to import each other, creating a circular dependency.

Solution: Restructure your code to avoid circular dependencies:

1. **Extract common code**: Move shared definitions to a third module that both can import.
2. **Use interfaces**: Define interfaces in a separate module to break the dependency cycle.
3. **Reconsider architecture**: Circular dependencies often indicate a design issue that needs rethinking.

18.9.4 Name Collision Errors

Problem: Two imported modules define members with the same name.

Solution: Use fully qualified names to disambiguate:

```
using { /GameA/Combat }
using { /GameB/Combat }

# Ambiguous
Damage := CalculateDamage(10.0)  # Error: which CalculateDamage?

# Explicit
DamageA := /GameA/Combat.CalculateDamage(10.0)  # Clear
DamageB := /GameB/Combat.CalculateDamage(10.0)  # Clear
```

18.9.5 Persistence Issues

Problem: Module-scoped variables aren't persisting as expected.

Common Causes and Solutions:

1. **Wrong type used**: Ensure you're using `weak_map(player, t)` for player persistence.
2. **Type not persistable**: Check that your custom types have the `<persistable>` specifier.
3. **Initialization timing**: Make sure you're initializing persistent data at the right time in the game lifecycle.

18.9.6 Local Qualifier Conflicts

Problem: Shadowing errors when local identifiers conflict with module members.

Solution: Use the `(local:)` qualifier to disambiguate:

```
module_x := module:
    Value:int = 10

    ProcessValue((local:)Value:int):int =
        (module_x:)Value + (local:)Value  # Clear distinction
```

Part V

Part V: Production

Chapter 19

Chapter 18: Persistable Types

Persistable types allow you to store data that persists beyond the current game session. This is essential for saving player progress, preferences, and other game state that should be maintained across multiple play sessions.

Persistable data is stored using module-scoped `weak_map(player, t)` variables, where `t` is any persistable type. When a player joins a game, their previously saved data is automatically loaded into all module-scoped variables of type `weak_map(player, t)`.

```
using { /Fortnite.com/Devices }
using { /UnrealEngine.com/Temporary/Diagnostics }
using { /Verse.org/Simulation }

# Global persistable variable storing player data
MySavedPlayerData : weak_map(player, int) = map{}

# Initialize data for a player if not already present
InitializePlayerData(Player : player) : void =
    if (not MySavedPlayerData[Player]):
        if (set MySavedPlayerData[Player] = 0) {}
```

19.1 Built-in Persistable Types

The following primitive types are persistable by default:

- Numeric Types:
  - `logic` - Boolean values (true/false)
  - `int` - Integer values (must fit in 64-bit signed range for persistence)

  - `float` - Floating-point numbers

- Character Types:
  - `string` - Text values
  - `char` - Single UTF-8 character
  - `char32` - Single UTF-32 character

- Container Types:
  - `array` - Persistable if element type is persistable
  - `map` - Persistable if both key and value types are persistable
  - `option` - Persistable if the wrapped type is persistable
  - `tuple` - Persistable if all element types are persistable

19.2 Custom Persistable Types

You can create custom persistable types using the `<persistable>` specifier with classes, structs, and enums.

Classes must meet specific requirements to be persistable:

```
player_profile_data := class<final><persistable>:
    Version:int = 1
    Class:player_class = player_class.Villager
    XP:int = 0
    Rank:int = 0
    CompletedQuestCount:int = 0
```

Requirements for persistable classes:

- Must have the `<persistable>` specifier
- Must be `<final>` (no subclasses allowed)
- Cannot be `<unique>`
- Cannot have a superclass (including interfaces)
- Cannot be parametric (generic)
- Can only contain persistable field types
- Cannot have variable members (`var` fields)
- Field initializers must be effect-free (cannot use `<transacts>`, `<decides>`, etc.)

Structs are ideal for simple data structures that won't change after publication:

```
coordinates := struct<persistable>:
    X:float = 0.0
    Y:float = 0.0
```

Requirements for persistable structs:

- Must have the `<persistable>` specifier
- Cannot be parametric (generic)
- Can only contain persistable field types (see Prohibited Field Types below)
- Field initializers must be effect-free (cannot use `<transacts>`, `<decides>`, etc.)
- Cannot be modified after island publication

Enums represent a fixed set of named values:

```
day := enum<persistable>:
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
```

Important notes:

- `<closed>` persistable enums cannot be changed to open after publication
- Only `<open>` persistable enums can have new values added after publication

19.3 Prohibited Field Types

Persistable types have strict restrictions on what field types they can contain. The following types **cannot** be used as fields in persistable classes or structs:

- Abstract and Dynamic Types:
  - `any` - Cannot be persisted (too dynamic)
  - `comparable` - Abstract interface type
  - `type` - Type values cannot be persisted
- Non-Serializable Types:
  - `rational` - Exact rational numbers (not persistable)
  - **Function types** (e.g., `int -> int`) - Functions cannot be serialized
  - `weak_map` - Weak references are not persistable
  - **Interface types** - Abstract interfaces cannot be persisted
- Non-Persistable User Types
  - **Non-persistable enums** - Enums without `<persistable>` specifier cannot be used
  - **Non-persistable classes** - Classes without `<persistable>` specifier cannot be used

- **Non-persistable structs** - Structs without `<persistable>` specifier cannot be used

19.4 Example

Initializing Player Data:

```
# Define a persistable player stats structure
player_stats := struct<persistable>:
    Level:int = 1
    Experience:int = 0
    GamesPlayed:int = 0

# Global persistent storage
PlayerData : weak_map(player, player_stats) = map{}

# Initialize or retrieve player data
GetOrCreatePlayerStats(Player : player) : player_stats =
    if (ExistingStats := PlayerData[Player]):
        ExistingStats
    else:
        NewStats := player_stats{}
        if (set PlayerData[Player] = NewStats):
            NewStats
        else:
            player_stats{}  # Fallback
```

19.5 JSON Serialization

Unreleased Feature

JSON Serialization have not yet been released and is not publicly available.

Verse provides JSON serialization functions for persistable types, enabling manual serialization and deserialization of data. While the primary persistence mechanism uses `weak_map(player, t)` for automatic player data, JSON serialization can be useful for debugging, data migration, or integration with external systems.

Converts a persistable value to JSON string:

```
player_data := class<final><persistable>:
    Level:int = 1
    Score:int = 100

Data := player_data{Level := 5, Score := 250}
```

```
JsonString := Persistence.ToJson[Data]
# Produces: {"$package_name":"/...", "$class_name":"player_data", "x_Level":5, "x_Score":250}
```

Deserializes JSON string to typed value:

```
JsonString := "{\"$package_name\":\"/.../\", \"$class_name\":\"player_data\", \"x_Level\":10,
if (Restored := Persistence.FromJson[JsonString, player_data]):
    # Restored.Level = 10
    # Restored.Score = 500
```

All serialized persistable objects include metadata fields:

```
{
  "$package_name": "/SolIdeDataSources/_Verse",
  "$class_name": "player_data",
  "x_Level": 5,
  "x_Score": 250
}
```

Metadata fields:

- `$package_name` - Package path of the type
- `$class_name` - Qualified class/struct name

Field names:

- Prefixed with `x_` in current format
- Old format used mangled names like `i___verse_0x123_FieldName`

19.5.1 Type-Specific Serialization

Primitives:

```
int_ref := class<final><persistable>:
    Value:int

# Serialized as JSON number
JsonString := Persistence.ToJson[int_ref{Value := 42}]
# {"$package_name":"...", "$class_name":"int_ref", "x_Value":42}
```

Optional types:

```
optional_ref := class<final><persistable>:
    Value:?int

# None serialized as false
Persistence.ToJson[optional_ref{Value := false}]
```

```
# {..., "x_Value":false}

# Some serialized as object with empty key
Persistence.ToJson[optional_ref{Value := option{42}}]
# {..., "x_Value":{"":42}}
```

Tuples:

```
tuple_ref := class<final><persistable>:
    Pair:tuple(int, int)

# Serialized as JSON array
Persistence.ToJson[tuple_ref{Pair := (4, 5)}]
# {..., "x_Pair":[4,5]}

# Empty tuple
empty_tuple_ref := class<final><persistable>:
    Empty:tuple()

Persistence.ToJson[empty_tuple_ref{Empty := ()}]
# {..., "x_Empty":[]}
```

Arrays:

```
array_ref := class<final><persistable>:
    Numbers:[]int

Persistence.ToJson[array_ref{Numbers := array{1, 2, 3}}]
# {..., "x_Numbers":[1,2,3]}
```

Maps:

```
map_ref := class<final><persistable>:
    Lookup:[string]int

Persistence.ToJson[map_ref{Lookup := map{"a" => 1, "b" => 2}}]
# {..., "x_Lookup":[{"k":{"":"a"},"v":{"":1}}, {"k":{"":"b"},"v":{"":2}}]}
```

Enums:

```
day := enum<persistable>:
    Monday
    Tuesday

enum_ref := class<final><persistable>:
    Day:day
```

```
7
8  Persistence.ToJson[enum_ref{Day := day.Monday}]
9  # {..., "x_Day":"day::Monday"}
```

19.5.2 Default Value Handling

When deserializing, missing fields are automatically filled with their default values:

```
1  versioned_data := class<final><persistable>:
2      Version:int = 1
3      NewField:int = 0  # Added in v2
4
5  # Old JSON without NewField
6  OldJson := "{\"$package_name\":\"...\", \"$class_name\":\"versioned_data\", \"x_Version\":1}"
7
8  # Deserializes successfully with default for NewField
9  if (Data := Persistence.FromJson[OldJson, versioned_data]):
10     Data.Version = 1
11     Data.NewField = 0  # Uses default value
```

This enables forward-compatible schema evolution - new fields with defaults can be added without breaking old saved data.

19.5.3 Block Clauses During Deserialization

Block clauses do not execute when deserializing from JSON:

```
1  logged_class := class<final><persistable>:
2      Value:int
3      block:
4          Print("Constructed!")
5
6  # Normal construction triggers block
7  Instance1 := logged_class{Value := 1}  # Prints "Constructed!"
8
9  # Deserialization does NOT trigger block
10 Json := Persistence.ToJson[Instance1]
11 Instance2 := Persistence.FromJson[Json, logged_class]  # No print
```

Block clauses are only executed during normal construction, not during deserialization. This means initialization logic in blocks won't run for loaded data.

19.5.4 Integer Range Limitations

Verse protects against integer overflow during serialization. Integers that exceed the safe serialization range cause runtime errors:

```
int_ref := class<final><persistable>:
    Value:int

# Safe range integers work fine
SafeData := int_ref{Value := 1000000000000000000}
Persistence.ToJson[SafeData]  # OK

# Overflow protection - runtime error for very large integers
var BigInt:int = 1
for (I := 1..63):
    set BigInt *= 2

# Runtime error: Integer too large for safe serialization
# Persistence.ToJson[int_ref{Value := BigInt}]
```

This prevents silent precision loss that could occur with floating-point representation of large integers.

19.5.5 Backward Compatibility

The serialization system maintains backward compatibility with older JSON formats:

Field name migration:

```
# Old format (V1) with mangled names
OldJson := "{\"$package_name\":\"...\", \"i___verse_0x123_Value\":42}"

# Deserializes correctly
Data := Persistence.FromJsonV1[OldJson, int_ref]

# Re-serializes with new format
NewJson := Persistence.ToJson[Data]
# {"$package_name":"...", "x_Value":42}
```

19.6 Best Practices

- **Schema Stability:** Design your persistable types carefully, as they cannot be easily changed after publication. Consider versioning strategies for future updates.
- **Use Structs for Simple Data:** For data that won't need inheritance or complex behavior, prefer persistable structs over classes.
- **Handle Missing Data:** Always check if data exists for a player before accessing it, and provide appropriate defaults.

- **Atomic Updates:** When updating persistent data, create new instances rather than trying to modify existing ones (Verse uses immutable data structures).
- **Consider Memory Usage:** Persistent data is loaded for all players when they join, so be mindful of the amount of data stored per player.

19.7 Example: Player Profile System

```
using { /Fortnite.com/Devices }
using { /Verse.org/Simulation }

# Player class enum
player_class := enum<persistable>:
    Warrior
    Mage
    Archer
    Rogue

# Achievement data
achievement := struct<persistable>:
    Name:string = ""
    Completed:logic = false
    CompletedDate:int = 0  # Timestamp

# Complete player profile
player_profile := class<final><persistable>:
    Username:string = "Player"
    Level:int = 1
    Experience:int = 0
    SelectedClass:player_class = player_class.Warrior
    TotalPlayTime:float = 0.0
    Achievements:[]achievement = array{}

# Global player profiles
PlayerProfiles : weak_map(player, player_profile) = map{}

# Profile management device
profile_manager := class(creative_device):

    OnBegin<override>()<suspends>:void =
        # Initialize all players
        AllPlayers := GetPlayspace().GetPlayers()
```

```
        for (Player : AllPlayers):
            InitializeProfile(Player)

    InitializeProfile(Player : player) : void =
        if (not PlayerProfiles[Player]):
            DefaultProfile := player_profile{}
            set PlayerProfiles[Player] = DefaultProfile
```

This demonstrates how to create and manage persistable player data, ensuring that player progress and achievements are maintained across game sessions.

Chapter 20

Chapter 19: Code Evolution

Verse takes a unique approach to code evolution, designed with the ambitious goal of creating software that could remain functional and valuable for decades or even centuries. This vision stems from Verse's role as the programming language for a persistent, global metaverse where code must coexist, evolve, and maintain compatibility across vast timescales.

At its core, Verse embraces three fundamental principles that shape how code evolves: future-proof design that avoids being rooted in past artifacts of other languages, a metaverse-first approach where code persistence and compatibility are critical, and strong static verification that catches runtime problems at compile time. These principles create a foundation for a language that can grow and adapt while maintaining the stability required for a global, persistent codebase.

20.1 The Nature of Code Publication

When developers publish code to the Verse metaverse, they enter into a social contract with all future users of that code. This contract is more than just a convention—it's enforced by the language itself. Consider what happens when you publish a simple value:

```
1 Thing<public>:int = 666
```

This seemingly straightforward declaration carries profound implications. By marking `Thing` as public, you're making a commitment that extends indefinitely into the future. Users can depend on `Thing` always existing and always being an integer. While you retain the freedom to change its actual value, the existence and type of `Thing` become permanent fixtures in the metaverse's landscape.

This permanence extends beyond simple values to encompass the entire structure of published code. Persistable structs, once published to an island, become

immutable schemas that cannot be altered. Closed enums remain closed forever, unable to accept new values after publication. When a class or interface is marked with the `<castable>` attribute, that decision becomes irreversible, as changing it could introduce unsafe casting behaviors that break existing code.

The publication model distinguishes between two contexts: the live metaverse and islands. In the envisioned live metaverse, publishing an update that attempts to change an immutable variable's value has no effect—the variable already exists with its original value. However, in the current island-based implementation, new instances of an island will adopt the updated value, providing a practical migration path while maintaining conceptual consistency.

20.2 The Architecture of Backward Compatibility

Backward compatibility in Verse goes beyond simple syntactic preservation—it encompasses semantic guarantees about how code behaves. The language enforces these guarantees through multiple mechanisms that work together to create a robust compatibility framework.

Function effects exemplify this approach. When a function is published with specific effects like `<reads>`, indicating it may read mutable heap data, this becomes part of its contract. Future versions of the function can have fewer effects—evolving from `<reads>` to `<computes>`—but never more. This restriction ensures that code depending on the function's effect profile continues to work correctly, as the function only becomes more pure, never less.

Type evolution follows similar principles. Types can become more specific over time, such as changing from `rational` to `int`, as this represents a refinement rather than a fundamental change. Structures must maintain all existing fields, though new fields can be added. Classes marked with `<final_super>` commit to their inheritance hierarchy permanently, ensuring that code relying on specific inheritance relationships remains valid.

The enforcement of these rules happens at publication time, not just at compile time. Verse actively prevents developers from publishing updates that would violate compatibility guarantees, turning what might be runtime failures in other systems into publication-time errors that must be resolved before code can be deployed.

20.3 Managing Breaking Changes

Despite the strong emphasis on compatibility, Verse recognizes that some breaking changes are occasionally necessary. The language provides two mechanisms for

managing such changes: a deprecation system for gradual migration and special privileges for essential breaking changes.

The deprecation system operates as a multi-phase process that gives developers ample time to adapt. When code patterns become deprecated, they first generate warnings rather than errors. These warnings appear when saving code, alerting developers to practices that won't be supported in future versions. The code continues to compile and run, allowing projects to function while migration plans are developed. Only when developers explicitly upgrade to a new language version do deprecations become errors, and even then, the option to remain on older versions provides an escape hatch.

Version 1 introduced several significant deprecations that illustrate this process. The prohibition of failure in set expressions, which previously allowed with warnings, now requires explicit handling of failable expressions. Mixed separator syntax, which created implicit blocks and confusing scoping rules, must now use consistent separation. The introduction of local qualifiers provides a new tool for disambiguating identifiers while deprecating the use of 'local' as a regular identifier name.

For truly exceptional circumstances, Epic Games and potentially other future authorities retain "superpowers" to make breaking changes outside the normal compatibility framework. These powers include the ability to delete published entities, change types in non-backward-compatible ways, and rewrite modules for legal or safety reasons. These capabilities acknowledge that being good stewards of the metaverse namespace sometimes requires violating the usual compatibility rules, though such actions should remain rare and justified by compelling reasons.

20.4 Catalog of Compatibility Rules

When you publish code in Verse, many aspects of your definitions become permanent commitments. Understanding exactly what can and cannot change is essential for designing APIs that can evolve gracefully. This catalog documents both the changes that Verse prohibits and the changes it allows to ensure backward compatibility.

The rules follow a general principle: changes that make types more specific (narrowing), add new capabilities, or relax restrictions are often allowed, while changes that make types more general (widening), remove capabilities, or impose new restrictions are typically forbidden. However, the rules vary significantly between final/non-instance members and non-final instance members, with the latter having much stricter requirements.

20.4.1 Definitions

Cannot remove public definitions. Once a public variable, function, class, or other definition is published, it must remain available. Removing it would break any code that depends on it.

Cannot change the kind of a definition. A class cannot become a struct, interface, enum, or function. A function cannot become a class or type alias. These fundamental changes alter how code interacts with your definitions.

Cannot rename definitions. Renaming is equivalent to deletion plus addition, which breaks existing references.

20.4.2 Enums

Cannot add or remove enumerators from closed enums. Closed enums (the default) commit to a fixed set of values forever. Code can exhaustively match all cases without a wildcard, so adding cases would break such matches.

Closed enums cannot become open. An enum published as closed cannot become open. This affects whether exhaustive matching is possible.

Cannot reorder enumerators. The order of enum values is part of the public contract.

Cannot rename enumerators. Each enumerator name is a permanent identifier.

Open enums can add new enumerators. This is the sole evolution path for enums—open enums trade the guarantee of exhaustive matching for the ability to grow.

20.4.3 Classes and Inheritance

Class Finality:

- **Can make non-final class final.** Adding the `<final>` specifier prevents future inheritance, which is a safe addition that strengthens guarantees.
- **Cannot make final class non-final.** Once a class is marked `<final>`, removing this restriction would allow unexpected subclasses that could break assumptions in existing code.

Class Uniqueness:

- **Can make non-unique class unique.** Adding the `<unique>` specifier enables identity-based equality, which doesn't break existing code.

- **Cannot make unique class non-unique.** Removing `<unique>` would change the equality semantics from identity to undefined, breaking code that relies on identity comparison.

Class Abstractness:

- **Can make abstract class non-abstract.** Allowing instantiation of a previously abstract class is a safe expansion of capabilities.
- **Cannot make non-abstract class abstract.** Preventing instantiation of a previously concrete class breaks code that creates instances.

Class Concreteness:

- **Can make non-concrete final class concrete.** Allowing default instantiation of a final class is safe since no subclasses exist to be affected.
- **Cannot change concreteness in other cases.** Making concrete classes non-concrete or changing concreteness of non-final classes could break instantiation code or subclass behavior.

Inheritance Changes:

- **Can add inheritance to non-abstract classes.** Adding a parent class or interface extends capabilities without breaking existing functionality.
- **Cannot remove or change inheritance from non-abstract classes.** Removing a parent breaks code that depends on the inheritance relationship.
- **Cannot add, remove, or change inheritance on abstract classes.** Abstract class hierarchies must remain fixed to prevent conflicts and maintain subtype relationships.
- **Cannot add, remove, or change inheritance on interfaces.** Interface hierarchies must remain stable for the same reasons.

Special Attributes:

- **Cannot add or remove the `<castable>` attribute.** Runtime type checks depend on this property. Adding it after publication would enable casts that weren't previously safe, while removing it would break existing casts.
- **Cannot remove `<final_super>` once added.** Derived types marked with `<final_super>` must continue inheriting from the same parent to maintain the type hierarchy that `GetCastableFinalSuperClass` depends on.
- **Derived types with `<final_super>` must remain derived from the same parent.** Changing the parent type would break runtime type queries.

Special Transformations:

- **Can change final class with no inheritance to struct.** This is a safe transformation since both are value-like and the class cannot have subclasses.
- **Can potentially change abstract class with no class inheritance to interface.** This transformation maintains the same contract but is not yet fully implemented (marked as TODO in tests).

20.4.4 Structs

Cannot add any fields to structs. Structs are immutable value types with a fixed memory layout. Adding fields would change this layout and break binary compatibility.

Cannot change between class and struct. These are fundamentally different types with different semantics—classes are references, structs are values.

20.4.5 Fields and Data Members

Adding Fields:

- **Can add fields with default values to classes.** New fields with defaults don't break existing construction code since the defaults are used automatically.
- **Cannot add fields without default values to classes.** Existing code that constructs instances would break since it doesn't provide values for the new fields.
- **Cannot add any fields to structs.** Structs have fixed memory layout and adding fields breaks binary compatibility.

Removing Fields:

- **Cannot remove fields from classes or structs.** All published fields must remain available since code may reference them.

Field Mutability:

- **Can change final instance field to non-final.** Allowing mutation where it was previously prohibited is a safe expansion of capabilities.
- **Cannot change non-final instance field to final.** Once code depends on being able to mutate a field, making it immutable breaks that code.

Field Type Changes:

For **final or non-instance data** (fields that can't be overridden): - **Can narrow the type (make more specific).** For example, changing from `any` to `int` strengthens the guarantee about what values the field holds. - **Cannot widen**

the type (make more general). For example, changing from `int` to `any` violates the guarantee that callers could read a specific type.

For **non-final instance data** (fields that can be overridden in subclasses): - **Cannot narrow or widen the type.** These must maintain their exact type to prevent breaking overrides in subclasses or calling code.

Default Initializers:

- **Can add a default initializer to a class field.** This makes construction easier without breaking existing code.
- **Cannot remove a default initializer from a class field.** Removing a default breaks construction code that relied on it.

Overrides:

- **Can add an override to a field.** Providing a more specific implementation in a subclass is allowed.
- **Can remove an override if it didn't narrow the type.** If the override maintained the same type, removing it is safe.

20.4.6 Functions and Methods

Function signature changes follow different rules depending on whether the function is **final/non-instance** (can't be overridden) or a **non-final instance method** (can be overridden). The rules reflect fundamental principles of type safety: functions can become more flexible about what they accept (contravariance) and more specific about what they return (covariance), but only when overriding isn't involved.

Overload Management:

- **Cannot remove function overloads.** Each published function signature must remain available since code may call it.
- **Function overloads are matched by signature.** When checking compatibility, functions with the same parameters are compared to ensure compatible types and effects.

Return Types (Covariance):

For **final or non-instance functions** (module functions, static methods, final methods): - **Can narrow the return type (make more specific).** Changing from `any` to `int` means the function now guarantees a more specific return value, which is always safe for callers. - **Cannot widen the return type (make more general).** Changing from `int` to `any` means the function might return different types, breaking code that expects an integer.

For **non-final instance methods** (overridable methods): - **Cannot narrow or widen the return type.** These must maintain their exact return type to ensure subclass overrides remain compatible.

Parameter Types (Contravariance):

For **final or non-instance functions**: - **Can widen parameter types (make more general).** Changing from `int` to `any` means the function accepts more inputs, which never breaks existing calls with integers. - **Cannot narrow parameter types (make more specific).** Changing from `any` to `int` means the function rejects some previously valid arguments. - **Can relax type parameter constraints.** Changing from `t:subtype(comparable)` to `t:type` allows more type arguments. - **Cannot strengthen type parameter constraints.** The reverse change restricts valid type arguments.

For **non-final instance methods**: - **Cannot narrow or widen parameter types.** These must maintain their exact parameter types to ensure subclass overrides and calls remain compatible.

Optional Parameters:

- **Can add optional named parameters with defaults.** This doesn't break existing calls since the parameters are optional.
- **Can change default values of optional parameters.** New callers get the new defaults while existing compiled code continues using the values it was compiled with.

Effects (Covariance):

For **final or non-instance functions**: - **Can narrow effects (reduce).** Changing from `<transacts>` to `<reads>` to `<computes>` makes the function more pure, which is always safe since code expecting more effects can handle fewer. - **Cannot widen effects (increase).** Changing from `<computes>` to `<reads>` violates the contract that the function has limited effects.

For **non-final instance methods**: - **Cannot narrow or widen effects.** These must maintain their exact effects to ensure overrides work correctly.

Conversions Between Callable Forms:

- **Cannot convert between normal functions and constructors.** These are fundamentally different callable entities with different calling conventions.
- **Cannot convert between functions and parametric types.** A function cannot become a type parameter or vice versa.

Function body:

- **Cannot change the body of transparent functions.** Verification of callers might depend on the function body of transparent functions, so changes could break callers.
- **Cannot change the body of opaque functions without the `<reads>` effect.** This is to ensure that `NonReadsFunction()=NonReadsFunction()` even when code is evolving.
- **Can change the body of opaque functions that have the `<reads>` effect.** Code evolution can be observed by the `<reads>` effect.

Understanding Variance:

The asymmetry in these rules reflects **variance** in type theory: - **Parameters are contravariant**: Accepting more general types (widening) is safe - **Returns are covariant**: Returning more specific types (narrowing) is safe - **Effects are covariant**: Having fewer effects is safe

These rules only apply to final/non-instance functions because overridable methods must maintain exact signatures to preserve the Liskov Substitution Principle—any subclass override must be callable wherever the base method is called.

20.4.7 Access Specifiers

Increasing Accessibility (Allowed):

- **Can increase accessibility of definitions.** Making a private definition public, or protected to public, expands access without breaking existing code.
- **Can make constructors more accessible.** Allowing more code to construct instances is a safe capability expansion.

Reducing Accessibility (Forbidden):

- **Cannot reduce accessibility of definitions.** Making a public definition protected, internal, or private breaks all external code using it. Making a protected definition private breaks subclass access.
- **Cannot make constructors less accessible.** A class constructor that was public cannot become private or protected.

20.4.8 Persistable Types

Persistable types require stricter compatibility rules because they define the format of saved player data that must remain loadable indefinitely. Changes to persistable types risk corrupting or losing saved data.

Persistable Attribute Changes:

- **Cannot add the `<persistable>` attribute after publication.** Making a type persistable changes its serialization behavior and imposes new restrictions.
- **Cannot remove the `<persistable>` attribute.** Saved data depends on the persistence format of these types.

Persistable Class Fields:

- **Can add fields with default values to persistable classes.** Saved data without the new field will load successfully using the default.
- **Cannot add fields without defaults to persistable classes.** Old saved data wouldn't have values for these fields.
- **Cannot remove any fields from persistable classes.** Saved data may contain these field values and must be able to load them.

Persistable Struct Fields:

- **Cannot add any fields to persistable structs.** Structs have fixed layouts and saved data expects the exact structure.
- **Cannot remove any fields from persistable structs.** All fields in saved data must be loadable.

Persistable Enum Changes:

For **closed persistable enums**: - **Cannot add enumerators.** Case statements may exhaustively match all values, and saved data deserialization depends on the fixed set. - **Cannot remove enumerators.** Saved data may contain removed values, making it unloadable.

For **open persistable enums**: - **Can add new enumerators.** Open enums are designed to grow, and the persistence system handles unknown values. - **Cannot remove enumerators.** Saved data may still reference removed values.

Type Lifecycle:

- **Can add new persistable types.** Publishing new types for data persistence is always allowed.
- **Cannot remove persistable types once published.** They must remain available to deserialize old saved data.
- **Cannot change the structure of persistable types.** Field types, inheritance relationships, and other structural changes break deserialization.

Module/Type Aliases:

- **Can add module or type aliases to persistable types.** This provides additional ways to reference existing types without changing them.
- **Can remove module or type aliases to persistable types.** Removing an alias doesn't affect the underlying type's persistence.
- **Module aliases must reference the same path.** Changing the target breaks all code using the alias.
- **Type aliases must reference the same type.** Changing the aliased type breaks all code using the alias.

Non-Persistable Changes:

- **Can freely add or remove non-persistable types.** Types without `<persistable>` don't affect saved data and can be added or removed as needed.

Persistent Variables:

Verse supports persistent variables (`var` declarations in module scope) that maintain values across sessions:

- **Can add new persistent variables.** New variables are initialized with their default values.
- **Cannot remove persistent variables.** The metaverse expects these variables to exist persistently.
- **Cannot change persistent variable types.** Saved values must match the expected type.
- **Non-persistent variables can be freely changed.** Local or instance variables don't persist and can be modified.

20.4.9 Parametric Types

Parametric types (generic types with type parameters) have additional compatibility considerations:

Type Parameter Constraints:

- **Can widen type parameter constraints in parametric type domains.** Making constraints more permissive (e.g., from `t:subtype(comparable)` to `t:type`) allows more type arguments.
- **Cannot narrow type parameter constraints.** Restricting valid type arguments breaks existing code using the parametric type.
- **Type parameters are treated as rigid when checking functions inside parametric types.** This ensures type safety within the generic context.

Parametric vs Non-Parametric:

- **Cannot convert between parametric and non-parametric forms.** A parametric type cannot lose its type parameters, and a regular type cannot gain them.
- **Cannot convert between functions and parametric types.** These are fundamentally different constructs.

Variance:

- **Variance is inferred from usage, not declared.** How type parameters are used (in input positions, output positions, or both) determines their variance.
- **Cannot change inferred variance.** Once a type parameter's usage pattern is published, it establishes a variance contract that must be maintained.

20.4.10 Summary

This catalog represents the core compatibility guarantees that Verse enforces. While these restrictions may seem extensive, they ensure that published code remains a stable foundation for the metaverse ecosystem.

Key principles to remember:

1. **Additions are usually safe**: New optional fields, new overloads, new enumerators in open enums
2. **Removals are usually forbidden**: Removing public definitions breaks dependent code
3. **Narrowing is often safe**: More specific return types, fewer effects
4. **Widening is selectively safe**: More general parameter types (contravariance)
5. **Final/non-instance members are more flexible**: They can evolve types and effects
6. **Non-final instance members are rigid**: They must maintain exact signatures
7. **Persistable types are strictest**: Saved data imposes permanent constraints

The key to working within these constraints is thoughtful initial design—choosing the right visibility, finality, effects, and type properties from the start. Consider future evolution needs when making these irreversible decisions.

20.5 Design Philosophy for Longevity

Creating code that remains viable across extended timescales requires a different approach to software design. Developers must think beyond immediate function-

ality to consider how their code will evolve and interact with future systems. This forward-thinking approach influences every aspect of development, from initial design to ongoing maintenance.

Schema planning becomes critical when working with persistable types. Since these cannot be changed after publication, developers must carefully consider not just current requirements but potential future needs. This might mean including optional fields that aren't immediately necessary or choosing open enums over closed ones when future expansion seems likely. The cost of getting these decisions wrong—being locked into inflexible schemas—encourages thorough upfront design.

Effect specification offers an interesting trade-off. While Verse allows and sometimes encourages over-specification of effects, marking a function as having effects it doesn't currently use, this provides flexibility for future implementation changes. A function marked as `<reads>` can later be optimized to `<computes>` without breaking compatibility, but the reverse isn't true. This asymmetry encourages conservative effect declarations that leave room for future modifications.

The choice between open and closed constructs represents another long-term decision. Open enums allow new values to be added after publication, providing extensibility at the cost of preventing exhaustive pattern matching. Closed enums offer the opposite trade-off. Understanding when flexibility or completeness is more valuable requires thinking about how the code will be used not just today, but years into the future.

Chapter 21

Concept Index

This index provides quick access to key concepts, language features, and important terms in the Verse documentation. Each entry links to the specific subsection where the concept is defined or explained in detail.

21.1 Type System

21.1.1 Primitive Types

- **any** - universal supertype: Primitives - Any, Type System
- **void** - empty type: Primitives - Void, Type System
- **logic** - boolean values: Primitives - Booleans, Type System
- **int** - integers: Overview, Primitives - Integers, Type System
- **float** - floating-point: Overview, Primitives - Floats, Type System
- **rational** - exact fractions: Primitives - Rationals, Operators, Type System
- **char** - UTF-8 character: Primitives - Characters and Strings, Type System
- **char32** - UTF-32 character: Primitives - Characters and Strings, Type System
- **string** - text values: Overview, Primitives - Characters and Strings, Type System

21.1.2 Composite Types

- **array** - ordered collections: Overview, Containers - Arrays, Type System
- **map** - key-value pairs: Containers - Maps, Type System
- **tuple** - fixed-size collections: Containers - Tuple, Expressions, Type System
- **option** - nullable values: Overview, Containers - Optionals, Type System
- **class** - reference types: Overview, Classes - Classes, Type System
- **struct** - value types: Structs - Structs, Type System

- **interface** - contracts: Classes - Interfaces, Type System
- **enum** - named values: Overview, Enums - Enums

21.1.3 Type Features

- **subtyping** - type relationships: Type System - Understanding Subtyping
- **comparable** - equality testing: Type System
- **parametric types** - generics: Classes - Parametric Classes, Type System
- **type{}** - type expressions: Expressions, Primitives - Type type, Type System
- **where clauses** - type constraints: Overview, Functions, Classes - Parameter Constraints, Type System

21.1.4 Type Variance

- **covariance** - type compatibility: Classes - Covariant, Type System - Covariance
- **contravariance** - reverse compatibility: Type System
- **invariance** - exact type match: Type System
- **bivariance** - both directions: Type System

21.1.5 Type Casting

- **casting** - type conversion: Type System - Class and Interface Casting
- **dynamic casts** - runtime type checking: Type System - Dynamic Type-Based Casting
- **fallible casts** - casts that may fail: Type System - Fallible Casts

21.1.6 Type Predicates and Metatypes

- **subtype** - runtime type values: Type System - subtype
- **concrete_subtype** - instantiable types: Type System - concrete_subtype
- **castable_subtype** - castable relationship: Type System - castable_subtype, Classes - Using castable_subtype
- **classifiable_subset** - type set tracking: Type System - classifiable_subset
- **classifiable_subset_var** - mutable type set: Type System - classifiable_subset
- **classifiable_subset_key** - type set keys: Type System - classifiable_subset

21.1.7 Type Query Functions

- **GetCastableFinalSuperClass** - get cast root from instance: Type System - GetCastableFinalSuperClass
- **GetCastableFinalSuperClassFromType** - get cast root from type: Type System - GetCastableFinalSuperClassFromType

- **MakeClassifiableSubset** - create immutable type set: Type System - classifiable_subset
- **MakeClassifiableSubsetVar** - create mutable type set: Type System - classifiable_subset

21.2 Effects

21.2.1 Effect Specifiers

- `<computes>` - pure computation: Overview, Effects, Mutability
- `<reads>` - observe state: Overview, Effects, Mutability
- `<writes>` - modify state: Effects, Mutability
- `<allocates>` - create unique values: Effects
- `<transacts>` - full heap access: Overview, Classes - Classes, Effects
- `<decides>` - can fail: Overview, Functions, Failure, Effects
- `<suspends>` - async execution: Overview, Effects, Concurrency
- `<converges>` - guaranteed termination: Effects
- `<diverges>` - may not terminate: Effects
- `<predicts>` - client execution: Effects
- `<dictates>` - server-only: Effects

21.3 Control Flow

21.3.1 Basic Control

- **if/then/else** - conditional execution: Overview, Expressions, Control Flow, Failure
- **case** - pattern matching: Overview, Enums - Using Enums, Control Flow
- **for** - iteration: Overview, Control Flow, Failure
- **loop** - infinite loops: Control Flow, Failure, Concurrency
- **block** - statement sequences: Control Flow, Failure, Classes - Blocks for Initialization, Concurrency
- **break** - exit loops: Control Flow
- **continue** - skip iteration: Control Flow
- **defer** - cleanup code: Control Flow
- **return** - exit functions: Functions

21.3.2 Failure System

- **failure** - control through failure: Overview, Failure, Effects
- **failable expressions** - can fail: Containers - Optionals, Functions, Failure
- **query operator (?)** - test values: Overview, Containers - Optionals, Operators, Failure

21.4 Concurrency

21.4.1 Structured Concurrency

21.4.2 Unstructured Concurrency

21.4.3 Timing Functions

21.5 Live Variables

21.5.1 Reactive Programming

21.5.2 Live Variable Features

21.6 Mutability

21.6.1 Mutation Control

- **var** - mutable variables: Mutability, Effects
- **set** - assignment: Overview, Mutability
- **immutability** - default behavior: Overview, Effects, Mutability
- **deep copying** - struct semantics: Mutability, Structs - Structs
- **reference semantics** - class behavior: Classes - Classes, Mutability
- **value semantics** - struct behavior: Structs - Structs, Mutability

21.7 Class & Type Specifiers

21.7.1 Structure Specifiers

- `<unique>` - identity equality: Overview, Classes - Unique, Mutability, Access Specifiers
- `<abstract>` - cannot instantiate: Classes - Abstract, Access Specifiers
- `<concrete>` - can instantiate: Classes - Concrete, Access Specifiers
- `<final>` - cannot inherit: Classes - Final, Persistable Types, Access Specifiers
- `<final_super>` - terminal inheritance: Classes - Final, Access Specifiers
- `<final_super_base>` - inheritance root: Classes - Final
- `<castable>` - runtime type checking: Classes - Castable, Access Specifiers, Code Evolution
- `<persistable>` - saveable data: Overview, Classes - Persistable, Structs - Persistable Structs, Persistable Types
- `<constructor>` - factory methods: Classes - Constructor Functions

21.7.2 Enum Specifiers

- `<open>` - extensible enums: Enums - Open vs Closed Enums, Access Specifiers, Code Evolution
- `<closed>` - fixed enums: Enums - Open vs Closed Enums, Access Specifiers, Code Evolution

21.8 Access Control

21.8.1 Visibility Specifiers

- `<public>` - universal access: Overview, Classes - Access Specifiers, Modules and Paths, Access Specifiers
- `<private>` - class/module only: Classes - Access Specifiers, Modules and Paths, Access Specifiers
- `<protected>` - subclass access: Classes - Access Specifiers, Modules and Paths, Access Specifiers
- `<internal>` - module access: Modules and Paths, Access Specifiers

- `<scoped>` - path-based access: Access Specifiers

21.8.2 Method Specifiers

- `<override>` - replace parent method: Classes - Method Overriding, Access Specifiers
- `<native>` - implemented in C++: Access Specifiers

21.9 Operators

21.9.1 Arithmetic

- +, -, *, /, % - math operations: Primitives - Mathematical Functions, Operators
- +=, -=, *=, /= - compound assignment: Operators

21.9.2 Comparison

- <, <=, >, >= - ordering: Operators
- =, <> - equality/inequality: Operators, Type System

21.9.3 Logical

- **and** - logical AND: Operators, Failure
- **or** - logical OR: Operators, Failure
- **not** - logical NOT: Operators, Failure

21.9.4 Access

- **.** - member access: Operators, Expressions
- [] - indexing: Containers - Arrays, Operators, Expressions
- () - function call: Operators, Expressions
- **{}** - object construction: Operators, Expressions

21.9.5 Special

- **:=** - initialization: Operators, Expressions
- **..** - range operator: Operators, Expressions
- **?** - query operator: Overview, Containers - Optionals, Operators, Failure

21.10 Functions

21.10.1 Function Features

- **parameters** - function inputs: Functions

- **named arguments** - explicit parameter names: Functions
- **return values** - function outputs: Functions
- **function types** - function signatures: Functions, Type System
- **overloading** - multiple definitions: Functions, Operators
- **lambdas** - anonymous functions (not yet supported, use nested functions): Functions, Expressions
- **nested functions** - local function definitions: Functions
- **higher-order functions** - functions as values: Overview, Functions

21.11 Modules & Organization

21.11.1 Module System

- **module** - code organization: Modules and Paths
- **using** - import statements: Overview, Modules and Paths
- **module paths** - hierarchical names: Modules and Paths
- **qualified names** - full paths: Modules and Paths
- **qualified access** - explicit paths: Modules and Paths
- **nested modules** - module hierarchy: Modules and Paths

21.12 Persistence

21.12.1 Save System

- **weak_map(player, t)** - player data: Containers - Weak Maps, Persistable Types
- **weak_map(session, t)** - session data: Containers - Weak Maps, Persistable Types
- **persistable types** - saveable data: Overview, Classes - Persistable, Structs - Persistable Structs, Persistable Types
- **module-scoped variables** - persistent storage: Modules and Paths, Persistable Types

21.13 Evolution & Compatibility

21.13.1 Version Management

- **backward compatibility** - preserving APIs: Overview, Effects, Code Evolution
- **versioning** - tracking changes: Code Evolution
- **deprecation** - phasing out features: Code Evolution
- **publication** - making code public: Modules and Paths, Access Specifiers, Code Evolution

- **breaking changes** - incompatible updates: Code Evolution
- **schema evolution** - data structure changes: Classes - Classes, Code Evolution

21.13.2 Annotations

- **@deprecated** - mark as deprecated: Code Evolution
- **@experimental** - mark as experimental: Code Evolution
- **@available** - version availability: Code Evolution

21.14 Built-in Functions

21.14.1 Math Functions

- **Abs()** - absolute value: Primitives - Mathematical Functions
- **Floor()** - round down: Primitives - Mathematical Functions
- **Ceil()** - round up: Primitives - Mathematical Functions
- **Round()** - round to nearest: Primitives - Mathematical Functions
- **Sqrt()** - square root: Primitives - Mathematical Functions
- **Min()** - minimum value: Primitives - Mathematical Functions
- **Max()** - maximum value: Primitives - Mathematical Functions

21.14.2 Utility Functions

- **Print()** - output text: Overview, Effects
- **ToString()** - convert to string: Primitives - ToString()
- **GetSession()** - current session: Modules and Paths

21.14.3 Array Methods

- **Find()** - find element index: Containers - Array Methods
- **Remove()** - remove by index: Containers - Array Methods
- **RemoveFirstElement()** - remove first occurrence: Containers - Array Methods
- **RemoveAllElements()** - remove all occurrences: Containers - Array Methods
- **ReplaceElement()** - replace by index: Containers - Array Methods
- **ReplaceFirstElement()** - replace first occurrence: Containers - Array Methods
- **ReplaceAllElements()** - replace all occurrences: Containers - Array Methods
- **ReplaceAll()** - pattern-based replacement: Containers - Array Methods

21.15 Syntax Elements

21.15.1 Literals

- **integer literals** - whole number values: Expressions
- **float literals** - decimal values: Expressions
- **string literals** - text values: Expressions
- **character literals** - single characters: Expressions
- **boolean literals** - true/false: Expressions

21.15.2 Special Values

- **false** - failure value, empty optional: Primitives - Booleans, Containers - Optionals, Failure
- **true** - success value: Primitives - Booleans
- **NaN** - not a number: Primitives - Floats
- **Inf** - infinity: Primitives - Floats

21.15.3 Language Constructs

- **comments** - code documentation: Overview
- **identifiers** - names: Expressions

21.16 Special Concepts

21.16.1 Language Features

- **archetype expression** - prototype patterns: Classes - Object Construction, Expressions
- **string interpolation** - embedded expressions: Primitives - Characters and Strings
- **pattern matching** - structural matching: Overview, Enums - Using Enums, Control Flow
- **inheritance** - class hierarchy: Classes - Inheritance, Type System, Access Specifiers
- **polymorphism** - multiple forms: Classes - Method Overriding, Type System
- **transactional semantics** - rollback behavior: Overview, Failure, Effects
- **option{}** constructor: Overview, Containers - Optionals
- **array{}** constructor: Overview, Containers - Arrays
- **map{}** constructor: Containers - Maps

Note: This index covers all major concepts in the Verse documentation. Each entry links directly to the subsection where the concept is defined or explained in detail. Use your browser's search function (Ctrl+F or Cmd+F) to quickly find specific terms.

Colophon

This book was typeset using LaTeX with XeLaTeX.

Page size: 7" × 10"
Includes 0.125" bleed on all sides.